МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Л. О. Левченко, Н. Г. Кучук,
Ю. А. Тарнавський, В. П. Колумбет

АРХІТЕКТУРА СИСТЕМНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Підручник

Затверджено Вченою радою КПІ ім. Ігоря Сікорського
як підручник для студентів,
які навчаються за освітньо-професійною програмою «Інженерія програмного забезпечення інтелектуальних кібер-фізичних систем і веб-технологій»
спеціальності 121 «Інженерія програмного забезпечення»

Київ
КПІ ім. Ігоря Сікорського
2022

УДК 004.45 (075.8)
А87

Рецензенти: *Нестеренко К. С.,* д-р. техн. наук, професор, декан факультету кібербезпеки, комп'ютерної та програмної інженерії Національного авіаційного університету

Коваленко А. А., д-р. техн. наук, професор, завідувач кафедри електронних обчислювальних машин Харківського національного університету радіоелектроніки

Відповідальний редактор *Ковальчук А. М.,* канд. техн. наук, доцент

Гриф надано Вченою радою КПІ ім. Ігоря Сікорського (протокол № 4 від 27.06.2022 р.)

АРХІТЕКТУРА СИСТЕМНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Електронне мережне навчальне видання

Левченко Лариса Олексіївна, д-р техн. наук, проф.
Кучук Ніна Георгіївна, д-р техн. наук, доц.
Юрій Адамович Тарнавський, канд. техн. наук, доц.
Колумбет Вадим Петрович, ст. викладач

А87 Архітектура системного програмного забезпечення [Електронний ресурс] : підручн. для студ. спец. 121 «Інженерія програмного забезпечення» / Л. О. Левченко, Н. Г. Кучук, Ю. А. Тарнавський, В. П. Колумбет; КПІ ім. Ігоря Сікорського. – Електронні текстові дані (1 файл: 6,6 Мбайт). Київ : КПІ ім. Ігоря Сікорського, 2022. – 497 с.

Викладено принципи побудови, архітектуру системного програмного забезпечення, основні функції, режими роботи, засоби операційних систем Linux і Windows. Особливу увагу приділено розробленню мікросервісів на основі технології *Docker*. Забезпечує студентів необхідними теоретичними та методичними знаннями для опанування відповідної теми робочої програми та виконання практичних завдань. Підручник корисний фахівцям у галузі системного програмування та системних адміністраторів

Призначений для студентів, які навчаються за спеціальністю 121 «Інженерія програмного забезпечення», також може бути використаний для студентів спеціальності 122 «Комп'ютерні науки» освітньо-професійної програми «Комп'ютерний моніторинг та геометричне моделювання процесів та систем» при вивченні операційних систем.

ЗМІСТ

ВСТУП……. 9

Розділ 1. Призначення, функції та архітектура операційної системи Linux…… 10

1.1. Концепції програмування в Linux. Завантаження й установка інструментарію для роботи з відкритим початковим кодом…… 10

1.1.1. Термінологія……..... 12

1.1.2. Історія виникнення Linux…….. 14

1.1.3. Стандарт POSIX……. 16

1.1.4. Дистрибутиви Linux (Debian, Ubuntu, Red Hat, SUSE, Solaris, HP-UX та AIX)…….. 17

1.1.5. Огляд системного програмування Linux…….. 20

1.2. Компоненти операційної системи. Ядро…… 23

1.2.1. Режим ядра і режим користувача…… 26

1.2.2. Конфігурація ядра Linux…… 29

1.2.3. Компіляція ядра…….. 30

1.3. Драйвери пристроїв……... 31

1.4. Файлові системи…… 31

1.5. Архіви……. 34

Контрольні запитання……. 38

Розділ 2. Керування Linux-пакетами…….. 39

2.1. Пакетний принцип організації операційних систем Linux…… 39

2.2. Менеджери пакетів…… 40

2.2.1. Формати пакетів програм Linux…… 41

2.2.2. Переваги компіляції початкових пакетів……. 43

2.2.3. Високо- і низькорівневі інструменти керування пакетами…… 43

2.2.4. Менеджери пакетів, засновані на Debian……. 45

2.2.4.1. Керування пакетами з командного рядка за допомогою APT…… 47

2.2.4.2. Менеджер пакетів Aptitude……. 58

2.2.4.3. Менеджер пакетів Synaptic……. 58

2.2.4.4. Диспетчер пакетів RPM (Red Hat Package Manager)…… 59
Контрольні запитання…… 67
Розділ 3. Файлова система Linux……. 68
3.1. Особливості файлової системи…… 68
3.1.1. Організація файлової системи…….. 89
3.1.2. Жорсткі диски: /dev/sd*……. 71
3.1.3. Монтування та демонтування файлової системи…… 72
3.1.4. Атрибути файлів, біти режиму……. 81
3.1.5. Інформація про файл. Системний виклик stat()…….. 83
3.1.6. Перенаправлення «введення – виведення»…….. 86
3.1.7. Стандартне «введення – виведення»…… 89
3.1.8. Файлові менеджери…… 89
3.2. Робота з файловою системою……... 94
3.2.1. Командні оболонки ОС Unix……. 94
3.2.2. Користувачі і групи. Права доступу……. 96
3.2.3. Загальні відомості про дозволи та права доступу до файлів у Linux……... 99
3.2.4. Команди для роботи з файловою системою…… 105
3.2.5. Система контролю версій та спільної розробки проектів з відкритим вихідним кодом – Git…….. 121
3.3. Фільтрація даних у файлі. Низькорівневе «введення – виведення». 129
3.3.1. Утиліта grep – потужний інструмент системного адміністратора..... 129
3.3.2. Регулярні вирази (regular expressions) або більш тонкі фільтри……... 132
3.3.3. Модель «введення – виведення» в системах UNIX…… 136
3.4. Сценарії в оболонці Bash…….. 147
3.4.1. Скрипт. Створення сценарію…… 147
3.4.2. Умовний оператор if then та else……... 153
3.4.3. Цикл ПОКИ……. 157
3.4.4. Оператор циклу for……. 160

3.4.5. Передача параметрів у командному рядку…….. 161
3.4.6. Функції…… 162
3.4.7. Передавання функції масиву в якості аргумента……..... 167
3.4.8. Рекурсивні функції……. 168
Контрольні запитання……. 169
Розділ 4. Керування процесами……... 171
4.1. Процеси та їх характеристики…….. 171
4.1.1. Характеристики процесів…….. 171
4.1.2. Системні виклики для роботи з процесами…… 179
4.1.3. Очікування завершених дочірніх процесів wait(int *status)….. 191
4.1.4. Стиль BSD: wait3() і wait4()…….. 199
4.1.5. Зомбі…… 203
4.2. Керування процесами в оболонці Bash……... 205
4.2.1. Типи процесів…… 205
4.2.2. Моніторинг процесів…….. 216
4.2.3. Пріоритети процесів……... 222
4.2.4. Керування сигналами……. 225
4.3. Потоковість…… 227
4.3.1. Багатопотоковість……... 230
4.3.2. Потокові моделі…….. 232
4.3.3. Реалізація потоковості в Linux…….. 240
4.3.4. М’ютекси Р-потоків……... 251
Контрольні запитання……. 255
Розділ 5. Керування пам’яттю……. 257
5.1. Адресний простір процесу……... 257
5.2. Інтерфейси для роботи з пам’яттю…….. 263
5.2.1. Виділення динамічної пам’яті……..……. 263
5.2.2. Виділення масивів…….. 265
5.2.3. Зміна розміру виділених областей…… 267
5.2.4. Звільнення динамічної пам’яті…….. 269
5.2.5. Вирівнювання даних…….. 271

5.2.6. Керування сегментом даних…….. 274
5.3. Анонімні відображення у пам’яті…… 275
5.3.1. Створення анонімних відображень у пам’яті…….. 277
5.3.2. Відображення /dev/zero…….. 279
5.4. Розширене виділення пам’яті……... 281
5.5. Отримання статистичної інформації……... 284
5.6. Виділення пам’яті на основі стека……... 285
5.7. Вибір механізму виділення пам’яті…… 290
5.8. Керування необробленими байтами пам’яті…….. 291
5.8.1. Установка байтів…… 291
5.8.2. Порівняння байтів…….. 292
5.8.3. Переміщення байтів……... 294
5.8.4. Пошук байтів…….. 295
5.8.5. Переклацування байтів…….. 296
5.9. Блокування пам’яті……... 297
5.9.1. Блокування частини адресного простору…… 298
5.9.2. Блокування всього адресного простору……... 299
5.9.3. Розблокування пам’яті……... 300
5.9.4. Ліміти блокування…….. 301
5.10. Стратегія пристосування / адаптивне (Opportunistic Allocation) виділення пам’яті…… 302
Контрольні запитання……. 305
Розділ 6. Застосування технології контейнерів…… 306
6.1. Розробка та впровадження програмного забезпечення за допомогою технології контейнерів…… 306
6.1.1. Контейнери та їх призначення…….. 306
6.1.2. Відмінності між віртуальними машинами та контейнерами…. 308
6.1.3. Класифікація контейнерів……. 311
6.1.4. Docker-контейнери…… 312
6.1.5. Архітектура Docker…… 315
6.1.6. Базові технології Docker…… 318

6.1.7. Інструментальні засоби для Docker…….. 320
6.1.8. Установка Docker……... 322
6.1.9. Використання команди Docker, опції, команди керування, підкоманди…… 324
6.2. Практична робота з Docker……... 327
6.2.1. Хостинг для Docker…… 327
6.2.2. Образи, контекст створення образу, реєстр образів……... 328
6.2.3. Команди для роботи з реєстром…… 330
6.2.4. Команди керування контейнерами протягом їх життєвого циклу……... 334
6.2.5. Робота з образами……... 338
6.2.6. Приклади керування контейнерами Docker…… 344
6.2.7. Dockerfile та синтаксис для його створення…… 346
6.2.8. Docker Compose для керування додатками з декількома контейнерами…… 350
6.3. Життєвий цикл програмного забезпечення при використанні Docker…… 357
6.4. Мікросервіси та контейнери…… 372
6.4.1. Характеристика мікросервісів…….. 372
6.4.2. Оркестрація, кластеризація і керування…….. 378
6.4.3. Інструментальні засоби оркестрації і кластеризації……... 380
Контрольні запитання……. 395
РОЗДІЛ 7. Операційні системи сімейства Windows…… 397
7.1. Архітектура Windows……..... 397
7.1.1. Історія Windows…….. 397
7.1.2. Модель операційної системи…… 400
7.1.3. Інтерфейс прикладного програмування Win32……... 410
7.1.4. Реєстр Windows…….. 413
7.1.5. Реалізація диспетчера об’єктів…….. 415
7.1.6. Підсистеми середовища оточення, DLL-бібліотеки підсистем, служби користувацького режиму……... 419

7.2. Процеси і потоки в Windows…… 421
7.2.1. Базові поняття: процеси і потоки в сучасних операційних систем. Керування процесами і потоками……. 421
7.2.2. Складові елементи процесів і потоків…….. 424
7.2.3. Фундаментальні концепції – процеси і потоки у Windows…….. 431
7.2.4. Виклики API для керування завданнями, процесами, потоками і волокнами…….. 443
7.3. Керування пам’яттю в Windows…….. 448
7.3.1. Віртуалізація пам’яті……. 448
7.3.2. Системні виклики керування пам’яттю……... 459
7.3.3. Кешування у Windows……... 462
7.4. «Введення – виведення» у Windows…… 465
7.4.1. Фундаментальні концепції – «введення – виведення» у Windows…….. 465
7.4.2. Інтерфейси прикладного програмування «введення – виведення»……... 470
7.4.3. Драйвери пристроїв…… 474
7.4.4. Файлова система Windows NTFS…….. 477
7.4.5. Системні виклики для роботи з файлами……. 486
7.4.6. Керування електроживленням в Windows……... 489
7.4.7. Безпека у Windows……. 490
7.4.8. Виклики інтерфейсу прикладного програмування безпеки….. 492
Контрольні запитання……. 493
СПИСОК ЛІТЕРАТУРИ…….. 496

ВСТУП

Навчальна дисципліна «*Архітектура системного програмного забезпечення»* входить до циклу професійної та практичної підготовки навчального плану бакалаврів спеціальності 121 «Інженерія програмного забезпечення» освітньої програми «Інженерія програмного забезпечення кіберфізичних систем і веб-технологій». Дисципліна складається з одного кредитного модуля та призначена для викладання студентам другого курсу у четвертому семестрі.

Оскільки на сьогоднішній день затребувані фахівці, які впроваджують новітні технології, в основі яких лежить саме операційна система (ОС) Linux, у підручнику детально описано архітектуру системного програмного забезпечення (СПЗ) Linux. Також розглядається й архітектура ОС Window 10 – ця система є найбільш вживаною серед користувачів. Отримані знання та набутий досвід дозволяють опановувати засоби та методи роботи в середовищі ОС Linux з використанням технології віртуалізації, виконувати роботи з адміністрування ОС, застосовувати технології контейнерів для розгортання, поширення та функціонування програмного забезпечення (ПЗ), створювати мікросервіси. Набуті знання й навички сприятимуть кар'єрному зростанню майбутніх фахівців, їх самореалізації, більш кваліфікованому вирішенню практичних завдань.

У підручнику розглянуто такі поняття: користувачі з точки зору системи, термінал, робота у командному рядку, особливості роботи з файловою системою, права доступу в Linux, можливості командної оболонки, текстові редактори, процеси, потоки. Особливу увагу приділено найпопулярнішій платформі керування контейнерами Docker, яка нещодавно з'явилася на ринку ІТ-технологій. Саме контейнери Docker спрощують як упаковку програмних додатків, так і їх перенесення, запуск у різних локальних середовищах й у хмарі.

Метою дисципліни є опанування студентами теоретичних знань архітектури СПЗ, побудови, функціонування, використання засобів ОС, технології контейнерів для реалізації упаковки, розгортання та функціонування ПЗ.

Предмет дисципліни – вивчення принципів побудови, архітектури, основних функцій, режимів роботи, засобів ОС, розроблення мікросервісів на основі технології *Docker*.

Матеріал є корисним для фахівців у галузі системного програмування та системних адміністраторів.

Розділ 1

ПРИЗНАЧЕННЯ, ФУНКЦІЇ ТА АРХІТЕКТУРА ОПЕРАЦІЙНОЇ СИСТЕМИ LINUX

1.1. Концепції програмування в Linux. Завантаження та установка інструментарію для роботи з відкритим початковим кодом

Розглянемо, для чого вивчають дисципліну «Архітектура системного програмного забезпечення» та де використовують набуті знання.

Не випадково у цій дисципліні детально вивчається архітектура СПЗ ОС Linux. Нині відбувається четверта індустріальна революція, яка змінює світ і сприяє появі нових технічних і технологічних можливостей та, відповідно, професій майбутнього. Для цього є декілька чинників:

– з'явився високошвидкісний інтернет, який майже всюди є доступним;

– набуває популярності інтернет із продажу різних речей;

– широко застосовується аналітика Big Date;

– у різних сферах активно впроваджується штучний інтелект;

– широко використовуються хмарні технології для вирішення завдань бізнесу.

Хмарні технології – це спосіб організації фізичних та програмних засобів, а також набір інструментів, за допомогою яких користувач отримує обчислювальні потужності, щоб виконувати поставлені перед ним завдання. Хмарні послуги – це:

– пошта: *gmail, hotmail*;

– віддалена робота з документами: *Google*-документи, *Office Web Apps*;

– зберігання даних: *Google Drive, OneDrive, Dropbox*;

– редагування зображень у режимі реального часу: *Figma*;

– сервіси для створення заміток, спільної роботи над завданнями: *Trello, Jira, Evernote;*

– онлайн-магазини додатків: *Google Play, App Store і Microsoft Store*;

– хмарний хостинг – розміщення свого сайту у «хмарі».

Організація Об'єднаних Націй прирівняла інтернет до води, наголошуючи на його важливості та можливостях задоволення потреб людства. На сьогоднішній день усі послуги, які надаються, підключаються до інтернету, тобто все продається через інтернет. Це генерує великі дані *Big Data*, які може обробити винятково штучний інтелект, тому пильна увага науковців спрямована на машинне навчання зі штучного інтелекту. Весь програмний інструментарій міститься у хмарах і все стає програмно доступним завдяки *devnet*-інженерам, які за допомогою програмних комплексів, сервісів створюють цілі системи. Фактично кількома кліками налаштовуються комутатор, маршрутизатор, у хмарі розгортається сервер, на який встановлюється *engine* (движок) і завантажується конфігурація, що буде підключатися до бази даних SQL.

Linux використовують у багатьох сферах людської діяльності. Мобільна ОС Android базується на Linux. Під керуванням Linux працює більшість суперкомп'ютерів (NASA) у світі. На софті, написаному під Linux, працює високошвидкісна залізниця Японії. На Linux працюють автомобілебудівна корпорація *Toyota*, комунікаційні системи *Toyota*, Нью-Йоркська фондова біржа. *Google, Amazon* та *Facebook* використовують Linux для своїх хмарних і веб-сервісів. Системи керування повітряним рухом використовують Linux, який забезпечує безпеку польотів. *CERN* – найбільша у світі лабораторія фізики – довірила Linux контролювати величезний прискорювач частинок. І таких прикладів дуже багато.

Сучасний світ недарма обирає Linux. У США у третьому кварталі 2020 року звільнили 115 тис., а в Індії – 60 тис. програмістів, тому що

впроваджується штучний інтелект. Щоб адаптуватися до світу, який швидко змінюється, потрібно швидко вчитися, тому й затребувані фахівці, які впроваджують новітні технології, в основі яких лежить саме Linux. Нині все функціонує з використанням Linux, а також *Docker, Kubernetes,* віртуалізації, кластеризації, оркестрації, – цього в Microsoft немає взагалі. Це означає, що лише натиснувши на кнопку можна виконати значну кількість робіт і налаштувань.

Отримані знання дозволяють виконувати роботи з адміністрування ОС, застосування технології контейнерів для розгортання, поширення та функціонування ПЗ, створення мікросервісів.

1.1.1. Термінологія

Операційна система (ОС, в англомовному варіанті – ***operating system***) – базове системне ПЗ, що керує роботою комп'ютера і є посередником (інтерфейсом) між апаратурою (**hardware**), прикладним ПЗ (**application software**) і користувачем комп'ютера (**user**).

Спосіб побудови системи зі складових частин та їх взаємозв'язок визначають *архітектуру ОС*.

Набір компонентів ОС, які відповідають за певні функції, і порядок їх взаємодії між собою та із зовнішнім середовищем називають **архітектурою ОС**.

Базові компоненти ОС, які відповідають за найважливіші функції, зазвичай перебувають у пам'яті постійно й виконуються у привілейованому режимі, їх називають *ядром ОС* (*operating system kernel*).

До найважливіших функцій ОС, які виконує ядро, належать обробка переривань, керування пам'яттю, керування «введенням – виведенням».

Операційна система має можливість втручатися в роботу прикладних програм, а прикладні програми не можуть втручатися в роботу ОС. Для реалізації таких привілеїв розроблено апаратну підтримку: процесор

підтримує два режими роботи: *привілейований* (*захищений режим, режим ядра, kernel mode*) і *режим користувача* (*user mode*).

У режимі користувача недопустимі команди, які є критичними для роботи системи (перемикання задач, звернення до пам'яті за заданими межами, доступ до пристроїв «введення – виведення» тощо).

Після завантаження системи ядро перемикає процесор у привілейований режим й отримує повний контроль над комп'ютером.

Системне програмне забезпечення (СПЗ, англ. *System Software*) – сукупність службових програм і програмних комплексів для забезпечення роботи комп'ютера і мереж ЕОМ. Системне програмне забезпечення керує ресурсами комп'ютерної системи і дозволяє користувачам програмувати високорівневими мовами, а не низькорівневою машинною мовою комп'ютера. Приклад – архівація файлів, очищення реєстру, відновлення даних та багато-багато іншого.

На сьогодні є три найпоширеніші операційні системи – *Windows, Unix і MacOS*. Всі вони написані мовою С (не плутати із С++). Будь-яка програма для цих систем матиме максимальну продуктивність, якщо буде написана мовою С.

До службових програм належать архівація файлів, очищення реєстру, відновлення даних та інше. Для системного програмування достатньо знати мову С (системні виклики, бібліотеки С та компілятор С). Однак нині серед розробників системного програмування набуває поширення мова Python, оскільки вона проста, лаконічна, підтримує спеціальні пакети, її можуть використовувати системні адміністратори як для автоматизації завдань системного адміністрування, обслуговування сховищ даних, хмарних сервісів, так і для проведення різних наукових досліджень. Ця мова має потужні бібліотеки, тому розробка на ній відбувається у рази швидше, а обсяг коду набагато менший, ніж в інших мовах. Вона не має компілятора, а інтерпретується, тому збільшується час виконання програм.

За замовчуванням вона встановлена на усіх серверах з ОС *Linux*. Але мову Python не застосовують для завдань, які потребують великого обсягу пам'яті, для цього виконують вставки на С або С++.

У системному програмуванні для Linux можна виділити три ключові складові: системні виклики, бібліотека С і компілятор С.

1.1.2. Історія виникнення Linux

Linux (Лінукс) – сімейство вільно розповсюджуваних ОС, заснованих на базі ядра ОС UNIX. Термін «сімейство ОС» означає наявність різних версій (дистрибутивів) цієї ОС [1]. Перший реліз (випуск) ядра відбувся 5 жовтня 1991 року, коли студент університету Хельсінкі Лінус Торвальдс почав працювати над написанням власної ОС, яка не мала б обмежень порівняно з ОС Minix, написаної Ендрю Таненбаумом. Операційна система Minix (мінікс, від слова *mini-UNIX*) мала ряд обмежень щодо використання, наприклад, тільки в освітніх цілях. Це означало, що будь-яке функціонування цієї ОС в комерційних цілях було протизаконним. У зв'язку з цим Лінус Торвальдс почав розробку ядра Linux на базі ОС Minix, яка, у свою чергу, є UNIX-подібною операційною системою. Назва Linux походить від поєднання імені його засновника Лінуса та назви ОС UNIX. Власник FTP-сервера (Арі Лемке), на якому Лінус виклав у загальний доступ початковий код ядра, перейменував його в Linux. Згодом така назва закріпилася за цією ОС.

Важливу роль у розвитку Linux відіграв проект розробки ОС з назвою GNU, заснований Річардом Столлманом.

У 1983 році Річард Столлман оголосив про початок роботи над проектом вільної ОС UNIX, яку він назвав **GNU** Unix. Абревіатура GNU розшифровується як **G**NU is **N**ot **U**NIX (GNU – це не UNIX). Передбачалося, що GNU буде містити ядро і всі утиліти, щоб писати й запускати програми на С: редактор, оболонку, компілятор С, лінковщик,

асемблер речей тощо й буде здатна запускати програми Unix, але не буде ідентична Unix.

Згодом Річард Столлман написав Маніфест GNU, який став основою ліцензії **GPL**. Відтоді абревіатура GNU стала розшифровуватись *General Public License* (Загальна публічна ліцензія). Із ПЗ під такою ліцензією дозволяється робити все що завгодно: копіювати, допрацьовувати, продавати і под. Правда, за умови, що ці програми також будуть поширюватися під ліцензією GPL. Інакше кажучи, ПЗ під ліцензією GPL робить його вільним і гарантує, що воно залишиться вільним, будь-хто може ознайомитися з початковим кодом програми, який перебуває під цією ліцензією. До 1990 року система GNU була майже закінчена, для неї було написано безліч утиліт, однак, як вважає Столлман, не вистачало тільки ядра. У цей час за неймовірним збігом обставин раптово з'являється Лінус Торвальдс і представляє світові створене ним ядро. Це було саме те, чого так не вистачало Річарду Столлману. Те, що Лінус Торвальдс зі своєю розробкою опинився в потрібний час у потрібному місці, і визначило майбутнє ОС Linux.

Після публікації початкових текстів ОС Linux стрімко почала набирати популярність, і ядро, написане під платформу x86, було перенесене й під платформу x64.

Linux – змінена й удосконалена реалізація ядра ОС UNIX. Вона відповідає стандарту POSIX, працює на різних апаратних платформах і сумісна з більшістю відомих додатків UNIX. Відмінність від безлічі інших (але не всіх) варіантів UNIX полягає в тому, що *Linux – безкоштовна ОС з відкритим початковим кодом,* яка розробляється колективними зусиллями тисяч ентузіастів та організацій. Між тим ОС Linux містить ряд технічних удосконалень, яких немає в UNIX, і тому вона є чимсь більшим, ніж простий клон UNIX. Водночас традиційні виробники UNIX

не припиняють удосконалювати свої продукти, що означає наявність областей, в яких комерційні UNIX-системи перевершують Linux-системи.

1.1.3. Стандарт POSIX

POSIX (*Portable Operating System Interface for Unix*) – набір стандартів, що описують інтерфейси між ОСі прикладною програмою (системні виклики, які реалізовані як набір функцій, що забезпечує інтерфейс між прикладною програмою та ядром ОС (*Application Programming Interface – API*), бібліотека мови С, набір додатків та їх інтерфейсів. Набір був створений для забезпечення сумісності різних UNIX-подібних ОС та переносимості прикладних програм на рівні початкового коду [1].

Відкриті стандарти – це стандарти, які публікуються у відкритих джерелах і найчастіше мають одну або кілька (часто обов'язковою є наявність щонайменше двох) еталонних реалізацій (reference implementation). Також зазвичай такі стандарти розробляють у межах чітко визначеного процесу.

Інтерфейс – сукупність правил (описів, угод, протоколів), що забезпечують взаємодію пристроїв і програм в обчислювальній системі або сполучення між системами. Це зовнішнє подання, абстракція якогось інформаційного об'єкта. Інтерфейс розділяє методи зовнішньої взаємодії і внутрішньої роботи.

Протокол – набір угод / правил інтерфейсу логічного рівня, які визначають обмін даними між різними програмами. Ці угоди задають однаковий спосіб передачі повідомлень й обробки помилок у разі взаємодії ПЗ та апаратного забезпечення.

Набір стандартів POSIX:

– спрощує перенесення прикладних програм на інші апаратні платформи;

– визначає необхідний мінімум інтерфейсів прикладної програми;

– допомагає заздалегідь визначити інтерфейси ще на етапі проектування;

– дозволяє враховувати спадковість раніше створених прикладних програм;

– вводить обмеження на використання бінарного (об’єктного) коду у простих системах;

– розвиває стандарти у сфері комунікаційних мереж, розподіленої обробки даних і захисту інформації.

Стандарт POSIX складається з чотирьох основних розділів:

1. Основні визначення (англ. *Base definitions*).

2. Оболонка й утиліти (англ. *Shell and utilities*).

3. Системні інтерфейси (англ. *System interfaces*).

4. Обґрунтування (англ. *Rationale*).

Образ Linux можна розвернути на USB-накопичувачі, запустити його з ПК, якщо, наприклад, пошкоджений внутрішній жорсткий диск. Це багатокористувацька ОС, в якій команди користувачів можуть одночасно авторизуватися в системі як для локальної, так і для віддаленої роботи, при цьому залишаючись впевненими у конфіденційності та стабільності системи.

1.1.4. Дистрибутиви Linux (Debian, Ubuntu, Red Hat, SUSE, Solaris, HP-UX та AIX)

Дистрибутив (англ. *distribute* – поширювати) – це форма поширення певного ПЗ, а саме, спеціалізований набір програм, об’єднаних єдиними системами установки, керування та поновлення пакетів, налаштування системи і підтримки користувачів, який постачається разом з ядром Linux та поширюється з інструментами для встановлення робочої версії Linux. Кожний дистрибутив призначений для певного кола користувачів, від новачка до професіоналів. Усі дистрибутиви засновані на єдиному сімействі ядер, проте набір службових програм, які доповнюють ядро,

може істотно варіюватися. Дистрибутиви розрізняються за призначенням, наявністю служб підтримки і ступенем популярності.

Основними характеристиками дистрибутиву є такі.

• *Політика дистрибутиву*. Яким чином у дистрибутив включається ПЗ, які вимоги до розміщення файлів пакетів на файловій системі, яка періодичність оновлення дистрибутиву та ін.

• *Програма-завантажник*. Відповідає за ініціалізацію ядра ОС, початкових налаштувань апаратного забезпечення.

• *Ядро, яке використовується*. Ключова компонента дистрибутиву, від якої залежить працездатність системи. Версії ядра дистрибутиву залежать від типу дистрибутиву (стабільний, тестовий і под.). У стабільних дистрибутивів версія ядра Linux не містить останніх оновлень драйверів та нових можливостей.

• *Набори пакетів*. Це дистрибутиви, орієнтовані на вирішення конкретних завдань.

• *Ліцензування.* Визначає політику використання ПЗ дистрибутиву.

• *Розробка.* Технічні, адміністративні, фінансові та інші рішення, які закладені у дистрибутив, підтримка користувачів. *Існує вел*ичезна кількість дистрибутивів ОС *Linux*. На сайті *DistroWatch* можна подивитися поточний статус за всіма дистрибутивами *Linux*. У наш час є три основні типи дистрибутивів ОС Linux, що розрізняються з погляду системи керування пакетами програм:

• ***Дистрибутиви, засновані на Debian GNU/Linux, або які використовують формат пакетів DEB*.** У дистрибутивах такого типу наявна найбільша кількість пакетів, вони підтримують більшість відомих апаратних архітектур ***Debian GNU/Linux* (*Debian, Ubuntu, Linux Mint, Knoppix, SteamOS, Android*).**

Debian – некомерційний дистрибутив ОС Linux, призначений для роботи як серверної, так і настільної ОС. Дистрибутив популярний

у професіоналів у сфері безпеки інформаційних систем завдяки швидкому реагуванню на знайдені помилки і загрози. До ПЗ, що поставляється з дистрибутивом Debian, висуваються жорсткі вимоги, що, у свою чергу, гарантує стабільність версій дистрибутиву. Дистрибутив Debian став відгалуженням лінійки Debian-подібних дистрибутивів: Ubuntu, Knoppix, Astra-Linux та. ін. Debian рекомендований насамперед фахівцям із безпеки, а також досвідченим користувачам.

Ubuntu – ОС, заснована на проекті Debian. Головним спонсором і розробником є компанія Canonical. Ubuntu – дуже популярна ОС не тільки у пересічних користувачів, але й у фахівців, які встановлюють її як серверну операційну систему. *Редакції Ubuntu: Ubuntu (основний), Ubuntu Gnome, Kubuntu, Xubuntu, Ubuntu MATE, Ubuntu Server, Mythbuntu, Ubuntu Studio.*

• ***Дистрибутиви, засновані на Red Hat Linux, або які використовують формат пакетів RPM.*** Класичним дистрибутивом ОС Linux є дистрибутив ***Red Hat Enterprise Linux***, який випускає компанія Red Hat (червоний капелюх). Red Hat Enterprise Linux поширюється за річною передплатою й актуальний тільки для комерційних організацій. Головною особливістю цього дистрибутиву є наявність платної підтримки протягом 10 років. Наприклад, щоб отримати бінарні (вже скомпільовані) пакети оновлень для ОС, потрібно заплатити певні кошти.

• ***Дистрибутиви, засновані на Slackware Linux.*** Дистрибутиви Slackware Slackware ***засновані на принципі KISS (Keep It Simple, Stupid, або Keep It Short and Simple – процес і принцип проектування***, за якого простота системи декларується як одна з основних цілей або цінностей). Крім того, менеджер пакетів не відстежує залежності через їх дуже незначну кількість.

Є й інші типи дистрибутивів:

• Американська кампанія Novell розробила ***дистрибутиви SUSE Linux Enterprise***: *openSUSE, Novell Linux Desktop, SLED – SUSE Linux Enterprise Desktop – операційну систему для робочих станцій, SLES – Suse Linux Enterprise Server – ОС для серверів.*

• ***Дистрибутив Solaris*** – ОС із 2010 року належить разом з активами Sun корпорації Oracle. Попри те, що Solaris – ОС із закритим вихідним кодом, більша його частина відкрита й опублікована у проекті *OpenSolaris (або Solaris 10)*.

• ***Дистрибутив HP-UX*** – версія ОС UNIX фірми Hewlett-Packard (HP). Це багатокористувацька, багатозадачна, багатопотокова ОС. Ядро забезпечує поділ часу між виконуваними програмами, а циклічна передача керування гарантує, що кожному процесу, який працює в системі, буде надано процесорний час. Система HP-UX, як і більшість інших різновидів UNIX, захищена від вірусів.

• ***Дистрибутив AIX*** – серія власних ОС Unix, розроблених і проданих IBM для кількох своїх комп'ютерних платформ. AIX тепер підтримує широкий спектр апаратних платформ, включаючи системи IBM RS/6000, більш пізні версії POWER та PowerPC, системні IBM System, системні/370 мейнфрейми, PS/2 Персональні комп'ютери, мережевий сервер Apple.

• Дистрибутиви для ***хмарних обчислень*** – Amazon Linux (AWS AMI) та Ubuntu Server (AWS AMI).

1.1.5. Огляд системного програмування Linux

Концепції програмування передбачають написання системного та прикладного ПЗ. *Системні програми є низькорівневими, взаємодіють безпосередньо з ядром та основними системними бібліотеками.* Командна оболонка і текстовий редактор, компілятор і налагоджувач, основні утиліти та системні демони – все це *системне ПЗ.* До цієї категорії належать також мережевий сервер, веб-сервер і база даних.

Прикладне ПЗ – це прикладні програми із графічними інтерфейсами користувача, які спираються на високорівневі бібліотеки і взаємодіють з низькорівневими системними програмами лише епізодично.

У системному програмуванні для Linux можна виділити три основні ключові складові: системні виклики, бібліотека С і компілятор С.

Системні виклики (syscall) – це виклики функцій, що здійснюються з користувацьких програм, а саме, спрямовані з додатків (наприклад, текстового редактора або програми користувача) до ядра. *Сенс системного виклику полягає у тому, щоб запросити в ОС певну службу або ресурс.* Системні виклики включають як всім знайомі операції, наприклад *read()* та *write(),* так і досить екзотичні, зокрема *get_thread_area()* та *set_tid_address().* У Linux реалізується набагато менше системних викликів, ніж у ядрах більшості інших ОС. Наприклад, у системах з архітектурою x86-64 таких викликів близько 300. Якщо порівняти з Microsoft Windows, то в цій ОС задіяні тисячі подібних викликів.

Неможливо прямо зв’язати додатки із простору користувача із простором ядра. Це пов’язано із забезпеченням безпеки і надійності, тому що додаткам користувачів не дозволяється безпосередньо виконувати код ядра або маніпулювати даними ядра.

Додаток повідомляє ядру, який системний виклик потрібно виконати і з якими параметрами. Це здійснюється за допомогою апаратних регістрів. *Системні виклики позначено за номерами, починаючи з 0.* В архітектурі i386, щоб запросити системний виклик 5 (зазвичай це виклик open()), користувальницький додаток записує 5 у регістр *eax,* після чого видає інструкцію int. Передавання параметрів обробляється схожим чином. Так, в архітектурі i386 регістр застосовний для всіх можливих параметрів – наприклад, регістри *ebx, ecx, edx, esi та edi*, у такому ж порядку містяться перші п’ять параметрів.

Бібліотека C (libc) – це серцевина всіх додатків UNIX, вона пов'язана з більш високорівневими бібліотеками і надає основні служби і сприяє активації системних викликів. У сучасних системах Linux бібліотека C надається у формі GNUlibc, скорочено glibc (вимовляється як «джілібсі»). Бібліотека GNU C надає набагато більше можливостей. Окрім реалізації стандартної бібліотеки C, glibc використовує бібліотеки для роботи системних викликів, роботи з потоками та основні функції додатків.

Компілятор C

У Linux стандартний компілятор мови C надається у формі колекції компіляторів GNU (GNU Compiler Collection, скорочено ***gcc***). Спочатку gcc являла собою версію cc (компілятора C) для GNU. Відповідно gcc розшифровувалась як GNU C Compiler. Однак згодом додалася підтримка інших мов, тому сьогодні gcc слугує загальною назвою всього сімейства компіляторів GNU. При цьому gcc – це ще й двійковий файл, який використовується для активації компілятора C. Компілятор, який використовується в UNIX-подібних системах, зокрема в Linux, має величезне значення для системного програмування, бо дає змогу впроваджувати стандарт мови C, а також системний двійковий інтерфейс додатків (API та ABI).

На системному рівні є дві окремі множини визначень та описів, які впливають на таку переносимість. Перша множина – це інтерфейс програмування додатків (*Application Programming Interface, API*), друга – двійковий інтерфейс додатка (*Application Binary Interface, ABI*). Обидві ці концепції визначають та описують інтерфейси між різними компонентами ПЗ.

API визначає інтерфейси у початковому коді, на них відбувається обмін інформацією між двома компонентами ПЗ на рівні початкового коду – один високорівневий компонент може викликатися з низькорівневого. Прикладом API служать інтерфейси, визначені відповідно до стандарту C і реалізовані стандартною бібліотекою C. Цей API-інтерфейс

визначає сімейство найпростіших і критично важливих функцій, таких як процедури для керування пам'яттю і маніпуляцій з рядками.

ABI призначений для визначення двійкового інтерфейсу між двома і більше програмними компонентами у конкретній архітектурі. ABI визначає, як додаток взаємодіє із самим собою, з ядром і бібліотеками. Якщо API забезпечує сумісність на рівні початкового коду, то *ABI відповідає за сумісність на двійковому рівні.* Це означає, що фрагмент об'єктного коду буде функціонувати у будь-якій системі з таким самим ABI без необхідності перекомпіляції. ABI допомагають вирішувати проблеми, пов'язані з угодами на рівні викликів, порядком проходження байтів, використанням регістрів, активацією системних викликів, зв'язуванням поведінки бібліотек і форматом двійкових об'єктів. Наприклад, угоди на рівні викликів визначають, як будуть викликатися функції, як аргументи передаються функціям, які регістри зберігаються, а які спотворюються, як сторона, яка викликає, отримує значення, що повертається.

Linux відповідає двом найбільш важливим і превалюючим стандартам – POSIX та Single UNIX Specification (SUS, єдина специфікація UNIX).

1.2. Компоненти операційної системи. Ядро

Ядро – це основа будь-якої ОС, яка організовує доступ до апаратних засобів через абстрактний високорівневий програмний інтерфейс.

Інтерфейс користувача – це зовнішня частина ОС, а ядро – внутрішня. У своїй основі *ядро – це ПЗ, яке надає базові функції для всіх інших частин ОС, управляє апаратним забезпеченням і розподіляє системні ресурси.* Ядро часто називають **супервізором** (supervisor), основною частиною (core) або нутрощами (internals) ОС.

Наведемо *типові компоненти ядра* (задачі, що виконує ядро) [1, 8, 10].

• *Диспетчеризація процесів або планувальник (система взаємодії між процесами)*, що розподіляє процесорний час між багатьма процесами. У комп'ютера є один або кілька центральних процесорів (CPU), що виконують інструкції програм. Як і інші UNIX-системи, Linux є багатозадачною ОСз витісненням. Багатозадачність означає, що кілька процесів (наприклад, запущені програми) можуть одночасно перебувати у пам'яті й кожний може використовувати центральний процесор (процесори). Витіснення означає, що правила, що визначають, які саме процеси отримують у розпорядження центральний процесор і на який термін, встановлює диспетчер процесів (а не самі процеси), який є в ядрі.

• *Система* керува*ння пам'яттю.* Оперативна пам'ять залишилася ще обмеженим ресурсом, який ядро розподіляє між процесами певним чином і керує адресним простором процесів. Як і в більшості сучасних ОС, в Linux використовується *керування віртуальною пам'яттю* (підкачка, сторінковий обмін, відображення віртуальних адрес у фізичній пам'яті) – технологія, що дає дві основні переваги:

1) процеси ізольовані один від одного та від ядра, тому один процес не може читати або змінювати вміст пам'яті іншого процесу чи ядра;

2) у пам'яті потрібно зберігати тільки частину процесу, зменшуючи таким чином обсяг пам'яті, необхідний кожному процесу, й дозволяючи одночасно збільшити кількість процесів в оперативній пам'яті. Це дозволяє підвищити ефективність використання центрального процесора, тому що в результаті збільшується ймовірність того, що в будь-який момент часу є принаймні один процес, який може бути виконаний центральним процесором (процесорами).

• *Надання файлової системи*. Ядро надає файлову систему на диску (файли, каталоги, простір імен), дозволяючи створювати, зчитувати, оновлювати, видаляти файли, виконувати їх вибірку і проводити з ними інші дії.

• *Створення та завершення процесів.* Ядро може завантажити нову програму в пам’ять, надати їй ресурси (наприклад, центральний процесор, пам’ять і доступ до файлів), необхідні для роботи. Такий екземпляр запущеної програми називається процесом. Як тільки виконання процесу завершиться, ядро забезпечує вивільнення використовуваних їм ресурсів для подальшого застосування іншими програмами.

• *Доступ до пристроїв*. Пристрої (миші, монітори, клавіатури, дискові і стрічкові накопичувачі, USB і т.п), підключені до комп’ютера, дозволяють обмінюватися інформацією між комп’ютером і зовнішнім світом – здійснювати загальне введення/виведення даних. Ядро надає програми з інтерфейсом для спрощення доступу до пристроїв. Цей доступ відбувається в рамках певного стандарту. Одночасно з цим ядро розподіляє доступ до кожного пристрою з боку декількох процесів.

• *Обробники переривань*, які обслуговують запити на переривання, що надходять від різних пристроїв.

• *Мережева підсистема (або робота в мережі)*. Ядро від імені користувача процесів відправляє і приймає мережеві повідомлення (пакети). Це завдання включає в себе маршрутизацію мережевих пакетів в напрямку цільової ОС.

• *Надання інтерфейсу прикладного програмування* (API – Application Programing Interface) системних викликів. Процеси можуть запитувати у ядра виконання різних завдань з використанням точок входу в ядро, відомих як *системні виклики*.

Ядро містить як драйвери пристроїв, які організовують взаємодію з окремими елементами апаратного рівня, так і іншу частину ядра, яка не залежить від зовнішніх пристроїв. Взаємозв’язок ядра і драйверів пристроїв аналогічний зв’язку між ядром і процесами призначеного для користувача рівня.

Окрім ядра ***до складу компонентів ОС входять*** такі програмні компоненти ОС, з якими взаємодіють користувачі:

• ***Оболонки командного рядка.*** Нині існує безліч оболонок, їх доступність і популярність варіюються від однієї ОС до іншої. У *Linux* часто використовується оболонка під назвою *Bourne Again Shell (bash або Bash)* [bon again shel bash]. Інші оболонки – *C shell, Korn shell (ksh), Bourne shell (sh).*

• ***Графічні інтерфейси користувачів.*** Графічний інтерфейс користувача (*Graphical User Interface – GUI*) в ОС Windows і *MacOS* передбачений як власний GUI. В Linux підтримується графічний інтерфейс під назвою **W Window System, або X**. Інтерфейс X є базовим GUI, тому Linux також використовує такі середовища робочого стола, як GNOME або K Desktop Environment (KOE) для забезпечення більш повноцінної роботи. Саме відмінність між середовищем робочого столу Linux і GUI для Windows або MacOS вражає найбільше.

• ***Утиліти.*** Сучасні ОС містять широкий діапазон простих службових програм, до яких належать калькулятори, календарі, текстові редактори, інструменти для обслуговування диска тощо.

• ***Бібліотеки.*** У Linux, наприклад, більшість програм ґрунтуються на бібліотеці *libc*. Інші бібліотеки надають функції, пов'язані з графічним інтерфейсом користувача або допомагають розібратися в параметрах, що передаються в командному рядку.

• ***Прикладні програми.*** На комп'ютері користувачі переважно працюють з такими прикладними програмами, як браузери, текстові редактори, графічні редактори та ін.

1.2.1. Режим ядра і режим користувача

У сучасних обчислювальних архітектурах центральний процесор працює у двох різних режимах: *користувача* і *ядра* (який іноді називають *захищеним*). Апаратні інструкції дозволяють перемикатися з одного

режиму на інший. Відповідно області віртуальної пам’яті можуть бути позначені як частина простору користувача або простору ядра. Під час роботи в режимі користувача центральний процесор може отримувати доступ тільки до тієї пам’яті, яка позначена як пам’ять простору користувача. Спроби звернення до пам’яті у просторі ядра призводять до виключення програми на апаратному рівні. Під час роботи в режимі ядра центральний процесор може отримувати доступ як до користувацького простору пам’яті, так і до простору ядра.

Системні змінні (*System state*) та область пам’яті, в якій міститься ядро, разом називають *простром ядра* (*kernel-space*) або ***привілейованим режимом***. Відповідно, програми користувачів виконуються у просторі завдань (*user-space*) або у ***режимі користувача***. Програмам користувача доступна лише деяка підмножина машинних ресурсів, вони не можуть виконувати деякі системні функції, безпосередньо працювати з апаратурою, звертатися до системної пам’яті (за межами адресного простору, яке ядром виділено користувальницькій програмі) і робити певні заборонені речі. Під час виконання програмного коду ядра система переходить у простір ядра або перемикається у привілейований режим. Коли запускаються звичайні процеси, система перемикається у режим користувача або непривілейований режим. Взаємодію прикладних програм із допомогою системних викликів (*system call*) з ядром, ядра з апаратним забезпеченням показано на рис. 1.1.

Прикладна програма зазвичай викликає функції різних бібліотек, наприклад бібліотеки функцій мови С, які, у свою чергу, звертаються до системних викликів, щоб віддати наказ ядру виконати певні дії від їхнього імені.

Фактично всі платформи, разом з тими, на яких працює ОС Linux, використовують механізм переривань (*interrupt*).

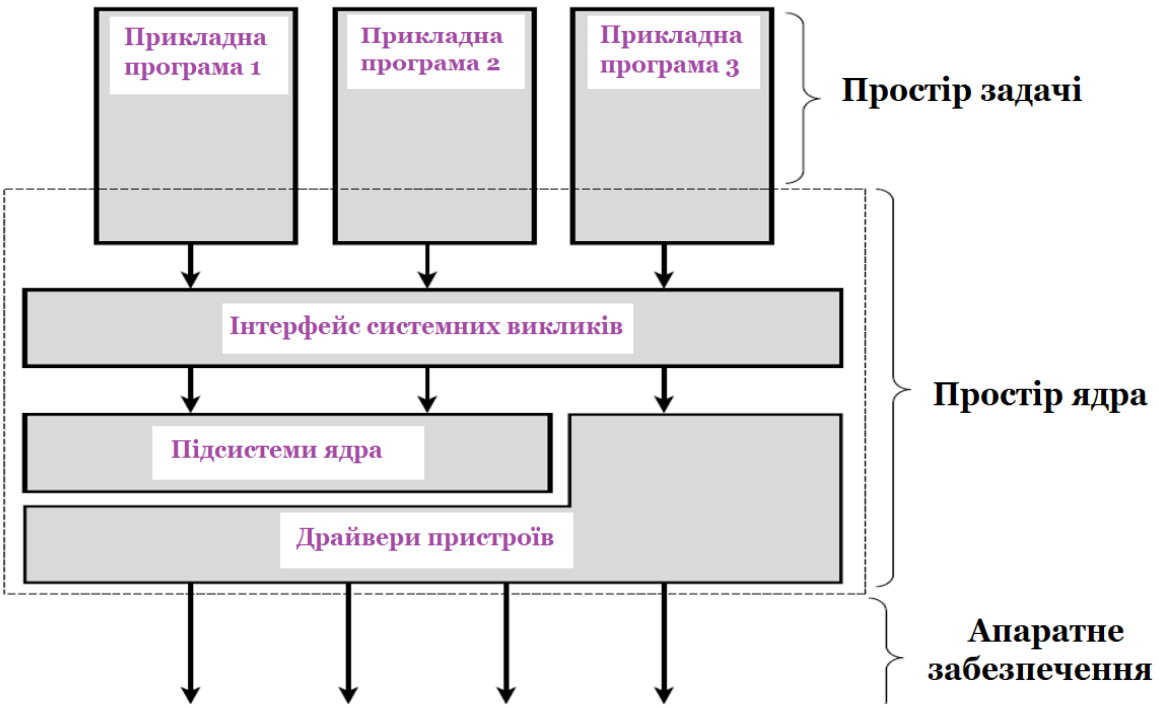


Рис. 1.1. Взаємодія прикладних програм, ядра та апаратного забезпечення [1]

Коли апаратному пристрою треба взаємодіяти із системою, він виставляє на шину спеціальний сигнал, який перериває роботу процесора. Процесор запускає спеціальну програму в ядрі. Кожному типу переривання призначається певне число або номер переривання. Ядро використовує цей номер для запуску спеціального обробника переривання (*interrupt handler*), який обробляє переривання і надсилає на цей номер відповідь. Наприклад, при введенні символу з клавіатури контролер клавіатури генерує переривання, щоб дати знати системі, що в буфері клавіатури є нові дані. Ядро визначає номер переривання, яке надійшло в систему, і запускає відповідний оброблювач переривання – він обробляє дані, що надійшли з клавіатури, і дає знати контролеру клавіатури про те, що ядро готове приймати нові дані.

Розрізняють ОС з *монолітним ядром* (один великий процес виконується в одному адресному просторі) і з *мікроядром* (виконується декілька процесів).

Операційна система Linux створена як модульна, багатопотокова, з пріоритетним плануванням ядра.

1.2.2. Конфігурація ядра Linux

Є чотири основні методи конфігурування ядра Linux [1, 2, 9]:

– модифікування параметрів ядра, які налаштовуються (динамічні);

– повторне створення ядра (компіляція вихідних файлів ядра з можливими модифікаціями та доповненнями);

– динамічне завантаження нових драйверів і модулів в існуюче ядро;

– передавання спеціальних інструкцій ядру під час початкового завантаження і / або з використанням завантажувача (наприклад, GRUB).

Найпростішим є перший – налаштування параметрів ядра. Найскладнішим є компіляція ядра з вихідних кодів.

Конфігурація для майбутньої збірки ядра Linux зберігається у файлі *.config*. Список основних команд (методів) для конфігурації збірки ядра:

– *config* – традиційний спосіб конфігурації. Програма виводить параметри конфігурації по одному, пропонуючи встановити для кожного з них своє значення. Викликається командою *make config*;

– *oldconfig* – файл конфігурації створюється автоматично, ґрунтуючись на поточній конфігурації ядра. Рекомендується для початківців. Викликається командою *make oldconfig*;

– *defconfig* – файл конфігурації створюється автоматично, ґрунтуючись на значеннях за замовчуванням для конкретної архітектури. Викликається командою *make defconfig*;

– *menuconfig* – псевдографічний інтерфейс ручної конфігурації, не вимагає послідовного введення значень параметрів. Рекомендується для використання у терміналі. Викликається командою *make menuconfig*;

– gconfig та xconfig – графічні конфігуратори для ручного налаштування. Викликаються командами *make gconfig* (графічне середовище GNOME) та *make xconfig* (графічне середовище KDE);

– *localmodconfig* та *localyesconfig* – автоматичні конфігуратори. Конфіг створюється на основі викликаних у певний момент модулів і запущеного ядра. Відмінність між цими двома конфігураторами у кількості модулів. У першому випадку їх буде не менше 50 % ядра, а у другому – не більше як 2 модуля. Викликаються командами *make localmodconfig* та *make localyesconfig* відповідно.

1.2.3. Компіляція ядра

Формування файлу ***.config*** – найважливіший етап конфігурації ядра Linux, але потрібно виконати ще ряд дій перш ніж буде отримане готове ядро.

Схема цього процесу така:

– перейти у каталог верхнього рівня, що містить вихідні коди ядра;

– виконати одну з команд *make xconfig*, *make gconfig* або *make menuconfig*;

– виконати команду *make clean*;

– виконати команду *make*;

– виконати команду *make modules_install*;

– скопіювати файл *arch/i386/boot/bzImage* під іменем */boot/vmlinuz*;

– скопіювати файл *arch/i386/boot/System.map* під іменем */boot/System.map*;

– відредагувати файл */etc/lilo.conf* (*LILO*) або */boot/grub/grub.conf* (GRUB) й додати в нього конфігураційні параметри для нового ядра.

Нові версії ядра можна скачати за адресою ***www.kernel.org*** або із сайту виробника дистрибутиву як оновлення системи. Оновлення ядра дозволяє отримати не тільки нові можливості щодо роботи із «залізом», але й підвищити продуктивність системи та виправити певні помилки. Ключовий момент полягає в тому, що оновлення ядра не передбачає, власне, переконфігурування системи, як це відбувається в інших ОС.

1.3. Драйвери пристроїв

Драйвер пристрою – це програма, яка організовує взаємодію системи з апаратним компонентом певного типу. Доступ до драйвера можливий як із ядра, так і з боку команд користувацького рівня. Для останніх створюються спеціальні файли пристроїв, що зберігаються в каталозі ***/dev.***

Файли пристроїв бувають двох типів: блочно-орієнтовані (блокові) і байт-орієнтовані (символьні). Читання з блочного пристрою і запис в нього здійснюються по одному блоку за раз (блок – це група байтів, розмір якої зазвичай кратний 512 байтам). Прикладом блочних пристроїв є диски та USB-накопичувачі. Для символьного пристрою читання з нього і запис в нього відбуваються в побайтовому режимі, тобто дані обробляються посимвольно. Прикладом таких пристроїв є термінали та клавіатура.

Кожний драйвер оснащений процедурами для виконання деяких або всіх таких функцій: *attach(), close(), dump(), open(), probe(), psize(), read(), receive(), reset(), select(), stop(), strategy(), timeout(), transmit(), write().*

У підкаталозі **drivers** дерева файлів початкового коду ядра розміщуються підкаталоги для відповідного типу пристрою, який додається. Ось як виглядає список наявих підкаталогів:

linux $ ls -F шлях_до_початкових_текстів_ядра/drivers

Найчастіше драйвери додаються в каталоги ***block/***, ***char/***, ***net/***, ***scsi/***, ***sound/*** та ***usb/***. У них зберігаються драйвери блокових пристроїв (наприклад, IDE-дисків – жостких дисків), символьних пристроїв (наприклад, послідовних портів), мережевих пристроїв, звукових плат і USB-пристроїв відповідно.

1.4. Файлові системи

Більшість розділів диска містять файлові системи, які являють собою структури даних, допомагають комп’ютеру організовувати каталоги і фай-

ли користувача. В ОС Windows кожній файловій системі присвоюється літера, що позначає пристрій, наприклад С: – для першого розділу - диска (зазвичай завантажувального розділу) і т. д. В Linux, навпаки, усі файлові системи є частиною єдиного дерева каталогів [1, 8, 9, 10].

Головну файлову систему називають кореневою – **root** (/), або «корінь» (з англ.). *Будь-який об'єкт у файловій системі ОС Ubuntu Linux вважається файлом*. Це означає, що звичний для нас каталог, який в ОС Windows називають папкою, – це файл, тобто послідовність байтів на фізичному диску. Якщо диск розділений на кілька файлових систем, кожна з них встановлена у *точці монтування* в кореневій файловій системі (/), тобто вміст додаткових файлових систем стає доступним у конкретних каталогах, наприклад /home (в якому зберігаються файли користувача з даними) або **/boot** (в якому зберігаються файли завантаження).

Є декілька файлових систем Linux, кожна з них має унікальні особливості [1]:

• **ext2fs (**Second Extended File System). Друга розширена файлова система була популярна в 1990-х роках, але сьогодні її майже не використовують.

• **ext3fs (**Third Extended Filesystem). Третя розширена файлова система – це система ext2fs, яка доповнена журналом. До 2010 року ця файлова була популярною, але її швидко змінила файлова система ext4fs. Вона підтримує файли розміром до 2 тебібайт і файлові системи до 16 тебібайт. **Тебібайт – це 2^{40} байт** (ця одиниця виміру інформації створена Міжнародною електротехнічною комісією). Цей тип файлової системи має код *ext3*.

• **ext4fs.** Четверта розширена файлова система є результатом удосконалення розглядуваної лінійки файлових систем. Вона характеризується більшою швидкістю і здатністю обробляти великі файли та диски, розмір файлу може досягати 16 тебібайт (2^{40} байт), а розмір

файлових систем – **1 ексбібайт (2^{60} байт**). (Для ОС Windows 1 кБ = 1024 Б, 1 МБ = 1024 кБ, 1 ГБ = 1024 МБ, 1 ТБ = 1024 ГБ, але у професійних оглядах ІТ-видань для ОС Linux використовують префікси «КіБ» – кібібайт, «МіБ» – мебібайт, «ГіБ» – гігібайт, «ТіБ» – тебібайт, «ПіБ» – пебібайт, «ЕіБ» – ексбібайт). Інструменти Linux використовують цю систему з кодом ext4. У системах *Ubuntu* та *SUSE* вона використовується за замовчуванням, а система *Red Hat* зберігає файлову систему *ext3*.

• **XFS.** Компанія *Silicon Graphics* розробила файлову систему, засновану на екстентах (Extents Filesystem, XFS) для своєї ОС IRIX, а згодом пожертвувала свій код на користь Linux. XFS підтримує файли розміром до 8 терабайт і файлові системи розміром до 16 ексабайт (10^{18} байт) (ЕІБ), що робить її найкращим вибором для роботи з дуже великими дисковими масивами. Система XFS добре працює з великими мультимедійними файлами і резервними копіями. У Linux для цієї системи використовує код **xfs**.

• **Btrfs (B-tree file system).** Ця нова файлова система була задумана як файлова система Linux наступного покоління. Зберігає інформацію у вигляді Б-дерев (це один із видів збалансованих дерев, що забезпечують ефективне збереження інформації на магнітних дисках та інших пристроях із прямим доступом) і працює за принципом «копіювання при записі» (copy-on-write). Опублікована компанією Oracle Corporation у 2007 році. Вона підтримує файли та файлові системи розміром до 16 ексабайт. Створена з метою реалізації додаткових функцій, які б покращили відмовостійкість, спростили адміністрування і ремонтні роботи. Крім того, вона передбачає безліч додаткових функцій, таких як об’єднання декількох фізичних дисків в одну файлову систему. Нещодавно систему Btrfs стали використовувати за замовчуванням у різних розділах Linux і її популярність зростає. В Linux для цієї системи використовується код btrfs.

• **Віртуальна файлова система (Virtual File System)**, або віртуальний файловий комутатор (Virtual File Switch), або просто **VFS**, – це підсистема ядра, в якій реалізований інтерфейс до файлів і файлових систем, що використовується в додатках користувача. VFS покладена в основу всіх файлових систем – це дозволяє їм не тільки разом співіснувати в одній ОС, а й взаємодіяти одна з одною. У результаті з'являється можливість використовувати в додатках користувача стандартні системні функції для читання і записування даних, розміщених у різних файлових системах і на різних фізичних носіях. Наприклад: жорсткий диск з файловою системою ext3, флеш-пам'ять має файлову систему ext2, а в ядрі файлова система VFS забезпечує взаємодію або інтерфейс різних файлових систем для роботи системних функцій, як open (), read () та write (), незалежно від типу файлової системи та фізичного середовища носія.

На сьогоднішній день найбільш вживаною є файлова система ext4, але Linux також може працювати з накопичувачами, відформатованими в ОС Windows файловими системами FAT32 та NTFS.

1.5. Архіви

В ОС Ubuntu, як і в будь-який інший ОС, використовуються архіви [1, 10]. Архіви призначені не тільки для зменшення обсягу займаного файлу, але й більшою мірою для зручності поширення кількох файлів. Операційна система Ubuntu за замовчуванням підтримує усі відомі типи архівів. Для зручності користування архівами у графічному режимі в системі є програма *Менеджер архівів*, яка дозволяє працювати з різними типами архівів в ОС Ubuntu Linux. За допомогою Менеджера архівів можно створювати нові архіви, переглядати і змінювати вміст наявних архівів, витягувати з архіву файли. Менеджер архівів дозволяє працювати із двома типами архівів:

– нестиснені типи архівів: *ar*, *iso* та *tar*;

– стиснені типи архівів: *jar*, *rar*, *tar.gz*, *tgz*, *tar.bz*, *tbz*, *tar.bz2*, *tbz2*, *tar.lzo*, *tzo*, *tar.7z*, *tar.xz*, *caЬ*, *cbz*, *zip* та *zoo.*

Найбільш поширеним нестисненим форматом є архів *tar*. Архіватор *tar* по суті не є архіватором, тому що під час створення архіву він не використовує стиснення файлів, і тільки після того, як архіватор *tar* створить архів, починає працювати інший архіватор, наприклад *gzip*, який вже стискає створений архів. Тому під час створення стисненого архіву ім'я файлу складається із двох розширень: *tar.gz*.

Архіватор tar – це найбільш поширений архіватор у сімействі ОС Linux. Він став стандартом де-факто в дистрибутивах на базі ОС Linux. Цей архіватор дозволяє архівувати файли й каталоги, оновлювати архіви, розпаковувати їх зі збереженням файлової структури з усіма атрибутами. Синтаксис програми tar такий: **tar [ключ] [файл]**

Параметр [файл] набуває ім'я одного або декількох файлів, які мають бути архівовані. Ключі (табл. 1.1) можна дізнатися з довідки за командою: **tar -help**. Наприклад, щоб створити архів з каталогу */home/sergey/sample/*, треба виконати в терміналі команду **tar -cf** new_archive.tar sample/

Отже, **-c** ***(--create***) – це створити архів, **-f** ***(--file***) – у вигляді файлу на диску.

Архіватор gzip (англ. GNU Zip) – це програма стиснення і відновлення файлів. Цей архіватор є найбільш популярним для стиснення даних і має формат GZ. Саме в цьому форматі поставляється більшість програм у вихідних текстах. Архіватор *gzip* використовується після оброблення файлу архиватором tar, тому отриманий файл має розширення *tar.gz*, або скорочене *tgz.*

Синтаксис команди gzip такий: **gzip [ключ] [файл]**

Архіватор bzip2. Крім архиватора gzip, є ще й архіватор bzip2, який трохи повільніший від gzip, але має більш високий ступінь стиснення. Цей архіватор під час роботи додає суфікс **.bz2**. За рахунок більш високого

ступеня стиснення архіватор bzip2 поступається за швидкістю іншим архіваторам. Синтаксис команди bzip2 такий: **bzip2 [ключ] [файл]** [1].

Таблиця 1.1

Значення параметра (ключ) команди tar

Скорочений варіант	Повний варіант	Опис
-A	--catenate, --concatenate	Дозволяє приєднати tar-файли до вже існуючого архіву
-b	--blocking-factor=*розмір_блоку*	Дозволяє розбити створюваний архів на блоки, які прирівнюються вказаному значенню *розмір_блоку*
-c	--create	Дозволяє створювати новий архів
-C	--directory=*каталог*	Дозволяє витягти вміст у зазначений каталог
	--delete	Дозволяє видалити непотрібний об'єкт з архіву
-f	--file=*архів*	Дозволяє створювати архів у вигляді файлу на диску. Задає ім'я архіву, який треба створити або витягти
-j	--bzip2	Дозволяє пропустити створений архів через архіватор *bzip2*. До імені файлу додається розширення *.bz2*
-J	--xz	Дозволяє пропустити створений архів через архіватор xz
-k	--keep-old-files	Дозволяє оновити архів без перезаписування наявних файлів
	--overwrite	Дозволяє перезаписати наявні файли під час витягання з архіву
-p	--preserve-permissions, --same-permissions	Дозволяє витягти інформацію про права доступу до файлу
-r	--append	Дозволяє додавати файли в кінець архіву
-t	--list	Дозволяє вивести список вмісту архіву
-u	--update	Дозволяє оновити файли в архіві (замінити на новіші)
-v	--verbose	Дозволяє вивести детальну інформацію про оброблювані файли
-x	--extract, --get	Дозволяє витягти файли з архіву
-z	--gzip, --gunzip, --ungzip	Дозволяє пропустити створений архів через архіватор gzip. До імені файлу додається розширення .gz
-Z	--compress, --uncompress	Дозволяє пропустити створений архів через compress

Інші архіватори. Відомі такі формати архівів – ZIP, RAR, 7Z. В Інтернеті найпоширенішими форматами архівів є ZIP та RAR. В ОС Ubuntu розробники передбачили можливість роботи із ZIP-файлами. Для цього є команда unzip, яка дозволяє витягти вміст ZIP-архіву: **unzip archive.zip.**

Більш детальну інформацію про команду unzip можна дізнатися, викликавши довідкову інформацію: **man unzip**.

Для створення архіву каталогу ~/sample/ у форматі ZIP призначена однойменна команда zip: **zip -r zip_archive sample.**

Параметр **-r** у цій команді вказує на те, що каталог буде оброблений рекурсивно, тобто в архів потраплять також усі вкладені каталоги й файли.

Щоб працювати з архівом у форматі **RAR**, необхідно встановити цей архіватор, використавши команду **sudo apt install rar**. Після підтвердження установки буде встановлений пакет rar, який надасть змогу працювати з RAR-архівами. Однак програма rar є триальною (пробною), тому буде доступний не весь функціонал. Щоб тільки витягти дані з архіву, треба встановити програму unrar: **sudo apt install unrar**. Після установки цієї програми можна отримати RAR-архів, виконавши таку команду: **unrar x rar_archive.rar**. Ця команда дозволяє витягти вміст архіву rar_archive.rar у поточний каталог, зокрема в домашній каталог користувача. Перед параметром **x** не ставиться символ «дефіс». Більш детально про команди *rar* та *unrar* можна дізнатися з довідкової інформації, виконавши команду **man rar** або **man unrar** відповідно.

Не менш поширеним форматом архіву є **7Z-архів**, який має суфікс 7z. Для вилучення файлів з таких архівів не потрібно встановлювати додаткові архіватори, оскільки цей формат підтримується програмою **7za**, яка встановлена в ОС Ubuntu.

Щоб вилучити архів у форматі 7Z, слід виконати команду **7za x 7z archive.7z**. Ця команда отримає всі файли зі збереженням структури

каталогів, тому що вказано параметр **x** без символу «дефіс» перед цим параметром. Після параметра введено ім’я архіву, який потрібно витягти.

КОНТРОЛЬНІ ЗАПИТАННЯ

1. Що таке ОС?
2. Що таке архітектура ОС?
3. Що таке POSIX і для чого він потрібний?
4. Які основні характеристики дистрибутиву?
5. Які основні дистрибутиви ОС *Linux*?
6. Що таке системне програмування *Linux*?
7. Що таке інтерфейс програмування додатків?
8. Які задачі виконує ядро?
9. З яких компонентів складається ОС?
10. В яких режимах працює центральний процесор?
11. Які методи конфігурації ядра вам відомі?
12. Які типи драйверів вам відомі?
13. Які файлові системи Linux вам відомі? Чим вони характеризуються?
14. Які архіватори в сімействі Linux вам відомі?

Розділ 2

КЕРУВАННЯ LINUX-ПАКЕТАМИ

2.1. Пакетний принцип організації операційних систем Linux

Усі ОС Linux організовані за пакетним принципом [1, 10, 11, 13].

Пакет – це початковий код (рос. *исходник*). Сукупність таких пакетів складає ОС або як додаткове застосування. Термін «пакет» використовують у двох варіантах:

– *як набір початкових текстів програм*, який створив розробник; тобто текст програми, написаний на певній мові програмування. Щоб програма запрацювала, її треба скомпілювати і перетворити її у виконуваний файл;

– *як виконуваний файл або бінарний*.

Бінарні пакети можуть різнитися один від одного форматом, наприклад, *.deb*.

Дистрибутив являє собою комплект програм, скомпільованих з початкових файлів, і деякий набір додаткових файлів, які зібрані за певним принципом. Такий процес збирання пакетів називають ***мантейн*** – «сборка» (*maintain*). Відповідно, фахівців, які створюють дистрибутив за певними принципами, називають *Maintainer*. ***Бібліотеки*** – це програмний модуль, який містить низку типових дій і типові функції. Для зберігання пакетів існує спеціальне сховище ***репозиторій***.

Офіційний репозиторій – сховища ПЗ, які можуть містити тисячі пакетів, кожний з яких був скомпільований, протестований і підтримується для поширення і використання у дистрибутиві Linux. *Це місце, звідки можна завантажити ПЗ*. Для Ubuntu доступні тисячі програм. Ці програми зберігаються у спеціальних сховищах ПЗ (репозиторіях) і легко доступні через інтернет, таким чином, можна встановити нові додатки.

Це перевірений і надійний спосіб, оскільки кожна програма в репозиторії зібрана і протестована спеціально для Ubuntu.

Репозиторії Ubuntu впорядковані у такі категорії: **Main, Restricted, Universe і Multiverse**. Це потрібно, щоб розділити програми з різними рівнями підтримки користувачів та різними умовами використання. Репозиторій дозволяє оновлювати пакети та усі їх залежності. Розглянемо, що таке залежності пакетів.

Додаткові файли, потрібні для роботи основного пакета, називають залежностями. тобто коли одному пакету ПЗ для коректної установки і роботи потрібні певні ресурси у вигляді додаткових загальних бібліотек або інших пакетів. Усі сучасні системи керування пакетами мають способи розв'язання (рос. разрешения) залежностей, щоб гарантувати, що під час установки пакета будуть встановлені усі його залежності, потрібні для нормальної роботи.

Пакети встановлюються з репозиторію спеціальними утилітами через командний рядок (термінал) за допомогою команд або через графічний інтерфейс за допомогою миші та кнопок.

2.2. Менеджери пакетів

Керування пакетами, або *Менеджер пакетів – це програма, яка здійснює установку та підтримку (оновлення, за потреби – видалення) ПЗ ОС.*

На ранніх стадіях розвитку ОС Linux ПЗ поширювалося тільки у вигляді початкового коду разом з іншими необхідними документами, файлами конфігурації тощо. У наш час більшість дистрибутивів Linux використовують вже скомпільовані програми, які називають *пакетами*. Пакети надаються користувачеві вже готовими для установки на ОС. Проте в Linux завжди можна отримати початковий код того чи іншого ПЗ для вивчення, поліпшення та компіляції.

Також менеджер пакетів відстежує залежності між програмами і зберігає систему цілою. *В Linux пакети мають такі характеристики*:

– кожний пакет являє собою єдиний файл, який можна зберігати на диску або передавати через Інтернет;

– файли пакетів у Linux, на відміну від інсталяторів у Windows, не є програмами; для установки додатків вони використовують зовнішні інструменти;

– пакети можуть містити інформацію про залежності, які сигналізують пакетним менеджерам про те, які ще пакети або окремі файли мають бути встановлені для коректної роботи пакета; багато програмних пакетів залежать від бібліотечних пакетів; бібліотеки надають код, який і використовує багато програм;

– пакети містять інформацію про версії для того, щоб пакетний менеджер міг визначити, який із двох пакетів новіший;

– пакети містять інформацію про архітектуру, щоб визначити тип центрального процесора (x86, x86-64, ARM – Advanced RISC Machines, британська корпорація тощо), для якого вони призначені; спеціальне позначення мають пакети, що не залежать від архітектури, наприклад шрифти або теми робочого стола;

– *бінарні пакети* (містять виконувані програми, зібрані для певного процесора) зазвичай збираються з *початкових пакетів* (складаються з початкового коду, зрозумілого для програміста). Маючи початковий пакет, можна його перетворити у бінарний, що корисно у деяких ситуаціях.

2.2.1. Формати пакетів програм Linux

Основні види форматів пакетів – формати файлів, які використовують системи керування пакетами ОС на основі Linux та GNU:

• *Бінарні (двійкові) пакети* – готові відкомпільовані пакети, закачані для конкретної системи, тобто це тільки виконувані файли [1, 10, 13].

• Пакети, що *містять початкові коди програм*, тобто це пакети, які потребують компіляції на локальній машині (створюється об’єктний код, а далі завантажник підключає потрібні додаткові файли, статичні, динамічні бібліотеки та створює виконуваний файл). Вони є більш універсальними, ніж бінарні, тому що можуть використовуватися для різних систем.

Розглянемо, що краще: бінарні пакети чи пакети з початковими кодами.

Бінарні пакети містять набагато більше інформації, яка полегшує роботу менеджера пакетів, ніж просто скомпільовані файли. Бінарні пакети дозволяють відслідковувати усі програми користувача. Наприклад, файли DEB (формат пакета для Debian і похідні Debian) також містять важливу інформацію, таку як «*яке ще ПЗ потрібне для запуску програми*», «*поточна версія програми*». Це значно спрощує установку пакетів. Користувачу не треба турбуватися про те, які інші файли потрібно встановити, щоб, наприклад, запустити програму. Оскільки менеджер пакетів може зчитувати цю інформацію, власне, з пакета, він обробляє все це автоматично.

Порівняймо це з *установкою програм із початкового коду*. Якщо не скомпілювати код у власний бінарний пакет, користувач відповідатиме за керування цим ПЗ. *Користувачу потрібно пам’ятати, які інші програми йому потрібні, щоб вони працювали, і встановлювати їх самостійно*.

Крім того, репозиторії пакетів, як правило, перевіряють свої двійкові файли на наявність проблем і роблять все можливе, щоб їх виправити, коли вони з’являються. Це може підвищити стабільність програм порівняно з тим, коли встановлення проводиться з початкового коду і цей момент просто буде втрачений.

2.2.2. Переваги компіляції початкових пакетів

Іноді для підвищення надійності програм необхідні виправлення, що потребує часу, і користувачу треба чекати, поки будуть усунені ці виправлення, тому він змушений використовувати стару версію ПЗ. Скомпілювавши власне ПЗ із початкового коду, користувач одразу ж зможе скористатися цими змінами.

Ще однією перевагою використання пакетів з вихідним кодом є те, що користувач отримує більший контроль над програмами, які він встановлює. Під час установки з бінарного сховища користувач обмежений у способах налаштування пакетів.

Є багато різних форматів Linux-пакетів. Більшість із них прив'язані до менеджерів пакетів певних Linux-дистрибутивів, наприклад, Debian-пакет (файли *.deb*), менеджер пакетів RPM (файли *.rpm*) та Тарбол (файли *.tar*).

Різні сімейства дистрибутивів Linux використовують різні системи упаковки – формати пакетів Linux (Debian – пакети у форматі *.deb*; CentOS – формат *.rpm*; openSUSE – теж *.rpm*, але створений спеціально для openSUSE), тому пакет, призначений для одного дистрибутиву, не буде сумісний з іншим дистрибутивом.

Більшість дистрибутивів Linux входять в одне із трьох основних сімейств Linux, включених у сертифікацію – *Linux Foundation Certified System Administrator* (*LFCS*).

2.2.3. Високо- і низькорівневі інструменти керування пакетами

Під час вирішення різних завдань з керування пакетами ПЗ треба знати, що є два типи утиліт [1]:

– низькорівневі інструменти (здійснюють фактична установка, оновлення та видалення файлів пакетів);

– високорівневі інструменти (відповідають за виконання завдань з вирішення залежностей і пошуку метаданих – «дані про дані»).

Низькорівневі системи керування пакетами:

- Debian, Ubuntu і подібні – менеджер пакетів *dpkg.*
- CentOS – менеджер пакетів *rpm.*
- OpenSUSE – менеджер пакетів *rpm* (*opensuse*).

Високорівневі системи керування пакетами:

- Debian, Ubuntu і подібні – *apt-get/aptitude.*
- CentOS – менеджер пакетів *yum*.
- OpenSUSE – менеджер пакетів *zipper*.

Дистрибутиви Linux на базі Debian використовують різні інструменти для роботи з пакетами, такі як **dpkg**, **apt**, **aptitude**, **synaptic**, **tasksel**, **deselect**, **dpkg-deb** та **dpkg-split**. Розглянемо кожний із них.

Apt – Advanced Package Tool. Цей інструмент працює з deb-архівами із джерел, зазначених у файлі конфігурації */etc/apt/sources.list.* Маючи права адміністратора та вибравши *Установка/удаление програм* з меню *Приложения*, можна встановити нові й видалити непотрібні програми.

Aptitude – інструмент для керування пакетами з командного рядка. По суті є зовнішнім інтерфейсом для інструмента *apt*, полегшує роботу з пакетами.

Synaptic – графічний пакетний менеджер, який дозволяє встановлювати, оновлювати і видаляти пакети, здійснювати розширене керування додатками й іншими компонентами системи.

Tasksel – дозволяє користувачеві встановлювати всі відповідні пакети, пов’язані з певним завданням.

Deselect – менеджер пакетів, працює через псевдоменю, натепер замінений aptitude.

Dpkg-deb – працює з файлами архівів Debian.

Dpkg-split – утиліта для поділу та об’єднання файлів великих файлів.

2.2.4. Менеджери пакетів, засновані на Debian

Менеджер пакетів Dpkg

Ubuntu і Debian вважають одними з найбільш широко використовуваних ОС на основі Linux, які є сьогодні на ринку. Їх менеджери пакетів є загальними, і належать до низькорівневої системи керування пакетами «*Dpkg*» [ді пі кей джей], скорочено від «Debian Package». Це скелет ПЗ для керування пакетами, з інструментами для установки, видалення та збирання пакетів, при цьому він не може автоматично завантажувати та встановлювати необхідні залежності для конкретних пакетів.

Команда dpkg: керування пакетами .deb

До корисних її опцій належать: ***--install***, ***--remove***, а ***-l*** виводить список пакетів, інстальованіх в системі:

1. *Отримання короткої довідки*:

$ dpkg --help

2. *Версія* dpkg:

$ dpkg --version

3. Dpkg, *встановлення пакета*, використовують команду:

dpkg --install

Щоб встановити *.deb*-пакет, використовують ключ ***-i***:

$ dpkg -i flashpluginnonfree_2.8.2+squeeze1_i386.deb

Команда виконана щодо пакета, який вже є в системі, й перед інсталяцією видаляє попередню версію пакета.

Наприклад, **sudo dpkg --install ./nvi_1.79-16a.1_i386.deb**

Щоб дізнатися, чи нормально пройшла інсталяція, треба скористатися командою **dpkg -*l* nvi**.

4. **Dpkg**, *список інстальованих програм*. Щоб переглянути список встановлених програм, використовують ключ ***-l***:

$ *dpkg -l*

Щоб дізнатися, чи встановлена конкретна програма, потрібно вказати її ім'я: **$ *dpkg -l nginx***

За допомогою команди $ **dpkg -L package** виводять список файлів пакета.

5. **Dpkg**, *видалити пакет*. Для видалення *.deb*-пакета використовують ключ **-r (remove)** із зазначенням імені пакета, наприклад "flashpluginnonfree", повну назву "flashplugin-nonfree_3.2_i386.deb" вказувати не обов'язково:

$ dpkg -r flashpluginnonfree

Щоб видалити пакет разом з файлами конфігурації, замість -r, використовують ключ ***-P (purge)***:

$ dpkg -P flashpluginnonfree

6. **Dpkg**, *перегляд вмісту пакета*, використовують **ключ -c (content)**:

$ dpkg -c flashplugin-nonfree_3.2_i386.deb

7. **Dpkg**, *перевірка, чи встановлений пакет*, використовують ключ **-s (status)**:

$ dpkg -s flashplugin-nonfree

8. **Dpkg**, *відображення місця, де встановлено файли пакетів*, **ключ -L**:

$ dpkg -L mysql-common

9. **Dpkg**, *установка всіх пакетів з конкретної директорії*, використовують ключі **-R** та **--install**. Наступна команда встановить усі * .deb-файли з директорії debpackages:

$ dpkg -R --install debpackages/

Менеджер пакетів APT

Менеджер пакетів APT (скорочено від *Advanced Package Tool*) має інтерфейси, такі як ***apt*** та ***aptitude***. APT набагато більш просунутий у функціональності порівняно з *dpkg*. Він також може встановлювати, видаляти і збирати пакети, однак його функціональність набагато ширша. APT може оновити свої пакети, встановити залежності автоматично, а також завантажити пакети з Інтернету. Це один з найбільш поширених менеджерів пакетів, встановлених на сучасних дистрибутивах, з попередньо встановленими на Ubuntu, Debian і більшості інших ОС на основі Debian.

2.2.4.1. Керування пакетами з командного рядка за допомогою APT

Для роботи в командному рядку з дистрибутивами *Linux (Debian, Ubuntu)* необхідно мати права адміністратора для використання *apt*. Для цього існують дві **утиліти** – ***Apt-get*** та ***Apt-cache***. Утиліта *Apt-get* працює з бібліотекою APT – це розширений інструмент упаковки, його використовують для установки нових пакетів ПЗ, видалення та оновлення наявних пакетів. Окрім того, *Apt-get* використовують для оновлення всієї ОС. Утиліту *Apt-cache* також використовують для пошуку пакетів ПЗ у кеші *apt*, збирання інформації про пакети, а також для пошуку готових пакетів для установки в ОС на базі Debian або Ubuntu. Зазвичай *apt-get* та *apt-cache* використовують спільно – *apt-get* для маніпуляцій з пакетами, *apt-cache* для отримання інформації.

Утиліта Apt-get

Розглянемо утиліту Apt-get.

1. Для *отримання списку команд та їх опцій* виконують команду:

apt-get help

2. **Apt-get**, *команда update оновлення локального списку пакетів, повторної синхронізації файлів* індексу пакетів з їх джерел, вказаних в */etc/apt/sources.list,* використовують команду ***apt-get*** із ключем ***update***.

Якщо в репозитариях доступні нові версії пакетів, вони будуть додані в індекс.

Виконувати *apt-get update* слід перед будь-якими маніпуляціями з пакетами!!!

Наприклад: якщо набрати в командному рядку

sudo apt-get update або **# apt-get update**

результатом команди може бути:

```
Hit http://mirror.mephi.ru wheezy Release.gpg
Get:1 http://mirror.mephi.ru wheezy-updates Release.gpg [1,554 B]
Hit http://mirror.mephi.ru wheezy Release
```

3. **Apt-get**, *команду upgrade використовують для оновлення всіх встановлених у системі пакетів*, що мають більш нову версію. Пакети, які вже встановлені в системі, не будуть видалятися в будь-якому випадку, так само як і не будуть встановлюватися пакети, яких не було в системі на момент поновлення.

Перед оновленням слід обов'язково виконати apt-get update, щоб система мала актуальну інформацію про наявні у цей момент версії пакетів.

Щоб встановити усі наявні оновлення, використовують команду

sudo apt-get upgrade або # apt-get upgrade

Результат команди буде, наприклад,

```
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following packages have been kept back:
```

linux-headers-generic linux-image-generic wine1.5 wine1.5-i386

The following packages will be upgraded:

activity-log-manager-common activity-log-manager-control-center adium-theme-ubuntu alacarte

alsa-base app-install-data-partner appmenu-gtk appmenu-gtk3 apport apport-gtk

4. **Apt-get**, *команда install-установка або оновлення пакетів.*

Щоб встановити .*deb*-файл, треба клікнути по ньому подвійним клацанням миші або, використовуючи командний рядок, ввести:

sudo apt-get install ім'я_пакета

sudo dpkg -i package_file.deb

Щоб встановити або оновити декілька пакетів, використовують команду *install* із зазначенням переліку імен потрібних пакетів:

 # apt-get install nethogs goaccess

Можна встановити кілька пакетів, замінивши частину імені символом маски "*":

 # apt-get install '*name*'

Apt-get, *оновлення конкретних пакетів*. Вказавши ключ **--only-upgrade**, apt-get оновлюють тільки встановлені пакети, установка нових пакетів при цьому буде відключено:

 # apt-get install pkg_name --only-upgrade

Apt-get, *встановити пакет певної версії*. Вказують ім'я пакета і версію, розділивши їх знаком "=":

 # apt-get install vsftpd = 2.3.5-3

5. **Apt-get**, *видалити пакет, залишивши файли конфігурації*. Щоб видалити пакет, не видаляючи при цьому файли конфігурації, використовують команду **remove**:

 # apt-get remove vsftpd

або **sudo dpkg -r** ім’я_пакета:

sudo apt-get remove package

Apt-get, *повністю видалити пакет*. Щоб видалити пакет разом з усіма його файлами конфігурації, використовують команду **purge**:

 # apt-get purge vsftpd

Purging configuration files for vsftpd ...

Крім того, можна комбінувати *remove* і *purge* в одній команді:

 # apt-get remove --purge vsftpd

6. **Apt-get**, *звільнити місце на диску*. Щоб звільнити певну кількість дискового простору, можна видалити з локального сховища викачані *deb* пакети:

 # apt-get clean

7. **Apt-get**, *скачати тільки вихідний код пакета*. Щоб завантажити вихідний код певного пакета, використовують ключ **--download-only source** з іменем відповідного пакета:

 # apt-get --download-only source vsftpd

Отримуємо:

```
Reading package lists ... Done
Building dependency tree
Reading state information ... Done
Need to get 216 kB of source archives.
Get: 1 http://mirror.mephi.ru/debian/ wheezy / main vsftpd 2.3.5-3 (dsc) [1,125 B]
Get: 2 http://mirror.mephi.ru/debian/ wheezy / main vsftpd 2.3.5-3 (tar) [188 kB]
Get: 3 http://mirror.mephi.ru/debian/ wheezy / main vsftpd 2.3.5-3 (diff) [26.9 kB]
Fetched 216 kB in 0s (252 kB / s)
Download complete and in download only mode
```

8. **Apt-get.** *скачати і розпакувати пакет*. Щоб завантажити вихідний код пакета і розпакувати його в певну директорію, використовують команду:

 # apt-get source vsftpd

9. **Apt-get**, *скачати, розпакувати і скомпілювати пакет*. Можна однією командою завантажити, розпакувати і скомпілювати пакет:

 # apt-get --compile source vsftpd

10. **Apt-get,** *скачати пакет без установки*. Щоб завантажити пакет без його *установки*, досить використавати команду **download** та ім'я пакета:

 # apt-get download nload

Отримаємо:

Get: 1 Downloading nload 0.7.4-1 [64.2 kB]

Fetched 64.2 kB in 4s (15.1 kB / s)

11. **Apt-get**, *подивитися список змін пакета*. Використовують команду **changelog**:

 # apt-get changelog nload

Отримаємо:

Get: 1 Changelog for nload (http://packages.debian.org/changelogs/pool/main/n/nload/ nload_0.7.4-1/changelog) [4,766 B]

Fetched 4,766 B in 0s (4,860 B / s)

nload (0.7.4-1) unstable; urgency = low

* [29fb6c5] Imported Upstream version 0.7.4

– Fixed incorrect output with amd64 kernel (Closes: # шістсот п'ятьдесят одна тисяча вісімсот сорок вісім)

12. **Apt-get**, *перевірити зламані залежності*. Використовують команду check:

 # apt-get check nload

Отримаємо:

Reading package lists ... Done

Building dependency tree

Reading state information ... Done

13. **Apt-get**, *знайти і встановити залежності пакета*. Використовують команду **build-dep**. Apt-get перегляне локальний репозитарій і встановить залежності пакета:

 # Apt-get build-dep nload

Отримаємо:

Reading package lists ... Done

Building dependency tree

Reading state information ... Done

The following NEW packages will be installed:

debhelper diffstat gettext html2text intltool-debian libgettextpo0 libunistring0 po-debconf quilt

0 upgraded, 9 newly installed, 0 to remove and 81 not upgraded.

Need to get 4,024 kB of archives.

After this operation, 10.8 MB of additional disk space will be used.

Do you want to continue [Y / n]? Y

14. **Apt-get**, *автоматично очистити кеш*. Використовують команду **autoclean**. Видаляє всі deb файли в папці */var/cache/apt/archives*, звільняючи місце на диску:

 # apt-get *autoclean*

15. ***Apt-get**, автоматично видалити пакет*. Використовують команду **autoremove**. Дозволяє видалити пакети, які були встановлені для

вирішення залежностей інших пакетів і в цей момент більше не потрібні. У прикладі нижче буде видалений пакет з усіма залежностями:

 # apt-get autoremove nload

Отримаємо:

```
Reading package lists ... Done
Building dependency tree
Reading state information ... Done
The following packages will be REMOVED:
  nload
0 upgraded, 0 newly installed, 1 to remove and 81 not upgraded.
After this operation, 145 kB disk space will be freed.
Do you want to continue [Y / n]?
(Reading database ... 58743 files and directories currently installed.)
Removing nload ...
Processing triggers for man-db ...
```

Наведені варіанти команд використовуються найчастіше, але це далеко не все. Більш повну інформацію можна отримати в офіційній документації:

 # Man apt-get або **# Man apt-cache**

Утиліта Apt-cache

Утиліта Apt-cache призначена для отримання інформації.

1. **Apt-cache**, *список всіх доступних пакетів*. Щоб отримати список всіх доступних пакетів, у системі використовують команду

 # apt-cache pkgnames

Лістинг буде досить значний, тому можна перенаправити виведення через утиліту ***more***, щоб була можливість гортати список:

 # apt-cache pkgnames | more

2. **Apt-cache**, *знайти пакет*. Використовують команду **search**:

 # apt-cache search ім’я_пакета

Команда *search* вказує *apt-cache* вивести список усіх пакетів, які збігаються з рядком пошуку, з їх короткими описами. Наприклад, знайдемо всі пакети, пов’язані іменем exim, введемо:

 # Apt-cache search exim

Отримаємо:

exim4 - metapackage to ease Exim MTA (v4) installation

exim4-base - support files for all Exim MTA (v4) packages

exim4-config - configuration for the Exim MTA (v4)

Або можемо вивести список пакетів, починаючи з рядка exim:

 # Apt-cache pkgnames exim

Отримаємо:

exim4-daemon-light

eximon4

exim4-config

3. **Apt-cache**, *отримання інформації про пакет*. Використовують команду **show**.

Якщо потрібно перевірити інформацію про пакет, наприклад версію пакета, контрольні суми, розмір до і після *установки* тощо, використовують підкоманду show:

 # apt-cache show exim4

Отримаємо:

Package: exim4

Version: 4.80-7 + deb7u4

Installed-Size: 44

Maintainer: Exim4 Maintainers

Architecture: all

Depends: debconf (> = 0.5) | debconf-2.0, debconf (> = 1.4.69) | cdebconf (> = 0.39), exim4-base (> = 4.80), exim4-daemon-light | exim4-daemon-heavy | exim4-daemon-custom

Description: metapackage to ease Exim MTA (v4) installation

Homepage: http://www.exim.org/

Description-md5: 458592f74d76e446735736c1d55ce615

Section: mail

Priority: standard

Filename: pool/updates/main/e/exim4/exim4_4.80-7 + deb7u4_all.deb

Size: 7790

MD5sum: 13bc5ca11b6cbb67d1c5c978fe975a58

SHA1: 6aae2042ee6fd6ba6c9ccdbd3a849620aa6fe690

SHA256: 90db48df64348b333805ea8af7bb4339ff164a3ed27f2c74e070c712ffb3db1c

4. **Apt-ceche**, *отримати інформацію про залежності пакета*. Використовують команду **showpkg**

Щоб отримати інформацію про прямі й зворотні залежності конкретного пакета, використовують підкоманду showpkg з іменем потрібного пакета:

 # apt-cache showpkg ftp

Отримаємо:

Package: ftp

Versions:

0.17-27 (/var/lib/apt/lists/mirror.mephi.ru_debian_dists_wheezy_main_binary-amd64_ Packages) (/var/lib dpkg/status)

Description Language:

File: /var/lib/apt/lists/mirror.mephi.ru_debian_dists_wheezy_main_binary-amd64_Packages

MD5: ce93e483dfc5fef0655f73f364b3c01c

Description Language: en

File: /var/lib/apt/lists/mirror.mephi.ru_debian_dists_wheezy_main_i18n_Translation-en

MD5: ce93e483dfc5fef0655f73f364b3c01c

Reverse Depends:

webcam, ftp

ftp-ssl, ftp 0.10

ftp-ssl, ftp 0.10

heimdal-clients, ftp 0.16-1

gabedit, ftp

Dependencies:

0.17-27 - libc6 (2 2.11) libreadline6 (2 6.0) netbase (0 (null)) netstd (0 (null))

Provides:

0.17-27 -

Reverse Provides:

ftp-ssl 0.17.23 + 0.2-1 + b1

heimdal-clients 1.6 ~ git20120403 + dfsg1-2

5. **Apt-cache**, *статистика кеша пакетів*. Використовують команду **stats**, що відображає загальну статистику кешу пакетів:

 # apt-cache stats

Отримаємо:

Total package names: 46396 (928 k)

Total package structures: 46396 (2,598 k)

Normal packages: 35015

Pure virtual packages: 524

Single virtual packages: 4057

Mixed virtual packages: 1078

Missing: 5722

Total distinct versions: 37929 (2,731 k)

Total distinct descriptions: 72393 (1,737 k)

Total dependencies: 226429 (6,340 k)

Total ver / file relations: 39862 (957 k)

Total Desc / File relations: 72393 (1,737 k)

Total Provides mappings: 7847 (157 k)

Total globbed strings: 85 (670)

Total dependency version space: 939 k

Total slack space: 34.7 k

Total space accounted for: 12.8 M

Перетворення файлів «.rpm» в файли «.deb»

Ще один тип пакетів – файли менеджера пакетів *Red Hat*, які мають розширення *.rpm*. Їх не рекомендовано встановлювати в Ubuntu. Такий самий пакет у форматі *.deb* здебільшого вже є. Однак за потреби *.rpm*-файл може бути перетворений в пакет .deb, використовуючи утиліту ***alien***, яка конвертує пакети у deb-формат:

1. Встановлюють пакет *alien*: **sudo apt install alien**
2. У Терміналі набирають: **sudo alien файл_пакета.rpm**

Установка з "Тарбола"

Файли з розширеннями *.tar*, *.tgz*, *.tar.gz* або *.tar.bz2* – пакети, відомі як *tarballs* (тарбол), які також використовують в Linux та Unix. Якщо в репозиторіях немає потрібного пакета для Ubuntu, використовуючи командний рядок, можна встановити / видалити пакет у форматі tar.

Першим кроком треба розпакувати і витягти файли із tarball. Якщо це *.tgz* або *.tar.gz*, у Терміналі вводять: ***tar xfvz ім'я_файла_tarball***

Потім потрібно виконувати інструкцію, яка додається до пакета. Зазвичай це файл README або INSTALL.

Пакети tarball часто містять вихідні коди програми, які мають бути скомпільовані перед використанням. Для компіляції програм потрібні пакети, які за замовчуванням не встановлено. Їх можна встановити пакетом *build-essential*.

2.2.4.2. Менеджер пакетів Aptitude

Менеджер пакетів Aptitude дуже схожий на APT і пропонує більшу частину такої самої функціональності. Але він може запропонувати кілька додаткових функцій, таких як безпечні оновлення, що дозволяють користувачам оновлювати свої пакети, не видаляючи наявні пакети із системи. Також є утримання пакетів, що запобігає автоматичному оновленню деяких пакетів.

Використовувати чи ні у прикладах *aptitude* або *apt-get* – це лише звичка.

Наведемо перелік команд та їх призначення для роботи з менеджером пакетів Aptitude:

$ **aptitude install package** # встановити пакет.

$ aptitude safe-upgrade package # оновити пакет.

$ **aptitude update** # перевірити і встановити оновлення.

$ **aptitude remove package** # видалити пакет.

$ **aptitude purge package** # видалити пакета з кінцями, усі дані і настройки.

$ **apt-get dist-upgrade # оновити ОС, killer-feature і воно працює!**

$ aptitude search package # шукати пакет.

$ **apt-cache depends package** # залежності пакета.

$ **apt-cache rdepends package** # зворотні залежності від пакета.

2.2.4.3. Менеджер пакетів Synaptic

Synaptic – менеджер пакетів, який є графічною оболонкою для утиліт, що дозволяє керувати пакетами в Linux – здійснювати пошук і установку пакетів, їх оновлення та видалення. Утиліта дозволяє відслідковувати залежності, і під час видалення основного пакета можна видаляти також залежні пакети [1].

Запустити пакет *Synaptic* можна і з головного меню **Система** через розділ **Адміністрування → Менеджер пакети Synaptic**. При цьому треба мати права адміністратора і ввести пароль.

У верхній частині програми є рядок з випадними меню.

Кнопка *Обновить* дозволяє заново зчитати базу пакетів з репозиторію. Додаток Synaptic в інтерфейсі дозволяє шукати, вибирати, встановлювати і видаляти ПЗ в Debian-сумісних системах. Вікно поділене на декілька частин.

У лівій частині вікна міститься список розділів репозиторію. Після обрання певного розділу у правій частині відображається перелік пакетів цього розділу. Після обрання певного пакета у нижній частині області відображаються властивості цього пакета. Напроти кожного пакета є два значки – перший (квадратик) вказує на статус пакета (чи встановлений він на комп'ютері та в якому стані він перебуває), другий показує, підтримується пакет чи ні (жовтий колір свідчить про те, що він офіційно підтримується розробниками). Натискаючи на пакети за допомогою контекстного меню, можна відмічати пакети для установки на комп'ютері або видаляти їх. Щоб обрана дія була виконана, треба натиснути на кнопку *Применить.* Для пошуку натискають кнопку *Поиск,* обравши певний критерій.

Також у правій частині, окрім кнопки *Розділи*, є й інші кнопки – *Состояние, Происхождение*, *Спеціалізовані фільтри, Результатіы поиска.*

2.2.4.4. Диспетчер пакетів RPM (Red Hat Package Manager)

У системах Linux широко поширені два формати пакетів. У Red Hat, SUSE і більшості інших дистрибутивів застосоввано диспетчер пакетів RPM (Red Hat Package Manager). В Ubuntu використовують пакети окремого формату .deb (названий «на честь» дистрибутиву Debian, на основі якого було створено дистрибутив Ubuntu). Обидва формати функціонально ідентичні [1].

Системи упаковки *RPM* і *.deb* працюють у вигляді дворівневих засобів керування конфігурацією. На нижньому рівні містяться засоби, які інсталюють, деінсталюють і запитують пакети: *rpm* для RPM і *dpkg* для *.deb*.

Над цими командами є системи, які знають, як потрібно проводити пошук пакетів в Інтернеті, аналізувати залежності між пакетами і модернізувати всі пакети у системі. Система **yum (Yellowdog Updater, Modified) працює з системою RPM**. Система **Red Hat Network працює для Red Hat Enterprise Linux і використовує RPM**. Система Advanced Package Tool (APT) спочатку була створена для роботи з пакетами .deb, а нині вона може працювати також з пакетами RPM.

Основні операції з yum

Розглянемо основні опреції, які можна виконувати, використовуючи yum:

yum search <рядок> – пошук текстового рядка у назвах пакетів і коментарях.

yum install <пакет> ... – установка пакетів і всього, що для них потрібно.

yum install <файл.rpm> – установка з локального файлу.

yum upgrade <пакет> ... – оновлення пакетів до останньої версії.

yum downgrade <пакет-версія> ... – відкат оновлення до певної версії.

yum remove <пакет> ... – видалення пакета. Якщо цей пакет потрібний іншим, то будуть видалені всі.

*yum list z** – список пакетів на букву z. Пакети розбиті на встановлені та доступні.

yum info <пакет> – перегляд інформації про пакет.

yum repolist – список усіх репозиторіїв.

yum clean – очищення кеша.

Відмінність *rpm* від *.deb* полягає у тому, що *rpm* для свого зберігання використовує контейнер *cpio*, а стиснення використовувалося утилітою *gzip*, у більш пізніх версіях використовувався архіватор *bzip2*, тобто основна відмінність полягає в технології стиснення та обробки пакета (бінарного коду).

До ***основних переваг RPM*** належать такі:

• Цей диспетчер є поширеним. Багато дистрибутивів Linux можуть встановлювати RPM-пакети або використовувати формат RPM як власний формат упаковки файлів. Крім того, RPM перенесений на чимало інших операційних систем.

• Він дозволяє встановлювати RPM-пакети за допомогою однієї команди. Його можна встановлювати автоматично, тому що формат RPM розроблений для роботи без обслуговування застосування. Видалити або оновити пакет також можна однією командою.

• Можливість працювати з одним файлом. RPM-пакет зберігається в єдиному файлі, полегшуючи перенесення пакета з однієї системи в іншу.

• RPM автоматично виконує перевірку залежностей. RPM-система включає базу даних усіх пакетів, встановлених користувачем, разом із даними про те, що саме кожний пакет дає системі та які вимоги до кожного пакета.

• RPM-пакети розроблені для формування виконуваних файлів з вихідних, дозволяючи користувачеві відтворювати збірку. Диспетчер RPM підтримує засоби ОС *Linux*, наприклад, команду *patch* для внесення змін у програмний код у процесі компіляції.

Утиліта *rpm* встановлює, перевіряє пакети і визначає їх стан. Раніше ця команда створювала пакети, але тепер цю операцію виконує команда *rpmbuild*. Однак опції *rpm* можна використовувати в різних комбінаціях і сприймати утиліту *rpm* потрібно таким чином – це кілька різних команд з одних і тим самим іменем.

Синтаксис утиліти; **$ rpm -режим опції ім'я_пакета**.

Режим, який користувач обирає для роботи *rpm* (наприклад, *-i* або *-q*), визначає, до яких функцій він хоче звертатись. Команда ***rpm --help*** виведе усі опції, розбиваючи їх на режими.

Зазвичай використовують такі опції:

-i *(install – встановити*);

-U (*upgrade – оновити*);

-V (*перевірити пакети*);

-e (*erase – видалити*);

-q (*query – запит на отримання інформації*);

-l (список файлів пакета).

Останню опцію використовують для включення інших опцій. Наприклад, команда *rpm -qa* відображає список усіх пакетів, встановлених у системах.

Команда установки буде виглядати так: **sudo rpm -i ім'я_пакета.rpm**

Для роботи з командою поточного каталогу має бути папка з пакетом. Тут слід встановити режим установки й передати файл пакета. У разі успішної установки утиліта не виведе нічого, якщо станеться помилка, користувач про це дізнається.

Щоб подивитися більш детальну інформацію у процесі установки, використовують опцію ***-v***: ***sudo rpm -iv ім'я_пакета.rpm***

Також можна відобразити статус бару у процесі установки:

sudo rpm -ivh ім'я_пакета.rpm

Щоб перевірити, чи встановлений пакет, потрібно використовувати режим запиту: ***sudo rpm -q ім'я_пакета***.

Наведемо ще один приклад використання утиліти *rpm*: потрібно встановити нову версію пакета OpenSSH, при цьому в опублікованому бюлетені повідомляється, що в попередній версії є прогалина. Після того,

як пакет завантажений на локальний комп'ютер, для його встановлення достатньо ввести команду *rpm -U*:

redhat $ sudo rpm -U openssh-2.9p2-12.i386.rpm

Установка RPM-пакетів в Ubuntu 20.04 (LTS) x64

Є два способи встановити RPM-пакет. Одним із них є перетворення файлу *.rpm* у файл *.deb*, а інший – установка безпосередньо *.rpm*-пакета. Розглянемо метод установки пакетів RPM в Ubuntu., а саме пакета під назвою «***alien»***. Він перетворює файли .rpm у файли .deb. Встановимо alien, додавши репозиторій ПЗ. Користувач має запустити таку команду у своєму терміналі, щоб додати репозиторій «Universe»:

sudo add-apt-repository universe

Далі поновити даний репозиторій: **sudo apt-get update**

Встановити *alien*, виконавши таку команду у своєму терміналі:

sudo apt-get install alien

Після установки *alien*, користувач зможе конвертувати *.rpm*-файли у *.deb*-файли. Для цього він має перейти у папку, де у нього є файл *.rpm*, і виконати команду

sudo alien <name of package> .rpm

Тепер файли будуть конвертовані у формат *.deb*.

Можна легко встановити перетворений файл, використовуючи команду

sudo dpkg -i <name of package> .deb

Перевірка автентифікації (достовірності) пакета

Довіреними ключами у пакеті **apt** керують командою **apt-key**. Ця програма підтримує кільце (зв'язку) для ***GnuPG*** (***GNU Privacy Guard – програма для шифрування інформації та створення електронних***

цифрових підписів) – відкритих ключів, які використовують для перевірки підписів у файлах Release.gpg, доступних на дзеркалах. **GNU Privacy Guard** – вільна програма для шифрування інформації і створення електронних цифрових підписів.

Адміністратори мають спочатку перевірити автентичність імпортованих ключів з використанням **опції fingerprint команди apt-get**, перш ніж довірити їм установку нових пакетів. Наприклад, перед інсталяцією пакета debianarchive-keyring необхідно перевірити підпис або відповідний ключ. Ключі розміщено в /etc/apt/trusted.gpg.d.

 # apt-key fingerprint /etc/apt/trusted.gpg.d/debian-archive-jessie-automatic.gpg

Процес роботи з GnuPG поділяється на такі етапи:

1. *Генерація ключів для підпису та шифрування, а також сертифіката для відкликання ключа*. Нову пару ключів генерують командою gpg **--gen-key**

2. *Експорт публічного ключа у файл, який потім передається всім охочим*. Перш ніж надіслати комусь відкритий ключ, слід його експортувати командою **--export**:

gpg **--armor --output alice.gpg --export alice@wonderland.uk**

Опція **--armor** вказує на необхідність виведення у форматі ASCII.

Опція **--output** вказує на необхідність збереження результату у файл **alice.gpg**.

3. *Імпорт публічних ключів, отриманих від партнерів в особисту «в'язку ключів»*. Після обміну ключами відкритий ключ партнера може бути доданий до зв'язки відкритих ключів за допомогою команди **--import**:

gpg --import blake.asc

Щоб переглянути ключі, використовують команду **gpg --list-keys**

Для перевірки достовірності ключа використовують відбиток (fingerprint) ключа: **gpg --fingerprint alice@wonderland.uk**

Підпис довіреного ключа

Для підпису ключа слід перейти у режим редагування ключа за допомогою команди **--edit-key**: **gpg --edit-key blake@anywhere.ru**

У діалоговому режимі вводять команду **sign**:

gpg> sign

Далі треба підтвердити бажання підписати ключ і ввести пароль секретного ключа, а потім вказати, з яким ступенем достовірності перевірений ключ:

gpg> trust

(0) I will not answer. (Default).

(1) I have not checked at all.

(2) I have done casual checking.

(3) I have done very careful checking.

Щоб вийти з режиму редагування і збереження змін, використовують команду

gpg> quit

4. *Шифрування і перевірка цифрового підпису публічними ключами партнерів.* Для шифрування документа використовують команду **--encrypt**. Для шифрування необхідно мати відкриті ключі передбачуваних одержувачів. Програма очікує параметр – ім’я документа, який треба шифрувати або, якщо імені немає, шифрує стандартне введення. Зашифрований результат поміщається у стандартне виведення, зазначена опція **--output**. Для підвищення захисту перед шифруванням документ додатково стискається:

gpg --output message.gpg --encrypt --recipient blake@anywhere.ru message.txt

Опція **--recipient** може повторюватися кілька разів для кожного одержувача і має як аргумент ідентифікатор відкритого ключа, яким має бути зашифрований документ.

5. *Дешифрування і цифровий підпис документів особистим секретним ключем.* Для розшифрування (дешифрування) повідомлення використовують команду **--decrypt**. Зашифрований документ може розшифрувати тільки той користувач, чий секретний ключ відповідає одному із зазначених відкритих ключів. Зокрема, відправник не можете розшифрувати зашифрований своїм ключем документ, якщо він не включив свій відкритий ключ у список одержувачів: **gpg --output message.txt --decrypt message.gpg**

Щоб отримати доступ до секретного ключа, треба ввести пароль.

Симетричне шифрування

Документи можна шифрувати і без відкритого ключа. Замість нього використовують симетричний алгоритм для шифрування документа. Ключ, використовуваний для шифрування, утворюють із ключової фрази. Для більшої безпеки ця ключова фраза не повинна збігатися з тією, яку використовують для захисту секретного ключа. Симетричний шифр застосують, коли є можливість обмінятися ключовою фразою. Для використання симетричного шифру використовують команду **--symmetric**:

gpg --output message.gpg --symmetric message.txt

Enter passphrase

Цифровий підпис засвідчує автора і дату створення документа. Якщо документ буде якимось чином змінено, то перевірка цифрового підпису буде невдалою. Для підпису документа використовують закритий ключ

підписувача, а перевіряється підпис з використанням його відкритого ключа.

Для підпису документів використовують команду **--sign**:

gpg --output file.sig --sign file.bin

Зазвичай цифрові підписи застосовують для надсилання електронної пошти. При цьому небажано стискати документи, які було підписано. Команда **--clearsign** додає до документа цифровий підпис у форматі ASCII, не змінюючи при цьому самого документа: **gpg --output doc.asc --clearsign doc.txt**

Щоб перевірити підпис, потрібна наявність і підпису, і документа. Для перевірки використовують команду **--verify**:

gpg --verify doc.sig doc.txt

КОНТРОЛЬНІ ЗАПИТАННЯ

1. Що таке пакет?
2. Що таке бінарний пакет?
3. Що таке репозиторій?
4. У чому полягають переваги компіляції початкових пакетів?
5. Які вам відомі менеджери пакетів?
6. Які можливості менеджера пакетів *Apt*?
7. Які можливості утиліти *apt-cache*?
8. Що таке RPM-пакети? Які їх можливості?
9. Як здійснюється автентифікація пакетів?
10. Що таке симетричне шифрування?

Розділ 3

ФАЙЛОВА СИСТЕМА LINUX

3.1. Особливості файлової системи

3.1.1. Організація файлової системи

Одним із найбільш важливих компонентів ОС є файлова система. В ОС Linux, як і в будь-якій іншій ОС сімейства UNIX, будь-який об'єкт є файлом, який зберігається у файловій системі. *В Linux об'єктами файлової системи є процеси, пристрої, структури даних ядра і параметри налаштування, канали міжзадачної взаємодії, папки, звичайні файли.* Фізично файлова система являє собою деякий пристрій (наприклад, жорсткий диск, SSD-накопичувач, USB флеш-накопичувач), від форматований для зберігання файлів.

Основними завданнями файлової системи є:

– упорядкування збережених даних;

– простий і швидкий доступ до збережених даних;

– забезпечення цілісності даних, що зберігаються.

Точний формат і способи зберігання файлів в ОС Linux не мають значення, тому що система забезпечує загальний інтерфейс для усіх типів файлових систем, які вона розпізнає. *В ОС Linux за замочуванням встановлюється файлова система ext4fs, яка є стандартом*. У разі доступу до будь-якої файлової системи з ОС Linux дані подаються у *вигляді ієрархії каталогів з розміщеними в них файлами разом з ідентифікаторами власників і груп, бітами прав доступу та іншими атрибутами*. Оскільки розробники Linux визнавали необхідність стандартизації структури каталогів, зокрема, програми мають знаходити певні системні файли конфігурації в одних і тих самих каталогах у всіх дистрибутивах, тому був створений Стандарт ієрархії файлової системи (*Filesystem Hierarchy Standard, FHS*). Тому ієрархія каталогів файлової

системи ОС Linux відповідає загальноприйнятому у світі UNIX стандарту FHS. Основна перевага такого стандарту полягає в тому, що певні типи файлів розміщені у відповідних каталогах. Наприклад, більшість конфігураційних файлів розміщується в каталозі ***/etc/***. Це файли, які визначають спосіб функціонування окремих програм та сервісів. Каталог ***/var/*** містить змінні файли, які належать до системи, або окремі додатки, вміст яких часто змінюється у процесі звичайної роботи системи. Файли журналів різних сервісів розміщуються в каталозі ***/var/log***, а власні документи – у ***/home***.

Таким чином, файлова система – це єдина ієрархічна структура, яка починається з каталогу «/» і розгалужується, охоплюючи довільну кількість підкаталогів. Каталог верхнього рівня називають кореневим.

Стандарт FHS розрізняє файли спільного доступу і файли без спільного доступу. Файли *спільного доступу* (файли даних користувача і програмні виконувані або двійкові файли) можуть в розумних межах переноситися між комп'ютерами. *Файли без спільного доступу* містять інформацію про конкретну систему (наприклад, файли конфігурації).

Стандарт FHS також розрізняє статичні (тобто ті, що не змінюються) і змінювані файлами. *Статичні* файли зазвичай не змінюються, за винятком випадків прямого втручання системного адміністратора. До статичних файлів відносять виконувані файли, бібліотеки, документацію та інші файли, змінювати які може тільки адміністратор системи. Для інших користувачів ці файли мають бути доступні тільки у режимі читання. Змінювані файли – це ті, які будь-який користувач може змінювати без залучення адміністратора.

Імена файлів

Файлові системи ext2, ext3, ext4 допускають наявність в іменах файлів будь-яких символів, крім роздільника «директорія» (/) довжиною до 256 символів. Рекомендується *використовувати такий набір символів* –

латинські літери (*великі іймаленькі*), *цифри*, *знак підкреслення*, *дефіс* (але не на початку), *крапка*. Не варто також починати імена файлів з дефіса (-), тому що багато програм, які працюють із файлами, приймають у командному рядку ключі (опції), які починаються з дефіса.

Також імена файлів є чутливими до регістру (*case sensitive*) – заголовні та маленькі букви в іменах розрізняються.

Розширення файлів

Файли, імена яких починаються із крапки (.), мають атрибут «*прихований*», тільда (~) після імені файлу додає атрибут «*архівний*» і робить файл «прихованим». Наприклад, ***.file*** – прихований файл, ***file~*** – архівний.

У Linux поведінка системи щодо того чи іншого файлу залежить від його типу, а для можливості запуску файл повинен мати встановлений атрибут «*виконуваний*».

Абсолютні та відносні шляхи до файлу

Абсолютний шлях починається від кореня файлової системи зі слеша «/» і вказує на те, де розміщений файл щодо кореневого каталогу. Приклади абсолютного або повного імен для файлів: */home/mtk/.bashrc*, */usr/include*.

Відносний шлях – це шлях до файлу відносно поточного каталогу, на відміну від абсолютного імені, не має початкового слеша і крапки «.», наприклад: *include/sys/types.h*, *../mtk/.bashrc*.

Шлях відносно домашнього каталогу поточного користувача – шлях у файловій системі, тільки не від кореня, а від каталогу поточного користувача, позначається **~/**, наприклад: *~/tmp/file1*.

Під час утановки системи наполегливо рекомендується розбивати жорсткий диск вручну, створюючи щонайменше два розділи (для кореня

файлової системи і для /home), що надалі полегшує процеси відновлення, перевстановлення, відновлення системи або перехід на інший дистрибутив.

3.1.2. Жорсткі диски: /dev/sd*

Більшості жорстких дисків у сучасних системах Linux відповідають імена пристроїв із префіксом *sd,* наприклад, /*dev/sda*, */dev/sdb*, */dev/sdc* і под., тобто в системі три диски [1, 8, 9]. Диск містить два або більше розділів, для розділів диска ядро створює окремі файли пристроїв, наприклад */dev/sda1* та */dev/sda2*. Диск містить таблицю розділів, кожний розділ містить структуру даних файлової системи та дані у файлі. Для створення розділів використовують утиліту **fdisk**.

Для роботи із системою використовують ***командний рядок***, який є інтерфейсом (взаємодією) між ОС та користувачем. Він дозволяє вводити текстові команди керування ОС або запитувати дані, якими вона керує. Командний рядок викликається як елемент із назвою **Термінал** сполученням клавіш **Ctrl+Alt+T**. За замовчуванням, відкриваючи оболонку командного рядка, користувач потрапляє в домашній каталог /home/ім’я користувача/.

Щоб переглянути список дисків, підключених до комп’ютера, треба виконати команду ***fdisk -l***

Для створення розділів на *sdd-накопичувачі* виконують команду

fdisk /dev/sdd

а на *жорсткому диску*

fdisk /dev/sd2/

Щоб додати новий розділ, використовують ключ ***-n***, щоб видалити розділ – ключ ***-d***, щоб вивести таблицю розділів – ключ ***-p***.

Оптичні приводи CD та DVD система Linux розпізнає як /dev/sr*, тобто використовуються імена /dev/sr0, dev/sr1 тощо.

Щоб створити різні типи файлових систем, використовують утиліту **mkfs**. Наприклад, можна створити розділ типу ext4 у пристрої /dev/sdf2 за допомогою такої команди: **# mkfs -t ext4 /dev/sdf2**

У більшості сучасних систем Linux використано ідентифікатор UUID (*Universally Unique Identifier*) – це універсальний, унікальний ID-ідентифікатор файлової системи для ідентифікації пристрою, що під'єднується. Такий підхід запобігає виникненню плутанини із присвоєнням імен пристроїв, які приєднуються до системи (оскільки ядро нумерує всі пристрої, що приєднуються фізично до комп'ютера у міру їх підключення).

UUID є 16-байтним (128-бітним) номером. У канонічному поданні UUID зображують у вигляді числа у шістнадцятковій системі счислення, розділеного дефісами на п'ять груп у форматі *8-4-4-4-12*. Таке подання має 36 символів: 123e4567-e89b-12d3-a456-426655440000

3.1.3. Монтування та демонтування файлової системи

Файлове дерево формується з окремих частин, які називають *файловими системами*, кожна з яких містить кореневий каталог і список його підкаталогів та файлів. Більшість файлових систем є розділами диска або логічними томами з пам'яттю на дисках. Файлову систему необхідно змонтувати до того, як вона стане видимою для процесів. *Точкою монтування для файлової системи може бути будь-який каталог, але файли і підкаталоги, розміщені нижче цієї точки в ієрархії, залишаться недоступними, доки файлова система не буде змонтована у цій точці.*

В основі файлового дерева міститься кореневий каталог, / (слеш). Інші файлові системи монтуються в кореневому каталозі й виникають як піддерева у загальній ієрархії. Для монтування файлової системи привілейований користувач може застосувати таку команду: **$ mount device directory**. Ця команда «прикріплює» файлову систему до пристрою

device у зазначеному каталозі *directory* в ієрархії каталогів – у точці монтування цієї файлової системи.

Щоб **вивести список змонтованих у певний момент файлових систем**, можна використовувати команду ***mount*** без аргументів.

Частину структури каталогів і файлів для системи показано на рис. 3.1, зокрема вказано точки монтування щодо ієрархії каталогу.

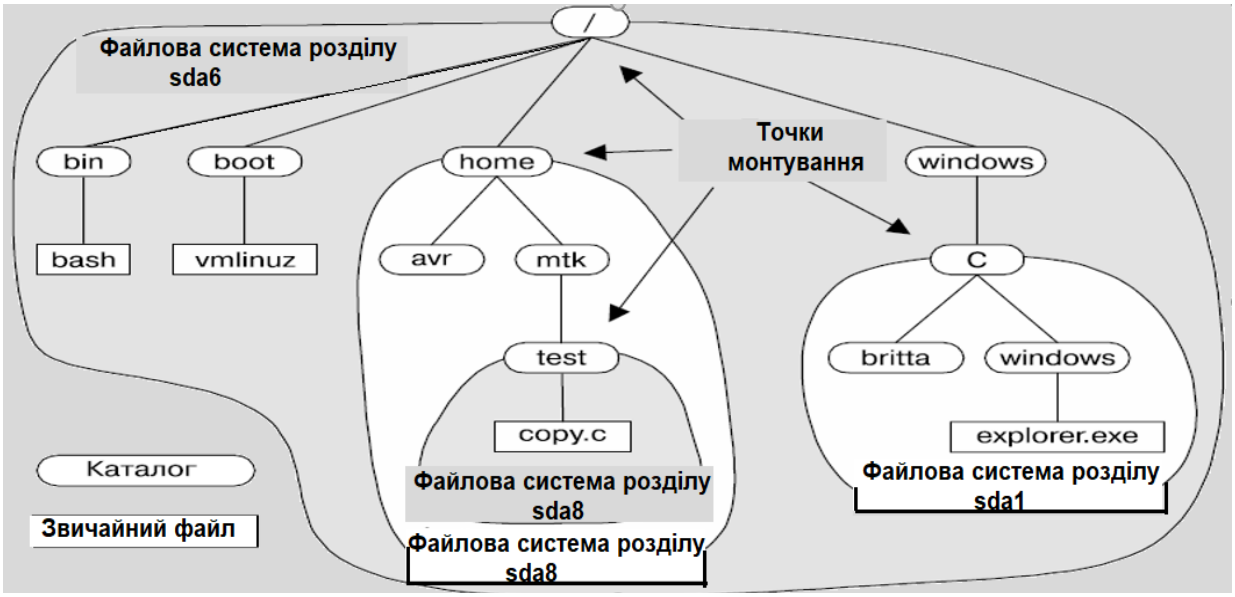


Рис. 3.1. Приклад ієрархії каталогу –
показано точки монтування файлової системи [1]

Список змонтованих файлових систем зберігається у файлі ***/etc/fstab***. Завдяки цьому можливі автоматична **перевірка цілісності файлової системи** за допомогою команди **fsck** й монтування файлових систем на етапі початкового завантаження, а також виконання скорочених команд на кшталт mount /var/spool.

Інформація, що міститься у цьому файлі, відображає розміщення файлових систем на диску.

Файлові системи демонтуються командою ***umount***. У ній не має бути ані відкритих файлів, ані виконуваних процесів з їх поточними каталогами, ані запущених виконуваних програм. Приклад розмонтування раніше підмонтованої файлової системи у каталозі */mnt*: **# umount /mnt**

Linux постійно зберігає список файлових систем та їх параметрів у таблиці **/etc/fstab**. Загальну структуру каталогів зображено на рис. 3.2. Основним каталогом файлової системи ОС Linux є кореневий каталог.

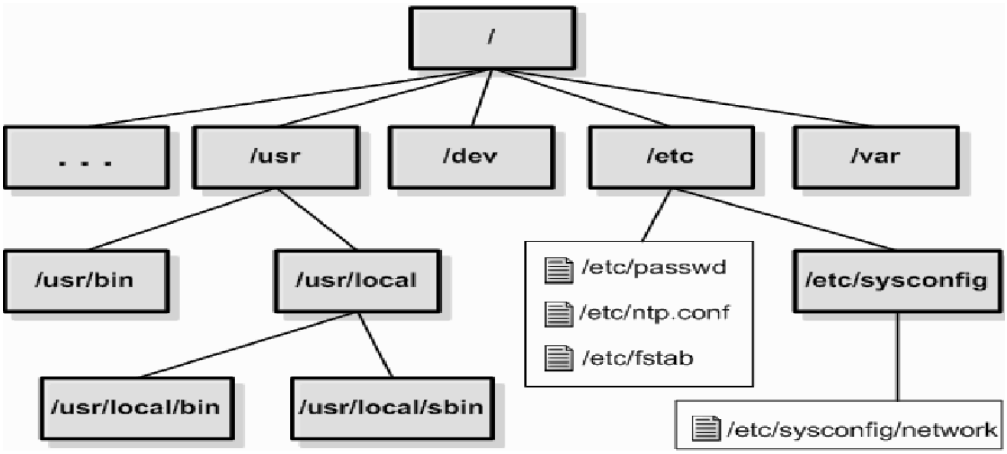


Рис. 3.2. Структура каталогів ОС Linux [1]

Відповідно до стандарту FHS у кореневому каталозі ***root*** передбачено такі стандартні підкаталоги Linux:

/bin/ (від *binaries* – двійкові файли) – у цьому каталозі зберігаються основні команди (утиліти), необхідні користувачу для роботи в ОС (командний інтерпретатор *bash*, основні команди, такі як *cp, mv, rm, ls, cat)*.

/boot/ – у каталозі (завантаження) зберігаються файли, необхідні для завантаження ОС, такі як завантажувач *grub*, образ пам’яті, модулі ядра, модуль тестування пам’яті тощо. Це незмінні файли, потрібні для завантаження системи.

/sbin/ – каталог, призначений для зберігання двійкових файлів, які слугують для керування і системного адміністрування ОС (програми *fdisk, fschk, ifconfig, mkfs, и init*). У цьому каталозі зберігаються основні програми, які виконуються користувачем root а також програми, що виконуються у процесі завантаження. Цей каталог різниться від каталогу /bin/ тим, що в останньому зберігаються програми, не призначені для виконання адміністративних завдань.

/dev/ (від англ. *device* – пристрій) – розміщені основні файли пристроїв, які є в системі (наприклад, жорсткі диски (вінчестери) з інтерфейсом *sata /dev/sda* (*файл sda*), відеокамери або TV-тюнери */dev/video* (*файл video*), послідовні порти, сканери).

/etc/ (від лат. *et cetera* – й інші) – це каталог, що містить загальносистемні конфігураційні файли, файли налаштувань – від конфігураційних файлів системи X Window, бази даних користувачів й до стартових сценаріїв, менеджерів пакетів.

/lib та /lib64 – це системні 32- та 64-бітні бібліотеки, потрібні для запуску двійкових файлів, що зберігаються в каталогах **/bin** та **/sbin**. У каталозі **/lib** зберігаються модулі ядра ОС.

/mnt/ – каталог, призначений для ручного підключення пристроїв. Його використовують як тимчасову точку монтування для жорстких дисків або пристроїв, якихі відключають.

/proc/ – спеціальний каталог, що містить псевдофайловую систему *procfs*, яку використовують в ОС Linux для організації взаємодії запущених процесів із простором ядра ОС. Систему називають псевдофайловою, оскільки на жорсткому диску немає такої файлової системи, як procfs, тому що вона розміщена у віртуальній пам’яті й монтується під час роботи ОС. У цьому каталозі зберігаються файли, в яких міститься інформація про систему і процеси. Також тут є ряд підкаталогів, названих відповідно до ідентифікаторів процесів (PID), які містять інформацію по кожному запущеному процесу. Так, у файлі */proc/modules* зберігається інформація про завантажені модулі ядра, а у файлі */proc/cpuinfo* – інформація про процесор комп’ютера користувача. У файлі */proc/uptime* зберігається час, протягом якого система перебуває в робочому стані.

/mnt – у цей каталог монтують файлові системи зовнішніх пристроїв – CD/DVD-дисків, USB/SSD-накопичувачів.

/opt (від англ. *optional* – опціональний) – у цьому каталозі розміщено додаткові пакети програм, встановлені користувачем в ОС. Найчастіше пакети ПЗ надані сторонніми розробниками, а не розробниками дистрибутиву. Підкаталоги у цьому каталозі мають власну структуру, і часто щоб видалити встановлений пакет, досить просто видалити однойменний каталог з /opt/.

/var/ – в цьому каталозі зберігаються системні лог-файли, кеш-файли і файли-замки програм. Це каталог для даних, які часто змінюються.

/home/ – містить усі домашні каталоги користувачів системи, тобто всі реальні дані системи; якщо в ОС користувача заведено тільки один обліковий запис, то в цьому каталозі буде щонайменше два каталоги: *lost+found* (для зберігання файлів, на які немає посилань) й каталог з іменем профілю користувача, наприклад *sergii*. У цьому каталозі, крім особистих файлів і даних користувача, зберігаються файли конфігурації його профілю. Наприклад, у разі збою роботи живлення ушкоджені й відновлені програмою *fsck* файли будуть поміщені у каталог /lost+found/.

/root/ – каталог, призначений для користувача з іменем root. Цей каталог спеціально був поміщений у кореневий розділ файлової системи, а не в каталог /home.

/usr – вторинна ієрархія для даних користувача, містить бінарні й конфігураційні файли програм, встановлених з початкових кодів, сторінки керівництва *man*, а також усі інші файли програм, встановлених у системі. Саме у цей каталог найчастіше встановлюють програми.

/media – точки монтування для змінних носіїв, таких як DVD-ROM, *flash*-дисків.

/tmp – тимчасовий каталог, в якому містяться тимчасові файли, створювані різними додатками у процесі роботи системи.

Під час встановлення Ubuntu Linux вимагає створення щонайменше двох розділів: один для самої ОС – позначаений «/», має назву «*root*»

(кореневий розділ), а другий для віртуальної пам’яті (для файлів підкачкки) – має назву «*swap*». Є ще третій розділ – Home, створюється за бажанням, на ньому будуть зберігатися основні налаштування додатків і файли користувача.

Файли розрізняються як за структурою, так і за своїм призначенням.

В ОС Linux визначено ***сім типів файлів***:

– *звичайні або регулярні файли (-)*;

– *каталоги (d)*;

– *файли байт-орієнтованих (символьних) пристроїв*;

– *файли блочно-орієнтованих (блокових) пристроїв*;

– *локальні сокети*;

– *іменовані канали (реалізують принцип обслуговування FIFO (First in First Out, тобто «першим надійшов – першим і обслуговується»)*;

– *символічні посилання.*

Якщо розробники додають у файлову систему що-небудь нове і незвичайне (наприклад, інформацію про процеси в каталог /proc), їм доводиться маскувати це під файли стандартних типів. Визначити тип файлу можна за допомогою команди ***ls -ld***. Це команда перегляду атрибутів файлу (лістинг). Для подання різних типів файлів застосовують коди, наведені у табл. 3.1. Команда ***rm*** є універсальним засобом видалення файлів. Але як видалити файл, ім’я якого, наприклад, *-f*? Для цього треба вказати більш повне ім’я (наприклад, ./-f) або скористатись спеціальним аргументом -, який повідомить команді *rm* про те, що інша частина командного рядка являє собою ім’я файлу, а не список аргументів: **rm -- -f**.

Приклад використання команди **ls**: **$ *ls -ld* /usr/include**

Можна видалити файл за шаблоном з використанням опції *-i*, щоб команда *rm* вимагала підтвердити видалення кожного файлу: **$ rm -i foo***.

Звичайний файл – це послідовність байтів. До таких файлів належать бінарні файли, бібліотеки, текстові файли і файли різних додатків. ОС Linux не накладає ніяких обмежень на структуру таких файлів.

Таблиця 3.1

Кодування типів файлів в лістингу команди ls

Тип файлу	**Символ**	**Створюється командою**	**Видаляється командою**
Звичайний файл	**-**	Редактори, *cp* і ін.	*rm*
Каталог	**d**	*mkdir*	*rmdir, rm -r*
Файл символьного пристрою	**c**	*mknod*	*rm*
Файл блочного пристрою	**b**	*mknod*	*rm*
Локальний сокет	**s**	*socket (2)*	*Rm*
Іменований канал	**p**	*mknod*	*rm*
Символічне посилання	***l***	*ln -s*	*rm*

Каталоги – файли цього типу зберігають посилання як на файли, так і на інші каталоги. Каталог **створюють** командою ***mkdir*** та **видаляють** (за умови, що він порожній) командою ***rmdir***. **Непусті каталоги можна видалити** командою ***rm -r***. Кожний каталог містить щонайменше два записи: «**.**» (крапка), яка являє собою посилання на сам каталог, і «**..**» (крапка-крапка), яка є посиланням на його батьківський каталог – той, що розміщений над ним в ієрархії. Кожний каталог, за винятком кореневого, має свій батьківський каталог. Для кореневого каталогу запис «..» є посиланням на нього самого (таким чином, позначення «/..» – те ж саме, що і «/»).

Файл насправді зберігається у батьківському каталозі, а не в самому файлі. На файл можна посилатися з декількох каталогів одночасно і навіть з кількох елементів одного і того самого каталогу, причому у всіх посилань можуть бути різні імена. Це створює ілюзію того, що файл одночасно є в різних каталогах.

В ОС Linux розрізняють *жорсткі* та *символьні посилання*, кожне з яких має особливе значення.

Жорстке посилання – це додаткове ім'я файлу, яке вказує безпосередньо на індексний дескриптор файлу.

Індексний дескриптор файлу (ІНД) – це структура даних, яка унікально характеризує кожний файл у межах однієї файлової системи, в якій зберігається інформація про файл, а саме – приналежність власнику (користувачу і групі), режим доступу (читання, запис, запуск на виконання) і тип файлу. Індексний дескриптор файлу має унікальний ідентифікатор у файловій системі. Файлова система підраховує кількість посилань на кожний файл і в разі видалення файлу не звільняє блоки даних доти, поки не буде видалене останнє посилання на нього. Посилання не можуть вказувати на файл, що перебуває в іншій файловій системі.

Жорсткі посилання створюють командою ***ln*** і видаляють командою *rm*. Синтаксис команди *ln* легко запам'ятати, оскільки вона є «дзеркальним відображенням» команди ***cp***.

Команда **cp oldfile newfile** створює копію файлу *oldfile* з іменем *newfile*, а команда **ln newfile oldfile** перетворює ім'я *newfile* у додаткове посилання на файл *oldfile*.

Символічне посилання – це спеціальний файл, для якого у файловій системі не зберігається ніякої інформації, крім *текстового рядка*, що містить *шлях до файлу*, який має бути відкритий у разі звернення за цим посиланням.

Символічне або «*м'яке*» посилання дозволяє замість імені файлу вказувати його *псевдонім*. Натрапивши під час пошуку файлу із символічним посиланням, ядро витягує ім'я, яке в ньому зберігається.

Відмінність між жорсткими і символічними посиланнями полягає в тому, що жорстке посилання є прямим, тобто вказує безпосередньо на індексний дескриптор файлу, тоді як символічне посилання вказує на файл

із назвою. Файл, що адресується символічним посиланням, і саме посилання – це різні об'єкти файлової системи.

Символічні посилання створюють командою ***ln -s*** й видаляють командою *rm*. Вони можуть мати будь-яке ім'я, тобто дозволяється вказувати на файли, що зберігаються в інших файлових системах, і навіть на неіснуючі файли. Іноді кілька символічних посилань утворюють петлю. Символічне посилання може зберігати як повне, так і скорочене ім'я файлу.

Наприклад, команда

$ sudo ln -s archived/secure /var/log/secure

за допомогою відносного шляху пов'язує ім'я */var/log/secure* з іменем */var/log/archived/secure*. Як видно з наступного результату виконання команди *ls*, отримане символічне посилання буде містити рядок «*archived/secure*».

Файли символьних і блокових пристроїв дозволяють додаткам отримувати доступ до апаратних засобів і периферійного устаткування системи. Драйвери пристроїв утворюють стандартний комунікаційний інтерфейс, який користувач сприймає як сукупність звичайних файлів. Отримавши запит до файлу символьного або блочного пристрою, файлова система передає цей запит відповідному драйверу. *Важливо відрізняти файли пристроїв від драйверів цих пристроїв*. Файли самі по собі не є драйверами – їх можна розглядати як шлюзи, через які драйвер приймає запити.

Файли *символьних пристроїв* не використовують буферизацію у процесі операцій введення – виведення. Вони разом зі своїми драйверами здійснюють власну буферизацію введення – виведення у міру надходження.

До таких пристроїв відносять *віртуальні термінали*, *модеми* та *інші пристрої*, які не підтримують довільного доступу до даних. Файли

блокових пристроїв обробляються драйверами, які здійснюють введення – виведення даних цільними блоками, а буферизацію виконує ядро. *До блокових пристроїв відносяться жорсткі диски, DVD-приводи, модулі пам'яті, а також усі інші пристрої, що підтримують довільний доступ до даних*.

Файли пристроїв мають два номери: старший і молодший. Старший номер пристрою дозволяє ядру визначити, до якого драйвера належить файл, а молодший номер зазвичай ідентифікує конкретний фізичний пристрій.

Локальні сокети – це спеціальний тип файлу, який використовують процеси для взаємодії один з одним – для передачі даних від одного процесу до іншого, тобто здійснюють обмін даними.

Локальні сокети створюють за допомогою системного виклику ***socket***. Коли з обох сторін з'єднання закрито, сокет можна видалити командою *rm* або за допомогою системного виклику **unlink**.

Іменовані канали (FIFO, First In, First Out – перший прийшов – першим пішов) – файли цього типу використовують для взаємодії між процесами, проте, на відміну від сокетів, в іменованих каналах дані передають тільки в одному напрямку. Їх створюють командою ***mknod***, а видаляють командою *rm*. Іменовані канали і локальні сокети мають однакове призначення, а їх обопільне існування склалося історично.

Якби системи UNIX та Linux розроблялися у наші дні, то про ці засоби взаємодії питання б не стояло – їх би замінили мережеві сокети.

3.1.4. Атрибути файлів, біти режиму

У файловій системі FAT є атрибути файлу (*A- archive, S-system, H-hidden, R-read-only*). Для файлових систем ext2fs, ext3, ext4 і частково для інших файлових систем Linux атрибути різняться.

Є дві команди для керування атрибутами файлу: *lsattr* та *chattr*.

Команда ***lsattr*** слугує для перегляду атрибутів (*lsattr file1.txt*), команда **chattr** слугує для зміни атрибутів («+» – додає, «-» – видаляє). Атрибути можуть бути встановлені тільки для каталогів і звичайних файлів. Доступними є такі атрибути:

A (**no Access time**): якщо для файлу або каталогу встановлено такий атрибут, то в разі звернення до цього файлу (для читання або запису) у файлі не буде модифікуватися час останнього звернення, тобто не відбудеться *ніяких оновлень*. Це може бути корисно, наприклад, у роботі з файлами і каталогами, до яких дуже часто звертаються і відкривають їх для читання. Це може дещо розвантажити систему, оскільки час останнього доступу – це єдиний параметр в *inode*, який змінюється в разі відкриття файлу в режимі *read-only* (тільки для читання).

a (append only): якщо для файлу або каталогу встановлено такий атрибут, а також цей файл (каталог) допускає запис, то єдина доступна операція запису – це *тільки додавання в кінець* файлу (*append*). Для каталогу це означає, що користувач може лише додавати файли, але не може перейменовувати або видаляти будь-який наявний файл всередині цього каталогу. Тільки root може встановлювати або очищати цей атрибут.

d (no dump): **dump (8)**: це стандартна для UNIX утиліта резервного копіювання. Вона створює резервну копію будь-якої файлової системи, в якій у файлі */etc/fstab* значення *dump counter* встановлено в **1**, *тобто це без дампу*. Якщо файл або каталог має цей атрибут, то він буде проігнорований під час створення резервної копії файлової системи. У разі установки атрибута для каталогу під дію атрибута рекурсивно потрапляють усі файли і каталоги, підпорядковані нижче.

i (immutable): файл або каталог з таким атрибутом не може змінюватися взагалі: його не можна перейменувати, на нього не може бути створене посилання, його не можна видалити. Тільки root може

встановлювати або очищати цей атрибут. Цей атрибут не можна використовувати спільно з атрибутами *A*.

s (secure deletion): після видалення файлу, позначеного таким атрибутом, місце на диску, яке займав файл, буде заповнене нулями.

S (Synchronous mode): якщо встановлено такий атрибут, то всі зміни у файл будуть записані негайно, тобто цей атрибут знімає буферизацію запису для цього файлу.

3.1.5. Інформація про файл. Системний виклик stat()

Системні виклики *stat()*, *lstat()* та *fstat()* отримують інформацію про файл переважно з індексного дескриптора файлу [10]:

```
#include <sys/stat.h>
int stat(const char *pathname, struct stat *statbuf);
int lstat(const char *pathname, struct stat *statbuf);
int fstat(int fd, struct stat *statbuf);
```

Усі виклики повертають 0 в разі успішного завершення й -1 у разі помилки.

Три ці системні виклики розрізняються тільки способом зазначення файлу:

– виклик *stat()* повертає інформацію про іменований файл;

– виклик *lstat()* подібний до виклику stat(), але якщо іменований файл є символічним посиланням, то повертається інформація про саме посилання, а не про файл, на який воно вказує;

– виклик *fstat()* повертає інформацію про файл, до якого звертається відкритий дескриптор.

Усі ці виклики повертають у буфер структуру *stat*, на яку вказує змінна *statbuf*.

Форма цієї структури така:

```
struct stat {
    dev_t     st_dev;  /* ідентифікатор пристрою, на якому міститься файл */
    ino_t     st_ino;   /* номер індексного дескриптора файлу */
    mode_t    st_mode;  /* тип файлу і права доступу */
    nlink_t   st_nlink;   /* кількість (жорстких) посилань на файл */
    uid_t     st_uid;    /* ID користувача – власника файлу */
    gid_t     st_gid;    /* ID група власника файлу */
    dev_t     st_rdev;   /* ідентифікатори файлів пристроїв */
    off_t     st_size;    /* загальний розмір файлу (у байтах) */
    blksize_t   st_blksize; / * оптимальний розмір блоку для «введення – виведення» (у байтах) */
    blkcnt_t    st_blocks;   /* кількість відведених блоків (по 512 байт) */
    time_t      st_atime;   /* час останнього доступу до файлу */
    time_t      st_mtime;    /* час останньої зміни файлу */
    time_t      st_ctime;     /* час останньої зміни статусу */
};
```

Різні типи даних (перша колонка), які використовуються для подання полів у структурі *stat*, визначені у стандарті SUSv3. Друга колонка – це поле.

Загальна форма оголошення структури така:

```
struct тип_структури
{
    тип ім’яЕлемента1;
    тип ім’яЕлемента2;
     . . .
    тип ім’яЕлементаN;
};
```

Поле *st_dev* ідентифікує пристрій, на якому міститься файл. Поле *st_ino* містить індексний дескриптор цього файлу. Комбінація значень *st_dev* та *st_ino* унікальним чином ідентифікує файл в усіх файлових системах. У типі *dev_t* записані старший і молодший номери пристрою.

Поля *st_uid* та *st_gid* ідентифікують відповідно ID користувача та ID групу, яким належить файл. Поле *st_nlink* – це кількість (жорстких) посилань на файл.

Поле ***st_mode*** – бітова маска, яка складається з 16 бітів та визначає тип файлу і права доступу до нього.

Перегляд атрибутів (значень цих бітів) здійснюють за допомогою команди ***ls -l*** (або ***ls -ld*** для каталогу).

Для керування атрибутами файлів використовують команду **chattr**.

Chattr (*Change Attribute*) – команда в Linux, яка змінює атрибути файлів. Спочатку застосовувалася тільки в ext2, ext3, ext4, але нині її можна використовувати і в інших файлових системах, наприклад XFS, ReiserFS, Btrfs та інших. Синтаксис команди:

chattr [-RV] [-v версія] [атрибути] файл/директорія

Опції:

-R – рекурсивна зміна атрибутів каталогів та їхнього вмісту, ігнорування символічних посилань під час рекурсії;

-v version – виведення версії та номера зборки файлової системи;

-V – виведення версії програми;

-d – відображення даних по всіх атрибутах для всіх каталогів;

-a – виведення атрибутів для всіх файлів.

Атрибути:

a – дозволяється додавати інформацію у файл або каталог, але змінювати або видаляти її не можна;

c – автоматичне стиснення файлу під час запису на диск;

D – синхронне оновлення каталогів;

d – якщо запущена утиліта *dump*, не виконується бекап для файлу;

i – файлу присвоюється статус незмінного;

j – усі дані файлу перед записом зберігаються в системний журнал;

s – обнуління блоків файлу;

S – синхронне оновлення;

u – вміст файлу зберігається навіть в разі його видалення.

Приклад 3.1. Присвоєння файлу або каталогу статус незмінного. Користувач хоче присвоїти файлу */etc/passwd* статус незмінного:

 # chattr + i /etc/passwd

Тепер внести правки, перезаписати або видалити цей файл не може навіть привілейований користувач. Будь-які маніпуляції із правами файлу можна робити тільки після відключення атрибута. Щоб перевірити, чи встановився атрибут, використовуємо команду *lsattr*: *# lsattr /etc/passwd*

Щоб видалити цей атрибут, пишемо команду *# chattr -i /etc/passwd*

Перевіряємо, чи справді атрибут скасовано, знову прописуючи команду

 # lsattr /etc/passwd

3.1.6. Перенаправлення «введення – виведення»

Стандартні потоки введення і виведення в Linux є одним з найбільш поширених засобів для обміну інформацією процесів, а перенаправлення >, >> та | є однією з найбільш популярних конструкцій командного інтерпретатора.

Стандартне введення користувач в терміналі здійснює через клавіатуру.

Стандартне виведення і стандартна помилка відображаються на дисплеї термінала користувача у вигляді тексту.

Введення і виведення розподіляються між трьома стандартними потоками [11, 13]:

stdin – стандартне введення (клавіатура),

stdout – стандартне виведення (екран),

stderr – стандартна помилка (виведення помилок на екран).

Потоки також пронумеровані: *stdin – 0, stdout – 1, stderr – 2.*

Зі стандартного введення команда може тільки зчитувати дані, а два інші потоки можна використовувати тільки для запису. Дані виводяться на

екран і зчитуються з клавіатури, оскільки стандартні потоки за замовчуванням асоційовані з терміналом користувача. Потоки можна підключати до чого завгодно: до файлів, програм і навіть до пристроїв. У командному інтерпретаторі *bash* таку операцію називають перенаправленням.

Розглянемо *часто використовувані оператори перенаправлення* та їх *позначення.*

Команди зі знаками > або < означають перезапис вмісту файлу:

> – стандартне виведення;

< – стандартне введення;

2> – стандартна помилка.

Команди зі знаками >> або << також не перезаписують вміст файлу, а приєднують дані до нього:

>> – переадресація стандартного виведення у файл;

<< – переадресація стандартного введення у файл;

2 >> – стандартна помилка.

А також:

| – програмний канал – стандартне виведення одного процесу є стандартним введенням іншого;

> – створює новий файл, який містить *стандартне виведення*. Якщо файла немає, він буде створений, якщо він є – буде перезаписаний зверху наявного;

< – відправляє вміст заданого файлу для використання як стандартного *введення*;

>> – направити стандартний *потік виведення* у файл. Якщо файлу немає, він буде створений, якщо він є – дані будуть дописані до нього у кінці;

<< – приймає текст як стандартне *введення*, починаючи з наступного рядка;

2> – створює новий файл, який містить стандартне виведення *помилок*. Якщо файлу немає, він буде створений. Якщо заданий файл вже є, він перезаписується;

2>> – додає стандартне виведення помилок у файл. Якщо заданого файлу немає, він створюється, якщо він є – дані будуть дописані до нього у кінці;

&>file або *> & file* – створює стандартний потік виведення і стандартний потік помилок, направляє у файл. Якщо заданий файл вже існує, він перезаписується.

Інша форма запису: *>file 2>&1*. Наприклад, команда *cat > note.txt* виводить на екран вміст файлу note.txt:

```
pl@comp:~$ cat > note.txt
Надо переименовать turbo
на younglinux
```

Якщо додати дату в цей самий файл, *date >> note.txt*, то вміст файлу буде *cat note.txt*:

```
pl@comp:~$ date >> note.txt
pl@comp:~$ cat note.txt
Надо переименовать turbo
на younglinux
Пт янв 22 13:25:31 MSK 2021
```

Канали «|» використовують для перенаправлення потоку з однієї програми в іншу. Стандартне виведення даних після виконання однієї команди перенаправляється в іншу через канал. Дані першої програми, які отримує друга програма, система не показує. На дисплей термінала будуть виведені тільки відфільтровані дані, які повертаються іншою командою. Наприклад, щоб зберегти імена файлів, що містять рядок «LOG», використовують таку команду: *dir /catalog | find "LOG" > loglist*. Виведення команди *dir* відсилається в команду-фільтр *find*. Імена файлів, що містять рядок «LOG», зберігаються у файлі *loglist* у вигляді списку (наприклад, *Config.log, Logdat.svd і Mylog.bat*).

3.1.7. Стандартне «введення – виведення»

Стандартний вхідний потік зазвичай передає дані від користувача до програми. Програми, які передбачають стандартне введення, зазвичай отримують вхідні дані від пристрою типу *клавіатура*. Стандартне введення припиняється після досягнення EOF (кінець файлу *end off),* який вказує на те, що даних для читання більше немає. EOF вводять натисканням комбінації клавіш Ctrl+D. Розглянемо роботу зі стандартним виведенням на прикладі команди *cat* (від CONCATENATE, у перекладі «зв'язати» або «об'єднати щось»).

Cat зазвичай використовують для об'єднання вмісту двох файлів. *Cat* надсилає отримані вхідні дані на дисплей термінала як стандартне виведення і зупиняється після того як отримує EOF.

Команда об'єднує три файли: file1, file2 та file3 в один файл bigfile:

cat file1 file2 file3> bigfile

Команда *cat* по черзі виводить вміст файлів, перерахованих як параметри на стандартний потік виведення. Стандартний потік виведення перенаправлений у файл *bigfile*.

3.1.8. Файлові менеджери

Файловий менеджер (*File manager*) – програма, призначена для роботи з файлами і папками в ОС.

Найбільш популярними операціями файлового менеджера є такі:

– перегляд файлової структури;

– створення файлів і папок;

– переміщення файлів і папок;

– перейменування файлів і папок;

– копіювання файлів і папок;

– керування файлами і папками (зміна властивостей, призначення прав);

– пошук файлів і папок.

Кожний дистрибутив Linux має файловий менеджер за замовчуванням, як, наприклад, у Windows є всім відомий файловий менеджер «Проводник».

В Linux є декілька файлових менеджерів, які розробники дистрибутивів найчастіше використовують як основний у своєму дистрибутиві. Найбільш популярними файловими менеджерами для *Ubuntu* є такі.

Nautilus

Цей файловий менеджер використовують за замовчуванням в *Ubuntu* та інших дистрибутивах із графічною оболонкою *GNOME* (рис. 3.3). Він функціональний, має простий і зручний графічний інтерфейс користувача для зручної навігації та керування файлами в системі Linux.

Домашня сторінка Nautilus: *https://wiki.gnome.org/Apps/Nautilus/*

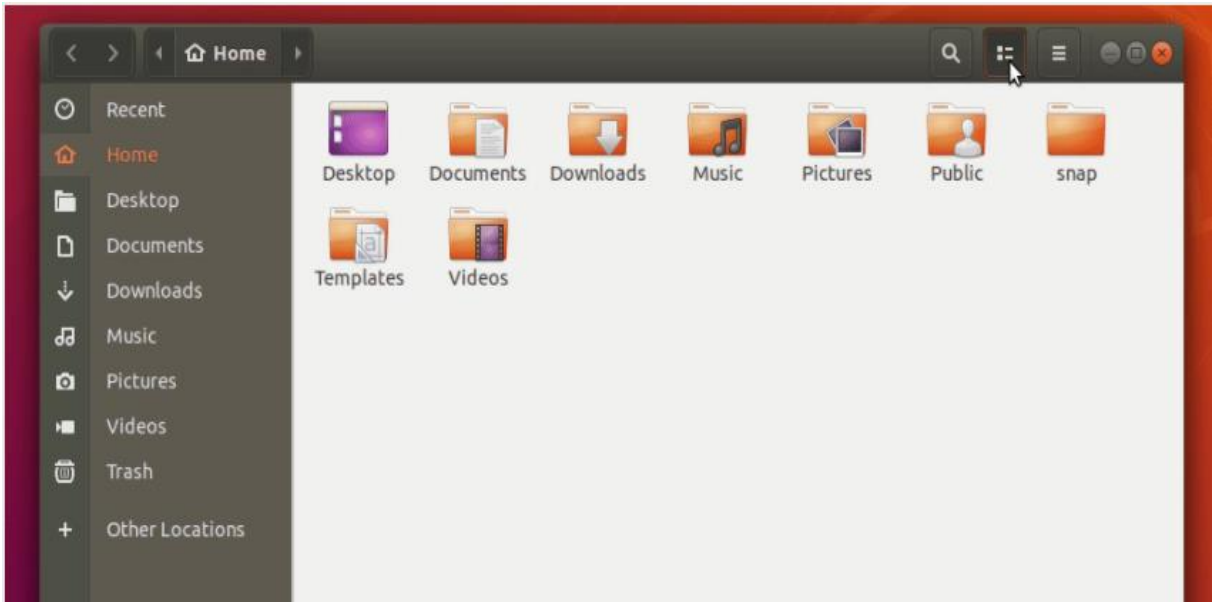


Рис. 3.3. Графічний інтерфейс файлового менеджера Nautilus

Midnight Commander (MC)

Текстовий файловий менеджер з інтерфейсом командного рядка (*cli*). Він має двопанельний інтерфейс і вбудовану командну оболонку. Також є вбудований редактор з підсвічуванням синтаксису і переглядач, що

підтримує двійкові файли. Програма підтримує віртуальну файлову систему (VFS), що дозволяє працювати з файлами на віддалених машинах. В Debian, Ubuntu можна встановити MC за допомогою команди *apt-get*:

$ sudo apt-get install mc

MC має багато корисних як для користувачів, так і для адміністраторів функцій: *копіювання*, *видалення*, *перейменування / переміщення*, *створення директорії*.

Після завершення інсталяції треба ввести «mc» (без лапок) у консолі для його запуску: **# mc**.

Можливості Midnight Commander

Інтерфейс MC розділений на дві колонки, незалежні одна від одної (рис. 3.4). Кожна колонка є активною директорією. Можна переміщатися між ними за допомогою клавіші *Tab*. У нижній частині екрана є кнопки з номерами, які належать до функціональних клавіш F1 – F10.

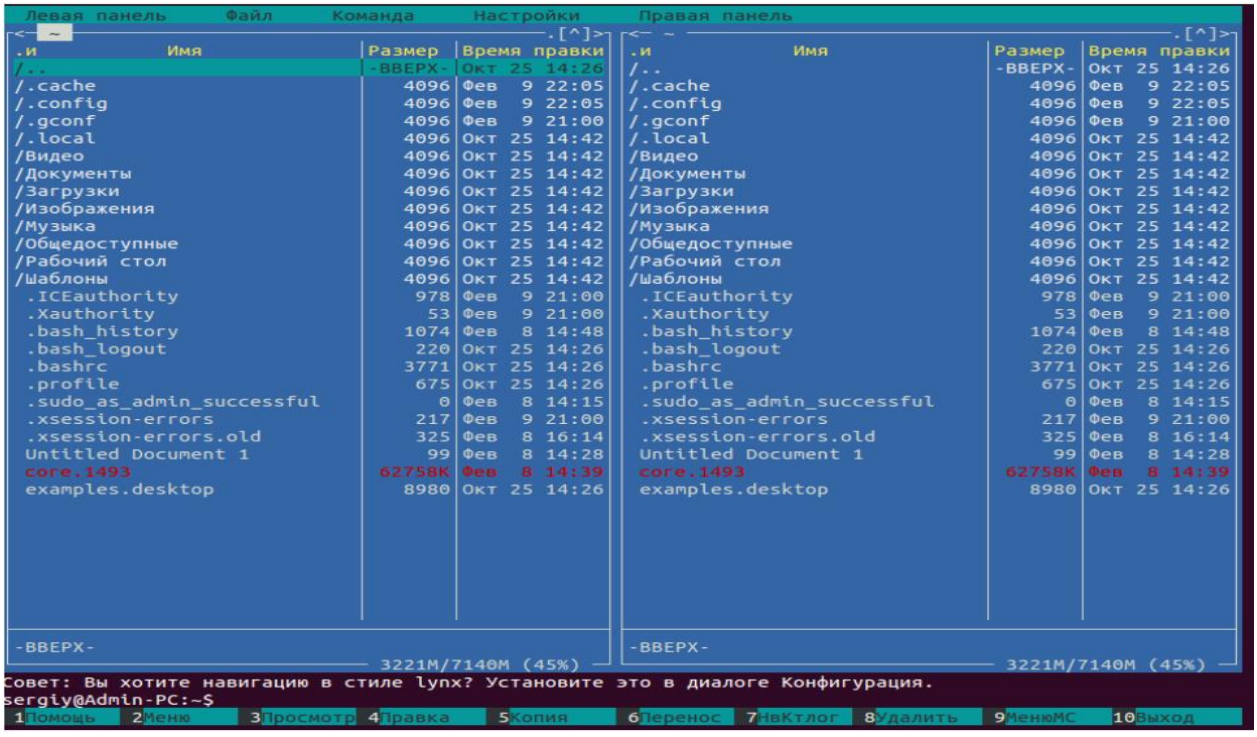


Рис. 3.4. Інтерфейс файлового менеджера *Midnight Commander*

Щоб скопіювати файл з однієї директорії в іншу, треба її виділити й натиснути «F5». Для копіювання декількох файлів одночасно необхідно спочатку виділити їх за допомогою клавіші «*Insert*». Є вбудована утиліта для перегляду файлів. У командному режимі є декілька текстових редакторів, таких як ***vi, joe*** і ***nano***. MC має власну вбудовану програму для перегляду вмісту текстових файлів, щоб її запустити, слід виділити файл і натиснути «F3».

Якщо треба редагувати файл, слід його виділити і натиснути «F4».

Права доступу до файлів

Файли і директорії мають права доступу, які вказують, хто може читати, записувати і виконувати файли та директорії.

Команда для керування правами доступу – ***chmod***.

Щоб дізнатися, як нею користуватися, слід набрати в терміналі «*man chmod*».

У MC треба тільки виділити файл, потім натиснути «F9»> File> Chmod або «Ctrl-x» і «c» – програма покаже поточні права доступу виділеного файлу і параметри, які можна змінити.

Власник файлу

Файли і директорії мають свого власника, а також групу власника. Привілеями власника можна керувати за допомогою описаної вище команди *chmod*. Керування власниками здійснюють за допомогою команди *chown*. Інструкцію щодо користування цією командою можна отримати, набравши у терміналі «*man chmod*».

Щоб задати власника файлу і групу власника, в MC потрібно виділити файл, потім натиснути «F9»> File> Chown, або «Ctrl-x» та «o». Тепер можна задати власника і групу власника зі списку доступних користувачів і груп. У MC також є функція «*Advanced Chown*», яка являє собою комбінацію *chmod* та *chown*, що дозволяє виконувати два описані вище

завдання в одному місці: натиснути «F9»> File> Advanced Chown. Щоб вийти з *Midnight Command*, треба натиснути «F9»> File> Exit або «F10».

Більш детально ознайомитися з можливостями MC можна переглянувши Midnight Commander FAQ:

https://midnight-commander.org/wiki/doc/faq

Dolphin

Це ще один файловий менеджер з KDE (K Desktop Environment, К-інтегроване робоче середовище). KDE – робоче середовище для усіх видів Unix: Linux, Solaris, FreeBSD, IRIX, HP-UX. Це вільне ПЗ згідно зі Стандартною публічною ліцензією GNU. Інтерфейс *Dolphin* є функціональним (рис. 3.5). Є можливість створити кілька панелей і вкладок. Також є бічна панель з ярликами швидкого доступу до файлової системи.

Встановлюється *Dolphin* командою ***sudo apt install dolphin***.

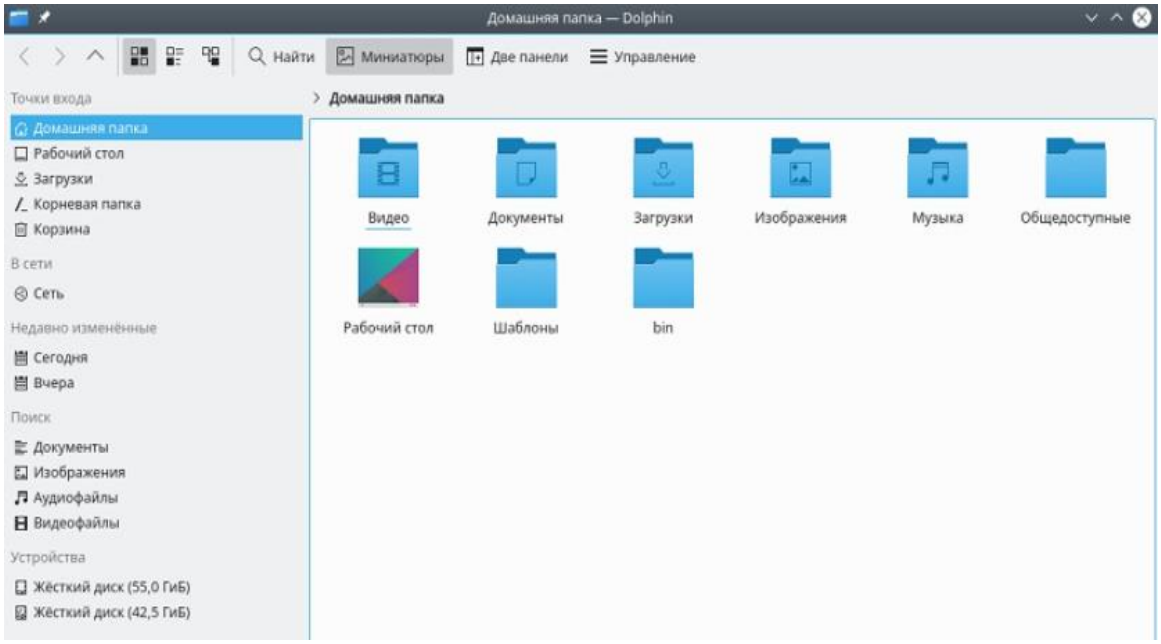


Рис. 3.5. Графічний інтерфейс файлового менеджера *Dolphin*

3.2. Робота з файловою системою

3.2.1. Командні оболонки ОС Unix

Усі сучасні ОС сімейства Unix забезпечують підтримку трьох типів інтерфейсів користувача:

– інтерфейс графічної оболонки (X Window);

– інтерфейс командної оболонки (Shell);

– інтерфейс прикладного програмування (API).

Графічна оболонка реалізує модель робочого стола (меню встановлених системних і прикладних програм, панелі керування і відображення станів ОС та її компонентів, віконні та іконографічні інтерфейси процесів ОС тощо) і призначена, головним чином, для кінцевих користувачів ОС.

Командна оболонка виконує такі основні функції:

– організує діалог з користувачем через введення команд;

– є інтерпретатором рядків команд, що складаються з команд ОС, системних утиліт і вбудованих в оболонку програмних каналів обміну даними;

– виконує внутрішні команди;

– запускає зовнішні програми;

– виконує командні файли;

– має власну мову сценаріїв, яка істотно розширює її функціональні можливості (зокрема, під час вирішення завдань адміністрування ОС).

Інтерфейс прикладного програмування (API) забезпечує доступ додатків, написаних на алгоритмічних мовах високого рівня, до бібліотеки системних викликів ядра ОС.

У системах Unix використано різні командні оболонки (*command shells*), які називають також командними процесорами або інтерпретаторами команд (*CLI – Command Language Interpreter*). *Командна оболонка забезпечує взаємодію між користувачем та ядром системи*.

З-поміж командних оболонок найбільш відомі й поширені такі:

• *sh (Bourne shell) /bin/sh – оболонка Борна, класична оболонка Unix* (не дуже зручна в роботі);

• *csh (C-shell) /bin/csh – оболонка С, в якій використано синтаксис мови С* (зручніша порівняно з sh, але несумісна з нею по командній мові);

• *ksh (Korn shell) /bin/ksh – оболонка Корна* (містить потужну командну мову, засновану на мові sh, та розвинені засоби інтерактивної роботи);

• *bash (Bourne-Again Shell)* /bin/bash – оболонка «Борна», найбільш поширена оболонка сімейства Unix (зручна для інтерактивної роботи, створена на основі *sh* і багато в чому з нею сумісна);

• *Z shell, zsh /bin/zsh* – одна із сучасних командних оболонок UNIX, яку використовують безпосередньо як інтерактивну оболонку або як скриптовий інтерпретатор. *Zsh* є розширеним *bourne shell* з великою кількістю розширених можливостей підстановки команд, автодоповнення та налаштувань.

Усі команді оболонки, встановлені в ОС Linux, прописані у файлі */etc/shells.*

У наш час в ОС сімейства Unix визнано стандартами командну оболонку BASH (Bourne-Again Shell) та інтерфейс прикладного програмування POSIX (Portable Operating System for Unix) [1, 14, 16]. Тип оболонки, як правило, можна визначити за останнім символом запрошення:

– знак долара «$» вказує на sh-сумісну оболонку (*sh, bash, ksh*);

– знак амперсанта «&» відповідає оболонці csh.

Між тим у привілейованого користувача незалежно від командного процесора, який використовується, останнім символом запрошення зазвичай буває знак решітки «#».

У графічній оболонці програма «Термінал», яка запускається комбінацією клавіш «Ctrl+Alt+T» або ярлик із загального меню програм  – це важливий елемент ОС, який дозволяє запускати програми, створювати папки, копіювати і видаляти файли, встановлювати додатки і под. Системну утиліту, в яку передають ці команди, називають *Shell*, або командною оболонкою. За замовчуванням в Ubuntu використовують командну оболонку *Bash*. Файли ***.bashrc***, ***.bash_profile*** та ***.profile*** – це файли налаштування користувацької оболонки. Вони складаються з команд *bash* і виконуються перед тим як запустити оболонку або завантажити систему. Відмінність між цими файлами полягає в тому, коли вони виконуються: ***.profile*** та ***.bash_profile*** виконуються один раз під час входу користувача в систему (логін користувача), а файл ***.bashrc*** виконується кожний раз, коли користувач відкриває нове вікно термінала (аналог автозавантаження у Windows).

3.2.2. Користувачі і групи. Права доступу

Авторизація в Linux забезпечується за допомогою системи користувачів і груп.

Кожному користувачеві присвоюється унікальне позитивне ціле число, а саме *ідентифікатор користувача* (***UID*** – ***user-ID***). У будь-якій ОС сімейства Linux завжди є *привілейований обліковий запис* ***root***. Його часто називають привілейованим користувачем *root*, із цим іменем здійснюють вхід у систему і він має значення **ID=0**. Цей обліковий запис має необмежені привілеї в ОС – максимальні повноваження, зокрема й на прості операції типу видалення системних файлів і зміни параметрів системи. Кожний користувач може належати до однієї або декількох груп, зокрема до *первинної групи*, або це група входу в систему, яка вказується в */etc/passwd*, а також до кількох додаткових груп, які перераховано в */etc/group*.

Облікові записи користувачів

Інформація про користувачів зберігається у файлі **/etc/passwd**, який доступний для читання всім користувачам та містить таку інформацію:

– ім’я користувача;

– пароль (символ «x» для тіньових паролів);

– UID користувача;

– GID користувача;

– довідкову інформацію про користувача;

– домашній каталог;

– оболонку.

Тіньовий файл /etc/shadow зберігає хеш-суми паролів користувачів для автентифікації. Використання системи тіньових паролів знижує небезпеку злому системи. Файл містить:

– ім’я користувача;

– зашифрований пароль;

– кількість днів від початку епохи Unix (01.01.70) до моменту останньої зміни пароля;

– час життя пароля, хв.;

– максимальний час життя пароля;

– період видачі попереджень;

– період до блокування облікового запису;

– термін життя облікового запису.

Реєструвати користувачів у системі може привілейований користувач.

Групи користувачів призначені для спільного доступу групи осіб до файлів. Інформація про групи зберігається у файлі ***/etc/group***. Формат запису:

<ім’я групи>: <пароль групи>: <GID групи>: <список користувачів>

Користувачі можуть бути в одній або декількох групах. У кожної групи також є унікальне ім’я та ідентифікатор групи (***GID– group-ID***). Це числовий ідентифікатор першої із груп, в яку входить користувач. Надалі приналежність до груп цього користувача визначена в системному файлі груп.

Профілі користувачів (специфічні налаштування і команди, які виконуються під час їх входу в систему) зберігаються у спеціальних файлах. Профіль користувача складається із глобального профілю (файл */etc/profile*) та користувацького профілю (файли *~/.bash_profile, ~/.bash_login, ~/. profile*).

Наведемо послідовність виконання файлів профілю під час входу в сеанс:

1. /etc/profile
2. ~/.bash_profile
3. ~/.bashrc
4. /etc/bashrc.

У разі запуску оболонки з командного рядка виконуються команди ~/.bashrc та /etc/bashrc.

Основне призначення користувацьких і групових ідентифікаторів – визначення приналежності різних системних ресурсів і керування правами, наданими процесам щодо доступу до цих ресурсів. Наприклад, кожний файл належить конкретному користувачу і групі, а у кожного процесу є кілька користувацьких і групових ідентифікаторів, які визначають власників процесу і набір прав для доступу до файлу.

Стандартний механізм прав доступу і безпеки в Linux полягає у тому, що для кожного файлу визначається, хто є власником-користувачем цього файлу і які права (або дозвіл) власника на цей файл. Розуміння прав власності на файл і дозволу має вирішальне значення для користувача.

Кожні файл і каталог в Linux мають три типи власників: **Користувач**, **Група**, **Інші**. *Користувач є власником файлу*. Коли людина створюєте файл, вона стаєте власником файлу. Власність також може бути змінена, але про це далі.

Кожний користувач є частиною певної групи (груп). Група складається з декількох користувачів, і це один із способів керування користувачами у багатокористувацькому середовищі.

Інші – це переважно супергрупа з усіма користувачами в системі. Зазвичай будь-хто, хто має доступ до системи, належить до цієї групи.

Інакше кажучи, «Користувач» – це один користувач, «Група» – це набір користувачів, а «Інший» складається з усіх користувачів системи.

3.2.3. Загальні відомості про дозволи та права доступу до файлів у Linux

В ОС Ubuntu Linux для кожного файлу є певний набір прав доступу, наданий у вигляді 9 символів. Кожні три символи із цього набору визначають *права для власника файлу (user), групи (group)* та *всіх інших (other)* (рис. 3.6) [1, 10].

Рис. 3.6. Схема коду прав доступу до файлів

Перший символ (-) – це прапорець, який вказує який це тип файлу (*flag або file type*). У табл. 3.2 наведено значення, яких може набувати прапорець типу flag. Перші три символи визначають права для власника-користувача (User) файлу – відображається набір дозволів (Permissions) для файлу. Наступні три символи визначають права для групи (Group), до якої належить певний файл. Останні три біта визначають права для всіх інших користувачів (Others).

Таблиця 3.2

Значення типу flag

Прапорець	Опис
-	Означає, що це файл
l	Символічнt посилання (symbolic link) на файл
d	Означає, що це каталог (directory)
b	Блоковий пристрій (block device), наприклад DVD-диск, жорсткий диск і под.
c	Символьний пристрій (character device), наприклад, стример, модем і под.
p	Канал, запам'ятовувальний пристрій FIFO (fifo device)
s	UNIX-сокет (unix domain socket), міжпроцесорної взаємодії

Для кожної категорії встановлюються види доступу, де кожна буква позначає конкретний дозвіл: «***r***» **–** дозвіл (право) **на читання** (**read**), «***w***» **–** дозвіл (право) **на запис** (**write**), «***x***» – право **на запуск файлу** (**execution**), «**-**» – **дозвіл не встановлено**. Однак дозволи «***r***», «***w***» та «***x***» для файлів та каталогів мають різне значення.

Дозволи для файлів (права на файл) у Linux

Кожний файл і каталог в Linux має три дозволи для всіх трьох типів власників – дозволи для файлів і дозволи для каталогів.

Передбачено такі *дозволи для файлів*: ***r*** – право на читання з файлу; ***w*** – дозволяє змінювати вміст і записати у файл (зокрема перезапис або зміна); ***x*** – дозволяє виконати файл.

Передбачено такі *дозволи для каталогів* (*права на каталоги*): ***r*** – дозволяє читати тільки імена файлів у каталозі; ***w*** – має сенс тільки в поєднанні з ***x***, дозволяє маніпулювати з файлами в каталозі (створювати, видаляти і перейменовувати) і додатково виконувати – ***x***); власне, ***x*** – дозволяє мати доступ до файлів та їх атрибутів (але не до їхніх імен), тобто може увійти в каталог; ***w*** без ***x*** не має ніякого ефекту.

Пояснення прав доступу на файл та каталог наведено у табл. 3.3 (OCT – octal, це восьмерична система счислення, BIN – двійкова, Mask –

маска режиму доступу). Код доступу до файлу задають одним із двох режимів:

– символьний режим (комбінація символів),

– восьмеричний режим (послідовність із трьох цифр).

Таблиця 3.3

Права доступу на файл та каталог

OCT (octal)	BIN (binary)	Mask	Права на файл	Права на каталог
0	000	---	немає прав	немає прав
1	001	--x	права на виконання	доступ до файлів та їхніх атрибутів
2	010	-w-	права на запис	немає прав
3	011	-wx	права на запис і виконання	все, крім доступу до імен файлів
4	100	r--	права на читання	тільки читання імен файлів
5	101	r-x	права на читання і виконання	читання імен файлів і доступ файлів та їх атрибутів
6	110	rw-	права на читання і запис	тільки читання імен файлів
7	111	rwx	повні права	усі права

Найбільш типові коди доступу до файлів наведено у табл. 3.4.

Таблиця 3.4

Типові коди доступу до файлів ОС Linux

Код доступу (символьний режим)	Код доступ (восьмеричний режим)	Опис
rwx rwx rwx	777	Операції читання, виконання, записи доступні всім користувачам
rwx r-x r-x	755	Операції читання та виконання доступні всім користувачам; власнику файлу доступна операція запису
rwx r-x ---	750	Повні права доступні тільки для власника, права на читання і виконання доступні групі; всі інші користувачі не мають доступу до файлу
rwx --- ---	700	Доступ до файлу має тільки власник
rw- r-- r--	644	Операції читання і записи доступні власнику; групі та іншим користувачам доступна тільки операція читання

У восьмеричному режимі використовується послідовність із трьох цифр. Сумарний дозвіл на файл обчислюють додаванням таких восьмеричних значень:

• 4 (читання); • 2 (запис); • 1 (виконання). Максимум – 7:

$111_2 = 1*2^0 + 1*2^1 + 1*2^2 = 7$ $110 = 1*2^1 + 1*2^2 = 6$ $101 = 1*2^0 + 1*2^2 = 5$ $100 = 1*2^2 = 4$

Дозволи завжди *мають* таку послідовність: *читати*, *писати* і *виконувати*, тобто **rwx**. І тоді ці дозволи встановлюються для всіх трьох категорій власників у послідовності Користувач, Група та Інші.

Для адміністрування часто зручніше використовувати не буквенне подання прав, а цифрове, у восьмеричній системі (воно коротше). Так, наприклад, правам на файл геть усім відповідає запис 777 (що аналогічно символьному поданню rwxrwxrwx).

Для повноцінного перегляду каталогу потрібні права на читання каталогу і доступ до файлів, а головне – до їх атрибутів, тобто мінімальні розумні права на каталог – це 5 (r-x). Прав 4 (r--) вистачить тільки на перегляд імен файлів, без атрибутів, тобто не будуть відомі розмір файлу, права доступу, власник.

На практиці для каталогів використовують тільки три режими: 7 (rwx), 5 (r-x) та 0 (---).

Додаткові атрибути файлів (sticky bit, SUID, SGID)

Окрім вже розглянутих дев’яти символів при визначенні прав доступу читання / запису /виконання, є ще три додаткові атрибути:

1. ***Sticky bit*** (***SVTX*** – біт закріплення у пам’яті) з’явився у п’ятій редакції UNIX у 1974 році для використання у виконуваних файлах. Його застосовують для зменшення часу завантаження найбільш часто використовуваних програм. Після закриття програми код і дані залишалися у пам’яті, а такий запуск відбувався швидше. (Звідси і назва – біт закріплення у пам’яті.)

Нині *sticky bit* використовують переважно для каталогів, щоб захистити файли в них. У такий каталог може писати будь-який користувач. З такої директорії користувач може видалити тільки ті файли, власником яких він є. Прикладом слугує директорія */tmp*, в якій запис відкритий для усіх користувачів, але небажано видаляти чужі файли.

2. ***SUID*** (***Set User ID***) – біт зміни ідентифікатора користувача. Атрибут виконуваного файлу, що дозволяє запустити його з правами власника. В Unix-подібних системах додаток запускається з правами користувача, що запустив вказаний додаток. Це забезпечує додаткову безпеку, тому процес із правами користувача не зможе отримати доступ на запис до важливих системних файлів, наприклад, */etc/passwd*, який належить суперкористувачеві *root*. Якщо на виконуваний файл встановлено біт *suid*, то під час виконання ця програма автоматично змінює «*ефективний userID*» на ідентифікатор того користувача, який є власником цього файлу. Тобто незалежно від того, хто запускає цю програму, вона під часвиконання має права власника цього файлу.

3. ***SGID*** (***Set Group ID***) – біт зміни ідентифікатора групи. Аналогічний SUID, але цей атрибут стосується групи, а ідентифікатори зберігаються у файлах */etc/group*. Якщо для каталогу встановлений біт SGID, то створювані у ньому об’єкти будуть отримувати групу власника каталогу, а не користувача. У Linux права доступу зберігаються в *inode* (*індексний дескриптор файлу або унікальний номер*), й оскільки *inode* у кожного файлу власний, то й права доступу у кожного файлу свої. Якщо програма виконується з правами користувача і групи, яким належить файл – працюють тільки права власника файлу. Виконуваний файл з установленим атрибутом *suid* є «потенційно небезпечним». Без установленого атрибуту файл не дозволить звичайному користувачеві зробити те, що виходить за межі прав користувача (наприклад, програма *passwd* дозволяє користувачеві змінити тільки власний пароль). Між тим навіть незначна помилка у такій програмі

може призвести до того, що зловмисник зможе змусити її виконати ще якісь дії, не передбачені автором програми. Слід дуже обережно ставитися до цих атрибутів! У разі створення нової директорії у директорії з уже встановленим SGID-бітом у новій директорії SGID-біт встановлюється автоматично!

Права доступу можуть позначатися не трьома символами, а чотирма. У числовому еквіваленті ці атрибути визначаються *першим символом* при чотиризначному позначенні (який часто випускають при призначенні прав), наприклад, у правах 1777 – символ 1 позначає sticky bit, решта атрибутів мають таку числову відповідність: **1** – sticky bit, **2** – SGID, **4** – SUID.

Восьмеричні значення: SUID = **4**000, SGID = **2**000, = **1**000.

Символьні значення: SUID=<u+s>, SGID=<g+s>, sticky bit=<t/T>.

Рядкова буква ***t*** відображається тоді, коли перед установленням sticky bit довільний користувач уже мав право на виконання (x), а прописна (***T***) – коли такого права у нього не було. Кінцевий результат один і той самий, але регістр символу дає додаткову інформацію про вихідні установки.

Таким чином, встановлення значень SUID або SGID дозволить користувачам запускати виконувані файли від імені власника (або групи) файлу, який запускається. Наприклад, команду *chmod* за замовчуванням може запускати тільки *root*. Якщо встановити SUID на виконуваний файл */bin/chmod*, то звичайний користувач зможе використовувати цю команду без використання *sudo*, тому що вона буде виконуватися від імені користувача *root*. У деяких випадках це дуже зручне рішення. До речі, за таким принципом працює команда *passwd*, з допомогою якої користувач може змінити свій пароль.

Якщо встановити SGID для каталогу, то всі файли, створені в ньому під час запуску, набуватимуть ідентифікатор групи каталогу, а не групи власника, який створив файл у цьому каталозі. Аналогічно для SUID: якщо користувач помістив виконуваний файл у такий каталог, запустивши його,

процес запуститься від імені власника (групи) каталогу, в якому лежить цей файл.

3.2.4. Команди для роботи з файловою системою

Синтаксис команд у ***Терміналі*** у загальному вигляді такий:

назва_ програми [-ключ] [значення]

• *Назва_програми* – це ім’я виконуваного файлу з каталогів, записаних у змінну $PATH (*/bin, /sbin, /usr/bin, /usr/sbin, /usr/local/bin, /usr/local/sbin* та ін.).

• [ключ] – опції програми, які може набути виконувана програма.

• [значення] – цей параметр може приймати як аргумент цифри, текст, спеціальні символи і навіть змінні.

Команда ***ls -l <файл>*** дозволяє прочитати права на файли і права власності одночасно, якщо додати ще й ключ ***–a***, будуть відображатися приховані об’єкти.

Наприклад: */media/Work/applicat > ls -la*

Кореневий користувач має повноваження суперкористувача і, як правило, має права на читання, записування й виконання усіх файлів, навіть якщо інші не бачать їх файлів дозволу. Один користувач може бути членом декількох груп, але тільки основна група користувача є власником групи файлу, створеного користувачем. Основна група користувача може бути знайдена за допомогою ідентифікатора команди **-gn <username>**. Якщо користувачу треба знайти власну основну групу, то його ім’я має бути порожнім.

Змінити режим доступу може тільки власник файлу або суперкористувач.

В ОС Ubuntu Linux ***для редагування прав доступу*** використовують команду **chmod** (від англ. *change mode*), яка розміщена в каталозі */bin/*. Використовують такий синтаксис програми chmod:

chmod [посилання] [оператор] [режими] файл ...

Параметр [посилання] визначає користувачів, до яких застосовуватимуться права доступу, і може набувати значень, поданих у табл. 3.5.

Таблиця 3.5

Параметр [посилання]

Значення параметра [посилання]	Клас користувачів	Опис
u	user	Власник файлу
g	group	Група користувачів файлу
o	other	Інші користувачі
a	all	Усі користувачі

Параметр **[оператор]** визначає ту операцію, яка буде передана програмі *chmod*, і набуває таких значень: «+» – додати права; «-» – видалити права, «=» – встановити права. Параметр **[режими]** визначає, які саме права будуть додані або видалені, й набуває значень, наведених у табл. 3.6.

Таблиця 3.6

Параметр [режими]

Параметр [режими]	Назва параметра	Опис
r	read	Додавання прав на читання файлу і вмісту каталогу
w	write	Додавання прав на запис у файл або каталог
x	execute	Додавання прав на виконання файлу або читання вмісту каталогу
X	special execute	Додавання прав на виконання файлу, якщо він є каталогом або вже має право на виконання
s	setuid/setgid	Додавання атрибутів SUID або SGID, що дозволяють запустити файл на виконання із правами власника файлу (SUID) або групи (SGID)
t	sticky	Додавання атрибута t для каталогів, який наділяє правами видалення файлів у цьому каталозі тільки власника цього файлу

Приклад 3.2. Для *восьмеричного подання*:

chmod 744 koshka.txt – встановити права для файлу:

koshka.txt – (*rwx r-- r--*);

chmod -R 775 sobaki – встановити права на каталог *sobaki* та на всі об’єкти, що всередині цього каталогу, включаючи вміст підкаталогів (*rwx rwx r-x*);

-R – рекурсивне призначення прав, тобто призначити права всім об’єктам;

*chmod 700 ** – встановити права тільки для власника на всі файли і каталоги в поточному каталозі, включаючи підкаталоги та їх об’єкти (*rwx --- ---*).

Приклад 3.3. Для *символьного подання*:

chmod g+w koshki.txt – додати користувачам групи файлу *koshki.txt* права на запис у цей файл;

chmod a=rwx sobaki.doc – замінити існуючі права на файлі *sobaki.doc* на повні права усім;

chmod o-w test.cgi – забере права на запис для користувача «Усі інші»;

chmod ug=rw spisok.doc – виставити права на читання і запис файлу *spisok.doc* для власника і групи. Зверніть увагу: якщо у користувача «усі інші» були будь-які права, вони збережуться в незмінному вигляді.

Використання символьного подання дозволяє редагувати права файлів більш гнучко:

chmod u+x, g+w-x koshki.txt – додати власнику файлу *koshki.txt* права на його виконання, користувачам групи дозволити запис і заборонити виконання та залишити права інших користувачів без змін;

chmod u=rwx, g+w, go-x sobaki.doc – встановити повні права для власника файлу, дозволити користувачам групи запис і заборонити виконання усім користувачам, крім власника файлу.

Робота з привілеями root

Для виконання операцій, що вимагають адміністративних прав root, розробники ОС придумали утиліту *sudo*, яка надає необхідні права для виконання тієї чи іншої команди.

Утиліта **sudo** (від англ. *substitute user and do* – підмінити користувача і виконати або *SuperUser Do*) – це утиліта, що надає адміністративні привілеї (привілеї root) за першим запитом користувача. Більш застаріла команда, *su*, мала менші можливості.

Після виконання команди *sudo* користувачеві доступні ті ж повноваження, що й користувачеві root. Виконувати команду sudo можуть не всі користувачі, а лише ті, які входять у групу *admin*.

За замовчуванням перший обліковий запис, яку користувач створив на етапі установки ОС Ubuntu Linux, отримує право на використання команди sudo. В ОС Ubuntu Linux є файл ***/etc/sudoers***, де прописані правила, на підставі яких система вже визначає, чи дозволено тому чи іншому користувачеві виконувати команду sudo [12, 13].

У загальному вигляді синтаксис команди sudo виглядає таким чином:

sudo <команда> [параметри]

У першому аргументі <*команда*> вказують ім’я програми, яку необхідно виконати із правами *root*. У другому аргументі [*параметри*] задають необов’язкові параметри, які можуть бути такими (табл. 3.7).

Приклад 3.4. Якщо ввести команду *sudo -ll Enter*, термінал запитає пароль користувача, після введення якого буде відображена поточна конфігурація прав користувача, від імені якого виконувалася команда sudo з параметрами *-ll*. Ця конфігурація міститься у файлі */etc/sudoers/*.

Таблиця 3.7

Параметри команди sudo

Параметр	Опис
-A (--askpass)	Дозволяє використовувати допоміжну програму для введення пароля
-b (--background)	Дозволяє виконати зазначену команду у фоновому режимі
-C (--close-from=num)	Дозволяє закрити всі дескриптори файлів, які більше або дорівнюють значенню, переданому в параметрі num
-E (--preserve-env)	Дозволяє зберегти оточення користувача при виконанні команди

-e (--edit)	Дозволяє редагувати файли замість виконання команди
-g (--group=group)	Дозволяє виконати команду від імені або *ID,* зазначеної в параметрі *group* групи
-H (--set-home)	Дозволяє встановити для змінної *HOME* домашній каталог
-h (--host=host_	Дозволяє виконати команду на вузлі, якщо така команда підтримується модулем ядра
-i (--login)	Дозволяє запустити оболонку входу в систему від імені зазначеного користувача, а також задати команду, яка буде виконана при вході в систему
-K (--remove-timestamp)	Дозволяє повністю видалити файл *timestamp*
-k (--reset-timestamp)	Дозволяє оголосити недійсним файл *timestamp*
-l (--list)	Дозволяє показати список прав користувача або перевірити задану команду
-n (--non-interactive)	Дозволяє використовувати автономний режим без виведення запитів користувачеві
-P (--preserve-groups)	Дозволяє зберегти вектор групи замість установки цільової групи
-p (--prompt=prompt)	Дозволяє використовувати вказаний запит пароля
-r (--role=role)	Дозволяє створити контекст безпеки *SELinux* із вказаною роллю, яка передана в параметрі role
-s (--stdin)	Дозволяє читати пароль зі стандартного вводу
-s (--shell)	Дозволяє запустити оболонку від імені зазначеного користувача, а також задати команду, яка буде виконана при запуску оболонки
-t (--type=type)	Дозволяє створити контекст безпеки *SELinux* зазначеного типу
-u (--other-user=user)	Дозволяє в режимі списку показувати права користувача
-u (--user=use)r	Дозволяє виконати команду (або редагувати файл) від імені або ID зазначеного користувача
-v (--validate)	Дозволяє оновити тимчасову мітку користувача без виконання команди
--	Дозволяє припинити обробку аргументів командного рядка

Зміна власника для всіх папок і файлів

Змінити власника для всіх папок і файлів, що містяться у папці, можна командою:

sudo chown -R user:group /path/to/dir/ ,

де *user* – новий власник, а *group* – група користувача.

-R вказує на те, що права необхідно змінити *рекурсивно*. Це означає, що потрібно змінити права усіх файлів і папок всередині зазначеної користувачем директорії.

Зміна права доступу на файл або папку

Змінити права доступу на файл або папку можна за командою

chmod -R 700 /path/to/file/or/dir

де **-R** вказує на те, що права слід змінити рекурсивно.

Щоб **додати користувача** в систему, необхідно виконати команду

sudo useradd -m <ім'я користувача>,

ключ «***-m***» означає, що треба створити домашній каталог для користувача, наприклад, ***sudo useradd -m user1***

Щоб ввести пароль, використовують команду:

sudo passwd <ім'я користувача>, sudo passwd user1

Двічі вводять пароль, при цьому створюється домашній каталог */home/user1*. Для видалення користувача використовують команду:

sudo userdel <ім'я користувача>

sudo userdel user1

але його папки залишаються.

Щоб повністю його видалити разом із папками, треба вказати ключ ***-r*** (***remove***):

sudo userdel -r user1

Для створення групи записують команду

sudo groupadd <ім'я групи>:

sudo groupadd Leaders

Щоб видалити групу, використовують команду

sudo groupdel <ім'я групи>:

sudo groupdel Leaders

Щоб додати користувача ***user2*** у групу, використовують: ***usermod*** – (mod- modification), -a (*addition*), G (*group*):

sudo usermod -aG Programer user2

Команда ***uname*** виводить інформацію про ОС, яка встановлена. Команда ***users*** відображає короткий перелік користувачів, які працюють у системі у цей момент.

Команду ***id*** використовують для визначення UID користувача, GID та імені його основної групи, а також списку інших груп, до якого включено користувача, У разі її використання без аргументів команда виведе інформацію про поточного користувача. Якщо же вказати як аргумент ім'я зареєстрованого користувача, виведення команди буде відповідати зазначеному користувачеві.

Інформація про користувача *в Unix зберігається* в декількох місцях:

– файл */etc/passwd* – список користувачів та їх основних властивостей;

– файл */etc/shadow* – зашифрований пароль користувача, доступ до якого є тільки у користувача "root";

– файл */etc/group* – список груп користувачів;

– директорія типу */home/username* – домашня директорія;

– файл */var/spool/mail/username* – поштова скринька.

Файл /etc/passwd – текстовий файл, в кожному рядку якого міститься інформація про одного користувача. Кожний рядок складається із семи полів, розділених двокрапкою:

account:encpassword:UID:GID:GECOS:homedir:shell

Поля мають такі значення:

account – ім'я користувача;

encpassword – зашифрований пароль;

UID – числовий ідентифікатор користувача (*User IDentifier*);

GID – числовий ідентифікатор первинної групи цього користувача (Group IDentifier);

GECOS – інформація про користувача – повне ім'я, телефон, кімната та ін.;

homedir – домашня директорія;

shell – інтерпретатор команд.

Файл /etc/shadow – текстовий файл, який містить інформацію про паролі користувачів. Один рядок – один обліковий запис. Файл належить користувачу *root* та групі *shadow*. Під час запису паролів в Unix застосовують необоротне шифрування за алгоритмом *DES*, тому навіть маючи доступ до зашифрованого паролю не можна дізнатися сам пароль. Система ж, коли треба перевірити пароль, зашифровує рядок, введений користувачем, і порівнює його з тим, що міститься в */etc/passwd*.

Команда ***pwd*** відображає каталог, в якому перебуває користувач.

Команда ***who*** без аргументів дозволяє отримати список користувачів, які працюють у певний момент у системі.

Команда ***whoami*** виведе інформацію про поточного користувача.

Для перегляду довідки про команду використовують команду

man [ім'я_команди],

наприклад, ***man ls***.

Щоб переглянути введені команди, використовують команду ***history***.

Щоб вийти з термінала, натискають «***Alt+F4***» або «***Ctrl+D***».

Для ***створення файлу*** в ОС Ubuntu Linux є команда ***touch [ключ] ім'я файлу***. Наприклад, команда ***touch myfile.txt*** у домашньому каталозі облікового запису користувача створить пустий файл myfile.txt.

Часто для створення файлів використовують команду ***cat***:

cat>ім'я файлу

далі вводять вміст тексту й завершують введення тексту Ctrl+D (ознака кінця файлу).

За командою *cat* виводять ***вміст файлу*** на екран: ***cat /шлях/ім'я_файлу***:

cat myfile.txt

Але слід пам'ятати: якщо файл великий, то на екрані можна побачити тільки кінець файлу. Наприклад, ***cat /etc/passwd*** – виведення файлу, який містить у текстовому форматі список облікових записів користувачів (акаунтів). Є першим і основним джерелом інформації про права користувача ОС, тому в цьому випадку такою командою користуватися недоцільно.

Вміст файлу ***/etc/shadow*** можна подивитися, виконавши команду

sudo cat/etc/shadow

За допомогою «!!» можна викликати всю попередню команду. Це надзвичайно корисно у тих випадках, коли виявляється, що для запуску команди потрібні *root*-привілеї. Швидке ***sudo !!*** дозволяє заощадити час.

Для *створення каталогу* в ОС Ubuntu Linux використовують команду *mkdir* (від англ. *make directory* – створити каталог).

Синтаксис команди такий: ***mkdir [ключ] каталог***

Команда mkdir також дозволяє однією командою створювати одразу декілька каталогів у поточному каталозі, розділяючи їх пробілами [16]:

mkdir folderl folder2 folder3

Для *видалення порожніх каталогів* призначена команда rmdir, синтаксис якої виглядає таким чином: ***rmdir [ключ] ... каталог*** ... Ключ ***-p*** дозволяє видалити усі вкладені каталоги у цьому каталозі.

Для видалення каталогів існує команда ***rm*** (від англ. *remove* – видалити):

rm [ключ] [файл]

За замовчуванням команда *rm* видаляє не каталоги, а тільки файли. Щоб видалити каталог, потрібно команді *rm* передати ключ ***-r***, який дозволить рекурсивно видалити каталог та його вміст. Наприклад, ***rm -r ny2***

Щоб переміщати між каталогами файлову систему, застосовують команду *cd*:

cd [ключ] шлях_до_директорії

Є й такі команди:

– перейти в домашній каталог – «***cd***» без параметрів або «***cd ~***»;

– перейти на рівень вище «***cd ..***»;

– перейти в директорію двома рівнями вище – «***cd ../..***»;

– перейти в кореневий каталог «***cd /***».

Для ***копіювання файлів і каталогів*** в ОС Linux призначена команда ***cp*** (від англ. *copy* – копіювати). За замовчуванням команда копіює тільки файли, але якщо додатково вказати відповідний ключ, то буде виконано копіювання каталогів. Команда дозволяє копіювати один файл в інший файл, а також декілька файлів у заданий каталог. Синтаксис команди *cp*:

cp [ключ] ... джерело каталог_ (призначення)

У разі використання команди *cp* рекомендується застосовувати опцію «***-i***», щоб отримати попередження, коли файл буде записуватись, наприклад,

cp -i s1.txt s2.txt

Ключ «**-r**» забезпечує рекурсивне копіювання каталогів:

cp -r /home/qwe/R1 /home/qwe/R2qwe – ім'я користувача

Для ***переміщення файлу з одного каталогу в інший*** використовують команду ***mv***. Синтаксис цієї команди аналогічний синтаксису команди *cp*. Команда працює таким чином: спочатку копіює файл (чи каталог), а тільки

потім видаляє вихідний файл (каталог). Наприклад, mv /home/qwe/r1 /home/qwe/r2.

Команду *mv* використовують і ***для перейменування файлів та каталогів*** (тобто переміщення їх всередині одного каталогу). Для цього треба задати як аргументи старе й нове імена файлу: ***mv oldname newname***.

Дерево каталогів виводять на термінал командою ***tree***. Однак цю команду треба встановити, вибравши одну з команд:

sudo apt install tree або ***sudo snap install tree***

tree шлях_до_папки

наприклад, tree /home/qwe/

Команда ***head*** за замовчуванням виводить перші десять рядків файлу, її синтаксис:

head [опції] ім'я файлу.

Наприклад: ***head alpha.txt***

До найбільш вживаних опцій належать:

-n (--lines) – показує задану кількість рядків замість 10, які виводяться за замовчуванням. Наприклад, ***head -n 5 alpha.txt*** виводяться перші п'ять рядків.

-v (--verbose) – перед текстом виводить назву файлу. Наприклад, ***head -v alpha.txt***.

Команда *cat* дозволяє проглядати вміст файлів, але іноді необхідно подивитися тільки те, що міститься у кінці файлу, наприклад, вміст лог-файлу саме про останні повідомлення про помилки. Для цього використовують команду ***tail***, яка дозволяє виводити задану кількість рядків з кінця файлу. Синтаксис команди: ***tail [опції] ім'я файлу***. Наприклад, якщо ввести ***tail /var/log/syslog***, виводяться останні десять рядків системного журналу.

До найбільш вживаних опцій належать:

-n – вивести вказану кількість рядків з кінця файлу, наприклад:

tail -n 100 /var/log/syslog

-v – перед текстом вивести назву файлу;

-f – оновлювати інформацію в разі появи нових рядків у файлі,

tail -f /var/log/syslog

Ця команда може також застосовувати фільтр, що спрощує пошук інформації в тому ж системному журналі.

Для читання великих текстових файлів (щоб не захаращувати екран терміналу), пошуку тексту і файлів моніторингу в режимі реального часу доцільно застосовувати команду ***less***. Ця команда не читає весь файл перед запуском, вона призначена «тільки для читання», тому немає ризику випадкового редагування файлів, які переглядає користувач.

Синтаксис: ***less [опції] ім'я файлу***. Для перегляду файлу: ***less alpha.txt*** Якщо файл короткий, *less* додає порожні рядки вгорі, але у наведеному файлі немає зайвих рядків.

Перелік усіх опцій можна подивитися, виконавши команду ***man less***.

Для опанування команди *less* рекомендовано скопіювати наявний файл з */etc/services* у домашній каталог користувача; оскільки це великий файл із сотнями рядків, то один раз зробивши копію, користувач може використовувати його для практики.

Щоб вийти з команди *less* у будь-якій точці файлу, слід натиснути «q» (quit).

Результат команди *less* поділяється на декілька сторінок. Можна побачити тільки текст, який заповнюється до екрана терміналу. Можна використовувати клавіші зі стрілками вгору і вниз для переміщення по рядках. Якщо треба перемістити сторінку за сторінкою, використовують клавішу *пробілу*, щоб перейти на наступну сторінку, і клавішу «*b*», щоб повернутися до попередньої сторінки.

Якщо треба перейти до початку файлу, використовують ключ «*g*», а коли треба перейти в кінець файлу, натискать клавішу «*G*».

Гарячі клавіші:

• Стрілка вгору – переміщення на один рядок вгору.

• Стрілка вниз – переміщення на один рядок вниз.

• Пропуск або PgDn – переміщення на одну сторінку вниз.

• *b* або *PgUp* – перемістити на одну сторінку вгору.

• *g* – перемістити у початок файлу.

• G – перемістити у кінець файлу.

• *ng* – перейти на *n*-й рядок.

Для виведення посторінково вмісту файлу акаунтів слід скористатися командою *less /etc/passwd*

Щоб відобразити номери рядків у виведенні команди *less*, слід використовувати опцію *N*: ***less – N ім'я файлу***.

Для пошуку слова або фрази, або шаблону регулярного виразу, користувач має натиснути «/» і ввести все, що він хоче знайти. Наприклад, «/text»/.

Шаблони імен файлів

Шаблони імен файлів та каталогів як у командній оболонці Bash, так і в засобах програмування дозволяють виконувати однотипні дії над групою файлів. Тобто шаблони дозволяють виділити групу файлів, чиї імена задовольняють певні умови. Також шаблони використовують для пошуку файлів. Шаблони імен файлів і каталогів задають за допомогою спеціальних символів. Найчастіше використовують два символи: «*****» – зірочка, «**?**» – знак питання.

Зірочка означає будь-яку групу символів, їх кількість не має значення. Наприклад, **.html* – шаблон виділення усіх html-файлів, шаблон *my*.docx* – усі імена файлів починаються на *my* та мають розширення *docx*. Шаблон **2020** – усі файли, в імені яких є група символів 2020.

Знак питання в шаблоні означає будь-який один символ. Шаблон descrip?ion.pdf – це файли description.pdf, descripsion.pdf. Шаблон zvit.??? – це файли zvit.txt, zvit.doc, zvit.xls.

Команда пошуку – утиліта **find**, яка встановлена за замовчуванням в усіх дистрибутивах Linux. Це потужний інструмент, який дозволяє ефективно виконувати різні завдання системного адміністрування, такі як керування дисковим простором, рекурсивні операції з файлами і резервне копіювання. Вона має багато можливостей.

Синтаксис **find [папка] [параметри] критерій пошуку шаблон [дія]**

• Папка – каталог для пошуку (разом з підкаталогами).

• Параметри – додаткові параметри (наприклад, глибина пошуку і под.).

• Критерій – критерій пошуку, слід вказати, які файли треба шукати (ім’я, дата створення, права, власник тощо).

• Шаблон [дія] – значення, за яким треба вибирати файли, та інформація, що робити з кожним знайденим файлом, який відповідає критеріям.

Команда **find** без параметрів виводить список файлів з поточного каталогу та його підкаталогів.

Файли за їх іменами можна знайти за командою

find [шлях до каталогу] -name [ім’я файлу]

Розглдянемо опції команди:

-name – це критерій пошуку за іменем з урахуванням регістру;

find . -name testfile1.txt – пошук в поточному каталозі файлу testfile1.txt

./testfile1.txt – «.» посилання на поточний каталог;

-iname – пошук файлу без врахування регістру, наприклад:

```
find . -iname "a.txt"
./a.txt
./A.txt
```

find . -name "*.jpg" – пошук усіх файлів зображень *.jpg* в поточному каталозі та підкаталогах, наприклад:

```
./genxfacebook2.jpg
./genxfacebook1.jpg
./Moodle2.jpg
./moodle.jpg
./moodle/moodle1.jpg
./genxfacebook.jpg
```

Приклад 3.5. *Пошук усіх файлів зображень .jpg в каталозі /home*:

```
find /home -name "*.jpg"
/home/vagrant/Moodle2.jpg
/home/vagrant/moodle.jpg
/home/me/hello.jpg
```

Пошук за типом файлу: параметр «-type». Типи файлів можуть бути такі: ***f*** – звичайні файли, ***d*** – каталоги, ***l*** – символічні посилання, ***b*** – блокові пристрої, ***c*** – символьні пристрої, ***p*** – іменовані канали, ***s*** – сокети.

Приклад 3.6. Пошук каталогів:

```
$ find . -type d
.
./.ssh
./.cache
./moodle
```

Пошук за розміром файлів: параметр «-size». Знак «+» означає, що програма буде шукати файли, розмір яких більший за вказане число, знак «-» означає, що розмір шуканих файлів беде, відповідно, меншим.

Приклад 3.7. Знайти файли, більші за 1 ГБ:

```
find . -size +1G
./Microsoft_Office_16.29.19090802_Installer.pkg
./android-studio-ide-183.5692245-mac.dmg
```

Одиницями виміру можуть бути:

• b – блоки по 512 байтів,

• c – байти,

• k – кілобайти,

• M – мегабайти,

• G – гігабайти.

Пошук порожніх каталогів і файлів: параметр «-empty».

Пошук усіх каталогів і файлів, які є порожніми:

```
find . -empty
./.cloud-locale-test.skip
./datafiles
./b.txt
...
./.cache/motd.legal-displayed
```

Пошук за іменем користувача: параметр «-user username». Пошук усіх файлів і каталогів, які належать конкретному користувачу.

Приклад 3.8. Знайти усі файли і каталоги користувача *ubuntu* в каталозі /home (2>/dev/null – перенаправлення повідомлення про помилки на пристрій /dev/null й видає чисті остаточні дані):

```
find /home -user ubuntu 2>/dev/null
/home/ubuntu
/home/ubuntu/.bash_logout
/home/ubuntu/.bashrc
/home/ubuntu/.ssh
/home/ubuntu/.profile
```

Пошук за режимом доступу: параметр «-perm». Шукаємо файли з усіма дозволами 777: find /home -perm 777

Утиліта **find** Linux виконує пошук файлів і папок за заданими критеріями, а потім *дозволяє виконувати дії* з результатами пошуку. Наведемо деякі зумовлені *дії*:

• «**-delete**» – видалення файлів, які відповідають критеріям пошуку;

• «**-ls**» – відображення детальних результуючих даних *ls* з розмірами файлів і кількістю входів;

• «**-print**» – показує повний шлях до відповідних файлів. Ця дія спрацьовує за замовчуванням, якщо не вказано іншої дії;

• «**-exec**» – виконує наступну команду в кожному рядку результатів пошуку.

Наприклад, якщо треба знайти всі порожні файли й видалити їх, виконують команду: find **.** -empty -delete

Але перед використанням дії видалення рекомендується виконати команду один раз із дією -print й підтвердити результати.

Дія **-exec** є особливою, вона дозволяє виконати обрану команду в результатах пошуку, а саме: **-exec command {} \;** Атрибути команди:

• command – команда, яку потрібно виконати в результатах пошуку, наприклад, ***rm, mv або ср.***;

• {} – власне результати пошуку.

Команда закінчується крапкою з комою зі зворотною косою.

Наприклад, команда пошуку і видалення всіх порожніх файлів може бути написана у такий спосіб: find **.** -empty -exec rm {} \;

Наступна команда копіює всі PNG-файли образів у каталог *backup/images*:

```
find . -name "*.png" -exec cp {} /backups/images \;
```

Таким чином, опанування можливостей команди *find* дозволить спростити завдання системного адміністрування під час роботи з файлами та каталогами.

3.2.5. Система контролю версій та спільної розробки проектів з відкритим вихідним кодом – Git

Система контролю версій (*Version control system– VCS*) дозволяє відстежувати зміни у початковому коді своїх проектів, повертатися до попередніх версій в разі критичних помилок, а також ділитися своїм кодом

з усіма бажаючими й приймати від них виправлення. Ці зміни у початковому коді записуються у файл або набір файлів протягом часу, тому розробник може повернутися до попередньої версії своєї системи або проекту в разі критичних помилок. Також є можливість переглядати, хто останній із розробників проекту змінював файл і це викликало проблему. Можна створювати безпечну резервну копію проекту, створювати декілька машин для роботи над одним проектом та інше [9, 30].

Git – це набір консольних утиліт, які відстежують і фіксують зміни у файлах, зазвичай початкового коду проекту, дозволяють повернутися до більш старої версії проекту, порівнювати, аналізувати, зливати зміни і багато іншого. Цей процес і є контролем версій.

У *Git* є три основних стани, в яких можуть перебувати файли розробника:

– зафіксований (*committed*),

– змінений (*modified*),

– підготовлений (*staged*).

Зафіксований стан означає, що файл вже збережено у локальній базі розробника (фіксація змін). Змінений стан – це файли, які змінилися, але ще не були зафіксовані. Підготовлені файли – це змінені файли, відмічені для включення у наступний *committed*.

Проект під керуванням системи контролю версій Git складається переважно з трьох розділів:

• *репозиторій* – база даних для запису стану або зміни файлів проекту. Він містить всі необхідні метадані Git та об’єкти для нового проекту. Зазвичай це те, що копіюється під час клонування сховища з іншого комп’ютера на мережевому або віддаленому сервері;

• *робочий каталог* або *область* – зберігає копію файлів проекту, з якими може працювати розробник (доповнювати, видаляти і виконувати інші дії з модифікації);

• *проміжна область* – файл (відомий як *index under Git*) у каталозі *Git*, що зберігає інформацію про зміни, які розробник готовий зберегти (*to commit*), стан файлу або набору файлів у репозиторій.

Є два основні типи VCS, причому головною їх відмінністю є кількість репозиторіїв.

• *Централізовані системи керування версіями* (*Centralized Version Control Systems – CVCS*): кожний член команди проекту отримує власний локальний робочий каталог, однак усі розробники вносять зміни тільки в один центральний репозиторій.

• *Системи керування розподіленою версією* (*Distributed version control systems – DVCS*): кожний член команди проекту отримує власний локальний робочий каталог та каталог Git, де він може здійснювати фіксацію версії проекту. Після того, як розробник зафіксує щось локально, інші члени команди не можуть отримати доступу до змін, поки він / вона не викладає їх в центральний репозиторій. Git є прикладом DVCS.

Крім того, репозиторій *Git* може бути «*bare*» (тобто порожній, репозиторій, який не має робочого каталогу) або «*non-bare*» (з одним робочим каталогом). *Shared* (або публічні) сховища завжди мають бути *bare* – усі репозиторії *Github* є *bare*. *Git* є безкоштовною, відкритою, розподіленою системою, тобто не залежить від одного центрального сервера, на якому зберігаються файли. Замість цього він працює повністю локально, зберігаючи дані в папках на жорсткому диску, які називають репозиторієм. Також можна зберігати копію сховища онлайн, що суттєво полегшує роботу над одним проектом для декількох людей. Для цього використовуютья сайти ***github і bitbucket***.

Git – ефективна система контролю версій під час роботи з великими проектами, має чудову систему розгалуження і злиття. Вона призначена для оброблення даних як набір версій міні-файлової системи, що зберігається в каталозі *Git*.

Технологія роботи з використанням Git дуже проста: розробник вносить зміни у файли у своєму робочому каталозі, а потім вибірково додає тільки ті файли, які були змінені, у проміжну область (*staging area* – область підготовлених файлів). Ця область стане частиною наступної версії проекту. Після того, як розробник готовий, він виконує команду *get commit*, яка бере файли з проміжної області і зберігає їх назавжди в каталозі Git.

Щоб встановити *Git*-пакет, треба виконати такі команди:

sudo apt update
sudo apt install git

Може виникнути помилка – «ресурс тимчасово недоступний». Коли такі помилки виникають, треба оновити систему. Для надійності краще перезавантажити систему, виконавши три такі команди:

sudo apt update оновлення даних,

sudo apt upgrade оновлення системи,

sudo apt update оновлення даних,

а потім **sudo apt install git**.

Далі треба виконати ще дві команди (вказати своє ім’я користувача та електронну пошту): **git config --global user.name "Your Name"**

git config --global user.email "youremail@domain.com

Щоб перевірити налаштування Git, треба виконати команду

git config --list

Створення нового репозиторію Git

Загальні репозиторії або централізовані робочі репозиторії дуже поширені. Наприклад, користувачу необхідно налаштувати віддалений загальний репозиторій для системних адміністраторів / програмістів з різних відділів організації для роботи над проектом під назвою *bashscripts*, який буде зберігатися в */projects/scritpts /* на сервері.

Потрібно створити *SSH* (*Secure Shell* – безпечний протокол для реалізації віддаленого доступу до ПК) на віддаленому сервері. Для забезпечення *SSH*-доступу користувачеві необхідні *SSH*-клієнт та *SSH*-сервер. Кожна ОС має власний набір програм, що забезпечують з'єднання.

Так, для Linux це *lsh* (*server i client*), *openssh* (*server i client*) *https://selectel.ru/blog/ssh-ubuntu-setup/*. Для Mac OS часто використовують NiftyTelnet SSH. В ОС Windows для реалізації з'єднання через *SSH*-протокол найчастіше використовують додаток PuTTY:

https://www.putty.org.

Тепер на віддаленому сервері треба створити потрібний каталог, потім – групу під назвою *sysadmins* (додати всіх членів команди проекту в цю групу, наприклад, *user admin*), і встановити відповідні дозволи для цього каталогу:

mkdir-p /projects/scripts/ /*створення каталогів*/

groupadd sysadmins /*створення групи *sysadmins* */

usermod -aG sysadmins admin /*додати користувача у групу*/

chown :sysadmins -R /projects/scripts/ *зміна власника папок на sysadmins*/

chmod 770 -R /projects/scripts /*повний доступ до себе та членів групи*/

Далі ініціалізують *bare* (порожній) репозиторій проекту:

git init --bare /projects/scripts/bashscripts

Цей порожній каталог Git є центральним сховищем для проекту. Створюють список каталогів, щоб побачити всі файли й каталоги:

ls -la /projects/scripts/bashscripts/

Клонування сховища Git

Для клонування віддаленого загального репозиторію Git на локальний комп'ютер через SSH слід виконати команду [30]:

git clone ssh://admin@remote_server_ip:/projects/scripts/bashscripts

Для клонування репозиторію в певний каталог *(~/bin/bashscripts*), використовують таку команду:

git clone ssh://admin@remote_server_ip:/projects/scripts/bashscripts ~/bin/bashscripts

Тепер користувач має локальний примірник проекту в *non-bare* репозиторії (з робочим каталогом) та може створити початкову структуру проекту (приміром, додати файл README.md, підкаталоги для різних категорій скриптів; або ж переконфігурувати для зберігання сценаріїв reconnaissance і т. д.):

cd ~/bin/bashscripts/
ls -la

Перевірка зведення стану Git

Для відображення статусу робочого каталогу користувача слід використати команду **status**, яка покаже йому всі зміни, які він виконав, які файли не відслідковуються Git; усі зміни, які були виконані тощо:

git status

Зміни в Git Stage і Commit

Користувач може зберегти всі зміни, використовуючи команду ***add*** із ключем **-a,** й виконати початковий *get commit*. Ключ **-a** вказує команді автоматично додавати файли, які були змінені, а ключ **-m** використовують для виведення повідомлення про виконання комміту:

git add -A
git commit -a -m "Initial Commit"

Публікація локальних коммітів у віддалений репозиторій Git

У міру того як команда проекту виконує поставлене завдання, можна пушити (розміщувати) зміни в центральному репозиторії за допомогою команди **push**: **git push origin master**

Це означає, що локальний репозиторій одного з учасників проекту **git** має бути синхронізований із центральним репозиторієм проекту. Для перевірки факту синхронізації треба знову запустити команду **status**:

git status

Керівник проекту також може повідомити своїм колегам, щоб вони почали роботу над проектом, і клонувати репозиторій на їх локальні комп'ютери.

Створення нового розгалуження в Git

Розгалуження (**Branch**) дозволяє розробнику працювати з функціями його проекту або виправляти проблеми, не використовуючи основу коду (основну гілку). Щоб створити нову гілку й потім переключитися на неї, використовують команди **branch** та **checkout** відповідно: **git branch latest**

git checkout latest

Також можна створити нову гілку і переключитися на неї за один крок, використовуючи команду ***checkout*** із ключем **-b**:

git checkout -b latest

Крім того, можна створити нову гілку, засновану на іншій гілці:

git checkout -b latest master

Щоб перевірити, в якій гілці користувач перебуває, використовують команду *branch* (символ зірочки вказує на активну гілку):

git branch

Після створення і переходу на нову гілку можна внести в неї деякі зміни і виконати декілька коммітів:

vim sysadmin/topprocs.sh
git status
git commit add sysadmin/topprocs.sh
git commit -a -m 'modified topprocs.sh'

Об'єднання змін в одній гілці до іншої

Щоб об'єднати зміни в *гілці-філії* в *гілку-майстер*, переходять на головну гілку й виконують злиття (**merge**):

git checkout master
git merge test

Якщо більше не потрібна конкретна гілка, можна видалити її, використовуючи ключ **-d**:

git branch -d test

Збереження змін з віддаленого центрального сховища у локальне

Якщо члени команди внесли зміни в центральний репозиторій проекту, координатор проекту може завантажити будь-які зміни у свій локальний екземпляр проекту за допомогою команди **pull**:

git pull origin

або

git pull origin master #if you have switched to another branch

Огляд репозиторія Git і виконання порівнянь

Git має корисні функції, які відстежують усі дії, які відбулися у репозиторії користувача, що дозволяє переглядати історію проекту.

Першою функцією є журнал *Git*, в якому відображається фіксація змін: **git log**.

Друга функція – команда **show**, яка відображає різні типи об’єктів (наприклад, коммітовів, теги, дерева і под.): **git show**.

Третя функція – команда **diff**, яку використовують для порівняння або відображення відмінності між гілками, відображення змін між робочим каталогом та індексом чи між двома файлами на диску і багато чого іншого. Наприклад, щоб показати відмінність між основною і дочірньої гілкою, треба виконати таку команду: **git diff master latest**.

Таким чином, **Git** дозволяє команді розробників працювати разом, використовуючи один і той самий файл (чи файли) з можливістю перегляду змін у файлі (файах). *Git* можна використовувати для керування вихідним кодом, файлами або будь-яким файлом, що зберігається на комп’ютері. Для більш детального ознайомлення з Git є документація *Git Online*.

3.3. Фільтрація даних у файлі. Низькорівневе «введення – виведення»

3.3.1. Утиліта grep – потужний інструмент системного адміністратора

Одним із потужних інструментів системних адміністраторів є утиліта *grep*. Саме при прийомі на роботу роботодавці, які потребують системних адміністраторів Linux, *DevOps*-інженерів (це архітектор, який проектує систему, відповідає за будь-яку автоматизацію задач під час створення програмного продукту – від написання коду до тестування та випуску додатка, причому система повинна вирішувати проблеми клієнтів), серед низки вимог обов’язково вказують, що претендент має вміти працювати з цією утилітою (як приклад, компанія Vodafon).

Утиліта *grep* (***G**lobal **R**egular **E**xpression and **P**rint – g/re/p*) шукає шаблон з файлу й виводить рядок на екрані. Вона постачається на кожному дистрибутиві Linux. Якщо з якоїсь причини її не встановлено у системі

розробника, її можна легко встановити через менеджер пакетів (apt-get на Debian / Ubuntu) [1]:

sudo apt-get install grep #Debian/Ubuntu

Синтаксис команди виглядає таким чином:

grep [опції] шаблон [ім'я файлу ...]

або

команда | grep [опції] шаблон

Опції – це додаткові параметри, за допомогою яких вказуються різні налаштування пошуку і виведення, наприклад кількість рядків або режим інверсії. *Шаблон* – це будь-який рядок або регулярний вираз, за яким вестиметься пошук.

Файл та команда – це те місце, де буде вестися пошук. Як буде видно далі, *grep* дозволяє шукати в декількох файлах і навіть в каталозі, використовуючи рекурсивний режим.

Розглянемо *основні опції утиліти*:

-b – показувати номер блоку перед рядком;

-c – підрахувати кількість входжень шаблону;

-h – не виводити імені файлу в результатах пошуку всередині файлів Linux;

-i – не враховувати регістр;

-l – відобразити тільки імена файлів, в яких знайдено шаблон;

-n – показувати номер рядка у файлі;

-s – більше не показувати повідомлення про помилки;

-v – інвертувати пошук, видавати всі рядки, крім тих, що містять шаблон;

-w – шукати шаблон як слово, оточене пробілами;

-e – використовувати *регулярні вирази* під час пошуку;

-An – показати входження і *n* рядків до нього;

-Bn – показати входження і *n* рядків після нього;

-Cn – показати *n* рядків до і після входження.

Приклади використання утиліти grep

Пошук тексту у файлі. Знайти користувача User у файлі користувачів Linux: ***grep User /etc/passwd***

Знайти користувача User без врахування регістра: ***grep -i "user" /etc/passwd*** або ***cat /etc/password | grep -i "user"***

Або можна скопіювати файл *passwd* (щоб нічого не пошкодити), наприклад, у каталог */Documents* – це один із каталогів користувача, наприклад, *sysadmin*, тобто шлях до цієї папки буде */home/sysadmin/Documents*:

```
:~/Documents$ cp /etc/passwd .
```

cp /etc/passwd . (*пропуск і крапка – це поточна директорія!*)

А тепер у файлі *passwd* знаходять тільки рядки із *sysadmin*:

```
:~/Documents$ grep sysadmin passwd
```

grep sysadmin passwd

```
sysadmin:x:1001:1001:System Administrator,,,,:/home/sysadmin:/bin/bash
```

Знаходять усі файли, в яких є слово Linux, виводять назви цих файлів та ці рядки:

```
~/Desktop/MyData$ grep linux ./*
```

grep linux ./* (Поточна директорія це «.», в ній шукаємо усі файли «*»)

```
./linux-history.txt:The GNU Project, started in 1983 by Richard Stallman, has the goal of creating a "complete Unix-compat
ible software system" composed entirely of free software. Work began in 1984.[32] Later, in 1985, Stallman started the Fre
e Software Foundation and wrote the GNU General Public License (GNU GPL) in 1989. By the early 1990s, many of the programs
 required in an operating system (such as libraries, compilers, text editors, a linux shell, and a windowing system) were
completed, although low-level elements such as device drivers, daemons, and the kernel were stalled and incomplete.[33][no
t in citation given]
./mydatfile.doc:                        <p>Donec@linux.ca</p>
./rockyou.txt:linux
./rockyou.txt:linuxx
./rockyou.txt:linux1
./rockyou.txt:linux80
./rockyou.txt:linux123
```

Якщо ігнорувати регістр (*case sensitive*), тобто шукати й *linux* й *Linux*, то

grep -i Linux ./*

```
~/Desktop/MyData$ grep -i linux ./*
```

```
./linux-history.txt:Although not released until 1992 due to legal complications, development of 386BSD, from which NetBSD,
 OpenBSD and FreeBSD descended, predated that of Linux. Torvalds has also stated that if 386BSD had been available at the
time, he probably would not have created Linux.[35]
./linux-history.txt:In 1991, while attending the University of Helsinki, Torvalds became curious about operating systems[3
7] and frustrated by the licensing of MINIX, which at the time limited it to educational use only.[36] He began to work on
 his own operating system kernel, which eventually became the Linux kernel.
./linux-history.txt:Torvalds began the development of the Linux kernel on MINIX and applications written for MINIX were al
so used on Linux. Later, Linux matured and further Linux kernel development took place on Linux systems.[38] GNU applicati
ons also replaced all MINIX components, because it was advantageous to use the freely available code from the GNU Project
with the fledgling operating system; code licensed under the GNU GPL can be reused in other computer programs as long as t
hey also are released under the same or a compatible license. Torvalds initiated a switch from his original license, which
 prohibited commercial redistribution, to the GNU GPL.[39] Developers worked to integrate GNU components with the Linux ke
rnel, making a fully functional and free operating system.[33]
```

Знаходять зашифрований пароль для користувача *sysadmin*:

sudo grep sysadmin /etc/shadow

Є можливість здійснювати пошук рекурсивно, тобто читати всі файли в кожному каталозі, щоб знайти рядок “192.168.1.5”:

```
grep -r "192.168.1.5" /etc/    (або -R)
/etc/ppp/options:# ms-wins 192.168.1.50
/etc/ppp/options:# ms-wins 192.168.1.51
```

3.3.2. Регулярні вирази (regular expressions) або більш тонкі фільтри

Регулярні вирази – це спеціалізована система пошуку та обробки тексту, заснована на порівнянні частини тексту із зразками для пошуку (шаблонами).

Регулярні вирази поділяються на такі види:

– звичайні регулярні вирази (*basic regexp*);

– регулярні вирази з розширеним синтаксисом (*extended regexp*).

Шаблон – особливий символ, який замінює один або більше звичайних символів. Є різновиди команди *grep*: *grep* – можливе використання звичайних регулярних виразів. Опція «**-E**» вказує на використання

розширеного синтаксису регулярного виразу ***grep -E*** або **egrep**, тобто розширена команда grep (extended).

Регулярний вираз – це текстовий рядок, який описує певний шаблон пошуку, що для цього використовується. Регулярні вирази можуть фільтрувати дані за положенням, наприклад, які містяться на початку або в кінці рядка, на початку або в кінці слова, можуть задавати діапазони (наприклад, "будь-яка буква від а до м"), виділяти класи або види символів ("друковані символи" або "знаки пунктуації"), а також багато іншого.

Найбільш часто вживаними операторами регулярних виразів є ,символи, які називають ***анкорами*** (спеціальні символи, які вказують на місце, де можна знайти в рядку необхідний збіг):

«.» (крапка) – у зазначеному місці може міститися будь-який одиночний символ. Наприклад, *grep "d.g" file1.* Тобто шукають слово, яке починається з «d», закінчується на «g», в середині може бути будь-який символ. Або шаблон такий: *"T.....h",* тобто всередині може бути будь-яка кількість символів.

«^» (символ з таблиці символів – *Circumflex*) − це SHIFT+6 на латинській розкладці, означає початок рядка – віртуальний символ, який є першим символом на початку кожного рядка.

«**$**» – останній символ у кінці кожного рядка.

Набір символів у квадратних дужках **[]** називають ***символьним класом***, він дозволяє вказати інтерпретатору регулярних виразів, що на певному місці в рядку може стояти один із перерахованих символів. Зокрема, [abc] задає можливість появи у тексті одного з трьох зазначених символів, а [1234567890] задає відповідність одній із цифр. Можна вказати діапазон символів: наприклад [A-Za-z] відповідає всім букв латинського алфавіту.

Якщо потрібно вказати символи, які не входять до зазначеного набору, то використовують символ ^ всередині квадратних дужок (**[^]** –

окрім цього символа), наприклад [^0-9] означає будь-який символ, крім цифр.

[A-E] – будь-яка буква з набору A, B, C або E у верхньому регістрі.

[^A-E] – будь-яка буква, окрім букв A, B, C або E у верхньому регістрі.

Квантифікація, або пошук послідовностей

Квантифікатор вказує кількість входжень символу або групи символів, яка містяться перед ним в регулярному виразі.

Квантифікатор **(?, *, +)** після символу, символьного класу або групи визначає, скільки разів може трапитися попередній вираз. Слід враховувати, що квантифікатор може належати більш ніж до одного символу в регулярному виразі тільки якщо це символьний клас або група.

Розглянемо квантори:

«?» – кількість повторень: нуль або одне входження (0|1), тобто буде відповідати нулю або одному з виразів, які стоять перед виразом.

Наприклад, «*colou*?r» відповідає й *color*, й *colour*.

«*****» – кількість повторень: символа немає або більше (0|>), тобто відповідає будь-якій кількості символів, які передують зірочці, у тому числі й нулю.

Наприклад, «*g*gle*» відповідає *ggle*, *gogle*, *google* та ін., тобто жодного символу або будь-яка їх кількість.

«+» – кількість повторень: один або більше разів (1|>).

Наприклад, «*go+gle*» відповідає *gogle*, *google* і т. д., але не *ggle,* тобто хоча б один раз.

«|» – логічне АБО.

Приклад 3.9. Маємо файл ***heroes.txt*** з такими рядками:

```
1 Catwoman
2 Batman
3 the tick
4 Spider Man
5 Black Cat
```

```
6 Batgirl
7 Danger Girl
8 Wonder Woman
9 luke cage
10 the punisher
11 Ant Man
12 Langer Karl
13 Aquaman
14 SCUD
15 Spider Woman
16 Blackbolt
17 Martian Manhunter
18 Bt-shop
```

Знайти збіги, які містять один символ та послідовність «at»:

```
grep -E '.at' heroes.txt
1 Catwoman
2 Batman
3 Batgirl
```

Знайти рядки, які починаються з *Bat*:

```
grep -E '^Bat' heroes.txt
1 Batman
2 Batgirl
```

Знайти рядки, які закінчуються на man:

```
grep -E 'man$' heroes.txt
1 Catwoman
2 Batman
3 Wonder Woman
4 Aquaman
5 Spider Woman
```

Знайти рядки незалежно від регістру, які починаються з *bat* або *cat*:

```
grep -i -E '^(bat|cat)' heroes.txt
```

Знайти рядки, які починаються на B, а далі містять одну з літер *l/a/t*:

```
grep -E 'B[lat]' heroes.txt
1 Batman
2 Black Cat
3 Batgirl
4 Blackbolt
5 Bt-shop
```

Знайти рядки, які збігаються з шаблоном, окрім символу D:

```
grep "[^D]ang" heroes.txt
1 Langer Karl
```

Знайти рядки, які починаються з маленької літери:

```
grep "^[a-z]" heroes.txt
1 the tick
2 luke cage
3 the punisher
```

Приклад 3.10. Знайти такі дані:

`grep "St[^1-9]d" file3` – шукаємо слова, які починаються із «St», закінчуються на «d» і не містять цифри від 1 до 9.

`grep "lak*" file4` – шукаємо збіги `'lake'` або `'la'` або `'lakkkk'`.

`grep "lak+" file5` – літера «к» повинна з'явитися хоча б один раз, результатом може бути `'lake'` або `'lakkkk'`, але не `'la'`.

`grep "ba?b" file6` – шукаємо одне входження символу або жодного, результат – рядок `bb` або `bbb`.

Щоб поєднати команду `grep` з іншими командами, необхідно вибирати не всі рядки лог-файлу в кінці, а відфільтрувати там, де є err:

```
tail -f /var/log/syslog | grep err
```

3.3.3. Модель «введення – виведення» в системах UNIX

У мові С для здійснення файлового «введення – виведення» використовують механізми стандартної бібліотеки мови, яка оголошена в заголовному файлі ***stdio.h***. Консольне «введення – виведення» – це окремий випадок файлового «введення – виведення». Як передбачено в моделі КІС (Клієнт-Інтерфейс-Сервер), сервером стандартних механізмів «введення – виведення» мови С (*printf, scanf, FILE *, fprintf, fputc* тощо) є бібліотека мови, а сервером низькорівневого «введення – виведення» в *Linux* є ядро ОС.

Користувацькі програми взаємодіють з ядром ОС за допомогою спеціальних механізмів, які називають *системними викликами* (*system calls, syscalls*). Зовні системні виклики реалізовані у вигляді звичайних функцій мови С, проте кожного разу викликаючи таку функцію, розробник звертається безпосередньо до ядра ОС.

Список усіх системних викликів Linux можна знайти у файлі /usr/include/asm/unistd.h.

Одна з відмінних рис *моделі* «введення – виведення» *UNIX* є поняття універсальності «введення – виведення». Це означає, що в кожній файловій системі і в кожному драйвері пристрою реалізується *один і той самий набір системних викликів* «введення – виведення»*:* ***open()***, ***read()***, ***write()***, ***close()***, ***lseek ()***, тобто ці системні виклики застосовують для виконання «введення – виведення» в усіх типах файлів, зокрема й пристрої, наприклад термінали [10]. Таким чином, програма, написана з використанням лише цих системних викликів, буде працювати з будь-яким типом файлу. Наприклад, вона виконує такі команди:

$ ./copy test test.old копіювання звичайного файлу;

$ ./copy a.txt /dev/tty копіювання звичайного файлу у заданий термінал;

$ ./copy /dev/tty b.txt копіювання введення із заданого терміналу у звичайний файл;

$ ./copy /dev/pts/16 /dev/tty копіювання введення з іншого терміналу відповідає програмі, яка використовує системні виклики «введення – виведення» для виконання команди в загальному вигляді $ ./copy oldfile newfile;

```
#include <sys/stat.h>
#include <fcntl.h>
#include "tlpi_hdr.h"
#ifndef BUF_SIZE /* Дозволяє "cc -D" перекрити визначення */
#define BUF_SIZE 1024
#endif
int
main(int argc, char *argv[])
{
      int inputFd, outputFd, openFlags;
      mode_t filePerms;
      ssize_t numRead;
      char buf[BUF_SIZE];

      if (argc!=3 || strcmp(argv[1], "--help")==0)
         usageErr("%s old-file new-file\n", argv[0]);
      /* Відкриття файлів введення та виведення */
      inputFd=open(argv[1], O_RDONLY);
```

```
      if (inputFd==-1)
         errExit("opening file %s", argv[1]);
      openFlags=O_CREAT | O_WRONLY | O_TRUNC;
      filePerms=S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP |
           S_IROTH | S_IWOTH; /* rw-rw-rw- */
      outputFd=open(argv[2], openFlags, filePerms);
      if (outputFd==-1)
           errExit("opening file %s", argv[2]);
      /* Переміщення даних по досягненні кінця файлу введення або
виникнення помилки */
      while ((numRead=read(inputFd, buf, BUF_SIZE)) > 0)
         if (write(outputFd, buf, numRead) != numRead)
             fatal("couldn't write whole buffer");
      if (numRead==-1)
          errExit("read");
      if (close(inputFd)==-1)
          errExit("close input");
      if (close(outputFd)==-1)
          errExit("close output");
      exit(EXIT_SUCCESS);
}
```

Отже, спочатку треба відкрити обидва файли, а потім переписати усі записи старого файлу в новий.

Оскільки деталі реалізації конкретної файлової системи або пристрою обробляються всередині ядра (ядро перетворює запити додатків на «введення – виведення» у відповідні операції файлової системи або драйверіпристроїв), то під час написання прикладних програм можна взагалі ігнорувати фактори, що належать до пристрою. Там, де потрібно отримати доступ до конкретних властивостей файлової системи або пристрою, у програмі можна використовувати системний виклик *ioctl()*, який надає інтерфейс для доступу до керування апаратними пристроями, але це вже виходить за межі моделі «введення – виведення».

Слід пам’ятати, що файл – це найпростіша базова абстракція в Linux, будь-який об’єкт в Linux – це файл. Екран, клавіатура, апаратні й віртуальні пристрої, канали, сокети – також файли. Це дуже зручно, оскільки до всього можна застосовувати одні й ті самі механізми «введення – виведення».

Перш ніж з файлу можна буде зчитати інформацію або записати до нього нові дані, файл потрібно відкрити. Ядро підтримує попроцесний спи-

сок відкритих файлів, який називають файловою таблицею (або *таблицею файлових дескрипторів*, тобто у системі ведеться облік відкритих файлів для кожного процесу). Файлова таблиця індексується за допомогою невід'ємних цілих чисел – тип *int у мові С*, які називають файловими дескрипторами (скорочено ***fd*** – ***file descriptor***), це **ціле число відповідає відкритому файлу**. Кожний запис у списку таблиці містить інформацію про відкритий файл, зокрема вказівник на збережену в пам'яті копію файлового де скриптора й асоційованих з ним метаданих. До *метаданих* відносять, зокрема, *позицію у файлі або вказівник на позицію у файлі та режими доступу*. Під час відкриття файлу повертається його дескриптор, а при наступних операціях (зчитування інформації, записування й под.) дескриптор файлу приймається як первинний аргумент. Файлові дескриптори починаються зі значення 0.

Копія таблиці дескрипторів (тобто таблиці відкритих файлів всередині процесу) *прихована в ядрі.* Розробник не може отримати прямий доступ до цієї таблиці. Програміст повинен лише розуміти, що *кожний процес має свою копію таблиці дескрипторів*. У межах одного процесу всі дескриптори унікальні (навіть якщо вони відповідають одному і тому самому файлу або пристрою). У різних процесах дескриптори можуть збігатися або не збігатися – це не має жодного значення, оскільки у кожного процесу є власний набір відкритих файлів.

У командній оболонці, наприклад bash, відкриті три файли: *стандартне введення (дескриптор 0), стандартне виведення (дескриптор 1) і стандартний потік помилок (дескриптор 2).* Коли під оболонкою запускається програма, у системі створюється новий процес, який для цієї оболонки є дочірнім, а отже, отримує копію таблиці дескрипторів свого батька (рос. «родителя», тобто всі відкриті файли батьківського процесу). Таким чином, програма може здійснювати консольне «введення – виведення» через ці дескриптори.

Таблиця дескрипторів, окрім іншого, містить інформацію про поточну позиції «читання – запису» для кожного дескриптора. Під час відкриття файлу позиція «читання – запису» встановлюється в нуль. Кожний прочитаний чи записаний байт збільшує на одиницю показник поточної позиції.

Щоб виконати файлове «введення – виведення» програми, зазвичай використовують функції «введення – виведення», що містяться у стандартній бібліотеці мови С – ***stdio***, яка включає функції *fopen(), fclose(), scanf(), printf(), fgets( ), fputs()* і под. Функції *stdio* нашаровуються поверх системних викликів «введення – виведення» (*open(), clos (), read(), write()* тощо).

Системний виклик різниться від звичайної функції, яка викликається із програми, винятково швидкістю роботи, оскільки звернення безносередньо до ядра відбувається значно швидше, ніж використання бібліотек. Більшість функцій бібліотеки С – просто обгортки над системними викликами.

Системний виклик open() або відкриває наявний файл, або створює і відкриває новий файл, при цьому отримується файловий дескриптор, який забезпечує найбільш простий спосіб доступу до файлу:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

Системний виклик *open ()* асоціює файл, на який указує *ім'я шляху pathname*, із файловим дескриптором, що повертається в разі успіху. Як файлову позицію вказують його початок (нуль), і файл відкривається для доступу відповідно до заданих прапорців (параметр *flags*).

Прапорці для open()

Аргумент *flags* – це порозрядне АБО, що складається з одного або декількох прапорців. Він має вказувати режим доступу, який може мати одне з таких значень: O_RDONLY (відкритий тільки для читання),

O_WRONLY (тільки для запису) або O_RDWR (одночасно і для читання, і для запису), – згідно зі стандартом SUSv3 (*System V release 3, який є поєднанням стандартів POSIX.1, POSIX.2 і SUS в єдиний документ).*

Аргумент бітової маски режиму (*mode*) вказує на права доступу, які мають бути присвоєні файлу. (Використовуваний тип даних mode_t є цілочисельним типом, визначеним у SUSv3.) Якщо під час виклику *open()* не вказано прапорець O_CREAT, то аргумент *mode* може бути випущений. Наприклад, такий код відкриває каталог */home/kidd/madagascar* для читання:

```
int fd;
fd= open("/home/kidd/madagascar", O_RDONLY);
if (fd==-1)
/* помилка */
```

Наведемо ще приклади використання *open()*:

```
    /* Відкриття існуючого файлу для читання */
    fd=open("startup", O_RDONLY);
    if (fd==-1)
    errExit("open");
    /* Відкриття нового або існуючого файлу для читання і запису з урізанням до нуля байтів; надання власнику виключних прав доступу на читання і запис */
    fd=open("myfile", O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
    if (fd==-1)
    errExit("open");
    /* Відкриття нового або існуючого файлу для запису; записуються дані повинні завжди додаватися в кінець файлу */
    fd = open("w.log", O_WRONLY | O_CREAT | O_APPEND,
    S_IRUSR | S_IWUSR);
    if (fd== -1)
    errExit("open");
```

Покажемо значення аргументу *flags*:

O_CREAT – створення файлу, якщо його ще не існує,

O_TRUNC – усікання наявного файлу до нульової довжини,

O_APPEND – записи додаються виключно в кінець файлу,

S_IRUSR (S_IREAD) – користувач має права на читання файлу (00400);

S_IWUSR (S_IWRITE) – користувач має права на запис інформації у файл (00200).

У разі виникнення помилки при спробі відкрити файл системний виклик *open()* повертає **-1**, а в *errno* ідентифікується причина помилки. Можливі помилки, які можуть статися, – EACCES, EISDIR, EMFILE, ENFILE, ENOENT, EROFS, ETXTBSY. Їх значення можна подивитись на відповідних сторінках керівництва для кожного системного виклику або бібліотечної функції.

Системний виклик creat() створює й відкриває новий файл із вказаним іменем шляху або, якщо файл вже було створено, відкриває його та усікає його до нульової довжини. У результаті *creat()* повертає дескриптор файлу, який може бути використаний у таких системних викликах:

```
#include <sys/stat.h>
#include <fcntl.h>
int creat(const char *pathname, mode_t mode);
```

Виклик функції: ***creat(pathname, mode)*** має бути еквівалентним:

open(pathname, O_WRONLY|O_CREAT|O_TRUNC, mode)

Наступний приклад створює файл ***/tmp/file*** із дозволом для читання і запису для власника файлу й дозволу на читання для групи та інших. Отриманий дескриптор файлу присвоюється змінній *fd*:

```
#include <fcntl.h>
...
int fd;
mode_t mode=S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH;
char *filename= /tmp/file";
...
fd= creat(filename, mode);
...
```

Системний виклик *create()* вважають застарілим, його можливості також надаються функцією *open()*.

Системний виклик read() дозволяє зчитувати дані з відкритого файлу, на який посилається дескриптор *fd*:

```
#include <unistd.h>
ssize_t read (int  fd,  void  *buf, size_t len);
```

Кожний виклик зчитує не більше *len* байтів (максимальна кількість байтів, які треба прочитати з пам'яті й записати у *buf*). Тип даних *size_t* – беззнаковий цілочисельний. *Buf* – вказівник на область пам'яті (адреса буфера пам'яті), куди записуються початкові дані. Зчитування відбувається з поточним значенням зміщення, у файлі, вказаному у файловому дескрипторі *fd*. Якщо виклик успішний, повертається кількість байтів, записаних у *buf*. У разі помилки виклик повертає **-1** та встановлює *errno*. Файлова позиція (вказівник позиції у файлі) просувається залежно від того, скільки байтів було зчитано з *fd*. Якщо об'єкт, зазначений у *fd*, не має можливості позиціонування (наприклад, це файл символьного пристрою), то зчитування завжди починається з «поточної» позиції.

Наприклад, зчитування усіх *len*-байтів з обробленням усіх помилок буде записано таким чином:

```
ssize_t ret;
while (len !=  && (ret=read(fd, buf, len)) !=0) {
        if (ret==–1) {
              if (errno==EINTR)       /*сигнал був отриманий раніше, ніж були прочитані будь-які байти, виклик буде повторений*/
                    continue;
        perror ("read");
        break;
        }
        len-=ret;             /*len=len-ret*/
        buf+=ret;             /*buf=buf+ret*/
}
```

Цикл зчитує *len*-байтів з актуальної файлової позиції, яка дорівнює значенню *fd*, й записує їх у buf. Зрозуміло, що значення buf щонайменше має дорівнювати значенню *len*. Читання триває, поки не буде отримано всі *len*-байти або до досягнення кінця файлу. Якщо прочитано нульову

кількість байтів, яка, однак, менша за *len*, то значення *len* зменшується на кількість прочитаних байтів, *buf* збільшується на ту ж кількість і виклик повторюється. Якщо виклик повертає -1 та значення *errno*, яке дорівнюється EINTR, то виклик повторюється без поновлення параметрів. Якщо виклик повертає -1 з будь-яким іншим значенням *errno*, викликається *perror()* – він виводить опис виниклої проблеми у стандартну помилку, а виконання циклу припиняється.

Системний виклик write() записує дані у відкритий файл:

```
#include <unistd.h>
ssize_t  write(int fd, const void * buf, size_t count);
```

Під час виклику *write()* записується деяка кількість байтів, яка менша за значення *count* або дорівнює йому. Запис починається з *buf* (адреса даних, які записуються), встановленого в поточну позицію файлу, *count* – кількість даних, які записується з буфера. Посилання на потрібний файл визначається за файловим дескриптором *fd*. Якщо файлом є об'єкт, який не підтримує позиціонування (це стосується символьних пристроїв), запис завжди починається з поточної позиції «курсора».

У разі успішного виконання виклику повертається кількість записаних байтів, а файлова позиція, відповідно, оновлюється. У випадку помилки повертається **-1** і встановлюється відповідне значення *errno*. Виклик *write()* може повернути 0, це означає, що було записано 0 байтів. Наприклад:

```
unsigned long   word=1720;
size_t  count;
ssize_t  nr;
count=sizeof(word);
nr=write(fd, &word, count);
if (nr==-1)
        /* помилка, перевірити errno */
else if (nr!=count)
        /* можлива помилка, але значення 'errno' не встановлено */
```

Системний виклик close() закриває відкритий дескриптор файлу, вивільняючи його для наступного повторного використання процесом.

Коли процес припиняє роботу, всі його відкриті дескриптори файлів автоматично закриваються:

```
#include <unistd.h>
int  close(int fd);
```

Виклик *close()* скасовує відображення відкритого файлового дескриптора *fd* й розриває зв'язок між файлом і процесом. Цей дескриптор файлу більше не є допустимим і ядро вільно може перевикористати його як значення, яке повертається для наступних викликів *open()* або *creat()*. В разі успішного виконання виклик *close()* повертає 0, у разі помилки він повертає -1 та встановлює *errno* у відповідне значення. Приклад використання такий:

```
if (close(fd)==–1)
      perror ("close");
```

Перед закриттям файлу слід переконатися, що він є на диску, а в додатку необхідно використати одну з можливостей синхронізації, розглянутих вище. Після закриття файлу можуть виникнути деякі побічні ефекти. Коли закривається останній з відкритих файлових дескрипторів, який посилається на певний файл, в ядрі вивільняється структура даних, за допомогою якої забезпечувалося подання файлу. Коли ця структура вивільняється, вона роз'єднується зі збереженою у пам'яті копією індексного де скриптора, асоційованого з файлом. Якщо індексний дескриптор ні з чим більше не пов'язаний, він також може бути вивільненим з пам'яті (цей дескриптор може залишитися доступним, оскільки ядро кеширує індексні дескриптори з міркувань продуктивності, але це не гарантується). У деяких випадках розривається зв'язок між файлом і диском, але файл залишається відкритим аж до цього розриву. В такому випадку фізичного видалення цього файлу з диска не відбувається, поки файл не буде закритий, а його індексний дескриптор видалений із пам'яті, тому виклик *close()* також може привести до того, що ні з чим не пов'язаний файл виявиться фізично віддаленим із диска.

Системний виклик lseek() встановлює файловий зсув відкритого файлу, на який вказує дескриптор *fd*, відповідно до значень, заданих в аргументах *pos* та *origin*. Цей виклик не здійснює ніяких інших операцій, крім поновлення файлової позиції, зокрема не ініціює жодних дій, пов'язаних із «введенням – виведенням»:

```
#include <sys/types.h>
#include <unistd.h>
off_t  lseek(int fd, off_t pos, int origin);
```

Поведінка виклику *lseek()* залежить від аргументу *origin*, який може мати одне із таких значень:

SEEK_CUR – поточна файлова позиція дескриптора *fd*, встановлена в його поточне значення, плюс *pos*. Якщо *pos* дорівнює нулю, то повертається поточне значення файлової позиції.

SEEK_END – поточна файлова позиція дескриптора *fd*, встановлена в поточне значення довжини файлу, плюс *pos*, який може мати негативне, позитивне або нульове значення. Якщо *pos* дорівнює нулю, то зміщення встановлюється в кінець файлу.

SEEK_SET – поточна файлова позиція дескриптора *fd*, встановлена в *pos*. Якщо *pos* дорівнює нулю, то зміщення встановлюється в початок файлу.

У разі успіху цей виклик повертає нову файлову позицію. В разі помилки він повертає -1 та присвоює *errno* відповідне значення, наприклад, повертає поточне розміщення файлового зсуву:

```
curr=lseek(fd, 0, SEEK_CUR);
```

У наступному прикладі файлова позиція дескриптора *fd* отримує значення 1825:

```
off_t ret;
ret=lseek (fd, (off_t) 1825, SEEK_SET);
if (ret==(off_t) -1)
    /* помилка */
```

Як альтернативу можна встановити файлову позицію дескриптора *fd* у кінець файлу:

```
off_t ret;
ret=lseek (fd, 0, SEEK_END);
if (ret==(off_t) -1)
    /* помилка */
```

Виклик *lseek()* повертає оновлену файлову позицію, тому його можна використовувати для пошуку поточної файлової позиції. Потрібно встановити значення SEEK_CUR у нуль:

```
int pos;
pos=lseek (fd, 0, SEEK_CUR);
if (pos==(off_t) -1)
    /* помилка */
else
    /* 'pos' – це поточна позиція fd */
```

Виклик *lseek()* найчастіше застосовують для пошуку початку файлу, кінця файлу або для визначення поточної позиції файлового дескриптора.

Заборонено застосовувати *lseek()* до конвеєра, FIFO-пристрою, сокета або термінала – виклик аварійно завершиться із встановленим для *errno* значенням ESPIPE.

3.4. Сценарії в оболонці bash

3.4.1. Скрипт. Створення сценарію

Командна оболонка *BASH* дозволяє створювати скрипти (*bash-скрипти)* за допомогою групування декількох команд, які виконують певну дію.

Скрипт – це звичайний текстовий файл, що містить *системні* або *вбудовані* команди оболонки. Вбудовані команди виконуються самою оболонкою, системні – це виконувані файли. Вбудовані команди в різних оболонках можуть виконуватися по-різному, зокрема вони виконуються

швидше, ніж системні. Для багатьох вбудованих команд є системні аналоги (*pwd і /bin/pwd*).

Місця розміщення системних команд: */bin, /sbin, /usr/bin, /usr/sbin, /usr/local/bin, /usr/local/ sbin.*

Сценарій командної оболонки (скрипт) – це програма, яка виконується командною оболонкою [15]. Вона містить набори тих самих команд, які можна вводити із клавіатури, зібрані у файли та об'єднані якоюсь спільною метою. Відповідно, ці команди у командній оболонці виконуються послідовно. При цьому результати роботи команд можуть становити як самостійну цінність, так і слугувати вхідними даними для інших команд. Сценарії *bash* можна використовувати для різних цілей, таких як виконання команди оболонки, одночасний запуск декількох команд, налаштування адміністративних завдань, виконання автоматизації часто виконуваних дій. Тому знати основи програмування на *bash* важливо для кожного користувача Linux.

Командний рядок дозволяє виконати кілька команд за один раз, ввівши їх через крапку з комою: ***pwd; whoami***.

Насправді це перший *bash*-скрипт, в якому задіяні дві команди. Працює він так: спочатку команда *pwd* виводить на екран відомості про поточну робочу директорію, потім команда *whoami* показує дані про користувача, під яким він увійшов у систему. Використовуючи подібний підхід, можна поєднувати як завгодно багато команд в одному рядку, обмеження полягає лише у максимальній кількості аргументів, яку можна передати програмі.

Скрипт може бути запущений на виконання таким чином:

$bash ім'я_файлу

Для файлів сценаріїв оболонки *bash* встановлюють розширення ***.sh***. Будь-який сценарій для Bash починається із вказівки у першому рядку комбінації [16]:

#!/bin/bash

В *Unix* два символи «**#!**» називають «*шебанг*», вони повідомляють системі про те, що наступний за ними аргумент – це програма, яку застосовують для виконання цього файлу. У цьому разі програма ***/bin/sh*** – командна оболонка, застосовувана за замовчуванням. У загальному випадку символ «*#*» у скрипті означає коментар. Програма пропонує створити порожній файл з використанням команди *touch* або в будь-якому текстовому редакторі (*nano* – за замовчуванням або *mc,* це комбінація клавіш Shift+F4).

Наприклад,

```
#!/bin/bash
# This is a comment
pwd
whoami
```

У файлі, як і в командному рядку, можна записувати команди в одному рядку, розділяючи крапкою з комою. Далі треба зберегти файл і надати йому ім'я, наприклад *myscript*. Для того, щоб його запустити, треба **зробити** цей **файл виконуваним**, тобто налаштувати дозвіл (рос. разрешение) за допомогою команди **chmod**, змінивши режим файлу (**file mode**) і зробивши його виконуваним для всіх користувачів:

$ chmod u+x <ім'я сценарію> $ chmod u+x first

Додані режими: для власника (User), виконання (x – eXecutable).

Аналогічну функціональність реалізує така команда:

chmod 744 <ім'я_файлу_сценарію>

тобто chmod u+x ./myscript

Сценарій на виконання з поточного каталогу запускають за допомогою такої команди:

./ <ім'я_файлу_сценарію>

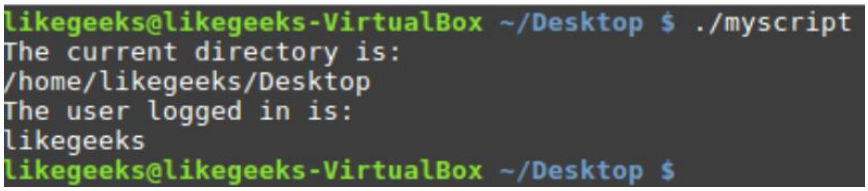
Тепер можна виконати цей файл: ./myscript

Для виведення тексту в консоль Linux застосовують команду ***echo***.

Відредагуємо скрипт:

```
#!/bin/bash
# our comment is here
echo "The current directory is:"
pwd
echo "The user logged in is:"
whoami
```

Результат роботи скрипта має такий вигляд:



Імена змінних мають починатися з літери і складатися з латинських букв, цифр і знака підкреслення (_). Для звернення до значення змінної слід перед іменем поставити символ **$**.

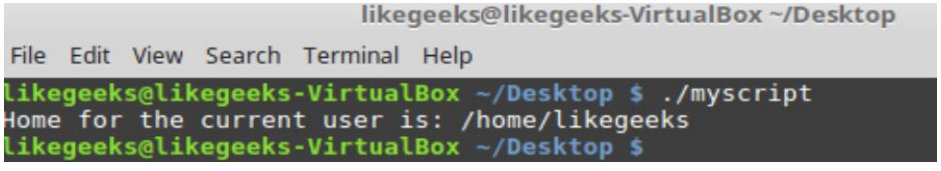
Є два типи змінних, які використовують у bash-скриптах: змінні середовища та змінні користувача.

Змінні середовища

Іноді в командах оболонки потрібно працювати з якимись системними даними. Наприклад, треба вивести домашню директорію поточного користувача [16]:

```
#!/bin/bash
# display user home
echo "Home for the current user is: $HOME"
```

Системну змінну *$HOME* беруть у подвійні лапки, тоди матимемо такий результат запуску сценарію:



Використання змінної середовища у сценарії

Потрібно вивести на екран значок долара. Для цього використовують керуючий символ – зворотний слеш перед знаком долара:

```
echo "I have \$1 in my pocket"
```

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
I have $1 on my pocket
```

Змінні користувача

Bash-скрипти дозволяють задавати і використовувати у сценарії власні змінні. Такі змінні зберігають значення доти, поки не завершиться виконання сценарію. Як і у випадку із системними змінними, до змінних користувача можна звертатися, використовуючи знак долара:

```
#!/bin/bash
# testing variables
grade=5
person="Adam"
echo "$person is a good boy, he is in grade $grade"
```

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
Adam is a good boy, he is in grade 5
```

Bash чутливий до пропусків під час установки змінних і виконання деяких операцій. Неправильні варіанти встановлення змінних (видають помилку):

```
var_a ="Hello World"
var_a= "Hello World"
var_a = "Hello World"
```

Значення змінній присвоюють таким чином: **ім’я змінної=значення**, причому без пробілів біля знака «=»: var_a="Hello World"

Підстановка команд

Одна з найбільш корисних можливостей bash-скриптів – це можливість отримувати інформацію з виведення команд і призначати її

змінним, що дозволяє використовувати цю інформацію де завгодно у файлі сценарію. Зробити це можна двома способами:

– за допомогою значка зворотного апострофа «`»:

– за допомогою конструкції $().

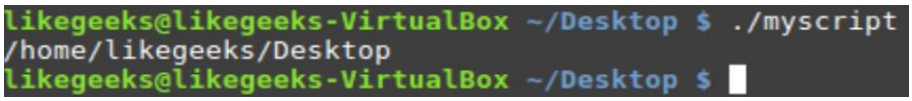
Використовуючи перший підхід, слід простежити за тим, щоб замість зворотного апострофа не ввели одиночну лапку. Команду потрібно записати з двома такими значками: **mydir=`pwd`**

Використовуючи другий підхід, те саме записують так: **mydir=$ (pwd)**

Скрипт має вигляд:

```
#!/bin/bash
mydir=$(pwd)
echo $mydir
```

Виведення команди *pwd* буде збережено у змінній *mydir*, вміст якої за допомогою команди *echo* буде виведений у консоль.



Командна оболонка дозволяє виконувати арифметичні операції. Для цього вираз, який треба інтерпретувати як арифметичний, записують у подвійні круглі дужки, і перед ними ставлять знак долара – конструкція має вигляд **$((a+b))**.

Наприклад:

```
$foo=$((((5+3*2)–4)/2))
echo $foo
```

Щоб отримати значення змінної, використовують такий синтаксис:

```
ALFA=$BETA+$GAMMA
```

Скрипт:

```
#!/bin/bash
var1=$((5 + 5))
echo $var1
var2=$(($ var1 * 2))
echo $var2
```

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
10
20
```

Виконувану команду необхідно записати у зворотні апострофи й присвоїти змінній: **`FILES=`/bin/ls -a/home/student`**

Користувач може надати значення змінній через командну оболонку за допомогою команди **read**, якій як аргумент передається ім'я потрібної змінної, наприклад:

```
$read CHOICE
Привіт !!!
$echo "Ви ввели "${CHOICE}
```

Використання коментаря

Символ "*#*" використовують для додавання коментаря до одного рядка в *bash*-скрипті. Створюють новий файл з назвою «*comment_example.sh*» та додають такий сценарій із коментарем в одному рядку:

```
#!/bin/bash
# Add two numeric value
((sum=25+35))
#Print the result
echo $sum
```

Запускають файл *bash*-командою: ***$ bash comment_example.sh***

```
ubuntu@ubuntu-VirtualBox:~/code$ bash comment_example.sh
60
ubuntu@ubuntu-VirtualBox:~/code$
```

3.4.2. Умовний оператор if then та else

Під час написання скриптів часто виникає потреба у перевірці (розгалуженні) будь-яких процесів. Оператори **if | then** перевіряють код завершення переліку команд на «успішне завершення (істина)», що означає «0». Якщо це так, то виконується одна або більше команд, які записані після оператора *then*. Якщо перевірка повертає «неуспішне завершення

(неправильно)», це означає «1», й виконується оператор *else* – «інакше» як того вимагає умова. На завершення умови обов'язково закриваємо його «**fi**»!

Конструкції перевірки if|then

if умова; then
команди
fi

Приклад 3.11.

```
#!/bin/bash
if pwd
then
echo "It works"
fi
```

Якщо виконання команди *pwd* завершиться успішно, у консоль буде виведений текст «It works».

Приклад 3.12. Знайти якогось користувача в */etc/passwd*, і якщо знайти його вдалося, повідомити про те, що він існує:

```
#!/bin/bash
user=likegeeks
if grep $user /etc/passwd
then
echo "The user $user Exists"
fi
```

Результат після запуску цього скрипта:

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
likegeeks:x:1000:1000:likegeeks,,,:/home/likegeeks:/bin/bash
The user likegeeks Exists
```

Конструкції перевірки if|then|else

if умова; then
команди 1
else
команди 2
fi

Приклад 3.13.

```
#!/bin/bash
user=anotherUser
if grep $user /etc/passwd
then
echo "The user $user Exists"
else
echo "The user $user does not exist"
fi
```

Виконання скрипта пішло по гілці *else*:

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
The user anotherUser doesn't exist
```

Використання дужок

Квадратні дужки «**[**» є спеціальною вбудованою командою ***test***, яка сприймає свої аргументи як вираз порівняння або файлову перевірку **[.....]**. Подвійні дужки «**[[**» є розширеним варіантом від «[» й зарезервованим словом, а не командою, його bash виконує як один елемент з кодом повернення. Всередині «**[[....]]**» дозволяється виконання операторів **&&**, ||, які спричиняють помилку у звичайних дужках «[....]», отже, варіант з подвійною дужкою більш універсальний. Круглі дужки «**((**» є арифметичними виразами, які так само повертають код «0» – такі вирази можна використовувати в операціях порівняння.

Список логічних операторів, які використовують для ***if*** | ***then*** | ***else***, наведено в табл. 3.8.

Таблиця 3.8

Логічні оператори для if | then | else

Оператор	Значення	Оператор	Значення
«**-z**»	рядок порожній	«**-n**»	рядок не порожній
«**=, (==)**»	рядки рівні	«**!=**»	рядки нерівні
«**-eq**» *equal*	дорівнює	«**-ne**» *not equal*	не дорівнює
«**-lt, (<)**»	менше	«**-gt, (>)**» *greater*	більше
«**-le, (<=)**» *less or equal*	менше або дорівнює	«**-ge, (> =)**» *greater or equal*	більше або дорівнює
«**!=**»	не дорівнює		

Наприклад: «-a, (&&)» – логічне «І»; «**-o, (||)**» – логічне «АБО».

Для порівняння чисел треба використовувати оператори «<» або «>», які вміщені у круглі дужки.

Приклад 3.14.

```
#!/bin/bash
if ((3 <6)); then
echo “Так”
fi
```

*Використання команди test, якою є квадратні дужки (*можна використовувати й подвійні квадратні дужки як розширений варіант команди test). Якщо «3» менше «6», набираємо «Так».

Приклад 3.15.

```
#!/bin/bash
if [3 -lt 6]; then
echo “Так”
fi
```

Приклад 3.16. Використаємо подвійні квадратні дужки (застосуємо оператор «**&&**»). Якщо перший вираз 2=2 (істина), тоді виконуємо другий, і якщо він теж 2=2 (істина), друкуємо «Вірно»:

```
#!/bin/bash
a="2"
b="2"
if [[2="$a" && 2="$b"]]; then
echo “Вірно”
else
echo “Не вірно”
fi
```

Якщо b="3", спрацює `“Не вірно”.`

Вкладання декількох перевірок

Bash дозволяє вкладати у блок кілька блоків:

```
#!/bin/bash
if [Умова 1]; then
    if [Умова 2]; then
```

```
            команда 1
        else
            команда 2
        fi
    else
        команда 3
    fi
```

Побудова багатоярусних конструкцій

Для побудови багатоярусних конструкції, коли треба виконувати декілька перевірок, а не одну, краще використовувати ***elif*** – коротку форма запису конструкції ***else if***:

```
if [Умова 1]; then
        команда 1
        команда 2
elif [Умова 2]; then
        команда 3
        команда 4
else
        команда 5
fi
```

Приклад 3.17.

```
        #!/bin/bash
        user=anotherUser
        if grep $user /etc/passwd
        then
              echo "The user $user Exists"
        elif ls /home
        then
        echo "The user doesn’t exist but anyway there is a directory under /home"
        fi
```

3.4.3. Цикл ПОКИ

Якщо потрібно повторити виконання послідовності команд, але заздалегідь невідомо, скільки разів слід їх виконати, застосовують цикл ***while***:

```
while <умова> do
<оператори>
Done
```

Доти, поки виконується умова, виконуються оператори.

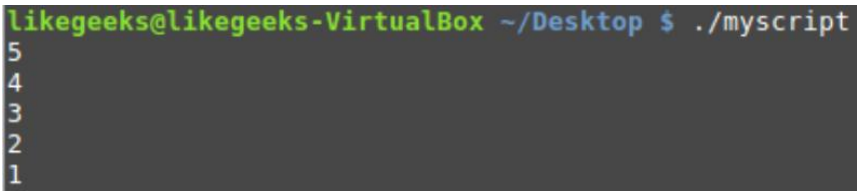
Приклад 3.18. Програма перевірки паролів:

```
#!/bin/sh
echo "Enter password"
read trythis
while ["$trythis"!="Secret"]; do
echo "Sorry, try again"
read trythis
done
exit 0
```

Приклад 3.19.

```
#!/bin/bash
var1=5
while [ $var1 -gt 0 ] do
echo $var1
var1=$[ $var1 - 1 ]
done
```

Якщо змінна $var1 більше нуля, виконується тіло циклу – віднімається одиниця на кожній ітерації. Якщо ж $var1=0, цикл припиняється:



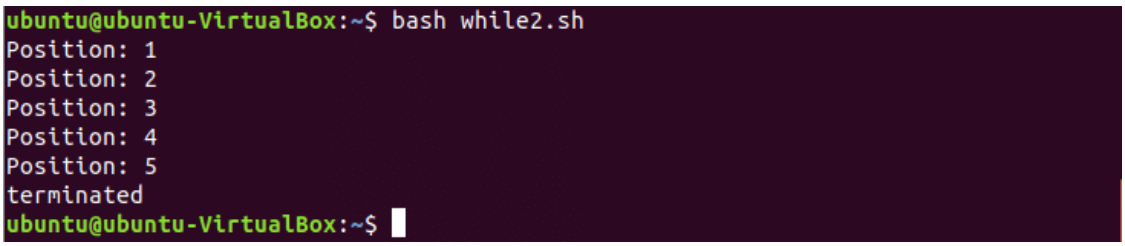
Приклад 3.20. Використання оператора ***break*** для умовного виходу.

Оператор *break* використовують для раннього виходу із циклу на основі певної умови. Створюють новий *bash*-файл з іменем *while2.sh* із таким кодом:

```
n=1
while [$n -le 10]
do
   if [$n==6]
   then
```

```
        echo "terminated"
        break
    fi
    echo "Position: $n"
     ((n++))
Done
```

У цьому прикладі цикл передбачається повторити 10 разів, але відповідно до сценарію він припиняється після 6-ї ітерації із заявою про перерву. Наступне виведення з'явиться після виконання сценарію:



Приклад 3.21. Створіть *bash*-файл з іменем "*while_example.sh*" для розуміння використання циклу *while*. У прикладі, поки цикл буде повторюватися п'ять разів, значення змінної підрахунку збільшуватиметься на одиницю на кожному кроці. Коли значення змінної *count* стане 5, цикл *while* припиниться:

```
#!/bin/bash
valid=true
count=1
while [ $valid ]
do
    echo $count
    if [ $count -eq 5 ];
       then
          break
    fi
       ((count++))
done
```

Виконайте файл *bash*-командою **`$ bash while_example.sh`**:

```
ubuntu@ubuntu-VirtualBox:~/code$ bash while_example.sh
1
2
3
4
5
ubuntu@ubuntu-VirtualBox:~/code$
```

3.4.4. Оператор циклу for

Оператор циклу *for* призначений для обробки в циклі низки значень, які можуть являти собою будь-яку множину рядків. Рядки можуть бути перераховані у програмі або являти собою результат виконаної командною оболонкою підстановки імен файлів. Синтаксис оператора циклу:

```
for змінна in значення
do
оператори
done
```

Приклад 3.22.

```
#!/bin/bash
for var in first "the second" "the third" "I'll do it"
do
echo "This is: $var"
done
```

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
This is: first
This is: the second
This is: the third
This is: I'll do it
```

Приклад 3.23. Вивести на екран усі імена файлів сценаріїв у поточному каталозі, що починаються з літери "f", та імена всіх сценаріїв, які закінчуються символами *.sh*. Це можна зробити таким чином:

```
#!/bin/sh
for file in $ (ls f*.sh); do
echo $ file
done
echo $ file
exit 0
```

3.4.5. Передача параметрів у командному рядку

Найбільш поширений спосіб передавання даних сценарію полягає у використанні параметрів командного рядка. Викликаючи сценарій із параметрами, можна передати йому інформацію, з якою він може працювати. Наприклад: $ ./myscript 10 20

Сценарію передано два параметри: «10» і «20».

Оболонка *bash* призначає спеціальним змінним, які називають ***позиційними параметрами***, введені під час виклику скрипта параметри командного рядка: $0 – ім'я скрипта, $1 – перший параметр, $2 – другий параметр – і так далі, аж до змінної $9, в яку потрапляє дев'ятий параметр.

Приклад 3.24.

```
#!/bin/bash
exo $0
exo $1
exo $2
exo $3
```

Запускаємо сценарій з параметрами, які розділяються пробілами:

```
./myscript 5 10 15
```

Результат:

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript 5 10 15
./myscript
5
10
15
```

Приклад 3.25. Знайдемо суму чисел, переданих сценарію:

```
#!/bin/bash
total=$[$1 + $2]
echo The first parameter is $1.
echo The second parameter is $2.
echo The sum is $total.
```

Результат обчислень:

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript 5 10
The first parameter is 5.
The second parameter is 10.
The sum is 15.
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

3.4.6. Функції

Періодично виникає потреба використовувати одні й ті самі фрагменти коду. Оболонка *bash* надає таку можливість, дозволяючи створювати функції. Функції *bash* – це іменовані блоки коду, які можна повторно використовувати у скриптах.

Синтаксис оголошення функції:

```
function <ім'я> ()
{
<список>;
}
```

Функцію визначають іменем <ім'я>. Тіло функції – <список> записують між дужками { }. Функцію можна викликати без аргументів та з аргументами.

Приклад 3.26. Скрипт, що містить оголошення функції та використовує її:

```
#!/bin/bash
function myfunc {
echo "This is an example of using a function"
}
count=1
while [$count -le 3] do
myfunc
count=$(($count + 1))
done
echo "This is the end of the loop"
myfunc
echo "End of the script"
```

Створено функцію з іменем *myfunc*. Для виклику функції треба вказати її ім'я:

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript.sh
This is an example of using a function
This is an example of using a function
This is an example of using a function
This is the end of the loop
This is an example of using a function
End of the script
```

Звертаємо увагу на те, що не можна використовувати функцію до її оголошення.

Наприклад, такий скрипт є помилковим:

```
#!/bin/bash
count=1
while [$count -le 3] do
myfunc
count=$(($count + 1))
done
echo "This is the end of the loop"
function myfunc {
echo "This is an example of using a function"
}
echo "End of the script"
```

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript.sh
./myscript.sh: line 5: myfunc: command not found
./myscript.sh: line 5: myfunc: command not found
./myscript.sh: line 5: myfunc: command not found
This is the end of the loop
End of the script
```

Використання команди return

Команда *return* дозволяє задавати цілочисельний код завершення, який повертається функцією. Для цього є два способи.

***Перший спосіб. Використання змінної* $:**

```
#!/bin/bash
function myfunc {
read -p "Enter a value:" value
echo "adding value"
return $(($ value + 10))
}
myfunc
echo "The new value is $?"
```

Уведено число 10, команда *echo* вивела суму введеного числа та числа 10:

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript.sh
Enter a value: 10
adding value
The new value is 20
```

Але є обмеження – максимальне число, яке може повернути команда *return* – 255. Функція *myfunc* повертає результат, використовуючи команду *return*.

Другий спосіб. Запис виведення функції у змінну. Такий підхід дозволяє обійти обмеження команди *return* і повертати з функції будь-які дані. Розглянемо приклад:

```
#!/bin/bash
function myfunc {
read -p "Enter a value:" value
echo $(($ value + 10))
}
result=$(myfunc)
echo "The value is $ result"
```

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript.sh
Enter a value: 10
The value is 20
```

Функція також можна викликати з аргументами, які записують після її імені, наприклад: *myfunc $val1 10 20*

Наведемо приклад, в якому функція, викликана з аргументами, їх обробляє:

```
#!/bin/bash
function addnum {
if [$# -eq 0] || [$# -gt 2]
then echo -1
elif [$# -eq 1]
then echo $(($1 + $1))
else echo $(($1 + $2))
fi
}
echo -n "Adding 10 and 15:"
value=$(addnum 10 15)
echo $value
echo -n "Adding one number:"
value=$(addnum 10)
echo $value
echo -n "Adding no numbers:"
value=$(addnum)
```

```
echo $value
echo -n "Adding three numbers:"
value=$(addnum 10 15 20)
echo $value
```

Функція *addnum* перевіряє кількість переданих їй аргументів під час виклику зі скрипта. Якщо аргументів немає або їх більше двох, функція повертає значення -1. Якщо параметр всього один, вона додає його до нього самого й повертає результат. Якщо параметрів два, функція складає їх:

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript.sh
Adding 10 and 15: 25
Adding one number: 20
Adding no numbers: -1
Adding three numbers: -1
```

Між тим функція не може безпосередньо працювати з параметрами, які передані скрипту під час його запуску з командного рядка. Наприклад, напишемо такий сценарій:

```
#!/bin/bash
function myfunc {
echo $(($1 + $2))
}
if [$# -eq 2]
then value=$(myfunc)
echo "The result is $value"
else echo "Usage: myfunc a b"
fi
```

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript.sh 10 20
./myscript.sh: line 3: +  : syntax error: operand expected (error token is "+  "
)
The result is
```

Функція не може безпосередньо використовувати параметри, передані сценарієм, тому якщо у функції планується використовувати параметри, передані скрипту при виклику з командного рядка, треба передати їх їй під час виклику:

```
#!/bin/bash
function myfunc {
echo $(($1 + $2))
```

```
}
if [$# -eq 2]
then value=$(myfunc $1 $2)
echo "The result is $ value"
else echo "Usage: myfunc a b"
fi
```

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript.sh 10 20
The result is 30
```

Робота зі змінними у функціях

Змінні, якими користуються у сценаріях, характеризуються областю видимості. Змінні, оголошені всередині функцій, ведуть себе не так, як ті змінні, які вже було розглянуто – вони можуть бути приховані від інших частин скриптів. Є два види змінних: *глобальні та локальні*.

Глобальні змінні зазвичай видно з будь-якого місця *bash*-скрипта. Якщо оголосили глобальну змінну в основному коді скрипта, до такої змінної можна звернутися з функції. Майже те саме справедливо і для глобальних змінних, оголошених у функціях. Звертатися до них можна і в основному коді скрипта після виклику функцій. *За замовчуванням всі оголошені у скриптах змінні глобальні*.

Так, до змінних, оголошених за межами функцій, можна без проблем звертатися з функцій:

```
#!/bin/bash
function myfunc {
value=$(($value + 10))
}
read -p "Enter a value:" value
myfunc
echo "The new value is: $value"
```

Звернення до глобальної змінної із функції:

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript.sh
Enter a value: 10
The new value is: 20
```

Таким чином, коли змінній присвоюється нове значення у функції, це нове значення не губиться, коли скрипт звертається до неї після завершення роботи функції. Саме це наведено у прикладі. Тому виникає питання, що ж тут не влаштовує? Відповідь проста – треба використовувати локальні змінні.

Локальні змінні. Змінні, які оголошують і використовують всередині функції, можуть бути оголошені локальними. Для цього використовують ключове слово ***local*** перед іменем змінної: **local temp=$(($value + 5))**

Ключове слово *local* дозволяє відокремити змінні, використовувані всередині функції, від інших змінних.

Приклад 3.27.

```
#!/bin/bash
function myfunc {
local temp=$[$value + 5]
echo "The Temp from inside function is $temp"
}
temp=4
myfunc
echo "The temp from outside is $temp"
```

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript.sh
The Temp from inside function is 5
The temp from outside is 4
```

Локальна змінна у функції. Коли користувач працює зі змінною *$temp* всередині функції, це не впливає на значення, призначене змінній з таким самим іменем за її межами.

3.4.7. Передавання функції масиву як аргумента

Наведена нижче конструкція не буде працювати:

```
#!/bin/bash
function myfunc {
echo "The parameters are: $@"
arr=$1
echo "The received array is ${arr[*]}"
}
myarray=(1 2 3 4 5)
```

```
echo "The original array is: ${myarray[*]}"
myfunc $myarray
```

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript.sh
The original array is: 1 2 3 4 5
The parameters are: 1
The received array is 1
```

Під час передавання функції масиву вона отримає доступ лише до його першого елемента. Щоб цю проблему вирішити, з масиву треба витягти наявні в ньому дані й передати їх функції як самостійні аргументи. Якщо треба, всередині функції отримані аргументи можна знову зібрати у масив:

```
#!/bin/bash
function myfunc {
local newarray
newarray=("$@")
echo "The new array value is: ${newarray[*]}"
}
myarray=(1 2 3 4 5)
echo "The original array is ${myarray[*]}"
myfunc ${myarray[*]}
```

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript.sh
The original array is 1 2 3 4 5
The new array value is: 1 2 3 4 5
```

Таким чином, функція зібрала масив з переданих їй аргументів.

3.4.8. Рекурсивні функції

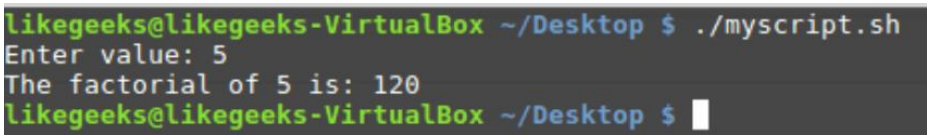
Рекурсія – це коли функція викликає сама себе. Класичний приклад рекурсії – функція для обчислення факторіала. Факторіал числа – це добуток усіх натуральних чисел від 1 до цього числа. Наприклад, факторіал 5 можна знайти так: 5! = 1 * 2 * 3 * 4 * 5

Якщо формулу обчислення факторіала написати у рекурсивному вигляді, вийде таке: x!=X * (x-1)!

Цією формулою можна скористатися, щоб написати рекурсивну функцію [16]:

```
#!/bin/bash
```

```
function factorial {
if [$1 -eq 1]
then echo 1
else
local temp=$(($1 - 1))
local result=$(factorial $temp)
echo $(($result * $1))
fi
}
read -p "Enter value:" value
result=$(factorial $value)
echo "The factorial of $value is: $result"
```



КОНТРОЛЬНІ ЗАПИТАННЯ

1. Які основні завдання файлової системи?
2. Який стандарт застосовують під час організації файлової системи Linux? Які його особливості?
3. Які файлові системи використовують у Linux?
4. Що таке абсолютні й відносні шляхи до файлу?
5. Які типи файлів існують в Linux та як їх кодують?
6. Що таке жорстке посилання?
7. Що таке символічне посилання?
8. Що таке канали, для чого їх використовують?
9. Які файлові менеджери вам відомі?
10. Які можливості файлового менеджера Midnight Commander?
11. Що таке власник файлу?
12. Що таке командна оболонка і які функції вона виконує?
13. Які найпоширеніші командні оболонки вам відомі? Де вони зберігаються у системі?
14. Що таке привілейований та обліковий запис користувача?

15. Що таке групи користувачів? Яке їх призначення?

16. Яке основне призначення користувацьких і групових ідентифікаторів?

17. Як визначити права для власника файлу, групи та всіх інших?

18. Для чого використовують команду chmod?

19. Яке призначення утиліти sudo?

20. Як змінити права власності на файл?

21. Які найбільш поширені коди доступу до файлів?

22. Які команди для роботи з каталогами вам відомі?

23. Як знайти файл?

24. Які можливості має система контролю версій ПЗ Git?

25. Які призначення та функції утиліти grep?

26. Що таке регулярні вирази?

27. Яке призначення має квантифікатор?

28. У чому полягає сутність моделі «введення – виведення» UNIX?

29. Як користуватися системним викликом open()?

30. Як користуватися системним викликом creat()?

31. Як користуватися системним викликом read()?

32. Як користуватися системним викликом write()?

33. Як користуватися системним викликом lseek()?

34. Що таке скрипт?

35. Як створити виконуваний файл?

36. Як під час написання скриптів здійснюється розгалуження будь-яких процесів?

37. Як передають параметри в командному рядку?

38. Як створити функцію?

39. Як здійснюють передачавання масиву як аргумента функції?

40. Що таке рекурсія?

Розділ 4

КЕРУВАННЯ ПРОЦЕСАМИ

4.1. Процеси та їх характеристики

4.1.1. Характеристики процесів

Запуск нового процесу

Слід чітко розуміти відмінності між процесом і програмою.

Програма – це файл, який містить виконуваний код, сегменти з даними для ініціалізації та з даними користувача.

Процес – це одна з найбільш фундаментальних абстракцій у системах UNIX після файлів. Процес як об'єкт виконуваного коду – це екземпляр активної програми, що виконується під керуванням ОС.

Процес – це *середовище виконання*, яке містить сегмент виконуваного коду, сегменти даних користувача і системних даних, а також набір додаткових ресурсів, отриманих під час виконання [1, 2, 8].

В UNIX дія завантаження у пам'ять і запуск образу програми виконується окремо від операції зі створення нового процесу. Один системний виклик завантажує бінарну програму в пам'ять та заміщає поточний діапазон адресного простору й починає виконання нової програми. Це називають *виконанням нової програми*, а функціональність забезпечується сімейством викликів *exec*. Тобто системний виклик *exec* – *єдиний спосіб запуску програм в Unix*.

Інший системний виклик ***fork()*** – єдиний спосіб запустити новий процес, його використовують для створення нового процесу, який спочатку є фактично копією свого батьківського [10]. Часто новий процес відразу починає виконання нової програми. *Створення нового процесу* називають *розгалуженням*, він забезпечується системним викликом *fork()*.

Відбуваються дві дії – спочатку розгалуження для створення нового процесу, а потім *exec* для запуску нової програми в новому процесі.

Завдання – це набір інструкцій, завантажений у пам'ять комп'ютера.

Кожний процес працює у власному віртуальному адресному просторі, отримати доступ до адресного простору іншого процесу можуть тільки деякі системні процеси. Користувацькі процеси можуть взаємодіяти з ядром за допомогою механізму системних викликів.

Потік – це *одна з дій всередині процесу*. Кожний потік має власний віртуалізований процесор, що містить стек, стан процесора, наприклад регістри, а також командні вказівники.

Якщо у процесу лише один потік, то процес і є потоком. У нього тільки один екземпляр віртуальної пам'яті та один віртуалізований процесор. У багатопотокових процесах потоків декілька. Віртуалізація пам'яті пов'язана із процесом, тому всі потоки одночасно використовують один і той самий адресний простір пам'яті.

Ядро Linux – основна програма, яка забезпечує взаємодію користувача програм та обладнання. Ядро ОС обробляє переривання від пристроїв, виконує запити системних процесів та додатків користувача, розподіляє віртуальну пам'ять, створює і знищує процеси, забезпечує багатозадачність за допомогою перемикання між ними, містить драйвери пристроїв, обслуговує файлову систему.

Файл ядра в *Ubuntu* зберігається у папці ***/boot***, його називають ***vmlinuz-номер_версії***.

Системні виклики – це функції, які реалізовані в ядрі ОС. Процеси ядра працюють у просторі ядра (*kernel space*). Додатки працюють у просторі користувача (*user space*), а системні виклики виконуються у просторі ядра. Програми користувачів, використовуючи системні виклики, звертаються до ядра системи, а ядро виділяє необхідні системні ресурси.

У Linux пам’ять, використовувану процесом, можна поділити на віртуальну і реальну (фізичну). *Фізична пам’ять* – реально використовувана процесом частина оперативної пам’яті. *Віртуальна пам’ять* процесу містить виконуваний код і дані, зокрема під час обробки нових даних можуть виділятися нові ділянки віртуальної пам’яті, які зв’язуються із початковою. Під час використання поділюваних бібліотек кожний процес Linux не зберігає свою копію бібліотеки в оперативній пам’яті. Замість цього код і дані процесу пов’язуються у віртуальних адресних просторах цього процесу та інших процесів, які спільно використовують бібліотеку. Ділянка віртуальної пам’яті процесу копіюється у фізичну пам’ять тільки тоді, коли процес намагається її використовувати. Linux створює таблицю сторінок пам’яті процесу, віртуальні області – це окремі сторінки, які не завантажені в оперативну пам’ять. Коли процес намагається отримати доступ до даних або коду, що зберігається тільки у віртуальній пам’яті, а не у фізичній, то виникає помилка доступу і спрацьовує механізм завантаження сторінок віртуальної пам’яті у фізичну.

Образ процесу в оперативній пам’яті складається:

– з даних користувача,

– програми користувача,

– системного стеку (простору ядра),

– блока керування процесом.

З погляду ядра процес являє собою запис у ***таблиці процесів***. Цей запис містить відомості про стан процесу й дані, що існують протягом усього часу його життя. Розмір таблиці процесів дозволяє запускати кілька сотень процесів одночасно. Процес також використовує таблицю всіх відкритих процесом файлів, які зберігаються в його адресному просторі. *Запис у таблиці процесів і простір процесу разом складають контекст, або оточення, процесу*.

Контекст кожного процесу *містить*:

– ідентифікатор процесу (***pid*** – *process ID*): унікальний номер для позначення процесу;

– ідентифікатор батьківського процесу (***ppid*** – *parent process identifier*), UID- та GID-ідентифікатори прав процесу (*UID – user identifier, ідентифікатор користувача, а GID – group identifier, ідентифікатор групи процесів*);

– пріоритет процесу користувача (число ***nice*** – пріоритет процесу або показник ввічливості);

– термінал (***tty***) – термінал, до якого запущений процес;

– статус/стан процесу ***stat*** (R=*Running* – виконується; S=*Sleeping* – у стані очікування; D=*Direct* – процес очікує певного сигналу виключно від апаратної частини; T=*Tracing* – процес перебуває у режимі «тросування / налагодження програми»; Z=*Zombie* – у стані зомбі, тобто процес закінчився, але з деяких причин не звільнений з ядра; «<» – підвищений пріоритет; «+» – перебуває в інтерактивному режимі);

– таблиця відкритих (використовуваних) файлів процесу;

– змінні оточення [9, 11, 13].

Ідентифікатор процесу

Кожний процес позначається унікальним ідентифікатором (*process ID*, зазвичай скорочується до **pid**). Pid обов’язково є унікальним у будь-який *конкретний момент часу*. Це означає, що в момент часу t + 0 може бути тільки один процес із pid 770 (або жодного процесу з таким значенням ідентифікатора), але не можна гарантувати, що в момент часу t + 1 не існуватиме іншого процесу з тим же ідентифікатором pid 770. На практиці ядро не обов’язково швидко змінює місцями ідентифікатори процесів, тому що це вельми небезпечно. З точки зору самого процесу його pid залишається незмінним.

Процес *бездіяльності* або *пасивний*, який виконується ядром, коли немає всіх інших процесів, має *pid 0*. Перший процес, який ядро виконує

під час запуску системи, називають *процесом ініціалізації*, він має *pid 1*. Зазвичай *init process* в Linux є програмою ініціалізації. Термін «ініціалізація» використовують для позначення і початкового процесу, що запускається під час завантаження, і спеціальної програми, яка використовується для цих цілей.

Якщо користувач не вказує ядру напряму, який процес слід запускати (за допомогою *init* у командному рядку), ядро вибирає відповідний процес ініціалізації самостійно. Ядро Linux перебирає чотири виконувані модулі у такій послідовності:

1. */sbin/init* – найбільш ймовірне розміщення процесу ініціалізації.

2. */etc/init* – наступне найбільш ймовірне розміщення процесу ініціалізації.

3. */bin/init* – резервне розміщення процесу ініціалізації.

4. */bin/sh* – місцезнаходження оболонки *Bourne* – оболонка Стіва Борна, яку ядро намагається запустити у випадку, коли знайти процес ініціалізації не вдалося.

Перший із цих процесів, який буде знайдений, і запуститься як процес ініціалізації. Якщо не вдалося запустити жодного з чотирьох виконуваних модулів, то ядро Linux переводить систему у стан «*паніки*». Після запуску процес ініціалізації відпрацьовує решту завантаження. Зазвичай у неї включені ініціалізація системи, запуск різних сервісів та програми авторизації.

Для кожного процесу створюється свій каталог у псевдофайловій системі ***/proc***, в якій можна переглянути атрибути процесу.

Переглянути вміст каталогу можна за командою ***ls /proc.***

Псевдофайлова система /proc відображає дерево процесів у вигляді дерева директорій, що мають назву, яка відповідає *pid* процесу. Для деяких процесів ядра можна змінювати параметри. У кожній директорії є такі файли:

• *cmdline* – командний рядок процесу;

• *status* – детальна інформація про статус процесу;

• *fd* – моніторинг файлів, відкритих процесом;

• *cwd* – посилання на поточний робочий каталог процесу;

• *environ* – оточення.

Механізм породження процесів

Для дублювання процесу використовують системний виклик *fork()*. У дочірньому процесі зазвичай здійснюється системний виклик *exec()*, щоб запустити окремий, незалежний процес, а в батьківському – системний виклик *wait()*, щоб чекати сигналу завершення від дочірнього процесу. У кожного процесу є рівно один батько, але може бути кілька нащадків, або дочірніх процесів, тому систему процесів зручно зображати у вигляді дерева. Якщо батьківський процес завершується раніше дочірнього, то процес-сирота (*orphaned process*) успадковується процесом *init (pid=1)*. Якщо дочірній процес завершується раніше, ніж батьківський процес встиг викликати *wait()*, то дочірній позначається як *zombie*, буква Z у статусі процесу (або «*defunct*» у кінці імені процесу).

Виділення ідентифікатора процесу

Кількість процесів у системі є обмеженою. Дізнатися максимально підтримувану кількість процесів для запущеної ОС Linux можна за командою:

cat /proc/sys/ernel /pid_max

За специфікацією Linux 4 194 304 – межа для архітектури x86_64, а 32767 – для архітектури x86 (для забезпечення сумісності з більш старими UNIX-системами, 16-бітними). Максимальне значення *pid* при цьому на одиницю менше. Кількість процесів у системі обмежена. Адміністратор системи може встановити більшу величину через

/*proc/sys/kernel/pid_max*, виділивши більший простір для *pid*, але знизивши таким чином сумісність.

Ідентифікатори призначаються ядром суворо лінійно. Якщо в якийсь момент найбільший наявний *pid* дорівнює 17, то наступною буде призначена величина 18, навіть якщо процес з останнім призначеним ідентифікатором 17 вже не виконується під час старту нового процесу. Ядро не призначить використані раніше ідентифікатори процесів, доки не пройде верхнього значення, тобто менші значення не будуть встановлюватися, поки не буде досягнута величина, записана в */proc/sys/kernel/pid_max*. Таким чином, Linux не гарантує унікальності ідентифікаторів процесів на довгі періоди, але тим не менше є деяка впевненість, що протягом коротких відрізків часу ідентифікатори будуть стабільними та унікальними.

Розгляемо основні **категорії процесів**:

• *Процеси ядра*. Виконуються у просторі ядра, розміщуються в оперативній пам’яті й не мають відповідних програм у вигляді виконуваних файлів. Відповідний код міститься в модулях ядра. Приклад – процес упорядкування процесів у черзі (*scheduling*).

• *Фонові неінтерактивні процеси*. Мають відповідні виконувані файли. Призначені для обслуговування з’єднань з будь-якого мережевого протоколу або регулярного виконання рутинних операцій у системі.

• *Прикладні процеси*. Усі інші процеси, які породжуються в межах сеансу роботи користувача.

Процеси можна запускати в *інтерактивному режимі* й у *фоновому режимі*. В інтерактивному режимі може працювати тільки один процес. У фоновому режимі процес не підключений до потоків «введення – виведення» поточного термінала.

Кожний процес може містити кілька потоків, які виконуються паралельно. Потік вимагає менших витрат на породження, і, крім того,

поділяє спільні дані з іншими потоками – має один адресний простір з ними. Потоки можуть здійснюватися при цьому на різних ядрах процесора, але в межах одного пристрою. Водночас процеси можуть взаємодіяти один з одним, перебуваючи навіть під керуванням різних ОС, через уніфіковані механізми міжпроцесного обміну. Відмінності процесу від потоку наведено у табл. 4.1.

Таблиця 4.1

Відмінності процесу і потоку

Процес	**Потік**
Незалежний	Є частиною процесу
Має власний адресний простір	Поділяє один адресний простір з іншими потоками процесу
Має свій стан (змінні)	У межах процесу використовують загальний стан (змінні)
Взаємодіють через спеціальні механізми зв'язку	Швидко взаємодіє з іншими потоками процесу

Ієрархія процесів

Процес, що запускає інший процес, називають *батьківським*; новий процес, таким чином, є *дочірнім*. Кожний процес запускається будь-яким іншим способом (окрім процесів ініціалізації). Таким чином, кожний дочірній процес має *батька*. Ці взаємини записані в кожному ідентифікаторі батьківського процесу (***ppid)***, значення якого для дочірнього процесу дорівнює значенню *pid* батьківського процесу.

Кожний процес належить певному користувачеві та групі. Ці приналежності використовують для керування правами доступу до ресурсів. З точки зору ядра *користувач і група – це просто якісь цілочисельні величини*. Вони зберігаються у файлах ***/etc/passwd*** та ***/etc/group***, які зіставляються з іменами користувачів UNIX (наприклад, *root*) та іменами груп (наприклад, *wheel*), але в ядрі – це числа. Кожний дочірній процес успадковує користувача і групу, яким належав батьківський процес [8, 10].

Кожний процес є також частиною *групи процесів*, що означає його відношення до інших процесів (не слід плутати групу процесів зі згаданими вище користувачем і групою). Дочірні процеси, як правило, належать до тих самих груп процесів, що й батьківські. Крім того, коли користувач запускає конвеєр (наприклад, увівши *ls | less*), усі команди в конвеєрі стають членами однієї і тієї самої групи процесів. (Нагадуємо, що конвеєри – це або відомі користувачам оболонки у вигляді оператора «|», або FIFO-буфери, які можуть застосовуватися для передавання даних між процесами).

Поняття групи процесів спрощує надсилання сигналів або отримання інформації від усього конвеєра, оскільки всі дочірні процеси перебувають у конвеєрі. З погляду користувача група процесів тісно пов'язана з поняттям *завдання*. ***Завдання*** – це набір інструкцій, завантажених у пам'ять комп'ютера.

4.1.2. Системні виклики для роботи з процесами

pid_t.

З погляду програмування ідентифікатор процесу позначається типом *pid_t*, величина якого визначається в заголовках <*sys/types.h*>. У Linux для *pid_t* найчастіше використовують тип даних *C int*.

Отримання ідентифікаторів процесу getpid()
і батьківського процесу getppid()

Системний виклик *getpid()* повертає ідентифікатор процесу, який викликається:

```
#include <sys/types.h>
#include <unistd.h>
pid_t  getpid(void);
```

Системний виклик *getppid()* повертає ідентифікатор батьківського процесу, що викликається:

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

```
#include <unistd.h>
pid_t  getppid(void);
```

Жодний з них не може повернути помилку, тому використання цих викликів нескладне:

```
printf("My pid=%jd\n",  (intmax_t)  getpid());
printf("Parent's  pid=%jd\n",  (intmax_t)  getppid());
```

У прикладі наводимо величину, що повертається, яка належить до типу *intmax_t*. Це тип C/C ++, який гарантовано здатний зберігати в системі будь-яке ціле число зі знаком. Тобто він дорівнює будь-яким іншим типам для цілих чисел зі знаками або більший від них. У поєднанні з *printf()* із модифікатором *(%j*) такий підхід дозволяє безпечно друкувати змінні типу *integer*, призначені *typedef* (надання змінній іншого типу).

Привілейовані процеси

Традиційно в системах UNIX *привілейованим* вважають процес, чий *діючий ідентифікатор користувача має значення 0* (привілейований користувач, суперкористувач). Такий процес не має обмеження прав доступу, які зазвичай використовуються ядром. І навпаки, *непривілейованим називають процес, запущений іншими користувачами.* Такі процеси мають ненульовий діючий UID й мають дотримуватися усіх правил дозволу доступу, яких вимагає ядро.

Процес може бути привілейованим через те, що був створений іншим привілейованим процесом, наприклад оболонкою входу в систему, яку запустив суперкористувач (*root*). Ще один спосіб отримання процесом привілейованості пов’язаний з механізмом установки ідентифікатора користувача (*set-user-ID*), який дозволяє присвоїти процесу такий же діючий ідентифікатор користувача, як і ідентифікатор користувача файлу виконуваної програми.

Сімейство викликів exec

Насправді єдиного системного виклику *exec* не існує; це ціле сімейство таких викликів. Спочатку розглянемо найпростіший із цих викликів, *execl()*:

```
#include <unistd.h>
int execl(const char *path, const char *arg, ... NULL);
```

Виклик *execl()* заміщає поточний образ процесу новим, завантажуючи в пам'ять програму, визначену *path* (*повний шлях до програми*). Параметр *arg* – перший аргумент цієї програми (*ім'я файлу*). Три крапки означають змінну кількість аргументів – у функції *execl()* їх кількість може бути будь-яка, додаткові аргументи можна вказувати в дужках один за іншим. Список аргументів завжди завершується значенням NULL (порожній вказівник, який завершує список) [10].

Аргумент *path* повинен містити повне ім'я виконуваного файлу з програмою діючого ідентифікатора користувача (наприклад, із правами 755). Виконуваний код процесу буде затертий виконуваним кодом нової програми, сегмент даних також буде затертий сегментом даних нової програми, а стек буде переініціалізований. Виконання нової програми почнеться з від початку.

Наприклад, наступний програмний код заміщає програму, яка у цей момент виконується з */bin/vi*:

```
int ret;
ret=execl("/bin/vi", "vi", NULL);
if (ret==-1)
perror("execl");
```

Як перший аргумент програми передаємо значення "vi". Оболонка вміщує останній компонент шляху, тобто "vi", у перший аргумент під час розгалуження або запуску процесів, завдяки чому програма може перевірити перший аргумент *argv[0]* для з'ясування імені двійкового образу. У більшості випадків кілька системних утиліт, що відображаються користувачу під різними іменами, насправді є єдиною програмою з

жорстко прописаними посиланнями до різних імен. Програма використовує перший аргумент, щоб визначити свою поведінку.

Інший приклад. Якщо треба редагувати файл */home/kidd/hooks.txt*, то запускають такий код:

```
int ret;
ret=execl("/bin/vi", "vi", "/home/kidd/hooks.txt", NULL);
if (ret==–1)
      perror ("execl");
```

Зазвичай *execl()* не повертає ніяких значень. Успішний виклик завершується переходом до вхідної точки нової програми, і тільки що виконаний код більше не перебуває у процесному адресному просторі. Якщо сталася помилка, *execl()* повертає -1 і встановлює *errno* для позначення проблеми.

У разі успішного виконання виклик *execl()* змінює не лише адресний простір та образ процесу, а й деякі інші атрибути процесу:

– будь-які очікують сигнали зникають;

– будь-які сигнали, які відловлюються процесом, повертаються до своєї поведінки за замовчуванням, оскільки оброблювачів сигналів більше немає у процесному адресному просторі;

– усі блокування пам’яті видаляються;

– більшість атрибутів потоку повертається до значень за замовчуванням;

– більша частина статистичних даних процесу скидається;

– весь адресний простір пам’яті, що належить до певного процесу, включаючи завантажені файли, очищується;

– все, що міститься виключно у просторі користувача, включаючи функціональності бібліотеки C, наприклад поведінка atexit(), видаляється.

Деякі властивості процесу, проте, *не* змінюються. Наприклад, ідентифікатор (свій і батьківський), пріоритет, а також користувач та група залишаються тими самими. Зазвичай під час роботи системних викликів

сімейства *exec* успадковуються і файли відкриті: у запущених програм зберігається повний доступ до всіх файлів, відкритих у первісному процесі, якщо їм відомі значення дескрипторів. Однак найчастіше це небажано. Звичайною практикою є закриття файлів перед запуском *exec*, але можна дати команду ядру робити це автоматично через *fcntl()*.

Інша частина сімейства

Крім *execl()*, у сімействі є ще п'ять членів – ***execv***, ***execlp***, ***execvp***, ***execle*** та ***execve***. Ці п'ять викликів сімейства *exec* надають три додаткові можливості, недоступні в *execl*:

– передати аргументи у вигляді масиву, а не у вигляді списку. Це потрібно, коли кількість аргументів на момент написання програми заздалегідь невідома;

– пошук файлу програми з використанням значення змінної оточення PATH, як це робить командна оболонка;

– передавати вказівник на сформоване вручну середовище оточення замість неявного передавання вказівника *environ*:

```
#include <unistd.h>
int  execlp(const  char  *file, const  char  *arg, ...);
int execle(const  char  *path, const  char  *arg, ...,  char  *const envp[]);
int execv(const  char  *path, char  *const  arg[]);
int execvp(const  char  *file, char * const argv[]);
int execve(const  char  *filename, char  *const argv[], char  *const envp[]);
```

/***execlp*** – запускає програму (*file – ім'я файлу з програмою), аргументи передаються у вигляді списку (*arg0 – перший аргумент – ім'я файлу), пошук файлу ведеться з використанням змінної PATH */

/***execle*** – запускає програму – **path* (шлях до файлу з програмою), аргументи передаються у вигляді списку, **arg0 – перший аргумент – це ім'я файлу,* також передається середовище оточення*/

/***execv*** – запускає програму (**path* – повний шлях до файлу з програмою), аргументи передаються у вигляді масиву*/

/***execlp*** – запускає програму (**file* – ім'я файлу з програмою), аргументи передаються у вигляді списку (**arg0 – перший аргумент – це ім'я файлу)*, пошук файлу ведеться з використанням змінної PATH*/

/***execve*** – запускає програму (*path – повний шлях до файлу з програмою), аргументи передаються у вигляді списку, так само передається середовище оточення *envp[]).*

Позначення у назвах сімейства *exec()* означають таке:

l – вказує, що аргументи передаються списком,

v – аргументи передаються масивом,

p – пошук файлу програми виконується за допомогою змінної *path,* повним шляхом користувача. У командах, де використовують варіанти з ***p,*** можна вказати тільки ім'я файлу, якщо він перебуває в межах користувацького шляху,

e – означає, що для нового процесу створюється нове специфічне середовище оточення.

Варіант *p* призначений для використання оболонками, а процеси, що виконуються в оболонках, як правило, успадковують своє оточення від них.

execvp() запускає програму *(*file*), аргументи передаються у вигляді масиву (*argv[]*).

У наступному фрагменті коду використано *execvp()* для виконання *vi*, як і в попередньому варіанті, враховуючи, що *vi* перебуває на користувацькому шляху:

```
int ret;
ret=execvp("vi", "vi", "/home/kidd/hooks.txt", NULL);
if (ret==-1)
perror( execvp");
```

Програми установки ідентифікаторів групи і користувача (*set-group-ID* і *set-user-ID*) – це процеси, що запускаються від імені групи або користувача, яким належить їх бінарний код, а не групи або користувача, які їх викликають. Отже, вони не повинні посилатися на оболонку або

операції, які, у свою чергу, впливають на оболонку. Ця дія призведе до появи вразливого місця в безпеці, оскільки користувач, який робить виклик, може встановити змінні оточення й керувати оболонкою. Найчастіша форма такої атаки – впровадження шляху, коли атакуючий вказує змінній PATH змусити процес виконати функцію *execl()* й запустити будь-який двійковий код за вибором зловмисника, у результаті чого останній може запустити будь-яку програму під обліковими даними *set-group-ID* та *set-user-ID.*

Системний виклик fork(). Новий процес, який запускає той самий системний образ, що і поточний, може бути створений за допомогою системного виклику *fork()*:

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

У разі успішного звернення до *fork()* створюється новий процес, в усіх відношеннях ідентичний процесу виклику. Обидва процеси виконуються від точки звернення до *fork(),* як ніби нічого не відбувалося.

Новий процес є дочірнім щодо процесу виклику, який, у свою чергу, називають батьківським. У дочірньому процесі успішний запуск *fork()* повертає *0*. У батьківському процесі *fork()* повертає *pid* дочірнього. Батьківський і дочірній процеси майже ідентичні, за винятком деяких особливостей:

– *pid* дочірнього процесу, призначається заново і різниться від батьківського;

– батьківський *pid* дочірнього процесу встановлений рівним *pid* батьківського процесу;

– ресурсна статистика дочірнього процесу обнуляється;

– будь-які очікуючі сигнали перериваються і не успадковуються дочірнім процесом;

– ніякі залучені блокування файлів не успадковуються дочірнім процесом.

У разі помилки дочірній процес не створюється, *fork()* повертає -1, встановлюючи відповідне значення *errno*. Ось два можливих значення *errno* та їхній зміст:

EAGAIN – ядро не здатне виділити певні ресурси, наприклад новий *pid*, або досягнуте обмеження за ресурсами RLIMIT_NPROC;

ENOMEM – недостатньо ресурсів пам'яті ядра, щоб завершити запит.

Використання системного виклику *fork()* дуже просте:

```
pid_t  pid;
pid=fork();
if (pid> 0)
printf("Я батьківський процес pid=%d!\n", pid);
else if (!pid)
printf("А я дочірній!\n");
else if (pid==-1)
perror ( "fork");
```

Найчастіше системний виклик *fork()* використовують для створення нового процесу й наступного завантаження в нього нового двійкового образу. Уявімо собі оболонку, в якій користувач запускає новий додаток або процес починає допоміжну програму. Спочатку процес відгалужує новий процес, а потім дочірній процес створює новий двійковий образ. Поєднання *fork* та *exec* використовують часто. У наступному прикладі відгалужується новий процес, який запускає бінарний файл */bin/windlass*:

```
pid_t  pid;
pid=fork();
if (pid==-1)
perror ("fork");
/*дочірній ... */
if (!pid) {
const  char  *args[]={"windlass", NULL};
int ret;
ret=execv("/bin/windlass", args);
if (ret == -1) {
```

```
perror ("execv");
exit (EXIT_FAILURE);
}
}
```

Батьківський процес продовжує виконуватися, як і раніше, за винятком появи у нього нового дочірнього процесу. Виклик *execv()* впливає тільки на дочірній процес, змушуючи його виконати програму */bin/windlass*.

Завершення процесу exit()

Стандартний виклик завершення поточного процесу такий:

```
#include <stdlib.h>
void  exit(int  status);
```

Виклик *exit()* виконує основні кроки перед завершенням, а потім відправляє ядру команду припинити процес. Ця функція не повертає жодних результатів.

Параметр *status* використовують для позначення статусу процесу завершення. Інші програми – як і користувач оболонки – можуть перевіряти цю величину. Зокрема, статус *status & 0377* повертається батьківському процесу.

Значення EXIT_SUCCESS та EXIT_FAILURE визначають як способи або успіху (значення 0), або невдачі (-1 або 1), тобто успішний вихід – це рядок:

exit EXIT_SUCCESS);

Перш ніж перервати процес бібліотека *C* виконує такі кроки:

1. Виклик усіх функцій, зареєстрованих з *atexit()* або *on_exit()*, у порядку, зворотному порядку реєстрації (пояснення див. нижче).

2. Скидання всіх стандартних потоків «введення – виведення».

Нагадуємо: це (функція fflush())

```
#include <stdio.h>
int fflush (FILE *stream);
```

3. Видалення всіх тимчасових файлів, створених функцією *tmpfile().*

Ці кроки завершують всю роботу, яку процес має виконати у просторі користувача, після чого *exit()* виконує системний виклик _exit(), що дозволяє ядру обробити решту завершення процесу:

```
#include <unistd.h>
void  _exit(int status);
```

Коли процес завершується, ядро очищає всі ресурси, які були виділені для потреб процесу й більш не використовуються. До них належать виділена пам’ять, відкриті файли, семафори *System V* та ін. Після очищення ядро знищує процес і попереджає батьківський процес про завершення дочірнього.

Інші способи завершення

Класичний спосіб припинити роботу програми – невикористання явного системного виклику, а просте «досягнення кінцевої точки» програми. Для мови С це відбувається, коли результат повертає функція *main().* Насправді при «досягненні кінцевої точки» компілятор вставляє неявний системний виклик *exit()* після завершення власного коду. Бажано повертати статус виходу явно, через exit (), або повертаючи яку-небудь величину з *main().* Тобто успішне значення – *exit(0)* або повернення 0 з *main().*

Процес також може завершитися, якщо йому відправлений сигнал, дія якого за замовчуванням – закінчення процесу. До таких сигналів належать SIGTERM та SIGKILL. Останній спосіб перервати виконання програми – спрацьовування захисних функцій ядра. Ядро може перервати процес, що виконує неприпустимі інструкції або порушує сегментацію, або вичерпані ресурси пам’яті тощо.

atexit().

Відмінність між *exit()* та *_exit()* полягає у тому, що *exit()* – це функція з бібліотеки С, а *_exit()* – це системний виклик, який перериває виконання програми зразу й *exitcode* зберігається як код повернення процесу. Коли використовують функцію *exit()*, викликаються функції, які зареєстровані *atexit(),* а потім викликається системний виклик *_exit(exitcode).*

Для реєстрація функцій, які мають бути запущені при виклику *exit(),* тобто в разі нормального завершення процесу, використовують функцію *atexit()*:

```
#include <stdlib.h>
int  atexit(void (*function) (void));
```

*void(*func) (void)* – вказівник на функцію, яка буде виконана під час завершення роботи програми.

Всього може бути зареєстровано до 32 функцій, які будуть виконані під час завершення роботи в порядку, зворотному порядку реєстрації. Також можна кілька разів зареєструвати одну й ту саму функцію. Вона буде викликана стільки разів, скільки разів була зареєстрована. У разі успішного спрацьовування *atexit()* реєструє зазначену функцію для запуску при нормальному завершенні процесу, тобто за допомогою системного виклику exit() або повернення результатів функцією *main().* Якщо процес запускає функцію exec, список зареєстрованих функцій очищається (оскільки функцій більше не існує в новому адресному просторі процесу). Якщо процес переривається сигналом, зареєстровані функції не викликаються. Ця функція не вимагає яких-небудь параметрів і не повертає значень. Прототип виглядає таким чином: *void my_function (void);*

Функції викликаються в порядку, зворотному реєстрації, – це означає, що вони зберігаються в стеку й остання функція, яка увійшла, буде викликана першою (LIFO). Зареєстровані функції не повинні викликати *exit(),* щоб уникнути нескінченної рекурсії. Якщо функція повинна закінчити процес завершення раніше, необхідно викликати *_exit().* Такий підхід не рекомендується, тому що потім можливий збій запуску важливих

завершальних функцій. Стандарт POSIX вимагає підтримки *atexit()* принаймні для ATEXIT_MAX зареєстрованих функцій, отже, ця величина має дорівнювати щонайменше 32. Точний максимум може бути визначений через *sysconf()* та величину _SC_ATEXIT_MAX:

```
long  atexit_max;
atexit_max=sysconf(_SC_ATEXIT_MAX);
printf("atexit_max=%ld\n", atexit_max);
```

У разі успіху atexit() повертає 0, у разі помилки вона повертає значення -1.

Приклад 4.1.

```
#include <stdio.h>
#include <stdlib.h>
void out (void)
{
printf("atexit() succeeded!\n");
}
int  main (void)
{
if (atexit(out))
fprintf(stderr, "atexit() failed!\n");
return 0;
}
```

on_exit()

SunOS 4 (Operation System – Unix-подібна ОС, випущена компанією Sun Microsystems для серверів власного виробництва) визначає власний еквівалент **atexit()**, а бібліотека *glibc* в Linux підтримує його:

```
#include <stdlib.h>
int  on_exit(void (*function)(int,  void *), void  *arg);
```

Ця функція працює так само, як atexit(), але прототип зареєстрованої функції дещо відрізняється: *void my_function(int status, void *arg);*

Аргумент *status* – це величина, передана *exit()* або повернута *main().* Аргумент *arg* – другий параметр, переданий *exit().* Слід упевнитися, що

дані, які містяться в пам’яті *arg*, є допустимими на момент виклику функції.

Остання версія Solaris вже не підтримує цю функцію. Замість неї потрібно використовувати стандартно компільовану *atexit().*

SIGCHLD

Коли процес завершується, ядро надсилає сигнал SIGCHLD батьківському процесу. За замовчуванням цей сигнал ігнорується й батьківський процес не виконує ніяких дій. Однак за потреби процеси можуть обробити цей сигнал за допомогою системних викликів *signal()* або *sigaction().*

Сигнал SIGCHLD може бути згенерований і надісланий у будь-який час, оскільки завершення дочірнього процесу не є синхронним щодо батьківського. Однак часто *батьківському процесу* (*процес-предок – parent process*) потрібні відомості про завершення його *дочірнього процесу* (*процес-нащадок – child process*) або навіть деякий час для очікування події. Батьківський та дочірній процеси виконуються незалежно, але батьківський процес завжди чекає закінчення дочірнього, а не навпаки Це є можливим завдяки допомозі системних викликів, описаних далі.

4.1.3. Очікування завершених дочірніх процесів wait(int *status)

Багатьом батьківським процесам необхідно отримати більше інформації, коли завершується їх дочірній процес – наприклад, якщо той повертає якесь значення.

Якби дочірний процес після завершення повністю зникав, то отримання будь-яких відомостей було б для його батьківського процесу неможливим. Тому перші розробники UNIX вирішили, що коли дочірній процес завершується раніше за батьківський, ядро має помістити дочірній процес в особливий процесний статус. Процес у цьому стані відомий як *зомбі*. У такому стані існує лише «скелет» процесу – деякі основні

структури даних, що містять потенційно потрібні відомості. Процес у такому стані очікує запит про свій статус від предка – батьківського (процедура, відома як *очікування процесу-зомбі*). Тільки після того, як предок отримає всю необхідну інформацію про завершений дочірній процес, дочірній формально видаляється й припиняє існувати навіть у статусі зомбі.

Ядро Linux надає кілька інтерфейсів для отримання інформації щодо завершеного дочірнього процесу. Найпростіший із них, визначений POSIX, називають *wait()*:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t  wait(int  *status);
```

Виклик *wait()* повертає *pid* завершеного дочірнього процесу або -1 у випадку помилки. Якщо ніякого дочірнього процесу не було перервано, виклик блокується, поки нащадок не завершиться. Якщо дочірній процес уже був завершений, виклик повертає результати негайно. Отже, якщо викликати *wait()* відразу після повідомлення про завершення дочірнього процесу, результат буде негайно видано без блокування.

У разі помилки можна присвоїти змінній *errno* одне із двох значень:

ECHILD – викликаючий процес не має дочірніх;

EINTR – сигнал був отриманий під час очікування, у результаті виклик повернув результат занадто рано.

Якщо вказівник *status* не містить значення NULL, там міститься додаткова інформація про дочірній процес. POSIX дозволяє розробникам самостійно визначати статус бітів, тому стандарт передбачає сімейство макросів для інтерпретації параметра:

```
#include <sys/wait.h>
int WIFEXITED (status);
int WIFSIGNALED (status);
int WIFSTOPPED (status);
int WIFCONTINUED (status);
```

```
int WEXITSTATUS (status);
int WTERMSIG (status);
int WSTOPSIG (status);
int WCOREDUMP (status);
```

Будь-який із перших двох макросів може повертати значення *true* (нульове) залежно від перебігу завершення процесу. Перший макрос WIFEXITED повертає *true*, якщо процес завершується через виклик *_exit()*, звичайним чином.

Відповідно, макрос WEXITSTATUS надає 8 біт молодших розрядів і передає їх *_exit()*.

Макрос WIFSIGNALED повертає *true*, якщо переривання процесу викликав сигнал. У цьому випадку макрос WTERMSIG повертає номер сигналу, який викликав переривання, а макрос WCOREDUMP повертає *true*, якщо процес скинув ядро у відповідь на отримання сигналу. Макрос WCOREDUMP не визначається POSIX, хоча багато систем UNIX, зокрема й Linux, підтримують його.

Макроси WIFSTOPPED та WIFCONTINUED повертають *true*, якщо процес, відповідно, був зупинений або продовжений і його можна у цей момент відстежити за допомогою системного виклику *ptrace()*. Ці умови зазвичай можливі тільки під час реалізацій відладчика, хоча в разі використання разом з *waitpid()* (виклики *ptrace()* та *waitpid() стосуються сигналів і будуть розглянуті нижче*) їх можна застосовувати для контролю роботи. Зазвичай *wait()* застосовують тільки для обміну інформацією про завершення процесу. Якщо макрос WIFSTOPPED повернув *true*, то макрос WSTOPSIG наводить номер сигналу, який зупинив процес. Макрос WIFCONTINUED не визначається POSIX, хоча більш пізні стандарти визначили його для *waitpid()*. У версії 2.6.10 ядра Linux також надано макрос для *wait()*.

Приклад 4.2. Програма, яка використовує *wait()* щоб визначити, що сталося з дочірнім процесом:

```
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
int main (void)
{
int status;
pid_t pid;
if (!fork ())
return 1;
pid=wait(&status);
if (pid==-1)
perror ( "wait");
printf("pid=%d\n", pid);
if (WIFEXITED (status))
printf("Нормальне завершення, статус=%d\n",
WEXITSTATUS (status));
if (WIFSIGNALED (status))
printf("Убитий сигналом=%d%s\n", WTERMSIG (status),
WCOREDUMP (status)? "(Dumped core)": "");
if (WIFSTOPPED (status))
printf("Зупинено сигналом=%d\n", WSTOPSIG (status));
if (WIFCONTINUED (status))
printf("Продовжено\n");
return 0;
}
```

Ця програма розгалужує дочірній процес, який негайно завершується. Після цього предок – батьківський процес запускає системний виклик *wait()* для визначення статусу нащадка – дочірнього процесу. Процес друкує *pid* нащадка – дочірнього процесу й відомості про його завершення. Оскільки в цьому випадку дочірній процес завершився поверненням результату *main()*, то на виході маємо приблизно таке:

```
$ ./wait
pid = 8529
```

Нормальне завершення зі статусом виходу = 1.

Очікування визначеного процесу *waitpid()*

Спостерігати за поведінкою дочірнього процесу дуже важливо, однак часто процес має кількох нащадків – дочірніх процесів і, якщо потрібний лише певний, очікування всіх інших процесів є небажаним. Можливим рішенням були б багаторазові виклики *wait()* з постійною перевіркою значення, що повертається, проте це дуже незручно: а раптом пізніше знадобиться перевірити статус іншого завершеного процесу? Батьківському процесу довелося б зберігати результати всіх викликів wait() на випадок, якщо вони знадобляться згодом.

Якщо користувачу відомий *pid* процесу, завершення якого він чекає, можна використовувати системний виклик *waitpid()*:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t  waitpid(pid_t pid, int  *status, int  options);
```

Виклик *waitpid()* – це більш потужна версія *wait()*. Його додаткові параметри дозволяють налаштувати його більш тонко.

Параметр *pid* точно визначає, який процес або процеси потрібно очікувати. Його значення можуть потрапляти в чотири проміжки:

<-1 – очікування будь-якого дочірнього процесу, чий ID групи процесів дорівнює абсолютному значенню цієї величини; наприклад, за -500 очікується будь-який процес із групи процесів 500;

-1 – очікування будь-якого дочірнього процесу; поведінка аналогічна wait ();

0 – очікування будь-якого дочірнього процесу, що належить тій же групі процесів, що і викликаючий процес;

> 0 – очікування будь-якого дочірнього процесу, чий *pid* точно дорівнює вказаній величині; наприклад, за величини 500 очікується дочірній процес із *pid*, який дорівнює 500.

Параметр *status* працює аналогічно системному виклику *wait()* й може бути оброблений за допомогою макросів, описаних вище.

Параметр *options* може передавати такі значення за допомогою логічного «АБО» або порожнє значення:

WNOHANG – не блокувати виклик, негайно повернути результат, якщо жодний відповідний процес ще не завершився (зупинився або продовжився);

WUNTRACED – у разі його вибору встановлюється параметр WIFSTOPPED, навіть якщо викликає, то не відстежує свій дочірній; ця властивість допомагає реалізувати більш загальне керування завданнями, як це зроблено в оболонці;

WCONTINUED – якщо встановлено, то біт WIFCONTINUED у повернутому параметрі статусу встановлюється навіть якщо викликаючий процес не відстежує свій дочірній; як і у випадку з WUNTRACED, параметр є корисним для реалізації оболонки.

У разі успіху *waitpid()* повертає *pid* процесу, статус якого змінився. Якщо встановлений WNOHANG, а вказаний дочірній процес (один або кілька) не змінив свого статусу, *waitpid()* поверне 0. У разі помилки виклик повертає -1, а *errno* набуває одного з таких трьох значень:

ECHILD – процес або процеси, вказані за допомогою аргументу *pid*, не існують або не є нащадками викликаючого процесу;

EINTR – параметр WNOHANG не встановлено, а сигнал був отриманий під час очікування;

EINVAL – аргумент *options* вказано некоректно.

Приклад 4.3. Програма повинна отримати значення, що повертається дочірнім процесом з *pid* 1742, причому зробити це негайно, якщо дочірній процес ще не завершився. Реалізувати це можна у такий спосіб:

```
int status;
pid_t  pid;
pid=waitpid(1742, &status, WNOHANG);
if (pid==-1)
perror ( "waitpid");
else {
```

```
printf("pid=%d\n", pid);
if (WIFEXITED(status))
printf("Звичайне завершення зі статусом виходу=%d\n",
WEXITSTATUS(status));
if (WIFSIGNALED (status))
printf("Убитий сигналом=%d%s\n", WTERMSIG (status),
WCOREDUMP (status)? "(Дамп ядра)": "");
}
```

Приклад 4.4. Використати *wait()* можна й таким чином:

wait (&status);

Це повністю аналогічно

waitpid(-1, &status, 0);

Ще більше гнучкості при очікуванні waitid()

Для додатків, які вимагають ще більшої гнучкості в їх функціональності очікування дочірніх процесів, розширення XSI стандарту POSIX визначає, а Linux підтримує системний виклик *waitid()*:

```
#include <sys/wait.h>
int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

Як і *wait()* та *waitpid()*, системний виклик *waitid()* використовують для очікування та отримання інформації про змінений статус (завершення, зупинка, продовження) дочірнього процесу. Він надає ще більше параметрів, але їх використання дещо складніше.

Аналогічно *waitpid()* за допомогою *waitid()* розробник може вибрати очікуваний процес. Однак *waitid()* вимагає для цього вказівки не одного, а двох параметрів. За допомогою аргументів *idtype* та *id* визначають, який дочірній процес потрібно очікувати (аналогічно використанню одного аргументу *pid* в *waitpid()*). Значення *idtype* можуть бути такими:

P_PID – очікування дочірнього процесу, pid якого збігається зі значенням аргументу *id;* P_GID – очікування дочірнього процесу,

ідентифікатор групи якого збігається зі значенням *id;* P_ALL – очікування всіх дочірніх процесів, *id* ігнорується.

Аргумент *id* належить до досить рідкісного типу аргументів *id_t*, який являє собою загальний ідентифікаційний номер. Його використовують, якщо в майбутньому планується ввести нове значення *idtype*, забезпечивши таким чином гарантію, що в заданому типі можна буде зберігати новий ідентифікатор (його розмір достатній для розміщення будь-якого значення *pid_t*). Розробники Linux можуть застосовувати його аналогічно типу *pid_t,* напряму передаючи значення *pid_t* або числових констант.

У параметрі *options* можуть бути вказані одне або декілька з таких значень за допомогою двійкового «АБО»:

WEXITED – виклик буде очікувати дочірніх процесів (визначених за допомогою *id* або *idtype*), які завершилися;

WSTOPPED – виклик буде очікувати дочірніх процесів, які зупинили виконання у відповідь на отримання сигналу;

WCONTINUED – виклик буде очікувати дочірніх процесів, які продовжили виконання у відповідь на отримання сигналу;

WNOHANG – виклик не може бути заблокований і поверне результати негайно, якщо жодний із зазначених дочірніх процесів ще не завершений (або зупинений, або продовжений);

WNOWAIT – виклик не буде виводити вказаний процес зі статусу зомбі; цей процес буде оброблений у майбутньому.

У разі успіху *waitid()* повертає параметр *infop*, який вказує на допустимий тип *siginfo_t.* Точна структура *siginfo_t* залежить від реалізації, але після виконання *waitid()* заповненими залишаються лише кілька полів. Таким чином, у разі успішного виклику допустимі значення будуть міститися в таких полях:

si_pid – pid дочірнього процесу;

si_uid – uid дочірнього процесу;

si_code – може набувати значення CLD_EXITED, CLD_KILLED, CLD_STOPPED або CLD_CONTINUED у результаті завершення дочірнього процесу, закінчення або продовження його за сигналом відповідно;

si_signo – встановлюється значення SIGCHLD;

si_status – якщо *si_code* встановився рівним CLD_EXITED, це поле містить код виходу дочірнього процесу, і навпаки, це поле набуває значення номера сигналу, який відправив дочірній процес і викликав його зміни.

У разі успіху *waitid()* повертає 0. У разі помилки *waitid()* повертає -1, а *errno* набуває одного з таких значень:

ECHLD – процесу або процесів, вказаних за допомогою *id* або *idtype,* не існує; EINTR – WNOHANG не встановлений в *options*, а сигнал перервав виконання; EINVAL – аргумент *options* або комбінація аргументів *id* та *idtype* некоректні.

Функція *waitid()* надає додаткову корисну семантику, якої немає в *wait()* та *waitpid().* Зокрема, інформація, що надається у структурі *siginfo_t,* може бути дуже корисною. Якщо ці відомості не потрібні, має сенс використовувати більш прості функції, які підтримуються великою кількістю систем, й отже можуть бути перенесені в інші системи, які не належать до Linux.

4.1.4. Стиль BSD: wait3() і wait4()

Виклик *waitpid()* є спадкоємцем System V Release 4 системи AT&T, а BSD йде власним шляхом і надає дві інші функції, які використовують для очікування зміни статусу дочірнього процесу:

```
#include <sys/types.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/wait.h>
pid_t wait3(int *status, int options, struct rusage *rusage);
```

*pid_t wait4 (pid_t pid, int *status, int options, struct rusage *rusage);*

Цифри 3 та 4 вказують, що ці дві функції є, відповідно, версіями *wait()* з трьома і чотирма параметрами. Функції працюють схоже на *waitpid()* за винятком аргументу *rusage*. Запуск *wait3()* такий:

pid=wait3(status, options, NULL);

еквівалентний наступному виклику *waitpid():*

pid=waitpid(-1, status, options);

у той час як цей запуск wait4():

pid= wait4(pid, status, options, NULL);

еквівалентний такому виклику waitpid ():

pid=waitpid(pid, status, options);

```
#include <sys/resource.h>
struct rusage {
    struct timeval ru_utime;   /* Витрачений для користувача час */
    struct timeval ru_stime;   /* Витрачений системний час */
    long ru_maxrss;        /*Максимальний розмір резидентної частини */
    long ru_ixrss;             /* Розмір загальної пам’яті */
    long ru_idrss;             /* Розмір власних даних */
    long ru_isrss;             /* Розмір власного стека */
    long ru_minflt;            /* Відновлення сторінок */
    long ru_majflt;            /* Сторінкові переривання */
    long ru_nswap;             /* Операції підкачки */
    long ru_inblock;           /*  Блокові операції введення */
    long ru_oublock;           /* Блокові операції виведення */
    long ru_msgsnd;            /* Відправлені повідомлення */
    long ru_msgrcv;            /* Отримані повідомлення */
    long ru_nsignals;          /* Отримані сигнали */
    long ru_nvcsw;             /* Добровільні перемикання контексту */
    long ru_nivcsw;            /* Вимушені перемикання контексту */
};
```

У разі успіху ці функції повертають *pid* процесу, який змінив статус. У разі невдачі вони повертають -1 і встановлюють в *errno* ті ж значення помилок, що описані для *waitpid().*

Оскільки *wait3()* і *wait4()* не визначені у стандарті POSIX1, бажано застосовувати їх тільки якщо отримати інформацію про використання ресурсів надзвичайно важливо. Потри відсутність стандартизації POSIX, майже всі системи UNIX підтримують ці два виклики.

Запуск та очікування завершення нового процесу system()

У стандартах ANSI *C* та POSIX визначається інтерфейс, який об'єднує запуск нового процесу та очікування його завершення – аналогом є синхронне створення процесів. Якщо процес запускає дочірній, тільки щоб негайно почати очікувати його завершення, найкраще використовувати такий інтерфейс:

```
#define _XOPEN_SOURCE     /* якщо потрібен WEXITSTATUS і т. д. */
#include <stdlib.h>
int system(const char *command);
```

Функцію *system()* називають так тому, що синхронний запуск процесів називають *виходом у систему*. Зазвичай *system()* використовують для запуску простої утиліти або сценарію оболонки, коли основною метою є отримання значення, яке ними повертається. Виклик *system()* запускає системну команду, яка надається параметром *command*, включаючи будь-які додаткові аргументи як би вона була набрана у командному рядку. По суті ця функція запускає стандартний інтерпретатор *Bourne shell*. Параметр *command* додається до аргументів ***/bin/sh -c***. У цьому сенсі параметр передається усій оболонці.

У разі успіху величиною, що повертається, є статус, який повернула команда і надала виклик *wait().* Таким чином, код виходу виконаної команди отримуємо за допомогою WEXIT STATUS. Якщо виклик */bin/sh* зазнав невдачі, величина, що передається WEXITSTATUS, така ж,

що повертається *exit(127)*. Оскільки команда, яка викликається, також може повернути 127, безпомилкового методу визначення джерела помилки – чи то вона викликана командою, чи то оболонкою – немає. У разі помилки виклик повертає значення -1. Якщо параметр *command* має значення NULL, *system()* повертає ненульову величину, коли *shell /bin/sh* доступний, і 0 в іншому разі.

Під час виконання команди SIGCHLD блокується, а SIGINT та SIGQUIT ігноруються. Ігнорування SIGINT та SIGQUIT має різні наслідки, зокрема, коли *system()* викликається всередині циклу. У цьому випадку потрібно переконатися, що програма правильно перевіряє статус виходу дочірнього процесу, наприклад:

```
do {
int ret;
ret=system("pidof rudderd");
if (WIFSIGNALED (ret) &&
(WTERMSIG (ret) == SIGINT ||
WTERMSIG (ret) == SIGQUIT))
break;    /* або інший варіант обробки */
} while (1);
```

Реалізація *system()* з використанням *fork()*, функції з сімейства *exec* та *waitpid()* – дуже корисне завдання. Наведемо приклад використання системних викликів *fork()*, *execv()* та *waitpid()*:

```
/ *
* my_system – синхронно розгалужує дочірній процес і чекає команди
* "/bin/sh  -c <cmd>".
*
* Повертає -1 в разі будь-якої помилки або код виходу запущеного процесу
* Не блокує і не ігнорує сигнали
* /
int  my_system(const  char  *cmd)
{
int status;
pid_t  pid;
```

```
pid=fork();
if (pid==-1)
return -1;
else if (pid==0) {
const  char  *argv [4];
argv[0]="sh";
argv[1]="-c";
argv[2]=cmd;
argv[3]=NULL;
execv("/bin/sh", argv);
exit (-1);
}
if (waitpid(pid, &status, 0)==-1)
return -1;
else if (WIFEXITED(status))
return WEXITSTATUS(status);
return -1;
}
```

У цьому прикладі не блокуються і не відключаються ніякі сигнали, на відміну від офіційної версії *system().* Цей підхід може бути кращим або гіршим залежно від конкретної програми, але не блокувати принаймні SIGING часто може бути корисно, тому що це дозволяє переривати запущену програму так, як звик користувач у звичайних ситуаціях. Ускладнити завдання можна, додавши додаткові вказівники у вигляді параметрів, які вказували б на помилки, що їх у цей час неможливо розрізнити, коли їх значення відмінне від NULL. Як приклад можна додати *fork_failed* і *shell_failed.*

4.1.5. Зомбі

Процес, який перервався, але чий предок – батьківський процес поки не очікує його завершення, називають зомбі. Процеси-зомбі продовжують займати системні ресурси, нехай і в невеликих кількостях, адже треба підтримувати лише «скелет» процесу, який працював раніше. Ці ресурси потрібні, щоб батьківські процеси, які хочуть перевірити статус своїх нащадків – дочірніх процесів, отримали інформацію, що належить до

існування і припинення цих процесів. Як тільки батьківський процес зробить це, ядро остаточно видаляє процес і зомбі відправляється на спокій.

Однак кожний, хто якийсь час використовував Linux, час від часу натикається на процеси-зомбі. Ці процеси часто називають *примарами* (рос. *призраками*), вони неначе діти безвідповідальних батьків. Якщо додаток користувача розгалужує дочірній процес, то він несе відповідальність за його обслуговування, навіть якщо це полягає лише у видаленні зібраної інформації. Інакше все дочірні процеси будуть ставати примарами та існувати у списках процесів системи, що аж ніяк не прикрашає додаток.

Що відбувається, якщо батьківський процес вмирає раніше дочірнього або якщо він завершується до того, як отримає можливість подбати про своїх нащадків-зомбі? Коли б не завершувався процес, ядро Linux переглядає список його нащадків і *перепризначає батьківський процес*, роблячи їх предком – батьківським процес ініціалізації, *pid* якого дорівнює 1. Таким чином, жодний процес не стане сиротою без батьківського. Процес ініціалізації, у свою чергу, періодично піклується про всіх своїх нащадків, у результаті жодний із них не залишається зомбі надмірно довго (ніяких привидів). Отже, якщо батьківський процес завершується раніше за своїх нащадків або не обслуговує їх перед завершенням, дочірні процеси переходять під опіку процесу ініціалізації та обслуговуються ним до повного завершення. Хоча подібна реалізація вважається хорошою практикою, така обережність означає, що короткострокові процеси можуть особливо не турбуватися про очікування всіх своїх нащадків.

4.2. Керування процесами в оболонці Bash

Процеси в ОС Ubuntu відіграють ключову роль. Від оптимального налаштування підсистеми керування процесами і кількості процесів, які одночасно виконуються, залежить завантаження ресурсів процесора, що, у свою чергу, має безпосередній вплив на продуктивність системи в цілому. Від того, які процеси виконуються у системі користувача, залежить, чи є вона сервером бази даних або сервером мережевого доступу, засобом проектування або обчислювальним сервером.

4.2.1. Типи процесів

Процеси поділяють на системні, демони (неінтерактивні) і прикладні (інтерактивні).

1. ***Системні процеси.*** Є частиною ядра і завжди розміщені в оперативній пам'яті. Вони не мають відповідних їм програм у вигляді виконуваних файлів і запускаються особливим чином при ініціалізації ядра системи. Тобто їх виконувані інструкції та дані перебувають в ядрі, тому такі процеси є складовою ядра. Системні процеси можуть викликати функції, а також звертатися до даних, які не мають доступу до інших процесів.

Системними процесами є: *shed* (*диспетчер свопінгу*), *vhand* (*диспетчер сторінкового заміщення*), *bd fflush* (*диспетчер буферного кешу*) та *kmadaemon* (*диспетчер пам'яті ядра*) [2].

Підкачка сторінок або *swapping* – один із механізмів віртуальної пам'яті, коли окремі фрагменти пам'яті (зазвичай неактивні) переміщаються з ОЗП (оперативного запам'ятовувального пристрою) у вторинне сховище (жорсткий диск або інший зовнішній накопичувач, такий як флеш-пам'ять), звільняючи ОЗП для завантаження інших активних фрагментів пам'яті. Такими фрагментами є сторінки пам'яті. Диспетчер сторінкового заміщення *vhand* звільняє сторінки пам'яті (*page*

stealing daemon). Диспетчер буферного кешу *bdfflush* звільняє «брудні» буфери, зберігаючи їх вміст у відповідних дискових блоках. Операції «введення – виведення» – це повільні операції, які здійснюються поблоково. Буфер – це подання одного блоку фізичного диска у пам'яті, тобто це те, що ще не записано на диск. Через «буфери» блоки дисків зіставляються з пов'язаними на них сторінками пам'яті.

До системних процесів слід віднести *init*, який є прабатьком усіх інших процесів в UNIX. Хоча *init* не є частиною ядра і його запуск відбувається з виконуваного файлу *(/etc/init*), його робота життєво важлива для функціонування усієї системи в цілому.

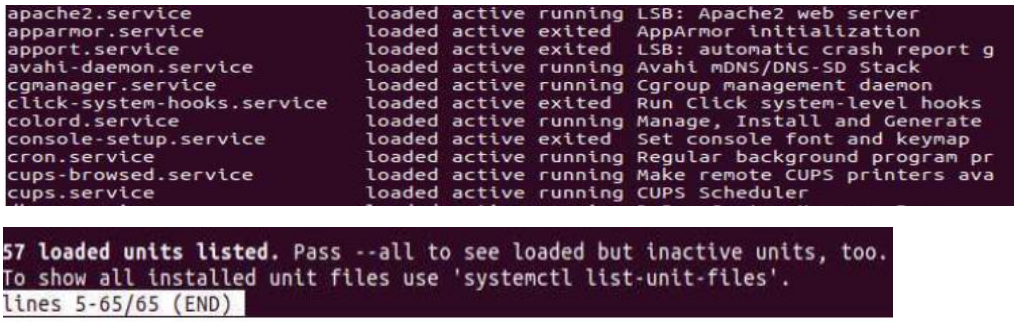
2. ***Демони (вони ж сервіси або служби) – неінтерактивний процес.*** Це програми, які працюють у фоновому режимі. Вони не мають графічного інтерфейсу і не прив'язані до жодного керуючого терміналу. Зазвичай демони запускаються під час ініціалізації системи, однак після ініціалізації ядра забезпечують роботу різних підсистем Unix: системи термінального доступу, системи друку, системи мережевого доступу, мережевих послуг тощо. Демони не пов'язані з жодним користувачем, тобто жодним чином не стосуються користувацьких процесів. Отримавши команду, вони виконують дію, для якої були створені. Наприклад, демон друку *cupsd* ставить у чергу документи, надіслані на друк, а потім надсилає їх на принтер. Як правило, демони перебувають у стадії очікування, доки для певного процесу не виникне потреба виконати певну послугу (звернення до архіву файлу, друк документа). Прикладами процесів-демонів слугують сервери протоколів HTTP (*httpd*) та FTP (*ftpd*), сервер системного журналу (*syslogd*), інші приклади – *usbd*, *sshd.* Зазвичай демони в кінці назви містять літеру «d». Для Linux неабияке значення має система ініціалізації демонів ***systemd***. Вона відповідає за завантаження, керування запущеними процесами, записування і зберігання логів, а також за багато інших процесів в Ubuntu та більшості дистрибутивів, заснованих на ядрі Linux.

Systemd використовують як головний демон. Під час завантаження *systemd* ініціалізує усі інші сервіси, керує їх роботою аж до вимикання. Якщо треба, можна запустити або зупинити потрібний процес, призначити або скасувати його автоматичний запуск або навіть створити власний сервіс.

Керування сервісами через systemd. Щоб дізнатися, які саме сервіси запущені у певний момент, треба скористатися командою ***systemctl***, яка забезпечує звернення до *systemd*, у термінал вводять

systemctl list-units -t service

і можна побачити велику кількість сервісів (наведено початок і кінець):



У цьому випадку запущено 65 сервісів. Складові команди означають:

systemctl – звернутися до *systemd*;

list-units – вивести список юнітів (*unit* – це базовий об’єкт, яким керує *systemd*; юніти можуть бути багатьох типів, але найбільш часто використовуваним є «сервіс», тобто назва юніта закінчується на «.*service*»);

-t – ключ, що означає, що далі треба вказати тип юніта (тут це сервіс).

У списку на скріншоті є *cups.service* – це служба друку. Щоб сервіс не запускався автоматично, є така команда:

sudo systemctl disable cups

Замість *cups* можна підставити назву будь-якого іншого демона, який є бажання вилучити з автозавантаження. За потреби його легко можна буде повернути командою: **sudo systemctl enable назва_демона**

Для негайної зупинки слугує команда

sudo systemctl stop назва_демона

Наприклад: **sudo systemctl stop nginx.service**

Для негайного запуску слугує команда

sudo systemctl start назва_демона

Наприклад: **sudo systemctl start nginx.service**

Щоб перезапустити сервіс використовують команду

sudo systemctl restart nginx.service

Щоб перевірити, чи запущений у цей момент *веб-сервер Apache*, слід виконати команду **systemctl status apache2**

Отже, для отримання інформації про запущені сервіси досить прав простого користувача з використанням *sudo.*

Між тим можна відключити та перезапустити сервер:

– відключення: **sudo systemctl poweroff**

– перезавантаження: **sudo systemctl reboot**

3. Усі інші ***процеси***, які виконуються в системі, вважають ***прикладними***, або ***інтерактивними***. Практичними вважають процеси, які запускаються під час роботи користувача. Наприклад, під час реєстрації користувача в системі запускається командний інтерпретатор (*shell*), який надає можливість працювати користувачу в Unix. Інші приклади інтерактивних процесів – *ls, sh, fsck.*

Користувацькі процеси можуть виконуватися як *в інтерактивному, так і у фоновому режимі,* але виключно в межах сеансу користувача. Під час виходу із системи усі користувацькі процеси знищуються.

Операційна система Ubuntu є багатозадачною. Це означає, що одночасно може виконуватися кілька процесів. Водночас *процеси* мають можливість *обмінюватися один з одним даними* за допомогою ОС завдяки ***міжпроцесній взаємодії*** (***Interprocess Communication – IPC***). Для цього є набір засобів взаємодії між процесами, а саме: ***канали*** (***pipes***), ***сигнали*** (***signals***), ***розділювана пам'ять*** (***shared memory***), ***повідомлення*** (**messages**), ***семафори*** (***semaphores***), ***сокети***.

Канали *(неіменовані (pipe, конвейєри) та іменовані (fifo)) – це засіб передавання даних між процесами.* Канал можна уявити як невеликий кільцевий буфер в ядрі ОС. З точки зору процесів канал виглядає як пара відкритих файлових дескрипторів – один на читання й один на запис. У канал можна писати доти, поки є місце в буфері, а якщо місце в буфері скінчиться, процес буде заблокований на запис. Також користувач може читати з каналу, поки є дані в буфері, а якщо даних немає, процес буде заблокований на читання. Якщо закрити дескриптор, який відповідає за запис, то спроба читання вказує на кінець файлу. Якщо закрити дескриптор, який відповідає за читання, то спроба запису призведе до доставки сигналу SIGPIPE й помилки EPIPE.

При використанні каналу («|») в програмуванні на мові *shell*:

```
ls | grep abc
```

блокування читання / записування забезпечує синхронізацію швидкості виконання двох програм та їх одночасне завершення.

Неіменований канал створюють викликом *pipe*, який заносить у масив int [2] два дескриптори відкритих файлів: fd[0] – відкритий на читання, fd[1] – відкритий на запис (згадаймо: STDIN==0, STDOUT==1). Канал знищується, коли будуть закриті усі файлові дескриптори, які на нього посилаються.

Іменований канал FIFO (*First In First Out*) доступний як об'єкт у файловій системі. При цьому до відкриття об'єкта FIFO на читання

комунікаційний об’єкт не створюється. Після відкриття об’єкта FIFO розглядається в одному процесі на читання, а в іншому на запис, й виникає ситуація, повністю еквівалентна використанню неіменованого каналу.

Об’єкт FIFO у файловій системі створюється викликом функції

*int mkfifo (const char * pathname, mode_t mode).*

Цей вид каналу створюється за допомогою *mknod* або *mkfifo.*

Основна відмінність між *pipe* та FIFO полягає у тому, що *pipe* можуть спільно використовувати тільки процеси, які перебувають у відношенні «батьківський – дочірній», а FIFO може використовувати будь-яка пара процесів.

Сигнали – це асинхронне повідомлення процесу про будь-яку подію. Коли сигнал надісланий процесу, ОС перериває його виконання. Якщо процес встановив власний обробник сигналу, ОС запускає цей обробник, передавши йому інформацію про сигнал [1]. Якщо процес не встановив обробника, то виконується оброблювач за замовчуванням. Сигнали утворюються:

- з термінала, натисканням спеціальних клавіш або комбінацій (наприклад, натискання Ctrl-C генерує SIGINT, а Ctrl-Z – SIGTSTP);
- ядром системи:

– при виникненні апаратних виключень (неприпустимих інструкцій, порушень під час звернення до пам’яті, системних збоях тощо);

– помилкових системних викликах;

– для інформування про події «введення – виведення»;

- одним процесом іншому (або самому собі), за допомогою системного виклику *kill(),* зокрема з оболонки *shell* утилітою */bin/kill*.

Сигнали посилаються такими засобами:

– *розділювана пам’ять* (це пам’ять, яка дозволяє здійснювати обмін інформацією не через ядро, в через певну частину віртуального адресного простору, в яку розміщують та зчитують дані);

– *черги повідомлень* (обмін повідомленнями здійснюється таким чином: один процес поміщає повідомлення в чергу за допомогою деяких системних викликів, а будь-який інший процес може прочитати його там, за умови, що і процес – джерело повідомлення, і процес – приймач повідомлення використовують один і той самий ключ для отримання доступу до черги).

Кожному виду сигналів у системі відповідають назва й номер. Чимало сигналів мають спеціальний сенс, зокрема, найбільш вживаними сигналами є такі:

SIGINT (код 2) – користувач натиснув Ctrl+C;

SIGKILL (код 9) – примусове завершення процесу;

SIGTERM (код 15) – попередження, що процес незабаром буде знищений;

SIGCONT (код 18) – відновлення виконання призупиненого процесу;

SIGSTOP (код 19) – припинення виконання процесу;

SIGTSTP (код 20) – натискання клавіш <CTRL>+<Z>, сигнал викликає зупинку виконання процесу.

Сигнали SIGKILL (примусове завершення процесу) і SIGSTOP неможливо ні перехопити, ні ігнорувати, тому що вони адресовані не процесу, а планувальнику ОС.

В ОС Linux ***семафори*** (*semaphore*) – це лічильники, які керують доступом до загальних ресурсів, їх використовують для синхронізації потоків [1]. Семафори являють собою механізм блокування, який переводить процеси у стан очікування. Після закінчення виконання завдання намагається захопити семафор, який вже утримується, семафор поміщає це завдання у чергу очікування (*wait queue*) і переводить це завдання у стан очікування (*sleep*). Коли процеси, які утримують семафор, звільняють блокування, одне із завдань черги очікування повертається до

виконання
і може захопити семафор.

Таким чином, семафор – це змінна особливого типу, яка може змінюватися з позитивним або негативним приростом, але звернення до змінної у відповідний момент завжди атомарно навіть у багатопотокових програмах. Це означає, що якщо два потоки (або кілька) у програмі намагаються змінити значення семафора, система гарантує, що всі операції будуть насправді виконуватися одна за одною. Щодо звичайних змінних, результат конфліктних операцій різних потоків в одній програмі довільний.

Сокети являють собою *віртуальний об'єкт*, який існує, поки на нього посилається хоча б один із процесів. *Сокети* UNIX бувають двох типів: локальні й мережеві. *Локальному сокету* присвоюється UNIX-адреса і буде створений спеціальний файл (файл сокета) відповідно до заданого шляху, через який зможуть повідомлятися будь-які локальні процеси простим читанням / записуванням з нього / у нього. Якщо використано *мережевий сокет,* створюється абстрактний об'єкт, прив'язаний до порту ОС, який слухає, та до мережевого інтерфейсу. Цьому типу сокета присвоюється INET-адреса, яка має адресу інтерфейсу та порту, який слухає.

Процеси можуть виконуватися на передньому плані (*foreground*) – режим за замовчуванням, та у фоновому режимі (*background*). На передньому плані в кожний момент для поточного термінала може виконуватися тільки один процес. Однак користувач може перейти в інший віртуальний термінал і запустити на виконання ще один процес, а на іншому терміналі ще один і т. д. Процес переднього плану – це процес, з яким користувач взаємодіє, такий процес отримує інформацію з клавіатури (стандартне введення) і надсилає результати на екран користувача (стандартне виведення). Фоновий процес після запуску завдяки використанню спеціальної команди командної оболонки

відключається від клавіатури та екрана, тобто не очікує введення даних зі стандартного введення і не виводить інформацію на стандартне виведення, а командна оболонка не очікує закінчення запущеного процесу, що дозволяє користувачеві негайно запустіть ще один процес. Зазвичай фонові процеси вимагають дуже великого часу для свого завершення і не потребують втручання користувача під час перебігу процесу. Наприклад, компіляцію програм або архівування великого обсягу інформації можна перевести у фоновий режим.

Щоб запустити програми у вигляді фонового процесу, досить набрати в командному рядку ім'я програми і в кінці додати знак амперсанта (**&**), відокремлений пропуском від імені програми та її параметрів командного рядка, якщо такі є. Наприклад, команда ***yes*** виводить рядок на екран нескінченну кількість разів. Щоб знищити цей процес, треба відправити йому сигнал переривання, тобто натиснути **Ctrl + C**:

```
qwe@vb:~$ yes "This is an example"
```

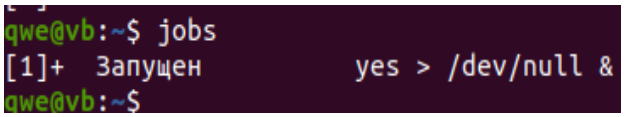
Зробимо тепер інакше. Щоб на екран не виводилася ця нескінченна послідовність, перенаправимо стандартне виведення команди *yes* на ***/dev/null***. Пристрій */dev/null* діє за принципом чорної діри, а саме: усі дані, надіслані у цей пристрій, пропадають. За допомогою цього пристрою дуже зручно позбавлятися від занадто великої кількості виведення у деяких програмах. Тобто команда ***yes*** виконується у фоновому режимі (**&**) і з подавленням виведення має такий вигляд:

```
qwe@vb:~$ yes >/dev/null &
[1] 3022
```

Після команди виводиться повідомлення, що складається із двох чисел. Перше число в дужках означає номер запущеного фонового процесу для користувача у поточному сеансі, з його допомогою можна проводити маніпуляції з цим фоновим процесом. Друге число показує

ідентифікаційний номер (PID) процесу. Відмінності цих двох чисел досить суттєві. Номер фонового процесу унікальний тільки для користувача, що запускає цей процес. Тобто якщо в системі три користувачі вирішили запустити фоновий процес (перший для поточного сеансу), то в результаті у кожного користувача з'явиться фоновий процес із номером 1. Навпаки, ідентифікаційний номер процесу (PID) є унікальним для всієї ОС й однозначно ідентифікує в ній кожний процес. *Для перевірки стану фонових процесів* можна скористатися командою командної оболонки – ***jobs***:
/home/user$ jobs

[1] + Running yes> /dev/null &

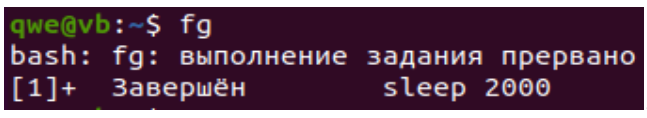


З вищенаведеного прикладу видно, що у користувача user у певний момент запущений один фоновий процес, і він виконується. Подавлення виведення команди здійснюється перенаправленням вихідного потоку на псевдопристрій */dev/null*. Усе, що записується в цей файл, «зникає» назавжди.

Команда ***yes*** може стати в нагоді, якщо скрипт або команда вимагає взаємодії з користувачем, яка полягає тільки в натисканні «Y» кожний раз:

yes | <команда чи скрипт>

Для виведення процесу із фонового режиму використовують команду ***fg*** (foreground) і процесу надається функція керування (керує потоками «введення – виведення» терміналу):



Команда *fg* без аргументів виводить завдання, позначене знаком «+» у списку завдань *jobs*, тобто останнє відправлене у «фон» завдання. Наприклад,

```
# jobs
[6]-  Stopped                 watch ls  (wd: /home/setevoy)
[7]+  Stopped                 top
[8]   Done                    tar cpf usr.lib.tar usr/lib 2> /dev/null
```

Команда *fg 7* поверне на екран утиліту *top* і переведе її в режим *Running*.

З точки зору ядра процес являє собою запис у ***таблиці процесів***. Цей запис містить відомості про стан процесу і дані, що існують протягом усього часу його життя. Розмір таблиці процесів дозволяє запускати кілька сотень процесів одночасно. Процес також використовує таблицю усіх файлів, відкритих процесом файлів, які зберігаються в його адресному просторі. *Запис у таблиці процесів і простір процесу разом становлять контекст, або оточення, процесу*.

Таблиця процесів

Системний планувальник використовує таблицю процесів, описану в заголовку */usr/include/linux/sched.h* Всередині структури ***struct task_struct*** уміщені всі відомості про стан процесу. Вони досить добре прокоментовані. Основними є такі відомості:

• *Ідентифікація процесу* чітко встановлює його права, які визначаються виходячи з ефективних або реальних номерів користувача номерів груп. Тут також міститься ідентифікатор процесу (PID).

• *Пріоритет процесу* визначає черговість його виконання. Кожний процес має певний час на виконання. Якщо цей час перевищено, процес має перервати роботу, перейти у стан неготовності й чекати, поки до нього дійде черга наступного разу. За пріоритетом процесу ядро може вирішити, який процес буде запущений наступним.

• *Облікові відомості* (*accounting*) – це інформація про можливості отримання доступу до певної області пам’яті, яка ще не завантажена. При цьому апаратура повідомляє про відсутність сторінки, після чого ядро завантажує цю сторінку в пам’ять.

• *Контрольний термінал* – кожний процес, крім процесів-демонів, потребує контрольного терміналу, в який виводяться повідомлення стандартного «введення – виведення» і помилки.

Контекст процесу (*pid, ppid, tty, nice, stat, таблиця відкритих файлів, змінні оточення*) був розглянутий раніше.

4.2.2. Моніторинг процесів

Процес – це об'єкт, який складається з адресного простору пам'яті та набору структур даних. Для перегляду інформації про працюючі у системі процеси можна використовувати наявні утиліти ***ps*** та ***top***.

Команда ***ps*** (*process status)* показує моментальний *знімок процесів у системі*. Команду *ps використовують для отримання списку запущених процесів, вона може показати їх PID*, скільки пам'яті вони використовують, команду, в якій вони були запущені й под.

Команда ***top*** (*table of processes*) дозволяє *переглядати інформацію про процеси у реальному часі: показує запущені процеси й оновлює екран кожні* кілька секунд, що дозволяє спостерігати за роботою комп'ютера в реальному часі.

Утиліта *ps* (*process state* – стан процесів) зчитує інформацію про роботу запущених процесів у системі з віртуальних файлів у псевдофайлову систему *procfs*, яка монтується в каталог */proc*. У цій файловій системі немає фізичного місця розміщення, немає блочного пристрою, такого, як жорсткий диск. Вся інформація, яка зберігається у цьому каталозі, міститься в оперативній пам'яті комп'ютера й контролюється ядром ОС, вона не призначена для зберігання файлів користувача. Ця одна з важливих утиліт для системного адміністрування спеціально створена для того, щоб зрозуміти, що відбувається в системі Linux. Для виведення списку всіх процесів у поточній оболонці використовують команду **ps (Get-Process)**: ***ps [PID] options***.

Команда *ps* має три типи стилів подання опцій:

– стиль Unix98: ps [-опції] (використовують дефіс);

– стиль BSD (*Berkeley Software Distribution*): ps [опції] (без дефіса); BSD – система *поширення* ПЗ у вихідних кодах, наприклад, FreeBSD, OpenBSD. Це дистрибутиви університету Бекрлі;

– стиль GNU-версії: ps [-- довге ім’я опції [--довге ім’я опції] ...] (використовують два дефіси) [16].

Деякі процеси вказані у квадратних дужках [] – це процеси, які входять безпосередньо до складу ядра і виконують важливі системні завдання, наприклад такі, як керування буферним кешем [*pdflush*] та організація свопінга [*kswapd*]. З ними краще не експериментувати – нічого доброго з цього не вийде. Інша частина процесів належить до користувацьких.

Довідку про опції команди виводять як ***man ps***. Команда ***ps*** без будь-яких аргументів відображає процеси для поточної оболонки:

```
qwe@vb:~$ ps
  PID TTY          TIME CMD
 2251 pts/1    00:00:00 bash
 2829 pts/1    00:00:00 ps
```

Виводиться *PID* (ідентифікатор процесу), *PPID* (у разі наявності батьківського процесу), *TTY* (ідентифікатор терміналу), *TIME* (час ЦП), CMD (ім’я команди).

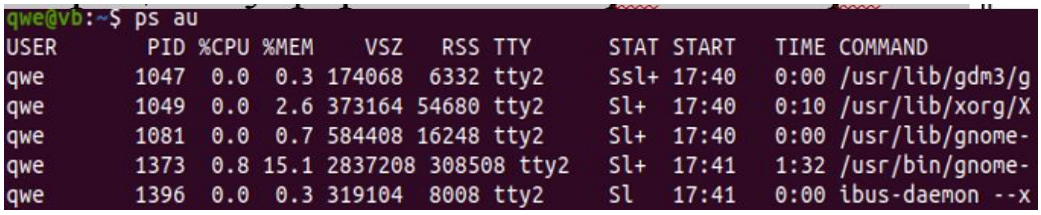
Виведення усіх процесів у різних форматах

Кожний активний процес у системі Linux в загальному форматі (Unix/Linux) відображається з опціями «-A» або «-e», тобто ***ps -A*** або ***ps -e***:

```
qwe@vb:~$ ps -A
  PID TTY          TIME CMD
    1 ?        00:00:08 systemd
    2 ?        00:00:00 kthreadd
    3 ?        00:00:00 rcu_gp
    4 ?        00:00:00 rcu_par_gp
```

```
qwe@vb:~$ ps -e
  PID TTY          TIME CMD
    1 ?        00:00:12 systemd
    2 ?        00:00:00 kthreadd
    3 ?        00:00:00 rcu_gp
    4 ?        00:00:00 rcu_par_gp
    6 ?        00:00:00 kworker/0:0H-kblockd
    8 ?        00:00:00 mm_percpu_wq
```

Усі активні процеси відображаються у форматі BSD (тобто це розширене виведення параметрів) у вигляді ***ps au*** або ***ps aux***:



% CPU – частка процесорного часу, виділена процесу;

% MEM – відсоток використовуваної оперативної пам'яті;

VSZ – віртуальний розмір процесу;

RSS– розмір фізичної пам'яті, яку використовує процес;

TTY – керуючий термінал;

STAT– статус процесу: *R* – виконується; *S* – перебуває у стані сну (очікування події для завершення); *s* – є лідером сесії; *l* – багатопотоковий; Z – зомбі;

<– підвищений пріоритет;

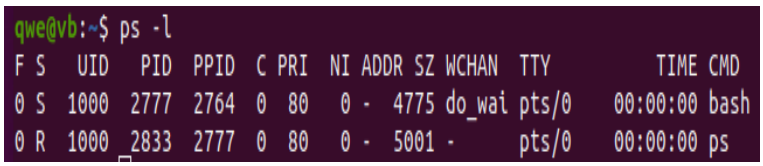
+ – перебуває в інтерактивному режимі;

START – час запуску,

TIME – час виконання на процесорі.

Приклад відображення запущених користувацьких процесів, ***ps -x***:

Команда ***ps -l*** у форматі BSD виводить розширений лістинг:



C – час обробки процесів у процесорі;

PRI – пріоритет процесу Linux на рівні ядра (зазвичай NI+20);

NI – пріоритет виконання процесу від -20 до 19;

ADDR – адреса процесу;

SZ – розмір на фізичних сторінках;

WCHAN – адреса події, якої очікує процес. В активного процеса ця колонка порожня.

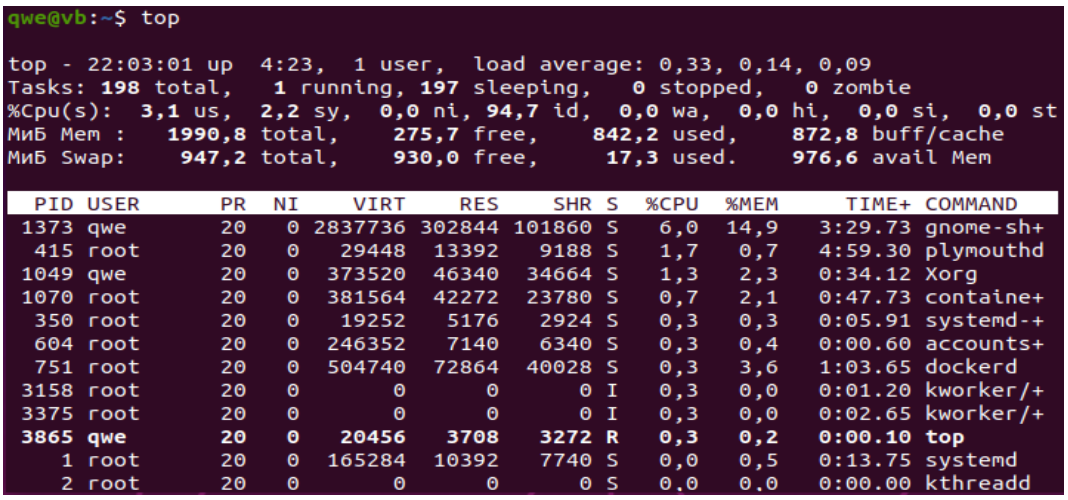
Приклад виведення дерева процесів ***ps -f –forest*** або ***pstree*** або ***pstree <ім'я користувача>***

```
qwe@vb:~$ ps -f --forest
UID          PID  PPID  C STIME TTY          TIME CMD
qwe         2251  2240  0 17:43 pts/1    00:00:00 bash
qwe         3787  2251  0 21:29 pts/1    00:00:00  \_ ps -f --forest
```

```
qwe@vb:~$ pstree
systemd─┬─ModemManager───2*[{ModemManager}]
        ├─NetworkManager─┬─dhclient
        │                └─2*[{NetworkManager}]
        ├─accounts-daemon───2*[{accounts-daemon}]
        ├─acpid
        ├─avahi-daemon───avahi-daemon
        ├─boltd───2*[{boltd}]
        ├─colord───2*[{colord}]
        ├─cron
        ├─cups-browsed───2*[{cups-browsed}]
        ├─cupsd
        ├─dbus-daemon
        ├─fprintd───{fprintd}
        ├─fwupd───4*[{fwupd}]
        ├─gdm3─┬─gdm-session-wor─┬─gdm-x-session─┬─Xorg───3*[{Xorg}]
        │      │                 │               ├─gnome-session-b─┬─evolution-+
        │      │                 │               │                 ├─gnome-shel+
        │      │                 │               │                 ├─gnome-soft+
        │      │                 │               │                 ├─gsd-a11y-s+
        │      │                 │               │                 ├─gsd-clipbo+
        │      │                 │               │                 ├─gsd-color─+
```

Команда top

Команда ***top*** відображає стан процесів, які здійснюються у режимі реальному часу, тобто ця команда виконує функцію системного монітора і дозволяє виявити причини нестабільної роботи ОС та процеси, які споживають більшість системних ресурсів:



Перший рядок (*top*) містить загальну інформацію:

– поточний час (22:03:01);

– час роботи системи (up 4 day, 4:23);

– кількість відкритих сесій користувачів (1 *user*);

– середнє завантаження системи (*load average*: 0,33, 0,14, 0,09), три значення відповідають завантаженню в останню хвилину, п’ять хвилин і п’ятнадцять хвилин відповідно.

Другий рядок (task) відтворює статистику процесів:

– загальна кількість процесів у системі (198 *total*);

– кількість процесів, які працюють у цей момент (1 *running*);

– кількість очікуючих подій процесів (197 *sleeping*);

– кількість зупинених процесів (0 *stopped*);

– кількість процесів, які очікують батьківський процес для передадавання статусу завершення (0 zombie).

Третій рядок (%Cpu(s)) – статистика використання центрального процесора (*cpu*):

– відсоток використання центрального процесора для користувача процесів (3,1% *us*);

– відсоток використання центрального процесора системними процесами (2,2% *sy*);

– відсоток використання центрального процесора процесами із пріоритетом, підвищеним за допомогою виклику *nice* (0,0% *ni*);

– відсоток часу, коли центральний процесор не використовується (94,7% *id*);

– відсоток використання центрального процесора процесами, які очікували завершення операцій «введення – виведення» (0,0% *wa*);

– відсоток використання центрального процесора обробниками апаратних переривань (0,0% *hi* – Hardware IRQ (апаратні переривання));

– відсоток використання центрального процесора обробниками програмних переривань (0,0% *si* – Software Interrupts (програмні переривання));

– кількість ресурсів центрального процесора, «запозичених» у віртуальної машини гіпервізором для інших завдань (таких, як запуск іншої віртуальної машини); це значення дорівнюватиме нулю на настільних комп'ютерах і серверах, які не використовують віртуальні машини (0,0% *st* – Steal Time (запозичений час)).

Підсумовування усіх значень має дорівнювати 100 %.

Четвертий і п'ятий рядки – статистика використання пам'яті (*memory usage*): у четвертому і п'ятому рядках виводиться інформація про використання фізичної оперативної пам'яті та розділу підкачки відповідно. Наведемо значення в порядку проходження: загальна кількість пам'яті (*total*), кількість використовуваної пам'яті (*used*), кількість вільної пам'яті (*free*), кількість пам'яті в кеші буферів (*buffers*). *Наступні рядки – список процесів*:

PID – ідентифікатор процесу;

USER – ім'я користувача, який є власником процесу (*root*);

PR – пріоритет процесу;

NI – значення "NICE", яке впливає на пріоритет процесу;

VIRT – обсяг віртуальної пам'яті, який використовується процесом;

RES – обсяг фізичної пам’яті, який використовується процесом;

SHR – обсяг розділюваної пам’яті процесора;

S – вказує на статус процесу: *S=sleep* (очікує подій), R=*running* (виконується), Z=zombie (очікує батьківський процес) (S);

%CPU – відсоток використання центрального процесора цим процесом;

%MEM – відсоток використання оперативної пам’яті цим процесом;

TIME+ – загальний час активності процесу;

COMMAND – ім’я процесу.

4.2.3. Пріоритети процесів

Планувальник процесів CFS (*Completely Fair Scheduler*) – це підсистема ядра, яка розподіляє обмежене значення ресурсу часу процесора між системними процесами. Тобто планувальник процесів (або просто планувальник) – це компонента ядра, що забезпечує вибір процесу, який буде виконуватися наступним. Приймаючи рішення, які процеси можуть бути запущені та коли, планувальник відповідає за максимізацію використання процесора одночасно із забезпеченням видимості того, що декілька процесів виконуються одночасно, не заважаючи один одному. Linux не планує процеси абияк. Кожному процесу планувальник ОС Linux призначає *пріоритет*, який впливає на те, як довго буде працювати процес. Це пояснюється тим, що частка ресурсів процесора, яка призначається процесу, зважується відповідно до його значення «ввічливості» (*niceness*). Пріоритети в Linux називають *значеннями ввічливості (NI),* оскільки їх основна ідея – «бути ввічливими» щодо інших процесів завдяки проходженню процесної пріоритетності, дозволяючи іншим процесам споживати більше системного процесорного часу. Ядро Linux виділяє завданням з більш високим пріоритетом більший квант часу, а завданням із меншим пріоритетам – менший квант часу, який в середньому становить 200 і 10 мс відповідно.

Діапазон значень ввічливості такий: від «**-20**» (найвищий пріоритет) до «**19**» (нижчий пріоритет) включно, а значення за замовчуванням – 0 (процеси, запущені звичайними користувачами, зазвичай мають пріоритет 0). Таким чином, чим нижче значення ввічливості процесу, тим вищий його пріоритет і більше квант часу.

Linux надає декілька викликів для отримання та установки значення ввічливості процесу. Найпростіший із них – ***nice()***:

#include <unistd.h>

int nice(int inc);

Успішна робота *nice()* збільшує значення ввічливості процесу на значення ***inc*** й повертає оновлену величину [10]. Тільки процес із характеристикою CAP_SYS_NICE (яким фактично володіє користувач *root*) може встановити негативну величину *inc*, зменшуючи його значення ввічливості й, таким чином, підвищуючи пріоритет процесу. Процеси без прав *root* можуть тільки знижувати свої пріоритети (збільшуючи значення ввічливості).

Кращим рішенням є застосування системних викликів ***getpriority()*** та ***setpriority()***, які надають більше можливостей для керування і контролю, але складніші у використанні:

#include <sys/time.h>

#include <sys/resource.h>

int getpriority(int which, int who);

int setpriority(int which, int who, int prio);

Ці виклики впливають на процес, групу процесів або користувачів, що визначено за допомогою *which* та *who*.

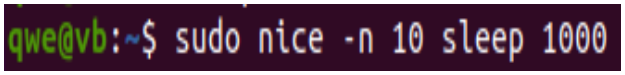
Значення *which* має бути PRIO_PROCESS, PRIO_PGRP або PRIO_USER, тоді як *who* вказує ідентифікатор процесу, групи процесів або ідентифікатор користувача відповідно. Якщо значення *who* дорівнює 0, то

виклик працює з поточними ідентифікатором процесу, групи процесів або ідентифікатором користувача відповідно.

Виклик *getpriority()* повертає найбільший пріоритет (найменшу чисельну величину значення ввічливості) кожного із зазначених процесів. Виклик *setpriority()* встановлює значення пріоритету кожного із зазначених процесів, яке дорівнює *prio*. Як і *nice()*, тільки процес із властивістю CAP_SYS_NICE може збільшити пріоритет процесу (знизити чисельне значення ввічливості). Отже, тільки процес із цією властивістю може збільшити або зменшити пріоритет процесу, що не належить користувачу, який викликає.

Використання команди ***nice***: ***nice [-n <число>]<команда>***

Команда *вказує пріоритет*, з яким процес буде виконуватися після запуску. Від'ємне значення пріоритету має право вказувати виключно суперкористувач (root). Якщо ключ «-n» не був введений у команді, то використовують число 10: *sudo nice -10 sleep 1000*



Для зміни пріоритету процесу використовують команду

renice -n <число> -p <список ідентифікаторів процесів>

Команда *renice* працює аналогічно *nice*, але змінюється не пріоритет створюваного процесу, а пріоритети усіх запущених процесів, ідентифікатори яких входять у список, розділений пробілами та переданий команді як параметр. Число у складі ключа «**-n**» додається до поточного пріоритету процесу, що пришвидшує виконання процесу. Ключ «**-p**» означає *priority [-p]*, тобто новий пріоритет. Така зміна виконується винятково від імені адміністратора (root): *sudo renice -n -15 -p 1000.*

Змінили пріоритет процесу 1000 на -15.

4.2.4. Керування сигналами

Одним зі способів взаємодії між процесами в Linux є сигнали.

Сигнали – це програмні переривання, які можуть бути послані процесу за допомогою системного виклику. Кожному виду сигналів у системі відповідають назва і номер. Багато сигналів мають спеціальний сенс. Зокрема, сигнал SIGTERM повідомляє процесу про необхідність завершення, сигнал SIGTERM використовують для безумовного завершення процесу, сигнали SIGSTOP та SIGCONT, відповідно, зупиняють і відновлюють виконання процесу. Сигнали SIGKILL (примусове завершення процесу) та SIGSTOP неможливо ні перехопити, ні ігнорувати, тому що вони адресовані не процесу, а планувальнику ОС.

Щоб надіслати сигнал процесу із програми-оболонки, використовують команду:

kill [-s <назва сигналу>] <список ідентифікаторів процесів>

Для виведення усіх сигналів, які підтримує ця система, використовують ключ «***-l»: kill -l***:

```
qwe@vb:~$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

Для зупинення виконання певного процесу, наприклад з ідентифікатором 123, треба скористатися командою ***kill -s SIGSTOP 123***. Для його відновлення використовують команду ***kill -s SIGCONT 123***.

Для завершення процесу з ідентифікатором 123 використовують команду ***kill 123***.

Найбільш вживані сигнали в Linux:

SIGINT (№ 2) – користувач натиснув Ctrl+C;

SIGKILL (№ 9) – припинення виконання процесу;

SIGTERM (№ 15) – попередження, що процес незабаром буде знищений;

SIGCONT (№ 18) – продовження призупиненого процесу;

SIGSTOP (№ 19) – припинення виконання процесу;

SIGTSTP (№ 20) – натискання клавіш <CTRL>+<Z> сигналу викликає зупинку виконання процесу.

Команда ***killall -s SIGNAL процес*** надсилає сигнал усім процесам з іменем ***процес***. Якщо сигнал не вказаний, надсилають SIGTERM:

killall mc – завершити процес, в імені якого є mc;

killall -u user – завершити процеси, які породжені користувачем user.

Системний виклик ***kill()***, який використовує утиліти *kill*, надсилає сигнал від одного процесу до іншого:

```
#include <sys / types.h>
#include <signal.h>
int kill(pid_t pid, int signo);
```

У звичайній ситуації (коли *pid* більший від нуля) *kill()* надсилає сигнал *signo* процесу з ідентифікатором *pid*. Якщо параметр *pid* дорівнює нулю, *signo* надсилається всім процесам, що належать до тієї ж групи, що і викликаючий процес. Якщо *pid* дорівнює -1, то сигнал *signo* надсилається всім тим процесам, яким процес виклику може відправити сигнал (має на це право доступу), за винятком *init* та себе самого. Якщо *pid* менший за -1, то сигнал надсилається групі процесів *-pid*.

У разі успіху *kill()* повертає 0. Виклик вважають успішним, як тільки вдасться надіслати один сигнал. У разі помилки (жодного сигналу надіслати не вдалося) виклик повертає -1 і присвоює *errno* одне з таких значень:

EINVAL – сигнал, позначений *signo*, є неприпустимим;

EPERM – процес виклику не має достатніх прав доступу, щоб надіслати сигнал будь-якому процесу, щодо якого здійснено запит;

ESRCH – процес або група процесів, що визначені *pid*, не існує або (в разі процесу) це процес-зомбі.

4.3. Потоковість

Сучасні ОС та мікропроцесори вже давно підтримують багатозадачність і, разом з тим, кожне з цих завдань може виконуватися в декілька потоків. Це дає відчутний приріст продуктивності обчислень і дозволяє краще масштабувати користувацькі додатки й сервери, однак за це доводиться платити ціну – ускладнюється розроблення програми та її налагодження.

Потоковість – це створення і керування безліччю виконуваних елементів всередині одного процесу. Потік є єдиним джерелом програмних помилок за рахунок впровадження конкурентності даних і клінчів (взаємних блокувань). Розглянемо основи потоковості API в Linux, щоб розібратися, як потоковість вписується в системний інструментарій програміста; коли потрібно використовувати потоки і, що важливіше, коли не потрібно; які шаблони проєктування можуть допомогти спланувати і побудувати потокові додатки; що таке конкуренція даних і як їй можна запобігти.

Бінарні модулі, процеси і потоки

Бінарні модулі – це програми, що перебувають у пасивному стані у сховищі і скомпільовані у форматі, прийнятному для певної ОС та машинної архітектури, готові до запуску, але ці поки не діють.

Процеси – це абстракції ОС, що являють собою конкретні модулі в дії: завантажений бінарний код, віртуальна пам’ять, ресурси ядра, такі як відкриті файли, пов’язаний користувач і под., тобто це *програма та системні ресурси, необхідні для її роботи.*

Однозначна відповідність між програмою і процесом встановлюється тільки у конкретний момент часу: *один процес у різний час може виконувати код декількох програм, код однієї програми можуть виконувати декілька процесів одночасно*. Для ОС процес є засобом організації багатьох задач, які вона повинна виконувати. Операційна система виділяє кожному процесу порцію системних ресурсів і гарантує, що програма кожного процесу буде направлятися на виконання у певному порядку та своєчасно.

Для успішного виконання програми потрібні певні ресурси. До них належать:

– ресурси, потрібні для послідовного виконання програмного коду (передусім процесорний час);

– ресурси, що дають можливість зберігати інформацію, яка забезпечує виконання програмного коду (регістри процесора, оперативна пам’ять тощо).

Ці групи ресурсів визначають дві складові частини процесу:

– послідовність виконуваних команд процесора;

– набір адрес пам’яті (адресний простір), в якому розміщені ці команди й дані для них.

Виділення цих частин виправдане ще й тим, що в межах одного адресного простору може бути кілька паралельно виконуваних послідовностей команд, що спільно використовують одні й ті самі дані. *Необхідність розмежування послідовності команд і адресного простору підводить до поняття потоку.*

Процес розбивається на виконувані одиниці – потоки (threads) (один і більше потоків). *Потік – це базова одиниця завантаження процесора*, що складається з *ідентифікатора потоку, лічильника, регістрів і стеку*.

Потоки всередині одного процесу поділяють на секції: коду, даних, а також різні ресурси: описувачі відкритих файлів, облікові

дані процесу, сигнали, значення *umask* (утиліта */usr/bin/umask* задає маску режиму створення файлу в поточному середовищі командного інтерпретатора і впливає на всі дочірні процеси, виконувані в цій оболонці; режим повного доступу для каталогів – 777, для файлів – 666), *nice* (пріоритет процесу), таймери та інше [1].

Потоки дозволяють процесу паралельно виконувати різні частини його програми та ефективно використовувати процесор, особливо на багатопроцесорних комп'ютерах.

Потік – набір послідовностей виконуваних команд процесора, які овикористовують загальний адресний простір процесу. Це елемент виконання всередині процесу: віртуальний процесор, стек або статус програми.

Інакше кажучи, *процеси – це виконувані бінарні модулі,* і потоки є найменшими виконуваними елементами, передбаченими планувальником ОС.

Якщо процес містить тільки один потік, то у процесі міститься лише один виконуваний елемент і тільки одна задача виконується за одиницю часу. Такі процеси можна назвати однопотоковими (рис. 4.1). Це класичні процеси UNIX. Якщо процес містить більше одного потоку, то одночасно виконується кілька дій. Такі процеси називають багатопотоковими (рис. 4.1).

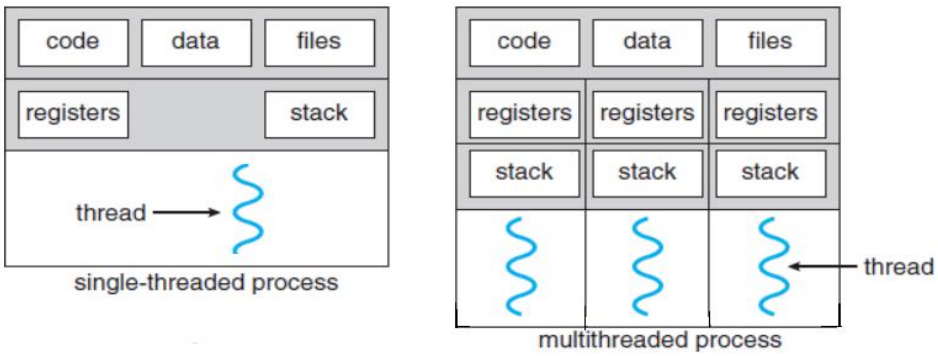


Рис. 4.1. Потоки [1]

Сучасні ОС представляють у просторі користувача дві основні віртуальні абстракції – віртуальну пам'ять і віртуальний процесор. Спільно вони створюють для кожного виконуваного процесу ілюзію, що він один користується всіма ресурсами машини.

Віртуальна пам'ять надає кожному процесу унікальний вид пам'яті, який нібито відповідає оперативній пам'яті або сховищу на диску (це досягається за допомогою розбивання на сторінки). Оперативна пам'ять системи насправді може містити дані сотні різних процесів, запущених одночасно, але кожному з них здається, що вся пам'ять належить тільки йому. *Віртуальний процесор дозволяє кожному процесу працювати так, наче він один у системі*, а ОС приховує, що кілька процесів працюють одночасно на (можливо) декількох процесорах.

Віртуальна пам'ять пов'язується із процесом, а не з потоком. Таким чином, *кожний процес має діапазон комірок оперативної пам'яті*, який належить винятково йому, але усі потоки в цьому процесі поділяють цю пам'ять. І навпаки, *віртуальний процесор пов'язаний саме з потоками, а не з процесами*. Кожний потік – це виконання певної запланованої дії, яка дозволяє кожному процесу виконувати більше однієї дії у кожний момент часу.

Відмінність у поняттях «процес» і «потік» полягає у тому, що для виконання потоку потрібні усі види ресурсів – процесорний час, пам'ять, файли, пристрої «введення – виведення» тощо; для виконання процесу потрібні усі наведені вище ресурси, окрім процесорного часу.

4.3.1. Багатопотоковість

Процеси зазвичай потрібні тому, що вони є абстракцією запущеної програми. Однак є причина, чому розділяють елементи виконання і вводять поняття «потоки». Розглянемо шість основних переваг багатопотоковості.

• *Програмна абстракція*. Розподіл роботи та присвоєння кожній одиниці виконання (потоку) – це природний підхід до вирішення багатьох завдань. У шаблони проектування, які використовують цей підхід, входять «один потік на одне з'єднання» і «пул потоків».

• *Паралелізм*. У машинах з декількома процесорами потоки забезпечують ефективний спосіб досягнення справжнього паралелізму. Кожний потік отримує власний віртуальний процесор, де його планування не залежить ні від чого, тому декілька потоків можуть виконуватися на декількох процесорах в один і той же час, покращуючи пропускну здатність системи.

• *Поліпшення реагування*. Навіть на однопроцесорній машині багатопотоковость може прискорити реагування процесів. Під час однопотокового процесу операція, яка виконується тривалий час, може зашкодити додатку зреагувати на якусь дію користувача, якому може здатися, що додаток «завис». При багатопотоковості такі операції можуть бути делеговані робочим потокам, дозволивши хоча б одному потоку залишатися здатним реагувати на дії користувача і виконувати операції «введення – виведення».

• *Блокування* «введення – виведення». Якщо немає потоків, блокування «введення – виведення» зупиняє весь процес. Це може згубно впливати як на загальну пропускну здатність, так і на час затримки. У багатопотоковому процесі індивідуальні потоки можуть блокуватися, очікуючи «введення – виведення», поки інші потоки продовжують роботу. Асинхронність і неблокування «введення – виведення» можуть бути альтернативними рішеннями цього завдання.

• *Перемикання контексту*. Витрати на перемикання від одного потоку до іншого всередині того самого процесу значно менші, ніж переміщення контексту від одного процесу до іншого.

• *Економія пам'яті*. Потоки забезпечують ефективний спосіб поділу пам'яті, одночасно дозволяючи виконання декількох елементів. Таким чином вони є альтернативою декільком процесам.

З названих вище причин потоковість досить поширена в різних ОС та їх додатках. Але багатопотоковість має й недоліки; під час багатопотокового процесу виконується одночасно кілька дій (паралелізм), які поділяють *одну і ту саму область пам'яті*, тобто потоки у процесі поділяють між собою ресурси, що може призвести до збою, оскільки потоки виконуються незалежно один від одного, з непередбачуваним вибором часу виконання та інструкцій, необхідних для правильної роботи. *Збій у синхронізації може призвести до перекриття виведення, некоректного виконання і загалом припинення роботи програми*.

Альтернативи багатопотоковості

Залежно від цілей користувача, пов'язаних із багатопотоковістю, можливі альтернативи. Наприклад, вигоди в затримці виконання та «введення – виведення» також є досяжними за допомогою мультиплексного, неблокованого та асинхронного «введення – виведення». Ці техніки дозволяють процесам здійснювати операції «введення – виведення», не блокуючи процес.

4.3.2. Потокові моделі

Майже всі сучасні ОС – зокрема Windows, Linux, Mac OS X, і Solaris – підтримують керування потоками в режимі ядра. Однак *потоки можуть бути створені не тільки в режимі ядра, але і в режимі користувача.* Тому є дві основні категорії реалізації потоків: *користувацькі потоки* реалізуються через спеціальні бібліотеки потоків і працюють у просторі користувача, а *потоки ядра* реалізуються через системні виклики і працюють у просторі ядра.

У разі використання цього рівня ядро не знає про існування потоків – усе керування потоками реалізується додатком за допомогою спеціальних бібліотек. Потоки користувача по-різному відображаються на потоки в режимі ядра. Є три моделі, з яких 1:1 є найбільш часто вживаною.

Найпростіша модель полягає в тому, що ядро забезпечує свою вбудовану підтримку потоків, і кожний з них безпосередньо відправляє у простір користувача свою інформацію. Таку модель називають потоковістю 1:1, оскільки в ній наявне **співвідношення 1:1 –** між тим, що надає ядро, і тим, що отримує користувач. Ця модель також відома як **потоковість на рівні ядра**, оскільки ядро є основою системної потокової моделі.

Ядро Linux реалізує потоки як процеси, які розділяють між собою ресурси. Потокова бібліотека створює новий потік через системний виклик *clone()*, і «процес», який повертається безпосередньо, керований як поняття потоку у просторі користувача. Таким чином у Linux те, що є потоком у просторі користувача і в цілому є тим самим і з точки зору ядра.

Розглянемо наступну модель – ***потоковість на рівні користувача N:1*** (рис. 4.2) [1].

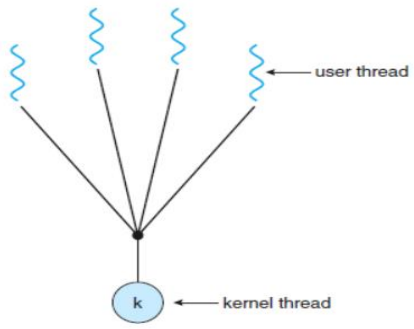


Рис. 4.2. Потоковість на рівні користувача N:1

На відміну від потоковості на рівні ядра, сутність цієї моделі полягає в тому, що кілька користувацьких потоків відображаються на один потік ядра ОС. Все керування потоками здійснює особлива бібліотека користувача, і саме в цьому і полягає перевага такого підходу. Недолік же

в тому, що якщо один-єдиний потік виконує блокувальний виклик, то гальмується весь процес.

Третя модель – ***комбінована потоковість***, *потоковість N:M*, дозволить взяти найкраще від обох підходів: ядро забезпечує вбудовану підтримку потоків, тоді як простір користувача реалізує потоки користувача. Простір користувача за участі ядра вирішує, як зіставити N потоків користувача з M потоками ядра, де $N \geq M$. Така модель значно складніша для програмування і впровадження.

Для Linux залишається найбільш популярною модель 1:1.

Шаблони потоковості

Під час побудови потокового додатка є два основних шаблони програмування – *це потік на з'єднання і потік, керований подією.*

Потік на з'єднання – це шаблон програмування, в якому одному елементу виконання призначається один потік, і цей потік призначається не більше ніж одному робочому елементу протягом всієї роботи. *Робочим елементом* можна назвати все, на що можна розкласти роботу застосування користувача: *запит, з'єднання* і под. Це може бути веб-сервер, який здатний обробити одночасно значну кількість запитів. Для шаблону потоку на з'єднання це означає величезну кількість потоків. Потоки викликають певні витрати, насамперед ресурсів ядра і стеку користувацького простору.

Потік, керований подією

Системні розробники помітили, що більшість потоків багато часу проводять в очікуванні: читання файлів, повернення результатів базами даних, відпрацювання віддалених процедур, тобто потоки через з'єднання просто очікують. Тому було запропоновано відокремити очікування від потоків, організувавши асинхронне «введення – виведення» (без інтервалів часу періодичноого передавання інформації на «введення – виведення») та

застосувавши мультиплексне «введення – виведення» (це одночасне виконання декількох операцій «введення – виведення» на декількох зовнішніх пристроях) для керування процесом на сервері.

У цій моделі обробка запитів перетворена в серію асинхронних запитів «введення – виведення» і пов'язаних з ними зворотних викликів, які можуть очікуватися через мультиплексне «введення – виведення»; якщо процес надходить саме так, його називають *циклом подій*. Коли запити «введення – виведення» повертаються, цикл подій знову передає зворотний виклик потоку, що очікує. Наприклад, у багатопотоковому сервері Apache, який розроблений за шаблоном, керований подією, є: асинхронне «введення – виведення», зворотні виклики, цикл подій і невелика кількість потоків – по одному на кожний процесор.

Конкурентність, паралелізм і гонки

Потоки породжують два взаємопов'язаних, але різних явища: конкурентність і паралелізм.

Конкурентністю називають здатність двох і більше потоків виконуватися в періоди часу, які перекриваються.

Паралелізм – здатність виконувати два (і більше) потоки одночасно. Конкурентність – це шаблон програмування, спосіб підходу до вирішення проблеми, паралелізм – функціональність апаратного забезпечення, яка досягається через конкурентність. Обидва явища корисні.

Умови гонки

При одночасній роботі потоки можуть виконуватися у непередбачуваному порядку один щодо одного. Формально *умовами гонки* називають ситуацію, коли несинхронізований доступ до загального ресурсу для двох і більше потоків призводить до помилкової поведінки програми. Загальним ресурсом може бути що завгодно: апаратне забезпечення системи, ресурси ядра, дані у пам'яті. Саме дані у пам'яті –

форма, яка трапляється найбільш часто та відома під назвою *гонки за даними*. Вікно, в якому може статися така гонка, – область коду, яка має бути синхронізована, – називають *критичною областю*. Гонки усуваються синхронізацією коду в критичних областях. Розглянемо кілька прикладів умов гонок.

Гонки в реальності

Уявімо банкомат (АТМ). Людина вставляє картку, вводить ПІН-код і позначає, кількість скільки грошей хоче зняти. Після цього отримує готівку. У процесі цієї операції банк має перевірити, чи є необхідна сума рахунку людини, і якщо так, відняти суму зняття. Алгоритм вийде приблизно такий:

1. Чи є на рахунку як мінімум X одиниць коштів?

2. Якщо так, зменшити розмір рахунку на величину X і видати користувачу X грошових одиниць.

3. Якщо немає, видати повідомлення про помилку.

Код на С буде виглядати приблизно так:

```
int withdraw (struct  account  *account, int amount)
{
      const int balance=account-> balance;
      if (balance <amount)
            return -1;
      account-> balance=balance – amount;
      disburse_money (amount);
      return 0;
}
```

Між тим за короткий час відбувається кілька операцій одночасно: і клієнт знімає гроші, і банк обробляє онлайн оплату його рахунків або стягує будь-який регулярний збір, – це якраз і є прикладом гонок. Тому що, наприклад, якщо на рахунку було 500 $ й одночасно прийшли два запити на зняття 200 $ та 400 $, вони обидва можуть бути виконані, хоча на рахунку виявиться негативна величина у мінус $ 100, чого, очевидно,

не бажав допустити програміст, який написав код. Насправді практично кожний рядок цієї функції перебуває всередині критичної області. Щоби банк не прогорів, потрібно синхронізувати доступ до функції *withdraw(),* забезпечивши гарантію, що навіть якщо будуть виконуватися конкурентно два і більше потоки, сама функція буде виконуватися як один атомарний елемент: банк відкриває баланс рахунку, перевіряє доступні засоби, а потім змінює їх кількість для кожної транзакції індивідуально.

Синхронізація

Основним джерелом появи гонок є такі критичні області, як вікна, в яких передбачається неприпустимість одночасного виконання потоків при коректній роботі програми. Щоб попередити виникнення умов гонки, програміст повинен синхронізувати доступ до цього вікна, забезпечивши взаємовиключний доступ до критичної області. Проблема критичних областей полягає у тому, що вони не є неподільними, тобто не виконуються миттєво, отже, вони не атомарні.

М’ютекси

Є безліч способів зробити критичні області атомарними: від поодиноких інструкцій до великих блоків коду. Найпоширеніша техніка – *замок, тобто механізм, що забезпечує взаємне виключення доступу до критичної області*, роблячи її атомарною. Замикання (рос. – запирание, то есть закрытие, блокировка) забезпечує взаємне блокування потоків, тому воно відоме у стандартній бібліотеці *Pthreads* (і повсюдно) як *м’ютекси* (mutexes – буквально «взаємно виключні», бінарні семафорні механізми).

М’ютекси реалізовані в багатьох ОС, їх основне призначення – організація взаємного виключення для потоків з одного і того ж або з різних процесів, тобто вони слугують у програмуванні для синхронізації потоків, які виконуються одночасно.

М'ютекси – це найпростіші двійкові семафори, які можуть перебувати в одному із двох станів – зазначеному або невідзначеному (відкритий і закритий відповідно) [1]. Коли будь-якої потік, що належить будь-якому процесу, стає власником об'єкта mutex, останній переводиться у невідзначений стан. Якщо завдання звільняє м'ютекс, його стан стає зазначеним. М'ютекс являє собою концепцію програмування, яка використовується для вирішення питань багатопотоковості. Завдання м'ютекса – захист об'єкта від доступу до нього інших потоків, відмінних від того, який заволодів м'ютексом. У кожний конкретний момент тільки один потік може володіти об'єктом, який захищений м'ютексом. Якщо іншому потоку буде потрібний доступ до змінної, яка захищена м'ютексом, то цей потік засинає доти, поки м'ютекс не буде звільнений. М'ютекс набуває двох значень:

– *відкритий* – потік може увійти у свою критичну секцію;

– *закритий* – потік не може увійти у критичну секцію.

Мета використання м'ютексів – захист даних від пошкодження, проте вони породжують інші проблеми – приміром, взаємне блокування (клінч) і стан гонки.

М'ютекси в Unix-подібних системах реалізовані у стандартній бібліотеці **Pthreads** [23]. Знову розглянемо приклад із банком: як м'ютекс міг би попередити катастрофічні (у всякому разі, для банку) умови гонки. Маємо функції *lock()* та *unlock()* для замикання і відмикання м'ютекса відповідно:

```
int withdraw (struct account *account, int amount)
{
    lock();
    const int balance=account-> balance;
    if (balance <amount) {
        unlock();
        return -1;
    }
```

```
        account-> balance = balance - amount;
        unlock();
        disburse_money(amount);
        return 0;
    }
```

Замикаємо тільки частину функціоналу, де може статися гонка: зчитування балансу рахунку, перевірку наявності необхідних коштів і оновлення балансу. Таким чином, усі потоки мають отримати в потрібних місцях права на необхідні замки, але потік може отримати несанкціонований доступ до замка, тому це потребує сумлінного програмування.

Взаємні блокування

Потоковість потрібна для забезпечення конкурентності, яка створює умови гонки. Для боротьби з гонками використовують м'ютекси, які можуть бути новим джерелом програмних помилок – *взаємних блокувань*. ***Взаємне блокування*** – це ситуація, коли два потоки чекають закінчення роботи один одного і, таким чином, жодний з них не може закінчитися. Прикладом помилки багатопотоковості є робота марсохода Mars Pathfinder у 1977 році, коли його кліматичні та геологічні дослідження були порушені частими перезавантаженнями системи. Серед безлічі різних потоків виявилося три, які й привели до збою: потік з низьким пріоритетом, що забезпечував збір метеорологічних даних, потік із середнім пріоритетом, який відповідав за зв'язок із Землею, і потік з високим пріоритетом, який керував сховищем всього марсоходу. Все йшло добре, поки випадково потік зв'язку не активувався і не запустився у той самий момент, коли метеорологічний потік утримував м'ютекс, а потік сховища очікував його. Для забезпечення виконання завдання потоку зв'язку метеорологічний не запуститься. Це і є класичний приклад помилок.

Р-потоки

Ядро Linux забезпечує тільки базові примітиви для забезпечення потоковості, наприклад системний виклик *clone().*

Основна частина будь-якої бібліотеки потоків перебуває у просторі користувача. POSIX регламентує бібліотеку потоків з IEEE Std 1003.1c-1995 року, відому як POSIX 1995 або POSIX.1c.

Розробники називають цей стандарт *POSIX-потоками* або *Р-потоками*.

4.3.3. Реалізація потоковості в Linux

У Linux реалізація стандарту Р-потоки здійснюється за допомогою ***glibc***, бібліотеки С Linux. Згодом *glibc* забезпечила **дві різні реалізації -потоків**: **LinuxThreads** і **NPTL** (***Native POSIX Thread Library***).

LinuxThreads забезпечує потоковість 1:1, була розроблена для ядра, яке забезпечувало досить слабку підтримку потоковості: крім системного виклику *clone(),* який створював новий потік, LinuxThreads реалізувала потоковість POSIX, використовуючи наявні інтерфейси UNIX.

Native POSIX Thread Library (NPTL) замінила LinuxThreads і залишається стандартною реалізацією Р-потоків у Linux, забезпечує потоковість 1:1, що ґрунтується на системному виклику *clone()* і моделі ядра, де потоки розглядаються як будь-які інші процеси, за винятком здатності потоків розділяти між собою деякі ресурси. На відміну від LinuxThreads, NPTL має вигоду з додаткових інтерфейсів ядра і містить системний виклик *futex()* для синхронізації потоків, системний виклик *exit_group()* для припинення всіх потоків у процесі й підтримку ядра для локального потокового сховища (TLS).

API для роботи з Р-потоками

Application Programming Interface (API) – інтерфейс програмування додатків. Ядро потокових бібліотек у системах UNIX побудовано на

Р-потоках. API для роботи з Р-потоками визначається у заголовку <*pthread.h*>. Кожна функція в API забезпечена префіксом ***pthread_*.** Наприклад, **функція для створення потоку – це *pthread_create().*** Функції Р-потоків можуть бути розбиті на дві великі групи:

– керування потоками – функції для створення, знищення, з'єднання і від'єднання потоків (їх буде розглянуто далі у цьому розділі);

– синхронізація – функції для керування синхронізацією потоків, що включають м]ютекси, умовні змінні та бар'єри.

Зв'язування Р-потоків

Хоча Р-потоки забезпечуються *glibc*, вони містяться в окремій бібліотеці ***libpthread***, яка вимагає явної прив'язки. Для *gcc* (із ключами компіляції попередження -*Wall* -*Werror*) це забезпечується автоматично за допомогою прапорця ***-pthread***, який гарантує, що з виконуваним модулем пов'язана потрібна бібліотека:

gcc -Wall -Werror -pthread beard.c -o beard

Створення потоків

При першому запуску програми і виконання функції *main()* програма є однопотоковою. З цього початкового потоку, який іноді називають *потоком за замовчуванням, або головним потоком*, можна створити один або кілька додаткових, щоб запустити багатопотоковість.

Р-потоки забезпечують одну функцію для визначення і запуску нового потоку, яку називають *pthread_create()*:

```
#include <pthread.h>
int pthread_create(pthread_t  *thread,  const pthread_attr_t  *attr,
void *(*start_routine) (void *), void  *arg);
```

Ця функція визначена в заголовку <***pthread.h***>.

Перший параметр цієї функції є вказівником на змінну типу ***pthread_t***, в яку буде записано адресу ідентифікатора створюваного потоку – ID.

Другий параметр є вказівником на змінну типу ***pthread_attr_t*** й використовується для встановлення атрибутів потоку. Цей об'єкт керує деталями взаємодії потоку з іншою програмою. Атрибути потоку дозволяють програмам змінювати безліч властивостей потоку, таких як розмір стеку, параметри планування, початковий окремий статус. Якщо параметр дорівнює NULL, то потік буде створений з атрибутами за замовчуванням.

Третім параметром функції *pthread_create* має бути адреса функції потоку – вказівник на функцію потоку. Ця функція відіграє для потоку ту ж роль, що й функція *main* для головної програми. Функція потоку набуває один параметр типу покажчик на *void* і повертає значення типу вказівник на *void*.

Четвертий параметр функції *pthread_create* має тип void*. Цей параметр можна використовувати для передавання значення як аргумент у функцію потоку. Через нього можна передавати новому потоку параметри.

Спочатку створюється потокова функція. Далі новий потік створюється функцією *pthread_create()*, яка об'явлена у заголовковому файлі *pthread.h.* Після виклику *pthread_create* функція потоку буде запущена на виконання паралельно з іншими потоками програми.

Функція *start_routine* повинна мати запис такого вигляду:

void *start_thread(void *arg);

Таким чином потік починає існування з виконання функції, яка приймає вказівник *void* як єдиний аргумент, а потім повертає його як свою величину, яка повертається. Аналогічно *fork()* новий потік успадковує більшість атрибутів, сумісність і статус від свого батька. Однак, на відміну від *fork(),* потоки *поділяють* ресурси зі своїм батьком замість отримання

копії. Найважливішим ресурсом є, зазвичай, адресний простір процесу, але потоки також ділять (замість отримання копії) обробники сигналів і відкриті файли.

Код, який використовує цю функцію, повинен відправляти ***-pthread*** в *gcc*. Це відноситься до всіх функцій Р-потоків.

В разі помилки *pthread_create()* повертає ненульовий код помилки (без використання *errno*) і вміст потоку *thread* є невизначеним. Помилки:

EAGAIN – викликаючому процесу істотно бракує ресурсів для створення нового потоку; зазвичай це викликано тим, що процес досяг границі кількості потоків для кожного користувача або для всієї системи;

EINVAL – об'єкт *pthread_attr_t*, зазначений через *attr*, містить неприпустимі атрибути;

EPERM – викликаючий процес не має повноважень для встановлення деяких атрибутів об'єкта *pthread_attr_t*, вказаного через *attr*.

Приклад використання:

```
#include <pthread.h>
pthread_t tread;
int ret;
ret=pthread_create(&thread, NULL, start_routine, NULL);
if (! ret) {
        errno= et;
        perror("pthread_create");
        return -1;
}
/* Новий потік створений і паралельно виконує start_routine ... */
```

Ідентифікатори потоків

Ідентифікатори потоків (TID -*Thread ID***)** для потоків є аналогами ідентифікаторів процесів (PID). У той час як *PID призначаються ядром Linux, TID призначаються лише бібліотекою Р-потоків*. Цей тип представлений *pthread_t*, і POSIX не вимагає, щоб він був арифметичним. TID нового потоку визначається за допомогою аргументу *thread* при

успішному виклику *pthread_create()*. Потік може отримати свій TID при запуску за допомогою функції ***pthread_self()***:

```
#include <pthread.h>
pthread_t pthread_self (void);
```

Використання функції:

```
const  pthread_t me=pthread_self();
```

Порівняння ідентифікаторів потоків. Стандарт Р-потоків не вимагає, щоб *pthread_t* був арифметичного типу, тому немає гарантії, що оператор рівності буде працювати. Отже, *щоб порівняти ідентифікатори потоків*, бібліотеці Р-потоків *потрібен спеціальний інтерфейс*:

```
#include <pthread.h>
int pthread_equal(pthread_t  t1, pthread_t  t2);
```

Якщо наведені ідентифікатори потоків рівні, то *pthread_equal()* повертає ненульову величину. Якщо вони не рівні, повертається 0; помилка не може відбутися. Ось простий приклад:

```
int ret;
ret=pthread_equal (thing1, thing2);
if (ret! = 0)
        printf("The TIDs are equal!\n");
else
        printf("The TIDs are unequal!\n");
```

Завершення потоків

Завершення потоків дуже схоже на завершення процесів, за винятком того, що, коли потік завершується, інші потоки в процесі продовжують виконуватися. В деяких потокових шаблонах, таких як потік на з'єднання, потоки часто створюються і знищуються.

Потоки можуть перериватися за певних обставин, які мають аналоги в завершенні процесів:

– якщо потік повертається зі стартової процедури, він переривається; це аналог «виходу за границі» в *main()*;

– якщо потік викликає функцію *pthread_exit()*, він завершується; це аналог виклику *exit()*;

– якщо потік скасовується іншим потоком через функцію *pthread_cancel(),* він завершується; це аналог відправки сигналу SIGKILL через *kill().*

У наведених трьох прикладах завершується тільки потік, на який спрямована дія. *Всі потоки в процесі завершуються, зупиняючи таким чином сам процес, при таких обставинах:*

– процес повертається зі своєї функції *main()*;

– процес завершується через *exit()*;

– процес виконує новий двійковий образ через *execve().*

Сигнали можуть вбити процес або окремий потік в залежності від того, як вони спрямовані. Р-потоки здійснюють обробку сигналів трохи складніше, тому краще мінімізувати використання сигналів в багатопотокових програмах.

Самозакінчення

Найпростіший шлях потоку для завершення самого себе, – це «вихід за границі» своєї початкової процедури. Однак часто потрібно завершити потік десь в глибині стека виклику функції, достатньо далеко від стартової процедури. Для таких випадків у Р-потоках є виклик *pthread_exit(),* потоковий еквівалент *exit()*:

```
#include <pthread.h>
void pthread_exit(void  *retval);
```

Після виконання викликаючий потік завершується; *retval* (вилучення) забезпечується для кожного потоку, що очікує завершення.

Використання:

```
/* Прощавай, жорстокий світ! */
pthread_exit (NULL);
```

Дострокове завершення інших потоків

Р-потоки викликають завершення інших потоків через їх скасування. Це забезпечує функція *pthread_cancel()*:

```
#include <pthread.h>
int pthread_cancel(pthread_t  thread);
```

Успішний виклик *pthread_cancel()* надсилає запит на скасування потоку, представленому через ідентифікатор потоку *thread*. Чи може потік бути скасований і коли, залежить від його *стану відміни і типу скасування відповідно*. У разі успіху *pthread_cancel()* повертає 0. Завершення відбувається асинхронно. У разі помилки *pthread_cancel()* повертає ESRCH, що означає, що значення *thread* неприпустимо. Умови, за яких потік може бути скасований залежить від стану відміни. *Стан відміни потоку може бути доступно або недоступно*. За замовчуванням він є доступним для нових потоків. З іншого боку, тип скасування вказує, коли відбувається скасування. Потоки можуть змінювати свій стан через ***pthread_setcancelstate()***:

```
#include <pthread.h>
int  pthread_setcancelstate(int  state, int  *oldstate);
```

У разі успіху *стан відміни* викликаючого потоку встановлюється на *state*, а попередній стан зберігається в *oldstate*. Значенням state може бути *PTHREAD_CANCEL_ENABLE* або *PTHREAD_CANCEL_DISABLE* для *дозволу* або *заборони* скасування відповідно. В разі помилки *pthread_setcancelstate()* повертає EINVAL, що означає неприпустиме значення *state*.

Тип скасування потоку може бути *асинхронним* або *відкладеним*; за замовчуванням зазвичай встановлений останній. З асинхронним типом скасування потік може бути убитий в будь-якій точці після отримання команди на скасування. З відкладеним типом потік може бути убитий тільки в спеціальних точках скасування, які є функціями Р-потоків або

бібліотеки С і являють собою безпечні моменти, в яких викликаючий потік може бути перерваний. Асинхронне скасування може бути корисним лише в певних ситуаціях, так як воно може залишити процес у невизначеному стані. Наприклад, якщо потік, який необхідно скасувати, був десь в середині критичної області? Щоб програма вела себе коректно, асинхронне скасування повинно використовуватися тільки потоками, для яких не передбачено спільне використання будь-яких ресурсів і є можливим виклик виключно сигнально-безпечних функцій.

Потоки можуть змінити свій тип через *pthread_setcanceltype()*:

```
#include <pthread.h>
int  pthread_setcanceltype(int  type, int  *oldtype);
```

У разі успіху статус скасування викликаючого потоку встановлюється в *type*, а старий тип зберігається в *oldtype*. Значенням *type* може бути *PTHREAD_CANCEL_ASYNCHRONOUS* або *PTHREAD_CANCEL_ DEFERRED* для установки *асинхронного* або *відкладеного скасування* відповідно. У разі помилки *pthread_setcanceltype()* повертає EINVAL, що означає неприпустиме значення *type*. Розглянемо приклад, коли один потік повинен завершити інший. Спочатку потік, який повинен завершитися, дозволяє своє скасування і встановлює цей тип як відкладене скасування:

```
int unused;
int ret;
ret=pthread_setcancelstate PTHREAD_CANCEL_ENABLE,  &unused);
if (ret) {
      errno=ret;
      perror("pthread_setcancelstate");
      return -1;
}
ret=pthread_setcanceltype (PTHREAD_CANCEL_DEFERRED,  &unused);
if (ret) {
      errno=ret;
      perror("pthread_setcanceltype");
      return -1;
}
```

Потім інший потік посилає команду на завершення:

```
int ret;
/* 'thread' в даному випадку означає ідентифікатор потоку, який завершується * /
ret=pthread_cancel thread);
if (ret) {
        errno=ret;
        perror("pthread_cancel");
        return -1;
}
```

Приєднання і від'єднання потоків

Враховуючи що потоки досить просто створюються і знищуються, має бути і спосіб їх синхронізувати замість завершення інших потоків – еквівалент *wait()* для потоковості. Таким способом є *приєднання потоків*.

Приєднання потоків

Приєднання дозволяє одному з потоків блокуватися в очікуванні завершення іншого:

```
#include <pthread.h>
int pthread_join(pthread_t  thread,  void  **retval);
```

У разі успішного виклику викликаючий потік блокується доти, поки потік, вказаний як *thread*, не завершиться (якщо *thread* вже завершено, функція *pthread_join()* повертається негайно). Як тільки *thread* завершується, викликаючий потік активізується і, якщо *retval* не дорівнює NULL, отримує значення завершеного процесу, яке повертається, і передане *pthread_exit()* або повернене від його стартової процедури. Після цього можна сказати, що потоки *приєдналися* один до одного. *Приєднання завжди дозволяє потокам синхронізувати своє виконання по відношенню до періоду існування інших потоків*. Всі потоки в Р-потоках є рівноправними; кожний потік може приєднуватися до будь-якого іншого. Один потік може приєднуватися до багатьох (фактично, як ми скоро

побачимо, найчастіше один головний потік очікує інших потоків, які сам і створив), але тільки один потік може намагатися приєднатися до певного іншого, декілька потоків не повинні намагатися приєднатися до будь-якого одного. У разі помилки *pthread_join()* повертає один з таких ненульових кодів помилок:

EDEADLK – сталося взаємне блокування – *thread* вже чекає приєднання до викликаючого потоку або сам є викликаючим потоком;

EINVAL – неможливо приєднати потік, який визначений через *thread*;

ESRCH – значення thread неприпустимо.

Приклад використання:

```
int ret;
/* приєднуємо до 'thread' і більше не дбаємо про величину, яка повертається */
ret=pthread_join(thread, NULL);
if (ret) {
        errno=ret;
        perror("pthread_join");
        return -1;
}
```

Від'єднання потоків

За замовчуванням потоки створюються *здатними до приєднання*. Однак вони можуть і від'єднуватися, але в цьому разі вони надалі стануть неприєднаними. Оскільки до приєднання потоки використовують будь-які системні ресурси, як роблять це і процеси, поки їхні предки викликають *wait()*, потоки, які ви не плануються приєднувати, мають бути від'єднані:

```
#include <pthread.h>
int pthread_detach(pthread_t thread);
```

У разі успіху *pthread_detach()* від'єднує потік, вказаний як *thread*, і повертає 0. Результати не визначені, якщо викликано *pthread_detach()* щодо потоку, який вже від'єднаний. У разі помилки функція повертає значення ESRCH, що означає, що значення *thread* неприпустиме.

Для кожного потоку у процесі необхідно викликати *pthread_join()* або *pthread_detach()*, щоб системні ресурси могли вивільнитися після завершення потоку (зазвичай після того, як завершується весь процес, усі потокові ресурси вивільняються, але приєднання або від'єднання всіх процесів в явній формі залишається гарною практикою).

Приклад 4.5 (потоковості). Наступний приклад повної програми з'єднає всі інтерфейси, розглянуті вище. Програма створює два потоки (всього їх буде три), починаючи обидва в одній і тій самій стартовій процедурі *start_thread()*. Поведінка потоків у стартовій процедурі різниться аргументами. Потім обидва потоки приєднуються один до одного; якби цього не сталося, головний потік міг би завершитися раніше інших, перервавши весь процес:

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
void  *start_thread(void  *message)
{
      printf("%s\n",  (const  char *)message);
      return message;
}
int main (void)
{
      pthread_t thing1, thing2;
      const  char  *message1="Thing 1";
      const  char  *message2="Thing 2";
      /* створюються два потоки, кожний зі своїм повідомленням */
      pthread_create(&thing1, NULL, start_thread, (void *)message1);
      pthread_create(&thing2, NULL, start_thread, (void *)message2);
      /*
      * Очікування завершення потоків. Якщо ми не приєднаємо їх тут,
      * то ризикуємо знищити головний потік до того,
      * як інші потоки завершаться. */
      pthread_join (thing1, NULL);
      pthread_join (thing2, NULL);
      return 0;
}
```

Програму завершено. Якщо збережегти її як *example.c*, то можна скомпілювати її за допомогою такої команди:

gcc -Wall -02 -pthread example.c -o example

Потім запустити таким чином: ***./example***

На виході отримаємо:

Thing 1
Thing 2

Або, можливо:

Thing 2
Thing 1

Однак непотрібної інформації ніколи не буде, оскільки функція *printf()* безпечна для потоковості.

4.3.4. М'ютекси Р-потоків

Основним методом запобігання взаємних блокувань є м'ютекс. За всієї їх потужності й важливості м'ютекси досить прості у використанні.

Ініціалізація м'ютексів

М'ютекси представлені об'єктом ***pthread_mutex_t***. Як і більшість об'єктів в API Р-потоків, йдеться про їх непрозору структуру, що забезпечує різноманітність інтерфейсів м'ютексів.

Функція ***pthread_mutex_init ()*** призначена для ініціалізації м'ютекса:

int pthread_mutex_init (pthread_mutex_t *mp,
const pthread_mutexattr_t * mattr);

М'ютекс, вказаний ***mp***, ініціалізується значенням за замовчуванням, якщо ***mattr*** дорівнює NULL, або певними атрибутами, які вже встановлені за допомогою *pthread_mutexattr_init ().*

Блокування через м'ютекс не має повторно ініціалізуватися або віддалятися, поки інші потоки можуть його використовувати. Якщо м'ютекс ініціалізується повторно або віддаляється, додаток повинен переконатися, що в певний час цей м'ютекс не використовується;

pthread_mutex_init () повертає 0 після успішного завершення, або інше значення, якщо сталася помилка. Приклад виклику:

```
#include <pthread.h>
pthread_mutex_t mp = PTHREAD_MUTEX_INITIALIZER;
pthread_mutexattr_t mattr;
int ret;
/* Ініціалізація м’ютекса значенням за замовчуванням */
ret = pthread_mutex_init (&mp, NULL);
```

Коли м’ютекс ініціалізується, він перебуває у відкритому стані. Статично визначені м’ютекси можуть ініціалізуватися безпосередньо значеннями за замовчуванням за допомогою макросу PTHREAD_MUTEX_ INITIALIZER. Приклад ініціалізації:

```
/* Ініціалізація атрибутів м’ютекса за замовчуванням */
ret = pthread_mutexattr_init (&mattr);
/* Зміна значень mattr за допомогою функцій */
ret = pthread_mutexattr_*();
/* Ініціалізація м’ютекса довільними значеннями */
ret = pthread_mutex_init (&mp, &mattr);
```

Хоча можна створювати м’ютекси динамічно, у більшості випадків їх використання є статичним:

```
/* визначимо і запустимо м’ютекс з іменем 'mutex' */
pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
```

Цей фрагмент коду визначає та ініціалізує м’ютекс під назвою *mutex*. Це все, що треба зробити, щоб почати його використовувати.

Замикання м’ютексів

Замикання (рос. – запирание), яке називають також заволодінням, Р-потокового мьютекса забезпечується за допомогою функції *pthread_mutex_lock()*:

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

Успішний виклик *pthread_mutex_lock()* заблокує потік виклику, поки м’ютекс, зазначений як mutex, не стане доступним. Після цього потік виклику активується, і ця функція поверне 0. Якщо м’ютекс доступний

у момент виклику, функція поверне значення негайно. У разі помилки функція повертає одне з таких ненульових кодів помилки:

EDEADLK – потік виклику вже володіє м'ютексом, яким запитується; цей код помилки не обов'язково повернеться за замовчуванням; спроба заволодіти вже наявним м'ютексом може призвести до взаємного блокування.

EINVAL – значення *mutex* неприпустиме.

Потоки виклику зазвичай не перевіряють величину, яка повертається, оскільки добре написаний код не повинен генерувати будь-яких помилок під час виконання. Ось приклад використання:

```
pthread_mutex_lock (&mutex);
```

Відмикання м'ютексів

Протилежністю замикання є *відмикання*, або *вивільнення*, м'ютексів:

```
#include <pthread.h>
int pthread_mutex_unlock(pthread_mutex_t  *mutex);
```

Успішний виклик *pthread_mutex_unlock()* вивільняє м'ютекс, зазначений як *mutex*, і повертає 0. Виклик не блокується; м'ютекс звільняється негайно.

У разі помилки функція повертає ненульовий код помилки, що включає:

EINVAL – значення *mutex* неприпустиме.

EPERM – процес виклику не володіє м'ютексом, який зазначений як *mutex*; цей код помилки не гарантується; спроба звільнити м'ютекс, яким користувач не володіє, є помилкою.

Як і в разі замикання, користувачі зазвичай не перевіряють величину, яка повертається:

pthread_mutex_unlock(&mutex);

Приклад 4.6 (використання м'ютексів). Розглянемо фрагмент коду, в якому показано використання м'ютексів для забезпечення синхронізації.

Уявний банк перебуває в загрозливих умовах гонки, що може мати небажані наслідки. Це можна виправити з використанням Р-потокових м'ютексів:

```
static pthread_mutex_t the_mutex=PTHREAD_MUTEX_INITIALIZER;
int withdraw(struct account * ccount, int amount)
{
    pthread_mutex_lock(&the_mutex);
    const int balance=account->balance;
    if (balance <amount) {
        pthread_mutex_unlock(&the_mutex);
        return -1;
    }
    account->balance=balance-amount;
    pthread_mutex_unlock(&the_mutex);
    disburse_money (amount);
    return 0;
}
```

У цьому прикладі використано *pthread_mutex_lock()* для заволодіння м'ютексом, а потім *pthread_mutex_unlock()* для остаточного звільнення його. Це дозволяє уникнути умов гонки, але створює для банку нову проблему: у кожний момент часу тільки один клієнт може зняти гроші! Оце і є найвужче місце, а для «занадто-великих-щоб-збанкрутувати» банків це справжній провал.

Тому в більшості випадків, використовуючи замки, намагаються уникати *глобальних блокувань* – замість цього пов'язують замки з конкретними структурами даних. Це називають *деталізованим блокуванням*. Такий підхід може ускладнити блокувальні семантики, зокрема, під час уникнення взаємного блокування, але він є ключовим моментом у разі збільшення кількості ядер на сучасних машинах.

У прикл. 4.5 замість того, щоб визначати загальним блокуванням *the_mutex*, визначимо м'ютекс всередині структури *account*, виділяючи кожному рахунку власне блокування. Це спрацює, тому що дані всередині критичної області – тільки структура *account*. Замикаючи тільки сам

рахунок у момент списання, банку дається можливість паралельно виконувати операції інших клієнтів:

```
int withdraw(struct  account  *account,  int  amount)
{
        pthread_mutex_lock(&account->mutex);
        const int balance=account->balance;
        if (balance <amount) {
                pthread_mutex_unlock(&account->mutex);
                return -1;
        }
        account-> balance = balance - amount;
        pthread_mutex_unlock (&account-> mutex);
        disburse_money (amount);
        return 0;
}
```

КОНТРОЛЬНІ ЗАПИТАННЯ

1. Що таке процес?
2. Із чого складається контекст процесу?
3. Який механізм породження процесу?
4. Які вам відомі способи завершення процесу?
5. Що таке стиль BSD?
6. Які вам відомі категорії процесів?
7. Що таке ієрархія процесів?
8. Яким чином запускається новий процес?
9. Які вам відомі способи завершення процесу?
10. У чому відмінність між системними викликами *wait()*, *waitpid()* та *waitid()*?
11. Як здійснюється запуск нового процесу та очікування його завершення?
12. Що таке зомбі?
13. Які типи процесів вам відомі?
14. Що таке канал? Які вам відомі канали?

15. Яке призначення сигналу?

16. Що таке іменовані канали?

17. Що таке семафор?

18. Що таке сокети?

19. Що таке таблиця процесів? Які відомості вона відображає?

20. Яка команда відображає дерево процесів?

21. Яка команда відображає стан процесу у реальному часі?

22. Що таке пріоритет процесу?

23. Як змінити пріоритет процесу?

24. Що таке сигнал та як здійснюється керування сигналами?

25. Що таке потоковість?

26. У чому полягає відмінність між бінарним модулем, процесом та потоком?

27. Які вам відомі потокові моделі?

28. Що таке умовні гонки?

29. Яке призначення м’ютекса?

30. Яким чином реалізована потоковість у Linux?

31. Як створюється потік?

32. Що таке ідентифікатор потоку?

33. Що таке приєднання та від’єднання потоків?

34. Для чого використовують м’ютекс?

РОЗДІЛ 5

КЕРУВАННЯ ПАМ'ЯТТЮ

Пам'ять (RAM – оперативна пам'ять) – це один з найпростіших і водночас фундаментальних ресурсів, доступних для використання процесами. Розглянемо, як керувати пам'яттю – як її виділяти, маніпулювати нею, а потім вивільняти.

Термін «виділяти» зазвичай позначає отримання пам'яті. В сучасних системах основні складності роботи з пам'яттю пов'язані не з тим, щоб задовольнити потреби всіх, кому потрібна пам'ять, маючи при цьому невеликі обсяги пам'яті, а з тим, щоб правильно використовувати виділену частину пам'яті й відслідковувати процес такого використання.

5.1. Адресний простір процесу

Оперативна пам'ять (ОП, або **пам'ять з довільним доступом, або RAM-random access memory**) організована як одновимірний масив елементів пам'яті розміром в 1 байт. Кожному байту відповідає своя унікальна адреса (номер), яку називають *фізичною адресою*.

Linux, як і всі сучасні ОС, віртуалізують фізичну пам'ять, яка є у наявності. ***Процеси не звертаються безпосередньо до фізичної пам'яті*** – замість цього ядро пов'язує кожний процес із власним унікальним *віртуальним адресним простором* цього процесу.

Цей адресний простір є *лінійним*. Його адреси починаються з нуля, неперервно збільшуючись аж до заданого максимального значення. Крім того, адресний простір є *плоским*, він існує в єдиній області, яка є доступною і не вимагає сегментування.

Сторінки та їх підкачка

Пам’ять складається з байтів, байти складаються у слова, а слова – у сторінки. У керуванні пам’яттю найбільш важливою з цих концепцій є саме сторінка. *Сторінка* – це найменша адресована сутність у пам’яті, якою може керувати блок керування пам’яттю (*MMU – memory management unit*), тому можна сказати, що сторінки «нарізаються» з адресного простору процесу. *Розмір сторінки* залежить від машинної архітектури, яка застосована. Найбільш поширеними розмірами сторінки є 4 Кб (у 32-бітних системах) і 8 Кб (у 64-бітних системах) [21].

У 32-бітному адресному просторі міститься близько мільйона сторінок розміром по 4 Кб кожна. У 64-бітному адресному просторі на кілька порядків більше сторінок по 8 Кб. Процес не обов’язково буде звертатися до всіх цих сторінок; конкретна сторінка може нічому не відповідати. Сторінки можуть бути *валідними* або *невалідними*. Валідна сторінка асоційована з наявною сторінкою даних, яка розміщується або в ОП / пам’яті з довільним доступом (RAM – Random Access Memory), або на вторинному носії (наприклад, у розділі підкачка або у файлі на диску). Невалідна сторінка ні з чим не асоціюється і являє собою невикористаний, невиділений фрагмент адресного простору. В разі звернення до невалідної сторінці відбувається порушення сегментації.

Якщо валідна сторінка асоційована з даними, розміщеними на додатковому носії, то процес не може отримати доступу до цієї сторінки, поки інформація не буде перенесена у фізичну пам’ять. Коли процес намагається звернутися до такої сторінки, блок керування пам’яттю генерує *помилку сторінки*. Потім у роботу втручається ядро, прозоро *підкачуючи* дані із вторинного носія у фізичну пам’ять. Обсяг віртуальної пам’яті зазвичай значно перевищує обсяг фізичної пам’яті, тому ядро може вивантажити дані з пам’яті, звільнивши таким чином простір для підкачка. *Вивантаження* – це процес перенесення даних з фізичної пам’яті на вторинний носій. Щоб мінімізувати кількість наступних операцій

підкачки, ядро намагається вивантажити з пам’яті дані, які з найменшою вірогідністю будуть використовуватися у найближчому майбутньому.

Спільне використання та копіювання при запису

Множинні сторінки віртуальної пам’яті можуть бути асоційовані з єдиною фізичною сторінкою, навіть якщо вони належать до різних процесів, кожний з яких має власний адресний простір. Таким чином, різні віртуальні адресні простори можуть *спільно використовувати* (*розділяти*) дані з фізичної пам’яті. Наприклад, у будь-який момент цілком ймовірно, що багато процесів системи спільно використовують стандартну бібліотеку С. Під час використання розділюваної пам’яті кожний із цих процесів може відображати цю бібліотеку на свій віртуальний адресний простір, але у фізичній пам’яті при цьому має бути наявна лише одна копія бібліотеки. Більш наочний приклад: два процеси можуть одночасно відображати у пам’ять велику базу даних. У той час як потрібна база даних буде присутня у віртуальному адресному просторі кожного з цих процесів, в ОП буде міститися всього одна копія цієї бази даних.

Розділювані дані можуть бути доступні тільки для читання, запису або надаватися і для читання, і для запису. Коли процес записує інформацію на спільно використовувану сторінку, яка допускає такий запис, можуть виникнути дві ситуації.

В одному випадку ядро дозволяє запис, після чого всі процеси, що використовують цю сторінку, можуть бачити результат запису. Зазвичай якщо дозволити множинним процесам зчитувати одну і ту саму сторінку або записувати на неї інформацію, потрібно забезпечити певний рівень координації та синхронізації між цими процесами, але на рівні ядра запис «просто працює» і всі процеси, що розділяють дані, негайно бачать всі зміни, що відбуваються.

У другому випадку блок керування пам’яттю може перехопити операцію запису і видати виняток (рос. исключение). У відповідь ядро

прозоро створює нову копію сторінки для записувального процесу і дозволяє процесу продовжувати запис інформації вже на новій сторінці. Такий підхід називають *копіюванням під час запису* (*COW copy only write* – виклик *fork()* використовує копіювання під час запису для дублювання і спільного використання батьківського адресного простору з нащадком). Фактично процеси отримують доступ до розділюваних даних для читання, завдяки чому економиться місце. Однак якщо процес намагається записати інформацію на розділювану сторінку, то відразу отримує унікальну копію цієї сторінки. Таким чином, ядро завжди може працювати з урахуванням наявності власної копії сторінки у кожного записувального процесу. *Копіювання під час запису відбувається посторінково*, тому така техніка фактично дозволяє розділяти величезний файл між багатьма процесами й окремі процеси будуть отримувати унікальні фізичні копії тільки сторінок, на які вони самі записують інформацію.

Області пам’яті

Ядро розподіляє сторінки по блоках, які мають набір тих чи інших загальних властивостей – наприклад, прав доступу. Ці блоки називають *відображеннями, сегментами або областями пам’яті.* Деякі області пам’яті наявні у будь-якому процесі.

• *Текстовий сегмент* містить програмний код процесу, рядкові літерали, значення констант змінних та інші дані, призначені тільки для читання. У Linux цей сегмент позначається доступним тільки для читання і відображається прямо з об’єктного файлу (це може бути виконуваний файл програми або бібліотека).

• *Стек* містить стек виконання процесу. Стек може динамічно розширюватись або стискатися зі збільшенням або зменшенням глибини стеку. У стеку виконання містяться локальні змінні та дані функцій, які повертаються. У багатопотоковому процесі кожному потоку відповідає власний стек.

• *Сегмент даних, або купа* (*Heap*), містить динамічну пам’ять процесу. Цей сегмент доступний для запису, може розширюватися або стискатися. Виклик *malloc()* може задовольняти запити пам’яті з цього сегмента.

• *Сегмент bss* (*block started by symbol* – «*блок, що починається із символу*») містить неініціалізовані глобальні змінні. У цих змінних містяться спеціальні значення (як правило, тільки нулі) відповідно до стандарту С [22].

Linux оптимізує ці змінні двома способами. По-перше, оскільки *bss*-сегмент призначений для неініціалізованих даних, компонувальник (*ld – loader*) не зберігає спеціальних значень в об’єктному файлі, тому розмір виконуваного файлу зменшується. По-друге, коли цей сегмент завантажується у пам’ять, ядро просто відображає його за принципом копіювання під час запису на сторінку нулів, фактично встановлюючи змінні у значення, які задані для них за замовчуванням.

Схематично адресний простір для виконання процесу складається (рис. 5.1):

– з машинних інструкцій, які повинен виконувати центральний процесор;

– даних, з якими будуть працювати ці машинні інструкції.

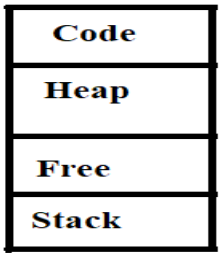


Рис. 5.1. Віртуальний адресний простір

Code – сегмент коду, область пам’яті, в якій зберігаються машинні інструкції скомпільованої програми. Ця частина пам’яті також може бути поділена на три частини (текст, дані та *bss*).

Heap – купа, область пам’яті, в якій програма може виконувати будь-що. Розмір цієї пам’яті може змінюватися. Програміст може скористатися

частиною пам’яті купи (функція *malloc()*), тоді область пам’яті буде збільшуватися. Функція *free()* очищує цю частину, і купа буде зменшуватися.

Free – вільний адресний простір.

Stack – стек, область пам’яті, в якій програма зберігає інформацію про функції, що викликаються, їх аргументи, локальні змінні у функції.

У більшості адресних просторів міститься кілька відображених файлів, до яких належать, наприклад, сам виконуваний файл програми, бібліотека *C* та інші спільні бібліотеки, а також файли з даними.

Приклад 5.1. Розглянемо файл */proc/self/maps* (файл *maps* містить карту всіх областей пам’яті, виділених для бінарного файлу, а також перелік усіх динамічних бібліотек.) або виведення програми *pmap*, щоб отримати уявлення про файли, які відображаються у процесі. Нагадаємо, що інформація про процеси збирається у файловій системі *procfs*. Кожному процесу відповідає своя директорія:

```
$ ls /proc/self/
attr             cwd      loginuid    numa_maps      schedstat  task
autogroup        environ  map_files   oom_adj        sessionid  timers
auxv             exe      maps        oom_score      setgroups  uid_map
cgroup           fd       mem         oom_score_adj  smaps      wchan
clear_refs       fdinfo   mountinfo   pagemap        stack
cmdline          gid_map  mounts      personality    stat
comm             io       mountstats  projid_map     statm
coredump_filter  latency  net         root           status
cpuset           limits   ns          sched          syscall
```

```
$ time cat /proc/*/maps > /dev/null
real 0m0.061
user 0m0.002s
sys 0m0.059s
time – це час виконання команди (перегляду вмісту файлу maps).
```

Далі розглянемо інтерфейси, що надаються Linux для отримання і повернення пам’яті, створення та знищення нових відображень, а також забезпечення всіх проміжних етапів роботи.

5.2. Інтерфейси для роботи з пам’яттю

5.2.1. Виділення динамічної пам’яті

Пам’ять також надається у формі автоматичних і статичних змінних, але основа будь-якої системи керування пам’яттю – це виділення, використання і повернення *динамічної пам’яті*. **Динамічна пам’ять повертається під час виконання**, а не під час компіляції, причому в обсязі, який може бути невідомий аж до моменту виділення [21, 22]. Розробник вдається до використання динамічної пам’яті, коли обсяг необхідної пам’яті або тривалість її використання можуть варіюватися, причому точні значення обсягу і тривалості стають відомі тільки після запуску програми. Наприклад, розробник хоче зберегти у пам’яті вміст файлу або дані введення користувача, які отримані через клавіатуру. Розмір файлу точно невідомий, а користувач може ввести досить багато інформації, тому розмір буфера буде варіюватися і треба буде його динамічно збільшувати у міру зчитування все більшої кількості даних.

У мові С немає змінної, основою для якої є динамічна пам’ять, – механізму отримання структури *struct pirate_ship*, яка є в динамічної пам’яті. Замість цього С надає можливість виділити динамічну пам’ять, достатню для утримання структури *pirate_ship*. Після цього програміст буде взаємодіяти з пам’яттю за допомогою вказівника, у цьому випадку *struct pirate_ship**.

***Класичний інтерфейс С для отримання динамічної пам’яті називають malloc()*:**

```
#include <stdlib.h>   /* прототип для цієї функції міститься в "stdlib.h"*/
void *malloc(size_t size); /*резервування блока пам’яті розміром */
```

Якщо виклик успішний, *malloc()* виділяє *size* байтів пам’яті та повертає вказівник у початкову точку щойно виділеної області; або до нульового вказівника, якщо блок не зміг би бути розміщеним. У разі

помилки *malloc()* повертає NULL, а *errno* встановлюється значення ENOMEM.

Приклад 5.2. Використання *malloc()*, в якому виділяено фіксовану кількість байтів:

```
char  *p;   /* змінна вказівник, в яку записується результат функції malloc */
/ * виділіть мені 2 Кб! * /
p=malloc(2048);
if (!p)
      perror("malloc");
```

або де виділено структуру:

```
struct  treasure_map  *map;
/*  виділяємо  достатню  кількість  пам'яті  для  утримання  вмісту структури
* treasure_map і вказуємо на неї за допомогою 'map' * /
map=malloc(sizeof(struct  treasure_map));  /*вказівник на структуру*/
if (!map)
      perror("malloc");
```

Мова *C* автоматично підвищує вказівники на *void* для будь-яких типів призначення. У прикл. 5.2 не використано приведення типу значення, що повертається *malloc()*, до типу *lvalue*, який використовують при призначеннях. (*lvalue (locator value*) являє собою об'єкт, що займає і дозволяє ідентифікувати місце в пам'яті (наприклад, має адресу): int rs; rs=5.

Але в мові C++ не виконується автоматичного підвищення вказівника на *void*, тому програмісти, які працюють із C++, мають приводити тип значення *malloc()* таким чином:

```
char  *name;          /*об'являється вказівник char на змінну name*/
/* виділяємо 512 байтів  char */
name=(char  *) malloc(512);  /*вказівник на масив*/
if (!name)
      perror("malloc");
```

Функція *malloc()* може повертати NULL (нульовий вказівник), тому розробник просто зобов'язаний *завжди* перевіряти цю умову й обробляти

помилки за їх наявності. Багато програм визначають і використовують функцію *malloc()*, яка виводить повідомлення про помилку і програму, що завершується, якщо *malloc()* поверне NULL.

За традицією розробники використовують іншу поширену функцію ***xmalloc()***, яка є нестандартною функцією і перевіряє: якщо вказівник нульовий в *malloc*(), то повертає помилку, якщо ні – повертає значення вказівника, відмінне від нуля. Якщо не вдалося виділити пам'ять, програма припиняє працювати і виводить повідомлення про помилку в *stderr*:

```
/* схожа на malloc(), але при помилці завершується */
void  *xmalloc (size_t size)  /*вказівник на функцію*/
{
      void *p;/* порожній вказівник, використовується, коли тип змінної невідомий*/
      p=malloc(size);
      if (!p) {
            perror ( "xmalloc");
            exit (EXIT_FAILURE);
      }
      return p;
}
```

Вважається, що функція *xmalloc()* – для лінивих, бажано використовувати функцію *malloc()*.

5.2.2. Виділення масивів

Динамічне виділення пам'яті також може бути досить складним, якщо зазначений розмір *size* сам по собі є динамічним. Прикладом такої роботи є динамічне виділення масивів, де розмір елемента масиву може бути фіксованим, а кількість виділених елементів змінне. Для спрощення цього сценарію в бібліотеці С надається функція ***calloc()***, яка резервує пам'ять та обнуляє її [10]:

```
#include <stdlib.h>
void  *calloc(size_t nr,  size_t size);
```

Після успішного виклику *calloc()* повертається вказівник на блок пам’яті, що підходить для утримання масиву з *nr* елементів, кожний з яких має розмір *size* байтів. Тобто обсяг пам’яті, щодо якого надійшов запит від цих двох викликів, є ідентичним (кожний виклик може повернути більше пам’яті, ніж спочатку запитував, але не менше):

```
int *x,  *y;  /*вказівники на цілі числа*/
x=malloc(50 *sizeof (int));  /* вказівник*/
if (!x) {
        perror("malloc");
        return -1;
}
y=calloc(50,  sizeof (int));  /*вказівник*/
if (!y) {
        perror("calloc");
        return -1;
}
```

Однак їх поведінка не є ідентичною. На відміну від *malloc(),* яка не дає ніяких гарантій щодо вмісту виділеної пам’яті, *calloc()* заповнює нулями усі байти у фрагменті пам’яті, який повертається. Таким чином, кожний із 50 елементів, які містяться в масиві цілих чисел ***y***, має значення 0, а вміст елементів в ***x*** залишається невизначеним. Якщо програма не збирається відразу ж встановити всі 50 значень, то програміст повинен сам гарантувати, що елементи в масиві не будуть заповнені сміттям. Звернімо увагу: двійковий нуль може не дорівнює нулю з плаваючою точкою!

Часто користувачі намагаються «обнулити» динамічну пам’ять, навіть якщо не працюють із масивами. Однак якщо наказзати *calloc()* виконати заповнення нулями, то робота піде швидше, тому що ядро може надати пам’ять, яка вже заповнена нулями. У разі помилки *calloc(),* як і *malloc(),* повертає NULL і встановлює *errno* значення ENOMEM.

Органи стандартизації так і не визначили окрему від *calloc()* функцію, яка «виділяє і заповнює нулями». Розробник може визначити для цього власний інтерфейс:

```
/* діє так само, як і malloc(), але пам'ять заповнюється нулями */
void  *malloc(size_t size)
{
        return calloc(1, size);
}
```

Можна скомбінувати *malloc()* разом із *xmalloc()*:

```
/* схожа на malloc(), але заповнює пам'ять нулями і завершується при помилці */
void  *xmalloc(size_t size)
{
        void  *p;
        p=calloc(1, size);
        if (!p) {
                perror("xmalloc");
                exit (EXIT_FAILURE);
        }
        return p;
}
```

5.2.3. Зміна розміру виділених областей

Мова С надає інтерфейс для зміни розміру (збільшення або зменшення) вже наявних виділених областей:

```
#include <stdlib.h>
void *realloc(void *ptr,  size_t  size);
```

Після успішного виклику *realloc()* ця функція змінює розмір області пам'яті, на яку спрямований вказівник *ptr*, що задає для неї нове значення і дорівнює *size* байтів. Вона повертає вказівник на заново виміряну область пам'яті, причому цей вказівник може дорівнювати *ptr*, а також мати інше значення. У разі збільшення області пам'яті *realloc()* може виявитися, що системний виклик *realloc()* не в змозі збільшити вихідний фрагмент до розміру, на який надходить запит, на тому ж місці, де цей фрагмент тепер міститься. У такому випадку функція може виділити нову область пам'яті розміром *size* байтів, скопіювати стару область у нову, а стару після цього вивільнити. За будь-якої подібної операції вміст області пам'яті зберігається або повністю, або в розмірі, який дорівнюватиме обсягу нової

виділеної області. Оскільки операції *realloc()* пов'язані з копіюванням, зі збільшенням області пам'яті вони можуть бути досить витратними.

Якщо розмір *size* дорівнює нулю, то ефект є аналогічним виклику *free()* щодо *ptr*. Якщо *ptr* дорівнює NULL, результат операції аналогічний «свіжому» використанню *malloc()*. Якщо вказівник *ptr* не дорівнює нулю, то він обов'язково має бути повернутий у попередній виклик, який направлений до *malloc()*, *calloc()* або *realloc()*. У разі помилки *realloc()* повертає NULL і присвоює *errno* значення ENOMEM. Стан області пам'яті, на яку вказує *ptr*, залишається незмінним.

Приклад 5.3. Розглянемо приклад зі зменшенням області пам'яті. Спочатку скористаємося *calloc()* і виділимо достатньо пам'яті, щоб утримувати в ній двохелементний масив структур *map*:

```
struct map *p;
/* виділяємо пам'ять для 2 структур map */
p=calloc(2, sizeof(struct map));
if (!p) {
perror("calloc");
return -1;
}
/* використовуємо p[0] і p[1] ... */
```

Тепер припустимо, що для однієї структури *map* пам'ять уже знайдено, для другої структури (*map*) її більше не потрібно, тому змінимо зайняту область пам'яті й віддамо її половину, відведену під структуру *map*, назад у розпорядження системи. Така операція не завжди доцільна, але вона корисна, якщо структура *map* дуже велика, а ту, що залишилася *map*, планується зберігати ще досить довго:

```
struct  map  *r;
/* пам'ять для зберігання тільки однієї структури map */
r=realloc(p, sizeof(struct map));
if (!r) {
/* значення 'p' як і раніше залишається допустимим! */
perror("realloc");
return -1;
}
```

```
/* використовуємо 'r' ... * /
free(r);  /* вивільнення пам’яті*/
```

У прикл. 5.3 p[0] зберігається після виклику *realloc()*. Які б дані там не містилися, вони залишаються доступними. Якщо виклик повернеться з помилкою, ***p*** залишається недоторканим, а отже, чинним. Можна продовжувати ним користуватися, а після закінчення роботи – вивільнити. Навпаки, якщо виклик завершиться успішно, ***p*** ігноруємо, а замість нього використовуємо ***r***. Тепер відповідаємо за вивільнення ***r*** після закінчення роботи.

5.2.4. Звільнення динамічної пам’яті

На відміну від областей, які автоматично виділяються та збираються системою без участі користувача, як тільки розпадається стек, динамічно виділені області залишаються невід’ємними частинами адресного простору процесу, доки не будуть вивільнені вручну, тому програміст відповідає за повернення пам’яті, яка динамічно виділяється, у систему. Зрозуміло, що всі виділені області пам’яті (як статичні, так і динамічні) автоматично вивільняються, як тільки завершується весь процес. **Пам’ять**, виділена за допомогою *malloc(), calloc()* або *realloc(),* **має бути повернута системі, якщо ця пам’ять більше не використовується**. Це робиться за допомогою ***free()***:

```
#include <stdlib.h>
void free(void  *ptr);
```

Виклик *free()* звільняє пам’ять, на яку вказує *ptr*. Параметр *ptr* має заздалегідь бути повернутий *malloc(), calloc()* або *realloc().* Це означає, що за допомогою *free()* не можна вивільняти довільні фрагменти пам’яті – наприклад, половину області пам’яті, – передавши покажчик на середину виділеного блоку. Така дія призведе до появи невизначеної пам’яті, що проявиться у вигляді збою.

Вказівник *ptr* може дорівнювати NULL, у разі чого *free()* повертає значення, тому поширена практика перевірки *ptr* на NULL перед викликом *free()* не має сенсу.

Приклад 5.4. Розглянемо такий приклад:

```
void  print_chars(int  n, char  c)
{
      int i;
      for (i = 0; i <n; i ++) {
          char  *s;
          int j;
          /* виділяємо і заповнюємо нулями масив елементів i+2, який
*/
          /* складається із символів. 'sizeof (char)' завжди дорівнює 1.* /
          s= calloc(i + 2, 1);
          if (!s) {
                perror("calloc");
                break;
          }
          for (j=0; j <i + 1; j ++)
                    s[j]=c;
          printf("%s\n", s);
          /* все виконано, можна повернути пам’ять. */
          free(s);
      }
}
```

У цьому прикладі виділено ***n*** масивів із символів *char*, які містять послідовно зростаючу кількість елементів, починаючи від двох (2 байтів) до n + 1 елементів (n + 1 байт). Потім, обробляючи кожний масив, цикл записує символ ***c*** у кожний байт, крім останнього (залишаючи 0, який вже міститься в останньому байті). Масив виводиться на екран як рядок, після чого пам’ять, яка динамічно виділена, вивільняється. При виклику *print_chars()* із ***n***, який дорівнює 5, і ***c***, встановленому в *X*, отримуємо таке:

```
X
XX
XXX
XXXX
XXXXX
```

Зрозуміло, є і більш ефективні способи реалізації цієї функції. Однак суть полягає у тому, що можна динамічно виділяти і вивільняти пам'ять, навіть якщо розмір і кількість виділених областей стають відомі тільки під час виконання.

Розглянемо, якими будуть наслідки, якщо у прикл. 5.4 не викликати *free()*. Програма так і не поверне пам'ять у систему. Гірше того, вона втратить своє єдине посилання на цю пам'ять – вказівник ***s***, у результаті чого і сама програма більше не зможе звернутися до пам'яті. Такий тип програмної помилки називають *витоком* (рос. *утечкой*) *пам'яті.* Витоки та інші подібні помилки, пов'язані з динамічною пам'яттю, є одними з найбільш поширених і, на жаль, належать до найбільш серйозних проблем при програмуванні мовою *C*. У мові *C виключно програміст є відповідальним за керування пам'яттю*, тому він повинен дуже уважно стежити за операціями виділення пам'яті.

Ще одна поширена проблема при програмуванні мовою *C* – *використання звільненої пам'яті.* Така вразливість виникає, якщо програма звертається до блоку пам'яті, який вже був звільнений. Щойно для блоку пам'яті викликана функція *free()*, програма в жодному разі не має більше звертатися до вмісту цього блоку. Програміст повинен проявляти особливу увагу, відстежуючи *завислі вказівники*: ненульові, які тим не менше вказують на недійсний блок пам'яті. Інструмент, який допомагає знаходити у програмі помилки, пов'язані з неправильним використанням пам'яті, називають *Valgrind.*

5.2.5. Вирівнювання даних

Вирівнювання даних – це спосіб розміщення даних у пам'яті. Про адресу A в пам'яті говорять, що вона вирівняна за *n* байтами та *n* є ступенем 2, а A є кратним *n.* Процесори, підсистеми пам'яті та інші системні компоненти висувають специфічні вимоги до вирівнювання даних. Наприклад, більшість процесорів оперують машинними словами і

можуть звернутися до пам’яті, лише якщо вона вирівняна за розміром слова. Відповідно одиниці керування пам’яттю працюють тільки з адресами, вирівняними за розміром сторінок.

Якщо змінна розміщена у пам’яті за адресою, яка кратна її розміру, то її називають *природно вирівняною*. Наприклад, 32-бітна змінна є природно вирівняною, якщо вона розміщена у пам’яті за адресою, яка кратна 4, – інакше кажучи, якщо два найнижчих біта цієї адреси рівні 0. Відповідно, тип розміром *2n* байтів повинен мати адресу, в якій *n* найменших значущих бітів мають нульові значення.

Правила вирівнювання обумовлені апаратним забезпеченням, тому різняться від системи до системи. Деякі машинні архітектури висувають дуже суворі вимоги до вирівнювання даних, в інших системах вимоги більш гнучкі. Деякі системи генерують помилки, які відловлюються – після цього ядро може вибрати потрібну дію: або завершити процес-порушник, або (що більш ймовірно) дозволити вручну виконати невирівняний доступ. Як правило, такий доступ складається з безлічі більш дрібних вирівняних операцій. У результаті знижується продуктивність, крім того, доводиться жертвувати атомарністю, але хоча б вдається не завершувати процес. Під час написання переносимого коду програміст повинен дуже уважно стежити, щоб не порушувалися вимоги, пов’язані з вирівнюванням.

Виділення вирівняною пам’яттю

У більшості випадків компілятор і бібліотека *C* прозоро обробляють проблеми, пов’язані з вирівнюванням. Стандарт POSIX рекомендує, що пам’ять, яка повертається *malloc(), calloc()* та *realloc(),* має бути правильно вирівняна і придатна для використання з будь-якими стандартними типами *C*. В Linux такі функції завжди повертають пам’ять, вирівняну по 8-байтній границі в 32-бітних системах і по 16-байтній границі в 64-бітних. Іноді програмісту потрібно, щоб динамічна пам’ять була вирівняна по ширшій

границі, наприклад по сторінці. Причини для цього можуть бути різні, але найбільш поширена причина – необхідність правильно вирівняти блоки, які використовують при безпосередньому блоковому «введенні – виведенні», або іншій взаємодії між апаратними та програмними компонентами. Для цієї мети в POSIX 1003.1d надається функція ***posix_memalign()***:

```
/* одна або інша – достатньо будь-якої */
#define  _XOPEN_SOURCE 600
#define  _GNU_SOURCE
#include <stdlib.h>
int  posix_memalign(void  **memptr, size_t  alignment,  size_t  size);
```

Виклик *posix_memalign()* виділяє *size* байтів динамічної пам'яті, забезпечуючи вирівнювання цієї пам'яті за адресою, яка кратна *alignment*. Параметр *alignment* має бути ступенем 2, а також бути кратним розміром вказівника *void – sizeof(void*).* Адреса виділеної пам'яті поміщається в **memptr*, і виклик повертає 0. У разі помилки пам'ять не виділяється, *memptr* залишається невизначеним, а виклик повертає один із таких кодів помилок:

EINVAL – параметр *alignment* не є ступенем 2 або не кратний розміру вказівника *void*;

ENOMEM – недостатньо пам'яті, щоб виділити обсяг, який запитаний.

У цьому випадку *errno* не встановлюється – функція повертає ці помилки напряму. Пам'ять, отримана за допомогою *posix_memalign()*, вивільняється за допомогою *free()*. Приклад використання цієї функції:

```
char  *buf;
int  ret;
/* виділяємо 1 Кб, вирівняний по 256-байтній границі */
ret=posix_memalign(&buf, 256, 1024);
if (ret) {
      fprintf(stderr, "posix_memalign: %s\n", strerror (ret));
      return -1;
}
```

```
/* використовуємо 'buf' ... */
free(buf);
```

5.2.6. Керування сегментом даних

Історично в системах UNIX надавалися інтерфейси для безпосереднього керування сегментом даних. Між тим у більшості програм ця можливість майже не знаходила застосування, оскільки *malloc()* та інші способи виділення пам'яті є значно простішими у використанні, а також потужнішими. Інтерфейси для реалізації власного механізму виділення пам'яті на основі роботи з купою:

```
#include <unistd.h>
int  brk(void *end);
void *sbrk(intptr_t increment);
```

Назви цих функцій походять зі старих UNIX-систем, в яких купа і стек ще перебували в одному сегменті пам'яті. Виділення динамічної пам'яті в купі виконувалося вгору, починаючи з нижньої частини сегмента; стек зростав вниз з верхньої частини сегмента, у напрямку до купи. Межу, яка розділяла області стека і купи, називали *зупинкою* або *точкою зупинки*. В сучасних системах, в яких сегмент даних перебуває у власному відображенні в пам'яті, як і раніше, називають кінцеву адресу відображення точкою зупинки.

Виклик ***brk()*** *встановлює точку зупинки* (*кінець сегмента даних*) за адресою, яка вказана в *end*. У разі успіху функція повертає 0, у разі помилки вона повертає -1 і встановлює *errno* значення ENOMEM.

Виклик ***sbrk()*** збільшує кінець сегмента даних на *increment* байтів (приріст /дельта), причому дельта може бути як позитивною, так і негативною. Виклик *sbrk()* повертає виправлену точку зупинки, відповідно, приріст, який дорівнює 0, дає актуальну точку зупинки:

```
printf("Потокова точка зупинки – %p\n", sbrk (0));
```

Ні у стандарті С, ні у POSIX навмисно не визначається жодна із цих функцій. Однак майже у всіх UNIX-системах підтримується щонайменше

одна з них, а то і дві. *Програми, які переносяться, повинні працювати тільки зі стандартизованими інтерфейсами*.

5.3. Анонімні відображення у пам’яті

Під час виділення пам’яті в *glibc* використовується комбінація сегмента даних і відображень у пам’яті. Класичний метод реалізації *malloc()* – поділ сегмента даних на послідовність фрагментів, розмір яких є ступенем 2; після цього запити на виділення пам’яті задовольняються після повернення фрагмента, який максимально відповідає запрошенням розміром. Вивільнити пам’ять не важко: досить просто помітити фрагмент як вільний. Якщо суміжні фрагменти вільні, їх можна об’єднати в один більший фрагмент. Якщо верхня частина купи зовсім вільна, то система може знизити точку зупинки за допомогою *brk()*, зменшивши таким чином купу й повернувши пам’ять ядра.

Цей алгоритм називають *схемою дружнього виділення пам’яті*. До його переваг належать висока швидкість роботи і простота, а основним недоліком є виникнення двох типів фрагментації. *Внутрішня фрагментація* відбувається, коли для задоволення запиту виділяється більше пам’яті, ніж було запитано. У результаті доступна пам’ять витрачається неефективно. *Зовнішня фрагментація* виникає, коли в наявності є достатньо вільного місця для задоволення запиту, але ця вільна пам’ять розбита на два і більше несуміжних фрагментів. У такому випадку пам’ять також витрачається неефективно (оскільки може використовуватися порівняно великий блок, який не дуже підходить у цьому випадку). Крім того, зростає кількість помилок при виділенні пам’яті (якщо немає альтернативного блоку). Більше того, за такої схеми одна операція виділення пам’яті може «зачіпати» іншу, що не дозволяє бібліотеці *C* повертати вивільнену пам’ять в ядро. Припустимо, було виділено два блоки пам’яті: A та B. Блок A впритул прилягає до точки

зупинки, а блок B міститься безпосередньо під A. Навіть якщо програма звільнить блок B, бібліотека *C* не зможе відкоригувати точку зупинки, поки таким же чином не буде звільнений фрагмент A. Отже, довгоживуча виділена область може утримувати в пам'яті й інші, вже порожні області. Така ситуація не завжди є проблемою, оскільки бібліотеки *C* не зобов'язані повертати пам'ять у систему. Як правило, купа не зменшується після кожного вивільнення. Замість цього реалізація *malloc()* притримує пам'ять до наступного виділення. Тільки якщо розмір купи значно перевищує обсяг виділеної пам'яті, *malloc()* справді звужує сегмент даних, але виділення великого фрагмента може перешкоджати такому звуженню.

Таким чином, у разі великих виділень пам'яті *glibc* не використовує купу. Замість цього вона створює *анонімне відображення в пам'яті*, за допомогою якого задовольняє запит на виділення. Анонімні відображення в пам'яті подібні файловим відображенням, не врахуючи того, що в їх основі не є якийсь файл – тому їх і називають анонімними. Насправді анонімне відображення в пам'яті – це просто великий блок пам'яті, заповнений нулями і готовий до використання. Можна вважати його «новоспеченою» купою, яка буде використовуватися в межах одного виділення в пам'яті. Такі відображення розміщуються поза купою, тому не призводять до додаткової фрагментації сегмента даних. Під час виділення пам'яті методом анонімних відображень є такі переваги:

• Немає проблем, пов'язаних із фрагментацією. Коли програма більше не потребує анонімного відображення у пам'яті, воно видаляється і пам'ять негайно повертається в систему.

• Можна змінювати розмір анонімних відображень у пам'яті, коригувати права доступу до них, вони можуть отримувати сповіщення, як і звичайні відображення.

• Кожне виділення існує в окремому відображенні у пам'яті. Немає необхідності керувати глобальною купою. При використанні анонімних

відображень пам’яті замість роботи з купою можна також стикнутися з двома недоліками.

• Розмір кожного відображення в пам’яті поділяється без залишку на розмір сторінки, яка застосовується в системі, тому якщо виділений простір не є цілочисельним кратним розміру сторінки, то буде утворюватися невикористовуваний «втрачений» простір. Він являє тим більшу проблему, чим менше розмір окремих виділень, де такі «хвости» виходять відносно великими порівняно з корисними областями.

• Створення нового відображення в пам’яті пов’язане з більшою кількістю витрат (рос. издержек), ніж задоволення запитів на виділення з купи, оскільки купа може працювати взагалі без взаємодії з ядром. Чим менше розмір виділених областей, тим істотніші витрати.

З огляду на всі «за» і «проти», функція *malloc()* з *glibc* використовує сегмент даних для задоволення невеликих виділень, а у випадку з великими вдається до анонімних відображеннь у пам’яті. Цей поріг є нежорстким і може змінюватися від одного релізу *glibc* до іншого. У наш час він дорівнює 128 Кбайт: виділення пам’яті, що не перевищує цього обсягу, робляться з купи, а при більших обсягах застосовуються анонімні відображення у пам’яті.

5.3.1. Створення анонімних відображень у пам’яті

Може бути так, щопри якомусь виділення програміст хоче примусово використовувати відображення в пам’яті, а не купу; можливо, він пише власну систему виділення пам’яті. У цих випадках потрібно вручну створювати відображення у пам’яті. Так чи інакше, в Linux це не становить труднощів. Системний виклик *mmap()* створює відображення в пам’яті, а системний виклик *munmap()* його знищує [10]:

```
#include <sys mman.h>
void *mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset);
int munmap(void *start, size_t length);
```

Створити анонімне відображення у пам’яті простіше, ніж відображення, в основі якого є файл, оскільки в першому випадку не доводиться ні відкривати файл, ні керувати ним. Основна відмінність полягає у наявності спеціального прапорця, який вказує, що відображення є анонімним.

Приклад 5.5. Розглянемо приклад:

```
void  *p;
p=mmap(NULL,            /* неважливо де */
        512 * 1024,     /* 512 Кб */
        PROT_READ | PROT_WRITE, /* читання / запис */
        MAP_ANONYMOUS | MAP_PRIVATE, /* анонімне, приватне
*/
        -1,             /* fd (ігнорується) */
        0);             /* зміщення (ігнорується) */
if (p==MAP_FAILED)
     perror("mmap");
else
     / * 'p' вказує на 512 Кб анонімної пам’яті ... * /
```

У більшості анонімних відображень параметри *mmap()* відповідають цьому прикладу, за винятком передавання довільного розміру (в байтах), що визначає програміст. Інші параметри зазвичай такі.

• Перший параметр, *start*, встановлюється у значення NULL, вказуючи, що анонімне відображення може починатися в будь-якій точці пам’яті, обраній ядром. Тут можна задати і нульове значення, якщо воно вирівняно по межах сторінок, але це обмежує переносимість. Зазвичай для програми не має значення, де саме в пам’яті міститься відображення.

• Параметр *prot* зазвичай одночасно задає біти PROT_READ та PROT_WRITE, допускаючи як зчитування відображення, так і запис в нього. Між тим виконання коду з анонімного відображення зазвичай небажано, оскільки це можливий вектор для атаки зловмисників.

• Параметр *flags* задає прапорець MAP_ANONYMOUS, який робить це відображення анонімним, а також прапорець MAP_PRIVATE, який робить відображення приватним.

• Параметри *fd і offset* ігноруються, якщо заданий прапорець MAP_ANONYMOUS. Однак у деяких старих системах передбачається, що у такому випадку *fd* повинен мати значення -1, тому таке значення доцільно ставити для забезпечення переносимості.

Пам'ять, отримана за допомогою анонімного відображення, виглядає як пам'ять, взята з купи. Одна з переваг виділення пам'яті саме з анонімних відображень полягає в тому, що можна отримати готові сторінки, вже заповнені нулями, причому без будь-яких витрат, оскільки ядро відображає анонімні сторінки додатка на сторінки в пам'яті за допомогою копіювання під час запису. Таким чином, *не треба застосовувати виклик memset() до пам'яті, яка повертається. У цьому й полягає одна з переваг використання calloc() порівняно з malloc()*, за яким слідує *memset()*: *glibc* відомо, що анонімні відображення вже заповнені нулями і що *calloc(),* який задовольняється з відображення, не потребує явного заповнення такого типу .

Системний виклик *munmap()* вивільняє анонімне відображення, повертаючи виділену пам'ять ядра:

```
int ret;
/* с 'p' все готово, віддаємо відображення в 512 Кбайт назад * /
ret=munmap (p, 512 * 1024);
if (ret)
      perror ( "munmap");
```

5.3.2. Відображення /dev/zero

В інших UNIX-системах, зокрема BSD, немає прапорця MAP_ANONYMOUS. Замість цього в них застосовується рішення, пов'язане з відображенням особливого файлу пристрою */dev/zero*. Цей файл пристрою забезпечує таку ж семантику, як і анонімна пам'ять. Відображення містить скопійовані під час запису сторінки, які заздалегідь заповнені нулями. Відповідно, з практичної точки зору воно не відрізняється від анонімної пам'яті.

В Linux завжди надавався пристрій */dev/zero*, а також можливість його відображення й отримання пам’яті, заповненої нулями. Насправді до появи прапорця MAP_ANONYMOUS програмісти Linux користувалися саме таким підходом. Для забезпечення зворотної сумісності з порівняно старими версіями Linux або переносимості на інші UNIX-системи розробник може використовувати */dev/zero* замість створення анонімного відображення. Синтаксис не відрізняється від операції відображення будь-якого іншого файлу:

```
void  *p;
int  fd;
/* відкриваємо /dev/zero для читання і запису */
fd=open("/dev/zero", O_RDWR);
if (fd <0) {
      perror("open");
      return -1;
}
/* відображаємо [0, розмір сторінки) з /dev/zero */
p = mmap (NULL,              /* неважливо де */
      getpagesize(),         /* відображаємо одну сторінку */
      PROT_READ    |    PROT_WRITE,    /*    відображаємо    для
читання/запису */
      MAP_PRIVATE,           /* приватне відображення */
      fd,                    /* відображаємо /dev/zero */
      0);                    / * без зміщення * /
if (p==MAP_FAILED) {
      perror("mmap");
      if (close(fd))
            perror( "close");
      return -1;
}
/* закриваємо /dev/zero, він нам більше не потрібен *
if (close(fd))
      perror("close");
/* 'p' вказує на одну сторінку в пам’яті, використовуємо її ... */
```

Пам’ять, яка відображається таким чином, вивільняється за допомогою *munmap()*. Цей підхід пов’язаний з додатковими витратами у вигляді зайвого системного виклику, який застосовують для відкривання

і закривання файлу пристрою. Отже, робота з анонімної пам’яттю – це більш швидке рішення.

5.4. Розширене виділення пам’яті

Багато операцій виділення пам’яті, що розглядаються у цьому розділі, обмежені й керуваються *glibc* або параметрами ядра, які програміст може змінювати. Для цього застосовують виклик ***mallopt()***:

```
#include <malloc.h>
int  mallopt(int param, int value);
```

Виклик *mallopt()* встановлює параметр *param*, пов’язаний з керуванням пам’яттю у значення *value*. У разі успіху виклик повертає нульове значення, при помилці – 0. Виклик *mallopt()* не встановлює *errno*. Цей виклик фактично завжди закінчується успішно.

У наш час Linux підтримує сім значень *param*, усі вони визначаються в <*malloc.h*>:

• M_CHECK_ACTION – значення змінної оточення MALLOC_CHECK_.

• M_MMAP_MAX – максимальна кількість відображень, які система може створити для задоволення запитів динамічної пам’яті. Коли ліміт досягнутий, сегмент даних буде використовуватися для всіх операцій виділення пам’яті, поки не буде вивільнено будь-яке з раніше створених відображень. Значення 0 повністю скасовує використання будь-яких анонімних відображень для виділення динамічної пам’яті.

• M_MMAP_THRESHOLD – поріг (вимірюваний у байтах), до якого запити на виділення пам’яті будуть задовольнятися за допомогою анонімного відображення, без участі сегмента даних. Виділення, менші цього порога, також можуть задовольнятися за допомогою анонімних відображень на розсуд системи. Значення 0 активує використання анонімних відображень у всіх випадках, фактично скасовуючи застосування сегмента даних під час виділень динамічної пам’яті.

• M_MXFAST – максимальний розмір швидкого кошика (у байтах). *Швидкі кошики (fast bins)* – це особливі фрагменти пам’яті в купі, які ніколи не об’єднуються із суміжними областями і не повертаються в систему. Таким чином, працюючи з швидкими кошиками, можна добитися дуже оперативного виділення пам’яті за рахунок збільшення фрагментації. Значення 0 деактивує будь-яке використання швидких кошиків.

• M_PERTURB – активує спотворення пам’яті (рос. искажение), що допомагає виявляти помилки в керуванні пам’яттю. Отримуючи нульове значення *value*, *glibc* встановлює усі виділені байти (окрім тих, які були запитані через *calloc()*) у значення, яке логічно доповнює найменший значущий байт у *value*. Це допомагає виявляти помилки, пов’язані з використанням до ініціалізації. Більше того, *glibc* встановлює всі вивільнені байти в найменший значущий байт із *value*. Так вдається ідентифікувати помилки, пов’язані з використанням після вивільнення.

• M_TOP_PAD – обсяг заповнення (у байтах), що використовується для коригування розміру сегмента даних. Кожний раз, коли *glibc* використовує *brk()* для збільшення сегмента даних, можна запросити більше пам’яті, ніж потрібно, розраховуючи обійтися без додаткового виклику brk(), який знадобився б для отримання нової порції пам’яті у найближчому майбутньому. Аналогічно щоразу, коли glibc зменшує розмір сегмента даних, вона може утримати додаткову пам’ять, віддавши трохи менше, ніж було потрібно. Ці додаткові байти і називають *заповненням*. Значення 0 відключає будь-яке використання заповнення.

• M_TRIM_THRESHOLD – мінімальна кількість вільної пам’яті (у байтах), яка міститься поверх сегмента даних перед тим, як *glibc* виконає виклик *sbrk()* для повернення пам’яті в ядро.

Тонке налаштування за допомогою *malloc_usable_size()* і *malloc_trim()*

Linux надає пару функцій, що забезпечують низькорівневе керування системою виділення пам’яті в *glibc*. Перша така функція дозволяє запитувати, скільки придатних для використання байтів міститься у зазначеному виділеному фрагменті пам’яті:

```
#include <malloc.h>
size_t  malloc_usable_size(void  *ptr);
```

Успішний виклик *malloc_usable_size()* повертає точний розмір фрагмента пам’яті, який виділяється і на який вказує вказівник *ptr*. Оскільки *glibc* іноді округляє області, які виділяються, щоб вони вміщалися у наявний фрагмент анонімного відображення, робочий обсяг такої області може бути більшим, ніж запитано. Зрозуміло, що виділена область не може бути меншою, ніж запитано.

Приклад 5.6. Наведемо приклад використання цієї функції:

```
size_t  len=21;
size_t  size;
char  *buf;
buf=malloc(len);
if (!buf) {
      perror("malloc");
      return -1;
}
size=malloc_usable_size(buf);
/* насправді в нашому розпорядженні є 'size' байтів з буфера 'buf' ... */
```

Друга функція *malloc_trim()* дозволяє програмі змусити *glibc* до того, щоб уся пам’ять, яку компілятор може вивільнити на цей момент, одразу ж повернулася до ядра:

```
#include <malloc.h>
int  malloc_trim(size_t  padding);
```

У разіи успішного виклику *malloc_trim()* сегмент даних максимально стає коротшим, без урахування *padding* байтів, які резервуються. У такому випадку функція повертає 1, при помилці вона повертає 0. Зазвичай *glibc*

виконує таке зменшення автоматично, відразу після того, як обсяг доступної для вивільнення пам’яті досягне M_TRIM_THRESHOLD байтів. При цьому використовується заповнення в M_TOP_PAD. Ці функції майже ніколи не знадобляться користувачу, окрім потреби в налагодженні або в навчальних цілях. Вони не переносяться і надають програмі низькорівневі деталі механізму *glibc* для виділення пам’яті.

5.5. Отримання статистичної інформації

В Linux надається функція *mallinfo()*, що видає статистичну інформацію щодо виділення пам’яті системою:

```
#include <malloc.h>
struct mallinfo mallinfo(void);
```

Виклик *mallinfo()* повертає статистичну інформацію у структурі mallinfo. Ця структура повертається за значенням, а не за вказівником. Її вміст також визначається в <malloc.h>:

```
/* всі розміри вказані в байтах */
struct mallinfo {
    int arena;      /* розмір сегмента даних, використовуваного malloc */
    int ordblks;    /* кількість вільних фрагментів */
    int smblks;     /* кількість швидких корзин */
    int hblks;  /* кількість анонімних відображень */
    int hblkhd;     /* розмір анонімних відображень */
    int usmblks;    /* максимальний загальний розмір пам’яті */
    int fsmblks;    /* розмір простору у звільнених блоках*/
    int uordblks;   /* загальний розмір виділеного простору */
    int fordblks;   /* розмір доступних фрагментів простору*/
    int keepcost;   /* розмір простору, який можна звільнити */
};
```

Працювати з нею просто:

```
struct mallinfo m;
m=mallinfo();
printf("free chunks: %d\n", m.ordblks);
```

В Linux також надається функція *malloc_stats()*, яка виводить в *stderr* статистку, що належить до пам’яті:

```
#include <malloc.h>
void malloc_stats(void);
```

Під час виклику *malloc_stats()* у програмі, яка інтенсивно споживає пам’ять, виходять досить великі числа:

```
Arena 0:
system bytes = 865939456
in use bytes = 851988200
Total (incl. Mmap):
system bytes = 3216519168
in use bytes = 3202567912
max mmap regions = 65536
max mmap bytes = 2350579712
```

5.6. Виділення пам’яті на основі стека

Досі йшлося про механізми виділення динамічної пам’яті, які використовували для отримання пам’яті купи або відображення. Це пояснюється тим, що купа і відображення в пам’яті є динамічними за природою. В адресному просторі процесу зазвичай є ще одна програмна конструкція – стек, в якому *містяться автоматичні змінні програми*.

Програміст цілком може використовувати стек для виділення динамічної пам’яті. Поки таке виділення не викликає переповнення стека, подібний підхід повинен залишатися простим і здійснюватися без проблем. Щоб виконати виділення динамічної пам’яті зі стека, використовуємо системний виклик **alloca()**:

```
#include <alloca.h>
void  *alloca(size_t size);
```

У разі успіху виклик *alloca()* повертає вказівник на *size* байтів у пам’яті. Ця пам’ять міститься у стеку й автоматично вивільняється після повернення функції, яка ініціювала виклик. Деякі реалізації в разі помилки повертають NULL, але більшість реалізацій *alloca()* за своєю природою не реагують на помилки і, відповідно, не можуть про них повідомляти. Збій проявляється тільки на етапі переповнення стека.

Цю функцію використовують аналогічно *malloc()*, але при цьому не потрібно вивільняти виділену пам’ять. Розглянемо приклад функції, яка відкриває вказаний файл у конфігураційному каталозі системи – ймовірно, це каталог */etc*, який машинонезалежно визначається під час компіляції. Функція повинна виділити новий буфер, скопіювати в нього конфігураційний каталог системи, а потім поєднати цей буфер з наданим ім’ям файлу:

```
int open_sysconf(const  char *file,  int  flags,  int  mode)
{
      const char *etc=SYSCONF_DIR;              /* "/etc /" */
      char  *name;
      name=alloca (strlen(etc) + strlen(file) + 1);
      strcpy(name, etc);
      strcat(name, file);
      return open (name, flags, mode);
}
```

Після повернення пам’ять, виділена за допомогою функції *alloca()*, автоматично вивільняється, оскільки стек повертається до функції, яка викликає. Таким чином, після повернення функції, що викликала *alloca()*, вже не можна використовувати цю пам’ять! Проте оскільки не потрібно виконувати будь-яку очистку (викликати *free()*), результуючий код є спрощеним.

Приклад 5.7. Наведемо приклад функції, яка реалізована за допомогою *malloc()*:

```
int open_sysconf (const char * file, int flags, int mode)
{
      const  char  *etc=SYSCONF_DIR;    /* "/etc /" */
      char  *name;
      int fd;
      name=malloc(strlen(etc) + strlen(file) + 1);
      if (!name) {
                perror("malloc");
                return -1;
      }
      strcpy(name, etc);
```

```
        strcat(name, file);
        fd=open (name, flags, mode);
        free(name);
        return fd;
}
```

Не слід використовувати пам’ять, виділену за допомогою *alloca()*, у параметрах, які передаються викликом функції, оскільки в такому випадку виділена пам’ять виявиться посеред стекового простору, зарезервованого для параметрів функцій. Наприклад, наступний код є хибним:

```
void  *alloca(size_t size);/* виділяємо розмір у байтів зі стека, */
                                /*повертаємо вказівник на нього*/

/* НЕ РОБІТЬ ТАК! */
ret=foo(x, alloca(10));  /*це як шаблон для функції*/
```

Функцію виклику foo можна використовувати як метасинтаксичну змінну, яка змінюється залежно від умов (чи як функція, чи як масив, які у процесі роботи змінюються).

В інтерфейса *alloca()* досить темна історія. У багатьох системах він працював неправильно або провокував непередбачувану поведінку. В системах з невеликим стеком та стеком, який має фіксований розмір, використання *alloca()* призводило до переповнення стека й аварійного завершення програми. У деяких інших системах *alloca()* просто не було. Згодом численні помилки й непослідовні реалізації остаточно зіпсували репутацію *alloca().*

Таким чином, якщо є потреба забезпечити переносимість створюваної програми, краще намагатися не використовувати *alloca().* Між тим у межах Linux *alloca()* є чудовим і недооціненим інструментом. Він працює надзвичайно добре – у багатьох архітектурах під час виділення пам’яті за допомогою *alloca()* треба лише підвищити вказівник стека. Цей виклик виявляється зручнішим і продуктивнішим, ніж *malloc().* Виділення невеликих обсягів пам’яті в унікальному для Linux коді *alloca()* дозволяє радикально поліпшити робочі характеристики системи.

Дублювання рядків у стеку

Один із найпоширеніших випадків використання *alloca()* пов'язаний з тимчасовим дублюванням рядка, наприклад:

```
/* ми хочемо дублювати 'song' */
char  *dup;
dup=alloca(strlen (song) + 1);
strcpy (dup, song);
/* маніпулювати 'dup' ... */
return;    /* 'dup' автоматично вивільняється */
```

Така операція потрібна досить часто. З огляду на істотне прискорення роботи під час застосування *alloca()* у системах Linux надаються варіанти *strdup()*, які дублюють зазначений рядок у стеку:

```
#define _GNU_SOURCE
#include <string.h>
char *strdupa(const char *s);
char * strndupa(const char *s, size_t n);
```

Виклик ***strdupa()*** *повертає дублікат s*. Під час виклику *strndupa() дублюється до n символів із s.* Якщо *s* довше за *n*, то дублювання припиняється на *n* і функція прикріплює нульовий байт. Ці функції мають усі переваги *alloca()*. Дубльований рядок автоматично вивільняється, як тільки повертається функція, яка зробила виклик. Стандарт POSIX не визначає функцій *alloca()*, *strdupa()* або *strndupa()*, в інших ОС вони трапляються епізодично. Якщо важливо забезпечити переносимість програми, то використовувати ці функції не рекомендується. Однак у Linux *alloca()* та її аналоги працюють цілком непогано і допомагають значно підвищити продуктивність, позбавляючи від різних маніпуляцій з виділенням динамічної пам'яті, – достатньо просто відкоригувати вказівник на кадр стека.

Масиви змінної довжини

У C99 з'явилися *масиви змінної довжини* (VLA – *The variable-length arrays*). Це масиви, геометрія яких задається під час виконання, а не

компіляції. У GNU C масиви змінної довжини вже підтримувалися деякий час, але після їх стандартизації у C99 з'явилося більше стимулів для їх використання. Масиви змінної довжини дозволяють уникнути витрат, пов'язаних з виділенням динамічної пам'яті, майже за таким же принципом, що і *alloca()*. Вони працюють таким чином:

```
for (i=0; i <n; ++ i) {
        char foo[i + 1];        /*метамасив зі змінним розміром */
/* використовуємо 'foo' ... *
}
```

Тут *foo* – це масив символів *char*, який має змінний розмір і + 1. Під час кожної ітерації циклу *foo створюється динамічно й автоматично очищається,* як тільки він виявляється за межами області визначення. Якщо замість VLA скористатися *alloca()*, то пам'ять не буде звільнена аж до повернення функції. Масив змінної довжини гарантує, що вивільнення пам'яті відбувається під час кожної ітерації циклу. Відповідно, якщо використовувати масив змінної довжини, потрібно не більше *n* байтів, але якщо використовувати виклик *alloca()*, буде потрібно n**(n + 1)/2 байтів.* Застосувавши масив змінної довжини, можна переписати функцію *open_sysconf()* таким чином:

```
int open_sysconf(const char *file, int flags, int mode)
{
        const char *etc;=SYSCONF_DIR;   / * "/etc/" * /
        char name[strlen (etc) + strlen (file) + 1];
        strcpy (name, etc);
        strcat (name, file);
        return open (name, flags, mode);
}
```

Основна відмінність між *alloca()* і масивами змінної довжини полягає в такому: пам'ять, отримана за допомогою першої функції, існує протягом життя функції, тоді як у другому випадку пам'ять є в розпорядженні змінної до її виходу з області визначення; а такий вихід може настати до повернення функції. Це може мати як позитивні, так і негативні наслідки.

У циклі розглядуваному *for* вивільнення пам’яті при кожній ітерації циклу знижує загальний обсяг споживаної пам’яті без будь-яких побічних ефектів (не потрібно тримати на підхваті додаткову пам’ять). Однак якщо з якоїсь причини потрібно, щоб пам’ять була в наявності довше, ніж протягом одного циклу, то доцільно використовувати *alloca()*.

5.7. Вибір механізму виділення пам’яті

У програміста цілком може виникнути запитання: яке рішення оптимальне для конкретного завдання? У більшості випадків доцільно скористатися *malloc()*. Проте в деяких ситуаціях рекомендовано інший підхід. У табл. 5.1 узагальнено рекомендації щодо вибору механізму виділення пам’яті.

Таблиця 5.1

Способи виділення пам’яті в Linux

Спосіб виділення	За	Проти
malloc()	Легкий, поширений	Пам’ять, що виділяється, не обов’язково заповнена нулями
calloc()	Забезпечує просте виділення масивів; пам’ять, яка повертається, заповнюється нулями	Інтерфейс складний, якщо використовується не для виділення масивів, а в інших цілях
realloc()	Змінює розмір наявного виділення	Корисний тільки при зміні розмірів наявних виділень
brk() і sbrk()	Забезпечують максимально повний контроль над купою	Занадто низькорівневі для більшості користувачів
Анонімні виділення пам’яті	Прості у використанні; такі, що розділяються; дозволяють розробнику налаштовувати рівень захисту і пропонують рекомендації-сповіщення. Оптимальні для великих відображень	Недостатньо хороші для невеликих відображень; у випадках, коли це доцільно, *malloc()* автоматично використовує анонімні відображення у пам’яті
posix_memalign()	Виділяє пам’ять, вирівняну за тією чи іншою розумною межею	Відносно нова функція, тому її переносимість ускладнена; зайвий захід, якщо немає дуже суворих вимог до вирівнювання

memalign() і valloc()	В інших UNIX-системах більш поширені, ніж *posix_memalign()*	Не належать до стандарту POSIX, меншою мірою контролюють вирівнювання, ніж *posix_memalign()*
alloca()	Дуже швидке виділення; немає необхідності явно вивільняти пам'ять; відмінно підходить для невеликих виділень	Не може повернути помилку, незручний для виділення великих областей, не працює в деяких UNIX-системах
Масиви змінної довжини	Аналогічні *alloca(),* але вивільняють пам'ять як тільки масив виходить з області видимості, а не після повернення функції	Корисні тільки для роботи з масивами; у деяких ситуаціях *alloca(), яка* вивільняє поведінку, може бути кращою; менш поширена в UNIX-системах, ніж *alloca()*

Нарешті, не слід забувати про альтернативні можливості виділення пам'яті: автоматичне і статичне виділення. Виділення автоматичних змінних у стек або глобальних змінних у купі часто виявляється простішим. У таких випадках програмісту не доводиться маніпулювати вказівниками і піклуватися про вивільнення пам'яті.

5.8. Керування необробленими байтами пам'яті

У мові *C* надається сімейство функцій для керування необробленими байтами пам'яті. За принципом роботи ці функції багато в чому нагадують інтерфейси для маніпуляцій з рядками, такі як **strcmp()** і **strcpy()**. Однак вони взаємодіють із буфером, розмір якого задається користувачем, а не спираються на припущення, що рядки завершуються нулем. Жодна із цих функцій не може повертати помилки. Запобігання збоям – завдання, яке доводиться вирішувати програмісту. Досить передати неправильну область пам'яті – і не залишиться виходу, крім як йти на вимушене порушення сегментування!

5.8.1. Установка байтів

Із функцій для керування пам'яттю найчастіше застосовують *memset()*:

```
#include <string.h>
void  *memset(void  *s, int  c, size_t  n);
```

Виклик *memset ()* заповнює перші *n* байтів області пам'яті, починаючи з вказівника на блок пам'яті для заповнення ***s***, який необхідно заповнити кодом символа ***c*** для заповнення масива. Часто цю функцію використовують для заповнення блоку пам'яті нулями:

```
/* обнуляємо [s, s+256] */
memset(s,  '\ 0',  256);
```

Функція ***bzero()*** – це ранній, застарілий інтерфейс, що з'явився в системі BSD для вирішення аналогічного завдання. У новому коді слід використовувати *memset()*, але Linux надає *bzero()* для забезпечення зворотної сумісності й переносимості на інші системи:

```
#include <strings.h>
void  bzero(void  *s, size_t  n);
```

Наступний виклик аналогічний до вище наведеного прикладу з *memset()*:

```
bzero (s, 256);
```

Звертаємо увагу: *bzero()* (поряд з іншими b-інтерфейсами) вимагає заголовка <**strings.h**>, а не <*string.h*>.

5.8.2. Порівняння байтів

Аналогічно ***strcmp()*** виклик ***memcmp()*** перевіряє два фрагменти пам'яті на еквівалентність:

```
#include <string.h>
int  memcmp(const  void  *s1,  const  void  *s2,  size_t  n);
```

Виклик порівнює перші ***n*** байтів ***s1*** із першими ***n*** байтами в ***s2***, після чого повертає 0, якщо блоки пам'яті еквівалентні. Якщо *s1* менше за *s2*, то повертається значення менше 0, а якщо *s1* більше за *s2*, то повертається значення більше 0.

BSD, у свою чергу, пропонує інтерфейс, який у наш час уже виходить з ужитку, але призначений для виконання приблизно таких же завдань:

```
#include <strings.h>
int bcmp(const void *s1, const void *s2, size_t n);
```

Після виклику *bcmp()* порівнюються перші *n* байтів у *s1* та *s2*. Якщо блоки пам'яті еквівалентні, то повертається нульове значення, якщо нееквівалентні – то нульове. Порівнювати дві структури на еквівалентність за допомогою *memcmp()* або *bcmp()* – це неточна операція. Під час заповнення може застосовуватися неініціалізоване сміття, склад якого у двох екземплярах структури різниться, тоді як в іншому ці структури ідентичні. Отже, наведений далі код є небезпечним:

```
/ * порівняння чи дві структури ідентичні (НЕБЕЗПЕЧНО) * /
int compare_dinghies(struct dinghy *a, struct dinghy *b)
{
        return memcmp(a, b, sizeof(struct dinghy));
}
```

Якщо програміст збирається порівнювати структури, то потрібно по черзі порівнювати один з одним кожний елемент цих структур. Такий похід допускає деяку оптимізацію, але, безумовно, є більш трудомістким, ніж небезпечна робота із застосуванням *memcmp()*. Наведемо приклад еквівалентного коду:

```
/* Ідентичні чи дві шлюпки? */
int compare_dinghies(struct dinghy *a, struct dinghy *b)

{
        int ret;
        if (a->nr_oars < b->nr_oars)
                return -1;
        if (a->nr_oars > b->nr_oars)
                return 1;
        ret = strcmp (a->boat_name, b->boat_name);
        if (ret)
                return ret;
/* і т. д. для кожного елемента ... */
}
```

5.8.3. Переміщення байтів

Функція memmove() копіює перші *n* байтів із src в dst, повертаючи *dst*:

```
#include <string.h>
void *memmove(void *dst, const void *src, size_t n);
```

BCD (Boot Configuration Data) – файл конфігурації завантаження системи, містить дані про установлену систему та параметри її завантаження. Цей файл використовує диспетчер завантаження, BCD надає інтерфейс, який виходить з ужитку для вирішення аналогічної задачі:

```
#include <strings.h>
void bcopy(const void *src, void *dst, size_t n);
```

Звертаємо увагу: обидві функції, *memmove()* та *bcopy()*, набувають однакових параметрів, однак *порядок перших двох в bcopy () є зворотним.* І *bcopy()*, і *memmove()* дозволяють безпечно обробляти області пам'яті, які перетинаються (наприклад, якщо частина *dst* міститься всередині *src*). Таким чином, можна, наприклад, переміщати байти в певній області пам'яті вгору-вниз.

Така ситуація трапляється рідко, а програміст має знати про її настання, тому стандарт *C* визначає варіант *memmove()*, який не підтримує такого взаємного перетину областей пам'яті. Потенційно цей варіант працює швидше:

```
#include <string.h>
void *memcpy(void *dst, const void *src, size_t n);
```

Функція працює аналогічно *memmove()*, із застереженням, що перетин *dst* та *src* забороняється. Якщо це відбувається, то результат не визначений.

Ще одну безпечну функцію для копіювання називають *memccpy()*:

```
#include <string.h>
void  *memccpy(void *dst, const void *src, int c, size_t n);
```

Функція *memccpy()* працює так само, як і *memcpy()*, але копіювання зупиняється, якщо функція знаходить байт ***c*** серед перших ***n*** байтів у ***src***. Виклик повертає вказівник на байт, розміщений в *dst* після ***c***, або NULL,

якщо ***c*** не був знайдений. Нарешті, можна використовувати *mempcpy()* для покрокового проходу через пам’ять:

```
#define _GNU_SOURCE
#include <string.h>
void  *mempcpy(void  *dst, const void *src, size_t  n);
```

Функція *mempcpy()* працює так само, як і *memcpy()*, але вона повертає вказівник на наступний байт після останнього скопійованого байта. Вона корисна, якщо безліч даних потрібно скопіювати в послідовно розміщені області пам’яті. Однак покращення незначне, оскільки значення, що повертається, – це просто сума *dst* + *n*. Ця функція є унікальною для GNU.

5.8.4. Пошук байтів

Функції *memchr()* та *memrchr()* знаходять заданий байтув блоці пам’яті:

```
#include <string.h>
void *memchr(const void *s, int c, size_t n);
```

Функція *memchr()* переглядає *n* байтів в області пам’яті, на яку спрямований вказівник *s*, і шукає символ *c*, що інтерпретується як *unsigned char:*

```
#define _GNU_SOURCE
#include <string.h>
void *memrchr(const void *s, int c, size_t n);
```

Виклик повертає вказівник на перший байт, що збігається із *c*, або NULL, якщо *c* не знайдений.

Функція *memrchr()* рівнозначна *memchr()*, але вона виконує пошук назад, починаючи з байта *n*, на який вказує *s*, а не вперед – від першого байта. На відміну від **memchr()**, **memrchr ()** – це розширення GNU, а не частина мови *C*.

Більш складні пошукові завдання вирішують за допомогою функції *memmem()*. Вона шукає у блоці пам’яті довільний масив байтів:

```
#define  _GNU_SOURCE
#include <string.h>
void   *memmem(const void *haystack, size_t haystacklen, const void * needle,
                                                    size_t needlelen);
```

Функція *memmem()* повертає вказівник на перший екземпляр підблоку *needle*, який має довжину *needlelen* байтів і перебуває у блоці пам’яті *haystack* довжиною *haystacklen* байтів. Якщо функції не вдається знайти *needle* в *haystack*, вона повертає NULL. Ця функція також є розширенням GNU.

5.8.5. Переклацування байтів

Бібліотека Linux *C* надає інтерфейс для тривіального згортання байтів даних:

```
#define _GNU_SOURCE
#include <string.h>
void  *memfrob(void  *s, size_t  n);
```

Виклик *memfrob()* затемнює перші *n* байтів у пам’яті, починаючи із *s*. При цьому застосовується виключаюче «АБО» (XOR) до кожного байту з числом 42. Виклик повертає *s*. Ефект від виклику *memfrob()* може бути скасований викликом *memfrob()* знову в тій самій області пам’яті, тому наступний код є холостою командою, яку застосовують із *secret*:

```
memfrob (memfrob (secret, len), len);
```

Ця функція у жодному разі не є повноцінною (і навіть прийнятною) заміною шифрування; її використання обмежене нескладним затемненням рядків. Функція є унікальною для GNU.

5.9. Блокування пам’яті

В Linux реалізується *підкачування сторінок за вимогою*. Відповідно до цієї технології сторінки викликаються з диска у міру необхідності й скидаються на диск як тільки стають непотрібні. Завдяки цьому віртуальні адресні простори процесів, які працюють у системі, можуть ніяк не зіставлятися із загальним обсягом фізичної пам’яті, оскільки вторинні носії здатні створювати видимість наявності фактично необмеженого обсягу фізичної пам’яті. Таке підкачування відбувається прозоро, і додатки зазвичай взагалі не повинні знати, як підкачування організовується в ядрі Linux. Однак можливі дві ситуації, в яких додатку буває доцільно впливати на процедури підкачування, що відбуваються у системі:

Детермінізм

Додатки, що працюють з обмеженнями за часом, вимагають детерміністської поведінки. Якщо деякі звернення до пам’яті призводять до виникнення сторінкових помилок, пов’язаних з витратними операціями дискового «введення – виведення», то додаток може відчувати потребу в додатковому часі для роботи. Якщо додаток може гарантувати, що необхідні йому сторінки завжди залишаються у фізичній пам’яті й не скидаються на диск, то він може забезпечити і відсутність сторінкових помилок при зверненнях до пам’яті. Так досягається узгодженість і детермінізм, а також поліпшується продуктивність.

Безпека

Якщо в пам’яті зберігаються конфіденційні відомості, їх можна звідти вивантажити і зберігати на диску в незашифрованому вигляді. Наприклад, якщо закритий ключ користувача зазвичай зберігається на диску у зашифрованому вигляді, то незашифрована копія ключа, що містилася у пам’яті, може виявитися у файлі підкачки. У середовищі, де діють

підвищені вимоги до безпеки, така поведінка може виявитися неприйнятною. Якщо в додатку виникає подібна проблема, він може зажадати, щоб область, яка містить ключ, завжди залишалася в межах фізичної пам’яті.

Зрозуміло, якщо змінити принципи підкачування, що діють в ядрі, це може негативно позначитися на загальній продуктивності. Детермінізм або безпека в одному додатку можуть покращитися, але поки його сторінки будуть заблоковані в пам’яті замість них будуть підкачуватися сторінки іншої програми. Якщо довіряти алгоритмам ядра, то можна бути впевненим, що алгоритм оптимальним чином буде підбирати сторінки для вивантаження з пам’яті – тобто вивантажувати сторінки, які з найменшою вірогідністю знадобляться користувачеві у найближчому майбутньому, тому якщо розробник змінить таку поведінку, ядро може вивантажити якісь порівняно затребувані сторінки.

5.9.1. Блокування частини адресного простору

Стандарт POSIX 1003.1b-1993 визначає два інтерфейси для «блокування» однієї або декількох сторінок у фізичній пам’яті. Це гарантує, що такі сторінки за жодних умов не будуть вивантажені на диск. Перша функція блокує заданий інтервал адрес:

```
#include <sys / mman.h>
int  mlock(const  void  *addr, size_t  len);
```

Виклик *mlock()* блокує область віртуальної пам’яті, яка починається з *addr* і продовжується у фізичну пам’ять на *len* байтів. У разі успіху цей виклик повертає 0; у разі помилки він повертає -1 і встановлює *errno* відповідне значення. Успішний виклик блокує в пам’яті всі ті сторінки, які містять [addr, addr+len]. Наприклад, якщо у виклику вказано лише один байт, то в пам’яті блокується вся сторінка, на якій він міститься. Стандарт POSIX наполягає, щоб *addr* був вирівняний по межі сторінок. Linux не

вимагає суворого виконання цього правила, за потреби округляє *addr* вниз до найближчої цілої сторінки. Однак якщо у програмі необхідно забезпечити переносимість в інші системи, то потрібно гарантувати, щоб *addr* буде вирівняна по межі сторінок. Можливі такі коди *errno*:

EINVAL – параметр *len* є негативним.

ENOMEM – викликаюча сторона спробувала заблокувати більше сторінок, ніж допускає ліміт ресурсу RLIMIT_MEMLOCK.

EPERM – ліміт ресурсу RLIMIT_MEMLOCK дорівнював 0, але процес не мав можливості CAP_IPC_LOCK.

Як приклад припустимо, що програма тримає в пам'яті розшифрований рядок. Процес може заблокувати сторінку, яка містить цей рядок, за допомогою коду на кшталт такого:

```
int ret;
/* блокуємо 'secret' в пам'яті */
ret=mlock(secret, strlen (secret));
if (ret)
perror("mlock");
```

5.9.2. Блокування всього адресного простору

Якщо процесу потрібно утримувати весь свій адресний простір у фізичній пам'яті, то інтерфейс *mlock()* для цього досить незручний. У такому випадку в додатках реального часу POSIX визначає спеціальний системний виклик, який блокує весь адресний простір:

```
#include <sys / mman.h>
int  mlockall(int flags);
```

Виклик *mlockall()* блокує всі сторінки в адресному просторі актуального процесу, утримуючи їх у фізичній пам'яті. Параметр *flags*, який керує такою поведінкою і являє собою побітове «АБО», може набувати одного з таких двох значень:

MCL_CURRENT – це значення означає, що розробник наказує *mlockall()* заблокувати в адресному просторі процесу усі сторінки, відображені у цей момент: стек, сегмент даних, відображені файли і под.;

MCL_FUTURE – вказавши це значення, розробник змушує *mlockall()* гарантувати, щоб всі сторінки, які відображатимуться у цей адресний простір у майбутньому, також блокуватимуться в пам'яті.

У більшості додатків вказується побітове «АБО» із двох цих значень.

У разі успіху цей виклик повертає 0; у випадку помилки він повертає -1 і присвоює *errno* один із таких кодів помилки:

EINVAL – параметр *flags* є негативним;

ENOMEM – викликаюча сторона спробувала заблокувати більше сторінок, ніж допускає ліміт ресурсу RLIMIT_MEMLOCK;

EPERM – ліміт ресурсу RLIMIT_MEMLOCK дорівнював 0, але процес не мав можливості CAP_IPC_LOCK.

5.9.3. Розблокування пам'яті

Щоб розблокувати сторінки, які містяться у фізичній пам'яті, і знову дозволити ядру за потреби вивантажити їх на диск, POSIX стандартизує ще два інтерфейси:

```
#include <sys/mman.h>
int munlock(const void *addr, size_t len);
int munlockall(void);
```

Системний виклик *munlock()* розблокує сторінки, починаючи з адреси *addr* й аж до *len* байтів. Він скасовує дію *mlock()*. Системний виклик *munlockall()* скасовує дію *mlockall()*. Обидва виклики в разі успіху повертають 0, а у випадку помилки повертають -1, привласнюючи *errno* одне з таких значень:

EINVAL – параметр *len* є неприпустимим (тільки для *munlock()*);

ENOMEM – деякі із зазначених сторінок є неприпустимими;

EPERM – ліміт ресурсу RLIMIT_MEMLOCK дорівнював 0, але процес не має можливості CAP_IPC_LOCK, тобто блокування в пам’яті будь-якої кількості сторінок.

Блокування пам’яті не вкладаються один в одного, тому окремий виклик *mlock()* або *munlock()* вивільнить заблоковану сторінку незалежно від того, скільки разів вона блокувалася за допомогою *mlock()* або *mlockall()*.

5.9.4. Ліміти блокування

Блокування пам’яті може впливати на загальну продуктивність системи. Справді, якщо буде заблоковано занадто багато сторінок, то виділення пам’яті можуть пробуксовувати, оскільки Linux лімітує кількість сторінок, які може заблокувати процес. Процеси, які мають можливість CAP_IPC_LOCK, можуть блокувати у пам’яті будь-яку кількість сторінок. Процеси, що не мають такої характеристики, можуть блокувати всього RLIMIT_MEMLOCK байтів. За замовчуванням цей ліміт дорівнює 32 Кбайт – цілком достатньо, щоб зберігати в пам’яті один-два секрети, але такий обсяг ще не впливає на загальну продуктивність.

З’ясуємо чи міститься сторінка у фізичній пам’яті. Для налагодження і діагностики Linux використовує функцію ***mincore()***, що дозволяє визначати, чи міститься зазначений діапазон у фізичній пам’яті, чи вивантажений на диск:

```
#include <unistd.h>
#include <sys/mman.h>
int mincore(void *start, size_t length, unsigned char *vec);
```

Виклик *mincore()* надає масив, який вказує, які сторінки, що відображаються, містяться у фізичній пам’яті у момент системного виклику. Виклик повертає масив за допомогою ***vec*** й описує сторінки, починаючи зі ***start*** (потрібне вирівнювання по межах сторінок) і до *length* байтів (вирівнювання по межах сторінок не потрібне). Кожний байт у *vec*

відповідає одній сторінці в наданому діапазоні. Перший байт описує першу сторінку в цьому діапазоні, другий – другу і т. д. лінійно. Отже, *vec* має бути досить великий, щоб утримувати (*length*-1 + розмір_сторінки) / розмір_сторінки байтів. Наймолодший біт у кожному байті дорівнює 1, якщо сторінка міститься у фізичній пам’яті, та 0, якщо не міститься. Решта бітів у цей час залишаються невизначеними і зарезервовані для використання в майбутньому.

У разі успіху виклик повертає 0, при помилці він повертає -1 і присвоює *errno* одне з таких значень:

EAGAIN – недостатньо ресурсів ядра для виконання цього запиту;

EFAULT – параметр *vec* вказує на недійсну адресу;

EINVAL – значення параметра *start* не вирівняно по межах сторінок;

ENOMEM – у діапазоні *[address, address+1]* міститься пам’ять, яка не є частиною файлового відображення.

У наш час цей системний виклик правильно працює тільки з файловими відображеннями, створеними за допомогою MAP_SHARED. Це значно обмежує практичне застосування виклику.

5.10. Стратегія пристосування / адаптивне (Opportunistic Allocation) виділення пам’яті

Linux використовує (*opportunistic allocation strategy*) стратегію пристосування / адаптивного виділення пам’яті. Коли процес вимагає від ядра додаткової пам’яті – наприклад, у разі збільшенні свого сегмента даних або створення нового відображення в пам’яті, – ядро фіксує *зобов’язання* виділити пам’ять, не надаючи ніякого фізичного сховища. Лише коли процес записує інформацію в новий виділений обсяг пам’яті, ядро задовольняє це зобов’язання, надаючи для обіцяного обсягу пам’яті фізичний простір. Ядро робить це послідовно для кожної сторінки, у міру

необхідності, виконуючи підкачування сторінок на вимогу і копіювання під час запису.

Така поведінка має деякі сильні риси. По-перше, подібне ліниве виділення пам’яті дозволяє ядру відкладати більшу частину роботи до останнього моменту, в який її необхідно виконати, – при тому, що, можливо, деякі виділення і не доведеться задовольняти. По-друге, оскільки запити задовольняються послідовно для кожної сторінки, і при цьому на вимогу, фізичний простір потрібний тільки для пам’яті, яка справді використовується у цей момент. У кінцевому рахунку обсяг обіцяної пам’яті може значно перевищувати кількість доступної пам’яті навіть з урахуванням поділу підкачування. Останню можливість називають *надмірним виділенням*.

Надмірне виділення і повна витрата пам’яті

(OOM – Out Of Memory)

Надмірне виділення пам’яті дозволяє системі одночасно запускати набагато більше додатків (при цьому досить об’ємних), ніж це було б у ситуації, коли кожна сторінка пам’яті, щодо якої надходить запит, має бути забезпечена фізичним сховищем уже в момент виділення, а не в момент фактичного використання. У разі надмірного виділення для відображення в пам’ять файлу, що має розмір 2 Гбайт, місце в пам’яті потрібно тільки для сторінки, на яку процес якраз записує дані. Аналогічно без надмірного виділення кожний виклик *fork()* зажадав би досить вільного простору, щоб продублювати весь адресний простір, незважаючи на те, що більшість сторінок так і не піддалися б копіюванню при запису.

Розглянемо що станеться, якщо процеси спробують задовольнити більше зобов’язань, ніж дозволяє доступний обсяг фізичної пам’яті та область підкачки у конкретній системі. У такому випадку один запит або більше задоволені не будуть. Ядро вже «пообіцяло» виділити пам’ять – системний виклик, який запитує пам’ять, був успішний, – тому ядру

залишається примусово завершити (вбити) процес, щоб вивільнити потрібну пам’ять.

Коли через надмірне виділення виникає нестача пам’яті й не вдається задовольнити запит, незважаючи зобов’язання, яке раніше було дано, виникає умова *нестачі пам’яті* (OOM). У разі настання такої ситуації ядро задіює механізм, який називають **OOM killer**, і вибирає процес для примусового завершення. У розглядуваному випадку ядро намагається знайти найменш важливий процес, який споживає при цьому максимальну кількість пам’яті.

Ситуація OOM виникає рідко, тому якщо надлишкове виділення буде дозволено, то в першу чергу це принесе розробникові величезну користь. Проте несподіваний дефіцит пам’яті дуже небажаний, а коли важко передбачити завершення процесу в подібних умовах, часто є неприпустимим.

У системах, в яких таку ситуацію справді краще не допускати, ядро дозволяє деактивувати надлишкове виділення пам’яті у файлі */proc/sys/vm/overcommit_memory*, а також аналогічний параметр *sysctl vm.overcommit_memory*.

За замовчуванням цей параметр має значення 0. За такого значення ядро має застосовувати *евристичну стратегію надлишкового виділення*: розумно використовувати надлишкове виділення, але відхиляти запити, які здаються кричущими. Значення 1 забезпечує успішне виконання всіх зобов’язань, при цьому попередження пропадає даремно. Деякі додатки інтенсивно споживають пам’ять (зокрема, дослідні інструменти), часто запитують набагато більше пам’яті, ніж їм насправді потрібно, тому такий параметр доцільний.

Значення 2 повністю відключає надлишкове виділення й активує *суворий облік*. У цьому випадку зобов’язання з виділення пам’яті обмежені обсягом розділу підкачки й настроюється відсоток фізичної пам’яті. Цей

відсоток задають у файлі */proc/sys/vm/overcommit_ratio* або за допомогою аналогічного параметра *sysctl*, який дорівнює *vm.overcommit_ratio*. За замовчуванням діє значення 50, що обмежує зобов'язання з виділення пам'яті обсягом розділу підкачки й 50 % фізичної пам'яті. У фізичній пам'яті міститься ядро, сторінки підкачки, сторінки, зарезервовані у системі, заблоковані сторінки і под., тому тільки частина цієї пам'яті справді доступна для підкачки і гарантовано може використовуватися для задоволення зобов'язань.

Слід бути обережними із суворим обліком! Багато розробників систем сахаються від OOM killer та вважають суворий облік панацеєю. Проте додатки часто виконують безліч непотрібних операцій виділення, які сильно захоплюють «надлишкову» область. Забезпечення такої поведінки – одна з основних причин, з якої машини оснащуються віртуальної пам'яттю.

Контрольні запитання

1. Що таке адресний простір процесу?
2. Які області пам'яті містить процес?
3. Що таке динамічна пам'ять?
4. Як змінити розмір виділених областей?
5. Як вивільнити динамічну пам'ять?
6. Що таке вирівнювання даних?
7. У чому полягає метод анонімних відображень у пам'яті?
8. Як створюються анонімні відображення?
9. Як отримати статистичну інформацію щодо виділення пам'яті?
10. Яким чином здійснюється виділення пам'яті на основі стека?
11. Яким чином здійснюється розблокування та розблокування пам'яті?

Розділ 6

ЗАСТОСУВАННЯ ТЕХНОЛОГІЇ КОНТЕЙНЕРІВ

6.1. Розробка та впровадження програмного забезпечення за допомогою технології контейнерів

6.1.1. Контейнери та їх призначення

Контейнери докорінно змінюють спосіб розроблення, поширення і функціонування ПЗ. Розробники можуть створювати ПЗ у локальній системі, яке буде працювати однаково у будь-якому операційному середовищі – у програмному комплексі ІТ-відділу або на ноутбуці користувача, або у хмарному кластері. Масштаби переходу у застосуванні контейнерів в інформаційних технологіях стрімко зростають під час розроблення і впровадження ПЗ у різних сферах промисловості.

Контейнери (*containers*) являють собою засоби інкапсуляції додатків разом із залежностями (поєднання в одному місці додатка і залежностей).

Тобто *контейнер – це стандартна одиниця ПЗ, в яку упаковано додаток з усіма необхідними для його роботи залежностями – кодом програми, середовищем запуску, системними інструментами, бібліотеками і налаштуваннями*. Контейнер не залежить від ресурсів або архітектури хоста, на якому він працює. На перший погляд контейнери можуть здаватися всього лише спрощеною формою віртуальних машин (*virtual machines* – VM) – як і віртуальна машина, контейнер містить ізольований екземпляр ОС, який можна використовувати для запуску додатків. Технологія віртуалізації дозволяє на одному комп’ютері одночасно запускати декілька ОС разом з основною системою, встановленою на комп’ютері. Віртуальна машина ізольована від решти системи, тобто ПЗ всередині неї не може вплинути на систему самого

комп’ютера або керувати нею. Це ідеальне середовище для тестування інших ОС, включаючи бета-випуски, оцінювання даних, заражених вірусом, створення резервних копій ОС, запуск ПЗ та додатків на ОС, для яких вони спочатку не призначені.

Для серверів декілька ОС виконуються разом з виділеним ПЗ (гіпервізором), яке використовується для керування ними, тоді як настільні комп’ютери зазвичай використовують одну ОС для запуску інших ОС у межах її вікон. Кожна віртуальна машина має власне віртуальне обладнання, зокрема центральний процесор, пам’ять, жорсткі диски, мережеві інтерфейси та інші пристрої. Віртуальне обладнання потім зіставляється з фактичним обладнанням на фізичному комп’ютері. Це дозволяє заощаджувати за рахунок усунення необхідності у фізичних апаратних системах і супутніх витратах на обслуговування, а також знижує вимоги до потужності й охолодження. Але контейнери мають деякі переваги, що забезпечують такі варіанти використання, які важко або неможливо реалізувати у звичайних віртуальних машинах:

– *контейнери спільно використовують ресурси основної ОС, що робить їх на порядок більш ефективними.* Контейнери можна запускати і зупиняти за частки секунди. Для додатків, що запускаються в контейнерах, накладні витрати мінімальні або їх немає взагалі порівняно з додатками, що запускаються безпосередньо під керуванням основної ОС; переносимість контейнерів забезпечує потенційну можливість усунення цілого класу програмних помилок, що викликаються незначними змінами робочого середовища, – позбавляється обґрунтування давній аргумент розробника «але це працює на моєму комп’ютері»;

– розробники можуть одночасно запускати десятки контейнерів, що дає можливість імітації роботи промислової розподіленої системи. Інженери з експлуатації можуть запустити на одному хості набагато більше контейнерів, ніж при використанні окремих віртуальних машин;

крім того, контейнери надають переваги кінцевим користувачам і розробникам без необхідності розгортання програми у хмарі. Користувачі можуть завантажувати і запускати складні додатки без багатогодинної метушні з конфігурацією і проблемами під час установки й при цьому не турбуватися про будь-які зміни в їх локальних системах. У свою чергу, розробники подібних додатків можуть уникнути проблем, пов'язаних з відмінностями в конфігураціях призначених для користувача середовищ і з доступністю залежностей для цих додатків.

6.1.2. Відмінності між віртуальними машинами та контейнерами

Є два основних варіанти віртуалізації, а саме два підходи до створення незалежних ізольованих обчислювальних середовищ (просторів) на одному фізичному сервері [4]:

– віртуальні машини, яким потрібний гіпервізор,

– віртуальні контейнери.

Для першого підходу для кожної віртуальної машини використовується власна гостьова ОС. Для другого підходу усі контейнери застосовують ядро однієї хостової машини. Тобто у першому випадку створюються *неоднорідні* обчислювальні середовища на одному комп'ютері, а у другому – тільки *однорідні*.

Розглянемо, у чому полягає відмінність контейнера від віртуалізації. Спочатку віртуалізація розроблялася з метою позбавлення від подібних проблем, але в ній є суттєві недоліки:

– повільне завантаження;

– можлива плата за надання додаткового простору;

– не всі віртуальні машини підтримують сумісне використання;

– підтримка VM часто вимагає складного налаштування;

– образ може бути занадто великим, оскільки «додаткова ОС» додає гігабайт простору у проект поверх ОС, а у більшості випадків на сервер ставиться кілька VM, які займають ще більше місця.

Принципові відмінності щодо використання віртуальних машин і контейнерів полягають у такому:

– метою застосування віртуальної машини є повна емуляція іншого програмного (операційного) середовища;

– мета застосування контейнера – зробити додатки переносимими, ізольованими, самодостатніми.

На рис. 6.1 показано три додатки, що працюють в окремих віртуальних машинах на одному хості. ***Гіпервізор*** (***монітор VM*** дозволяє одночасне, паралельне виконання декількох ОС на одному і тому ж хост-комп'ютері) для створення і запуску VM керує доступом до ОС, яка міститься нижче до апаратури, а також за потреби інтерпретує системні виклики.

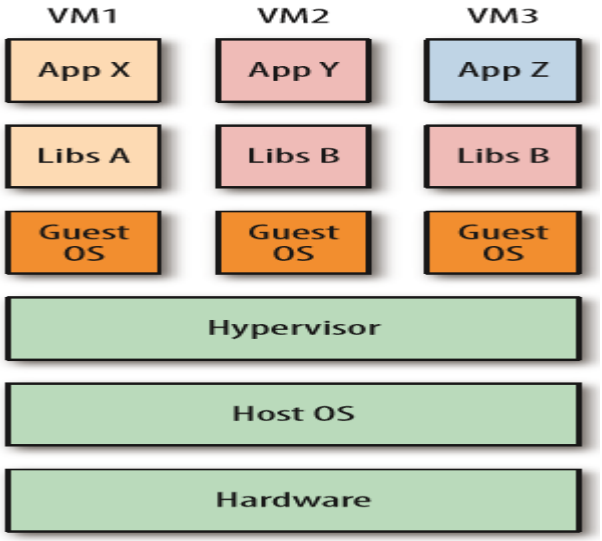


Рис. 6.1. Три віртуальні машини, що працюють на одному хості [4]

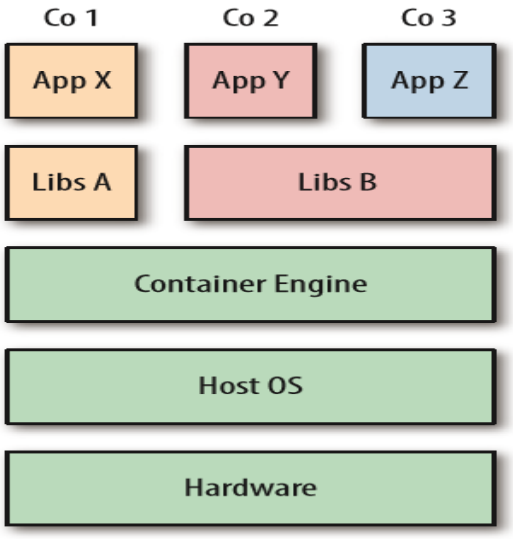


Рис. 6.2. Три контейнери, що працюють на одному хості [4]

Для кожної VM необхідні: повна копія ОС, яка запускається, й усі бібліотеки підтримки. На противагу описаній схемі на рис. 6.2 показано, як ті самі три додатки можуть працювати в системі з контейнерами. На відміну від віртуальних машин, ядро 2 хоста спільно використовується (розділяється) працюючими контейнерами. Це означає, що контейнери завжди обмежуються використанням того ж ядра, яке функціонує на хості.

Таким чином, усі компоненти, потрібні для запуску додатка, упаковуються як один образ і можуть бути використані повторно. *Додаток у контейнері працює в ізольованому середовищі і не використовує пам'ять, процесор або диск хостової ОС. Це гарантує ізольованість процесів всередині контейнера*.

Перевагами контейнерів є:

– *гнучке середовище*, контейнери можна створювати значно швидше, ніж екземпляри VM, їх розмір невеликий – вимірюється в мегабайтах, на відміну від розмірів VM, яка містить ОС, розмір якої може бути декілька гігабайтів; крім того, контейнери швидко запускаються;

– *підвищена продуктивність,* кожний контейнер можна розглядати як окремий мікросервіс, його оновлення не потребує синхронізації;

– *керування версіями,* дозволяє відстежувати версії контейнера та їх відмінності;

– *переносимість середовища обчислення,* контейнери інкапсулюють ОС та залежності додатків, тобто легко переносити образ контейнера з одного середовища в інше, наприклад, Windows/Linux;

– *стандартизація*, більшість контейнерів основані на відкритих стандартах і можуть працювати в усіх основних дистрибутивах Windows/Linux;

– *безпека,* контейнери ізолюють процеси одного контейнера від іншого, тобто будь-яке оновлення в одному контейнері не впливає на інший.

До ***недоліків контейнерів*** можна віднести такі:

– *підвищена складність,* якщо працює *n*-контейнерів з додатком, це збільшує коефіцієнт складності, а керування множиною контейнерів може бути складним завданням в умовах виробництва;

– *більшість контейнерних технологій, таких як Docker,* засновані на Linux-контейнерах (LXC), тому їх запуск у середовищі Microsoft може викликати певні складнощі;

– *незрілість,* контейнери – це відносно нова технологія на ринку, тому в разі виникнення проблем необхідно витратити певний час для їх вирішення.

6.1.3. Класифікація контейнерів

Для створення контейнерів ОС використовують такі технології:

– *LXC* (*Linux Container*), технологія віртуалізації на рівні операційної системи для запуску декількох ізольованих примірників ОС Linux на одному комп'ютері. LXC не використовує віртуальних машин, а створює віртуальне оточення із власним простором процесів та мережевим стеком. Усі примірники LXC використовують один примірник ядра ОС;

– *Open VZ,* технологія базується на ядрі ОС Linux і дозволяє на одному фізичному сервері створювати й запускати ізольовані один від одного копії обраної ОС (*Debian, CentOS, Ubuntu*);

– *Linux VServer* (*Linux Virtual Server*), технологія дозволяє створювати декілька віртуальних серверів, які працюють незалежно під керуванням одного ядра ОС;

– *BSD Jailis*, технологія віртуалізації в системі *FreeBSD*, що дозволяє створювати всередині однієї ОС FreeBSD кілька FreeBSD, які працюють незалежно, на тому ж ядрі ОС, але абсолютно незалежно налаштовуються з незалежним набором встановлених додатків;

– *Solaris Container* (*Solaris Zone)*, технологія, що дозволяє розділити на програмному рівні Solaris 10 ОС на контейнери (зони), майже самостійні ОС, які можуть мати окремі незалежні ресурси (процесори, пам'ять, дисковий простір) і власних користувачів.

Для створення контейнерів для додатків на різних платформах Windows, Mac, Linux Ubuntu використовують контейнерну технологію

Docker, а для ОС CoreOS на базі Linux – контейнерну технологію *Rocket*. Це власний контейнерний движок *Rocket (rkt – keep reading)*, який дозволяє працювати з публічними хмарними додатками.

6.1.4. Docker-контейнери

Нині Docker є найпопулярнішою платформою керування контейнерами. Docker-контейнери задіяні протягом усього циклу розроблення ПЗ, їх основне застосування полягяє у розгортанні ПЗ. Це ПЗ з відкритим кодом, принцип роботи якого найпростіше порівняти із транспортними контейнерами, які є стандартними та їх можна переміщувати між різними видами транспорту (вантажівки, поїзди, кораблі) або з картриджами, які використовувалися у приставках для ігор – вставляють картридж у приставку, і гра одразу ж працює. Docker функціонує за схожим принципом – дозволяє запускати своє ПЗ настільки просто, що це можна порівняти зі вставкою картриджа і натисканням кнопки ON на приставці.

Така ідея була перенесена на ІТ-сферу для переміщення коду між різними програмними середовищами з мінімальними обсягами роботи. Коли розробляється додаток, необхідно надати код разом з усіма його складовими, такими як бібліотеки, сервер, бази даних і под. При цьому може виникнути така ситуація, коли додаток працює на комп’ютері розробника, але відмовляється працювати на комп’ютері користувача. Ця проблема вирішується через *створення ПЗ, яке не залежить від системи.*

Саме контейнери Docker спрощують перенесення програмних додатків.

У 2008 році Соломон Хайкс (*Solomon Hykes*) заснував компанію *dotCloud* для реалізації хмарної технології «платформа як послуга» (*Platform-as-a-Service – PaaS*), яка повністю незалежна від будь-якої мови програмування. У березні 2013 року компанія *dotCloud* відкрила початкові коди *Docker*, ключової компоненти, ядра програмного сервісу *dotCloud*.

Між тим деякі компанії побоювалися розголошення своїх «чудо-секретів», компанія *dotCloud* зрозуміла, що *Docker* принесе набагато більше користі, якщо стане відкритим проектом, керованим вільною спільнотою.

Таким чином, *контейнери* – це *технологія упаковки і запуску додатків Windows, Linux, MacOS у різних локальних середовищах та у хмарі* [4, 5].

64-бітовий Linux-контейнер працює тільки на хості зі встановленою 64-бітовою версією ОС Linux. Контейнер на Windows Server може працювати тільки на хості під керуванням ОС Windows Server. Для Linux – це просто процеси, які використовують механізми ізоляції ядра.

Контейнер створюється і складається з образів. Його можна легко видалити і знову створити за короткий проміжок часу.

Образ – базовий елемент кожного контейнера.

Образи зберігаються у реєстрі. Реєстр – це *сервер, на якому зберігаються образи*. Порівняємо його з *GitHub* (веб-портал для хостингу IT-проектів та їх спільної розробки, дозволяє контролювати версії проекту): можна витягнути образ з реєстру, щоб розгорнути його локально, і так само локально можна вносити у реєстр створені образи.

Контейнеризація – це віртуалізація на рівнів ОС (тобто не апаратна), коли ядро ОС підтримує декілька ізольованих екземплярів простору користувача замість одного.

Простір користувача – це адресний простір віртуальної пам’яті ОС, що відводиться для програм користувача.

Усі контейнери використовують ядро хостової машини. Ядро – це центральна частина ОС, яка координує для додатків доступ до ресурсів комп’ютера: процесорного часу, пам’яті, зовнішнього апаратного забезпечення, внутрішніх пристроїв «введення – виведення»; ядро надає сервіси файлової системи, мережевих протоколів. Контейнер збирається поверх ядра, але ядро не надає всі інтерфейси API (*Application Programing Interface*) та служби, які необхідні для запуску програми. Більшість із них

надаються системними файлами (бібліотеками), які працюють на рівні вище ядра в режимі користувача. Оскільки контейнер ізольований від середовища режиму користувача сервера, контейнеру потрібно власна копія цих системних файлів режиму користувача, які упаковуються в базовий образ. Базовий образ виступає як основний рівень, на якому збирається контейнер, надаючи йому служби ОС, які не надаються ядром.

Образи контейнерів являють собою набір файлів, організованих у стек шарів, розміщених на локальному комп'ютері або у віддаленому реєстрі контейнерів. *Образ контейнера складається з*:

– файлів ОС режиму користувача, необхідних для підтримання додатка;

– будь-яких середовищ виконання або залежностей додатків;

– будь-якого іншого файлу конфігурації, потрібного для правильної роботи додатка.

Набір утиліт, які забезпечують роботу з контейнерами, називають «*движком*» (*container engine*), а його частина, яка відповідає за запуск та моніторинг контейнерів, – *середовищем виконання (container runtime).* Середовище виконання присвоює контейнерам власні ідентифікатори (ID).

Таким чином, Docker – це стандартизоване пакетне ПЗ, призначене для розроблення, розгортання проектів та використання розроблених додатків, які є переносимими та самодостатніми, тоді як метою віртуальної машини є емуляція іншого операційного середовища. Docker дозволяє відокремити додаток розробника від інфраструктури і запустити будь-який додаток, який безпечно ізольований у контейнері. Тобто Docker дозволяє упаковувати, розповсюджувати, встановлювати та використовувати *opensource*-додатки.

Система написана мовою *Go*. Працює в різних UNIX/Linux-системах(x86_64, ARM, s390x, ppc64le), підтримується у Windows (x86_64).

Платформа Docker складається із двох окремих компонентів:

• *Docker Engine* (рос. движок *Docker*,) механізму, що відповідає за створення і функціонування контейнерів, це *клієнт-серверний додаток*;

• *Docker Hub* хмарного сервісу для поширення контейнерів.

Механізм *Docker Engine* надає ефективний і зручний інтерфейс для запуску контейнерів. До цього для запуску контейнерів, що використовують таку технологію, як, наприклад, LXC (Linux Container), були потрібні неабиякий запас спеціальних знань у цій сфері та великий обсяг ручної роботи.

Docker Hub – *публічний репозиторій* з інтерфейсом, який надається *Docker Inc* (назва компанії розробника *Docker Hub).* Цей ресурс є джерелом «офіційних» образів, розроблених командою Докер або створених у співпраці з розробником ПЗ. Для офіційних образів перераховані їх потенційні уразливості. Ця інформація відкрита для будь-якого зареєстрованого користувача. Доступні як безкоштовні, так і платні акаунти.

Компанія *Docker* розділила движок *Docker* на два продукти:

– *Docker Community Edition (CE*) – це безкоштовне ПЗ;

– *Docker Enterprise* – це платна версія системи, яка надає користувачам додаткові можливості у сфері підтримки систем, керування ними та безпекою.

6.1.5. Архітектура Docker

Схема взаємодії основних компонентів платформи Docker зображена на рис. 6.3. Docker використовує клієнт-серверну технологію *Docker Engine* та складається з ***утиліти docker*** (на клієнті), яка звертається до сервера за допомогою REST API, та демона в ОС Linux – ***dockerd. (***REST розшифровується як *Representational State Transfer*. Це був термін, спочатку введений Роєм Філдінгом (Roy Fielding), який також був одним з авторів протоколу HTTP. Особливістю сервісів REST є те, що вони

дозволяють найкращим чином використовувати протокол HTTP.) REST API – це архітектурна надбудова HTTP для роботи клієнта та сервера.

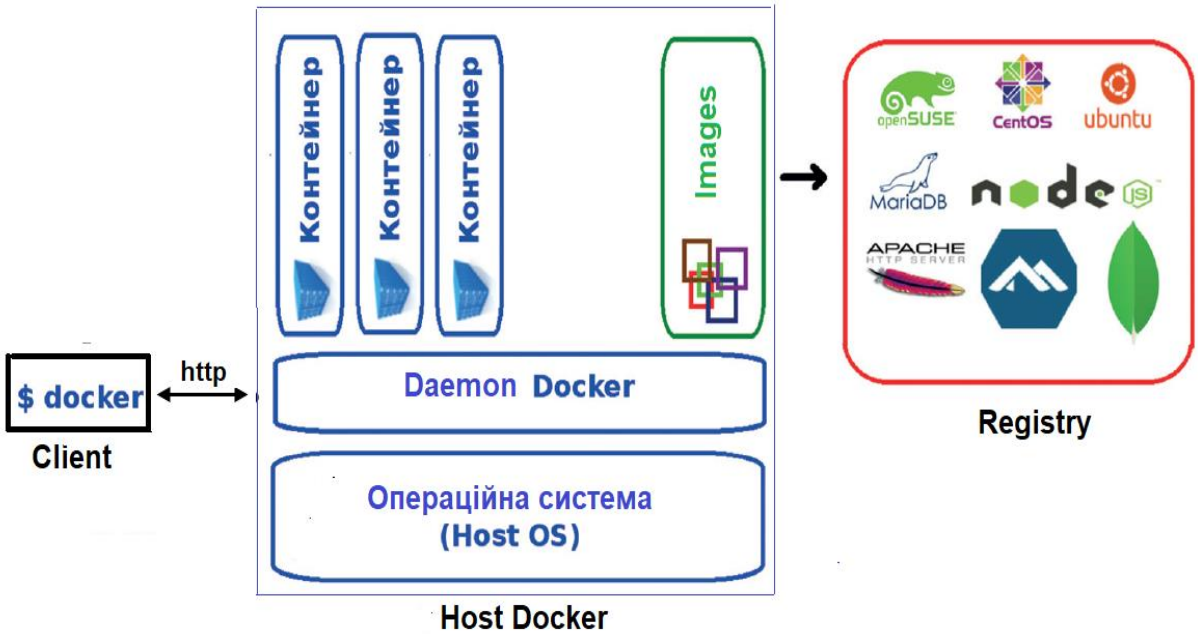


Рис. 6.3. Загальна схема взаємодії основних компонентів Docker [4]

Основними складовими частинами архітектури *Docker* є:

– ***сервер,*** який містить сервіс *Docker*, образи та контейнери; сервіс зв'язується з Registry, образи – метадані додатків, які запускаються в контейнерах *Docker;*

– ***контейнери*** – ізольовані за допомогою технологій ОС користувацькі оточення, в яких виконуються додатки. Контейнер *Docker* запускається з образу додатка. Розробники Docker дотримуються єдиного принципу: один контейнер – це один додаток;

– ***образи (Images)*** – шаблони додатків, які доступні тільки для читання. Поверх існуючих образів можуть додаватися нові рівні образів, які спільно представляють файлову систему, змінюючи або доповнюючи попередній рівень. Зазвичай новий образ створюється або за допомогою зберігання вже запущеного контейнера в новий образ, або за допомогою спеціальних інструкцій для утиліти *Dockerfile*. Для розділення різних

рівнів контейнера на рівні файлової системи можуть використовуватися *UnionFS, aufs, btrfs, vfs, OverlayFS ma Device Mapper*;

– ***REST API, що визначає інтерфейси***, які програми можуть використовувати для спілкування з демоном і вказувати йому що робити;

– ***клієнт інтерфейсу командного рядка (CLI) (команда docker).*** Застосовується для запуску різних дій на сервері *Docker*;

– інтерфейс командного рядка використовує ***API-інтерфейс Docker REST*** для керування або взаємодії з додатками-демонами Docker з використанням базових API і CLI;

– **реєстри (registry)**, використовуються для зберігання образів й містять репозитарії (*repository*) образів, тобто це мережеві сховища образів. Вони можуть бути як приватними, так і загальнодоступними. Найбільш відомими хмарними реєстрами є *Docker Hub* (https://hub.docker.com/search?q=&type=image) та репозитарій *Docker Cloud* (https://cloud.docker.com). *Docker Hub* – найбільш значуща публічна система зберігання образів контейнерів, яка повністю підтримується безліччю розробників та іншими учасниками спільноти, тобто це сервіс для пошуку і спільного використання образів контейнерів. Дозволяє створювати, тестувати, зберігати образи у хмарі Docker.

Слід пам’ятати, що є репозитарії для установки пакетів ОС та репозитарії Docker. Відмінність між *registry* та *repository* полягає у такому.

Registry – це сервіс зберігання і поширення образів (*DockerHub* – це *Registry* за замовчуванням). *Repository* – це набір зв’язаних образів. У них одне і те ж ім’я, але різні мітки.

Наведемо *схему роботи Docker* [4, 17]:

1. Користувач віддає команду за допомогою клієнтського інтерфейсу *Docker*-демона, розгорнутому на *Docker*-хості. Наприклад, скачати готовий образ з реєстру (сховища *Docker*-образів) за допомогою команди *docker*

pull. Взаємодія між клієнтом і демоном забезпечує REST API. Демон може використовувати публічний (*Docker Hub*) або приватний реєстри.

2. Відповідно до команди, заданої клієнтом, демон виконує різні операції з образами на основі інструкцій, прописаних у файлі *Dockerfile*. Наприклад, виконує їх автоматичне збирання за допомогою команди *docker build.*

3. Робота образу в контейнері. Наприклад, запуск *docker-image* за допомогою команди *docker run* або видалення контейнера через команду *docker kill*.

6.1.6. Базові технології Docker

Демон *Docker* використовує «*драйвер виконання (execution driver)»* для створення контейнерів. За замовчуванням використовується власний драйвер *Docker runc,* а також забезпечується підтримка більш старого драйвера для механізму LXC [24].

Для ізоляції контейнерів та забезпечення безпеки в ОС Linux використовують такі стандартні технології ядра Linux (*механізми ядра*), які тісно пов’язані з драйвером *runc*:

– *простори імен* (Linux Namespaces);

– *контрольні групи* (cgroups);

– *засоби керування привілеями* (Linux Capabilities);

– *додаткові мандатні системи забезпечення безпеки* AppArmor або SELinux [25, 26].

Простори імен контролюють доступ до структур даних ядра. Власне, це гарантує ізоляцію процесів один від одного – ізоляцію контейнерів, тобто неперетинання один з одним ієрархії процесів, користувачів, мережевих інтерфейсів – файлова система, ім’я хоста, мережеве середовище та процеси будь-якого контейнера відокремлені від іншої частини системи.

Простори імен, які використовує Docker:

PID, Process ID – ізоляція ієрархії процесів;

User – ізоляція ідентифікаторів користувачів (UID);

NET, Networking – ізоляція мережевих інтерфейсів;

IPC, InterProcess Communication – керування взаємодією між процесами;

MNT, Mount – керування точками монтування;

UTS, Unix Timesharing System – ізоляція ядра ідентифікаторів версії, імені хоста та доменного імені **NIS**.

Механізм контрольних груп (*cgroups*) надає інструмент для керування розподілом, пріоритетами та керуванням системними ресурсами (процесор, оперативна пам'ять тощо), які використовує контейнер. Контрольні групи реалізовані в ядрі Linux. В сучасних дистрибутивах керування контрольними групами реалізовано через *systemd,* між тим забезпечується можливість керування за допомогою бібліотеки *ibcgroup* та утиліти *cgconfig.*

Наведемо ***основні ієрархії контрольних груп*** *(контролери)*:

• **blkio** – задає ліміти на операції «введення – виведення» та на доступ до блокових пристроїв;

• **cpu –** за допомогою планувальника процесів розподіляє процесорний час між задачами;

• **cpuacct** – створює автоматичні звіти ресурсів центрального процесора, працює спільно з *cpu*;

• **cpuset** – закріплює за задачами певні процесори та вузли пам'яті;

• **devices** – регулює доступ задач до певних пристроїв;

• **freezer** – призупиняє або відновлює задачі;

• **memory** – встановлює ліміти та генерує звіти про використання пам'яті задачами контрольної групи;

• **net_cls** – здійснює тегування мережевих пакетів ідентифікатором класу (*classid*). Це дозволяє контролеру трафіка (команда *tc*) та брандмауеру (*iptables*) враховувати ці теги під час обробки трафіка;

• **perf_event** – дозволяє здійснювати моніторинг контрольних груп за допомогою утиліти ***perf***;

• **hugetlb** – дає можливість використовувати віртуальні сторінки пам'яті великого розміру та застосовувати до них ліміти.

Механізм засобів керування привілеями під назвою *Capabilities* дозволяє розбити привілеї користувача *root* на невеликі групи привілеїв та призначати їх окремо. За допомогою опцій команди *Docker* можна дозволяти або забороняти такі дії: операції монтування, доступ до сокетів, змінювати атрибути файлів або власника. У системах мандатного контроля доступу (MAC) (*Security-Enhanced Linux*) на відміну від класичної дискреційної системи контролю доступу (DAC), в якій є три рівня – власник, група-власник та інші щодо розподілу користувачів до ресурсів, права доступу визначаються самою системою за допомогою певних політик. У реалізації *SELinux* політики працюють на рівні системних викликів і застосовуються самим ядром. Ще однією ключовою технологією *Docker* є *файлова система з каскадно-об'єднуючим монтуванням (Union File System – Union FS)*, яка забезпечує зберігання рівнів для контейнерів.

6.1.7. Інструментальні засоби для Docker

Механізму *Docker* та реєстру *Docker Hub* недостатньо для повноцінної роботи з контейнерами, оскільки більшість користувачів потребують сервіси підтримки, допоміжного ПЗ, розширених мережевих можливостей, тому трохи пізніше були розроблені додаткові ***технології підтримки*** – ***інструментальні засоби для Docker***:

• ***Swarm*** – менеджер кластерів; *Swarm* дозволяє згрупувати декілька Docker-хостів і користувач може працювати з цією групою як з єдиним ресурсом, що вирішує задавдання кластеризації;

• ***Docker– Compose*** – інструмент для створення та виконання додатків, скомпонованих з декількох Docker-контейнерів; такі компоновки використовують для розроблення і тестування ПЗ;

• ***Docker Machine*** – утиліта командного рядка для підтримки роботи Docker-хостів; встановлює та конфігурує Docker-хости як на локальних, так і віддалених ресурсах, а також конфігурує клієнта Docker;

• ***Kinematic*** – графічний інтерфейс користувача для роботи з контейнерами в ОС Mac OS та Windows;

• ***Docker Trusted Registry*** – локально встановлюване програмне рішення для зберігання та керування образами Docker, але це єдиний програмний продукт, який на цей момент є закритим.

Переносимість та ізольованість контейнерів сприяють спрощенню співпраці з іншими розробниками та інженерами з експлуатації: розробники можуть бути цілком упевнені в тому, що їх код буде працювати в будь-яких програмних середовищах, а інженери з експлуатації можуть зосередитися на організації та налагодженні роботи контейнерів на хостах, не переймаючись тим, який код виконується всередині контейнерів.

У разі розгортання великої кількості контейнерів важливо мати *інструмент для оркестрації та керування кластером.* Кожний новий контейнер має бути розміщений на певному хості, його треба контролювати, оновлювати, при цьому система повинна правильно реагувати на збої або зміни навантаження при переміщенні, запуску або зупинці контейнерів. Для цього пропонується декілька конкуруючих рішень:

– *Kubernetes* (http://kubernetes.io) від Google;

– Marathon (https://github.com/mesosphere/marathon);

– фреймворк для Mesos (https://mesos.apache.org);

– Fleet (https://github.com/coreos/fleet) від CoreOS;

– власний інструмент Docker Swarm.

6.1.8. Установка Docker

У літературі для автоматичної установки *Docker* пропонується скористатися скриптом, який надається на сайті ***https://get.docker.com***. Офіційні інструкції пропонують виконати команду **curl -sSL | sh** або **wget -q0- | sh** [28].

Але завжди рекомендуються спочатку оновити наявний перелік пакетів: ***sudo apt update***, а потім щось встановлювати.

Якщо працюємо з Ubuntu, то Docker ставиться на версії Ubuntu 18.04 та Ubuntu 20.0 4. Далі встановлюємо необхідні пакети, які дозволяють менеджеру пакетів *apt* використовувати пакети по HTTPS:

sudo apt install apt-transport-https ca-certificates curl software-properties-common

Складові цієї команди означають таке:

– ***apt-transport-https*** дозволяє диспетчеру пакетів передавати файли і дані за протоколом https;

– ***ca-certificates*** дозволяє браузеру і системі перевіряти сертифікати безпеки;

– ***curl*** передає дані;

– ***software-properties-common*** додає скрипти для керування ПЗ.

Потім додаємо у систему ключ GPG (англ. GNU Privacy Guard) офіційного репозиторію Docker:

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add –

Додаємо репозиторій Docker у список джерел пакетів *APT*:

sudo add-apt-repository «deb [arch=amd64] https://download.docker.com/linux/ubuntu bionic stable»

Усі версії *Ubuntu* поділяються на дві категорії: «*звичайні*» і *LTS* (англ. Long-Term *Support*). Термін підтримки «звичайних» версій становить 9 місяців. Для LTS-версій, що випускаються раз на два роки, поновлення пакетів надається протягом п'яти років після релізу. Версія «***ubuntu bionic stable»*** – це *LTS*, яка вийшла у квітні 2018 року, діє до 2028 року, має повну назву ***Bionic Beaver*** (біонічний бобер).

Далі оновлюємо базу даних пакетів інформацією про пакети Docker зі знову доданого сховища: ***sudo apt update.***

Слід переконатися, що Docker встановлено з репозиторію Docker, а не з репозиторію за замовчуванням Ubuntu: ***apt-cache policy docker-ce.***

Для установки буде використаний репозиторій Docker для Ubuntu 20.04.

Далі встановлюємо Docker: ***sudo apt install docker-ce***

docker-ce (Community Edition) – це сертифікований випуск, що надається безпосередньо *docker.com (https://download.docker.com/linux/)*, і його також можна зібрати з вихідного коду

(https://github.com/docker/docker-ce/tree/master/components/packaging/deb).

Docker встановлений, демон запущений, і процес буде запускатися при завантаженні системи. Переконаємося, що процес запущений, тобто сервіс є активним: ***sudo systemctl status docker***

За замовчуванням команду ***docker*** може запустити користувач *root* або користувач із групи *docker*, яка автоматично створюється під час установки *Docker*.

Щоб не вводити *sudo* щоразу під час запуску команди *docker*, додаємо ім'я свого користувача у групу *docker*: ***sudo usermod -aG docker username***

Наприклад, ***sudo usermod -aG docker leo*** свідчить про те, що у групу *docker* додали користувача *leo*. Для застосування цих змін у складі групи необхідно розлогінитися і знову залогінитися на сервері або задати таку команду: ***su – username***

Наприклад, ***su – leo***.

Щоб продовжити роботу, треба ввести пароль користувача.

Щоб переконатися, що користувач доданий у групу *docker*, слід набрати команду ***id -nG***.

6.1.9. Використання команди Docker, опції, команди керування, підкоманди

Детальну інформацію про команди можна отримати на сайті *Docker* у розділі офіційної документації ***http://docs.docker.com***.

Команда *docker* дозволяє використовувати різні опції, команди з аргументами [4, 24]. Синтаксис команди такий:

docker [option] [command] [arguments]

Для перегляду усіх опцій, доступних команд керування та підкоманд, вводять: ***docker***

Розглянемо зміст ***опцій***:

--config string	Розміщення файлів конфігурації клієнта (за замовчуванням *"/home/userbody/.docker*").
-c, --context string	Назва контексту для підключення до демона (замінює DOCKER_HOST env var і контекст, за замовчуванням встановлені з "Використання контексту docker").
-D, --debug	Включити режим налагодження.
-H, --host list	Сокет(и) демона для підключення.
-l, --log-level string	Встановіть рівень реєстрації ("відладка" | "попередження" | "помилка" | "фатальний") (за замовчуванням "інформація"),
--tls	Використовуйте TLS; мається на увазі --tlsverify
--tlscacert string	Сертифікати довіри, підписані лише цим ЦС (за замовчуванням "/Home/userbody/.docker/ca.pem").
--tlscert string	Шлях до файлу сертифіката TLS (за замовчуванням "/home/userbody/.docker/cert.pem").

--tlskey string	Шлях до файлу ключа TLS (за замовчуванням “/Home/userbody/.docker/key.pem”).
--tlsverify	Використовуйте TLS і перевірте видалення.
-v, --version	Інформація про версію для друку та вихід.

***Команди керування*:**

builder	Manage builds,
config	Manage Docker config,
container	Manage containers,
context	Manage context,
engine	Manage the docker engine,
image	Manage images,
network	Manage networks
node	Manage Swarm nodes
plugin	Manage plugins
secret	Manage Docker secrets
service	services
stack	Manage Docker stacks
swarm	Manage Swarm
system	Manage Docker
trust	Manage trust on Docker images
volume	Manage volumes

***Повний список підкоманд Docker*:**

docker attach	«приєднує» локальні стандартні потоки «введення – виведення» та помилок до запущеного контейнера
docker build	збирає образ з *Dockerfile*
docker builder	керування збірками
docker checkpoint	керування контрольними точками
docker commit	створює новий образ зі змін контейнера
docker config	керування *Docker configs*
docker container	керування контейнерами
docker context	керування контекстами збірки
docker cp	копіює файли / папки між контейнером і локальною файловою системою
docker create	створює новий контейнер
docker diff	перевіряє зміни файлів або каталогів у файловій системі контейнера
docker events	отримує події сервера в режимі реального часу

docker exec	запускає команду в запущеному контейнері
docker export	експортує файлову систему контейнера як tar-архів
docker history	виводить історію образу
docker image	керування образами
docker images	виводить список образів
docker import	імпортує вміст з архіву для створення образу файлової системи
docker info	виводить загальну інформацію про систему
docker inspect	виводить детальну інформацію про контейнер
docker kill	завершує один або кілька запущених контейнерів
docker load	завантажує образ з tar-архіву або STDIN
docker login	вхід до реєстру Docker
docker logout	вихід з реєстру Docker
docker logs	отримує логи контейнера
docker network	керування мережами
docker node	керування кластерами Swarm
docker save	охороняє один або кілька образів в tar-архіві (за замовчуванням передається в STDOUT)
docker search	пошук образів в Docker Hub
docker service	керування сервісами
docker stack	керування стеком
docker start	запускає один або кілька зупинених контейнерів
docker stats	виводить реальний потік статистики використання ресурсів контейнера(ів)
docker stop	зупиняє один або декілька запущених контейнерів
docker swarm	керування *Swarm*
docker system	керування системою *Docker*
docker tag	створює тег *КІНЦЕВИЙ_ОБРАЗ*, який посилається на *ПОЧАТКОВИЙ_ОБРАЗ*
docker top	відображаються запущені процеси контейнера
docker unpause	відміняє паузу всіх процесів в одному або декількох контейнерах
docker update	оновлює конфігурацію одного або декількох

	контейнерів
docker version	виводить інформацію про версію *Docker*
docker volume	керування томами
docker wait	блокує до зупинки одного або декілька контейнерів, потім виводить їх код завершення

Для перегляду опцій використання певної команди вводять:

sudo docker docker-subcommand --help

Наприклад, переглянемо список опцій команди ***images***:

docker images --help

Options:

-a, --all	Показати всі зображення (за замовчуванням приховуються проміжні зображення)
--digest	Показати дайджести
-f, --filter	Вихідний фільтр на основі наданих умов
--format string	Друковані зображення за шаблоном Go
--no-trunc	Не обрізати результат
-q, --quit	Показати лише числові ідентифікатори

Офіційну документацію щодо Docker розміщено на сайті https://docs.docker.com/

6.2. Практична робота з Docker

6.2.1. Хостинг для Docker

Хостинг – це платна послуга, яка надає в оренду місце на сервері та його потужності. Хостинг-провайдери – це компанії, які надають таку послугу. Завдання компанії полягає у наданні цілодобового доступу до сайту користувачів. Таким компаніями є хмарні провайдери *Amazon*, *Google* та *Digital Ocean,* які надають певний рівень підтримки *Docker*.

Google Container Engine є найцікавішим варіантом, оскільки створений безпосередньо на основі Kubernetes. Якщо хмарний провайдер не пропонує пряму підтримку *Docker*, зазвичай є можливість надання віртуальних машин, в яких можна запускати Docker-контейнери.

У цій області також працює компанія *Joyent*, яка пропонує власний механізм контейнерів під назвою *Triton* на основі *SmartOS*. Компанія *Joyent* змогла створити загальнодоступний хмарний сервіс, який взаємодіє зі стандартним *Docker*-клієнтом. Реалізація контейнера компанії *Joyent* має високий рівень безпеки, що дозволяє працювати безпосередньо з апаратним забезпеченням, без необхідності розміщення у віртуальній машині, а це може означати більш високу ефективність та суттєве скорочення накладних витрат, особливо з точки зору операцій «введення – виведення». Є ще кілька проектів, які організували PaaS-платформу на основі *Docker, – Deis* (http://deis.io/), *Flynn* (https://flynn.io/) *ma Paz* (http://paz.sh).

6.2.2. Образи, контекст створення образу, реєстр образів

Образи Docker є результатом процесу їх збирання, а контейнери Docker – це виконувані образи [17]. Кожному образу *Docker* відповідає файл, який називають *Dockerfile*. Його ім’я записують саме так – без розширення. Нові образи контейнерів створюють за допомогою файлів *Dockerfile* і команди *docker build,* тому важливо розуміти внутрішнє функціонування команди створення. Для команди *docker build* необхідно: *Dockerfile* і *контекст створення образу (build context)* (який може бути порожнім).

Контекст створення – це набір локальних файлів і каталогів, до яких можна звертатися з інструкцій ADD і / або COPY в *Dockerfile* і які зазвичай визначаються як шлях до потрібного каталогу. Наприклад, у каталозі *test* створюється файл образ *dockerfile*:

docker build -t test/dockerfile

Під час запуску команди *docker build* для створення нового образу вважають, що *Dockerfile* міститься у потоковому робочому каталозі – визначено як крапка «**.**» (це і є контекст створення). Це означає, що усі файли і каталоги, розмішені по зазначеному шляху, формують контекст створення образу і передаються в демон *Docker* як частина процесу створення. Якщо цей файл перебуває в якомусь іншому місці, його розміщення можна вказати з використанням прапорця «**-f**». Контекст вважають невизначеним, якщо заданий тільки URL для Dockerfile або вміст Dockerfile передається з програмного каналу зі стандартного потоку введення (STDIN). У таких випадках контекст створення цього способу вважають порожнім.

Реєстр образу Docker (Docker Registry) являє собою віддалену платформу (сервіс), яку використовують для зберігання та поширення образів *Docker*. У процесі роботи з *Docker* образи завантажуються до реєстру і вивантажуються з нього. *Офіційним загальнодоступним та безкоштовним реєстром* є *Docker Hub.* Можна вивантажувати свої образи в реєстр *Docker Hub*, щоб інші користувачі скачували та використовували їх. Доступ до реєстру *Docker Hub* можна отримати як із командного рядка, так і через сайт. Пошук наявних образів виконують за допомогою спеціальної команди пошуку Docker або безпосередньо на сайті *http://registry.hub.docker.com*.

Для вивантаження в реєстр образу потрібно зареєструватися (створити обліковий запис) у реєстрі (в онлайн-режимі на сайті або за допомогою команди *docker login*).

Є приватні реєстри – Public Cloud (*Docker Hub Registry, Docker Trusted Registry on-cloud, Azure Container Registry, AWS Container Regisry, Google Container Registry, Quay Registry, Other Cloud*).

6.2.3. Команди для роботи з реєстром

Розглянемо команди, які використовують для роботи з реєстрами, у тому числі й із *Docker Hub*. *Docker* зберігає аутентифікаційну інформацію у файлі ***.dockercfg***, розміщеному у домашньому каталозі розробника.

Команда ***docker login*** виконує процедуру реєстрації або входу на заданий сервер реєстру. Якщо сервер не задано, то за замовчуванням передбачається *Docker Hub*. За потреби під час цієї процедури в інтерактивному режимі буде запитуватися додаткова уточнювальна інформація у вигляді аргументів.

Команда ***docker logout*** виконує процедуру виходу з реєстру *Docker*. Якщо сервер не заданий, то за замовчуванням передбачається *Docker Hub*.

Команда ***docker pull*** завантажує заданий образ з реєстру. Реєстр визначається за іменем образу, а за замовчуванням – Docker Hub. Якщо не задане ім’я тега, буде завантажений образ з тегом *latest* (за наявності такого способу в певному реєстрі). Для завантаження всіх образів зі сховищ використовують аргумент «-a».

Команда ***docker push*** вивантажує образ або репозиторій у заданий реєстр. Якщо тега немає, вивантажуються всі образи зазначеного сховища в заданий реєстр, а не тільки образи з тегом *latest*.

Команда ***docker search*** виводить список загальнодоступних репозиторіїв з реєстру Docker Hub, які відповідають заданим шаблоном пошуку.

Репозиторій (repository) – набір взаємопов’язаних образів (зазвичай це різні версії однієї програми або сервісу). Docker-репозиторій дозволяє зберігати одну або кілька версій певного *docker*-образу.

Тег (tag) – алфавітно-цифровий ідентифікатор, який присвоюється образам всередині сховища (наприклад, 14.04 або stable). Образ може мати одну або більше версій (у термінології *docker* версію називають "тег").

Таким чином, можна сказати, що в *docker*-реєстрі (*docker-registry*) є багато *docker*-репозиторіїв, у кожному з яких є одна або кілька різних

версій одного і того ж *docker*-образу (*docker image*). Керування версіями *docker*-образів здійснюється за допомогою тегів.

Рівні образу контейнера

Кожний *Docker*-образ складається з шарів (layers). Кожна інструкція в *Dockerfile* призводить до появи нового рівня (*layer*) образу. *Кожний шар, крім останнього, що міститься поверх усіх інших, призначений тільки для читання*. *Docker* відстежує кожний рівень. Образ може складатися з десятків шарів (границя дорівнює 127). Перевага шарів полягає в тому, що образи можуть спільно використовувати шари. Далі *Docker* об'єднує інформацію з кожного шару і створює шаблон-образ, з якого запускається контейнер, в якому виконуються інструкції з кожного шару, який був включений у цей образ.

Приклад 6.1. *Я*к приклад розглянемо образ *ubuntu:latest*:

```
1  # docker run -ti ubuntu
2  Unable to find image 'ubuntu: latest' locally
3  latest: Pulling from ubuntu
4  9377ad319b00: Downloading [>] 1.08 MB/65.67 MB
5  a82f81f25750: Download complete
6  b207c06aba70: Download complete
7  d55e68e6cc9c: Download complete
```

У процесі завантаження видно *чотири шари*. Ці шари можна побачити з використанням команди *docker images -a*:

```
1  # docker images -a
2  REPOSITORY  TAG       IMAGE ID        CREATED       VIRTUAL  SIZE
3  ubuntu      latest    d55e68e6cc9c    2 weeks ago   187.9 MB
4  <none>      <none>    b207c06aba70    2 weeks ago   187.9 MB
5  <none>      <none>    a82f81f25750    2 weeks ago   187.9 MB
6  <none>      <none>    9377ad319b00    2 weeks ago   187.7 MB
```

Ім'я репозиторію <*none*> і тег <*none*> тут вказують, що він є проміжним шаром для якогось цілого образу.

Знаходимо усі файли й каталоги, створені для цих шарів:

```
01  # IDS=(9377ad319b00 a82f81f25750 b207c06aba70 d55e68e6cc9c)
02  # for i in $ {IDS [@]}; do
03  > echo -e "nChecking layer $in"
```

```
04  > find / -name "*$i*" -exec file {};
05  > done
06
07  Checking layer 9377ad319b00
08
09
/var/lib/docker/graph/9377ad319b00884df249b7820e3cf540b1c4631b3b1ee6998a0f7c3d53962e03: directory
10
/var/lib/docker/devicemapper/metadata/9377ad319b00884df249b7820e3cf540b1c4631b3b1ee6998a0f7c3d5
3962e03: ASCII text, with no line terminators
11
var/lib/docker/devicemapper/mnt/9377ad319b00884df249b7820e3cf540b1c4631b3b1ee6998a0f7c3d53962e
03: directory
12
13  Checking layer a82f81f25750
14
15              /var/lib/docker/graph/a82f81f257507f5cb74e833ff1ae4a6a39dfa654a161f5393f641832872b87d3:
directory
16
/var/lib/docker/devicemapper/metadata/a82f81f257507f5cb74e833ff1ae4a6a39dfa654a161f5393f641832872
b87d3: ASCII text, with no line terminators
17
/var/lib/docker/devicemapper/mnt/a82f81f257507f5cb74e833ff1ae4a6a39dfa654a161f5393f641832872b87d
3: directory
18
19 Checking layer b207c06aba70
20
21            /var/lib/docker/graph/b207c06aba70227e0a2561bb7df20a5fd1310901da98ecc6f4da7dccdc40d961:
directory
22
/var/lib/docker/devicemapper/metadata/b207c06aba70227e0a2561bb7df20a5fd1310901da98ecc6f4da7dccd
c40d961: ASCII text, with no line terminators
23
/var/lib/docker/devicemapper/mnt/b207c06aba70227e0a2561bb7df20a5fd1310901da98ecc6f4da7dccdc40d9
61: directory
24
25  Checking layer d55e68e6cc9c
26
27              /var/lib/docker/graph/d55e68e6cc9c7f78f1c02001e1a5ce76511db044c659e5c0a4275c54473f2869:
directory
28
/var/lib/docker/devicemapper/metadata/d55e68e6cc9c7f78f1c02001e1a5ce76511db044c659e5c0a4275c544
73f2869: ASCII text, with no line terminators
29
/var/lib/docker/devicemapper/mnt/d55e68e6cc9c7f78f1c02001e1a5ce76511db044c659e5c0a4275c54473f28
69: directory
```

Проаналізуємо каталог */var/lib/docker/graph/*, який має назву *graph database. Саме в ньому зберігаються файли шарів образів*, потрібні для отримання повної інформації щодо розуміння рівнів образу контейнера. Кожний каталог відповідає одному з наявних у Docker шарів:

```
1  # ls -l /var/lib/docker/graph/
2  total 20
```

```
3 drwx ------ 2 root root 4096 Dec 26 17:51 9377ad319b00884df249b7820e3cf540b1c4631b3b1ee6998a0f7c3d53962e03
4 drwx ------ 2 root root 4096 Dec 26 17:51 a82f81f257507f5cb74e833ff1ae4a6a39dfa654a161f5393f641832872b87d3
5 drwx ------ 2 root root 4096 Dec 26 17:51 b207c06aba70227e0a2561bb7df20a5fd1310901da98ecc6f4da7dccdc40d961
6 drwx ------ 2 root root 4096 Dec 26 17:51 d55e68e6cc9c7f78f1c02001e1a5ce76511db044c659e5c0a4275c54473f2869
7 drwx ------ 2 root root 4096 Dec 26 17:51 _tmp
```

Розглянемо список доступних образів:

```
1 # docker images
2 REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE
3 ubuntu latest d55e68e6cc9c 2 weeks ago 187.9 MB
```

Перевіримо його каталог:

```
1 # ls -l /var/lib/docker/graph/d55e68e6cc9c7f78f1c02001e1a5ce76511db044c659e5c0a4275c54473f2869/
2 total 12
3 -rw ------- 1 root root 71 Dec 26 17:51 checksum
4 -rw ------- 1 root root 1288 Dec 26 17:51 json
5 -rw ------- 1 root root 1 Dec 26 17:51 layersize
```

Файл *checksum* містить контрольну суму шару:

```
1 # cat /var/lib/docker graph/d55e68e6cc9c7f78f1c02001e1a5ce76511db044c659e5c0a4275c54473f2869/checksum
2 sha256: a3ed95caeb02ffe68cdd9fd84406680ae93d633cb16422d00e8a7c22955b46d4
```

Файл структура json журналів контейнера

Файл *layersize* містить розмір шару. Весь набір рівнів, які формують образ, можна побачити, виконавши команду *docker history*, наприклад: *docker history ubuntu:latest*

Для перегляду всієї інформації про *Docker* використовують таку команду:

docker info

Ця команда відображає загальносистемну інформацію про встановлення Docker, а саме: версія ядра, кількість контейнерів і образів. Аналогічно виконується команда із прапорцем ***--help***: ***docker help***

Виведення інформації про версію клієнта і сервера Docker, а також про версії мови програмування *Go*, яка використовується при компіляції, виконують за допомогою такої команди: ***docker version***

6.2.4. Команди керування контейнерами протягом їх життєвого циклу

Крім команди *docker run*, існують й інші команди *docker*, призначені для *керування контейнерами протягом їх життєвого циклу*.

Команда *attach* дозволяє користувачеві спостерігати або взаємодіяти з основним процесом всередині контейнера *docker attach [OPTIONS] CONTAINER*

Наприклад:

```
$ ID=$(docker run -d debian sh -c "while true; do echo 'tick'; sleep 1; done;")
$ docker attach $ ID
tick
tick
...
```

Комбінація клавіш *Ctrl+C* призначена для виходу з контейнера – вона завершує процес, який спостерігається, і приводить до завершення роботи контейнера.

Команда ***docker create*** створює контейнер із заданого образу, але не запускає його. Аргументи цієї команди в основному ті самі, що й для команди *docker run*. Щоб запустити створений контейнер, потрібно виконати команду *docker start*.

Команда ***docker cp*** дозволяє копіювати файли між файловими системами контейнера і хоста.

Команда ***docker exec*** запускає задану команду всередині контейнера, її можна використовувати для виконання завдань супроводу або як заміну *ssh* (мітка *«--ssh»* для направлення агентського з'єднання за протоколом SSH або ключа у програму-компонувальник, тобто передавання файлів компонувальнику з віддаленого сервера за протоколом SSH) під час входу (реєстрації) в контейнер. Наприклад:

```
$ ID=$(docker run -d debian sh -c "while true; do sleep 1; done;")
$ docker exec $ID echo "Hello"
Hello
$ docker exec -it $ID /bin/bash
```

```
root @ 5c6c32041d68:/# ls
bin   dev  home  lib64  mnt  proc  run  selinux  sys  usr
boot  etc  lib   media  opt  root  sbin  srv     tmp  var
root @ 5c6c32041d68:/# exit
exit
```

Команда ***docker kill*** надсилає сигнал основного процесу (PID=1) у контейнері. За замовчуванням надсилає сигнал SIGKILL, за яким виконується негайне завершення роботи контейнера. Можна передати іншій сигнал за допомогою додаткового аргументу «**-s**». Повертає ідентифікатор контейнера. Наприклад:

```
$ ID=$(docker run -d debian bash -c \
      "trap 'echo caught' SIGTRAP; while true; do sleep 1; done;")
$ docker kill -s SIGTRAP $ID
e33da73c275b56e734a4bbbefc0b41f6ba84967d09ba08314edd860ebd2da86c
$ docker logs $ID
caught
$ docker kill $ID
e33da73c275b56e734a4bbbefc0b41f6ba84967d09ba08314edd860ebd2da86c
```

Команда ***docker pause*** тимчасово призупиняє всі процеси всередині заданого контейнера. Процеси не отримують ніяких сигналів припинення, тобто не можуть бути повністю зупинені й завершені або видалені. Продовжити виконання цих процесів можна за допомогою команди *docker unpause*. Команда *docker pause* використовує внутрішню функціональну можливість призупинення (*freezing*) механізму *cgroups* в ядрі Linux. Ця команда значно різниться від команди *docker stop*, яка повністю зупиняє виконання всіх процесів і надсилає процесам сигнали в явній формі.

Команда ***docker restart*** перезапускає один або кілька контейнерів. Її можна вважати приблизним аналогом виконання для заданих контейнерів команди *docker stop*, за якою одразу виконується команда *docker start*. Є додатковий аргумент «***-t***», що визначає інтервал часу очікування, необхідного для завершення роботи контейнера, перед його зупинкою за сигналом SIGTERM.

Команда ***docker rm*** видаляє один або декілька контейнерів. Повертає імена або ідентифікатори успішно видалених контейнерів. За замовчуванням *docker rm* не видаляє наявні томи. Аргумент «***-f***» дозволяє видаляти працюючі контейнери. За допомогою аргументу «***-v***» можна видалити томи, створені контейнером, видаляється (якщо ці томи не змонтовані на каталоги і не використовуються іншими контейнерами). Наприклад, для видалення всіх зупинених контейнерів потрібно виконати таку команду:

```
$ docker rm $(docker ps -aq)
b7a4e94253b3
e33da73c275b
f47074b60757
```

Команда *docker start* запускає зупинений контейнер (або декілька контейнерів). Може використовуватися для повторного запуску зупиненого контейнера або для запуску контейнера, який був раніше створений командою *docker create*, але ніколи не виконувався.

Команда *docker stop* зупиняє (але не видаляє) один або декілька контейнерів. Після виконання цієї команди заданий контейнер переходить у стан «зупинений» («завершений»). Додатковий аргумент «-t» визначає інтервал часу очікування, необхідний для завершення роботи контейнера, перед його зупинкою за сигналом SIGTERM.

Команда *docker unpause n*ерезапускає контейнер, виконання якого було попередньо призупинено командою *docker pause*.

Інформація про контейнер

Наступні команди надають інформацію про працюючі та зупинені контейнери.

Команда *docker diff* показує зміни у файловій системі контейнера порівняно з файловою системою образу, який був використаний для запуску цього контейнера. Наприклад:

```
$ ID=$(docker run -d debian touch /NEW-FILE)
```

```
$ docker diff $ID
A /NEW-FILE
```

Команда *docker events* виводить у реальному часі події від демона демону. Для виходу з цього режиму використовують комбінацію клавіш Ctrl+C.

Команда *docker inspect* надає докладну інформацію про задані контейнери або образи. В основному це інформація про конфігурації, що включають параметри налаштування мережі та параметри прив'язки томів. Аргумент «**-f**» використовують для визначення шаблонів мови Go при форматуванні та фільтрації виведення.

Команда *docker logs* виводить журнали (журнали) для контейнера. Виводиться все, що було записано в потоки STDERR та STDOUT всередині контейнера.

docker port виводить список відображень відкритих портів для заданого контейнера. Додатково можуть бути задані номери внутрішнього порту контейнера та шуканий протокол. Час використовується після виконання команд *docker run -P <image>* для отримання списку призначених портів. Наприклад*:*

```
$ ID=$(docker run -P -d redis)
$ docker port $ID
6379/tcp -> 0.0.0.0:32768
$ docker port $ID 6379
0.0.0.0:32768
$ docker port $ID 6379/tcp
0.0.0.0:32768
```

Команда *docker ps н*адає загальну інформацію про працюючі контейнери: ім'я, ідентифікатор, стан. У цій команді велика кількість різноманітних аргументів, серед яких найважливішим є «**-a**», що дозволяє викласти **інформацію про всі контейнери**, а не тільки про працюючі в поточний момент. Також слід відзначити аргумент «**-q**», який **повертає лише ідентифікатори контейнерів**, що дуже зручно для організації потоку введення для інших команд, наприклад, *docker rm*.

Команда *docker top* надає інформацію про процеси, що виконуються всередині заданого контейнера. Насправді ця команда запускає утиліту *Unix ps* на хості для розміщення та вибору для виведення процесів, що виконуються у заданому контейнері. Аргументи для цієї команди відповідають аргументам утиліти *ps*, а за замовчуванням встановлені аргументи «**-ef**» (слід пам'ятати про необхідність зберегти поле ідентифікатора процесу PID під час виведення результатів). Наприклад:

```
$ ID=$(docker run -d redis)
$ docker top $ID
UID  PID   PPID  C  STIME  TTY  TIME      CMD
999  9243  1836  0  15:44  ?    00:00:00  redis-server *:6379
$ ps -f -u 999
UID  PID   PPID  C  STIME  TTY TIME      CMD
999  9243  1836  0 15:44  ?   00:00:00 redis-server *:6379
$ docker top $ID -axZ
LABEL          PID   TTY STAT  TIME COMMAND
docker-default 9243  ?    Ssl  0:00 redis-server *:6379
```

6.2.5. Робота з образами

Розглянемо команди, які надають інструменти для створення образів і роботи з ними.

Команда *docker build* створює образ з файлу *Dockerfile.*

Команда *docker commit* створює образ із вказаного контейнера. Ця команда іноді може бути корисною, але у більшості випадків доцільно користуватися інструкцією команди *docker build*, яка з легкістю відтворюється повторно. За замовчуванням контейнери тимчасово призупиняються перед створенням образу, але призупинення можна відмінити за допомогою аргументу *--pause=false*. Для налаштування метаданих застосовують аргументи «-a» та «-m». Наприклад:

```
$ ID=$docker run -d redis touch /new-file)
$ docker commit -a "Joe Bloggs" -m "Comment" $ID commit:test
ac479108b0fa9a02a7fb290a22dacd5e20c867ec512d6813ed42e3517711a0
cf
$ docker images commit
REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE
commit test ac479108b0fa About a minute ago 111 MB
$ docker run commit:test ls /new-file
/new-file
```

Команда *docker export* експортує вміст файлової системи заданого контейнера у вигляді *tar*-архіву, направляючи його у стандартний потік виведення STDOUT. Створений таким чином архів може бути завантажений командою *docker import*. Слід зазначати, що експортується лише файлова система, але усі метадані, такі як оголошені порти, команди CMD, параметри налаштування ENTRYPOINT, будуть втрачені. В експортовану файлову систему не входять будь-які томи (порівняйте з командою *docker save*).

Команда *docker history* виводить інформацію про кожний рівень в образі.

Команда *docker images* виводить список локальних образів, ім'я сховища, ім'я тега, розмір та ін. У стовпці VIRTUAL SIZE відображено загальний сумарний розмір образу з урахуванням усіх рівнів, що розміщені нижче. Оскільки ці рівні можуть спільно використовуватися декількома образами, додавання їх розміру в усі відповідні образи не дозволяє точно оцінити ступінь використання дискового простору. Крім того, образи можуть з'являтися у списку кілька разів, якщо мають більше одного тега; образи можна розрізняти за ідентифікатором. Наприклад, для обмеження виведення інформації у файл використаємо конвейєр «|» для виведення перших чотирьох рядків з файлу (утиліта head -n, де n-кількість рядків):

```
$ docker images  head -4
REPOSITORY                TAG IMAGE   ID                  CREATED       VIRTUAL SIZE
identidock_identidock     latest      9fc66b46a2e6 26     hours ago     839.8 MB
redis                     latest      868be653dea3        6 days ago    110.8 MB
containersol/pres-base    latest      13919d434c95        2 weeks ago   401.8 MB
```

Команда *docker import* створює образ з архівного файлу, що містить файлову систему і створений командою *docker export*. Архів може бути заданий шляхом до файлу або у формі URL, а також переданий через стандартний потік введення STDIN (якщо вказано прапорець «-»). Повертає ідентифікатор нового створеного образу. Образ може бути позначений тегом, що складається з імені сховища та імені тега. Слід

зазначити, що образ, створений командою *docker import*, складається тільки з *одного рівня*, і в ньому немає параметрів конфігурації *Docker*, таких як оголошені порти і значення інструкцій CMD (порівняйте з командою *docker load*). Наведемо приклад «зрощування» образу до одного рівня в результаті операцій експорту та імпорту:

```
$ docker export 35d171091d78 | docker import - flatten:test
5a9bc529af25e2cf6411c6d87442e0805c066b96e561fbd1935122f988086009
$ docker history flatten:test
IMAGE          CREATED   CREATED BY    SIZE        COMMENT
981804b0c2b2   59        seconds ago   317.7 MB    Imported from –
```

Команда *docker load* завантажує репозиторій з *tar*-архіву, переданого через стандартний потік введення STDIN. Репозиторій може містити кілька образів і тегів. На відміну від команди *docker import*, у завантажувальні образи включені метадані та історія. Архівні файли для завантаження створюються командою *docker save*, що робить комбінацію *save/load* цілком життєздатною альтернативою реєстрам для поширення образів і виконання операцій резервного копіювання. Приклад наведено в описі команди *docker save*.

Команда *docker rmi* видаляє заданий образ або кілька образів. Образи визначаються ідентифікатором або комбінацією імен сховища та тега. Якщо вказано тільки ім’я сховища без тега, то передбачається тег *latest*. Для видалення образів, наявних в декількох репозиторіях, необхідно вказати ідентифікатор способу і скористатися аргументом «-f». Таку команду потрібно виконати по одному разу в кожному сховищі. Наприклад, docker rmi my_image.

Можна комбінувати команди для очищення всіх контейнерів і образів:

– зупиняємо всі запущені контейнери: docker kill $(docker ps -q);

– видаляємо все зупинені контейнери: docker rm $(docker ps -a -q);

– видаляємо все образи: docker rmi $(docker images -q).

Команда *docker save* зберігає іменовані образи або репозиторії в *tar*-архів, який передається у стандартний потік виведення STDOUT (для

запису у файл використовують аргумент «-o»). Образи можна задавати за ідентифікаторами або у формі *repository:tag*. Якщо задано тільки ім'я сховища, то в архіві будуть збережені всі образи із цього сховища, а не тільки ті, які мають тег *latest*. Команду *docker save* у поєднанні з командою *docker load* часто використовують у поєднанні з командою *docker load* для поширення або резервного копіювання образів. Наприклад:

```
$ docker save -o /tmp/redis.tar redis:latest
$ docker rmi redis:latest
Untagged: redis:latest
Deleted: 868be653dea3ff6082b043c0f34b95bb180cc82ab14a18d9d6b8e27b7929762c
...
$ docker load -i /tmp/redis.tar
$ docker images redis
REPOSITORY    TAG       IMAGE ID            CREATED VIRTUAL     SIZE
redis         latest    0f3059144681        3 months ago        111 MB
```

Команда *docker tag* пов'язує ім'я сховища та тега із заданим образом. Образ можна вказувати за ідентифікатором або за допомогою імен сховища та тега (якщо тег не вказано, то це *latest*). Якщо для нового імені тег не задано, то за замовчуванням присвоюється тег *latest*. Наприклад:

```
$ docker tag faa2b75ce09a newname
```

Образ з ідентифікатором faa2b75ce09a додається в репозиторій *newname*, присвоюється образу тег *latest* за замовчуванням, оскільки тег не було зазначено:

```
$ docker tag newname:latest amouat/newname
```

Образ *newname:latest* додається в репозиторій amouat/newname, присвоюється за замовчуванням тег *latest*. Цей формат підходить для передавння образів в реєстр *Docker Hub*, якщо операцію виконує користувач *amouat*.

Приклади роботи з Docker

Контейнери *Docker* запускаються з образів *Docker*. За замовчуванням *Docker* отримує образи з хаба *Docker Hub [https://hub.docker.com]*, який

являє собою реєстр образів і підтримується компанією *Docker*. Будь-хто може створити і завантажити свої образи *Docker* у *Docker Hub*, тому для більшості додатків і дистрибутивів *Linux*, які можуть знадобитися для роботи, вже є відповідні образи в *Docker Hub*.

Щоб перевірити правильність встановлення програмної системи (*Ubuntu*), виконують таку команду: ***sudo docker run hello-world***

Якщо отримано коректний результат, маємо

Hello from Docker!
This message shows that your installation appears to be working correctly.

Це означає, що Docker працює правильно.

Наприклад, для встановленої програмної системи ***Debian Linux*** для перевірки правильності встановлення *Docker* слід виконати таку команду:

sudo docker run debian echo "Hello World"

Команда ***docker run*** ініціалізує запуск контейнерів. Аргумент ***debian*** – це ім’я образу, яке буде використовуватися. Під час відпрацювання цієї команди може виводитися повідомлення про те, що локальної версії образу немає (*ubuntu/debian*). Далі *Docker* в онлайн-режимі перевіряє *Docker Hub* та завантажує найновішу версію образу. Після завантаження та перевірки образу *Docker* розміщує образ у працюючий контейнер та виконує задану команду *echo "Hello World"* всередині контейнера. Результат виконання цієї команди наводиться в останньому рядку. Якщо виконати цю команду ще раз, то контейнер зразу запуститься без попереднього завантаження образу, тому що підготував та запустив контейнер, виконав команду *echo* та зупинив контейнер.

Образи, доступні в *Docker Hub*, можна шукати за допомогою команди *docker* і підкоманди *search*. Наприклад, для пошуку образу *Ubuntu* вводимо:

docker search ubuntu

Коли потрібний образ обраний, можна завантажити його на комп'ютер за допомогою підкоманди ***pull***. Щоб завантажити офіційний образ *ubuntu* на комп'ютер, запускають таку команду: ***sudo docker pull ubuntu***

Для перегляду завантажених на комп'ютер образів потрібно ввести:

sudo docker images

За допомогою *Docker* можна запустити команду **оболонку *shell*** *всередині контейнера* наступним чином: ***sudo docker run -it ubuntu***

Командний рядок повинен змінитися, показуючи, що розробник тепер працює в контейнері. Командна оболонка буде мати такий вигляд:

```
Output
root @ d9b100f2f636: / #
```

У командному рядку відображається ідентифікатор контейнера. У цьому прикладі це *d9b100f2f636.* Або для образу *debian*:

```
sudo docker run -i -t debian /bin/bash
root@622ac5689680:/# echo "Hello from Container-land!"
Hello from Container-land!
root@622ac5689680:/# exit exit
```

Прапорці ***-i*** та ***-t*** означають запуск сеансу інтерактивної роботи з *підключенням термінального пристрою tty та запуск командного рядка* ***/bin.*** Команда */bin/bash* ініціалізує командну оболонку *bash (Bourne again shell).* Під час виходу з командної оболонки контейнер припиняє роботу (контейнери працюють, поки існує їх основний процес).

Тепер можна запускати всередині контейнера будь-які команди. Наприклад, оновити базу даних пакета всередині контейнера. Перед командами не потрібно використовувати *sudo*, оскільки ми працюємо всередині контейнера як користувач з привілеями ***root***: ***apt update***.

Також в контейнері можна встановити будь-який додаток, наприклад, Node.js: ***apt install nodejs***.

Ця команда встановлює *Node.js* у контейнер з офіційного репозиторію *Ubuntu*. Щоб вийти з контейнера, вводимо команду ***exit*** або комбінацію клавіш ***Ctrl+C***. Відключитися від контейнера без його зупинки можна за допомогою клавіш ***Ctrl+P, Ctrl+Q***. Цей спосіб працює тільки в разі використання термінального пристрою ***TTY*** (тобто при використанні прапорців ***-i*** та ***-t***).

6.2.6. Приклади керування контейнерами Docker

Через певний час після початку використання *Docker* на машині буде безліч активних (запущених) і неактивних контейнерів. Для перегляду активних контейнерів використовують команду: ***docker ps***.

Для перегляду як активних, так і неактивних контейнерів, запускаємо *docker ps* за допомогою параметра ***-a***: ***ocker ps -a***.

Для перегляду останнього створеного контейнера задаємо параметр ***-l***:

docker ps -l

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
d9b100f2f636 ubuntu "/bin/bash" About an hour ago Exited (0) 10 minutes ago sharp_volhard

Для запуску зупиненого контейнера використовуємо команду ***docker start***, потім вказуємо ідентифікатор контейнера або його ім’я. Запустимо завантажений з *Ubuntu* контейнер з ідентифікатором *d9b100f2f636*:

docker start d9b100f2f636

*Я*кщо ***контейнер*** більше не потрібний, ***видаляємо*** його командою ***docker rm*** із зазначенням або ідентифікатора, або імені контейнера.

Щоб знайти ідентифікатор або ім’я контейнера, пов’язаного з образом *hello-world*, використовують команду *docker ps -a*. Потім контейнер можна видалити:

docker rm festive_williams

Запустити новий контейнер і надати йому ім’я можна за допомогою параметра «***-name***». Параметр «***-rm***» дозволяє створити контейнер, який самостійно видалиться після зупинки.

Базові образи

Для створення власних образів необхідно вибрати один із базових образів як відправний [29]. Тобто можна використовувати наявний образ, об’єднавши з ним свої конфігураційні файли і / або дані. У багатьох випадках такий підхід можна застосувати для широко поширеного прикладного ПЗ, наприклад для СУБД і веб-серверів, для яких доступні готові офіційні образи. Але набагато краще скористатися офіційним образом, ніж намагатися сформувати власний, тому що користувачам пропонується успішний результат роботи людей, які мають солідний досвід організації роботи ПЗ всередині контейнерів. Якщо офіційний образ не придатний для роботи розробника з деякої конкретної причини, то можна спробувати сформулювати цю причину як тему для обговорення у вихідному проекті, й напевно знайдуться користувачі, що мали справу з подібними проблемами або знають, як їх вирішити. Наприклад, виникає потреба у використанні невеликого, але повноцінного дистрибутиву Linux. Якщо справді необхідний граничний мінімалізм, рекомендується звернути увагу на образ *alpine* розміром всього 5 Мб, який містить потужний менеджер пакетів для простої установки додатків та інструментальних засобів. Але якщо потрібний образ з більш широкими можливостями, пропонується скористатися одним з образів *debian*, які за розміром набагато менше за образи найпопулярнішого дистрибутиву *ubuntu*, але пакетна база у них одна й та сама. Для організацій, які постійно використовують конкретний дистрибутив *Linux*, з великою ймовірністю знайдеться відповідний *Docker*-образ. Слід пам’ятати, що базові рівні спільно використовуються різними образами, тому якщо розробник вже працює з образом *ubuntu:14.04* і завантажує зі сховищ *Hub* образ,

заснований на *ubuntu*, то фактично скачуються лише зміни, а не весь образ цілком.

Для створення контейнерів треба вміти працювати з ***Dockerfile*** та ***Docker* Compose**. ***Dockerfile*** – скрипт, який дозволяє автоматизувати процес побудови контейнерів через виконання відповідних команд (дій) у базовому образі (*base)* для формування нового образу. ***Docker Compose*** – це утиліта, яка полегшує збирання і запуск системи, що складається з декількох контейнерів, пов'язаних між собою. Утиліта спрощує організацію процесів контейнерів Docker, включаючи запуск, зупинку й налаштування зв'язків і томів всередині контейнера.

6.2.7. Dockerfile та синтаксис для його створення

Dockerfile – це звичайний текстовий файл, який являє собою сценарій, що складається з послідовності команд і аргументів, необхідних для створення Docker-образу образу. Кожна команда, яка модифікує файлову систему, створює новий шар. Такі сценарії спрощують розгортання і процес підготовки програми до запуску. Спочатку Dockerfile визначає образ, на основі якого буде відбуватися збирання образу. Потім йде ряд методів, команд і аргументів, які створюють новий образ. Вміст *Dockerfile* передається демону *Docker* для збирання образу.

Усі подібні файли починаються з позначення ***FROM*** (базового образу, наприклад, ***FROM ubuntu***), як і процес побудови нового контейнера, далі йдуть різні методи, команди, аргументи або умови, після застосування яких створиться Docker-контейнер.

Розглянемо синтаксис *Dockerfile*. У Докер-файлах міститься два типи основних блоків – *коментарі та команди з аргументами*. Причому для всіх команд передбачається певний порядок. Наведемо типовий приклад синтаксису, де перший рядок є коментарем, а другий – командою:

```
# Print «Hello from User!»
RUN echo «Hello from User!!»
```

Наведемо команди, які надають інструменти для створення образів та роботи з ними. Повна документація щодо команд *Dockerfile* доступна на сайті ***http://docs.docker.com/reference/builder/***

Коментарі в *Dockerfile* позначені символом «#» на початку рядка.

Усі команди в Докер-файлах прийнято вказувати великими літерами, наприклад **RUN**, **CMD** і т. д.

• Команда ***ADD*** бере два аргументи, шлях, звідки можна скопіювати файл, і шлях, куди скопіювати файли у власну файлову систему контейнера. Якщо ж *source* шляхом є *URL* (тобто адреса веб-сторінки), то вся сторінка буде скачана і поміщена в контейнер.

Синтаксис команди: ***ADD [вихідний шлях або URL] [шлях призначення]***

ADD /my_ friend_app /my_ friend_app

• Команда ***CMD*** схожа на команду RUN, її використовують для виконання певних програм, але, на відміну від RUN, цю команду зазвичай застосовують для запуску / ініціації додатків або команд вже після їх установки за допомогою RUN у момент побудови контейнера.

Синтаксис команди: ***CMD %додаток% «аргумент», «аргумент»***, ..

CMD «echo» «Hello from User!»

• Команда ***ENTRYPOINT*** встановлює конкретний додаток за замовчуванням, який використовується щоразу в момент побудови контейнера за допомогою образу. Наприклад, якщо вже встановлено певний додаток всередині образу і треба використовувати цей образ тільки для цього додатка, можна вказати це за допомогою ENTRYPOINT, і кожного разу після створення контейнера з образу додаток буде сприймати команду CMD, наприклад. Тобто не буде потреби вказувати конкретний додаток, необхідно буде тільки вказати аргументи.

Синтаксис команди: ***ENTRYPOINT %додаток% «аргумент»***

Врахуйте, що аргументи опційні – вони можуть бути надані командою CMD або під час створення контейнера: ***ENTRYPOINT echo***

Синтаксис команди спільно з CMD:

CMD «Hello from World!» ENTRYPOINT echo

• Команду ***ENV*** використовують для установки змінних середовища (однієї або багатьох). Ці змінні виглядають таким чином: «*ключ = значення*», і вони доступні всередині контейнера скриптів та різних додатків. Цей функціонал Докера, по суті, дуже сильно збільшує гнучкість щодо різних сценаріїв запуску додатків.

Синтаксис команди: ***ENV %ключ% %значення%***

ENV BASH /bin/bash

• Команду ***EXPOSE*** використовують для прив'язки певного порту для реалізації мережевої зв'язності між процесом всередині контейнера і зовнішнім світом – хостом.

Синтаксис команди: *EXPOSE %номер_порту%*

EXPOSE 8080

• Команда ***FROM*** є однією з найнеобхідніших при створенні Докер-файлу. Вона визначає базовий образ для початку процесу побудови контейнера. Це може бути будь-який образ, зокрема і створені розробником до цього. Якщо вказаний користувачем образ не знайдений на хості, Докер спробує знайти і завантажити його. **Ця команда в Докер-файлі завжди має бути вказана першою**.

Синтаксис команди: ***FROM %назва_образу%*** *Наприклад, FROM centos*

• Команда **MAINTAINER** не є виконуваною, вона визначає значення поля автора образу. Найкраще її вказувати відразу після команди FROM.

Синтаксис команди: ***MAINTAINER %ваше_ім'я%***

MAINTAINER User Networks

• Команда ***RUN*** є ***основною командою*** для виконання команд під час написання Докер-файла. Вона бере команду як аргумент і запускає її з образу. На відміну від CMD, цю команду використовують для побудови образу (можна запустити кілька RUN поспіль, на відміну від CMD).

Синтаксис команди: ***RUN % імя_команди%***

RUN yum install -y wget

• Команду ***USER*** використовують для установки *UID* або імені користувача, яке буде використовуватися в контейнері.

Синтаксис команди: ***USER %ID_користувача%***

USER 751

• Команду ***VOLUME*** використовують для організації доступу контейнера користувача до директорії на хості (те саме, що і монтування директорії).

Синтаксис команди: ***VOLUME [«/ dir_1», «/ dir2» ...]***

VOLUME [«/home»]

• Команда ***WORKDIR*** вказує директорію, з якої буде виконуватися команда *CMD*.

Синтаксис команди: ***WORKDIR /шлях***

WORKDIR ~/

Приклад 6.2. Наведемо приклад Docker-файлу:

Створюємо в командному рядку каталог *test*, в якому створимо *Dockerfile*

```
mkdir test
cd test
touch Dockerfile
```

або в редакторі *nano* `nano Dockerfile`

```
FROM ubuntu
RUN apt-get update
RUN echo «Hello from User!!»
```

За допомогою команди **`docker build`** створюємо образ з *Dockerfile*:

```
docker build -t test/dockerfile .
```

Запускаємо образ:

```
docker run test/dockerfile.
```

Точка «.» визначає як контекст створення образу з поточного робочого каталогу і передається в демон Докера як частина процесу створення.

6.2.8. Docker Compose для керування додатками з декількома контейнерами

Docker Compose – це інструмент для створення та виконання додатків, скомпонованих з декількох *Docker*-контейнерів. Встановити *Docker Compose* можна із сайту ***http://docs.docker.com/compose/install/***. *Docker Compose* призначений для швидкого налаштування та запуску різних варіантів середовищ розробки *Docker*. *Compose* використовує файли *YAML (YAML Ain't Markup Language* – мова серіалізації даних) для зберігання конфігурації груп контейнерів.

Повний опис усіх команд *Compose* є на сайті Docker ***https://docs.docker.com/compose/reference/***.

Найостанніші версії *Docker Compose* також є у сховищах *Docker* у репозитарії *на GitHub*. Команда нижче трохи різниться від команди, яку можна знайти знайдете на сторінці *Releases*. Завдяки використанню прапорця «***-o***» для вказівки файлу виведення замість перенаправлення виведення цей синтаксис дозволяє уникнути помилки відсутності прав доступу, що виникає при використанні *sudo*. Перевіряємо поточну версію, за потреби оновимо її за допомогою такої команди (***curl – утиліта доступу до сервісу)***:

sudo curl -L "https://github.com/docker/compose/releases/download/1.25.5/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose

Після зберігання необхідно налаштувати дозвіл, тобто встановити права доступу (*rwx*) до файлу і зробити його виконуваним (+x):

sudo chmod +x /usr/local/bin/docker-compose

Пересвідчимося, чи встановлення відбулось успішно, за допомогою перевірки версії:

docker-compose --version

У результаті має бути виведена встановлена нами версія:

Output
docker-compose version 1.25.5, build 8a1c60f6

У загальнодоступному реєстрі *Docker, Docker Hu*b, міститься образ *Hello World*, який використовують для демонстрації та тестування. Він демонструє мінімальні параметри конфігурації, необхідні для запуску контейнера за допомогою *Docker Compose*: файл *YAML*, що викликає окремий образ.

Створимо директорію для файлу *YAML* і перейдемо в неї:

mkdir hello-world
cd hello-world

Створимо в цій директорії файл *YAML*:

nano docker-compose.yml

У файл ***docker-compose.yml*** запишемо дані, збережемо його і закриємо текстовий редактор:

my-test:
image: hello-world

Перший рядок файлу *YAML* використовують як частину імені контейнера. Другий рядок вказує, який образ використано для створення контейнера.

Під час запуску команди ***docker-compose up*** вона буде шукати локальний образ за вказаним іменем, тобто *hello-world*. Після цього можна зберегти і закрити файл. Можна вручну переглянути образи в нашій системі за допомогою команди *docker images:* ***docker images***

Далі, перебуваючи в директорії *«~/hello-world»*, запускаємо контейнер, виконуючи команду ***docker-compose up***.

Ця команда аналогічна команді ***docker run***.

Після завантаження образу *docker-compose* створює контейнер, поміщає в нього програму *hello* і запускає її, що, у свою чергу, підтверджує, що встановлення виконане успішно.

Команди для роботи з Docker Compose

Найбільш вживаними командами під час роботи з *Compose* є такі:

- ***up*** – запуск усіх контейнерів, визначених у *Compose*-файлі. Зазвичай для запуску *Compose* у фоновому режимі використовують аргумент *«-d»*;

- ***build*** – перетворення усіх образів, створених з файлів *Dockerfile.* Команда *up* створює образи, якщо раніше їх не було, а команду *build* використовують, якщо треба поновити образи;

- ***ps*** – виводить інформацію про стан контейнерів, якими керує *Compose;*

- ***run*** – одноразовий запуск контейнера виконанням однієї команди, а не як сервісу. Також запускаються усі контейнери, з якими мають бути встановлені з’єднання. Слід враховувати, що ця команда за замовчуванням не створює портів, які визначаються у файлі конфігурації сервісу;

- ***logs*** – виведення журнальних записів для всіх контейнерів *Compose*;

- ***stop*** – зупинка контейнерів без їх видалення;

- ***rm*** – видалення зупинених контейнерів (-v – аргумент, який дозволяє видаляти усі томи, якими керує механізм *Docker*.

Робота починається з виконання команди *docker-compose up* для запуску додатка. Команди *docker-compose logs* та *docker-compose ps* використовують для перевірки стану додатка. Після внесення змін у початковий код необхідно виконати команду *docker-compose build,* далі *docker-compose up -d.* Буде створений новий образ та замінений працюючий контейнер, при цьому бази та кеші під час переходу до нових

версій контейнерів залишаються незмінними. Після завершення сеансу роботи з додатком виконують команду *docker-compose stop* для його зупинки. Для повторного запуску цього ж комплекту контейнерів слід виконати команду *docker-compose start* або *docker-compose start up*, якщо не був змінений код. Для видалення набору контейнерів слід скористатися командою *docker-compose rm.*

Встановлення зв'язку контейнерів із зовнішнім світом

Консольна команда *docker run* створює з *docker-образу* контейнер (тобто створює доступний для запису шар поверх шарів, що містяться в *docker-образі*) і запускає його (виконує зазначену команду). У загальному вигляді це подається так:

docker run [OPTIONS] IMAGE [COMMAND] [ARG ...]

Ця команда має багато опцій, найбільш вживаними є такі:

--detach, -d – за замовчуванням, *docker*-контейнер запускається приєднаним (*attached*) до стандартних потоків «введення – виведення». Параметр -d, --detach дозволяє запускати контейнер у фоні й не виводить на екран вміст потоків «введення – виведення»;

--entrypoint – встановлює або перевизначає використовувану за замовчуванням команду (і параметри) з *docker*-образу;

--env, -e – встановлює змінні оточення у форматі пар *КЛЮЧ = ЗНАЧЕННЯ*. Якщо є список змінних оточення у файлі, тоді слід використовувати опцію ***--env-file***;

--ip – призначає *docker*-контейнеру IP-адресу, наприклад, --ip=10.10.9.75;

--name – призначає *docker*-контейнеру ім'я *(--name my-super-container*);

--publish, -p або ***--publish-all, -P*** – зіставляє порт (порти) docker-контейнера порту (портам) на хост системі. Слід зазначити, що *--publish-all*

відкриє доступ до портів, описаних у *Dockerfile* за допомогою директиви *EXPOSE*;

--rm – автоматично видаляє контейнер після завершення його роботи (у тому числі при завершенні з помилкою);

--tty, -t – призначає псевдо-TTY, підключену до STDIN контейнера. Найчастіше використовується з опцією *--interactive, -i* – наприклад, якщо потрібно підключитися до оболонки (*bash*) всередині *docker*-контейнера для виконання якихось дій;

--volume, -v – монтування томи;

--workdir, -w – встановлює робочу директорію всередині *docker*-контейнера. Наприклад, якщо скопіювати якісь файли в папку *app* контейнера, то цілком розумно буде встановити цей каталог як робочу директорію.

Припустимо, розробник запустив веб-сервер всередині контейнера. Як забезпечити зв'язок сервера із зовнішнім світом? Для цього треба відкрити необхідні порти для загального доступу за допомогою аргументів **-p** або **-P** у команді запуску. Така команда перенаправляє порти хоста в контейнер. Наприклад: $ docker run -d -p 8000:80 nginx

Аргумент ***-p 8000:80*** повідомив механізму *Docker* про необхідність перенаправлення порту 8000 хоста на порт 80 у контейнері. Як альтернатива при використанні аргументу -P механізм Docker має автоматично вибрати вільний порт для перенаправлення з хоста в контейнер.

Наприклад:

```
$ ID=$(docker run -d -P nginx)
$ docker port $ID 80
```

Головна перевага використання аргументу *-P* полягає в усуненні додаткового рівня відповідальності за коректне призначення портів, що особливо важливо при наявності декількох контейнерів з портами,

відкритими для загального доступу. Щоб визначити номери портів, які призначені механізмом Docker, можна виконати команду **docker port**.

З'єднання між контейнерами

Механізм *Docker*-з'єднання (*links*) – найпростіший спосіб забезпечення обміну інформацією між контейнерами на одному хості: передається інформація про IP-адресу та відкриті порти з одного контейнера у другий. З'єднання ініціалізуються за допомогою аргументу ***--link CONTAINER: ALIAS*** у команді ***docker run***, де

– *CONTAINER* – ім'я контейнера-адресата (*link container*), тобто того, з яким встановлюється зв'язок,

– *ALIAS* – локальне ім'я, яке використовується всередині керуючого контейнера для звернення до контейнера-адресата.

Крім того, у разі використання з'єднань *Docker* внутрішнє ім'я та ідентифікатор контейнера-адресата будуть додані у файл */etc/hosts* в керуючому контейнері, що дозволить звертатися з цього імені до контейнера-адресата з керуючого контейнера. Далі *Docker* створює в керуючому контейнері набір змінних середовища, призначених для спрощення діалогу з контейнером-адресатом. Наведемо приклад створення контейнер *Redis* й установки з'єднання з ним:

```
$ docker run -d --name myredis redis
c9148dee046a6fefac48806cd8ec0ce85492b71f25e97aae9a1a75027b1c8423
$ docker run --link myredis:redis debian env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=f015d58d53b5
REDIS_PORT=tcp://172.17.0.22:6379
REDIS_PORT_6379_TCP=tcp://172.17.0.22:6379
REDIS_PORT_6379_TCP_ADDR=172.17.0.22
REDIS_PORT_6379_TCP_PORT=6379
REDIS_PORT_6379_TCP_PROTO=tcp
REDIS_NAME=/distracted_rosalind/redis
REDIS_ENV_REDIS_VERSION=3.0.3
REDIS_ENV_REDIS_DOWNLOAD_URL=http://download.redis.io/releases/redis-
3.0.3.tar.gz
REDIS_ENV_REDIS_DOWNLOAD_SHA1=0e2d7707327986ae652df717059354b358b833
58
```

```
HOME=/root
```

У результаті *Docker* створює змінні середовища із префіксом *REDIS_PORT*, що містять інформацію, необхідну для підтримання зв'язку з контейнером *Redis*. Деякі значення видаються надмірними, оскільки потрібна інформація вже міститься в імені змінної. Також Docker імпортував кілька змінних середовища з контейнера-адресата, їх можна розрізнити за префіксом *REDIS_ENV*. Docker створює змінні середовища із префіксом REDIS_PORT, що містять інформацію, необхідну для підтримання зв'язку з контейнером Redis. Деякі значення здаються надмірними, оскільки потрібна інформація вже міститься в імені змінної. Проте всі змінні та їх значення в будь-якому випадку корисні, хоча б як своєрідна форма документації.

Розглянемо інший приклад:

```
$ sudo docker run -d --name www --link db:db www /usr/sbin/nginx
```

Ця команда запускає контейнер *www* з веб-сервером *nginx* всередині й передає всередину нього інформацію про контейнер з іменем *db*. При цьому відбувається таке:

– у контейнері ***www*** з'являється низка змінних оточення, таких як DB_PORT_8080_TCP_ADDR=172.17.0.82, DB_PORT_8080_TCP_PORT=1234 та DB_PORT_8080_TCP_PROTO=tcp, по три на кожний відкритий порт в контейнері ***db***. Ці змінні можна використовувати у файлах конфігурації та скриптах, щоб зв'язати контейнер ***www*** з ***db***;

– файл */etc/hosts* контейнера ***www*** автоматично оновлюється, і в нього потрапляє інформація про IP-адресу контейнера ***db*** (у цьому прикладі рядок 172.17.0.82 *db*). Це дозволяє використовувати ім'я хоста замість IP-адреси для звернення до контейнера ***db*** з контейнера ***www***.

6.3. Життєвий цикл програмного забезпечення при використанні Docker

Контейнеризація додатків є одним із головних трендів сучасних IT-розробок. Ще раз нагадуємо, що контейнеризація різниться від віртуалізації тим, що віртуалізація абстрагує апаратне забезпечення, тоді як контейнеризація абстрагує і базову ОС. Із практичної точки зору це означає, що можна взяти додаток (або групу додатків) й помістити їх у контейнер (або контейнери), щоб зробити їх модульними, легко переносити, компонувати та при цьому займати значно менше місця за обсягами. Переносимість означає, що можна встановити *Docker Engine* у найрізноманітніших ОС і будь-який функціональний контейнер, написаний кимось, буде працювати на ньому.

Використання Docker у процесі розробки

Docker використовують для створення, тестування і розгортання додатка. За допомогою Docker можна:

– зробити процес налаштування простішим, спростити налаштування на рівні інфраструктури;

– розробникам зосередитися виключно на коді, скорочуючи час розробки і збільшуючи продуктивність;

– посилити можливості налагодження з використанням вбудованих функцій;

– ізолювати додатки;

– поліпшити щільність використання серверів у формі контейнеризації;

– робити швидке розгортання на рівні ОС.

Життєвий цикл ПЗ з використанням контейнера Docker полягає у проходженні таких стадій:

docker create – створення контейнера без його запуску, може бути використана для зберігання і виведення ідентифікатора контейнера для майбутнього використання;

docker run – робота контейнера: створити і запустити контейнер;

docker stop – призупинення роботи контейнера;

docker start – відновлення роботи контейнера: запустити наявний зупинений контейнер;

docker restart – перезапуск контейнера;

docker rm – видалення контейнера;

docker kill – відправити сигнал SIGKILL контейнеру про завершення роботи;

docker attach – підключитися до працюючого контейнера;

docker wait – блокувати команду і чекати, поки контейнер не зупиниться.

Відповідно усі можливі стани контейнера *Docker*:

Created – контейнер створений, але не активний.

Restarting – контейнер у процесі перезапуску.

Running – контейнер працює.

Paused – контейнер призупинений.

Exited – контейнер закінчив свою роботу.

Dead – контейнер, в якому сервіс спробував зупинити роботу, але не зміг.

Традиційне вітання світу

Щоб перевірити, чи є доступ до образів і чи можна завантажувати образи з *Docker Hub*, вводять таку команду:

docker run hello-world

Покажемо коректний результат роботи цієї команди, який означає, що *Docker* працює правильно:

Output

Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
9bb5a5d4561a: Pull complete
Digest:
sha256:3e1764d0f546ceac4565547df2ac4907fe46f007ea229fd7ef2718514bcec35d
Status: Downloaded newer image for hello-world:latest
Hello from Docker!
This message shows that your installation appears to be working correctly.

Життєвий цикл контейнера

На рис. 6.4 наведено життєвий цикл (кінцевий автомат) контейнера. Кружечками зображено стани, жирним виділено консольні команди, а квадратиками показано те, що виконується насправді [4].

Прослідкуємо шлях виконання команди *docker run.* Вона виконує дві дії: створення контейнера і запуск. Далі команда виконується (стан *running*) і можливі варіанти:

– перезапуск контейнера (команда *docker restart*);

– призупинення усіх контейнерів (команда *docker pause*, *стан paused*), а потім відміна призупинення усіх процесів у контейнерах (команда *docker unpause*);

– повне зупинення запущених контейнерів (команда *docker stop*);

– завершення запущених процесів (команда *docker kill*);

– завершення виконання запущеного контейнера через відсутність достатньої пам’яті (*kill by out of memory*);

– завершено процес контейнера (*container process exited)*;

– видалення контейнера (*docker rm).*

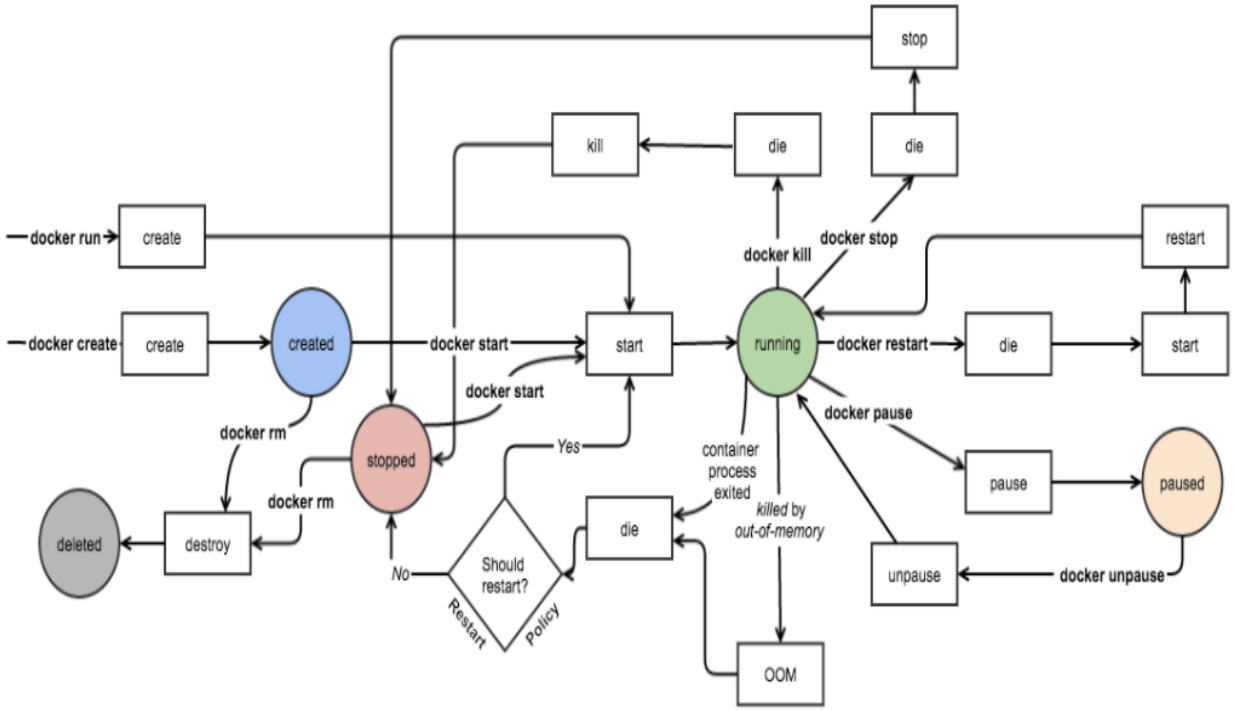


Рис. 6.4. Життєвий цикл контейнера [4]

Автоматизація з використанням Compose

Для спрощення роботи використовують ще одну функцію автоматизації – це *Docker Compose,* яка спрощує організацію процесів контейнерів Docker, включаючи запуск, зупинку і налаштування зв'язків і томів всередині контейнера. Ця утиліта полегшує збирання і запуск системи, що складається з декількох контейнерів, пов'язаних між собою. Інструмент *Compose* (*http://docs.docker.com/compose/*) призначений для швидкого налаштування та запуску різних варіантів середовищ розробки *Docker*. Слід зазначити, що *Compose* використовує файли *YAML* (*YAML Ain't Markup Language*; мова серіалізації даних) для зберігання конфігурації груп контейнерів, які дозволяють розробникам позбавитися від рутинної роботи та витрачати сили на дублювання вже відомих рішень. *Compose* звільнить розробника від необхідності співпрацювати з усіма допоміжними скриптами, призначеними для організації роботи, включаючи запуск, встановлення з'єднань, оновлення і зупинку контейнерів. Встановити *Compose* можна з веб-сайту Docker (*http://docs.docker.com/compose/install/*). Інший варіант – встановити *Docker*

Compose можна зі сховищ *Docker* на *GitHub*. Завдяки використанню прапорця «***-o***» для вказівки файлу виведення замість перенаправлення виведення цей синтаксис дозволяє уникнути помилки відсутності прав доступу, що виникає при використанні *sudo.*

Порядок роботи Compose

Під час роботи із *Compose* найбільш часто використовують такі команди:

up – запуск усіх контейнерів, визначених у *Compose*-файлі. Виведення журнальних записів об'єднується в один потік. У більшості випадків використовують аргумент «***-d***» для запуску *Compose* у фоновому режимі;

build – перетворення всіх образів, створених з файлів *Dockerfile*. Команда *up* буде створювати образи, тільки якщо існували їх не було раніше, тому команду *build* слід використовувати, якщо треба поновити образи;

ps – виведення інформації про стан контейнерів, які керовані *Compose;*

run – одноразовий запуск контейнера з виконанням однієї команди (не як сервісу). Також запускаються всі контейнери, з якими мають бути встановлені з'єднання, якщо не заданий аргумент «***--no-deps***». (Команди, що передаються через *run*, замінюють команди, певні у файлі конфігурації сервису. Крім того, за замовчуванням команда *run* не створює портів, визначених у файлі конфігурації сервісу.);

logs – виведення журнальних записів із кольоровим підсвічуванням, об'єднаний для всіх контейнерів, керованих *Compose*;

stop – зупинка контейнерів без їх видалення;

rm – видалення зупинених контейнерів. Слід пам'ятати про те, що аргумент «***-v***» дозволяє видалити всі томи, керовані механізмом *Docker*.

Порядок роботи починається з виконання команди для запуску програми:

docker-compose up -d

Команди ***docker-compose logs*** та ***ps*** можна використовувати для перевірки стану програми і як допоміжний засіб при налагодженні.

Після внесення змін до початкового коду потрібно виконати команду

docker-compose build,

а потім

docker-compose up -d.

При цьому буде створено новий образ і замінений працюючий контейнер. Зазначимо, що *Compose* зберігає всі колишні томи зі старих контейнерів, таким чином, бази даних і кеші залишаються незмінними у разі переходу до нових версій контейнерів (це може призвести до безладу, тому слід бути обережним із заміною контейнерів). Якщо створення нового образу не потрібне, але внесені зміни в *Compose* YAML-файл, то треба виконати команду ***docker-compose up -d***, щоб замінити контейнер на такий самий, але з новими налаштуваннями. Якщо потрібно примусово зупинити весь механізм *Compose* і заново створити всі контейнери, то використовують прапорець «***--force-recreate***».

Після завершення сеансу роботи з додатком виконують команду для його зупинки:

docker-compose stop.

Той самий комплект контейнерів буде повторно запущений при виконанні команди ***docker-compose start або up***, якщо не був змінений вихідний код. Для остаточного видалення набору контейнерів використовують команду ***docker-compose rm***.

Детальний опис усіх команд *Compose* є на сторінці довідкового керівництва сайту Docker – ***https://docs.docker.com/compose/reference/***.

Установка Docker Compose в Ubuntu 20.04.2.0

Docker Compose встановлюється саме на версію *Ubuntu 18.04* або 20.04.2.0.

Перевіряємо поточну версію, за потреби оновимо її за допомогою такої команди (***curl – утиліта доступу до сервісу***):

sudo curl -L "https://github.com/docker/compose/releases/download/1.25.5/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose

Встановимо дозвіл (+x – на виконання /rwx), тобто зробимо файл виконуваним: **sudo chmod +x /usr/local/bin/docker-compose**

Перевіримо, чи установка була успішною, за допомогою перевірки версії: **docker-compose –version**. У результаті має бути виведена встановлена користувачем версія:

```
Output
docker-compose version 1.25.5, build 8a1c60f6
```

Після встановлення Docker Compose можемо запустити приклад «Hello World».

Запуск контейнера за допомогою Docker Compose

У загальнодоступному реєстрі *Docker*, *Docker Hub*, міститься образ *Hello World*, який використовують для демонстрації та тестування. Він демонструє мінімальні параметри конфігурації, необхідні для запуску контейнера за допомогою *Docker Compose*: файл *YAML*, що викликає окремий образ.

Створимо директорію для файлу *YAML* і перейдемо в неї:

mkdir hello-world cd hello-world

Створимо у цій директорії файл *YAML:* ***nano docker-compose.yml***

Помістимо у файл ***docker-compose.yml*** наступні дані, збережемо його і закриємо текстовий редактор:

my-test:
image: hello-world

Перший рядок файлу *YAML* використовують як частину імені контейнера. Другий рядок вказує, який образ використовують для створення контейнера. Після запуску команди ***docker-compose up*** вона буде шукати локальний образ за вказаним іменем, тобто *hello-world*. Після цього можна зберегти і закрити файл. Можна вручну переглянути образи у системі за допомогою команди *docker images*: ***docker images.*** Коли локальних образів немає, будуть відображені тільки заголовки стовпців:

Output
REPOSITORY TAG IMAGE ID CREATED SIZE

Далі, перебуваючи в директорії *«~/hello-world*», виконаємо команду ***docker-compose up***. При першому запуску команди, якщо локального образу з іменем *hello-world* немає, *Docker Compose* буде завантажувати його з відкритого сховища *Docker Hub*:

Output
Pulling my-test (hello-world: latest) ...
latest: Pulling from library / hello-world
c04b14da8d14: Downloading
[== ====>]
c04b14da8d14: Extracting [==================================== ==========>] c04b14da8d14: Extracting [================================ ==============>]
c04b14da8d14: Pull complete
Digest: sha256:
0256e8a36e2070f7bf2d0b0763dbabdd67798512411de4cdcf9431a1feb60fd9
Status: Downloaded newer image for hello-world: latest
. . .

Після завантаження образу *docker-compose* створює контейнер, поміщає в нього і запускає програму *hello*, що, у свою чергу, підтверджує, що установка виконана успішно:

Output
. . .
Creating helloworld_my-test_1...
Attaching to helloworld_my-test_1

my-test_1 |
my-test_1 | Hello from Docker.
my-test_1 | This message shows that your installation appears to be working correctly.
my-test_1 |
. . .

Далі програма відображає пояснення того, що вона зробила:

Output of docker-compose up
1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
3. The Docker daemon created a new container from that image which runs the executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.

Контейнери *Docker* продовжують працювати, поки команда залишається активною, тому після завершення роботи *hello* контейнер зупиняється. Отже, коли коли користувач буде переглядати активні процеси, заголовки стовпців будуть з'являтися, але контейнер hello-world НЕ буде з'являтися у списку, оскільки він не запущений:

docker ps
Output
```
CONTAINER ID    IMAGE    COMMAND    CREATED    STATUS    PORTS    NAMES
```

Переглянемо інформацію контейнера, яка буде потрібна, на наступному кроці, використовуючи ключ «***-a***», за допомогою якого можна відобразити всі контейнери, а не тільки активні: ***docker ps -a***

Output
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES PORTS NAMES
06069fd5ca23 hello-world "/hello" 35 minutes ago Exited (0) 35 minutes ago drunk_payne

Можна отримати інформацію, яка буде потрібна для видалення контейнера, після закінчення роботи з ним. За допомогою команди ***docker rm***, після якої записується CONTAINER ID або NAME вашого контейнера видаляємо контейнер: ***docker rm 06069fd5ca23***. Після видалення всіх

контейнерів, які містять образ, можна видалити образ: ***docker rmi hello-world***

Створення простого веб-застосування

Після встановлення *Docker Client* з'являється доступ до інструменту командного рядка, який дозволяє взаємодіяти з нашими контейнерами. На практиці багато системних адміністраторів використовують *Nginx (Nginx [engine x] – це HTTP та зворотний проксі-сервер, поштовий проксі-сервер й універсальний TCP-проксі-сервер)* для обслуговування веб-контенту, починаючи з веб-сайтів із плоскими файлами і закінчуючи API-інтерфейсами в *NodeJS*. Тому розглянемо, як створити основну веб-сторінку і зосередимося на налагодженні *Nginx* за допомогою контейнера *Docker*.

Потрібно отримати образ *Nginx Docker*. Для цього виконаємо команду

sudo docker pull nginx,

яка завантажує всі необхідні компоненти для контейнера.

Docker підтримує сайт *Dockerhub*, загальнодоступний репозиторій файлів *Docker* (включаючи як офіційні, так і образи користувача). Завантажений образ є офіційним *Nginx*, він позбавляє від необхідності створювати нам власний образ.

Запустимо і створимо наш контейнер *Nginx Docker*:

sudo docker run --name docker-nginx -p 80:80 nginx

• *run* – команда для створення нового контейнера.

• *Прапорець «--name*» – вказуємо ім'я контейнера.

• «*-p*» визначає порт, поданий у форматі

«-p local-machine-port:internal-container-port».

У цьому випадку зіставляємо порт 80 у контейнері з портом 80 на сервері.

• *nginx* – це ім’я образу на *Dockerhub* (ми завантажували його раніше за допомогою команди *pull*, але *Docker* зробить це автоматично, якщо образ відсутній).

Щоб підняти *Nginx,* слід вставити IP-адресу сервера у веб-браузер, тоді побачимо сторінку «Добро пожаловать в nginx!». Щоб повернутися до нашого сеансу оболонки, натискаємо комбінацію клавіш «CTRL+C». Якщо ви спробуєте завантажити сторінку одразу, отримаєте сторінку «Отказано в соединении». Це тому, що ми закрили наш контейнер. Ми можемо перевірити це за допомогою команди ***sudo docker ps -a*** . Отримаємо:

```
OutputCONTAINER ID  IMAGE    COMMAND                  CREATED             STATUS                       PORTS
NAMES
05012ab02ca1        nginx    "nginx -g 'daemon off"   57 seconds ago      Exited (0) 47 seconds ago
docker-nginx
```

Тобто наш контейнер *Docker* вийшов. *Nginx* не буде дуже корисним, якщо потрібно приєднатися до образу контейнера, щоб він працював, тому на наступному кроці нам треба від’єднати контейнер, щоб він міг працювати незалежно. Видаляємо наявний контейнер «*docker-nginx*»: ***sudo docker rm docker-nginx***. Тепер нам треба запустити його в автономному режимі. Для цього створюємо новий окремий *Nginx*-контейнер за допомогою такої команди:

sudo docker run --name docker-nginx -p 80:80 -d nginx

У цій команді додано прапорець «***-d***» для запуску цього контейнера у фоновому режимі. Вихідні дані мають бути просто ідентифікатором нового контейнера. Запустимо команду list: ***sudo docker ps***

```
OutputCONTAINER ID  IMAGE    COMMAND                  CREATED             STATUS              PORTS
NAMES
b91f3ce26553        nginx    "nginx -g 'daemon off"   About a minute ago  Up About a minute   0.0.0.0:80->80/tcp,
443/tcp   docker-nginx
```

Тобто замість «*Exited (0) X хвилин назад*» тепер у нас є «*Up близько хвилини*», і тепер можна бачити відображення портів. Якщо знову перейти на IP-адресу нашого сервера у нашому браузері, зможемо знову побачити

сторінку «*Ласкаво просимо в nginx!*». Наразі контейнер працює у фоновому режимі, тому що вказано прапорець «***-d***», який вказує *Docker* запускати цей контейнер в окремому режимі. Тепер маємо запущений екземпляр *Nginx* в окремому контейнері. Нам необхідно створити сторінку індексу, яка налаштовується, для вашого сайту. Це налаштування дозволяє мати постійний контент веб-сайту, який розміщений поза (тимчасового) контейнера. Створимо новий каталог для вмісту нашого веб-сайту в нашому домашньому каталозі й перейдемо до нього, виконавши команди, наведені нижче:

mkdir -p ~/docker-nginx/html
cd ~/docker-nginx/html

Тепер створимо *HTML*-файл (наводимо команди для редактора *Vim*, але можна використовувати будь-який текстовий редактор):

vim index.html

Увійдемо в режим вставки, натиснувши «i». Вставляємо вміст, як показано нижче (або додаємо власну HTML-розмітку):

```
<html>
 <head>
  <link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.5/css/bootstrap.min.css"
rel="stylesheet" integrity="sha256-MfvZlkHCEqatNoGiOXveE8FIwMzZg4W85qfrfIFBfYc= sha512-
dTfge/zgoMYpP7QbHy4gWMEGsbsdZeCXz7irItjcC3sPUFtf0kuFbDz/ixG7ArTxmDjLXDmezHubeNikyKG
VyQ==" crossorigin="anonymous">
  <title>Docker nginx Tutorial</title>
 </head>
 <body>
  <div class="container">
   <h1>Hello Digital Ocean</h1>
   <p>This nginx page is brought to you by Docker and Digital Ocean</p>
  </div>
 </body>
</html>
```

Це базова веб-сторінка. Додано тег «<link>», який вказує на CDN для *Bootstrap* (CSS-фреймворк, який дає вашій веб-сторінці набір адаптивних стилів). Зберігаємо цей файл, натиснувши «ESC», а потім «: wq» та «ENTER»:

• write («w») означає для Vim записати зміни у файл;

• quit (« q») означає для Vim вийти з редактора.

Тепер маємо просту сторінку індексу, щоб замінити цільову сторінку *Nginx* за замовчуванням. Наступний крок – це *зв'язування контейнера з локальною файловою системою*, потрібно зібрати все разом. Запускаємо наш контейнер *Nginx*, щоб він був доступний через Інтернет через порт 80, підключаємо його до вмісту нашого веб-сайту на сервері. *Docker* дозволяє пов'язувати каталоги локальної файлової системи нашої віртуальної машини з нашими контейнерами. Контейнер *Nginx* за замовчуванням налаштований на пошук індексної сторінки в *«/usr/share/nginx/html*», тому в нашому новому контейнері *Docker* маємо надати йому доступ до наших файлів у цьому місці. Оформлюємо посилання. Для цього використовуємо прапорець «***-v***» щоб зіставити папку з нашого локального комп'ютера (***«~/docker-nginx/html»***) з відносним шляхом у контейнері (***«/usr/share/nginx/.HTML»***).

Можна зробити це, виконавши таку команду:

sudo docker run --name docker-nginx -p 80:80 -d -v ~/docker-nginx/html:/usr/share/nginx/html nginx

Очевидно, що нове доповнення до команди «***-v***»:

– ***«~/docker-nginx/html:/usr/share/nginx/html»*** і є нашим посиланням на том;

– «-v» вказує, що ми пов'язуємо том;

– «~/docker-nginx/html» – розміщення вашого файлу/каталогу на вашій віртуальній машині;

– «/usr/share/nginx/html» – розміщення, на яке ми посилаємося в нашому контейнері. Після виконання цієї команди, якщо вказати тепер свій браузер на IP-адресу свого сервера (*DigitalOcean Droplet – DigitalOcean є хмарним сервісом-хостингом, який надає послуги щодо оренди серверів, Droplet -це і віртуальний сервер з високою продуктивністю за мінімальну плату 5$*), можна побачити перший

заголовок * Hello Digital Ocean * (або будь-яку іншу веб-сторінку, створену в HTML-файлі).

Якщо немає потреби в інших налаштуваннях *Nginx* за замовчуванням, все готово. Можна завантажувати більше контенту в каталог *«~/docker-nginx/html/»*, і він буде доданий на ваш живий сайт.

Наприклад, якщо змінити наш індексний файл і перезавантажити вікно нашого браузера, можна побачити його оновлення в режимі реального часу. Таким чином, є можливість побудувати цілий сайт із плоских HTML-файлів, якщо захотіти. Наприклад, якщо додати сторінку «*about.html*», можна було б отримати до неї доступ за адресою «*http:///about.html*» без необхідності взаємодії з контейнером.

Приклад роботи Docker з БД Mongo

MongoDB – найбільш популярна нереляційна база даних. Створимо порожній файл і відкриємо його за допомогою редактора *nano*:

nano Dockerfile

Надалі як коментар запишемо, для чого цей *Dockerfile* буде використовуватися, це може бути корисно для наступної роботи. Усі коментарі починаються із символу #:

```
########
# Dockerfile to build MongoDB container images
# Based on Ubuntu
########
# Set the base image to Ubuntu – вказуємо базовий образ Ubuntu
FROM ubuntu

# Update the repository sources list and install gnupg2 – Оновіть список пакетів
# репозитарію та встановіть gnupg2 (вільна програма для шифрування
# інформації і створення електронних цифрових підписів):

RUN apt-get update && apt-get install -y gnupg2
```

 # Add the package verification key – Додайте ключ підтвердження пакета
RUN apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10

 # Add MongoDB to the repository sources list
 # – Додайте MongoDB до списку джерел репозитарію
RUN echo 'deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart dist 10gen' > tee /etc/apt/sources.list.d/mongodb.list

 # Update the repository sources list – Оновіть список джерел репозитарію
RUN apt-get update

 # Install MongoDB package (.deb) – Встановіть пакет MongoDB (.deb)
RUN apt-get install -y mongodb

 # Create the default data directory – Створіть каталог даних за замовчуванням
RUN mkdir -p /data/db

 # Expose the default port – Встановіть порт 27017 за замовчуванням для реалізації
 # процесу на хост машині
EXPOSE 27017

 # Default port to execute the entrypoint (MongoDB)
 # – За замовчуванням порт для виконання точки входу (MongoDB)
CMD ["--port 27017"]

 # Set default container command
 # – Встановіть команду контейнера за замовчуванням
ENTRYPOINT usr/bin/mongodb

Запуск контейнера Docker

Створимо наш перший *MongoDB* образ за допомогою Docker! ***sudo docker build -t user_mongodb .***

-t та ім’я тут використовують для присвоювання тега образу, а *точка в кінці* команди **«.»** означає, що *Dockerfile* перебуває в тому ж поточному робочому каталозі, в якому виконується команда.

Для виведення всіх можливих ключів введіть вводимо

sudo docker build -help

Запускаємо новий MongoDB у контейнері!

sudo docker run -name UserMongoDB -t -i user_mongodb

Ключ ***-name*** використовують для *присвоєння простого імені контейнеру*, інакше це буде досить довга цифро-буквена комбінація. Після запуску контейнера для того, щоб повернутися в систему хоста, треба натиснути ***CTRL+P,*** а потім ***CTRL+Q***.

6.4. Мікросервіси та контейнери

6.4.1. Характеристика мікросервісів

Будь-яка технологія завжди призначена для вирішення конкретних завдань. Зі збільшенням обсягу коду і функціональності програмного продукту виникає необхідність керування складністю програми. Добре продумана архітектура і правильне розбиття програми на модулі допомагають справлятися з поставленим завданням. Варіантом реалізації архітектури може бути монолітний додаток, коли вся або більша частина бізнес-завдань має одну кодову базу. Однак актуальним рішенням у наш час є додаток, побудований на мікросервісах, в якому загальна бізнес-задача розбита на окремі частини, кожна з яких має окремий додаток (мікросервіс) із власною кодовою базою.

Додатки з монолітною архітектурою. У класичному підході до розроблення ПЗ, як правило, використовують монолітну архітектуру додатка, коли його розробляють як єдиний блок. Під час розробки промислових додатків найчастіше використовують трирівневу клієнт-серверну архітектуру. За такого підходу додаток складається із трьох основних частин:

– клієнтська частина (тонкий клієнт, який надає весь функціонал через використання *HTML*-сторінок у браузері, написаних із застосуванням різних *Javascript*-бібліотек або фреймворків);

– бази даних (БД), яка складається з безлічі таблиць в єдиній, як правило, реляційній системі керування базами даних (СКБД);

– серверної частини. Серверна частина програми обробляє *HTTP*-запити, виконує бізнес-логіку, отримує і змінює дані в БД, а також вибирає необхідні *HTML*-сторінки і дані для відображення, надсилає їх як *HTTP*-відповідь клієнту (браузеру). Ця серверна частина додатка являє собою монолітну структуру, і будь-які зміни в ній вимагають створення і розгортання саме нової версії програми.

Монолітні сервери є природним підходом до побудови такого роду систем. На рис. 6.5 зображено структурну схему монолітного блоку [5]. У монолітній архітектурі всі компоненти об’єднуються в єдиний модуль.

Рис. 6.5. Структурна схема монолітного додатка [5]

Але такий підхід передбачає виконання декількох умов: вимоги не повинні змінюватися, команда розробників має залишатися єдиною на весь час розробки, ніякого супроводу системи не передбачається. Цей підхід доцільно використовувати для швидкої апробації технічних рішень і для створення невеликих утиліт. Однак з точки зору замовника навіть якщо система буде створена і впроваджена у зазначений термін, буде відповідати всім його бажанням, але реалії такі, що предметна область

змінюється, як саме життя, у компанії завжди відбувається плинність кадрів, а нові кадри не завжди розуміють, що робити із заплутаним кодом системи й супроводжувати монолітну систему стає дуже дорого, простіше зробити нову. У 2010 році з'явився мікросервісний підхід при проектуванні ПЗ, який став тенденцією в останні роки, оскільки дедалі більше підприємств стають гнучкими і переходять на DevOps.

DevOps (development + operations) – методологія активної взаємодії фахівців з розробки із фахівцями з інформаційно-технологічного обслуговування і взаємна інтеграція їх робочих процесів один в одного для забезпечення якості продукту, тобто ***це розробка, тестування та експлуатація ПЗ як єдиного циклу***. Призначена вона для ефективної організації створення та оновлення програмних продуктів і послуг. Заснована на ідеї тісного взаємозв'язку створення продукту та експлуатації ПЗ, яка прищеплюється команді як культура створення продукту.

MICROSERVICES – це шаблон сервіс-орієнтованої архітектури, в якому додатки створюються як сукупність різних найменших незалежних сервісних одиниць (сервісів). Це програмний підхід, який фокусується на розкладанні додатка на однофункціональні модулі з чітко визначеними API-інтерфейсами. Ці модулі можуть бути розгорнуті незалежно та експлуатуватися невеликими групами фахівців.

Термін «мікро» належить до розміру мікросервіса, яким має керувати одна команда розробників – від 5 до 10. Ці сервіси розробляються з урахуванням бізнес-вимог і незалежно розгортаються за допомогою повністю автоматизованих механізмів розгортання.

Автор *Microservices* ***Лукас Краус*** сказав: «Мікросервіси важливі тому, що вони додають унікальну споживчу цінність завдяки спрощенню систем. Розбиваючи вашу систему або додаток на безліч більш дрібних частин, ви реалізуєте *спосіб зменшення дублювання, підвищення узгодженості та зменшення зв'язку між частинами,* що робить ваші загальні частини

системи більш зрозумілими, більш масштабованими і більш легкими для зміни».

Переваги мікросервісів полягають у такому [7]:

• *Легко розробляти, тестувати і розгортати.*

Найбільша перевага мікросервісів перед іншими архітектурами полягає в тому, що невеликі окремі сервіси можна створювати, тестувати і розгортати *незалежно*. Оскільки одиниця розгортання невелика, це полегшує і прискорює розробку і реліз (остання версія ПЗ). Крім того, реліз однієї одиниці не обмежений випуском іншої, яка ще не завершена, а ризики розгортання знижуються в міру того, як розробники оновлюють частини ПЗ, а не цілий додаток.

• *Підвищена гнучкість*.

За допомогою мікросервісов декілька команд можуть працювати над своїми сервісами незалежно і швидко. Кожна окрема частина програми може бути побудована незалежно через ізольованість компонентів мікросервісів. Наприклад, у команді може бути з 100 людей, які працюють над усім додатком (як у монолітному підході), або може бути 10 команд із 10 осіб, що розробляють різні сервіси. Підвищена гнучкість дозволяє розробникам оновлювати компоненти системи, не відключаючи додаток. Крім того, гнучкість забезпечує більш безпечний процес розгортання і підвищений час безвідмовної роботи. Якщо треба, можуть бути додані нові функції ще до запуску всього додатка.

• *Можливість масштабування по горизонталі*.

Масштабованість означає працювати під великим навантаженням пропорційно додатковим ресурсам. Є декілька видів масштабування – горизонтальне, вертикальне, параметричне, функціональне. Горизонтальне масштабування (створення більшої кількості сервісів в одному пулі) не обмежене і може працювати з мікросервісами динамічно та бути повністю автоматизоване.

Вертикальне масштабування (з використанням того ж ПЗ, але на більш потужних машинах) може бути обмежена пропускною здатністю кожного сервісу.

Недоліки мікросервісів такі:

• *Складність налаштування масштабування.*

Це ключовий недолік. Поділ програми на незалежні мікросервіси тягне за собою більше складових керування. Цей тип архітектури вимагає ретельного планування, величезних зусиль, командних ресурсів і навичок. Причини високої складності такі:

– підвищений попит на автоматизацію, оскільки кожний сервіс має бути перевіреним і підконтрольним;

– доступні інструменти не працюють із сервісними залежностями;

– узгодженість даних і керування транзакціями стають складнішими, тому що кожний сервіс має базу даних.

• *Проблеми безпеки.*

У мікросервісному додатку кожний функціонал, який взаємодіє через API ззовні, збільшує ймовірність атак. Ці атаки можуть відбутися тільки в тому випадку, якщо під час створення програми не будуть виконані належні заходи безпеки.

• *Різні мови програмування.*

Можливість вибирати різні мови програмування є, по суті, палицею із двома кінцями. Використання різних мов ускладнює розгортання. Крім того, важче перемикати програмістів між етапами розробки, коли кожний сервіс написаний іншою мовою.

Мікросервісна архітектура базується на декількох принципах:

– моделювання навколо бізнес-концепцій;

– культура автоматизації;

– приховані подробиці внутрішньої реалізації;

– усебічна децентралізація;

– незалежне розгортання;

– ізольовані збої;

– усебічне спостереження [20].

Отже, мікросервісна архітектура є складним завданням. Для великих підприємств, які зазвичай розробляють складні програмні продукти, мікросервісна архітектура є єдиним способом, який дозволить впоратися зі складністю та бути конкурентоспроможними на ринку, при цьому має бути декілька досвідчених команд, щоб складний додаток розбити на сервіси. Для малих та середніх підприємств мікросервісну архітектуру треба використовувати для сталої розробки ПЗ щодо довгострокової вигоди.

Тепер можна порівняти три програмні архітектури, щоб візуально визначити відмінності між ними (рис. 6.6). Монолітні додатки складаються із взаємозалежних, неподільних блоків і мають дуже низьку швидкість розробки.

Сервіс-орієнтована архітектура SOA (Service-Oriented Architecture) – це підхід у проектуванні, за якого декілька сервісів працюють спільно для надання якогось кінцевого набору можливостей, вони помірно пов’язані між собою і різняться повільною розробкою. Під сервісом тут зазвичай розуміють повністю окремий процес ОС. Зв’язок між такими сервісами здійснюють через Інтернет-дзвінки, а не через виклики методів у межах процесу. Цей підхід був спрямований на забезпечення можливості повторного використання програмних засобів, щоб два і більше додатка для кінцевого користувача могли застосовувати одні й ті самі сервіси. У цій архітектурі *Enterprise Svc Bus* – це сервісна шина підприємства, яка спрощує виклик служби додатками (або їх частинами), але і допомагає їм передавати дані та поширювати повідомлення про події. Дизайн ESB реалізує безліч визнаних шаблонів проектування і специфікацій стандартів.

Мікросервіси – це дуже маленькі, слабо пов’язані незалежні сервіси, яким властива швидка розробка і безперервна інтеграція [5].

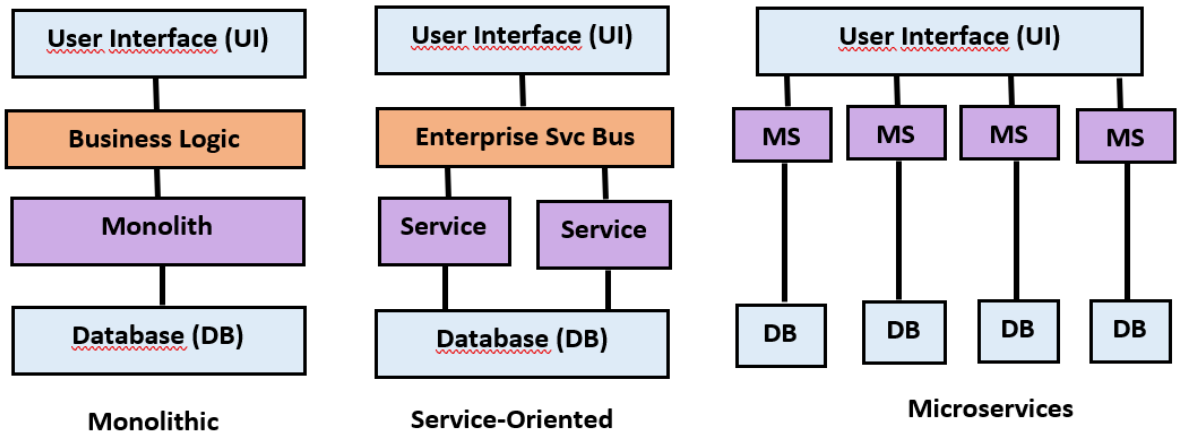


Рис. 6.6. Архітектури розробки програмного забезпечення [5]

6.4.2. Оркестрація, кластеризація і керування

Більшість програмних систем з часом необхідно вдосконалювати: додавати нові функціональні можливості, вилучати застарілі. Крім того, у користувачів постійно змінюються вимоги, тому потрібно забезпечувати оперативне масштабування ресурсів як у бік збільшення, так і в бік зменшення. Також в організаціях часто застосовують численні аналогічні системи для підтримання виконання нерегулярних, допоміжних завдань, таких як інтелектуальний аналіз даних (*data mining*), які відокремлені від основної системи, але вимагають значних ресурсів чи організації взаємодії з наявною системою.

Для вирішення цих серйозних проблем використовують різноманітні програмні засоби і рішення, що більшою чи меншою мірою охоплюють такі області:

– ***кластеризація (clustering)*** – об’єднання у групи «хостів» – віртуальних машин або апаратних – та створення для них загального мережевого середовища. *Кластер слід сприймати як єдиний ресурс, а не як групу окремих комп’ютерів*;

– ***оркестрації (orchestration)*** – забезпечення спільної роботи всіх елементів системи. Запуск контейнерів на відповідних хостах і встановлення з'єднань між ними. Організаційна система також може включати підтримку масштабування, автоматичного відновлення після критичних збоїв та інструменти зміни балансування навантаження на вузли;

– ***керування (management)*** – забезпечення загального контролю і спостереження за системою та підтримка різних адміністративних завдань.

Керування життєвим циклом контейнера і керування самим контейнером стає надзвичайно важким, коли кількість контейнерів постійно збільшується в міру збільшення попиту.

Оркестрація контейнерів – це автоматизація і керування життєвим циклом контейнерів і послуг [6]. Оркестрація дозволяє створювати інформаційні системи, які складаються з безлічі контейнерів, *кожний з яких відповідає тільки за одне певне завдання*, тобто *один контейнер – це один сервіс*, а спілкування між контейнерами здійснюється через мережеві порти і загальні каталоги. Оркестрація дозволяє керувати роботою контейнерів, запущених у великих і динамічних середовищах, автоматизувати і керувати такими завданнями:

– створення та розгортання контейнерів;

– балансування навантаження;

– поділ ресурсів між контейнерами;

– моніторинг контейнерів і серверів;

– масштабування контейнерів;

– перенесення контейнерів з одного сервера на інший, якщо на першому виникає нестача ресурсів.

6.4.3. Інструментальні засоби оркестрації і кластеризації

На ринку IT-послуг є багато інструментів для оркестрації контейнерів. До основних інструментальних засобів оркестрації і кластеризації в екосистемі *Docker* належать: *Swarm, Fleet, Kubernetes* та *Mesos* [6].

Swarm – це власне рішення кластеризації компанії Docker, в якому велика увага приділяється проблемам оркестрації, зокрема під час спільного використання Docker Compose.

Fleet – це низькорівнева система кластеризації і планування, яка використовується CoreOS.

Kubernetes – це високорівневе рішення оркестрації, в якому за замовчуванням вбудовані функції відновлення після критичних збоїв і масштабування, яке може працювати поверх інших рішень кластеризації.

Mesos – низькорівнева система кластеризації, здатна працювати із «програмними середовищами» більш високого рівня, забезпечуючи надійне, повноцінне рішення кластеризації та оркестрації.

Для кожного інструменту розглянемо його особливі функціональні можливості та властивості.

Swarm (**https://docs.docker.com/swarm/**) являє собою інструментальний засіб компанії *Docker* для керування та створення кластерів *Docker*-контейнерів, тобто для швидкого розгортання кластера хостів на вашому локальному комп'ютері або на підтримуваних хмарних платформах. *Docker Swarm* перетворює набір хостів *Docker* (або пул *Docker*-хостів) в один послідовний кластер (або віртуальний хост), який називають *Swarm. Swarm* використовує стандартний інтерфейс програмування *Docker API*, тому контейнери можна запускати звичайними командами *docker run*, а *Swarm* подбає про вибір відповідного хоста для кожного контейнера, який запускається.

Базова архітектура кластера проста: на кожному хості (його ще називають доступним вузлом – *available node*) запускають агент (*agent*)

Swarm, а на окремому хості запускають менеджер *Swarm* (*node manager*). Менеджер відповідає за внутрішній стан кластера – оркестрацію і планування роботи контейнерів, розподілених по хостах (рис. 6.7). Агенти запускають контейнери, їх у літературі називають ще робочими нодами або воркерами) .

Swarm доцільно використовувати для малих та середніх підприємств.

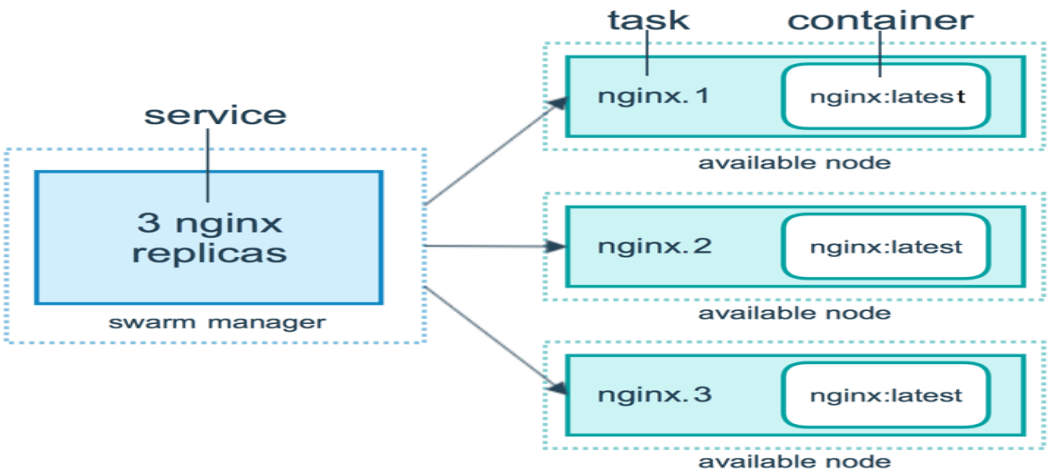


Рис. 6.7. Структура сервісу *Swarm manager*

Установка і налаштування режиму Docker *Swarm* на сервері Ubuntu 20.04

Оновимо системний репозиторій до останньої версії:

```
sudo apt-get update -y && sudo apt-get upgrade -y
```

Запускаємо службу Docker:

```
sudo systemctl start docker && sudo systemctl enable docker
```

Для запуску Docker необхідні права *root*, а для інших користувачів доступ виходить тільки за допомогою *sudo*. Якщо треба запустити *docker* без використання sudo, є можливість створити Unix і включити в нього необхідних користувачів за рахунок виконання таких рядків коду:

```
sudo groupadd docker && sudo usermod -aG docker dockeruser
```

Для коректної роботи кластера *Docker Swarm* потрібні такі порти для коректної роботи брандмауера Linux:

• TCP-порт 2376 – для безпечної взаємодії клієнтів Docker і роботи Docker Machine (програми, яка забезпечує оркестровку хостів Docker);

• TCP-порт 2377 – для взаємодії нод кластера Docker Swarm (потрібно відкрити тільки на нодах-менеджерах);

• TCP- та UDP-порт 7946 – для зв'язку між нодамі (виявлення мережі контейнера);

• UDP-порт 4789 – для трафіку оверлейної мережі.

TCP та UDP – відповідні протоколи.

Вийшовши із системи, робимо вхід через *dockeruser*:

sudo ufw allow 2376/tcp && sudo ufw allow 7946/udp && sudo ufw allow 7946/tcp && sudo ufw allow 80/tcp && sudo ufw allow 2377/tcp && sudo ufw allow 4789/udp

Перезавантажуємо брандмауер, включивши його при завантаженні:

sudo ufw reload && sudo ufw enable

Виконуємо перезавантаження Docker:

sudo systemctl restart docker

Створюючи кластер *Docker Swarm,* необхідно визначитися з IP-адресою, за рахунок якої ваш вузол буде діяти як диспетчер:

docker swarm init --advertise-addr 192.168.0.103

Результат буде таким:

```
Swarm initialized: current node (iwjtf6u951g7rpx6ugkty3ksa) is now a
manager.
To add a worker to this swarm, run the following command:
   docker swarm join --token SWMTKN-1-
5p5f6p6tv1cmjzq9ntx3zmck9kpgt355qq0uaqoj2ple629dl4-
5880qso8jio78djpx5mzbqcfu 192.168.0.103:2377
To add a manager to this swarm, run 'docker swarm join-token manager' and
follow the instructions.
```

Перевіряємо його стан: ***docker node ls***

Якщо все працює правильно, буде виведене таке:

ID	HOSTNAME	STATUS	AVAILABILITY	MANAGER STATUS
iwjtf6u951g7rpx6ugkty3ksa *	Manager-Node	Ready	Active	Leader

Перевіряємо статус *Docker Swarm Cluster* таким чином:

code> docker info

Результат має бути таким:

Containers: 0
Running: 0
Paused: 0
Stopped: 0
Images: 0
Server Version: 17.09.0-ce
Storage Driver: overlay2
Backing Filesystem: extfs
Supports d_type: true
Native Overlay Diff: true
Logging Driver: json-file
Cgroup Driver: cgroupfs
Plugins:
Volume: local
Network: bridge host macvlan null overlay
Log: awslogs fluentd gcplogs gelf journald json-file logentries splunk syslog
Swarm: active
NodeID: iwjtf6u951g7rpx6ugkty3ksa
Is Manager: true
ClusterID: fo24c1dvp7ent771rhrjhplnu
Managers: 1
Nodes: 1
Orchestration:
Task History Retention Limit: 5
Raft:
Snapshot Interval: 10000
Number of Old Snapshots to Retain: 0
Heartbeat Tick: 1
Election Tick: 3
Dispatcher:
Heartbeat Period: 5 seconds
CA Configuration:

```
    Expiry Duration: 3 months
    Force Rotate: 0
 Autolock Managers: false
 Root Rotation In Progress: false
 Node Address: 192.168.0.103
 Manager Addresses:
   192.168.0.103:2377
Runtimes: runc
Default Runtime: runc
Init Binary: docker-init
containerd version: 06b9cb35161009dcb7123345749fef02f7cea8e0
runc version: 3f2f8b84a77f73d38244dd690525642a72156c64
init version: 949e6fa
Security Options:
  apparmor
  seccomp
    Profile: default
Kernel Version: 4.4.0-45-generic
Operating System: Ubuntu 16.04.1 LTS
OSType: linux
Architecture: x86_64
CPUs: 1
Total Memory: 992.5MiB
Name: Manager-Node
ID: R5H4:JL3F:OXVI:NLNY:76MV:5FJU:XMVM:SCJG:VIL5:ISG4:YSDZ:KUV4
Docker Root Dir: /var/lib/docker
Debug Mode (client): false
Debug Mode (server): false
Registry: https://index.docker.io/v1/
Experimental: false
Insecure Registries:
  127.0.0.0/8
Live Restore Enabled: false
```

Вузол налаштований правильно, його треба додати у *Swarm Cluster*. Спочатку копіюємо виведення команди «***swarm init***» з виведення результату вище, а потім вставляємо це виведення в робочий вузол для приєднання до ***Swarm Cluster***:

docker swarm join --token SWMTKN-1-5p5f6p6tv1cmjzq9ntx3zmck9kpgt355qq0uaqoj2ple629dl4-5880qso8jio78djpx5mzbqcfu 192.168.0.103:2377

Маємо побачити таке виведення:

This node joined a swarm as a worker.

Виводимо список робочого вузла: ***docker node ls***

```
ID                            HOSTNAME        STATUS    AVAILABILITY     MANAGER STATUS
iwjtf6u951g7rpx6ugkty3ksa *   Manager-Node    Ready     Active           Leader
snrfyhi8pcleagnbs08g6nnmp     Worker-Node     Ready     Active
```

Docker Swarm Cluster запущений і працює, тепер можна запустити веб-сервіс у *Docker Swarm Mode* – розгортання служби веб-сервера:

docker service create --name webserver -p 80:80 httpd

Має бути створено контейнер веб-сервера *Apache* і зіставлено його з 80-м портом, дозволивши мати повний доступ до необхідного веб-сервера *Apache* з віддаленої системи. Тепер запускаємо перевірку працюючого сервісу за допомогою команди:

docker service ls

Отримуємо результат:

```
ID             NAME
MODE           REPLICAS    IMAGE         PORTS
nnt7i1lipo0h   webserver   replicated    0/1         apache:latest      *:80->80/tcp
```

Запускаємо службу масштабування веб-сервера:

docker service scale webserver = 2

Перевіряємо стан за допомогою команди: ***docker service ps webserver***

```
ID             NAME          IMAGE          NODE          DESIRED STATE   CURRENT STATE
ERROR   PORTS
7roily9zpjvq   webserver.1   httpd:latest   Worker-Node   Running         Preparing about a minute ago
r7nzo325cu73   webserver.2   httpd:latest   Manager-Node  Running         Preparing 58 seconds ago
```

Веб-сервер Apache працює, можна отримати доступ до веб-сервера:

Служба веб-сервера Apache тепер розподілена по двох вузлах. *Docker Swarm* забезпечує доступність вашого сервісу. Якщо веб-сервер відключається на робочому вузлі, то новий контейнер буде запущений на вузлі менеджера. Для перевірки доступності слід зупинити службу Docker на робочому вузлі:

sudo systemctl stop docker

Запустимо службу веб-сервера за допомогою команди:

docker service ps webserver

Буде виведено таку інформацію:

ID NAME IMAGE NODE DESIRED STATE CURRENT STATE ERROR PORTS
ia2qc8a5f5n4 webserver.1 httpd:latest Manager-Node Ready Ready 1 second ago
7roily9zpjvq _ webserver.1 httpd:latest Worker-Node Shutdown Running 15 seconds ago
r7nzo325cu73
(продовження рядка)
webserver.2 httpd:latest Manager-Node Running Running 23 minutes ago

Таким чином, встановлено і налаштовано кластер *Docker Swarm* для ОС Ubuntu 16.04. Головною перевагою *Swarm* є використання тільки прямих звернень до функцій *Docker API*, що дає можливість переміщати навіть великі навантажені робочі середовища та програми в інші кластери або розподіляти їх по декількох кластерах. Використання *Mesos* або *Kubernetes* ускладнює створення переносимої архітектури.

Fleet (https://coreos.com/fleet/ або https://github.com/coreos/fleet) – це інструментальний засіб керування кластерами зі складу *CoreOS*. Він позиціонується як «низькорівневий механізм кластера», тобто призначений для формування «базового рівня» для рішень більш високих рівнів, таких як *Kubernetes*. Основою *Fleet* є механізм *systemd*, який забезпечує ініціалізацію системи і сервісів на окремому комп'ютері, але *Fleet*

розширює ці функції для кластерів з багатьох комп'ютерів. Механізм *Fleet* зчитує юніт-файли systemd, потім розподіляє ці файли по всіх вузлах кластера.

Kubernetes

Проект *Kubernetes* (скорочено *K8s*) виріс із системи керування кластерами *Borg*. Внутрішній продукт пошукового гіганта Google отримав назву на честь кіборгів з легендарного серіалу «Зоряний шлях» [6]. Команді розробників *Google Borg* була поставлене масштабне завдання – створити відкрите ПЗ для оркестрування контейнерів, яке стане внеском *Google* у розвиток світових IT-технологій. Додаток було написано на основі мови *Go*.

Основи роботи *K8s* – застосування декларативного підходу, тобто розробнику потрібно вказати, чого необхідно досягти, а не способи досягнення. У *Kubernetes* застосовано базу *Linux*-контейнерів (наприклад, *Docker*, *Containerd* або *CRI*-O) та опис – скільки буде потрібно контейнерів і в якій кількості їм потрібні ресурси. Саме розгортання контейнерів відбувається на основі робочих *нод* – віртуальних або фізичних машин.

Основними завданнями *Kubernetes* є:

– розгортання контейнерів й усіх операцій для запуску необхідної конфігурації;

– масштабування і запуск декількох контейнерів одночасно на великій кількості хостів;

– балансування безлічі контейнерів у процесі запуску з рівномірним розподілом навантаження.

Основні компоненти *Kubernetes* наведено на рис. 6.8. Пояснімо їх значення.

• *Node (Нода)* – ноди або вузли – це віртуальні або фізичні машини, на яких розгортають і запускають контейнери. Сукупність нод утворює кластер *Kubernetes*.

Перша запущена нода або майстер-нода безпосередньо керує кластером, використовуючи для цього менеджер контролерів (*controller manager*) і планувальник (*scheduler*). Вона відповідальна за інтерфейс взаємодії з користувачами через сервер API і містить сховище «*etcd*» з конфігурацією кластера, метаданими та статусами об’єктів.

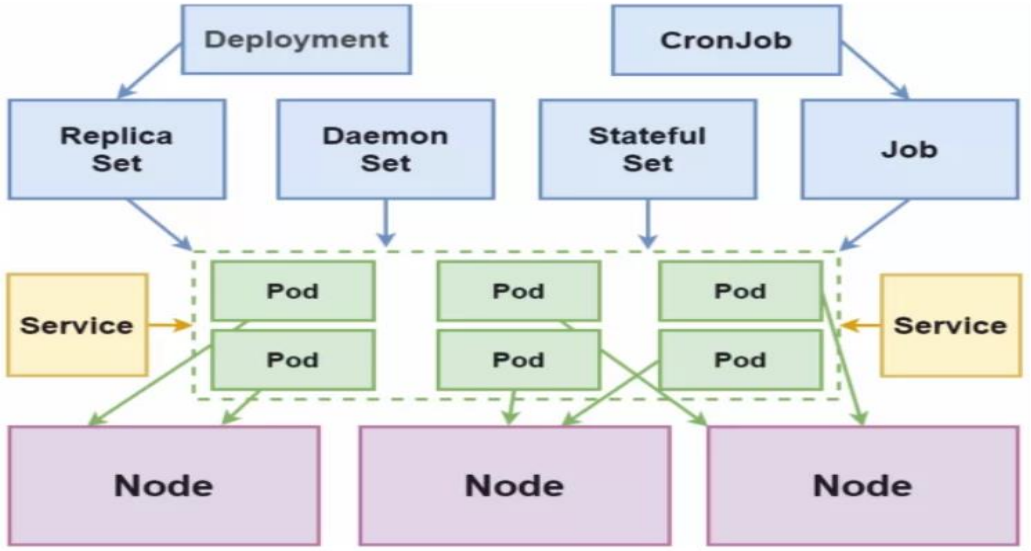


Рис. 6.8. Схема взаємодії основних компонентів *Kubernetes* [6]

• *Namespace (Простір імен (назв))* – об’єкт, призначений для розмежування ресурсів кластера між командами і проектами. Простори імен – це декілька віртуальних кластерів, запущені на одному фізичному.

• *Pod (Под)* – первинний об’єкт розгортання й основний логічний юніт у *K8s*. Поди – набір з одного і більше контейнерів для спільного розгортання на *ноді*.

Групування контейнерів різних типів потрібне в тому випадку, коли вони взаємозалежні й повинні запускатися в одній *ноді*. Це дозволяє збільшити швидкість відгуку під час взаємодії. Наприклад, це можуть бути контейнери, що зберігають веб-додаток і сервіс для його кешування.

• *ReplicaSet (Набір реплік)* – об’єкт, який відповідає за опис і контроль за декількома екземплярами (репліками) подів, створених на кластері. Наявність більше однієї репліки дозволяє підвищити стійкість від відмов і

масштабування додатків. На практиці *ReplicaSet* створюють з використанням *Deployment*.

• *ReplicaSet* є більш просунутою версією попереднього способу організації створення реплік (реплікації) у *K8s – Replication Controller*.

• *Deployment (Розгортання)* – об'єкт, в якому зберігається опис подів, кількість реплік та алгоритм їх заміни в разі зміни параметрів. Контролер розгортання дозволяє виконувати декларативні поновлення (за допомогою опису потрібного стану) на таких об'єктах, як ноди і набори реплік.

• *StatefulSet (Набір стану*) – дозволяє розгортати і керувати одним або декількома подами. Але на відміну від них, ідентифікатори подів мають передбачувані значення і зберігають їх під час перезапуску.

• *DaemonSet (Набір демона)* – об'єкт, який відповідає за те, щоб на кожній окремій ноді (або ряді обраних) запускався один екземпляр обраного пода.

• *Job/CronJob (Завдання/Завдання за розкладом)* – об'єкти для регулювання одноразового або регулярного запуску обраних подів і контролювання завершення їх роботи. Контролер *Job* відповідає за одноразовий запуск, *CronJob* – за запуск декількох завдань за розкладом.

• *Label/Selector (Мітки/селектори)* – мітки, призначені для марку вання ресурсів. дозволяють спростити групові маніпуляції з ними. Селектори дозволяють вибирати/фільтрувати об'єкти на основі значення міток.

• *Service (Сервіс)* – засіб для публікації додатка як мережевого сервісу, використовується також для балансування трафіку/навантаження між подами.

Установка Kubernetes, Ubuntu 20.04

Перед початком встановлення *K8s* необхідно спочатку встановити *Docker*. У розд. 6, п. 6.1.8, розглянуто варіант встановлення сертифікованого випуску *docker-ce (Community Edition).*

Однак є ще пакет *docker.io*, який підтримується розробниками *Ubuntu та використовується Debian/Ubuntu для випуску докерів*. Це стара версія двійкового файлу *docker*. Розробники *Ubuntu* не пов'язані з «офіційним сайтом докерів». Хоча веб-сайт докерів не посилається на нього, але це нічого не означає.

Існує інший варіант встановлення Docker, який активно використовують для *Ubuntu 20.04.* Оновлення пакетів залежностей:

sudo apt-get update
sudo apt-get install -y ***docker.io***

Якщо треба працювати з більш новими версіями контейнерів, запускають такі команди:

sudo apt-get update
sudo apt-get install -y apt-transport-https ca-certificates curl software-properties-common

Додаємо офіційний ключ репозитарію:

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | apt-key add -

sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/$(./etc/os-release; echo "$ID") $(lsb_release -cs) stable"

Файл */etc/os-release* містить ідентифікаційні дані ОС і може бути знайдений тільки в більш нових дистрибутивах Debian, що працюють під керуванням *systemd*. ***$(lsb_release -cs)*** – версія реліз, ***stable*** – останній офіційно випущений дистрибутив. Оновимо базу пакетів ***apt-get update***.

Далі встановлюємо Docker (*docker-ce, docker-ce-cli, containerd.io – пакети, які необхідно встановити*):

sudo apt-get install -y docker-ce docker-ce-cli containerd.io

Для роботи з *Kubernetes* знадобиться встановити компоненти ***kubeadm***, ***kubelet*** та ***kubectl***. Ці утиліти знадобляться для створення керування кластером *Kubernetes*. ***Установка kubeadm, kubelet і kubectl в Ubuntu*:**

kubectl – дозволяє створювати і налаштовувати об’єкти у кластері;

kubelet – займається запуском контейнерів на хостах;

kubeadm – дозволить налаштовувати компоненти, що складають кластер.

В *Ubuntu* ці компоненти можна встановити таким способом:

```
sudo   apt-get update && apt-get install -y apt-transport-https software-properties-common
sudo   curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -
sudo  add-apt-repository "deb http://apt.kubernetes.io/kubernetes-xenial main"
sudo  apt-get update

sudo   apt-get install -y kubelet kubeadm kubectl
sudo   systemctl enable kubelet && systemctl start kubelet
```

Слід переконатися, що «*kubelet*» та «*docker*» користуються одним і тим самим драйвером «*cgroup*». Для цього треба виконати таку команду:

```
cat << EOF> /etc/docker/daemon.json
{
"Exec-opts": ["native.cgroupdriver = systemd"]
}
EOF
```

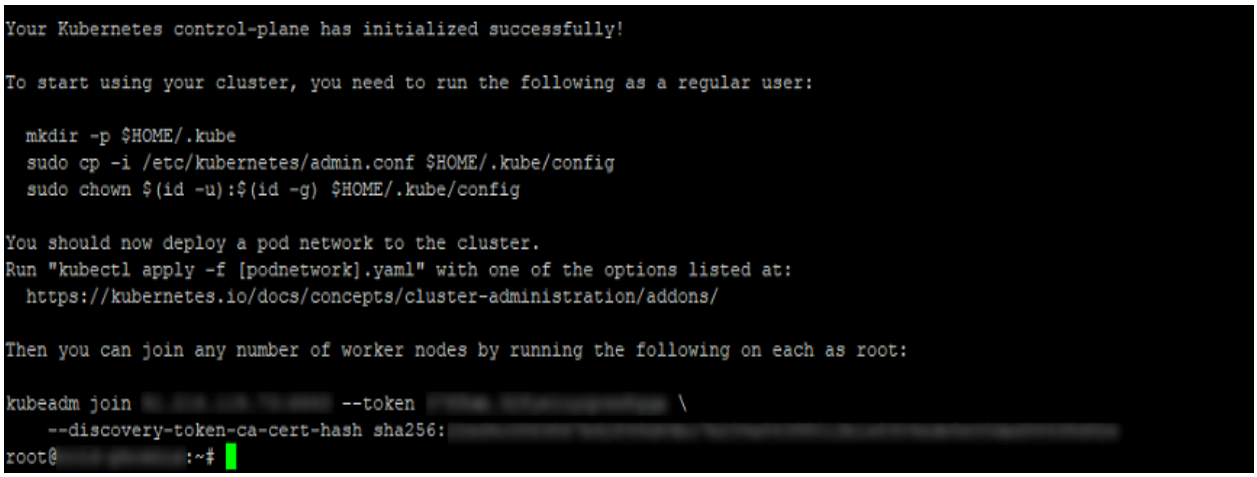
Налаштування Kubernetes

Ініціалізація кластера. Потрібно вказати сервер, на якому встановлений *K8s* (він буде первинним, на якому будуть запускатися інші операції) і виконати ініціалізацію кластера:

```
kubeadm init --pod-network-cidr=10.244.0.0/16
```

Опція *«-pod-network-cidr*» задає адресу віртуальної мережі подів і може різнитися залежно від використовуваного мережевого плагіна. У прикладі був використаний найбільш поширений мережевий плагін *Flannel*. За замовчуванням він використовує мережу «*10.244.0.0/16*», яку було вказано у параметрі. Якщо все зроблено правильно, на екрані відобразиться команда, що дозволяє приєднати інші Ноди кластера до первинного хоста. Команда може відрізнятися залежно від структури

кластера. Її потрібно зберегти на майбутнє. Результат роботи має бути такий:



Тепер необхідно виконати команди від імені користувача, який буде керувати кластером:

mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config

Налаштування мережевого інтерфейсу контейнера (CNI – container network interface). Перед початком запуску додатка у кластері потрібно налаштувати CNI. CNI потрібний для налаштування взаємодії і керування контейнерами всередині кластера.

Відомо багато плагінів для створення CNI (*плагін* – це програмний модуль, який динамічно підключається до основної програми і призначений для розширення її можливостей, його використовують у вигляді бібліотек загального призначення). Пропонуємо застосовувати плагін *Flannel* як найбільш просте і перевірене рішення. Для встановлення *Flannel* виконаємо в терміналі таку команду:

kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel.yml

У результаті будуть відображені імена усіх створених ресурсів.

Додавання вузлів (нод) у кластер

Для додавання нових нодів в наявний кластер треба виконати такий алгоритм:

– підключитися до сервера через мережевий протокол SSH (Secure Shell Access), який забезпечує безпечне віддалене з'єднання та передачу даних зашифрованими каналами;

– встановити на сервер *Docker*, *Kubelet, Kubeadm* (як показано вище);

– виконати команду:

kubeadm join --token <token> <control-plane-host>:<control-plane-port> --discovery-token-ca-cert-hash sha256:<hash>

Отримання токена авторизації кластера (<token>)

Токен – це програмний ключ, призначений для аутентифікації особи для безпечного віддаленого доступу до служб, інформаційних ресурсів.

Для отримання точена застосовують такий алгоритм:

1. Підключитися до сервера через SSH.

2. Запустити команду, яка була наявна на виведенні команди «*kubeadm init*». Наприклад:

kubeadm join --token <token> <control-plane-host>: <control-plane-port> --discovery-token-ca-cert-hash sha256: <hash>

Якщо токена немає, його можна отримати, виконавши таку команду на майстер-ноді:

kubeadm token list

Результат має бути приблизно такий: за замовчуванням, термін дії токена – 24 години. Якщо потрібно додати новий вузол у кластер після закінчення цього періоду, можна створити новий токен командою ***kubeadm token create***. Результат виконання команди буде приблизно такий: ***5didvk.d09sbcov8ph2amjw***.

Якщо значення параметра «***-discovery-token-ca-cert-hash***» невідоме, його можна отримати за допомогою команди:

openssl x509 -pubkey -in /etc/kubernetes/pki/ca.crt | openssl rsa -pubin -outform der 2>/dev/null | openssl dgst -sha256 -hex | sed 's/^.* //'

Буде отримано приблизно такий результат:

8cb2de97839780a412b93877f8507ad6c94f73add17d5d7058e91741c9d5ec78

Для введення *IPv6-адреси* у параметр «*<control-plane-host>: <control-plane-port>*», адреса має бути записана у квадратні дужки. Наприклад: ***[fd00::101]:2073***. Через декілька секунд нода повинна з'явитися у виведенні команди: ***kubectl get nodes***

Наступна команда дозволяє запускати контейнери на майстрі, особливо, якщо кластер містить лише одну ноду:

kubectl taint nodes --all node-role.kubernetes.io/master-

Перевірка працездатності кластера

Перевірити, чи запустився кластер і ви правильно він працює можна так: ***kubectl -n kube-system get pods***

Будуть відображені системні POD'и *k8s*:

```
root@ ~# kubectl -n kube-system get pods
NAME                                READY   STATUS    RESTARTS   AGE
coredns-66bff467f8-4f4mr            1/1     Running   0          43m
coredns-66bff467f8-r2x4h            1/1     Running   0          43m
etcd-                               1/1     Running   1          43m
kube-apiserver-                     1/1     Running   1          43m
kube-controller-manager-            1/1     Running   1          43m
kube-flannel-ds-amd64-4hsj6         1/1     Running   0          4m26s
kube-proxy-7bmw7                    1/1     Running   1          43m
kube-scheduler-                     1/1     Running   1          43m
```

Установка завершена. Далі можна продовжити налаштування *K8s* для роботи з веб-додатками. Наприклад, підключити диспетчер пакетів «*helm*»

для автоматичного розгортання додатків або контролер «*nginx ingress*», який відповідає за маршрутизацію зовнішнього трафіку.

Контрольні запитання

1. Що таке контейнер, яке його призначення?
2. У чому полягають відмінності між віртуальними машинами та контейнерами?
3. У чому полягають переваги та недоліки контейнерів?
4. Які технології використовують під час створення контейнерів?
5. Що таке образ контейнера та з чого він складається?
6. Що таке репозиторій?
7. Які основні компоненти Docker?
8. Що таке реєстри? Яке їх призначеня?
9. Які вам відомі інструментальні засоби для Docker?
10. Що таке Dockerfile?
11. Які команди використовують для роботи з реєстрами?
12. Що таке рівні образу контейнера?
13. Які вам відомі команди щодо керування контейнерами?
14. Як створити образ контейнера?
15. Як переглянути активні та неактивні контейнери?
16. Які команди використовують під час роботи з Dockerfile?
17. Що таке Docker Compose?
18. Які команди використовують під час роботи з Docker Compose?
19. Як забезпечити обмін інформацією між контейнерами на одному хості?
20. У чому полягає життєвий цикл програмного забезпеченні з використанням контейнера Docker?
21. Як запустити контейнер за допомогою Docker Compose?
22. Як зв’язати контейнер з локальною файловою системою?

23. Як запустити конейнер в автономному режимі?
24. Як запустити контейнер Docker за допомогою створеного образу?
25. Які складові додатка з монолітною архітектурою?
26. У чому полягає мікросервісний підхід під час розробки ПЗ?
27. Що означає DevOps?
28. Які переваги та недоліки мікросервісної архітектури?
29. Що таке кластерізація?
30. Що таке оркестрація? Які завдання вона виконує?
31. Які інструментальні засоби оркестрації та кластерізаііі вам відомі?
32. Які можливості менеджера Swarm?
33. У чому полягають особливості Kubernetes?
34. Що таке токен?

Розділ 7

ОПЕРАЦІЙНІ СИСТЕМИ СІМЕЙСТВА WINDOWS

7.1. Архітектура Windows

7.1.1. Історія Windows

Windows – це сучасна ОС, розробником якої є компанія Microsoft. Процес розробки цієї системи можна розбити на чотири епохи:

– MS-DOS (*Microsoft Disk Operating System*);

– Windows на базі MS-DOS (1985-1990);

– Windows на базі NT;

– сучасна Windows.

MS-DOS (1980–1985) – це 16-бітна ОС з інтерфейсом командного рядка, яка працювала в реальному режимі. Windows 3.0 на базі MS-DOS (1985–1990) – це було графічне середовище, яке працювало поверх системи MS-DOS, що, як і раніше, керувала комп'ютером і файловою системою. Усі програми працювали в одному адресному просторі, й помилка у будь-якій із них могла спричинити зупинку всієї системи.

У 1993 році було випущено Windows NT 3.1 (***New Technology***), яка проектувалася як система, здатна до переносу на різні процесори. Це вже були багатопроцесорні 32-розрядні комп'ютерні системи з великою кількістю пам'яті та віртуальною пам'яттю.

У 1995 році було випущено Windows 95. Вона мала багатофункціональні можливості повноцінної ОС, у тому числі віртуальну пам'ять, керування процесами, багатозадачність, а також ввела 32-бітові інтерфейси програмування. Однак їй не вистачало безпеки, а також ізоляції додатків один від одного й від ОС. Тому проблеми зі стабільністю залишилися навіть після випуску Windows 98 і Windows Me (*Millennium*

Edition), в яких, як і раніше, працював 16-бітний асемблерний код ОС MS-DOS.

У 1996 році було випущено Windows NT 4.0. Наявність загального інтерфейсу прикладного програмування (*API – Application Programming Interface*) у Windows на базі MS-DOS було надзвичайно важливим для успіху для Windows NT.

У 2000 році версія Windows 2000 являла собою істотний розвиток системи NT. Було додано підтримку таких важливих технологій: Plug-and-Play (що рятувало від необхідності розбиратися з перемичками під час установки карт розширення PCI), служби мережевого каталогу (для підприємств), поліпшене керування електроживленням (для ноутбуків); був поліпшений графічний інтерфейс користувача. Технологія Plug-and-Play дозволяє робити установку нового обладнання на комп'ютер (або на інші технічні пристрої) в автоматичному режимі. Слова Plug and Play можна перекласти як «підключив і працюй». Ця технологія звільняє користувача від такої рутинної роботи, як пошук та установка драйверів на новий пристрій [3].

PCI (англ. *Peripheral component interconnect* – взаємозв'язок периферійних компонентів) – шина «введення – виведення» для підключення периферійних пристроїв до материнської плати комп'ютера.

Технічний успіх Windows 2000 привів до того, що компанія Microsoft стала покращувати сумісність додатків і драйверів пристроїв у новій версії NT (Windows XP), щоб остаточно витіснити Windows 98. Windows XP мала новий і більш зручний графічний інтерфейс, а також надійно працювала. Windows XP була інстальована на сотнях мільйонів персональних комп'ютерів, що дозволило компанії Microsoft досягти своєї мети із завершення епохи Windows на базі MS-DOS. З цією ОС справді було дуже зручно і надійно працювати.

Наприкінці 2006 року постала Windows Vista, яка вийшла у світ більш ніж через п’ять років після Windows XP, але не мала популярності серед користувачів, оскільки споживачі перебували у захваті від появи ноутбуків та нетбуків.

Проблеми з Windows Vista були враховані у наступному випуску, Windows 7. Слід зазначити, що Windows 7 та Windows XP і стали найпопулярнішими на сьогодні версіями Windows, але їх підтримка корпорацією Microsoft вже припинена.

У 2012 році була випущена Windows 8, яка не отримала загального визнання через багато незручностей для користувача (наприклад, відсутність кнопки Пуск на панелі завдань). У 2013 році Microsoft випустила оновлення під назвою Windows 8.1.

У 2015 році було випущено Windows 10, яка написана на *C*, *C++*, *C Sharp*. Це ОС для персональних комп’ютерів і робочих станцій. На сьогоднішній день багато чого вже виправлено. однак весь час відбуваються оновлення цієї системи. Windows 10 також має командний рядок – CMD (Win+R або *Выполнить CMD*).

Операційна система Windows 10 спрямована на уніфікацію, тобто допускається установка на комп’ютери, ноутбуки, планшети, а також смартфони та ігрову приставку Xbox One. Єдина платформа забезпечує можливість синхронізації налаштувань. Для основної версії для комп’ютерів висуваються такі системні вимоги:

• Процесор із частотою не менше 1 ГГц.

• ОЗУ від 1 Гб (для 32х систем) і 2 Гб (для 64х систем).

• Від 16 до 20 Гб вільного місця на жорсткому диску.

• Наявність DirectX 9 і вище.

DirectX (від англ. Direct – прямий, безпосередній) – це набір API, розроблених для вирішення завдань, пов’язаних із програмуванням під

Microsoft Windows при написанні комп'ютерних ігор. Найчастіше оновлені версії DirectX поставляються разом з ігровими додатками.

Є велика кількість інших нововведень Windows 10, які слід перерахувати:

– оновлені екран вітання та екран блокування пристрою;

– вхід у систему за рахунок служби біометричних даних Windows Hello;

– «Панель керування» замінена «Параметрами» з більш орієнтованим на користувача інтерфейсом;

– частина значків перемальована;

– оновлено годинник і календар;

– магазин додатків Windows став більш зручним;

– замість Internet Explorer використано Microsoft Edge;

– встановлено новий додаток «Початок роботи».

Редакцій Windows 10 є досить велика кількість: домашня, професійна, корпоративна, мобільна, для навчальних закладів.

7.1.2. Модель операційної системи

Windows – одна з найбільш багатогранних і гнучких ОС, вона працює на абсолютно різних архітектурах і доступна в різних варіантах [2, 21]. На сьогодні вона підтримує архітектури x86, x64, ARM та ARM64. Нині всі Windows-додатки написані за допомогою API-інтерфейсів (Application Programming Interface – програмний інтерфейс режиму користувача для ОС сімейства Windows). Це стосується як 32-розрядних, так і 64-розрядних програмних інтерфейсів Windows. У більшості багатокористувацьких ОС додатки відокремлені від самої ОС. *Код ядра ОС виконується у привілейованому режимі процесора* (*або режимі ядра*), для нього доступні системні дані та обладнання. Код додатків виконується у *непривілейованому режимі процесора* (*або режимі користувача*), йому надається обмежений набір інтерфейсів та обмежений доступ до системних даних,

а прямий доступ до обладнання заблокований. Коли програма режиму користувача викликає системну функцію, процесор виконує спеціальну команду, яка перемикає потік виклику у режим ядра. Після завершення системної функції ОС перемикає контекст потоку назад у режим користувача і дає можливість викликаючій стороні продовжити роботу.

Як і багато UNIX-подібних систем, Windows є монолітною ОС. Це означає, що велика частина коду ОС і коду драйверів пристроїв використовує один захищений простір пам'яті захищеного режиму з одночасним впровадженням додаткових технологій захисту ядра, таких як безпека на базі віртуалізації, функції *Device Guard* (пристрій охорони) та *Hyper Guard* (гіперохорона). Захист – одна із причин, з яких Windows користується репутацією надійної і стабільної системи як серверних додатків, так і платформи робочих станцій, але при цьому забезпечується швидка робота базових сервісів ОС, таких як керування віртуальною пам'яттю, файлове «введення – виведення», мережеві комунікації та спільне використання принтера.

У компонентах режиму ядра Windows також втілюються базові принципи об'єктно-орієнтованого проектування, а саме, використовуються формальні інтерфейси для передачі параметрів, а також читання і / або запису структур даних. Велика частина коду ОС написана мовою С, в якій немає прямої підтримки об'єктно-орієнтованих конструкцій, таких як поліморфні функції або успадкування класів, однак реалізація об'єктів на базі С у системі Windows запозичує деякі можливості об'єктно-орієнтованих мов, але не залежить від них.

Огляд архітектури

Спрощену версію архітектури системи зображено на рис. 7.1. Звернімо увагу на лінію, що розділяє частини користувацького режиму і режиму ядра ОС Windows. Блоки вище лінії є процесами користувацького режиму, а компоненти нижче лінії – *сервісними засобами ОС режиму*

ядра. Потоки користувацького режиму виконуються в закритому адресному просторі процесів (хоча під час виконання в режимі ядра вони отримують доступ до системного простору).

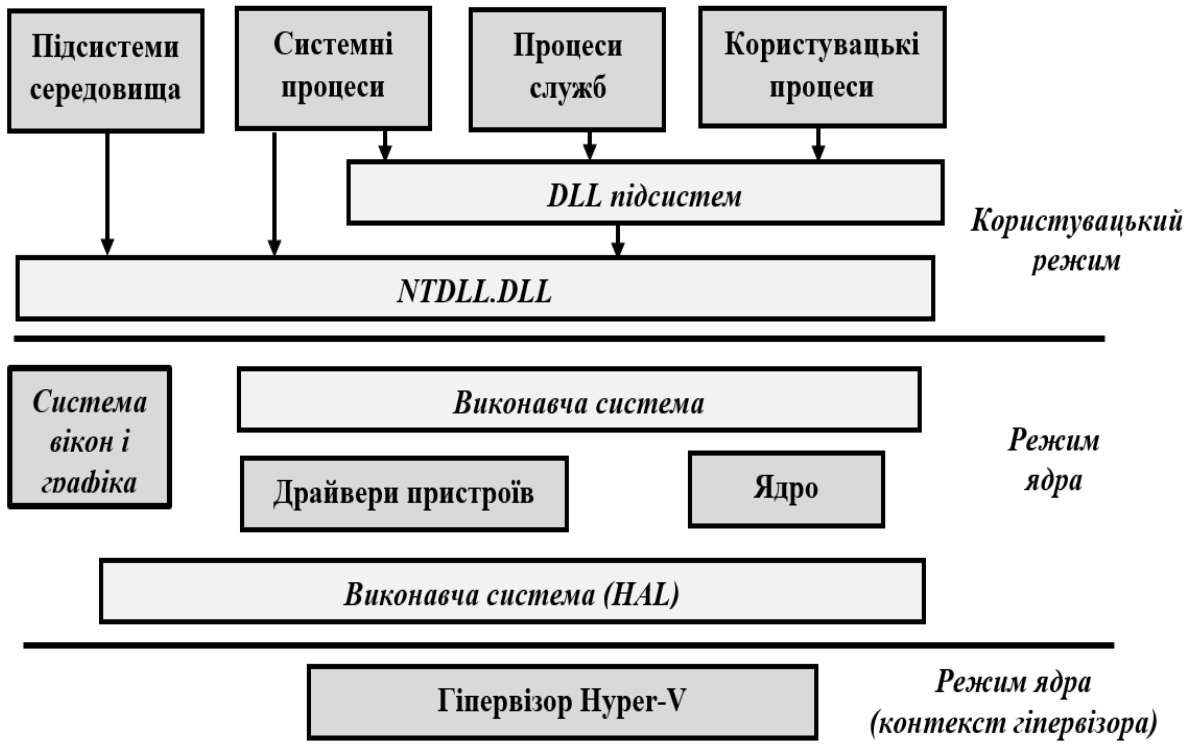


Рис. 7.1. Спрощена архітектура Windows [3]

Таким чином, системні процеси, процеси служб, користувацькі процеси і підсистеми середовища мають власні закриті адресні простори. Друга лінія розділяє компоненти режиму ядра Windows і гіпервізор.

Гіпервізор (монітор віртуальних машин) – програма або апаратна схема, що забезпечує або дозволяє одночасне, паралельне виконання декількох ОС на одному комп'ютері. Гіпервізор також забезпечує ізоляцію ОС одна від одної, захист і безпеку, розподіл ресурсів між різними запущеними ОС і керування ресурсами. Гіпервізор працює на рівні привілеїв процесора (0), він може ізолюватися від ядра з одночасним збереженням контролю над ним (і додатками).

Розглянемо чотири базових *типи процесів режиму користувача*.

• ***Користувацькі процеси.*** Це такі типи: 32-розрядні або 64-розрядні додатки Windows (додатки Windows Apps, що працюють на базі середовища Windows Runtime у Windows 8 і вище, включаються у цю категорію); 16-розрядні додатки Windows 3.1; 16-розрядні додатки MS-DOS; 32-розрядні й 64-розрядні додатки POSIX. Слід зауважити, що 16-розрядні додатки можуть виконуватися тільки в 32-розрядних версіях Windows, а додатки POSIX у Windows 8 вже не підтримуються.

• ***Процеси служб.*** До цієї категорії належать процеси, які є хостами для служб Windows, наприклад служби планувальника завдань і диспетчера друку. (*Хост* – це будь-який пристрій, яке має підключення до локальної мережі або до всесвітньої мережі Інтернет, є складовою частиною цієї мережі, має унікальну IP-адресу та бере участь в обміні інформацією). Служби виконуються незалежно від входу користувача. Багато серверних додатків Windows (такі як Microsoft SQL Server та Microsoft Exchange Server) також містять компоненти, які виконуються як служби.

• ***Системні процеси.*** Фіксовані процеси (такі як процес входу або диспетчер сеансів) не є службами Windows, тобто вони не запускаються диспетчером служб, а саме:

– процес *Idle*, містить один потік для кожного процесора для обліку часу бездіяльності процесора;

– процес *System*, містить більшість системних потоків і дескрипторів режиму ядра;

– процес *Secure System*, містить адресний простір безпечного ядра в VTL 1 (це один із режимів захисту ядра);

– процес *Memory Compression*, містить стислий робочий набір процесів режиму користувача;

– диспетчер сеансів (*Smss.exe*);

– підсистема Windows (*Csrss.exe*);

– ініціалізація сеансу 0 (*Wininit.exe*);

– процес входу (Winlogon.exe):

– диспетчер служб (*Services.exe*) і створювані ним дочірні процеси (такі як системний узагальнений процес, що виконує функції хоста служб (*Svchost.exe*));

– локальна служба аутентифікації (*Lsass.exe*), і якщо активний механізм *Credential Guard* – ізольований локальний сервер аутентифікації (*Lsaiso.exe*).

• ***Серверні процеси підсистем середовища.*** Ці процеси реалізують частину підтримки середовища ОС, що надається користувачеві та програмісту.

Підсистема середовища потрібна, щоб надавати прикладним програмам деяку підмножину сервісних функцій базової виконавчої системи Windows. Різні підсистеми відкривають доступ до різних підмножин вбудованих сервісних функцій Windows. Кожний виконуваний образ (*.exe*), тобто виконуваний модуль, зв'язується з однією і тільки однією підсистемою. Під час виконання образу код створення процесу аналізує код типу підсистеми у заголовку образу, щоб сповістити правильну підсистему про новий процес.

Блок DLL підсистем

У системі Windows користувацькі програми не викликають низькорівневих сервісних функцій ОС Windows безпосередньо. Замість цього вони проходять через одну або *декілька динамічних бібліотек (DLL) підсистем*. Роль DLL-підсистем полягає у перетворенні документованих функцій у відповідні внутрішні (й зазвичай недокументовані) виклики системних функцій, реалізовані в основному в *Ntdll.dll*. Користувацькі програми використовують DLL-бібліотеки підсистеми Windows (такі як *Kernel32.dll, Advapi32.dll, User32.dll* та *Gdi32.dll*) через реалізацію функції Windows API.

Ntdll.dll – спеціальна системна бібліотека, призначена для використання DLL-підсистем та «рідних» (*native*) додатків. Вона містить функції двох типів:

– заглушки диспетчеризації для системних функцій виконавчої системи Windows;

– внутрішні допоміжні функції, що використовуються підсистемами, DLL-підсистем і под.

Перша група функцій надає інтерфейс для системних функцій Windows, які можуть викликатися з користувацького режиму. Відомо понад 450 таких функцій: *NtCreateFile*, *NtSetEvent* тощо. Багато цих функцій доступні через Windows API. Для кожної з цих функцій *Ntdll.dll* містить однойменну точку входу. Код функції містить команду для конкретної архітектури, яка ініціює перехід у режим ядра для виклику диспетчера системних сервісних функцій.

До категорії режиму ядра Windows належать такі компоненти:

• *Виконавча система*. Містить базові сервісні функції ОС: керування пам’яттю, керування процесами і потоками, безпека, «введення – виведення», мережева підтримка та міжпроцесна комунікація. Виконавча система утворює верхній рівень *Ntoskrnl.exe*.

• *Ядро Windows*. Утворює нижній рівень функцій *Ntoskrnl.exe*, що надають фундаментальні механізми системи. Це низькорівневі функції ОС: планування потоків, диспетчеризація переривань і виключень, багатопроцесорна синхронізація. Воно також надає набір функцій і базових об’єктів, які виконавча система використовує для реалізації високорівневих конструкцій. Код ядра пишуть переважно на С, а вставки на мові асемблера резервують для завдань, що вимагають доступу до спеціалізованих команд процесора і регістрів, недоступних безпосередньо з мови С [21].

Для посилання на об’єкти ядра у Windows використовують уніфікований засіб на основі описувачів і назв простору імен NT, а також уніфіковану реалізацію в центральному ***диспетчері об’єктів (object manager)***.

Диспетчер об’єктів (*object manager*) керує більшістю об’єктів режиму ядра, які використовуються на виконавчому рівні. Це процеси, потоки, файли, семафори, пристрої та драйвери «введення – виведення», таймери і багато іншого. Об’єкти режиму ядра фактично є структурами даних, які розміщуються і використовуються ядром. Диспетчер об’єктів надає такі засоби: керування розміщенням і звільненням пам’яті для об’єктів, облік квот, підтримка доступу до об’єктів за допомогою описувачів, підрахунок посилань на вказівники у режимі ядра і посилань на описувачі, іменування об’єктів у просторі імен NT, представлення розширюваного механізму для керування життєвим циклом будь-якого об’єкта, надає також уніфіковані засоби для синхронізації, забезпечення безпеки та керування існуванням об’єкта. Тими структурами даних ядра, яким потрібні ці засоби, керує диспетчер об’єктів.

Не тільки додатки використовують об’єкти, якими керує диспетчер об’єктів. Сама ОС також може створювати і використовувати об’єкти для того, щоб дозволити одній компоненті системи зберегти на значний період часу деяку інформацію або передати деяку структуру даних іншій компоненті (при цьому використовувати наявну в диспетчері об’єктів підтримку іменування і часу існування).

Кожний об’єкт диспетчера об’єктів має тип, який використовується для зазначення того, як необхідно керувати життєвим циклом об’єкта цього типу. Це не тип в об’єктно-орієнтованому сенсі – це просто колекція параметрів, що вказуються при створенні типу об’єкта. Для створення нового типу компонент виконавчого рівня просто викликає API диспетчера

об’єктів. Об’єкти займають надзвичайно важливе місце у функціонуванні Windows.

• *Драйвери пристроїв*. У це сімейство входять як драйвери фізичних пристроїв, що перетворюють виклики функцій «введення – виведення» користувачів у конкретні запити «введення – виведення» до пристрою, так і драйвери пристроїв, що не належать до фізичного обладнання, наприклад драйвери файлової системи або мережеві драйвери.

• *Шар абстрагування від обладнання (HAL – Hardware Abstraction Layer, шар апаратних абстракцій)* взаємодіє безпосередньо із процесором, шинами та іншим обладнанням. Відповідає за забезпечення стандартного інтерфейсу до платформеннозалежних ресурсів для ядра, диспетчера «введення – виведення» і драйверів пристроїв.

HAL забезпечує підтримку і відповідає за надання стандартного інтерфейсу до ресурсів процесора, які можуть змінюватися залежно від моделі всередині одного сімейства процесорів. Можливість заміни шару *HAL* забезпечує всім шарам ОС, що лежать вище, незалежність від апаратної архітектури.

• *Віконна і графічна система.* Реалізація функцій графічного інтерфейсу користувача (*GUI – Graphical User Interface*), також відомих як функції *GDI (Graphical Device Interface*, віконний менеджер): робота з вікнами, елементи користувацького інтерфейсу і графічне виведення.

• *Рівень гіпервізора*. Складається всього з одного компонента: власне, гіпервізора. У цьому середовищі немає ані драйверів, ані інших модулів. При цьому гіпервізор складається з декількох внутрішніх рівнів і служб: *власний диспетчер пам’яті, планувальник віртуальних процесів, керування перериваннями і таймером, функції синхронізації, розділи (екземпляри віртуальних машин) і внутрішньопроцесні комунікації (IPC, Inter-Process Communication)* та багато інших.

У табл. 7.1 вказано імена файлів деяких базових компонентів ОС Windows.

Таблиця 7.1

Основні системні файли Windows

Ім’я файлу	Компоненти
Ntoskrnl.exe	Виконавча система та ядро
Hal.dll	HAL
Win32k.sys	Частина підсистеми Windows режиму ядра (GUI)
Hvix64.exe (Intel), Hvax64.exe (AMD)	Гіпервізор
.sys les in\SystemRoot\System32\Drivers	Основні файли драйверів: Direct X, Volume Manager, TCP/IP, TPM і підтримка ACPI
Ntdll.dll	Внутрішні допоміжні функції і заглушки диспетчеризації системних сервісних функцій
Kernel32.dll, Advapi32.dll, User32.dll, Gdi32.dll	DLL основних підсистем Windows

Завантаження Windows

Історично склалося так, що система BIOS (*Basic Input / Output System* – базова система «введення – виведення») – невелика спеціальна програма, яка вказує, де розміщено програму завантаження (завантажувач) ОС: чи то це розділ жорсткого диска (наприклад, C:), чи завантажувальна флешка, чи у мережі. Такою програмою початкового завантаження Windows є ***BootMgr*** (*Boot manager*, Windows Boot Manager) (менеджерів завантаження). Програма BootMgr визначає, чи була система раніше переведена у стан глибокого сну або очікування (це спеціальні режими енергозбереження, які дозволяють системі «прокидатися» без перезапуску від початку процесу початкового завантаження). Якщо це так, то BootMgr завантажує та виконує *WinResume.exe*. В іншому разі BootMgr завантажує та виконує *WinLoad.exe* для виконання нового завантаження.

WinLoad завантажує у пам’ять завантажувальні компоненти системи: ядро і програму виконавчого рівня (зазвичай це *ntoskrnl.exe*), HAL (*Hal.dll*),

що містить розділ SYSTEM-файл, драйвер Win32k.sys (який містить частини режиму ядра підсистеми Win32), а також образи будь-яких інших драйверів, які перераховані в розділі SYSTEM як *завантажувальні драйвери* (*boot drivers*) (це означає, що вони потрібні під час завантаження системи). Якщо в системі є включений *Hyper-V*, WinLoad також завантажує і запускає програму гіпервізора. Після завантаження у пам'ять завантажувальних компонентів Windows керування передається коду низького рівня в NTOS, який починає форматувати HAL, ядро і виконавчий рівень, прив'язувати образи драйверів, а також звертатися до даних конфігурації в розділі SYSTEM (й оновлювати їх). Після ініціалізації всіх компонент режиму ядра створюється перший процес користувацького режиму (який використовує для виконання програму smss.exe, яка подібна до /etc/init у системах UNIX).

Є й інші варіанти завантаження: ***у без-печному режимі*** (*safe-boot*), коли відключається безліч необов'язкових драйверів, через ***консоль відновлення*** (*recovery console*), коли з'являється вікно командного рядка *cmd.exe*.

Winlogon, LogonUI та Userinit

Процес входу Windows (%SystemRoot%\System32\Winlogon.exe) забезпечує вхід і вихід інтерактивних користувачів. *Winlogon.exe* отримує повідомлення про запити на вхід, коли користувач вводить комбінацію клавіш SAS (*Secure Attention Sequence – Ctrl+Alt+Delete*). SAS існує для захисту користувачів від програм перехоплення паролів, що імітують процес входу, тому що комбінація клавіш не може бути перехоплена додатком користувацького режиму.

Оскільки *Winlogon.exe* є критичним системним процесом, від якого залежить робота системи, користувацький інтерфейс для виведення діалогового вікна входу виконують у дочірньому процесі *Winlogon.exe* з іменем *LogonUI.exe*. Коли користувач вводить свої облікові дані або

закриває інтерфейс входу, процес *LogonUI.exe* завершується. Після введення імені користувача та пароля ці дані передаються процесу *локальної служби аутентифікації* (*Lsass.exe*) для перевірки *Lsass.exe*. У разі успішної аутентифікації *Lsass.exe* викликає функцію SRM (наприклад, *NtCreateToken*) для генерування об'єкта маркера доступу, що містить *профіль безпеки користувача.*

7.1.3. Інтерфейс прикладного програмування Win32

Виклики функцій Win32 називають інтерфейсом прикладного програмування Win32 (Win32 API). Ці інтерфейси розкриті й повністю документовані. Вони реалізовані як бібліотечні процедури, які поміщають в оболонку власні системні виклики NT (їх використовують для виконання завдання), або (в деяких випадках) виконують це завдання прямо в користувацькому режимі. Попри те, що власні інтерфейси прикладного програмування NT не опубліковані, велика частина наданої ними функціональності доступна через Win32 API [3]. У табл. 7.2 показано різні низькорівневі виклики Win32 API і власні виклики NT API, для яких вони слугують оболонками.

Філософія інтерфейсів прикладного програмування Windows сильно різниться від філософії UNIX. У системах UNIX функції ОС прості, мають мало параметрів й існує мало таких місць, де одну і ту саму операцію можна виконати декількома способами. Win32 надає дуже детальні інтерфейси з безліччю параметрів (часто є три-чотири способи виконання одного і того ж), причому використовується гібрид функцій низького і високого рівнів (на кшталт *CreateFile* та *CopyFile*). Це означає, що Win32 надає дуже багатий набір інтерфейсів, який, однак, є дуже складним через погане розбиття на рівні цієї системи, де в одному API змішані функції високого і низького рівнів.

Таблиця 7.2

Приклади викликів Win32 API і тих викликів NT API, для яких вони є оболонками

Виклик Win32	Власний виклик NT API
CreateProcess	NtCreateProcess
CreateThread	NtCreateThread
SuspendThread	NtSuspendThread
CreateSemaphore	NtCreateSemaphore
ReadFile	NtReadFile
DeleteFile	NtSetInformationFile
CreateFileMapping	NtCreateSection
VirtualAlloc	NtAllocateVirtualMemory
MapViewOfFile	NtMapViewOfSection
DuplicateHandle	NtDuplicateObject
CloseHandle	NtClose

Win32 має виклики для створення і керування процесами й потоками. Відомо також багато викликів, які належать до міжпроцесного обміну (такі, як створення, знищення та використання м'ютексів, семафорів, подій, портів обміну та інших об'єктів IPC (*Inter Process Communications* – комп'ютерна технологія, обмін даними між потоками одного і / або різних процесів).

Windows реалізує відображення файлів у пам'ять за допомогою трьох абсолютно різних засобів.

По-перше, вона забезпечує інтерфейси, які дозволяють процесам керувати своїм віртуальним адресним простором (у тому числі можна резервувати діапазони адрес для наступного використання).

По-друге, Win32 підтримує абстракцію під назвою *відображення файлів* (*file mapping*), яку використовують для представлення об'єктів у вигляді файлів (відображений файл на рівні NT називають *сегментом*).

Найчастіше відображені файли створюють для посилання на файли за допомогою описувача/вказівника файлу, але їх також можуть створювати для посилання на приватні сторінки, виділені в системному файлі підкачки.

Третій спосіб перетворює «представлення» відображень файлів в адресний простір процесу. Win32 дозволяє створити уявлення тільки для поточного процесу, але засіб NT має більш загальний характер, що дозволяє створювати уявлення для будь-якого процесу (для якого у вас є описувач з достатніми правами). Відображення файлу на адресний простір – це зовсім інший підхід (на відміну від в UNIX функції *mmap*).

Для багатьох програм дуже важливим є файлове «введення – виведення». У загальному поданні Win32-файл є просто лінійною послідовністю байтів. Win32 надає понад 60 викликів для створення і знищення файлів та каталогів, відкриття і закриття файлів, читання й запису в них, запиту і встановлення атрибутів файлів, блокування діапазону байтів, а також багатьох інших основних операцій з організації файлової системи і звернення до файлів.

Ще одна область, для якої у Win32 є виклики, – це безпека. Кожний потік асоціюється з об’єктом режиму ядра, який називають маркером (*token*), і надає інформацію про ідентифікатор і привілеї потоку. Кожний об’єкт може мати ACL (*Access Control List* – список керування доступом), який дуже докладно вказує, які користувачі можуть до об’єкта звертатися та які операції вони можуть з ним виконувати. Такий підхід забезпечує тонке налаштування безпеки, коли конкретним користувачам може бути надано конкретний доступ до будь-якого об’єкта. Модель безпеки здатна до розширення, що дозволяє додаткам додавати нові правила безпеки (такі, як обмеження годин доступу).

Win32 API підтримує також багато функцій для роботи із графічним інтерфейсом користувача (зокрема всі виклики для керування графічним інтерфейсом системи). Є виклики для створення, знищення та

використання вікон, меню, панелей інструментів, рядків стану, смуг прокрутки, діалогових вікон, значків (і безлічі інших наявних на екрані елементів), а також керування ними.

Є виклики для малювання геометричних фігур, їх зафарбовування, керування палітрами використовуваних кольорів, роботи із шрифтами, розміщення значків на екрані. І нарешті, є виклики для роботи з клавіатурою, мишею та іншими пристроями введення, а також для аудіо, друку та інших пристроїв виведення.

7.1.4. Реєстр Windows

Реєстр Windows (*registry*) – системна база даних з інформацією, потрібною для завантаження і налаштування конфігурації системи, загальносистемними параметрами, які керують роботою Windows, базою даних безпеки та поточними налаштуваннями (наприклад, екранною заставкою, яка використовується), а також налаштуваннями, що впливають на швидкодію та поведінку системи [21]. Крім того, реєстр відкриває доступ до тимчасових даних, які зберігаються в пам’яті, таких як поточний стан обладнання системи (які драйвери пристроїв завантажені, які ресурси вони використовують тощо), а також лічильників продуктивності Windows.

Реєстр організований в окремі томи, які називають *розділами* (*hives*). Кожний розділ зберігається в окремому файлі (у каталозі C:\Windows\system32\config\). Коли Windows завантажується, розділ SYSTEM завантажиться у пам’ять тією самою програмою завантаження, яка завантажує ядро та інші файли завантаження (такі, як завантажувальні драйвери) із завантажувального тому.

У розділі SYSTEM Windows містить велику кількість важливої інформації, у тому числі які драйвери з якими пристроями використовувати, яке ПЗ спочатку запускати, а також безліч інших параметрів, які керують роботою системи. До таких завантажувальних

драйверів належать драйвери файлових систем і драйвери дисків для того тому, на якому міститься сама ОС.

Інші конфігураційні розділи використовуються після завантаження системи для опису інформації про ПЗ, яке інстальовано в системі, про користувачів, про інстальовані в системі класи об'єктів COM (*Component Object-Model)* режиму користувача. Реєстраційна інформація про локальних користувачів зберігається в розділі SAM (Security Access Manager). Інформація щодо мережевих користувачів підтримується службою *lsass* у розділі SECURITY, причому вона узгоджується із серверами мережевого каталогу, щоб користувачі могли мати єдині ім'я облікового запису та пароль у всій Мережі. Список використаних у Windows розділів наведено в табл. 7.3.

Таблиця 7.3

Розділи реєстру в Windows*

Файл розділу	Ім'я після монтування	Застосування
STEM	HKLM\SYSTEM	Інформація про конфігурацію ОС (використовується ядром)
HARDWARE	HKLM\HARDWARE	Розділ у пам'яті, в якому записано обладнання, яке виявлено
BCD	HKLM\BCD*	База даних конфігурації завантаження
SAM	HKLM\SAM	Інформація про облікові записи локальних користувачів
SECURITY	HKLM\SECURITY	Інформація служби *lsass* про облікові записи та інша інформація безпеки
DEFAULT	HKEY_USERS\DEFAULT	Розділ за замовчуванням для нових користувачів
NTUSER.DAT	HKEY_USERS<user id>	Розділ для користувачів, що зберігається в домашньому каталозі
SOFTWARE	HKLM\SOFTTWARE	Зареєстровані у COM класи додатків
COMPONENTS	HKLM\COMPONENTS	Маніфести і залежності для компонентів системи

***HKLM – це скорочення для HKEY_LOCAL_MACHINE.**

Для вивчення реєстру у Windows є програма із графічним інтерфейсом ***regedit***, яка дозволяє відкривати і вивчати каталоги (які називають ключами) та елементи даних (які називають значеннями).

7.1.5. Реалізація диспетчера об'єктів

Диспетчер об'єктів – це найважливіший компонент виконавчого рівня Windows, який надає уніфікований інтерфейс для керування ресурсами системи і структурами даних, такими як відкриті файли, процеси, потоки, сегменти пам'яті, таймери, пристрої та семафори. Навіть більш спеціалізованими об'єктами (такими як транзакції ядра, профілі, маркери безпеки і робочі столи Win32) керує диспетчер об'єктів. Об'єкти пристроїв використовують опис системи «введення – виведення» (включаючи зв'язок між простором імен NT і томами файлової системи). Диспетчер конфігурації використовує об'єкт типу **Key** для зв'язку з розділами реєстру. Сам диспетчер об'єктів має такі об'єкти, які він використовує для керування простором імен NT і реалізації об'єктів за допомогою звичайних засобів. Це каталоги, символічні посилання, а також об'єкти «об'єкт – тип».

Однаковість, яку забезпечує диспетчер об'єктів, має різні аспекти. Усі ці об'єкти використовують один і той самий механізм для створення, знищення та обліку в системі квот. До всіх цих об'єктів можна звертатися із процесів режиму користувача за допомогою описувачів. Існує уніфікована угода для керування вказівниками, що посилаються на об'єкти з ядра. Об'єктам можна давати імена у просторі імен NT (яким керує диспетчер об'єктів). Об'єкти диспетчеризації (які починаються зі звичайної структури даних для сигналізації подій) можуть використовувати звичайні інтерфейси синхронізації і повідомлення (на кшталт *WaitForMultipleObjects*). Є звичайна система безпеки з використанням списків керування доступом (ACL), що є обов'язковою для об'єктів, які відкриваються по імені об'єктів, а також перевірки доступу при кожному

використанні описувача. Є навіть засоби (для трасування використання об’єктів) для допомоги розробникам режиму ядра у налагодженні програм.

Щоб зрозуміти об’єкти, треба засвоїти, що (*виконавчий*) **об’єкт** – це просто **структура даних у віртуальній пам’яті, яка доступна режиму ядра**. Ці структури даних зазвичай використовують для подання більш абстрактних концепцій. Наприклад, об’єкти файлів виконавчого рівня створюються для кожного екземпляра файлу файлової системи, який був відкритий, об’єкти процесів створюються для подання кожного процесу.

Оскільки об’єкти є всього лише структурами даних ядра, то під час перезавантаження (або збою) системи усі об’єкти втрачаються. Коли система завантажується, у ній зовсім немає об’єктів (навіть дескрипторів типів об’єктів).

Усі типи об’єктів (і самі об’єкти) повинні створюватися динамічно іншими компонентами виконавчого рівня (через виклик інтерфейсів, які надаються диспетчером об’єктів). Після створення об’єктів і вказівки імені на них можна посилатися через простір імен NT. Тому побудова об’єктів у міру завантаження системи також створює і простір імен NT.

Об’єкти мають структуру (рис. 7.2). Кожний об’єкт містить заголовок з певною інформацією, загальною для всіх об’єктів усіх типів. Поля цього заголовка містять: ім’я об’єкта, каталог об’єкта (в якому він міститься у просторі імен NT), а також вказівник на дескриптор безпеки, який представляє список керування доступом (ACL) для об’єкта.

Виділена для об’єктів пам’ять береться з однієї із двох куп (рос. – куча) (або пулів) пам’яті, які підтримуються виконавчим рівнем.

Купа (англ. *Heap*) – назва структури даних, за допомогою якої реалізована динамічно розподілена пам’ять програми. Розмір купи – розмір пам’яті, виділений ОС для зберігання купи (під купу). Під час запуску процесу ОС виділяє пам’ять для розміщення купи. Надалі пам’ять для купи (під купу) може виділятися динамічно. Програма користувача,

застосовуючи функції, подібні *malloc()*, може отримувати вказівники на області пам’яті, що належать купі. Програми використовують купу для розміщення динамічно створюваних структур даних.

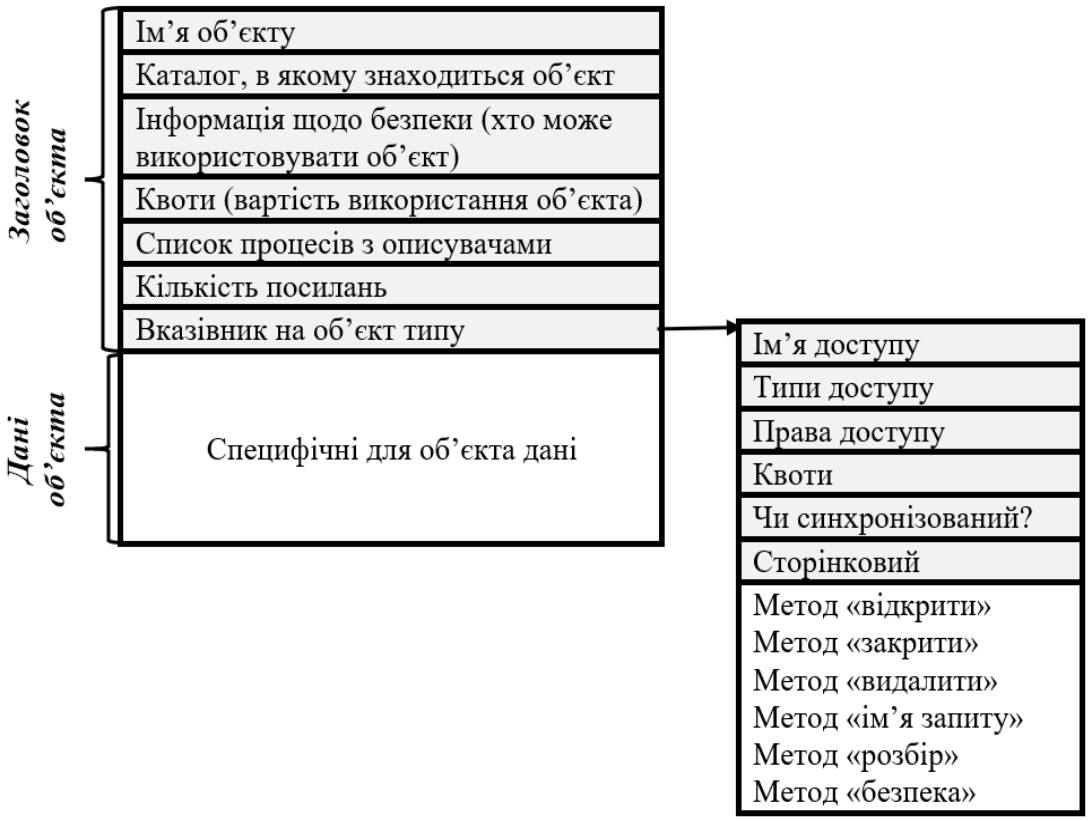


Рис. 7.2. Структура виконавчого об’єкта,
керованого диспетчером об’єктів

Службові функції (типу *malloc*) виконавчого рівня дозволяють компонентам режиму ядра виділяти або сторінкову пам’ять ядра, або безсторінкову пам’ять ядра.

Безсторінкова пам’ять потрібна для будь-якої структури даних або об’єкта режиму ядра, до якого потрібно звернутися з рівня пріоритету процесора номер 2 (або більш високого).

Більша частина виділень пам’яті диспетчером купи ядра здійснюється за допомогою довідкових списків, які містять списки (типу LIFO «*last in – first out*» – *стек*) виділених блоків однакового розміру.

Стек – це область оперативної пам’яті, яка створюється для кожного потоку. В цю область програміст записує для тимчасового зберігання змінні, їх адреси, адреси масивів даних.

Відмінність стеку від купи полягає у тому, що купа не працює за принципом стеку, купа – це сховище пам’яті, розміщене в оперативному запам’ятовувальному пристрої, яке допускає динамічне виділення пам’яті та є просто складом для ваших змінних.

Ці списки типу LIFO оптимізуються для операцій без блокувань, що покращує продуктивність і масштабованість системи.

Кожний заголовок об’єкта має поле квоти, в якому міститься «плата», що стягується з процесу за відкривання об’єкта. Квоти застосовують для того, щоб користувач не використав занадто багато системних ресурсів. Є окремі ліміти на безсторінкову пам’ять ядра (яка вимагає виділення як фізичної пам’яті, так і віртуальних адрес ядра) і сторінкову пам’ять ядра (що використовує віртуальні адреси ядра). Коли сумарна плата за будь-який із типів пам’яті досягає значення ліміту квоти, наступні спроби виділення пам’яті для певного процесу закінчуються невдачею (через недостатність ресурсів). Квоти використовують також диспетчер пам’яті (для керування розміром робочого набору) і диспетчер потоків (для обмеження ступеня використання процесора).

Описувачі

Посилання користувацького режиму на об’єкти режиму ядра не можуть використовувати вказівники, оскільки їх важко перевірити. Для посилання на об’єкти режиму ядра Windows використовує **описувачі** (*handles*, які називаються **дескрипторами**). Описувачі – це неявні значення, які диспетчер об’єктів перетворює у посилання на специфічні структури даних режиму ядра, які представляють об’єкт.

На рис. 7.3 показано структуру даних таблиці дескрипторів, яка використовується для трансляції описувачів у вказівники на об’єкти.

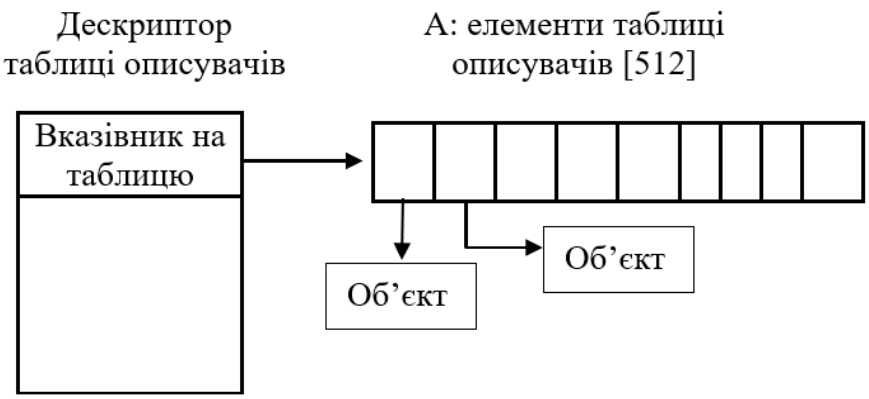


Рис. 7.3. Структури даних таблиці описувачів
(для мінімальної таблиці для однієї сторінки
з не більше ніж 512 описувачів)

Таблиця описувачів розширюється додаванням додаткових рівнів непрямого звернення. Кожний процес має власну таблицю, включаючи і системний процес, який містить усі потоки ядра, не пов’язані з процесом призначеного для користувача режиму.

7.1.6. Підсистеми середовища оточення, DLLбібліотеки підсистем, служби користувацького режиму

Операційна система Windows складається з компонентів режиму ядра і компонентів користувацького режиму. Після огляду компонентів режиму ядра розглянемо компоненти користувацького режиму. Для цього режиму у Windows особливо важливі три типи: підсистеми середовища оточення, DLL і процеси служб.

Підсистема середовища оточення призначена для надання прикладним програмам деякого піднабору базових сервісних функцій виконавчої системи Windows.

Кожна підсистема може надати доступ до різних піднаборів служб, властивих Windows. Це означає, що з додатків, побудованих на використанні однієї підсистеми, можуть виконуватися деякі дії, які не можуть виконуватися додатком, побудованим на використанні іншої підсистеми. Наприклад, додаток Windows не може використовувати

SUA-функцію *fork* (аналізатор *Standard User Analyzer* призначений для перевірки додатків і спостереження за викликами API для виявлення проблем сумісності, пов'язаних із функцією контролю облікових записів у Windows).

Кожний виконуваний образ (*.exe*) прив'язаний до однієї й тільки однієї підсистеми. Під час виконання образу код створення процесу аналізує код типу підсистеми в заголовку образу, щоб оповістити відповідну підсистему про новий процес. Код типу задається параметром компонувальника Microsoft Visual Studio /SUBSYSTEM (або за допомогою запису *Subsystem* у сторінці властивостей *Linker/System* властивостей проекту).

Оскільки програми користувачів не викликають системних функцій Windows безпосередньо, вони проходять через одну або декілька DLL-підсистем. Ці бібліотеки експортують документований інтерфейс, який може викликатися програмами, скомпонованими з цією підсистемою. Наприклад, ***DLL-бібліотеки*** підсистеми Windows (такі, як *Kernel32.dll, Advapi32.dll, User32.dll і Gdi32.dll*) реалізують функції Windows API.

У Windows функціональність системи розширюється за рахунок служб для користувацького режиму. Деякі з цих служб суворо прив'язані до роботи компонентів режиму ядра. Такий *lsass.exe*, який є локальною службою аутентифікації, керує об'єктами маркерів (які являють собою ідентифікацію користувача), а також ключами шифрування для файлової системи. Диспетчер *Plug-and-Play* призначений для режиму користувача і відповідає за визначення правильного драйвера (який потрібно використовувати в разі виявлення нового апаратного пристрою), його інсталяцію і видачу ядру вказівки про його завантаження. Багато засобів сторонніх розробників (приміром, антивіруси і засоби керування цифровими правами) реалізовані як комбінація драйверів режиму ядра і служб користувацького режиму.

У Windows **диспетчер задач *taskmgr.exe*** має вкладку, в якій вказано, які служби працюють у системі. В одному процесі (*svchost.exe)* можна побачити декілька працюючих служб. Windows робить так для багатьох служб етапу завантаження (для зменшення часу запуску системи). Служби можна комбінувати в одному процесі, якщо вони можуть безпечно працювати з однаковими атрибутами системи безпеки.

Індивідуальні служби всередині будь-якого спільно використовуваного процесу завантажуються як DLL. Вони зазвичай спільно використовують пул потоків Win32, тому для всіх резидентних служб потрібна мінімальна кількість потоків. Служби є частими джерелами вразливостей у системі, оскільки до них можна отримати віддалений доступ (це залежить від налаштувань мережевого екрану TCP/IP і налаштувань *IP Security*). Кількість постійно працюючих у Windows служб просто приголомшлива. У результаті все більше служб Windows за замовчуванням вимикається (особливо у версіях Windows Server).

7.2. Процеси і потоки у Windows

7.2.1. Базові поняття: процеси і потоки в сучасних операційних системах. Керування процесами і потоками

У сучасній ОС одночасно виконуються код ядра, який належить до різних його підсистем, і код програм користувача. При цьому відбуваються різні дії: одні програми й підсистеми виконують інструкції процесора, інші зайняті «введенням – виведенням», ще деякі очікують на запити від користувача або інших застосувань. Програма – це статична послідовність команд. Для опису виконання програмного коду в ОС використовують дві основні абстракції ОС – процеси і потоки. Процес – це фундаментальний засіб розподілу роботи і ресурсів в ОС, який дозволяє ефективно використовувати процесор.

Під *процесом* розуміють абстракцію ОС, яка об'єднує все необхідне для виконання однієї програми в певний момент часу (*процес – це програма у стадії виконання, тобто це програма та системні ресурси, які потрібні для її роботи*). Однозначна відповідність між програмою і процесом встановлюється тільки у конкретний момент часу: *один процес у різний час може виконувати код декількох програм, код однієї програми можуть виконувати декілька процесів одночасно*. Для ОС процес є засобом організації багатьох завдань, які вона повинна виконувати. Операційна система виділяє кожному процесу порцію системних ресурсів і гарантує, що програма кожного процесу буде направлятися на виконання у певному порядку та своєчасно [21].

У різних ОС процеси реалізовані по-різному. Процеси розрізняються:

– поданням (структурами даних),

– способами іменування та захисту,

– відношеннями між собою.

Зазвичай ОС містить блок коду, який керує створенням та видаленням процесів, а також відношенням між ними (M:1, 1:1, M:N). Цей код називають структурою процесу (*process structure*), реалізований як диспетчер процесів (*process manager*).

Для успішного виконання програми потрібні певні ресурси. До них належать:

– ресурси, необхідні для послідовного виконання програмного коду (передусім процесорний час);

– ресурси, що дають можливість зберігати інформацію, яка забезпечує виконання програмного коду (регістри процесора, оперативна пам'ять тощо).

Ці групи ресурсів визначають дві складові частини процесу:

– послідовність виконуваних команд процесора;

– набір адрес пам’яті (адресний простір), в якому розміщені ці команди і дані для них.

Виділення цих частин виправдане ще й тим, що в межах одного адресного простору може бути кілька паралельно виконуваних послідовностей команд, що спільно використовують одні й ті самі дані. Необхідність розмежування послідовності команд і адресного простору підводить до поняття потоку. Процес розбивається на виконувані одиниці – потоки (threads) (один і більше потоків). Потік – це більш дрібна одиниця роботи процесора. Потоки дозволяють процесу паралельно виконувати різні частини його програми та ефективно використовувати процесор, особливо на багатопроцесорних комп’ютерах.

Потік (потік керування, нитка, thread) – набір послідовно виконуваних команд процесора, які використовують загальний адресний простір процесу.

Відмінність у поняттях «процес» і «потік» полягає у тому, що для виконання потоку потрібні всі види ресурсів – процесорний час, пам’ять, файли, пристрої «введення – виведення» тощо; для виконання процесу потрібні усі наведені вище ресурси, окрім процесорного часу. Оскільки в системі може одночасно бути багато потоків, завданням ОС є організація перемикання процесора між ними і планування їх виконання. У багатопроцесорних системах код окремих потоків може виконуватися на окремих процесорах.

Можна дати ще одне визначення процесу. *Процесом називають сукупність одного або декількох потоків і захищеного адресного простору, в якому ці потоки виконуються.*

Захищеність адресного простору процесу є його найважливішою характеристикою. Код і дані процесу не можуть бути прямо прочитані або перезаписані іншим процесом; у такий спосіб їх захищають від багатьох програмних помилок і спроб несанкціонованого доступу.

На відміну від процесів, *потоки розпоряджаються загальною пам'яттю*. Дані потоку не захищені від доступу до них інших потоків за умови, що всі вони виконуються в адресному просторі одного процесу.

Захищений адресний простір процесу задає абстракцію виконання коду на окремій машині, а потік забезпечує абстракцію послідовного виконання команд на одному виділеному процесорі.

Багатопотоковість

Можливість розпаралелювання обчислень на потоки у межах процесу підвищує ефективність ОС. Механізм розпаралелювання обчислень для одного застосування називають багатопотоковою обробкою (*multithreading*). Потоки процесу мають один адресний віртуальний простір. Розпаралелювання прискорює виконання процесу за рахунок відсутності перемикання ОС з одного адресного простору на інше, яке відбувається під час виконання процесів. Програми стають логічніші. Паралелізм характерний для багатопроцесорних систем, під час виконання операцій «введення – виведення», під час взаємодії з користувачем, у розподілених системах.

Особливий ефект при цьому досягається в мультипроцесорних системах.

7.2.2. Складові елементи процесів і потоків

До елементів *процесу* належать:

– виконувана програма, яка має код і дані;

– захищений адресний простір (*address space*), тобто це набір адрес віртуальної пам'яті, який процес може використовувати;

– дані, спільні для всього процесу (ці дані можуть спільно використовувати всі його потоки);

– інформація про використання системних ресурсів (*семафори* – лічильник, який регулює кількість потоків, що використовують деякий

ресурс; *комунікаційний порт* – логічна точка в системі, через яку пересилаються повідомлення для міжпроцесного обміну; *секція* – область пам'яті, яка спільно використовується; таймер, параметр реєстру – індексний ключ для посилання на записи у базі даних конфігурації системи; відкриті файли або пристрої «введення – виведення», мережеві з'єднання тощо);

– інформація про потоки процесу.

Потік містить такі елементи:

– стан процесора (набір поточних даних із його регістрів), зокрема лічильник поточної інструкції процесора;

– стек потоку (ділянка пам'яті, де перебувають локальні змінні потоку й адреси повернення функцій, що викликані у його коді).

Потік може породити інший потік – нащадок.

Розрізняють потік користувача і потік ядра. *Потік користувача* – це послідовність виконання команд в адресному просторі процесу. Ядро ОС не має інформації про такі потоки, вся робота з ними виконується в режимі користувача. Засоби підтримки потоків користувача надають спеціальні системні бібліотеки; вони доступні для прикладних програмістів у вигляді бібліотечних функцій. Бібліотеки підтримки потоків зазвичай реалізують набір функцій, визначений стандартом POSIX (відповідний розділ стандарту називають POSIX.1b); тут говорять про підтримку *потоків POSIX*. *Потік ядра* – це послідовність виконання команд в адресному просторі ядра. Потоками ядра керує ОС, перемикання ними можливе тільки у привілейованому режимі. Є потоки ядра, які відповідають потокам користувача, і потоки, що не мають такої відповідності.

Стани процесів і потоків

Для ***потоку*** дозволені такі стани:

– *створення* (new) – потік перебуває у процесі створення;

– *виконання* (running) – інструкції потоку виконує процесор (у конкретний момент часу на одному процесорі тільки один потік може бути в такому стані);

– *очікування* (waiting) – потік очікує деякої події (наприклад, завершення операції «введення – виведення»); такий стан називають також заблокованим, а потік – припиненим;

– *готовність* (ready) – потік очікує, що планувальник перемкне процесор на нього, при цьому він має всі необхідні йому ресурси, крім процесорного часу;

– *завершення* (terminated) – потік завершив виконання (якщо при цьому його ресурси не були вилучені із системи, він переходить у додатковий стан – стан зомбі).

У деяких системах, коди і дані можуть відразу не поміщатися у пам’ять, а переписуються у спеціальну область диска – область підкачки.

Можливі переходи між станами зображено на рис. 7.4.

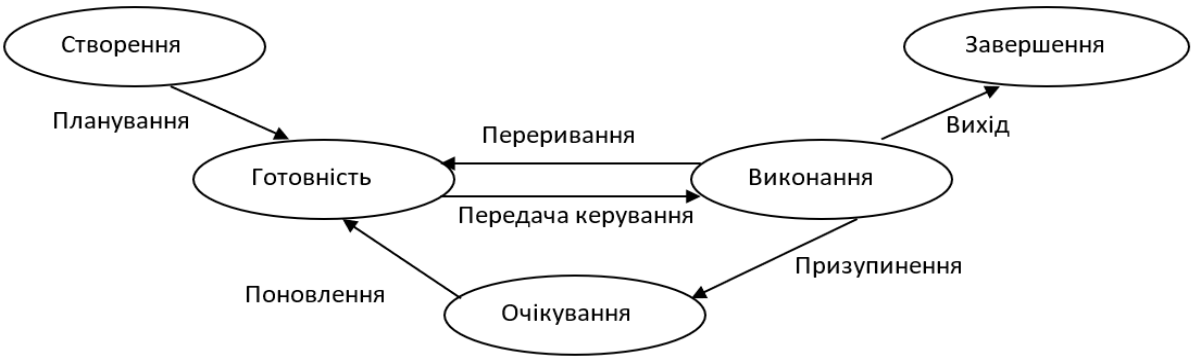


Рис. 7.4. Стани потоку

Перехід потоків між станами очікування і готовності реалізовано на основі *планування завдань* або *планування потоків.* Під час планування потоків визначають, який із потоків треба відновити після завершення операції «введення – виведення», як організувати очікування подій у системі.

Для здійснення переходу потоків між станами готовності та виконання необхідне *планування процесорного часу.* На основі алгоритмів такого планування визначають, який із готових потоків потрібно виконувати у конкретний момент, коли потрібно перервати виконання потоку, щоб перемкнутися на інший готовий потік тощо.

Перехід потоків між станами очікування і готовності реалізовано на основі *планування завдань* або *планування потоків.* Під час планування потоків визначають, який із потоків треба відновити після завершення операції «введення – виведення», як організувати очікування подій у системі.

Для здійснення переходу потоків між станами готовності та виконання необхідне *планування процесорного часу.* На основі алгоритмів такого планування визначають, який із готових потоків потрібно виконувати в конкретний момент, коли потрібно перервати виконання потоку, щоб перемкнутися на інший готовий потік тощо.

Опис процесів і потоків

Одним з основних завдань ОС є розподіл ресурсів між процесами і потоками. Такими ресурсами є, насамперед, процесорний час (його розподіляють між потоками під час планування), засоби «введення – виведення» й оперативна пам’ять (їх розподіляють між процесами).

Для керування розподілом ресурсів ОС повинна підтримувати структури даних, які містять інформацію, що описує процеси, потоки і ресурси. До таких структур даних належать:

– **таблиці розподілу ресурсів**: таблиці пам’яті, таблиці «введення – виведення», таблиці файлів тощо;

– **таблиці процесів і таблиці потоків**, де міститься інформація про процеси й потоки, наявні у системі у конкретний момент.

Керуючі блоки процесів і потоків

Інформацію про процеси і потоки в системі зберігають у спеціальних структурах даних, які називають керуючими блоками процесів та керуючими блоками потоків. Ці структури дуже важливі для роботи ОС, оскільки на підставі їх інформації система здійснює керування процесами і потоками.

Керуючий блок потоку (*Thread Control Block*, ***TCB***) відповідає активному потоку, тобто тому, який перебуває у стані готовності, очікування або виконання. Цей блок може містити таку інформацію:

– ідентифікаційні дані потоку (зазвичай його ***унікальний ідентифікатор***);

– стан процесора потоку: ***користувацькі регістри процесора, лічильник інструкцій, покажчик на стек***;

– інформацію для ***планування*** по*т*оків.

Таблиця потоків – це зв'язний список або масив керуючих блоків потоку. Вона розміщена в захищеній області пам'яті ОС.

Керуючий блок процесу (Process Control Block, ***PCB***) відповідає процесу, що є у системі. Такий блок може містити:

– ідентифікаційні дані процесу (унікальний **ідентифікатор**, інформацію про інші процеси, пов'язані з даним);

– інформацію про потоки, які виконуються в адресному просторі процесу (наприклад, **покажчики на їх керуючі блоки**);

– інформацію, на основі якої можна визначити **права процесу на використання** різних ресурсів (наприклад, ідентифікатор користувача, який створив процес);

– інформацію з розподілу адресного простору процесу;

– інформацію про ресурси «введення – виведення» та файли, які використовує процес.

Образи процесу і потоку

Сукупність інформації, що відображає процес у пам’яті, називають *образом процесу* (process image), а всю інформацію про потік – *образом потоку* (thread image).

До *образу процесу* належать:

– керуючий блок процесу;

– програмний код користувача;

– дані користувача (глобальні дані програми, загальні для всіх потоків);

– інформація образів потоків процесу.

Програмний код користувача, дані користувача та інформація про потоки завантажуються в адресний простір процесу. Образ процесу зазвичай не є неперервною ділянкою пам’яті, його частини можуть вивантажуватися на диск.

До *образу потоку* належать:

– керуючий блок потоку;

– стек ядра (стек потоку, який використовують під час виконання коду потоку в режимі ядра);

– стек користувача (стек потоку, доступний у користувацькому режимі).

Схему розміщення у пам’яті образів процесу та його потоків зображено на рис. 7.5.

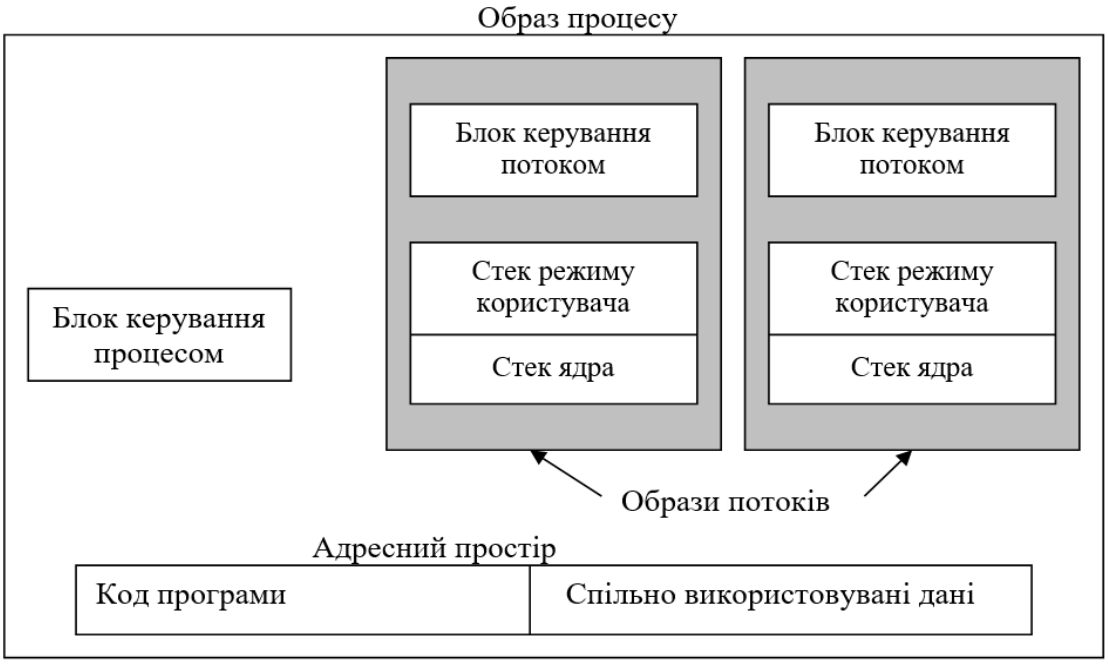


Рис. 7.5. Образи процесу та його потоків

Організація перемикання контексту

Кожний процес характеризується *контекстом* (набір регістрів, які потрібні процесу, файлів із даними, режими роботи процесора) та *дескриптором* (набір описувачів – ім’я процесу, його стан, привілеї, місце знаходження).

Для організації керування процесами і потоками найважливішим завданням ОС є передати керування від одного потоку до іншого зі збереженням стану процесора. Це називають перемикання контексту. Для перемикання контексту слід виконати такі операції:

– зберегти стан процесора потоку в деякій ділянці пам’яті (області зберігання стану процесора потоку);

– визначити, який потік слід виконувати наступним;

– завантажити стан процесора цього потоку з його області зберігання;

– продовжити виконання коду нового потоку.

Перемикання контексту здійснюється за допомогою засобів апаратної підтримки (використовується спеціальна ділянка пам’яті – сегмент стану задачі TSS).

7.2.3. Фундаментальні концепції – процеси і потоки у Windows

У Windows процеси є контейнерами для програм.

Контейнери – це технологія упаковки та запуску додатків Windows і Linux у різних локальних середовищах та у хмарі. Контейнери надають невимогливе до ресурсів ізольоване середовище, яке спрощує розроблення, розгортання і керування додатками. Контейнери містять віртуальний адресний простір, описувачі об'єктів режиму ядра, а також потоки.

Процес – це контейнер для набору ресурсів, які використовуються для виконання програми.

Програма – статична послідовність команд.

На верхньому рівні абстракції *процес Windows включає такі компоненти*:

• **закритий віртуальний простір** – множина адрес віртуальної пам'яті, яка може використовуватися процесом;

• **виконувана програма**, яка визначає початковий код і дані та відображається у віртуальний адресний простір процесу;

• **список відкритих дескрипторів** для різних системних ресурсів (семафорів, об'єктів синхронізації портів, файлів тощо), які доступні для всіх програмних потоків у процесі;

• **контекст безпеки** – *маркер доступу* (*access token*), який ідентифікує користувача, групи безпеки, привілеї, стан віртуалізації UAC (*User Account Control*), сеанс та обмежений стан облікового запису користувача, пов'язаний із процесом, а також ідентифікатор контейнера додатків і пов'язана з ним ізольована інформація;

• **ідентифікатор процесу** – унікальний ідентифікатор, який є частиною *ідентифікатора клієнта*;

• **принаймні один програмний потік (thread)**.

Кожний процес має системні дані користувацького режиму, які називають ***PEB*** (***Process Environment Block – блок середовища процесу***).

PEB включає список завантажених модулів (EXE та DLL), область пам’яті з рядками оточення, поточний робочий каталог, а також дані для керування купами процесу. Існують різні програми для перегляду (і зміни) процесів та інформації процесів. Для перегляду інформації про процеси найчастіше використовують диспетчер завдань (*Task Manager*). Диспетчер завдань можна запустити чотирма способами:

• Натиснути Ctrl+Shift+Esc.

• Клацнути правою кнопкою миші на панелі завдань і вибрати команду *Диспетчер задач (Start Task Manager).*

• Натиснути Ctrl+Alt+Del і клацнути на кнопці *Запустить диспетчер задач (Start Task Manager).*

• Запустити виконуваний файл (*Taskmgr.exe*).

Для повного продання диспетчера задач треба клацнути на кнопку *Подробнее* (*More Details*). За замовчуванням вибирається вкладка *Процессы* (*Processes*). На вкладці *Процессы* (Processes) виводиться список процесів, що складається з чотирьох стовпців: *ЦП* (*CPU*), *Пам’ять* (*Memory*), *Диск* (*Disk*) і *Мережа* (*Network*).

Щоб додати у список інші стовпці, треба клацнути правою кнопкою миші на заголовку. Також доступні стовпці *Process* (*Image*) *name, ИД процесса* (*Process ID*), *Тип* (*Type*), *Стан* (*Status*), *Издатель* (*Publisher*) і *Командная строка* (*Command Line*). Деякі процеси можна додатково розгорнути з виведенням інформації про видимі вікна верхнього рівня, створені процесом.

Щоб отримати ще більше інформації про процес, треба клацнути на кнопці *Подробнее* (*Details*). Також можна натиснути правою кнопкою миші на процесі й вибрати команду *Подробнее* (*Go to Details*), щоб переключитися на вкладку *Подробности* (*Details*) та вибрати цей конкретний процес.

Деякі найважливіші стовпці:

• *Потоки (Threads)* – в цьому стовпці виводиться кількість програмних потоків у кожному процесі. Це число зазвичай не менше 1, оскільки неможливо безпосередньо створити процес, який не містить жодного потоку. Якщо у списку є процес із 0 потоків, зазвичай це означає, що процес не вдається видалити через помилки в коді драйвера.

• *Дескриптори (Handles)* – у цьому стовпці виводиться кількість дескрипторів об’єктів ядра, відкритих програмними потоками, виконуваними у процесі.

Кожний процес також містить ідентифікатор свого батьківського процесу або створеного процесу (создателя).

Для перегляду дерева процесів можна виконати команду *tlist /t.*

Створення процесу

Зазвичай процес створюється іншим процесом викликом Win32-функції **CreateProcess** (а також **CreateProcessAsUser** та **CreateProcessWithLogonW**). Створення процесу здійснюється в декілька етапів.

На **першому етапі**, що виконується бібліотекою kernel32.dll в режимі користувача, на диску відшукується потрібний файл-образ, після чого створюється об’єкт "розділ" пам’яті для його проектування на адресний простір нового процесу.

На **другому етапі** виконується звернення до системного сервісу **NtCreateProcess** для створення об’єкта "процес". Формуються блоки *EPROCESS, KPROCESS* і блок змінних оточення *PEB*. Менеджер процесів ініціалізує у блоці процесу маркер доступу (копіюючи аналогічний маркер батьківського процесу), ідентифікатор та інші поля.

На **третьому етапі** в уже повністю проініціалізованому об’єкті "процес" необхідно створити первинний потік. Це за допомогою системного сервісу **NtCreateThread** робить бібліотека kernel32.dll. Потім

kernel32.dll надсилає підсистемі Win32 повідомлення, яке містить інформацію, необхідну для виконання нового процесу. Дані про процес і потоки поміщаються, відповідно, у список процесів і список потоків цього процесу, потім встановлюється пріоритет процесу, створюється структура, яка використовується тією частиною підсистеми Win32, яка працює в режимі ядра, і т. д.

Нарешті, запускається первинний потік, для чого формують його початковий контекст і стек й виконують запуск стартової процедури потоку режиму ядра **KiThreadStartup**. Після цього стартовий код з бібліотеки C/C++ передає керування функції **main**() і запускається програма.

Завершення процесу може бути здійснено різними способами, наприклад, за допомогою функцій **ExitProcess**, **TerminateProcess**. Однак єдиним способом, що гарантує коректну очистку всіх ресурсів, є повернення керування вхідний функції первинного потоку.

Внутрішній устрій процесів в ОС Windows

Кожний процес Windows проданий структурою **EPROCESS** (*Executive Process, виконавчий процес*).

Крім багатьох атрибутів, які стосуються процесу, EPROCESS містить ряд інших взаємопов'язаних структур даних і вказівників на них. Наприклад, кожний процес містить один або кілька потоків, кожний з яких проданий структурою **ETHREAD** (Executive Thread).

Структура *EPROCESS* та більшість пов'язаних із нею структур даних існують у системному адресному просторі. Винятком є тільки блок *PEB* (*Process Environment Block,* процес навколишнього середовища), який існує в адресному просторі процесу (користувацькому), тому що він містить інформацію, доступну для коду користувацького режиму. Крім того, деякі структури даних процесу, використовувані для керування пам'яттю (наприклад, список робочих наборів), дійсні лише у контексті

поточного процесу, оскільки зберігаються в системному просторі, що належить процесу.

Для кожного процесу, що виконує програму Windows, процес підсистеми Windows (*Csrss*) підтримує паралельну структуру з іменем **CSR_PROCESS**.

Крім того, частина підсистеми Windows, що належить до режиму ядра (Win32k.sys), підтримує структуру даних рівня процесу W32PROCESS, яка створюється при першому виклику з потоку функції Windows USER або GDI (*Graphics Device Interface*), реалізованої в режимі ядра. Це відбувається відразу ж після завантаження бібліотеки User32.dll. Типові функції, які ініціюють завантаження цієї бібліотеки, – *CreateWindow(Ex*) і *GetMessage*.

Об’єкт *PROCESS* має декілька важливих структур *EPROCESS*, KPROCESS (*Kernel Process*) та *PEB.*

Ключові поля структури EPROCESS зображено на рис. 7.6, структури KPROCESS – на рис. 7.7, а структури PEB – на рис. 7.8.

Блок PEB розміщується в адресному просторі користувача режиму описуваного ним процесу. Він містить інформацію, необхідну для завантажувача образів, диспетчера купи та інших компонентів Windows, які повинні звертатися до нього з призначеного для користувача режиму; надавати доступ до всієї цієї інформації через системні виклики було б занадто витратно.

Структури EPROCESS та KPROCESS доступні тільки з режиму ядра.

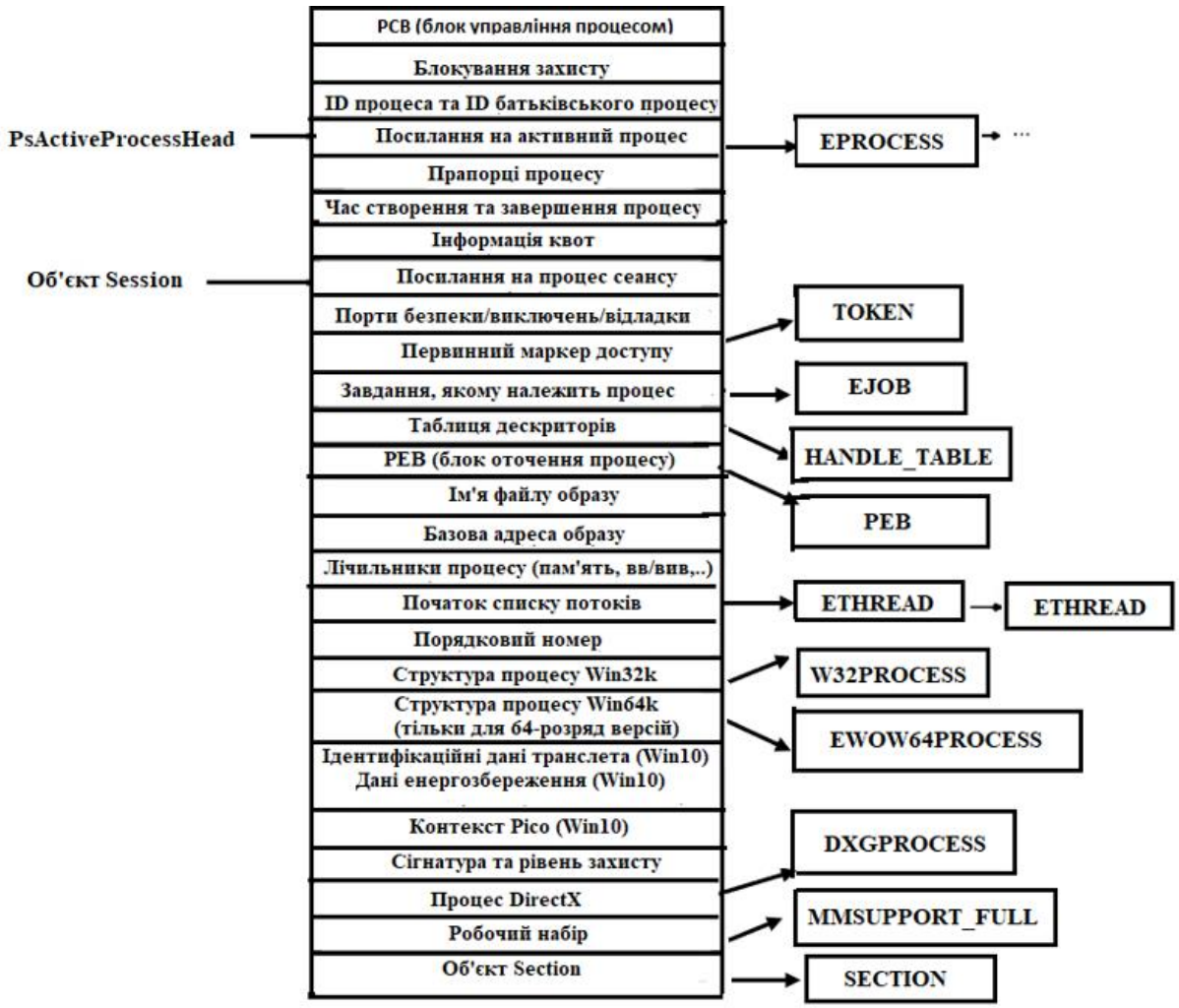


Рис. 7.6. Ключові поля структури *EPROCESS*

Рис. 7.7. Ключові поля структури *KPROCESS*

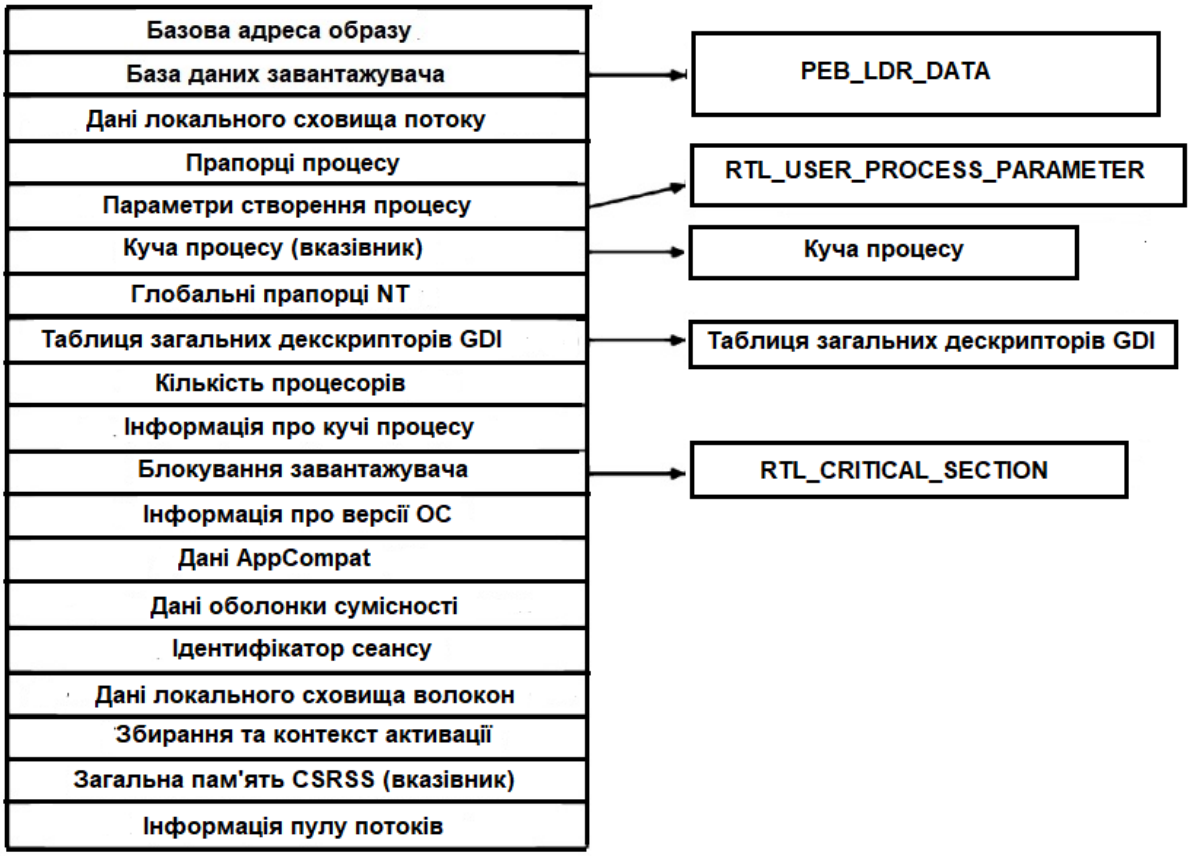


Рис. 7.8. Ключові поля структури PEB

Потоки

Програмний потік (або просто *потік*) – послідовність команд всередині процесу, яка планується Windows для виконання. Без потоків програма процесу виконуватися не зможе. Потік містить такі найважливіші компоненти:

• *Вміст набору регістрів CPU*, що подають стан процесора.

• *Два стека*: один для програмного потоку, який має використовуватися під час виконання в режимі ядра, другий – для виконання в користувацькому режимі.

• Закрита область пам’яті, яку називають *локальною пам’яттю потоку команд* (*TLS, Thread-Local Storage*); використовується підсистемами, бібліотеками часу виконання і динамічними бібліотеками DLL.

• Унікальний ідентифікатор – *ідентифікатор потоку* (*TID, Thread ID*); є частиною внутрішньої структури ідентифікатора клієнта (*client ID*). Ідентифікатори процесу і потоку генеруються з одного простору імен, тому вони ніколи не перекриваються.

Крім того, потоки іноді мають власний контекст безпеки, який часто використовується багатосерверними додатками, які реалізують контекст безпеки клієнтів, що обслуговуються.

Регістри, стеки і закриту область пам'яті називають контекстом потоку. Потоки – це абстракції ядра для планування процесора у Windows. Кожному потоку присвоюється пріоритет (залежно від значення пріоритету його процесу).

Потоки можуть бути ***афінізованими*** (*affinitized*), це означає, що кожний потік виконується тільки на певних процесорах. Це допомагає паралельним програмам, які працюють на багатоядерному мікропроцесорі або декількох процесорах, розподіляти навантаження явним чином. Кожний потік має два окремі стека викликів: один для виконання в користувацькому режимі, другий для режиму ядра. Є також блок ***TEB*** (***Thread Environment Block*** – блок середовища потоку), який зберігає специфічні для потоку дані користувацького режиму, у тому числі **області зберігання для потоку** (***Thread Local Storage***) і поля для Win32, локалізації мови і культури, а також інші спеціальні поля, які були додані різними засобами.

Окрім PEB та TEB є ще одна структура даних, яку режим ядра використовує спільно з усіма процесами, – **спільно використовувані дані користувача** (**user shared data**). Це сторінка, в яку ядро може вести запис, а процеси користувацького режиму можуть з неї тільки читати. Вона містить деякі значення, які підтримуються ядром, такі як різні форми часу, інформація про версії, кількість фізичної пам'яті. Ця спільно використовувана сторінка застосовується виключно для оптимізації

продуктивності, оскільки всі ці значення можна отримати і за допомогою системного виклику в режим ядра.

Оскільки перемикання виконання між потоками вимагає участі планувальника ядра, ця операція може бути досить затратною, особливо якщо два потоки часто передають керування між собою. У Windows реалізовано два механізми для скорочення цих витрат: *волокна (fibers) і планування призначеного для користувача режиму (UMS, User Mode Scheduling).*

Волокна (нитки)

Волокна дозволяють додаткам планувати свої потоки виконання, а не покладатися на механізм пріоритетного планування, вбудований у Windows. Волокна також часто називають «полегшеними потоками». Відносно планування волокна невидимі для ядра, тому що вони реалізуються в користувацькому режимі у *Kernel32.dll.* Щоб використовувати волокна, слід спочатку викликати функцію Windows *ConvertThreadToFiber*. Ця функція перетворює потік у працююче волокно. Надалі перетворене волокно може створювати додаткові волокна функцією *CreateFiber.* (Кожне волокно може мати власний набір волокон.) Однак, на відміну від потоків, волокно не починає виконуватися до того, як воно буде вручну вибрано викликом функції *SwitchToFiber*. Нове волокно залишиться активним, поки не завершиться або не викличе *SwitchToFiber* з вибором іншого волокна для виконання. За додатковою інформацією звертаються до опису функцій волокон в документації *Windows SDK.*

Завдання

У Windows реалізовано розширення моделі процесу – завдання. Головна функція **об’єкта завдання (job) – забезпечити можливість керування й виконання операцій із групами процесів як з єдиним цілим**. Об’єкт завдання дозволяє керувати деякими атрибутами і

встановлює обмеження для процесів, пов’язаних із завданням. Також він зберігає основну облікову інформацію для всіх процесів, пов’язаних із завданням, і для всіх завдань, які були пов’язані із завданням, але встигли завершитися. В якомусь сенсі об’єкт завдання компенсує відсутність структурованого дерева процесів у Windows, проте у багатьох відношеннях він потужніший за дерево процесів у стилі UNIX.

Завдання і волокна

Windows може об’єднувати процеси у завдання, які групують процеси, щоб обмежити потоки, які в них містяться, та використовувати спільне квотування ресурсів або ***маркер обмеженого доступу*** (***restricted token***), який не дозволяє потокам звертатися до багатьох системних об’єктів. Найважливішою властивістю завдань (у плані керування ресурсами) є те, що з того моменту, як процес виявився у завданні, усі створені (у цих процесах) потоками процеси також будуть перебувати в цьому завданні. Виходу немає. Цілком відповідно до своєї назви завдання були призначені для таких ситуацій, які швидше нагадували пакетну обробку завдань, ніж звичайні інтерактивні обчислення. Процес може перебувати всередині тільки одного завдання (максимум). У Windows завдання використовуються для угруповання процесів, що виконують програми. Процеси, які містять додаток, що виконується, мають бути ідентифіковані операційною системою, щоб вона змогла керувати всім додатком від імені користувача. На рис. 7.9 показано зв’язок між завданнями (*jobs*), процесами (*processes*), потоками (*threads*) і волокнами (*fibers*). Завдання містять процеси. Процеси містять потоки. Але потоки не містять волокон. Зв’язок між потоками і волокнами зазвичай має тип «багато-до-багатьох».

Завдання і волокна не обов’язкові: не всі процеси перебувають у завданнях або містять волокна. Волокна створюються через виділення місця у стеку і структури даних «волокна» в користувацькому режимі (для зберігання регістрів і даних, пов’язаних із цим волокном). Потоки

перетворюються у волокна, проте волокна можуть створюватися і незалежно від потоків. Такі волокна не будуть виконуватися доти, поки волокно, яке вже виконується в потоці, не викличе явно *SwitchToFiber* (для запуску волокна). Потоки можуть спробувати переключитися на те волокно, яке вже виконується, так що програміст повинен передбачити синхронізацію (щоб уникнути цього явища).

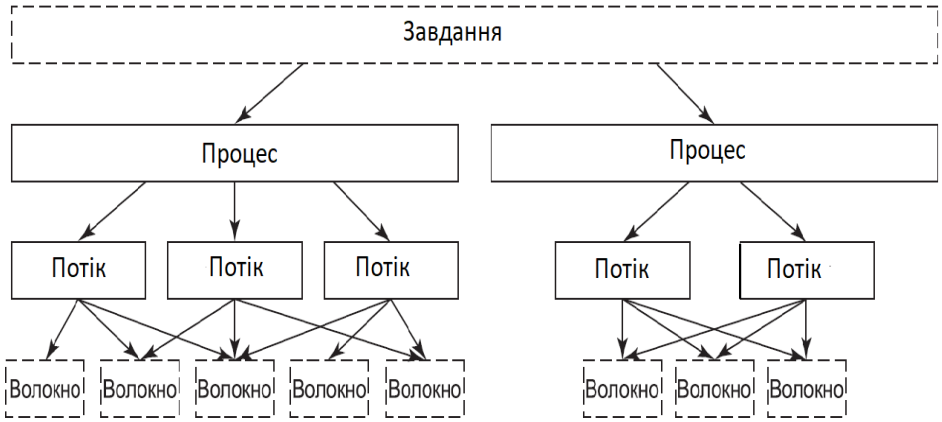


Рис. 7.9. Зв’язок між завданнями, процесами, потоками і волокнами

Основною перевагою волокон є те, що витрати перемикання між волокнами набагато нижчі, ніж перемикання між потоками. Для перемикання між потоками треба увійти в ядро і вийти з нього. Перемикання між волокнами зберігає і відновлює кілька регістрів (без усякої зміни режиму).

Для планування в режимі користувача використовують *пули потоків*. Пул потоків Win32 є надбудовою над моделлю потоків Windows, що надає більш вдалу абстракцію для певного типу програм. Ідея створення пулу полягає у тому, що для програми буде виділено обмежену кількість потоків та є підтримка черги завдань, що вимагають виконання. Як тільки потік завершить виконання завдання, він бере з черги наступне завдання. Ця модель відокремлює питання керування ресурсами (скільки процесорів доступно і скільки потоків має бути створено) від моделі програмування (що являє собою завдання та як завдання синхронізуються). У Windows це

рішення оформлено в пул потоків Win32, набір API-функцій для автоматичного керування динамічним пулом потоків й у відправлення йому завдань.

Програмісти бачать, як один Windows-потік фактично являє собою два потоки: той, що запускається в режимі ядра, і той, що запускається в режимі користувача.

Така сама модель є і в UNIX. Кожному з цих потоків виділяються його власний стек і його власна пам’ять для зберігання його регістрів, коли він не виконується. Два потоки стають одним потоком, тому що вони не працюють в один і той же час. Потік користувацького режиму працює як розширення потоку режиму ядра, що запускається тільки коли потік ядра на нього перемикається, повертаючись із режиму ядра в користувацький режим. Коли потік користувацького режиму хоче виконати системний виклик, зіткнувшись з помилкою відсутності сторінки або витісненням, система входить у режим ядра і перемикається назад на відповідний потік ядра. Зазвичай неможливо перемикатися між потоками в режимі користувача без попереднього перемикання на відповідний потік режиму ядра, перемикання на новий потік режиму ядра, а потім перемикання на його потік користувацького режиму.

Таким чином, *основні концепції, що використовуються для керування процесором і ресурсами, такі*:

• Завдання – це колекція процесів, у яких є загальні квоти і ліміти.

• Процес – це контейнер для ресурсів.

• Потік – це одиниця планування для ядра.

• Волокно – це «легкий» потік, який повністю керований у просторі користувача.

• Пул потоків – це модель програмування, орієнтована на застосування завдань.

• Потік користувацького режиму – це абстракція, що дозволяє перемикати потоки в користувацькому режимі.

Таким чином, потоки є концепцією планування, а не концепцією володіння ресурсами. Будь-який потік може звертатися до всіх об'єктів, які належать його процесу. Все, що для цього потрібно зробити, – використовувати значення описувача і зробити відповідний виклик Win32.

7.2.4. Виклики API для керування завданнями, процесами, потоками і волокнами

Нові процеси створюються за допомогою функції *CreateProcess* інтерфейсу Win32 API. Ця функція має багато параметрів і опцій.

Функція ***CreateProcess*** створює новий процес і його первинний (головний) потік. Новий процес виконує вказаний виконуваний файл у контексті безпеки та процесу виклику.

Якщо процес виклику представляє іншого користувача, новий процес використовує маркер доступу для процесу виклику, а не маркер запозичення прав. Щоб запустити новий процес у контексті системи безпеки користувача, позначеного маркером запозичення прав, використовують функцію ***CreateProcessAsUser*** або **CreateProcessWithLogonW**.

Синтаксис

BOOL CreateProcess (
  LPCTSTR *lpApplicationName*, // вказівник на ім'я виконуваного модуля
  LPTSTR *lpCommandLine,* // вказівник на командний рядок
  LPSECURITY_ATTRIBUTES *lpProcessAttributes,* // вказівник на атрибути
безпеки процесу
  LPSECURITY_ATTRIBUTES *lpThreadAttributes,* // вказівник на атрибути
безпеки потоку
  BOOL *bInheritHandles,* // вказівник на прапорець успадкування
  DWORD *dwCreationFlags,* // прапорці створення
  LPVOID lpEnvironment, // вказівник на новий блок середовища
  LPCTSTR *lpCurrentDirectory,* // вказівник на ім'я поточного каталогу
  LPSTARTUPINFO *lpStartupInfo,* // вказівник на структуру
передустановки
StartupInfo

LPPROCESS_INFORMATION *lpProcessInformation* // вказівник на структуру
ProcessInformation інформації про процес
);

Параметри (*Опис параметрів функції можна подивитися в MSDN*):

lpApplicationName – рядок може задавати повний шлях та ім'я виконуваного модуля.

lpCommandLine – може бути NULL, у цьому випадку функція використовує рядок, на яку вказує параметр *lpApplicationName*. Інакше – вказує на командний рядок.

lpProcessAttributes – вказівник на структуру SECURITY_ATTRIBUTES, яка визначає, чи буде повернений дескриптор на процес успадкованим для дочірніх процесів. Якщо цей параметр нульовий, то дескриптор не може бути успадкованим.

lpThreadAttributes – якщо нульовий, потік отримує дескриптор безпеки за замовчуванням.

bInheritHandles – вказує на те, чи буде процес успадковувати дескриптори процесу виклику.

dwCreationFlags – вказує додаткові прапорці, які контролюють клас пріоритету і створення процесу.

Значення яке повертається: якщо функція успішна, повертається ненульове значення. Якщо функція невдала, повертається нуль. Для отримання докладної інформації про помилку викликають функцію *GetLastError*.

Зауваження. Функція *CreateProcess* призначена для запуску нової програми. Функції *WinExec* та *LoadModule* також працюють для створення процесу, але не пропонують таких можливостей, як *CreateProcess*.

CreateProcess, окрім створення процесу, також створює потоковий об'єкт. Потік створюється з ініціалізованим стеком, розмір якого описаний

у заголовку образу виконуваного файлу. Потік починає виконання в точці входу цього образу.

Дескриптори нового процесу і нового потоку створюють з повними правами доступу. Для обох дескрипторів, якщо дескриптор безпеки не забезпечується, то дескриптор може бути використаний будь-якою функцією, яка запитує дескриптор об'єкта цього типу. В разі забезпечення дескриптора безпеки перевірка доступу проводиться при будь-якому запиті на використання дескриптора. Якщо перевірка не гарантує доступу, то запитуючий процес не може використовувати дескриптор цього потоку.

Приклад 7.1. *Створення процесу за допомогою функції CreateProcess*.

/ * Цей фрагмент коду слугує для запуску програми MS Word та відкриття в ній текстового файлу 1.txt за допомогою функції CreateProcess зі значенням за замовчуванням для більшості її параметрів. Для коректної роботи прикладу треба вказати правильний шлях до програми MS Word і текстового файлу 1.txt. * /

```
char *cmdline=new char [1000];
STARTUPINFO si = {sizeof (si)};
PROCESS_INFORMATION pi;
if (CreateProcess("f:\\path\\winword.exe",
                    strcpy (cmdline, "f:path\\winword.exe e:\\1.txt"),
                    NULL, NULL, FALSE, 0, NULL, NULL, & i, &pi))
      printf("\n\nEnd of work function CreateProcess\n");
else
      printf("\n\nError working function CreateProcess!!!\n");
delete [] cmdline;
```

Приклад 7.2. *Програми створення процесу*.

```
#include <windows.h>
#include <stdio.h>
void main (VOID)
{
   STARTUPINFO StartupInfo;
   PROCESS_INFORMATION ProcInfo;
   TCHAR CommandLine [] = TEXT("sleep");
    ZeroMemory (& StartupInfo, sizeof (StartupInfo));
```

```
    StartupInfo.cb = sizeof (StartupInfo);
    ZeroMemory (& ProcInfo, sizeof (ProcInfo));
     if (! CreateProcess (NULL, // Не використовується ім'я модуля
       CommandLine, // Командний рядок
       NULL, // Дескриптор процесу не успадковується.
       NULL, // Дескриптор потоку не успадковується.
       FALSE, // Установка описувачів успадкування
       0, // Немає прапорів створення процесу
       NULL, // Блок змінних оточення батьківського процесу
       NULL, // Використовувати поточний каталог батьківського процесу
       & StartupInfo, // Вказівник на структуру STARTUPINFO.
       & ProcInfo) // Вказівник на структуру інформації про процес.
      )
    printf  "CreateProcess failed.");
     // Чекати закінчення дочірнього процесу
    WaitForSingleObject (ProcInfo.hProcess, INFINITE);
     // Закрити описувачі процесу і потоку
    CloseHandle (ProcInfo.hProcess);
    CloseHandle (ProcInfo.hThread);
}
```

У наведеній програмі ім'я програми передається через другий параметр функції **CreateProcess**.

У прикл. 7.1 як дочірню програму використано найпростішу команду sleep, завдання якої – витримати паузу тривалістю 10 секунд:

```
#include <windows.h>
#include <stdio.h>
void main (VOID)
{
printf ( "Дана програма буде спати протягом 10000 мс\n ");
Sleep (10000);
}
```

Приклад 7.3.

```
#include "stdafx.h"
#include "windows.h"
#include "iostream.h"
void main()
{
```

```
    STARTUPINFO cif;
    ZeroMemory(&cif, sizeof(STARTUPINFO));
    PROCESS_INFORMATION pi;
    if (CreateProcess("c:\\windows\\notepad.exe",NULL,
            NULL,NULL,FALSE,NULL,NULL,NULL,&cif,&pi)==TRUE)
    {
            cout << "process" << endl;
            cout << "handle " << pi.hProcess << endl;
            Sleep(1000);                // почекати
            TerminateProcess(pi.hProcess,NO_ERROR);// прибрати
 процес
     }
  }
```

Функція ***CreateThread*** створює потік, який виконується в межах віртуального адресного простору процесу виклику. Щоб створити потік, який запускається у віртуальному адресному просторі іншого процесу, використовують функцію ***CreateRemoteThread.***

Синтаксис:

```
HANDLE CreateThread (
    LPSECURITY_ATTRIBUTES lpThreadAttributes, // дескриптор захисту
    SIZE_T dwStackSize,                 // початковий розмір стека
    LPTHREAD_START_ROUTINE lpStartAddress, // функція потоку
    LPVOID lpParameter,                 // параметр потоку
    DWORD dwCreationFlags,              // опції створення
    LPDWORD lpThreadId                  // ідентифікатор потоку
);
```

Параметри:

lpThreadAttributes – аргумент визначає, чи може створюваний потік бути успадкований дочірнім процесом. Ми не будемо створювати дочірні процеси, тому ставимо NULL.

dwStackSize – розмір стека у байтах. Якщо передати 0, то буде використовуватися значення за замовчуванням (1 мегабайт).

lpStartAddress – адреса функції, яка буде виконуватися потоком. Тобто можна сказати, що функція, адреса якої передається в цей аргумент,

є створюваним потоком. Ця функція повинна відповідати певному прототипу.

lpParameter – вказівник на змінну, яка буде передана в потік.

dwCreationFlags – прапорці створення. Можна відкласти запуск виконання потоку. Для запуску потоку відразу передаємо 0.

lpThreadId – вказівник на змінну, куди буде збережений ідентифікатор потоку. Якщо цей ідентифікатор не потрібний, передаємо NULL.

Приклад коду виклику CreateThread:

```
HANDLE thread = CreateThread (NULL, 0, thread2, NULL, 0, NULL);
```

Описувач потоку зберігаємо у змінну *thread*. Зверніть увагу на третій аргумент – адреса функції потоку. thread2 – ім’я функції, яка і буде другим потоком. Ось її код:

```
DWORD WINAPI thread2 (LPVOID t)
{
/ * Код другого потоку * /
return 0;
}
```

Функція потоку повинна відповідати такому прототипу:

```
DWORD WINAPI ThreadProc (LPVOID lpParameter)
```

7.3. Керування пам’яттю у Windows

7.3.1. Віртуалізація пам’яті

Оперативна пам’ять організована як одновимірний масив елементів пам’яті розміром 1 байт. Кожному байту відповідає унікальна адреса (номер), яку називають *фізичною адресою*. **Діапазон значень фізичних адрес залежить від розрядності шини адреси процесора.** Для 64-х розрядних процесорів діапазон значень адрес коливається в межах від *0* до $(2^{64} – 1)$.

Віртуальна пам’ять – метод керування пам’яттю, який реалізується з використанням апаратного і ПЗ комп’ютера [21]. Вона відображає фізичні

адреси у пам’яті комп’ютера, які використовують програми. Основна пам’ять подається у вигляді неперервного адресного простору або набору суміжних неперервних сегментів. Операційна система здійснює керування віртуальними адресними просторами із співвіднесенням оперативної пам’яті з віртуальною. Програмне забезпечення в ОС може розширити ці можливості, щоб забезпечити віртуальний адресний простір, який може перевищити обсяг оперативної пам’яті й таким чином мати більше пам’яті, ніж є в комп’ютері. Віртуальна пам’ять дозволяє модифікувати ресурси пам’яті, зробити об’єм оперативної пам’яті набагато більшим, щоб користувач, помістивши туди якомога більше програм, реально заощадив час і підвищив ефективність своєї праці. Реалізація віртуальної пам’яті стала суттєвим внеском у розвиток сучасних технологій, що полегшило роботу як професійному програмісту, так і звичайному користувачу, забезпечуючи процес більш ефективного вирішення завдань на ЕОМ.

До основних переваг віртуальної пам’яті відносять:

– позбавлення програміста від необхідності керувати загальним простором пам’яті;

– підвищення безпеки використання програм за рахунок виділення пам’яті;

– можливість мати в розпорядженні більше пам’яті, ніж це може бути фізично є на комп’ютері.

Властивості віртуальної пам’яті. Віртуальна пам’ять дозволяє програмувати додатки простіше:

– приховувати фрагментацію фізичної пам’яті;

– усувати необхідність обробки накладень в явному вигляді у програмі;

– забезпечити ізоляцію кожного процесу за рахунок створення власного виділеного адресного простору;

– надавати програмам більше пам’яті, ніж фізично встановлено у системі.

Віртуалізація пам’яті може розглядатися як узагальнення поняття віртуальної пам’яті. Майже всі реалізації віртуальної пам’яті поділяють віртуальний адресний простір на сторінки, блоки суміжних адрес віртуальної пам’яті.

Під час роботи машини з віртуальною пам’яттю використовують методи сторінкової і сегментної організації пам’яті.

Сторінкова організація пам’яті

У разі сторінкової організації всі ресурси пам’яті – як оперативної, так і зовнішньої – подані для користувача єдиним цілим. Користувач працює із загальним адресним простором і не замислюється, яка пам’ять при цьому використовується: оперативна або зовнішня, а ця спільна пам’ять носить назву віртуальної (така, що моделюється). Віртуальна пам’ять розбивається на сторінки, які містять певну фіксовану кількість комірок пам’яті. При цьому всі сторінки мають бути однакові за кількістю комірок. Типові розміри сторінок 256, 512, 1024, 2048 Байт і більше (числа кратні 256).

Переваги віртуальної пам’яті зі сторінкової організацією

- Досить великий обсяг пам’яті, який прямо адресується. Обсяг пам’яті може обчислюватися сотнями мегабайтів (і навіть гігабайтами). Розмір віртуальної пам’яті цілком залежить від обсягу накопичувача на [жорсткому] магнітному диску. Створений SWAP-файл розміщується на диску й емулює оперативну пам’ять. При цьому користувач не замислюється про те, куди буде вміщена «частина» його програми, з якої він тільки що відпрацював.
- Програми користувача можуть розміщуватися в будь-яких вільних сторінках.

- Підвищує рівень мультипрограмної роботи. З появою віртуальної пам’яті та сторінкової організації користувач отримав реальну можливість завантажувати у пам’ять більшу кількість програм для того, щоб машина обробляла програми відразу (насправді процесор встановлює пріоритет для кожної програми, що міститься у пам’яті, і далі відповідно до пріоритету виділяє певну кількість часу на реалізацію кожної програми або команди).

Недоліки віртуальної пам’яті зі сторінковою організацією

Основним недоліком віртуальної пам’яті є та кількість часу, на яку машина витрачає на звернення до зовнішньої пам’яті. Витягти необхідну інформацію з комірок оперативної пам’яті не становить особливих труднощів і великих витрат часу. Зовсім інша ситуація з диском: щоб знайти необхідну інформацію, потрібно спочатку «розкрутити» диск, потім знайти необхідну доріжку, у доріжці знайти сектор, кластер, далі зчитати побітову інформацію в оперативну пам’ять. Все це потребує часу. Крім того, при використанні методу випадкового видалення сторінок процесору може знадобитися відразу декілька сторінок, які зберігаються у зовнішній пам’яті. І це також потребує більшого часу.

- На жаль, цей недолік належить до «невиправних».
- Наявність надоперативної пам’яті.

Сегментна організація пам’яті

Сегментна організація пам’яті (*segmentation*) – схема розподілу пам’яті у вигляді сегментів змінної довжини, яка відповідає трактуванню користувача щодо розподілу пам’яті, тобто логічній структурі програм і даних. З точки зору користувача (розробника програми), програма – це набір модулів коду і даних, кожному з яких повинен відповідати свій сегмент у пам’яті. *Сегмент* – це логічна одиниця розподілу пам’яті, призначена для розміщення у пам’яті одного модуля програмного коду або даних. Наприклад, у вигляді сегментів пам’яті можуть бути представлені подані основна програма; процедура;

функція; метод; об’єкт; набір локальних змінних; набір глобальних змінних; загальний блок даних (наприклад, COMMON-блок у мові FORTRAN); стек; таблиця символів; масив.

Логічна адреса при сегментній організації пам’яті – пара:

<Segment-number, offset>,

де *segment-number – номер сегмента, offset – зміщення в сегменті*.

Таблиця сегментів слугує для відображення логічних адрес у фізичні при сегментній організації пам’яті. Кожний її елемент містить таку інформацію:

base – початкова адреса сегмента в оперативній (фізичній) пам’яті;

limit – довжина сегмента.

Сегментно-сторінкова організація віртуальної пам’яті

Цей метод організації віртуальної пам’яті спрямований на поєднання переваг сторінкового і сегментного методів керування пам’яттю. У такій комбінованій системі адресний простір користувача розбивається на ряд сегментів на розсуд програміста. Кожний сегмент, в свою чергу, розбивається на сторінки фіксованого розміру, які дорівнюються сторінці фізичної пам’яті. З погляду програміста логічна адреса в цьому випадку складається з номера сегмента і зміщення в ньому. Кожний сегмент являє собою послідовність адрес від нуля до певного максимального значення.

Відмінність сегмента від сторінки полягає в тому, що довжина сегмента може змінюватись у процесі роботи. Сегменти, як і будь-яка структура віртуальної пам’яті, можуть розміщуватися як в оперативній пам’яті, так і в зовнішній пам’яті (магнітних носіях).

Віртуальна пам’ять із сегментно-сторінковою організацією функціонує подібно віртуальній пам’яті зі сторінковою організацією: якщо на певний момент потрібного сегмента немає в оперативної пам’яті, то за потреби роботи з ним його заздалегідь переміщають в оперативну пам’ять.

Сегментно-сторінкова організація пам’яті вимагає більш складної апаратно-програмної організації.

На рис. 7.10. показано сегментно-сторінкову організацію пам’яті.

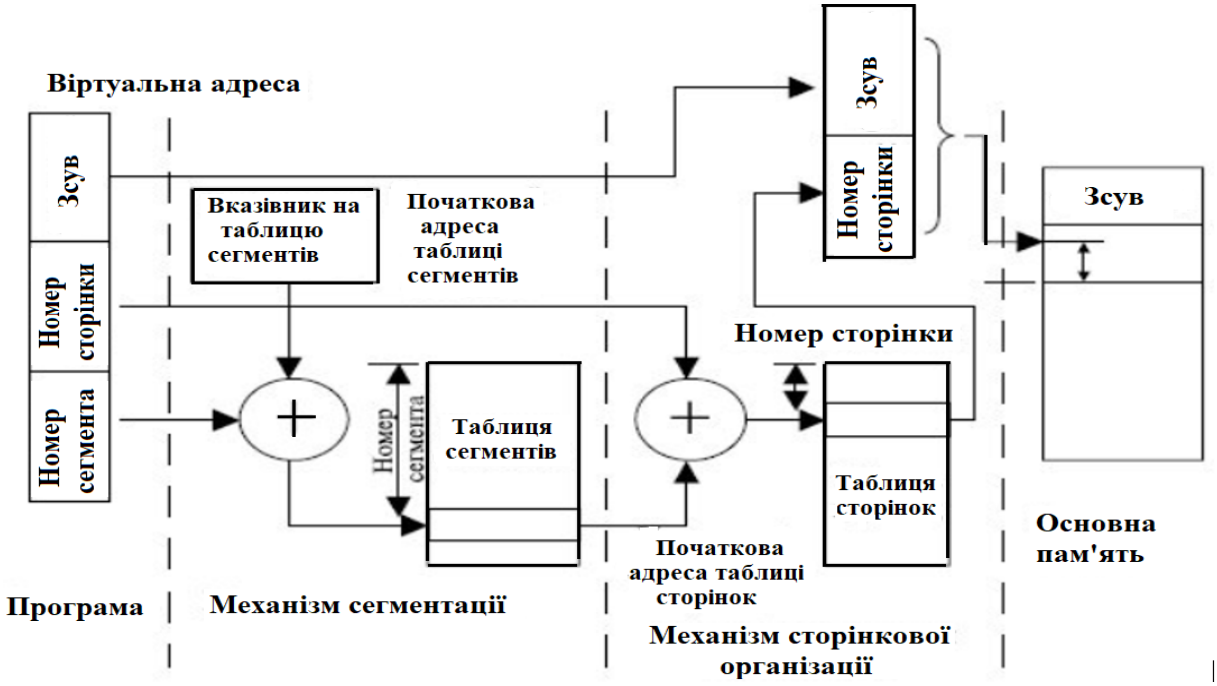


Рис. 7.10. Сегментно-сторінкова організація пам’яті [21]

Менеджер віртуальної пам’яті

Ця частина ОС створює і керує таблицями сторінок. Якщо обладнання видає помилку, Менеджер віртуальної пам’яті отримує доступ до вторинного сховища, повертає сторінку, яка має віртуальну адресу та призвела до несправності сторінки, оновлює таблиці сторінок, щоб відобразити фізичне місце перебування віртуальної адреси і вказує механізм переведення для перезапуску запиту.

Коли вся фізична пам’ять вже використовується, Менеджер віртуальної пам’яті повинен звільнити сторінки в основному сховищі для зберігання вивантажених сторінки. Використовується один з безлічі алгоритмів заміщення найменш використовуваних сторінок, щоб звільнити їх.

У Windows кожний користувацький процес має власний вір туальний адресний простір. Для комп'ютерів x86 віртуальні адреси мають довжину 32 біта, тому кожний процес має 4 Гб віртуального адресного простору, по 2 Гб користувачу та ядру. На машинах x64 і користувач, і ядро отримують більше віртуальних адрес, ніж вони зможуть доцільно використовувати їх у майбутньому. І в комп'ютерах x86, і в комп'ютерах x64 віртуальний адресний простір має заміщення сторінок за вимогою зі сторінкою фіксованого розміру – 4 Кб, але в деяких випадках застосовуються також великі сторінки розміром по 2 Мб (за рахунок використання каталогу сторінок і обходу відповідної таблиці сторінок).

Віртуальний адресний простір для трьох користувацьких процесів на комп'ютері x86 показано у спрощеній формі на рис. 7.11.

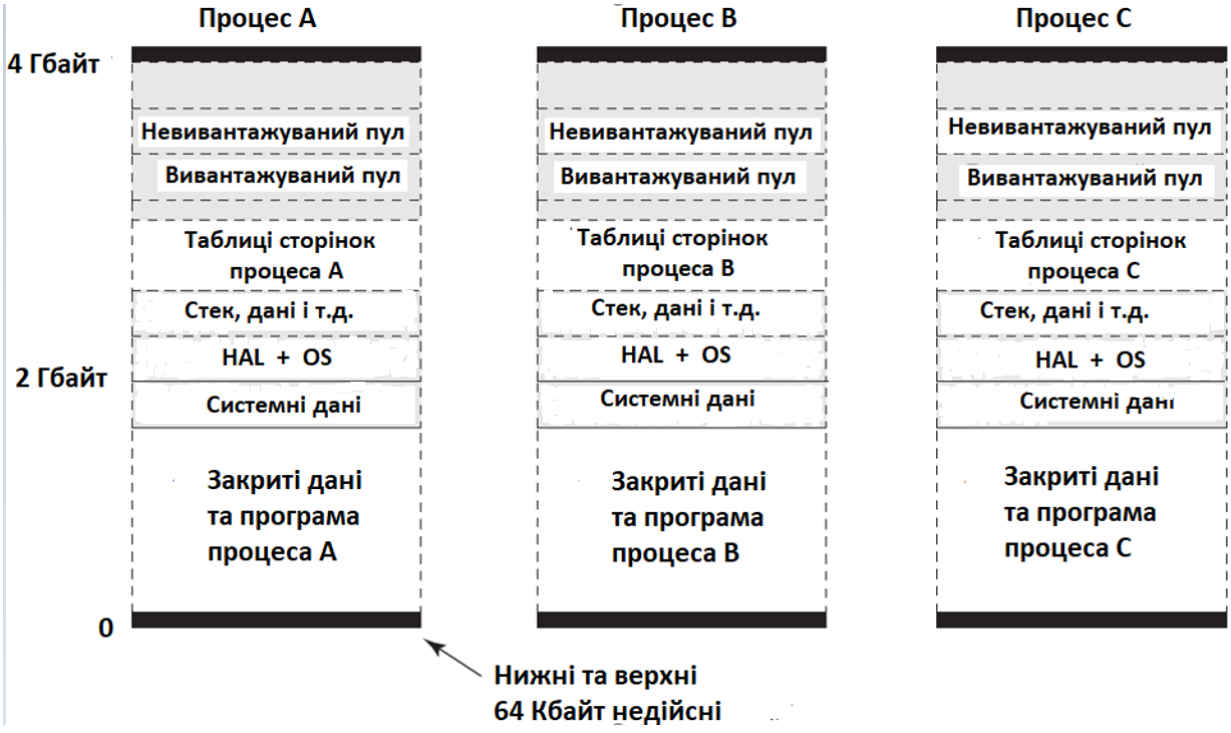


Рис. 7.11. Віртуальний адресний простір для трьох користувацьких процесів на комп'ютері x86. Білим кольором показано закриті області кожного процесу. Залиті області є загальними для всіх процесів [21]

Нижні і верхні 64 Кб віртуального адресного простору кожного процесу зазвичай нікуди не відображаються (рис. 7.11). Так було зроблено спеціально, щоб допомогти відловлювати програмні помилки і пом'якшити уразливості певного типу.

Віртуальна пам'ять ядра доступна тільки з режиму ядра. Причина спільного використання віртуальної пам'яті процесів ядром полягає в тому, що коли потік робить системний виклик, то він захоплюється в режим ядра і може продовжувати виконання без зміни карти пам'яті. Для цього треба переключитися на стек ядра потоку. Це суттєво підвищує продуктивність системи, тому що сторінки користувацького режиму процесу, як і раніше, доступні, й код режиму ядра може читати параметри і здійснювати доступ до буферів без необхідності перемикатися між адресними просторами або тимчасового дублювання сторінок в обидва простори.

Windows дозволяє потокам прикріплятися до інших адресних просторів (під час виконання в ядрі). Прикріплення до адресного простору дозволяє потоку звертатися до всього простору адрес користувацького режиму, а також до специфічних для процесу частин адресного простору ядра (таким, як карта таблиць сторінок). Перед поверненням у користувацький режим потоки повинні перемикатися назад у вихідний адресний простір.

Виділення віртуальних адрес

Сторінка віртуальних адрес може бути в одному із трьох станів: недійсна, зарезервована або зафіксована. **Недійсна сторінка** (*invalid page*) не відображається на об'єкт розділу пам'яті й посилання на неї викликає сторінкову помилку, яка призводить до порушення доступу. Після того як код або дані відображаються на віртуальну сторінку, сторінку називають *зафіксованою* (*commited*). Сторінкова помилка на зафіксованій сторінці призводить до відображення сторінки, яка була викликана з помилковою

віртуальною адресою, на одну зі сторінок, наданих об’єктом сегмента або збережених у файлі підкачки. Часто для цього необхідно буває виділити фізичну сторінку, а також виконати «введення – виведення» для файлу, поданого об’єктом сегмента (щоб зчитати дані з диска). Однак сторінкові помилки можуть виникати і тому, що потрібно оновити елемент таблиці сторінок, оскільки та фізична сторінка пам’яті, на яку він посилається, все ще *кешується* в пам’яті (в цьому випадку «введення – виведення» не потрібно). Це називають м’якою помилкою (*soft faults*).

Кеш-пам’ять є буферною пам’яттю в основній пам’яті, вона містить дані у сукупності з їх адресою в основній пам’яті. Буферна пам’ять організована як асоціативна, тобто вона дозволяє серед тих елементів, що зберігаються, вибрати дані, в яких збігаються значення розрядів з шаблоном вибірки. Такий шаблон вибірки називають ключем. Наприклад, якщо ключ має значення 1 у третьому розряді та 0 у п’ятому, то при вибірці будуть знайдені усі комірки асоціативної пам’яті, в яких третій розряд дорівнює 1, п’ятий – 0.

Звертання до такої пам’яті з використанням поля адреси як ключа дозволяє вибирати дані незалежно від їх розміщення в асоціативній пам’яті. Зберігання адрес разом із даними дозволяє однозначно ідентифікувати дані з їх адресами під час передавання між рівнями пам’яті та легко знаходити дані на будь-якому рівні пам’яті.

Таким чином, кешування – це високошвидкісний рівень зберігання, на якому потрібний набір даних, як правило, тимчасового характеру. Доступ до даних на цьому рівні здійснюється значно швидше, ніж до основного місця їх зберігання. За допомогою кешування стає можливим ефективне повторне використання раніше отриманих або обчислених даних.

Існує **ієрархічна структура пам’яті**: на верхньому рівні містяться *регістри процесора*, доступ до яких здійснюється найшвидше; далі йде *кеш-пам’ять*, обсяг якої може становити від 32 Кб до декількох

мегабайтів; потім йде *основна пам'ять*, яка може вміщувати від 16 Мб до десятків гігабайтів; далі йдуть *магнітні диски*; потім накопичувачі на магнітній стрічці та оптичні диски для зберігання архівів. *У міру просування зверху вниз по ієрархії змінюються три параметри: збільшується час доступу* – доступ до регістрів триває декілька наносекунд, до кеш-пам'яті дещо більше, до основної пам'яті – декілька десятків наносекунд, до дисків – 10 мкс, до магнітних стрічок та оптичних дисків – секунди; *зростає обсяг пам'яті* – регістри можуть містити 128 Кб, кеш-пам'ять – декілька мегабайтів, основна пам'ять – десятки тисяч мегабайттів, магнітні диски – до десятків гігабайтів, магнітні стрічки та оптичні диски зберігаються окремо від комп'ютера, тому їх сукупний обсяг обмежується фінансовими можливостями власника; *збільшується кількість бітів, які користувач отримує за долар* – вартість обсягу основної пам'яті становить декілька доларів за мегабайт, магнітних дисків – декелька центів за мегабайт, магнітної стрічки – декілька доларів за гігабайт).

Кеш-пам'ять має сукупність рядків (*cache-lines*), кожний з яких складається з фіксованої кількості адресованих одиниць пам'яті (байтів, слів), що зберігаються в основній пам'яті як комірки з послідовними адресами. Типовий розмір кеш-рядка – 16, 64, 128, 256 байт.

Віртуальна сторінка може бути також у **зарезервованому (reserved)** стані. Зарезервована віртуальна сторінка є недійсною, але ці віртуальні адреси ніколи не будуть виділятися диспетчером пам'яті для інших цілей. Наприклад, коли створюється новий потік, то багато сторінок простору стека користувацького режиму резервуються у просторі віртуальних адрес процесу, але тільки одна сторінка фіксується. У міру зростання стека диспетчер віртуальної пам'яті буде автоматично фіксувати додаткові сторінки (доти, поки зарезервований обсяг фактично закінчиться). Зарезервовані сторінки діють як сторінки захисту – вони оберігають від занадто великого зростання стека і перезапису даних інших процесів.

Резервування всіх віртуальних сторінок означає, що стек може в підсумку розростися до максимального розміру (не ризикуючи, що деякі необхідні для стека послідовні сторінки віртуального адресного простору можуть бути віддані для інших цілей). *Крім атрибутів «недійсна», «зарезервована» і «зафіксована» сторінки мають й інші атрибути, такі як «читається», «записується» і «виконується».*

Файли підкачки

Певний компроміс спостерігається при призначенні резервного сховища у зафіксованих сторінках, які не мають відповідності конкретним файлам. Ці сторінки використовують **файл підкачки** (**pagefile**). Питання полягає в тому, як і коли відображати віртуальну сторінку на конкретне місце у файлі підкачки. Стратегія могла бути такою: треба призначити кожній віртуальній сторінці сторінку в одному з файлів підкачки на диску (під час фіксації віртуальної сторінки). Це гарантує, що завжди буде відомо місце для запису кожної фіксованої сторінки (якщо її треба буде видалити з пам’яті).

Windows використовує *синхронну* (*just-in-time*) *стратегію*. Зафіксованим сторінкам, які підтримуються файлом підкачки, не виділяється місце у файлі підкачки до того моменту, коли їх необхідно витіснити у файл підкачки. Для тих сторінок, які ніколи не витісняються, дисковий простір не виділяється. Якщо сумарна віртуальна пам’ять менша, ніж наявна фізична пам’ять, то файл підкачки не потрібний зовсім. Це зручно для вбудованих систем на базі Windows. Саме так завантажується система, оскільки файли підкачки ініціалізується після запуску першого процесу користувацького режиму (*smss.exe*).

Якщо використано стратегію попереднього виділення, загальний обсяг використовуваної для закритих даних (стеки, купа, кодові сторінки копіювання при записі) віртуальної пам’яті може перевищувати обсяги файлів підкачки. Для синхронної стратегії загальний обсяг віртуальної

пам’яті може бути майже таким самим великим, як сумарний обсяг файлів підкачки і фізичної пам’яті. За наявності таких великих і дешевих (порівняно з фізичною пам’яттю) дисків економія місця не так важлива, як можливість отримання підвищеної продуктивності.

У разі підкачування на вимогу запити на читання сторінок з диска повинні запускатися відразу ж, оскільки потік, що натрапив на сторінку, якої немає, не може тривати до завершення цієї операції підкачки сторінки. Можливим способом оптимізації підкачки сторінок у пам’ять є спроба підготовки додаткових сторінок (у цій же операції «введення – виведення»). Однак ті операції, які записують на диск модифіковані сторінки, зазвичай не синхронізовані з виконанням потоків.

Синхронна стратегія виділення простору у файлі підкачки використовує це для підвищення продуктивності запису модифікованих сторінок у файл підкачки. Модифіковані сторінки групуються і записуються великими шматками. Оскільки виділення простору у файлі підкачки не виконується до запису сторінок, то кількість (необхідних для запису групи сторінок) операцій пошуку доріжки можна оптимізувати, виділяючи сторінки файлу підкачки поруч одна з одною (або навіть записуючи їх неперервною ділянкою).

7.3.2. Системні виклики керування пам’яттю

Інтерфейс прикладного програмування Win32 має кілька функцій, які дозволяють процесу явно керувати своєю віртуальною пам’яттю.

Найважливіші з цих функцій навелено в табл. 7.4. Усі вони працюють з областю, що складається або з однієї сторінки, або з двох і більше послідовних у віртуальному адресному просторі сторінок. Зрозуміло, процеси не повинні керувати своєю пам’яттю; підкачка відбувається автоматично, але ці виклики дають процесам додаткову можливість і гнучкість.

Перші чотири функції API використовують для виділення, звільнення, захисту та запиту областей віртуального адресного простору. Виділені області завжди починаються на межі 64 Кб (для мінімізації проблем при перенесенні на архітектури майбутнього, сторінки яких будуть більші за нинішні). Реальний розмір виділеного адресного простору може бути менше 64 Кб, але він має бути кратний розміру сторінки.

Таблиця 7.4

Основні функції Win32 для керування віртуальною пам'яттю у Windows

Функція Win32	Опис
VirtualAlloc	Зарезервувати або зафіксувати область
VirtualFree	Звільнити або скасувати фіксування області
VirtualProtect	Змінити захист (читання / запис / виконання) для області
VirtualQuery	Зробити запит про статус області
VirtualLock	Зробити область резидентною (відключити для неї підкачування)
VirtuelUnlock	Зробити область підкачуваною
CreateFileMapping	Створити об'єкт відображення файлу і (необов'язково) присвоїти йому ім'я
MapViewOfFile	Відобразити файл (або його частину) на адресний простір
UnmapViewOfFile	Видалити відображений файл з адресного простору
OpenFileMapping	Відкрити раніше створений об'єкт відображення файлу

Наступні два виклики API дають процесу можливість зафіксувати сторінки у пам'яті (щоб вони не витіснялися) і скасувати цю фіксацію. Такі сторінки можуть знадобитися програмі реального часу для того, щоб уникнути сторінкових помилок під час критичних операцій. Операційна система встановлює межу, щоб процеси не ставали занадто ненажерливими. Сторінки можуть вилучатися з пам'яті, але тільки в разі витіснення всього процесу. Коли він повертається назад, то всі заблоковані сторінки завантажуються знову (після чого потоки можуть починати роботу). Незважаючи на те, що це не показано в табл. 7.1, Windows має також функції власного API, які дозволяють процесу звертатися до віртуальної пам'яті іншого процесу, для якого у нього є описувач (див. табл. 7.1).

Останні чотири функції API призначені для керування файлами, відображеними у пам’ять. Для відображення файлу необхідно спочатку створити об’єкт відображення файлу за допомогою *CreateFileMapping*.

Ця функція повертає описувач для об’єкта відображення файлу (тобто об’єкта сегмента) і (необов’язково) вводить його ім’я у простір імен Win32, щоб інші процеси також могли його використовувати.

Наступні дві функції відображають і знімають відображення представлень для об’єктів сегментів (з віртуального адресного простору процесу).

Останній виклик API-процес може застосовувати для спільного використання відображення, яке було створено іншим процесом (за допомогою *CreateFileMapping*), – зазвичай це здійснюється для відображення анонімної пам’яті.

Таким чином, два або більше процеси можуть спільно використовувати області своїх адресних просторів. Ця методика дозволяє їм писати в деякі області віртуальної пам’яті один одного.

Реалізація керування пам’яттю

На процесорах x86 ОС Windows підтримує (для процесу) один лінійний адресний простір розміром 4 Гбайт (з підкачкою на вимогу). Сегментація не підтримується. Теоретично розмір сторінок може бути ступенем двійки (до 64 Кб). Для процесора x86 вони зазвичай фіксовані й сстановлять 4 Кб. Крім того, ОС може використовувати сторінки в 4 Мб для збільшення ефективності ***буфера швидкого перетворення адрес*** (*Translation Lookaside Buffer (TLB)*) у блоці керування пам’яттю процесора. Використання в ядрі та великих додатках сторінок розміром 2 Мб значно підвищує продуктивність (збільшуючи частоту успішних звернень до TLB і зменшуючи кількість проходів за таблицями сторінок для пошуку тих елементів, яких немає в TLB).

На відміну від планувальника, який вибирає потоки для виконання і не піклується про процеси, диспетчер пам’яті повністю займається процесами і не піклується про потоки.

Зрештою, саме процеси (а не потоки) володіють адресним простором, яким займається диспетчер пам’яті. Коли виділяється область віртуального адресного простору (для процесу А на рис. 7.12 виділені чотири області), диспетчер пам’яті створює *дескриптор VAD (Virtual Address Descriptor*), в якому міститься діапазон відображених адрес, секція подання файлу резервного зберігання і зміщення його відображення, а також дозволу.

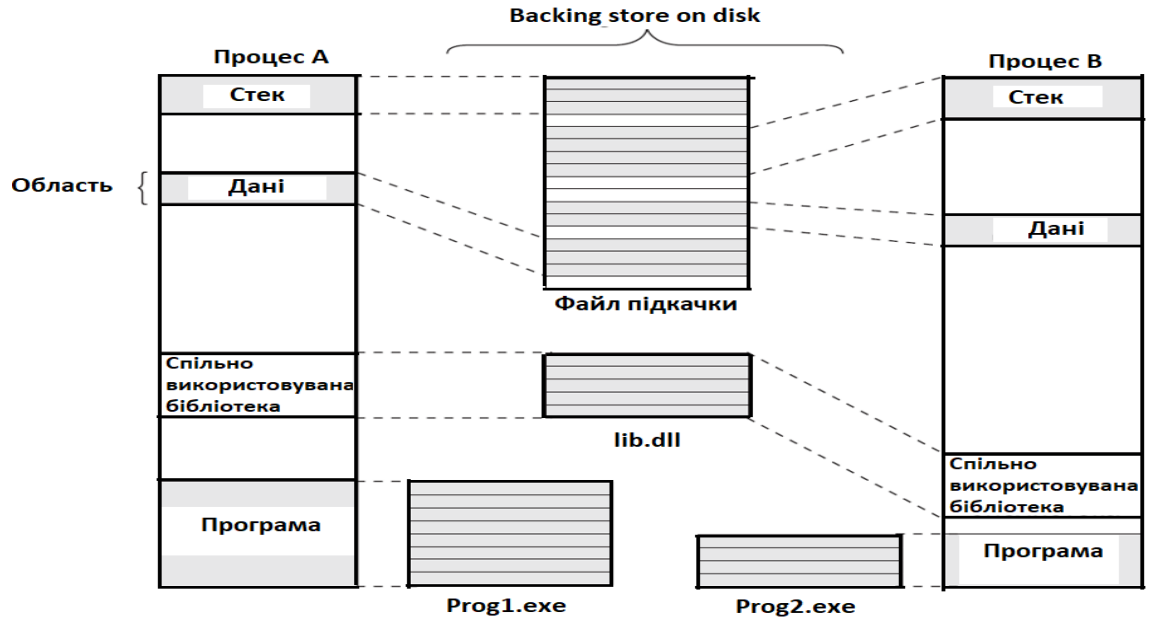


Рис. 7.12. Відображені області з їх тіньовими сторінками на диску. Файл lib.dll відображений на два адресних простори одночасно [21]

Коли зачіпається перша сторінка, створюється каталог таблиць сторінок і його фізична адреса вставляється в об’єкт процесу. Адресний простір повністю визначається списком його VAD. Вони організовані у збалансоване дерево (щоб пошук дескриптора для конкретної адреси був ефективним). Така схема підтримує розріджені адресні простори. Невикористані області (між відображеними областями) ресурсів (дискових або пам’яті) не використовують, тому вони фактично вільні.

7.3.3. Кешування у Windows

Кеш у Windows покращує продуктивність файлових систем (зберігаючи недавно і часто використовувані області файлів у пам’яті). Замість кешування фізичних адресованих блоків з диска диспетчер кешування керує віртуально адресованими блоками, тобто областями файлів. Такий підхід добре узгоджується зі структурою файлової системи NTFS. NTFS зберігає всі свої дані як файли (зокрема й метадані файлової системи). Кешовані області файлів називають *поданнями* (*views*), оскільки вони подають області віртуальних адрес ядра, які відображаються на файли файлової системи. Таким чином, реальне керування фізичною пам’яттю кеша забезпечується диспетчером пам’яті. Роль диспетчера кешування полягає в керуванні використанням віртуальних адрес ядра для подань, організації (спільно з диспетчером пам’яті) закріплення сторінок у фізичної пам’яті й надання інтерфейсів для файлових систем.

Засоби диспетчера кешування у Windows спільно використовуються всіма файловими системами. Оскільки кеш адресується віртуально (відповідно до окремих файлів), то диспетчер кешування може виконувати попередній виклик для кожного файлу окремо. Запити на доступ до кешуватися даними надходять з усіх файлових систем. Віртуальне кешування є зручним, оскільки файлові системи не повинні попередньо перетворювати зміщення у файлі у номер фізичного блоку (перед запитом кешованої сторінки файлу). Це перетворення відбувається пізніше, коли диспетчер пам’яті викликає файлову систему для звернення до сторінки на диску. Крім керування використовуваними для кешування віртуальними адресами ядра і ресурсами фізичної пам’яті, диспетчер кешування також повинен координувати свої дії з файловими системами (з таких питань, як несуперечливість подань, скидання на диск, а також правильне обслуговування відміток кінця файлу – зокрема, при розширенні файлів.

Одним з найбільш складних аспектів файлу, якими доводиться керувати файловій системі, диспетчеру кешування і диспетчеру пам’яті, є зміщення останнього байта файлу, що його називають *ValidDataLength*. Якщо програма пише за кінцем файлу, то пропущені блоки мають бути заповнені нулями, і з міркувань безпеки дуже важливо, щоб записана (у метаданих файлу) довжина *ValidDataLength* не давала доступу до неініціалізованих блоків. Тому до поновлення метаданих (новим значенням довжини) на диск потрібно записати нульові блоки. Зрозуміло, що якщо система дає збій, то деякі із блоків файлу можуть виявитися застарілими із вмісту пам’яті, проте абсолютно неприпустимо, щоб деякі блоки містили такі дані, які раніше належали іншим файлам.

Робот диспетчера кешування

Коли виконується посилання на файл, то диспетчер кешування здійснює відображення блоку (розміром 256 Кб) віртуального адресного простору ядра на файл. Якщо файл більше, ніж 256 Кб, то за один прийом відображається тільки частина файлу. Якщо у диспетчера кешування закінчуються такі блоки адрес, то перед відображенням нового файлу він повинен скасувати відображення старого файлу. Після відображення файлу диспетчер кешування може виконувати запити на його блоки (копіюючи з віртуального адресного простору ядра у буфер для користувача режиму). Якщо блоку, який копіюється, немає у фізичній пам’яті, то відбувається сторінкова помилка і диспетчер пам’яті обробляє її звичайним шляхом. Диспетчер кешування навіть не знає, був блок у пам’яті чи ні. Копіювання завжди виконується успішно.

Диспетчер кешування працює також з тими сторінками, які відображені у віртуальну пам’ять і доступ до яких здійснюється за допомогою покажчиків (а не за рахунок копіювання між буферами режиму ядра і призначеного для користувача режиму). Коли потік звертається до відображеної на файл віртуальної адреси і відбувається сторінкова

помилка, диспетчер пам’яті в багатьох випадках може обробити доступ як м’яку помилку. Йому не потрібно звертатися до диска, оскільки він бачить, що сторінка вже перебуває у фізичній пам’яті (оскільки вона відображена диспетчером кешування).

7.4. «Введення – виведення» у Windows

7.4.1. Фундаментальні концепції – «введення – виведення» у Windows

Мета диспетчера «введення – виведення» Windows – забезпечити засобами для ефективної обробки широке різноманіття пристроїв і служб «введення – виведення», підтримка автоматичного розпізнавання пристроїв та інсталяції драйверів (*Plug-and-Play*), а також для керування електроживленням пристроїв і процесора – і все це з використанням переважно асинхронної структури, яка дозволяє виконувати обчислення одночасно з передаванням даних під час «введення – виведення».

Відомо сотні тисяч пристроїв, які працюють із Windows. Для великої кількості пристроїв навіть не потрібно інсталювати драйвер, оскільки у складі ОС він уже є. Навіть з урахуванням цього є майже мільйон різних двійкових драйверів, які працюють під керуванням Windows.

Диспетчер «введення – виведення» тісно співпрацює з диспетчером Plug-and-Play (*PnP*). Диспетчер *PnP* – основний компонент, від якого залежить здатність Windows до розпізнавання змін в апаратній конфігурації. Завдяки цьому від користувача не вимагається знання тонкощів налаштування пристроїв і системи під час їх установки і видалення.

Підтримка Plug and Play вимагає взаємодії на рівнях обладнання, драйверів пристроїв та ОС. Ця підтримка у Windows базується на промислових стандартах перерахування (рос. перечисления) та ідентифікації підключених до шин пристроїв. Наприклад, стандарт USB

визначає спосіб самоідентифікації пристроїв, підключених до шини USB. На цій основі у Windows реалізуються такі можливості Plug and Play:

• Диспетчер *PnP* автоматично розпізнає встановлені пристрої, і цей процес містить перерахування пристроїв під час завантаження і виявлення їх додавання або видалення під час роботи системи.

• Диспетчер *PnP* виділяє апаратні ресурси, збираючи інформацію про вимоги пристроїв до апаратних ресурсів (переривання, діапазони адрес «введення – виведення», регістри «введення – виведення» або ресурси, які специфічні для шин). У процесі арбітражу ресурсів (*resource arbitration*) диспетчер *PnP* розподіляє ресурси між пристроями з урахуванням їх вимог. Оскільки пристрої можуть бути додані в систему після розподілу ресурсів на етапі завантаження, диспетчер *PnP* повинен вміти перерозподіляти ресурси.

• Інша функція диспетчера *PnP* – завантаження відповідних драйверів. На основі ідентифікаційних даних пристрою він визначає, чи інстальовано драйвер, який здатний керувати цим пристроєм. Якщо так, диспетчер PnP вказує диспетчеру «введення – виведення» завантажити його. Під час завантаження кожного драйвера для нього створюється **об’єкт драйвера** (*driver object).* Якщо відповідний драйвер не встановлений, диспетчер PnP режиму ядра взаємодіє з диспетчером PnP користувацького режиму, щоб встановити пристрій. При цьому він може попросити користувача вказати місцезнаходження потрібних драйверів.

• Диспетчер *PnP* також реалізує механізми, що дозволяють додаткам і драйверам виявляти зміни в апаратній конфігурації. Іноді для роботи драйверів і додатків потрібний певний пристрій, тому у Windows є засоби, які дають можливість таким драйверам і додаткам запитувати повідомлення про наявність додавання та видалення пристроїв.

Перерахування пристроїв

Основна ідея *PnP* полягає в тому, що існує перерахована (рос. перечисляемая) шина. Багато шин (у тому числі *PC Card*, *PCI*, *PCIe*, *AGP*, *USB*, *IEEE-1394*, *E-IDE* та *SATA*) були спроектовані таким чином, щоб диспетчер PnP міг послати запит на кожний роз'єм шини і попросити ідентифікуватися пристрій, який У ньому встановлений.

Диспетчер *PnP* починає перерахування пристроїв з віртуального драйвера шини з іменем *Root*, який представляє всю систему і виступає в ролі драйвера шини для драйверів, які не підтримують *Plug and Play*, і для *HAL*. *HAL* (*Hardware Abstraction Layer*, шар апаратних абстракцій) – шар абстрагування, реалізований у програмному забезпеченні, яке перебуває між фізичним рівнем апаратного забезпечення і ПЗ та запускається на цьому комп'ютері. *HAL* призначений для приховування відмінностей в апаратному забезпеченні від основної частини ядра ОС, таким чином, щоб більша частина коду, що працює в режимі ядра, не потребувала заміни в разі її запуску на системах з різним апаратним забезпеченням. На персональних комп'ютерах *HAL*, по суті, може розглядатися як *драйвер материнської плати*, що дозволяє взаємодіяти з інструкціями високорівневих мов програмування з низькорівневими компонентами, такими як апаратне забезпечення. В ОС сімейства Windows NT *HAL* є невід'ємною частиною коду, що виконується в режимі ядра, міститься в окремому завантажувальному модулі, який завантажується спільно з ядром.

HAL працює як драйвер шини, перераховує пристрої, безпосередньо підключені до материнської плати, і такі системні компоненти, як акумулятори. Визначаючи основну шину (зазвичай це PCI-шина) і пристрої типу акумуляторів і вентиляторів, *HAL* насправді покладається на опис обладнання, зафіксований програмою *Setup* у реєстрі ще під час установки ОС.

Драйвер основної шини перераховує пристрої на цій шині, при цьому він може знайти інші шини, драйвери яких ініціалізуються диспетчером PnP. Ці драйвери, у свою чергу, можуть виявляти інші пристрої, включаючи допоміжні шини. Такий рекурсивний процес – перерахування, завантаження драйвера (якщо він ще не завантажений), наступне перерахування – триває доти, поки не будуть виявлені і сконфігуровані всі пристрої у системі.

В ОС Windows усі файлові системи, антивірусні фільтри і навіть служби ядра (яким не відповідають ніякі апаратні засоби) реалізовані за допомогою драйверів «введення – виведення».

У системній конфігурації має бути вказано на необхідність завантаження хоча б деяких із цих драйверів, оскільки не існує пристрою-лічильника на шині. Інші (подібно файловим системам) завантажуються спеціальним кодом, який розпізнає їх необхідність, – наприклад, розпізнавач файлових систем визначає, до якого типу належить файлова система. Windows має функціональну можливість – це підтримка **динамічних дисків** (*dynamic disks*). Ці диски можуть поділятися на кілька розділів або на кілька дисків і можуть переконфігуровуватися на ходу, без необхідності перезавантаження. Тобто логічні диски більше не обмежені одним розділом (або навіть одним диском), так що одна файлова система може захоплювати кілька дисків. «Введення – виведення» для томів даних може фільтруватися спеціальним драйвером Windows, який реалізує **тіньові копії томів** (*Volume Shadow Copies*). Драйвер-фільтр створює моментальний знімок тому, який можна монтувати окремо і який являє собою том даних у деякий попередній момент часу. Тіньові копії важливі також для виконання точних резервних копій для серверних систем. Система працює із серверними додатками до того часу, коли настає зручний момент для виконання резервного копіювання. Коли всі програми

готові, система ініціалізує моментальний знімок тому, а потім повідомляє додаткам, що вони можуть продовжувати свою роботу.

Ще одна функціональна можливість Windows – це **підтримка асинхронного «введення – виведення»**. Потік може почати операцію «введення – виведення», а потім продовжити виконання паралельно із «введенням – виведенням». Ця функціональна можливість особливо важлива для серверів.

Є кілька способів, за допомогою яких потік може визначити завершення «введення – виведення». Перший спосіб – вказати об'єкт події в момент виконання виклику і потім чекати його. Другий – вказати чергу, в яку система помістить подію про завершення «введення – виведення». Третій – надати процедуру зворотного виклику, яку система викличе після завершення «введення – виведення». Четвертий – опитати адресу пам'яті, яку диспетчер «введення – виведення» оновлює після завершення «введення – виведення».

Останній аспект – це **пріоритетне «введення – виведення»**. Пріоритет «введення – виведення» визначається пріоритетом потоку (або його можна вказати явним чином). *Є п'ять рівнів пріоритету*: *critical* (критичний), *high* (високий), *normal* (нормальний), *low* (низький) і *very low* (дуже низький). Критичний зарезервований для диспетчера пам'яті (щоб уникнути взаємоблокувань, які в іншому випадку могли б відбутися в періоди гострої нестачі пам'яті). Низький і дуже низький пріоритети використовуються фоновими процесами (службою дефрагментації дисків або сканерами, які здійснюють пошук шпигунського ПЗ, а також пошуком по комп'ютеру).

Більшість операцій «введення – виведення» отримують нормальний пріоритет, але мультимедійні додатки можуть помітити своє «введення – виведення» високим пріоритетом (щоб уникнути проблем). Мультимедійні програми можуть також використовувати **резервування смуги**

пропускання (*bandwidth reservation*) – вони запитують гарантовану ширину смуги пропускання для звернення до критичних у сенсі тимчасових затримок файлів (на кшталт музики або відео). Система «введення – виведення» повідомить додатку оптимальний розмір передачі та кількість операцій, що очікують виконання «введення – виведення» (яке слід підтримувати для того, щоб система «введення – виведення» гарантовано досягла ширини запитуваної смуги пропускання).

7.4.2. Інтерфейси прикладного програмування «введення – виведення»

Інтерфейси системних викликів, які надаються диспетчером «введення – виведення», не дуже різняться від тих, які пропонують більшість інших ОС. Основні операції – *open*, *read*, *write*, *ioctl* та *close*, але є й інші операції: *Plug-and-Play*, *керування енергоспоживанням*, *встановлення параметрів*, *скидання системних буферів* і под. На рівні Win32 ці API розміщуються в оболонку інтерфейсів, які надають операції більш високий рівень (специфічні для конкретних пристроїв). На нижньому рівні ці оболонки відкривають пристрої і виконують основні операції. Навіть деякі операції метаданих (такі як перейменування файлу) реалізовані без специфічних системних викликів. Вони просто використовують спеціальну версію операції ***ioctl***. Це стане більш зрозуміло, якщо розглянути реалізацію стеків пристроїв «введення – виведення» і використання пакетів запитів «введення – виведення» (***I/O request packets*** (***IRP***)) диспетчером «введення – виведення».

Власні системні виклики «введення – виведення» NT (відповідно до загальної філософії Windows) мають безліч параметрів і багато варіантів. У табл. 7.5 наведено основні інтерфейси системних викликів диспетчера «введення – виведення».

NtCreateFile використовують для відкриття наявних або нових файлів. Він надає дескриптори безпеки для нових файлів, опис необхідних прав

доступу, дає автору нових файлів деякі функції керування виділенням блоків.

NtReadFile та NtWriteFile приймають описувач файлу, буфер і довжину. Вони також приймають явне зміщення файлу і дозволяють вказати ключ для доступу до заблокованих діапазонів байтів у файлі. Більша частина параметрів стосується зазначення того, які методи слід використовувати для повідомлення про завершення «введення – виведення» (можливо, асинхронного).

Таблиця 7.5

Власні виклики інтерфейсу прикладного програмування NT для виконання «введення – виведення»

Системний виклик «введення – виведення»	Опис
NtCreateFile	Відкрити новий або наявний файл чи пристрій
NtReadFile	Читати з файлу або пристрою
NtWriteFile	Писати у файл або пристрій
NtQueryDirectoryFile	Запитати інформацію про каталог (включаючи файли)
NtQueryVolumeInformationFile	Запитати інформацію про том
NtSetVolumeInformationFile	Модифікувати інформацію тому
NtNotifyChangeDirectoryFile	Завершується, коли будь-який файл каталогу або його підкаталогів буде модифікований
NtQueryInformationFile	Запросити інформацію про файл
NtSetInformationFile	Модифікувати інформацію файлу
NtLockFile	Заблокувати діапазон байтів файлу
NtUnlockFile	Зняти блокування діапазону
NtFsControlFile	Різні операції з файлом
NtFlushBuffersFile	Скинути файлові буфери на диск, які містяться у пам’яті
NtCancelIoFile	Скасувати невиконані операції «введення – виведення» для файлу
NtDeviceIoControlFile	Спеціальні операції із пристроєм

NtQueryDirectoryFile – це приклад стандартної парадигми виконавчого рівня, де є різні API для звернення (або модифікації) до інформації про певні типи об’єктів. У цьому випадку це об’єкти файлів, які посилаються на каталоги. Параметр вказує, якого типу інформації

стосується запит, – наприклад, списку назв файлів у каталозі або детальної інформації про кожний файл. Оскільки це фактично операція «введення – виведення», то підтримуються всі стандартні способи повідомлень про завершення «введення – виведення».

NtQueryVolumeInformationFile подібний операції запиту каталогу, але очікує отримання описувача файлу, який є відкритим томом і може містити файлову систему. На відміну від каталогів, для томів є параметри, які можна модифікувати, тому існує окремий виклик

NtSetVolumeInformationFile.

NtNotifyChangeDirectoryFile – це приклад парадигми NT. Потоки можуть виконувати «введення – виведення», щоб визначити, чи відбуваються з об'єктами будь-які зміни (пеерважно це каталоги файлових систем або ключі реєстру). Оскільки «введення – виведення» асинхронне, то потік повертається і продовжує виконання. Незавершений запит ставиться в чергу у файловій системі та очікує виконання операції «введення – виведення» (з використанням пакета запитів IRP).

NtQueryInformationFile – це специфічна (для файлів) версія системного виклику для каталогів. Аналогом цієї версії є системний виклик *NtSetInformationFile*. Ці інтерфейси звертаються до усіх видів інформації про імена файлів, файлові функціональні можливості (на кшталт шифрування, стиснення і розрідженості), а також інші атрибути файлів, і модифікують їх.

Ці системні виклики є, по суті, специфічною формою ***ioctl*** для файлів. Для перейменування або видалення файлу можна використовувати операцію ***set***. Їх можна також використовувати для перейменування альтернативних потоків даних у NTFS. Окремі виклики (*NtLockFile* та *NtUnlockFile*) існують для встановлення і видалення побайтових блокувань для файлів. *NtCreateFile* дозволяє обмежити доступ до всього файлу за допомогою режиму спільного використання. Альтернатива йому – виклики

блокування, які накладають обов'язкові обмеження доступу на діапазон байтів у файлі. Операції читання і записування повинні надавати ключ, що збігається із заданим у *NtLockFile* (щоб працювати із блокованими діапазонами).

Аналогічні засоби є і в UNIX, але там додатки дотримуються блокування діапазонів на свій розсуд. *NtFsControlFile* багато в чому схожа на попередні операції *query* та *set*, але є операцією більш загального типу, що націлена на обробдення специфічних для файлів операцій (які не виконуються іншими викликами). Наприклад, деякі операції специфічні для конкретної файлової системи.

І нарешті, є різноманітні виклики типу *NtFlushBuffersFile.* Подібно до виклику *sync* ОС UNIX, він змушує записати дані файлової системи на диск.

NtCancelIoFile скасовує незавершені запити «введення – виведення» для конкретного файлу.

NtDeviceIoControlFile реалізує ***операції ioctl для пристроїв***. Список операцій насправді набагато довший. Є системні виклики для видалення файлів по імені, для запиту атрибутів конкретного файлу, але це просто оболонки для інших операцій диспетчера «введення – виведення».

Є також системні виклики для роботи з **портами завершення «введення – виведення»** *(I/O completion ports)* – це засіб формування черг у Windows, який допомагає багатопотоковим серверам ефективно використовувати операції асинхронного «введення – виведення».

Реалізація «введення – виведення»

Система «введення – виведення» у Windows складається зі служб Plug-and-Play, диспетчера електроживлення, менеджера «введення – виведення», а також моделі драйвера пристроїв. Plug-and-Play виявляє зміни в конфігурації апаратного забезпечення, створює (або знищує) стеки

пристроїв (для кожного пристрою), а також завантажує і вивантажує драйвери пристроїв.

Диспетчер електроживлення налаштовує стан електроживлення пристроїв «введення – виведення», щоб зменшити споживання енергії системою, коли пристрої не використовуються.

Диспетчер «введення – виведення» надає підтримку маніпулюванню об'єктами ядра для «введення – виведення», а також операцій типу IoCallDrivers та IoCompleteRequest. *Однак більша частина роботи з підтримки «введення – виведення» у Windows реалізована в самих драйверах пристроїв.*

7.4.3. Драйвери пристроїв

Щоб гарантувати, що драйвери пристроїв добре працюють з Windows, компанія Мікрософт описала модель **WDM** (**Windows Driver Model**), якій мають відповідати драйвери пристроїв.

Є *набір розробника* (**WDK – Windows Driver Kit**), що містить документацію та приклади, які допомагають створювати драйвери, відповідні WDM. Більшість драйверів Windows починається з копіювання відповідного зразкового драйвера з WDK і його модифікації автором нового драйвера.

Компанія Microsoft також надає **верифікатор для драйверів** (***driver verifier***), який перевіряє багато дій драйвера, щоб забезпечити впевненість у тому, що він відповідає вимогам WDM (за структурою і протоколами запитів «введення – виведення», керування пам'яттю тощо).

Верифікатор поставляється разом із системою, адміністратори можуть запустити його командою ***verifier.exe***, яка дозволяє їм вказати, які драйвери будуть перевірятися і наскільки всебічними (тобто дорогими) мають бути ці перевірки.

Попри всю цю підтримку розробки та верифікації драйверів у Windows, як і раніше, дуже важко написати навіть простий драйвер, тому

компанія Мікрософт створила ***систему оболонок*** під назвою **WDF** (**Windows Driver Foundation**), яка працює поверх WDM і спрощує багато стандартних вимог (в основному пов'язаних із правильною взаємодією з керуванням електроживленням та операціями *Plug-and-Play*).

Щоб ще більше спростити написання драйверів, а також підвищити живучість системи, WDF включає інфраструктуру **UMDF** (User-Mode Driver Framework) для написання драйверів у вигляді служб, які виконуються у процесах.

Є також **KMDF** (Kernel-Mode Driver Framework) для написання драйверів у вигляді служб, що виконуються в ядрі, при цьому багато подробиць WDM реалізуються автоматично.

Пристрої у Windows представлені об'єктами пристроїв. Об'єкти пристроїв використовують для подання апаратних засобів (таких як шини), а також як програмні абстракції (на зразок файлових систем, мережевих протоколів і розширень ядра як антивірусні драйвери-фільтри). Усі вони формують те, що Windows називає *стеком пристроїв*.

Операції «введення – виведення» ініціюються диспетчером «введення – виведення», який викликає інтерфейс *IoCallDriver* виконавчого рівня із вказівниками на верхній об'єкт пристрою і на IRP (який представляє запит «введення – виведення»). Ця процедура знаходить об'єкт драйвера, пов'язаний з об'єктом пристрою. Зазначені в IRP типи операцій зазвичай відповідають описаним раніше системним викликам диспетчера «введення – виведення», таким як CREATE, READ та CLOSE.

На рис. 7.13 показано зв'язки для одного рівня стека пристроїв.

Для кожної з таких операцій драйвер має вказати точку входу. *IoCallDriver* бере тип операції з IRP, використовує об'єкт пристрою на поточному рівні стека пристроїв (для пошуку об'єкта драйвера) і шукає (за типом операції) у таблиці переходів для драйверів відповідну точку входу у драйвер.

Потім драйвер викликається і йому передається об’єкт пристрою та IRP.

Після того як драйвер завершив обробку поданого пакетом IRP запиту, у нього є три варіанти. Він може ще раз викликати *IoCallDriver*, передавши йому IRP і наступний об’єкт пристрою у стеку пристроїв. Він може оголосити запит «введення – виведення» завершеним і повернутися до його викликаючої сторони. Або він може поставити IRP у внутрішню чергу і повернутися до його викликаючої сторони (оголосивши, що запит «введення – виведення» ще не завершений). В останньому випадку здійснюється операція асинхронного «введення – виведення».

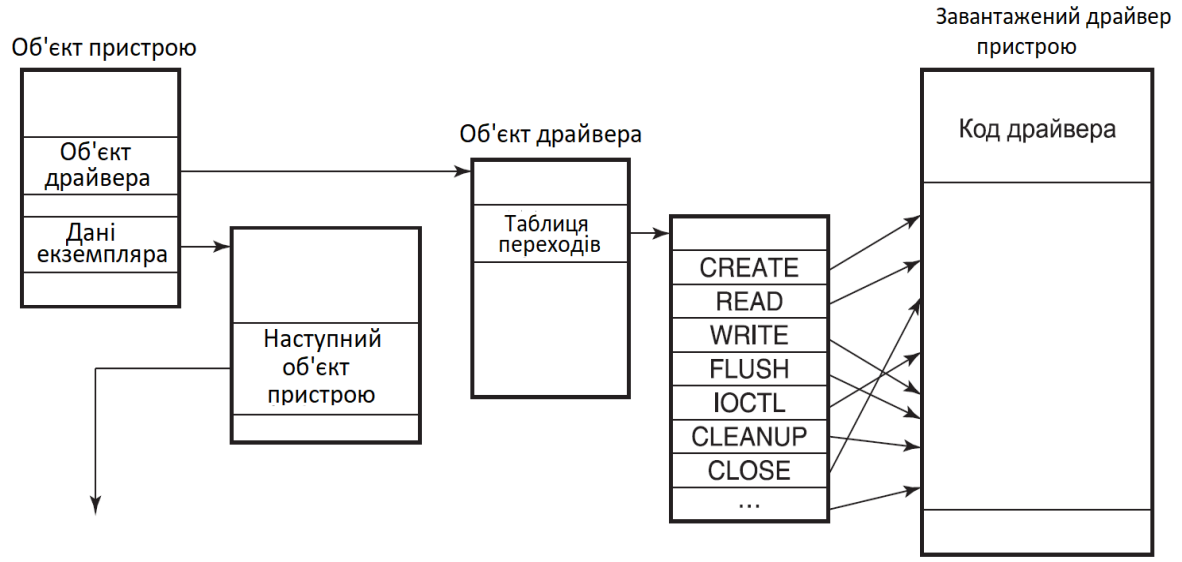


Рис. 7.13. Один рівень у стеку пристроїв [21]

Пакети запиту «введення – виведення»

На рис. 7.14 показано основні поля IRP. По суті, IRP – це масив (з динамічно змінним розміром), що містить поля, які можуть бути використані будь-яким драйвером (який оброблює запит стека пристроїв). Ці стекові поля дозволяють драйверу вказати процедуру, яку потрібно викликати під час завершення запиту «введення – виведення». Під час завершення всі рівні стека пристроїв проходяться у зворотному порядку,

при цьому по черзі викликаються усі процедури завершення (для всіх драйверів). На кожному рівні драйвер може завершити запит або прийняти рішення, що ще є якась робота (яку потрібно зробити), і залишити запит незавершеним (відклавши на цей момент завершення «введення – виведення»).

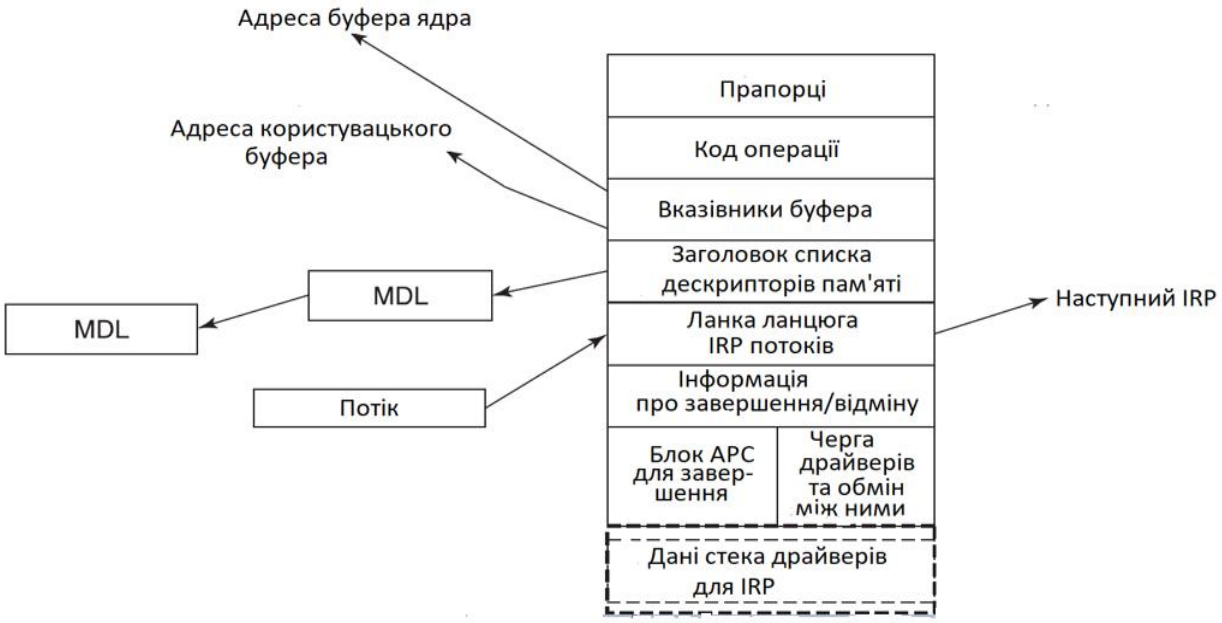


Рис. 7.14. Основні поля пакета IRP [21]

IRP містить прапорці, код операції для пошуку по таблиці переходів, вказівники для користувацького буфера і буфера ядра, а також список **MDL** (*Memory* ***Descriptor Lists***), які використовуються для опису наданих буферами фізичних сторінок (тобто для операцій DMA). Є також поля для операцій скасування і завершення.

7.4.4. Файлова система Windows NTFS

Windows підтримує кілька файлових систем, найважливішими з яких є **FAT-16**, **FAT-32** та **NTFS** (*NT File System*). FAT-16 – це стара файлова система ОС MS-DOS. Вона використовує 16-бітні дискові адреси, що обмежує розмір дискових розділів 2 Гбайт. Її переважно застосовують для доступу до флоппі-дисків (для тих клієнтів, які їх досі використовують). FAT-32 використовує 32-бітні дискові адреси і підтримує дискові розділи

розміром до 2 Тб. FAT-32 не має жодної системи безпеки, нині її фактично використовують тільки для переносних носіїв (таких як флеш-диски).

Файлову систему NTFS було розроблено спеціально для версії Windows NT. Вона істотно збільшує безпеку і функціональність Windows. NTFS використовує 64-бітні дискові адреси й теоретично може підтримувати дискові розділи розміром до 2^{64} байт.

Фундаментальні концепції

Імена файлів у NTFS обмежені 255 символами; розмір повного маршруту обмежений 32 767 символами. Імена файлів зберігаються у кодуванні *Unicode*, що дозволяє в тих країнах, де не використовується латинський алфавіт (наприклад, у Греції, Японії, Індії, Росії та Ізраїлі), писати імена файлів своєю мовою. Наприклад, *φιλε* – це допустиме ім'я файлу. NTFS повністю підтримує чутливі до регістру імена (так що Foo різниться від foo та FOO). *Інтерфейс прикладного програмування Win32 у повному обсязі підтримує чутливість до регістру імен файлів* і зовсім не підтримує її для імен каталогів. Підтримка чутливості до регістру є при роботі підсистеми POSIX (для сумісності з UNIX). Win32 не є чутливою до регістру, але зберігає регістр, так що імена файлів можуть мати у складі букви різних регістрів. Попри те, що чутливість до регістру добре знайома користувачам UNIX, вона дуже незручна для звичайних користувачів, які зазвичай не роблять таких відмінностей. Наприклад, майже весь сучасний Інтернет не має чутливості до регістру.

Файл у NTFS – це не просто лінійна послідовність байтів (як файли у FAT-32 та UNIX). Файл складається з безлічі атрибутів, кожний з яких поданий потоком байтів. Більшість файлів має кілька коротких потоків (таких як назва файлу та його 64-бітний ідентифікатор об'єкта) та один довгий (неіменований) потік із даними. Однак файл може мати також два або більш довгих потоків даних. Кожний потік має ім'я, що складається з імені файлу, двокрапки та імені потоку (як, наприклад, foo: stream1).

Кожний потік має розмір і може блокуватися незалежно відувсіх інших потоків. Ідея безлічі потоків у NTFS не нова.

Файлова система комп'ютерів *Apple Macintosh* використовує два потоки на файл (гілка даних і гілка ресурсів). Спочатку потоки в NTFS застосовували для того, щоб файловий сервер NT міг обслуговувати клієнтів *Macintosh*. Множинність потоків даних використовується також щоб подавати метадані файлів, такі як контрольні картинки зображень у форматі JPEG (які є в графічному інтерфейсі користувача Windows). Однак, на жаль, множинні потоки даних уразливі й часто губляться при перенесенні в інші файлові системи або по мережі (і навіть під час резервного копіювання і наступного відновлення, оскільки багато утиліт їх ігнорують).

NTFS – це ієрархічна файлова система, схожа на файлову систему UNIX, однак у ній роздільником компонентів імені є знак лівий слеш «\», а не правий «/» (це атавізм від MS-DOS). На відміну від UNIX, тут концепції поточного робочого каталогу, жорстких посилань на поточний каталог (**.**) і батьківський каталог (**..**) реалізовані як угоди, а не як фундаментальна частина файлової системи. Жорсткі посилання підтримуються, але використовуються тільки для підсистеми POSIX, так само як і підтримка перевірки обходу каталогів (дозвіл '*x*' в UNIX).

Символічні посилання для NTFS підтримуються. Створення символічних посилань зазвичай дозволяється тільки адміністраторам. Реалізація символічних посилань використовує функціональну можливість NTFS під назвою «***точка повторної обробки***» (*reparse points*).

Реалізація файлової системи NTFS

NTFS – дуже складна файлова система, що була розроблена спеціально для NT як альтернатива файлової системи HPFS (High Performance File System – високопродуктивна файлова система), яка була розроблена для OS/2.

Структура файлової системи

Кожний том NTFS (наприклад, дисковий розділ) містить файли, каталоги, бітові масиви й інші структури даних. Кожний том організований як лінійна послідовність блоків (які в термінології компанії Microsoft називають кластерами), причому розмір блоків для кожного тому фіксований (залежно від розміру тому він може змінюватися від 512 байт до 64 Кб). Більшість дисків NTFS використовує блоки розміром 4 Кб – це компроміс між застосуванням великих блоків (для ефективної передачі даних) і використанням маленьких блоків (для зниження внутрішньої фрагментації). Посилання на блоки роблять з використанням зміщення від початку тому (за допомогою 64-бітних чисел).

Головна структура даних кожного тому – це **MFT** (***Master File Table*** – головна таблиця файлів), яка є лінійною послідовністю записів фіксованого розміру (1 Кб). Кожний запис MFT описує один файл або один каталог. Вона містить атрибути файлу (такі як його ім’я і тимчасову мітку), а також список дискових адрес (де розміщені його блоки). Якщо файл дуже великий, то іноді доводиться використовувати два або більше записів MFT (щоб розмістити в них список всіх блоків). У цьому випадку перший запис у MFT, який називають *основним записом* (*base record*), вказує на додаткові записи у MFT.

Сама MFT також є файлом і в цій іпостасі може бути розміщена в будь-якому місці тому (таким чином усувається проблема наявності дефектних секторів на першій доріжці). Більше того, за потреби цей файл може рости (до максимального розміру 2^{48} записів).

MFT показано на рис. 7.15. Кожний запис MFT складається з послідовності пар (заголовок атрибута – значення). Кожний атрибут починається із заголовка, в якому вказано, що це за атрибут і яку довжину має його значення. Деякі значення атрибутів (такі як ім’я файлу та його дані) мають змінну довжину.

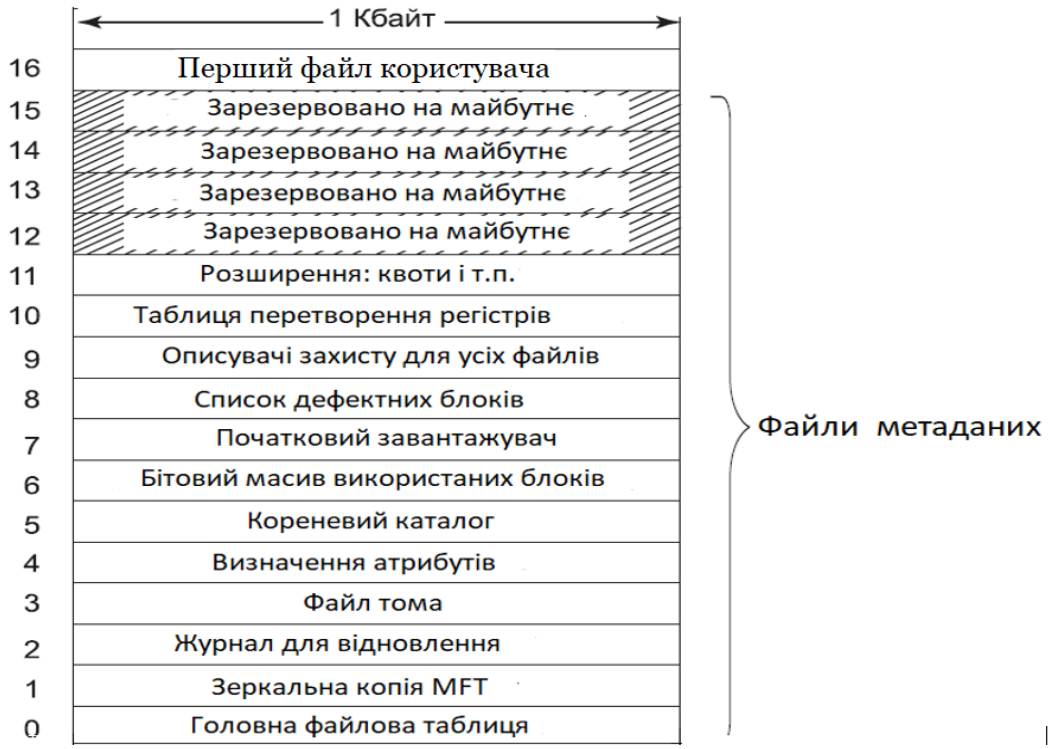


Рис. 7.15. Головна таблиця файлів NTFS [21]

Перші 16 записів MFT резервуються для файлів метаданих NTFS. Кожний із цих записів описує нормальний файл, який має атрибути і блоки даних (як і будь-який інший файл). Кожний із цих файлів має ім’я, яке починається зі знака долара (щоб позначити його як файл метаданих). Перший запис описує сам файл MFT. Зокрема, у ній йдеться про те, де містяться блоки файлу MFT (щоб система могла знайти файл MFT). Очевидно, що Windows потрібний спосіб знаходження першого блоку файлу MFT, щоб знайти іншу інформацію щодо файлової системи. Інформацію Windows дивиться у завантажувальному блоці – саме туди записується адреса першого блоку файлу MFT ппід час форматування тому.

Запис 1 є дублікатом початку файлу MFT. Ця інформація настільки цінна, що наявність другої копії може бути просто критичною (у тому випадку, якщо один із перших блоків MFT перестане читатися). Другий запис – файл журналу. Коли у файловій системі відбуваються структурні зміни (такі як додавання нового або видалення наявного каталогу), то така дія журналюється тут до його виконання (щоб підвищити ймовірність

коректного відновлення в разі збою під час операції, наприклад такого, як відмова системи). Тут також журналюються зміни у файлових атрибутах. Фактично не журналюються тут тільки зміни в даних користувача. Запис 3 містить інформацію про том (таку, як його розмір, мітка і версія). Як уже згадувалося, кожний запис MFT містить послідовність пар «заголовок атрибута – значення». Атрибути визначаються у файлі $ *AttrDef,* інформація про цей файл міститься в MFT (у запису 4). Потім йде кореневий каталог, який сам є файлом і може рости до довільного розміру. Він описується записом номер 5 у MFT.

Вільний простір тому відстежується за допомогою бітового масиву. Сам бітовий масив – теж файл, його атрибути й дискові адреси подано в запису 6 у MFT.

Файли NTFS мають пов’язаний із ними ідентифікатор, подібний до номера вузла *i-node* в UNIX. Файли можна відкривати за ідентифікатором, але ідентифікатор, який присвоюється файловою системою NTFS, не завжди можна використовувати, оскільки він заснований на запису MFT і може змінитися при переміщенні запису для цього файлу (наприклад, якщо файл відновлюється з резервної копії). NTFS дозволяє використовувати окремий атрибут «ідентифікатор об’єкта», який може бути встановлений для файлу і який немає потреби змінювати. Його можна зберегти разом із файлом (наприклад, якщо він копіюється на новий том).

Файл у NTFS має один (або декілька) пов’язаних з ним потоків даних. Саме тут і є його корисний сенс. Потік даних за замовчуванням *(default data stream*) назви не має (наприклад, *dirpath\filename::$DATA*), але альтернативні потоки даних (*alternate data stream*) мають імена, наприклад*: dirpath\filename:streamname:$DATA.* Ім’я кожного потоку (якщо він є) міститься в заголовку цього атрибута. Слідом за заголовком йде або список дискових адрес (це блоки, які містяться в потоці), або (для потоків всього в кілька сотень байтів, а таких багато) сам потік.

Виділення дискового простору

Модель відстеження дискових блоків полягає в тому, що вони виділяються послідовними ділянками, наскільки це можливо (з міркувань ефективності). Наприклад, якщо перший логічний блок потоку поміщений у блок 20 диска, то система буде дуже старатися помістити другий логічний блок у блок 21, третій логічний блок – у блок 22 і т. д. Одним зі способів досягнення безперервності цих ділянок є виділення дискового простору по кілька блоків за один раз (у міру можливості).

Блоки потоку описуються послідовністю записів, кожний з яких описує послідовність логічно суміжних блоків. Для потоку без пропусків буде тільки один такий запис. Якщо потрібно дуже багато записів MFT, то з'являється проблема: може не вистачити місця в основній MFT для розміщення всіх їх індексів. Для цієї проблеми також є рішення: список записів розширення MFT робиться нерезидентним, тобто зберігається в інших дискових блоках (замість основного запису MFT). У цьому випадку він може збільшуватися настільки, наскільки це потрібно.

Елемент MFT для невеликого каталогу показаний на рис. 7.16. Запис містить деяку кількість елементів каталогу, кожний з яких описує один файл або каталог. Кожний елемент містить структуру фіксованої довжини, за якою слідує ім'я файлу (змінної довжини). Фіксована частина містить індекс елемента MFT для певного файлу, довжину імені файлу, а також різноманітні інші поля і прапори. Пошук елемента каталогу складається з опитування всіх імен файлів по черзі. Великі каталоги використовують інший формат.

На додаток до звичайних файлів і каталогів NTFS підтримує і жорсткі посилання (в UNIX-сенсі), а також символічні посилання (з використанням механізму під назвою точка монтування – *reparse points*). NTFS підтримує позначку файлу або каталогу як точки повторної обробки та асоціювання з нею блоку даних. Коли такий файл або каталог трапляється під час

розбору імені файлу, операція закінчується невдачею і диспетчеру об’єктів повертається цей блок даних.

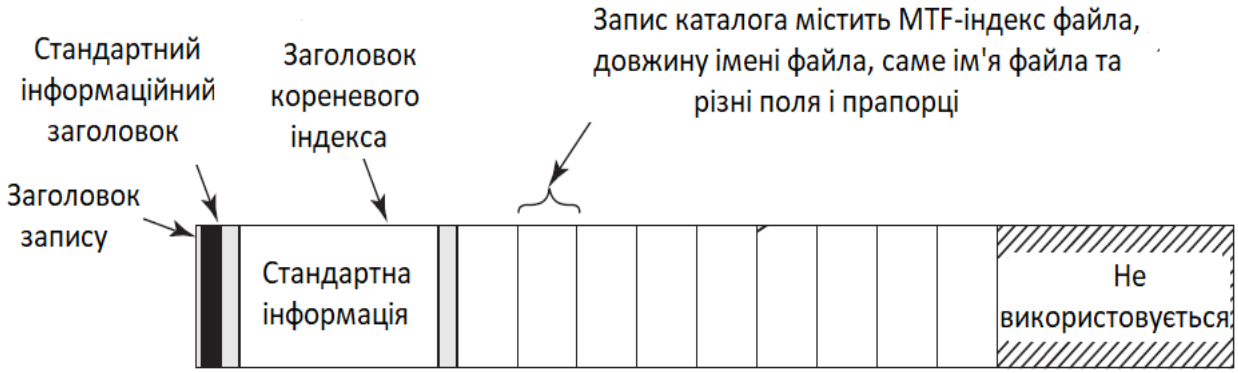


Рис. 7.16. Запис MFT для невеликого каталогу

Диспетчер об’єктів може інтерпретувати дані, які показують альтернативний маршрут, після чого він оновлює рядок для розбору і повторює операцію «введення – виведення». Цей механізм використовується для підтримання як символічних посилань, так і змонтованих файлових систем, виконуючи перенаправлення пошуку в іншу частину ієрархії каталогів або навіть до іншого розділу диска.

NTFS підтримує прозоре стиснення файлів. Файл може створюватися у стислому режимі, а це означає, що NTFS намагається автоматично стиснути блоки під час їх запису на диск й автоматично розпаковує їх при читанні назад. Ті процеси, які читають або пишуть стислі файли, абсолютно не в курсі того, що відбувається стиснення або розпакування. Стиснення працює таким чином. Коли NTFS пише файл (позначений як стиснений) на диск, то вона вивчає перші 16 (логічних) блоків файлу – незалежно від того, скільки ділянок вони займають. Потім запускає по них алгоритм стиснення. Якщо отримані дані можна записати у 15 або менше блоків, то стислі дані записуються на диск (за можливості – однією ділянкою). Якщо стислі дані, як і раніше, займають 16 блоків, то ці 16 блоків записуються у стислому вигляді. Потім досліджуються блоки 16-31, щоб дізнатися, чи можна їх стиснути до розміру 15 блоків (або менше), і т. д.

Журналювання

NTFS підтримує два механізми, за допомогою яких програми можуть виявити зміни у файлах і каталогах. Перший – це операція *NtNotifyChangeDirectoryFile*, передає системі буфер, який повертається після виявлення зміни в каталозі або підкаталозі. У результаті «введення – виведення» буфер заповнюється списком записів про зміни. Якщо він дуже маленький, записи губляться.

Другий механізм – це журнал змін NTFS. NTFS містить список усіх записів про зміни для каталогів і файлів тому у спеціальному файлі, який програми можуть читати за допомогою спеціальних операцій керування файловою системою (Опція *FSCTL_QUERY_USN_JOURNAL* виклику *NtFsControlFile*). Файл журналу зазвичай дуже великий.

Шифрування файлів

У наші дні комп’ютери використовують для зберігання найрізноманітнішої конфіденційної інформації. Windows вирішує ці проблеми за допомогою опції шифрування файлів. Для використання шифрування звичайним способом необхідно позначити потрібні каталоги як зашифровані, після чого всі файли, які містяться в них, будуть зашифровані, а нові (переносяться або створювані файли) також будуть шифруватися. Шифруванням і розшифруванням керує не файлова система NTFS, а драйвер під назвою *EFS* (*Encryption File System*), який реєструє зворотні виклики у NTFS. EFS шифрує конкретні файли і каталоги. У Windows є ще один засіб шифрування під назвою *BitLocker*, який шифрує майже всі дані тому, що може допомогти захистити дані, незважаючи ні на що – якщо тільки користувач використовує механізми сильних ключів.

7.4.5. Системні виклики для роботи з файлами

Файлова система ОС повинна надавати користувачам набір операцій для роботи з файлами, оформлений у вигляді системних викликів. У різних ОС є різні набори файлових операцій. Розглянемо системні виклики для роботи з файлами.

• *Create* (створення). Файл створюється без даних. Цей системний виклик оголошує про появу нового файлу і дозволяє встановити деякі його атрибути.

• *Delete* (видалення). Непотрібний файл видаляється, щоб звільнити простір на диску.

• *Open* (відкриття). До використання файлу його потрібно відкрити. Цей виклик дозволяє прочитати атрибути файлу і список дискових адрес для швидкого доступу до вмісту файлу.

• *Close* (закриття). Після завершення операцій з файлом його атрибути і дискові адреси не потрібні. Файл слід закрити, щоб звільнити простір у внутрішній таблиці.

• *Read* (читання). Файл читається з поточної позиції. Процес, який працює з файлом, має вказати (відкрити) буфер і кількість читаних даних.

• *Write* (запис). Дані записуються у файл у поточну позицію. Якщо вона міститься в кінці файлу, його розмір автоматично збільшується. Інакше запис проводиться поверх наявних даних.

• *Append* (додавання). Це урізана форма попереднього виклику. Дані додаються в кінець файлу.

• *Seek* (пошук). Цей системний виклик встановлює файловий покажчик у певну позицію.

• *Get attributes* (отримання атрибутів). Процесам для роботи з файлами буває необхідно отримати їх атрибути.

• *Set attributes* (встановлення атрибутів). Цей виклик дозволяє встановити необхідні атрибути файлу після його створення.

• *Rename* (перейменування). Цей системний виклик дозволяє змінити ім’я файлу, однак таку дію можна виконати копіюванням файлу, тому цей системний виклик не є необхідним.

• *Execute* (виконати). Використовуючи цей системний виклик, файл можна запустити на виконання.

Розглянемо приклади файлових операцій в ОС Windows та UNIX. У Windows є свій набір системних викликів, які вона може виконувати. Однак корпорація Microsoft ніколи не публікувала список системних викликів Windows, крім того, вона постійно змінює їх від одного випуску до іншого. Замість цього Microsoft визначила набір функціональних викликів, який має назву Win 32 API (*Win 32 Application Programming Interface*). Ці виклики опубліковані і повністю документовані. Вони являють собою бібліотечні процедури, які або звертаються до системних викликів, щоб виконати необхідну роботу, або виконують її прямо у просторі користувача. Філософія Win 32 API полягає у наданні всеосяжного інтерфейсу, з можливістю виконати одну й ту саму вимогу декількома (трьома-чотирма) способами. В ОС UNIX усі системні виклики формують мінімальний інтерфейс: видалення навіть одного з них призведе до зниження функціональності ОС. Багато викликів API створюють об’єкти ядра того чи іншого типу (файли, процеси, потоки, канали тощо). Кожний виклик, що створює об’єкт, повертає викликаючому процесу результат, який називають дескриптором (невелике ціле число). Дескриптор використовується згодом для виконання операцій з об’єктами. Він не може бути переданий іншому процесу і використаний ним. Однак за певних обставин дескриптор може бути дубльований і переданий іншому процесу захищеним способом, що надає другому процесу контрольований доступ до об’єкта, який належить першому процесу. З кожним об’єктом асоційований дескриптор безпеки, який описує, хто і які дії може, а які не може виконувати із цим об’єктом.

Основні функції Win 32 API для файлового «введення – виведення» і відповідні системні виклики ОС UNIX наведено у табл. 7.6.

Таблиця 7.6

Функції Win 32 API для файлового «введення – виведення»

Функція Win 32 API	Системні виклики UNIX	Опис
CreateFile	open	Створити або відкрити файл; повернути дескриптор файлу
DeleteFile	unlink	Видалити наявний файл
CloseHandle	close	Закрити файл
ReadFile	read	Прочитати дані з файлу
WriteFile	write	Записати дані у файл
SetFilePointer	lseek	Встановити покажчик у файлі в певну позицію
GetFileAttributes	stat	Повернути атрибути файлу
LockFile	fcntl	Заблокувати область файлу для забезпечення взаємного вимикання
UnlockFile	fcntl	Розблокувати області файлу

Основні функції Win 32 API і системні виклики UNIX для керування каталогами наведено у табл. 7.7.

Таблиця 7.7

Основні функції Win 32 API і системні виклики UNIX для керування каталогами

Функція Win 32 API	Системні виклики UNIX	Опис
CreateDirectory	mkdir	Створити новий каталог
RemoveDirectory	rmdir	Видалити порожній каталог
FindFirstFile	opendir	Ініціалізація, щоб почати читання записів каталогу
FindNextFile	readdir	Прочитати наступний запис каталогу
MoveFile	rename	Перемістити файл з одного каталогу в інший
SetCurrentDirectory	chdir	Змінити поточний робочий каталог

7.4.6. Керування електроживленням у Windows

Диспетчер електроживлення (*power manager*) контролює показники використання електроенергії у всій системі. Історично керування споживанням енергії складалося з вимикання монітора і зупинки обертання дисководів.

Нові засоби керування електроживленням включають зменшення споживання енергії компонентами, коли система не використовується, для цього окремі пристрої перемикаються у стан резервування або навіть повністю вимикаються (за допомогою вимикача живлення). Мультипроцесорні системи відключають окремі процесори, коли вони не потрібні, і навіть можуть зменшувати тактову частоту процесорів (для зменшення енергоспоживання). Коли процесор перебуває у стані бездіяльності, споживання ним енергії також зменшується, оскільки йому не потрібно нічого робити, крім очікування виникнення переривання.

Windows підтримує спеціальний режим вимкнення під назвою *гібернація* (*hybernation*), коли виконується копіювання всієї фізичної пам’яті на диск, а потім споживання енергії знижується до мінімального (у стані глибокого сну ноутбуки можуть працювати тижнями), при цьому батарея розряджається мінімально. Оскільки весь стан пам’яті записано на диск, то можна навіть замінити батарею ноутбука (поки він перебуває в гібернації). Коли система відновлює роботу, виходячи із глибокого сну, вона відновлює збережений стан пам’яті (і повторно ініціалізує пристрої). Це переводить комп’ютер у той самий стан, в якому він був перед гібернацією (без необхідності виконувати повторно реєстрацію і запускати всі програми і служби, які виконувалися). Windows намагається оптимізувати цей процес, ігноруючи немодифіковані сторінки (мають резервування на диску), і стискає інші сторінки пам’яті для зниження необхідного обсягу «введення – виведення».

Алгоритм гібернації передбачає автоматичне балансування пропускної здатності системи «введення – виведення» і процесора. Якщо використано мультипроцесори останнього покоління, вхід у стан гібернації і вихід з нього можуть становити всього кілька секунд, навіть якщо оперативна пам'ять системи має велику місткість.

Альтернатива гібернації – *стан очікування* (*standby mode*), за якого диспетчер електроживлення переводить всю систему на нижчий стан споживання енергії (використовується рівно стільки енергії, скільки потрібно для регенерації стану динамічної пам'яті). Оскільки пам'ять не потрібно копіювати на диск, то перехід у цей стан на деяких системах здійснюється швидше, ніж гібернація.

7.4.7. Безпека у Windows

Кожний користувач (і група) у Windows ідентифікується з використанням ідентифікатора безпеки (*Security* ID (SID)). SID – це двійкове число з коротким заголовком, за яким слідує довгий випадковий компонент. Кожний SID має бути глобально-унікальним. Коли користувач запускає процес, то цей процес і його потоки виконуються під призначеним для користувача ідентифікатором SID. Велика частина системи безпеки спроектована так, щоб забезпечити доступ до будь-якого об'єкта тільки потоків з авторизованими SID.

Кожний процес має маркер доступу (*access token*), в якому вказано SID та інші властивості. Маркер зазвичай створюється модулем *winlogon*. Формат маркера показано на рис. 7.18.

Заголовок	Термін дії	Групи	DACL	Обмежені ідентифікатори SID	SID користувачі	SID групи	Привілеї

Рис. 7.18. Структура маркера доступу

Заголовок містить деяку адміністративну інформацію. Поле *Термін дії* може показати, коли маркер втрачає актуальність (у наш час його не використовують). Поле *Групи* вказує групи, яким належить процес (це потрібно для підсистеми POSIX). DACL (*Discretionary* ACL) – це список керування доступом, який присвоюється створеним процесом об'єктів (якщо не вказано інший ACL). Користувацький SID показує, хто володіє процесом. Обмежені ідентифікатори SID дозволяють ненадійним процесам брати участь у завданнях разом з надійними процесами (і при цьому в них менше можливостей щось зіпсувати).

І нарешті, привілеї (якщо вони є) надають процесу більш доступного режиму (яких немає у звичайних користувачів), такі як право вимикання комп'ютера або доступ до файлів. По суті, привілеї ділять владу суперкористувача на кілька прав, які можна присвоїти процесам. Таким чином, користувач може отримати деяку владу суперкористувача (але не всю). Підводячи підсумок: маркер доступу показує, хто володіє процесом, а також які значення за замовчуванням і які повноваження із ним пов'язані.

Коли користувач реєструється, то *winlogon* дає початковому процесу маркер доступу. Наступні процеси зазвичай успадковують цей маркер. Маркер доступу процесу спочатку застосовується до всіх потоків процесу.

Ще одна фундаментальна концепція – ***дескриптор безпеки*** (*security descriptor*). Кожний об'єкт має пов'язаний з ним дескриптор безпеки, який показує, хто і які операції може виконувати з ним. Дескриптори безпеки вказуються під час створення об'єктів. Файлова система NTFS і реєстр підтримують постійну форму дескриптора безпеки, який використовується для створення дескриптора безпеки для об'єктів File і Key (це об'єкти диспетчера об'єктів, що надають відкриті екземпляри файлів і ключів).

Дескриптор безпеки складається із заголовка, за яким слідує список *DACL* з одним (або більше) елементом керування доступом *ACE* (*Access*

Control Entries). Два основні типи такого елемента – *Allow* та *Deny*. Елемент Allow вказує SID і бітовий масив, який, у свою чергу, вказує, які операції цей SID може виконувати над об’єктом. Елемент *Deny* працює аналогічно (проте збіг означає, що викликає сторона не може виконувати операцію).

На додаток до списку DACL дескриптор безпеки має також список *SACL* (*System Access Control*), який схожий на DACL, за винятком того, що він вказує не тих, хто може використовувати об’єкт, а те, які операції над об’єктом записуються в системний журнал подій безпеки.

7.4.8. Виклики інтерфейсу прикладного програмування безпеки

Велика частина механізму керування доступом у Windows заснована на дескрипторах безпеки. Зазвичай схема така: коли процес створює об’єкт, він надає дескриптор безпеки як один із параметрів *CreateProcess* або *CreateFile*.

Цей дескриптор безпеки потім стає прикріпленим до об’єкта дескриптором безпеки. Багато викликів системи безпеки у Win32 належать до керування дескрипторами безпеки, тому розглянемо саме їх.

Найважливіші виклики наведено в табл. 7.8.

Під час створення дескриптора безпеки спочатку під нього виділяється, а потім й ініціалізується (за допомогою *InitializeSecurityDescriptor*) місце зберігання. Цей виклик заповнює заголовок.

Якщо SID власника невідомий, то його можна пошукати за іменем за допомогою *LookupAccountSid*.

Потім його можна вставити у дескриптор безпеки. Те саме можна зробити і з SID групи (якщо він є).

У цей момент можна форматувати список DACL (або SACL) дескриптора безпеки (за допомогою *InitializeAcl*). Елементи ACL можна додавати за допомогою *AddAccessAllowedAce* та *AddAccessDeniedAce*. Ці виклики можна повторювати багато разів (для додавання необхідної

кількості елементів ACE). *DeleteAce* можна використовувати для видалення елемента (під час модифікації наявного ACL). Коли ACL готовий, можна використовувати *SetSecurityDescriptorDacl* для прикріплення його до дескриптора безпеки. І нарешті, коли об'єкт створюється, то підготовлений дескриптор безпеки можна передати як параметр (щоб прикріпити його до об'єкта).

Таблиця 7.8

Основні функції безпеки у Win32

Функція Win32	Опис
InitializeSecurityDescriptor	Підготувати до використання новий дескриптор безпеки
LookupAccountSid	Шукати SID для заданого імені користувача
SetSecurityDescriptorOwner	Ввести SID власника у дескриптор безпеки
SetSecurityDescriptorGroup	Ввести SID групи у дескриптор безпеки
InitializeAcl	Ініціалізувати DACL або SACL
AddAccessAllowedAce	Додати новий ACE в DACL або SACL для дозволу доступу
AddAccessDeniedAce	Додати новий ACE в DACL або SACL для заборони доступу
DeleteAce	Видалити ACE з DACL або SACL
SetSecurityDescriptorDacl	Прикріпити DACL до дескриптора безпеки

Контрольні запитання

1. Що являє собою модель ОС Windows?
2. Що таке гіпервізор?
3. Які типи процесів режиму користувача вам відомі?
4. Що таке Win32 API?
5. Що таке реєстр Windows?
6. З яких розілів складається реєстр Windows?
7. Що тае дескриптор?
8. Що таке *Boot*-менеджер?
9. Яке призначення має підсистема середовища оточення?
10. Яке призначення диспетчера задач?

11. Що таке процес? Які його складові?

12. Що таке потік? Які його складові?

13. Які стани має потік?

14. Із чого складається образ процесу?

15. Із чого складється образ потоку?

16. Що таке контекст процесу? Як здійснюється його перемикання?

17. Як створюється процес і як він поданий у Windows?

18. Що таке завдання?

19. Що таке волокна?

20. Яка функція створює новий процес?

21. Що таке віртуальна пам’ять?

22. Які вам відомі переваги віртуальної пам’яті?

23. Як визначається діапазон значень фізичних адрес оперативної пам’яті?

24. Що таке сегментна організація пам’яті? Які її переваги та недоліки?

25. Для чого використовують таблицю сегментів?

26. Що таке сегментно-сторінкова організація віртуальної пам’яті?

27. Що таке менеджер віртуальної пам’яті?

28. Що являє собою ієрархічна структура пам’яті?

29. Що таке файл підкачки?

30. У чому полягає кешування у Windows?

31. Яке призначення диспетчера «введення – виведення» Windows?

32. Що таке асінхронне «введення – виведення»?

33. Що таке модель драйверів пристроїв Windows?

34. Які можливості диспетчера Plug-and-Play?

35. Які інтерфейси системних викликів надає диспетчер «введення – виведення»?

36. Які файлові системи підтримує Windows?

37. Яка структура файлової системи NTFS?

38. У чому полягає модель відстеження дискових блоків?

39. Що таке режим гіпернації?

40. Які фундаментальні концепції безпеки у Windows?

Список літератури

Базова література

1. Немет Эви, Гарт Снайдер, Трент Хейн, Бэн Уэйли. Unix и Linux: руководство системного администратора / Немет Эви, Гарт Снайдер, Трент Хейн, Бэн Уэйли ; пер. с англ. – 4-е изд. – М. : ООО «И. Д. Вильямс», 2012. – 1312 с.

2. Керриск М. Linux API. Исчерпывающее руководство / М. Керриск. – СПб. : Питер, 2019. – 1248 с.

3. Руссинович М. Внутреннее устройство Windows / М. Руссинович, Д. Соломон, А. Ионеску, П. Йосифович. – СПб. : Питер, 2018. – 944 с.

4. Моэт Э. Использование Docker / Э. Моэт. – М. : ДМК Пресс, 2017. – 354 с.

5. Парминдер Сингх Кочер. Микросервисы и контейнеры Docker / Парминдер Сингх Кочер. – М. : ДМК Пресс, 2019. – 240 с.

6. Джиджи С. Осваиваем Kubernetes. Оркестрация контейнерных архитектур / С. Джиджи. – СПб. : Питер, 2019. – 400 с.

7. Ричардсон К. Микросервисы. Паттерны разработки и рефакторинга / К. Ричардсон. – СПб. : Питер, 2020. – 544 с.

8. Кетов Д. В. Внутреннее устройство Linux / Д. В. Кетов. – СПб. : БХВ-Петербург, 2021. – 400 с.

Додаткова література

9. Уорд Б. Внутреннее устройство Linux / Б. Уорд. – СПб. : Питер, 2016. – 384 с.

10. Лав Р. Linux. Системное программирование. 2-е изд. — СПб.: Питер, 2014. — 448 с.

11. Волох С. Ubuntu Linux с нуля / С. Волох. – СПб. : БХВ-Питер, 2016. – 400 с.

12. Бреснахэн К. Linux на практике / К. Бреснахэн, Р. Блум. – СПб. : Питер, 2017. – 384 с.

13. Колисниченко Д. Linux от новичка к профессионалу / Д. Колисниченко. – СПб. : БХВ-Питер, 2016. – 672 с.

14. Шотт У. Командная строка Linux. Полное руководство / У. Шотт. – СПб. : Питер, 2016. – 480 с.

15. Тейлор Д. Сценарии командной оболочки. Linux, OS X и Unix / Д. Тейлор, Б. Перри. – СПб. : Питер, 2017. – 448 с.

16. Роббинс А. Bash. Карманный справочник системного администратора / А. Роббинс. – СПб. : ООО «Альфа-книга», 2017. – 152 с.

17. Сейерс Э. Х. Docker на практике / Э. Х. Сейерс, А. Милл. – М. : ДМК, 2019. – 516 с.

18. Turnbull J. The Docker Book: Containerization is the new virtualization. Kindle Edition. 2014. 332 p.

19. Cannon J. Docker: A Project-Based Approach to Learning. Kindle Edition. 2021. 298 p.

20. Хорсдал Кристиан. Микросервисы на платформе .NET / Кристиан Хорсдал. – СПб. : Питер, 2018. – 352 с.

21. Танебаум Э. Современные операционные системы / Э. Танебаум, Х. Бос. – СПб. : Питер, 2019. – 1120 с.

22. Дейтел Х. Операционые системы. Основы и принципы / Х. Дейтел, П. Дейтел, Д. Чофнес. – М. : Бином, 2016. – 1024 с.

Інформаційні ресурси

23. Программирование для Linux. Потоки [Электронный ресурс]. – Режим доступа: http://www.citforum.ru/programming/unix/threads/. 12.11.2008.

24. Docker-документация на русском [Электронный ресурс]. – Режим доступа: https://dker.ru/docs/

25. Модуль безпеки SELinux [Електронний ресурс]. – Режим доступу: https://www.ibm.com/developerworks/ru/library/l-se_linux_01/index.html

26. Модуль безпеки AppArmor [Електронний ресурс]. – Режим доступу: https://www.ibm.com/developerworks/ru/library/l-apparmor-3/index.html

27. Приклади вихідного коду, навчальні завдання [Електронний ресурс]. – Режим доступу: https://github.com/using-docker/

28. Автоматичне встановлення Docker [Електронний ресурс]. – Режим доступу: https://get.docker.com

29. Бінарні пакети Docker [Електронний ресурс]. – Режим доступу: https://docs.docker.com/installation/binaries/

30. Як використовувати систему Git [Електронний ресурс]. – Режим доступу: https://blog.sedicomm.com/2019/11/27/kak-ispolzovat-sistemu-upravleniya-versiyami-git-v-linux-vseobemlyushhee-rukovodstvo/