МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
"КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
ІМЕНІ ІГОРЯ СІКОРСЬКОГО"

ОПЕРАЦІЙНІ СИСТЕМИ

КОМП'ЮТЕРНИЙ ПРАКТИКУМ

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для студентів,
які навчаються за спеціальністю 121 «Інженерія програмного забезпечення»,
освітньої програми «Інженерія програмне забезпечення розподілених систем»

Київ

КПІ ім. Ігоря Сікорського

2020

Операційні системи. Комп'ютерний практикум [Електронний ресурс] : навч. посіб. для студентів спеціальності 121 «Інженерія програмного забезпечення», освітньої програми «Інженерія програмного забезпечення розподілених систем» / КПІ ім. Ігоря Сікорського ; уклад.: Л.О. Левченко, В.В. Шпурик, В.П. Колумбет – Електронні текстові дані (1 файл: 4,19 Мбайт). – Київ : КПІ ім. Ігоря Сікорського, 2020. – 138-с.

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського (протокол № 10 від 18.06.2020 р.)
за поданням Вченої ради Теплоенергетичного факультету (протокол № 10 від 25.05.2020 р.)

Електронне мережне навчальне видання

Операційні системи

Комп'ютерний практикум

Укладачі: Левченко Лариса Олексіївна, канд. екон. наук, доцент,
Шпурик Вадим Вадимович, канд. техн. наук, доцент,
Колумбет Вадим Петрович, старший викладач.

Відповідальний редактор: Коваль О.В., канд. техн. наук, доцент, в.о. зав. кафедри автоматизації проектування енергетичних процесів і систем ТЕФ КПІ ім. Ігоря Сікорського

Рецензент: Баранюк Олександр Володимирович, канд. техн. наук, доцент кафедри атомних електричних станцій і інженерної теплофізики теплоенергетики ТЕФ КПІ ім. Ігоря Сікорського

Посібник розроблений на підставі робочої програми кредитного модуля з дисципліни «Операційні системи» та призначений для якісної організації виконання практичних робіт студентами обсягом 36 годин, підвищення розуміння основ побудови та функціонування операційної системи Linux з використанням технології віртуалізації.

Призначений для студентів денної форми, які навчаються за спеціальністю 121 – «Інженерія програмного забезпечення», освітньої програми підготовки бакалаврів «Інженерія програмного забезпечення розподілених систем».

Спрямований на формування у студентів умінь та навичок з основ роботи у середовищі операційної системи Linux, створення контейнерів (сервісів) за технологією Docker для розробки програмних додатків. Забезпечує студентів необхідними теоретично-методичними знаннями для опанування відповідної теми комп'ютерного практикуму та виконання завдань, запланованих впродовж семестру.

ЗМІСТ

ВСТУП

Навчальна дисципліна «*Операційні системи»* входить до циклу професійної та практичної підготовки навчального плану бакалаврів спеціальності 121 «Інженерія програмного забезпечення» освітньої програми «Інженерія програмного забезпечення розподілених систем». Дисципліна складається з одного кредитного модуля та викладається студентам другого курсу у четвертому семестрі.

Знання та досвід, отримані у результаті вивчення даної дисципліни, дозволяють студентам опановувати засоби та методи роботи в середовищі операційної системи Linux з використанням технології віртуалізації.

Розглянуто: користувачі з точки зору системи, поняття термінал, робота у командному рядку, особливості роботи з файловою системою, права доступу в Linux, можливості командної оболонки, текстові редактори, процеси, потоки. Особливу увагу заслуговує розгляд найпопулярнішої платформи управління контейнерами Docker, яка нещодавно з'явилася на ринку IT-технологій. Саме контейнери Docker спрощують як упаковку програмних додатків, так і їх перенесення, запуск в різних локальних середовищах і в хмарі.

Комп'ютерний практикум містить перелік завдань, в результаті виконання яких студенти здійснюють налаштування середовища операційної системи.

Теоретичною основою щодо виконання комп'ютерного практикуму є курс лекцій з дисципліни „Операційні системи". Комп'ютерний практикум є обов'язковим до виконання згідно робочої програми дисципліни.

Для виконання комп'ютерного практикуму необхідно мати ноутбук з обсягом оперативної пам'яті не менше 4 Гб, системою команд, сумісних з командами мікропроцесора Intel/AMD, використовувати 64-бітне програмне забезпечення у 64-розрядній Microsoft Windows операційній системі, встановити у Windows віртуальну машину Oracle VirtualBox, на якій встановити ISO-образ «Ubuntu Desktop 18.04 LTS».

КОМП'ЮТЕРНИЙ ПРАКТИКУМ 1

Установка операційної системи Ubuntu в Oracle VirtualBox

Мета роботи:

- набуття навичок установки і створення віртуальних машин в Oracle VirtualBox;
- набуття навичок установки і початкового налаштування ОС Ubuntu.

Теоретичні відомості

Віртуалізація – це технологія, яка дозволяє на одному комп'ютері одночасно запускати декілька операційних систем (Linux, Windows, Mac та ін.). Така технологія реалізується за допомогою програми VirtualBox. Це означає, що за допомогою VirtualBox знаходячись, наприклад, в Windows можна без перезавантаження комп'ютера завантажитися в Linux та працювати з будь-якими додатками Linux.

Віртуалізація має низку переваг:

- одночасний запуск декількох операційних систем (Linux, Windows, Mac та ін.) разом з основною системою, встановленою на комп'ютері;
- віртуальна ОС повністю відокремлена від основної ОС, що дозволяє проведення різних тестів у віртуальній ОС для виявлення як працездатності різних додатків, так і шкідливого програмного забезпечення;

Хостова операційна система (хост-система) - це операційна система фізичного комп'ютера, на якому був встановлений VirtualBox. Існують версії VirtualBox для Windows, Mac OS X, Linux і Solaris.

Гостьова операційна система (гостьова ОС) - це операційна система, яка працює всередині віртуальної машини.

Завдання:

1. Скачайте дистрибутив Oracle VirtualBox останньої версії програми з офіційного сайту
https://www.virtualbox.org/wiki/Downloads
2. Встановіть в Windows віртуальну машину Oracle VirtualBox.
3. Скачайте з офіційного сайту https://ubuntu.com/#download ISO-образ «Ubuntu Desktop 18.04 LTS» установочного диска ОС Ubuntu.

4. Встановіть ОС Ubuntu на створену віртуальну машину.

При скачуванні ОС Ubuntu вам запропонують вибрати між звичайною версією і LTS-версією (*Long Term Support* – підтримка протягом тривалого періоду). Нова версія Ubuntu з'являється кожні шість місяців, в даному випадку *Ubuntu Desktop 19.10* як найсвіжіший реліз (індекс "19.10" означає - 2019 рік, 10-й місяць). Це «проміжний» щорічний реліз, в якому реалізовують нові технології.

Однак вам пропонується скачати файл ***ubunto-18.04-desktop-amd64.iso***, оскільки для подальших робіт необхідно буде встановити ще одне програмне забезпечення, яке працює тільки з даним образом**.**

Більшість LTS-релізів Ubuntu підтримується до п’яти років – клієнтські версії 3 роки, серверні – 5 років. Їх основна особливість – стабільна робота, але без впровадження нових технологій. Не LTS-версії підтримуються півтора року.

5. У встановленій операційній системі:

- встановіть розширення (доповнення) гостьової ОС,
- налаштуйте робочі столи (ефекти, зображення);
- змініть розкладку клавіатури за замовчуванням;
- визначте тип сеансу як такий, що завантажується за замовчуванням.

6. Необхідно ознайомитися та описати панель інструментів Ubuntu.

Хід виконання роботи

Спочатку необхідно встановити Oracle VirtualBox і створити віртуальну машину. Установка Oracle VirtualBox проводитися з налаштуваннями за замовчуванням (погоджуємося з ходом установки). Якщо у VirtualBox необхідно забезпечити підтримку пристроїв USB 2.0 і USB 3.0, VirtualBox RDP, шифрування диска, завантаження NVMe і PXE для карт Intel, після установки віртуальної машини треба додатково встановити пакет розширення (доповнень) до гостьової ОС (Oracle_VM_VirtualBox_Extension_Pack). Цей пакет доповнень скачуєте з того ж сайту. Після установки VirtualBox виконуємо послідовність «Файл-Настройки-Плагины» - значком «+» додаємо попередньо скачаний файл Extension_Pack.

Установка операційної системи Ubuntu здійснюється на віртуальну машину від Oracle VirtualBox.

1) ***Встановлення на віртуальну машину дистрибутива операційної системи Linux Ubuntu***:

Запускаємо менеджер віртуальних машин Oracle VirtualBox (Пуск ▸ Программы ▸ Oracle VM VirtualBox ▸ Oracle VM VirtualBox). Створюємо нову віртуальну машину: **Создать/New/Ctrl+N.**

Вказуємо: місце розташування VM *E:\VM\VB,*

Имя: Ubuntu-01. Тип: Linux. Версия: Ubuntu (64 bit) (рис. 1.1).

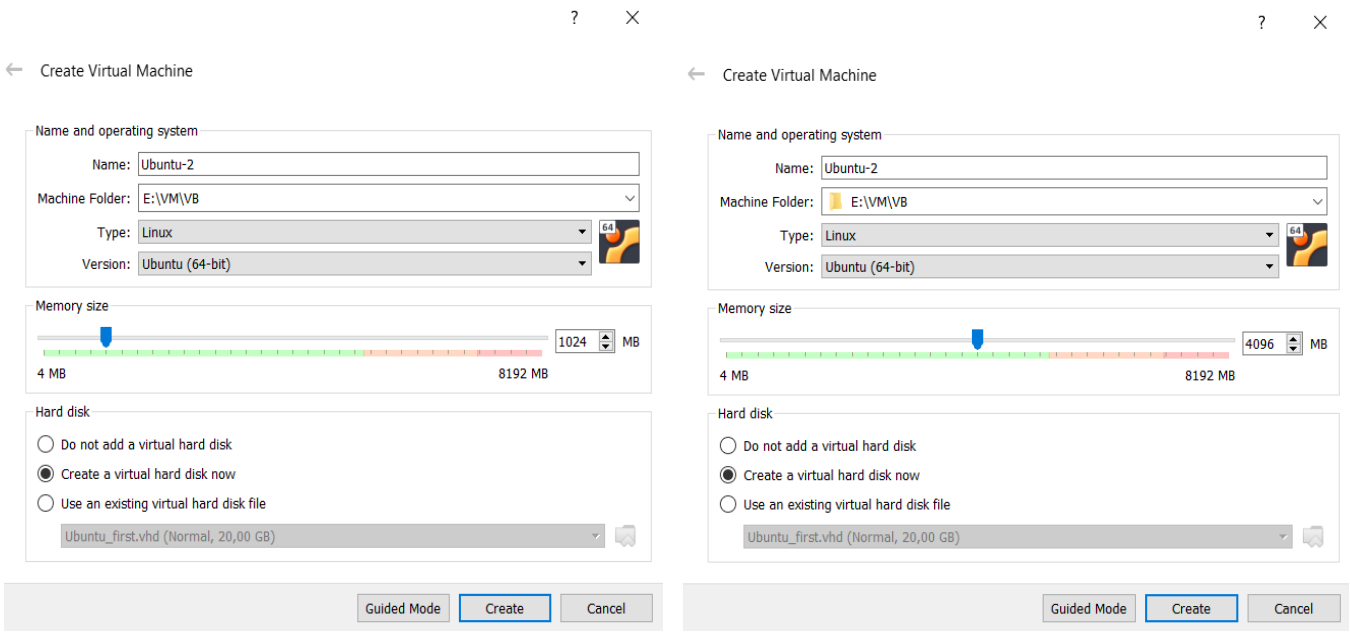


Рис. 1.1. Створення нової віртуальної машини в менеджері Oracle VirtualBox

Натискаємо кнопку «Create» (рис. 1.1).

Встановіть обсяг оперативної пам'яті, яка виділяється віртуальній машині (за замовчуванням1024 МБ). Бажано виділяти не менше 2 ГБ (оптимально 4 ГБ). Безпечно виділяти під віртуальну машину половину від загального обсягу оперативного пам’яті хостової машини, але можна і більше (за формулою: обсяг оперативної пам’яті комп’ютера мінус 4 ГБ, залишаючи під потреби системи). Якщо на хостовій машині обсяг оперативної пам’яті 8 ГБ, то для неї треба залишати не менше 4 ГБ пам’яті, а для гостьової операційної системи виділяти обсяг пам’яті до 4096 МБ. Якщо на хостовій машині обсяг ОЗП 16 ГБ, то сумарний обсяг одночасно запущених гостьових систем на повинен перевищувати 12 ГБ. Натискаємо кнопку «Создать»/«Create» (рис. 1.2).

Вказуємо тип нового віртуального жорсткого диска: VHD (Virtual Hard Disk). Формат VHD дозволяє у разі потреби перенести віртуальну машину на реальну машину. Вказуємо формат зберігання: Dynamically allocated (Динамический виртуальный жёсткий диск), Це означає, що файл, в якому зберігається вся інформація віртуальної машини, буде мати мінімально можливий розмір, а потім буде збільшувати свій розмір у міру заповнення віртуального диска. Вказуємо шлях, де буде зберігатися новий віртуальний жорсткий диск, та ім'я Ubuntu-2 віртуальної машини, максимальний розмір віртуального жорсткого дика: 20 ГБ. Натискаємо кнопку «Создать»/«Create» (рис. 1.2).

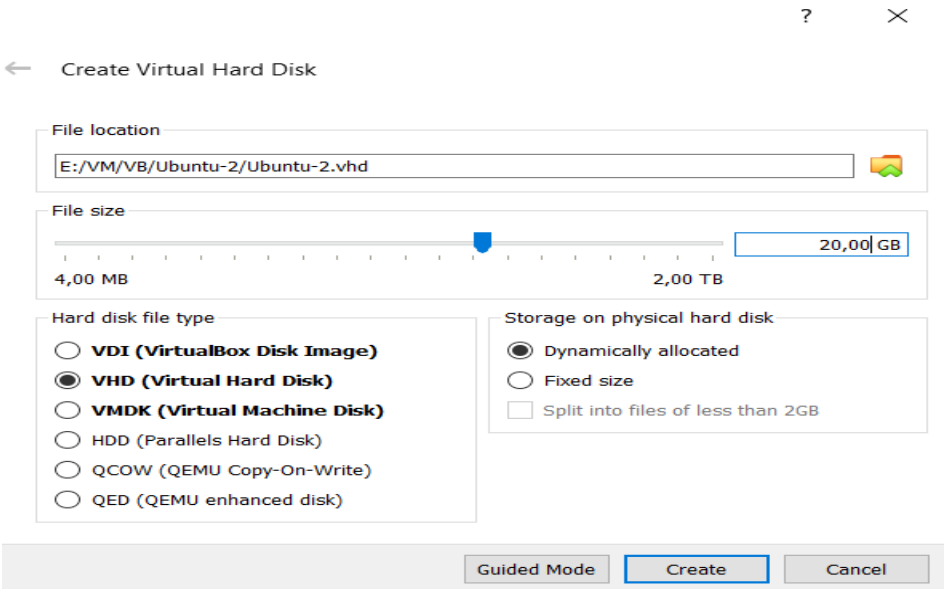


Рис. 1.2. Визначення імені та розміру віртуального жорсткого диска

Дочекатися створення віртуальної машини. На екрані з’являться основні параметри створеної віртуальної машини. Її ім'я з'явилося у списку доступних віртуальних машин у вікні менеджера Oracle VirtualBox (рис. 1.3).

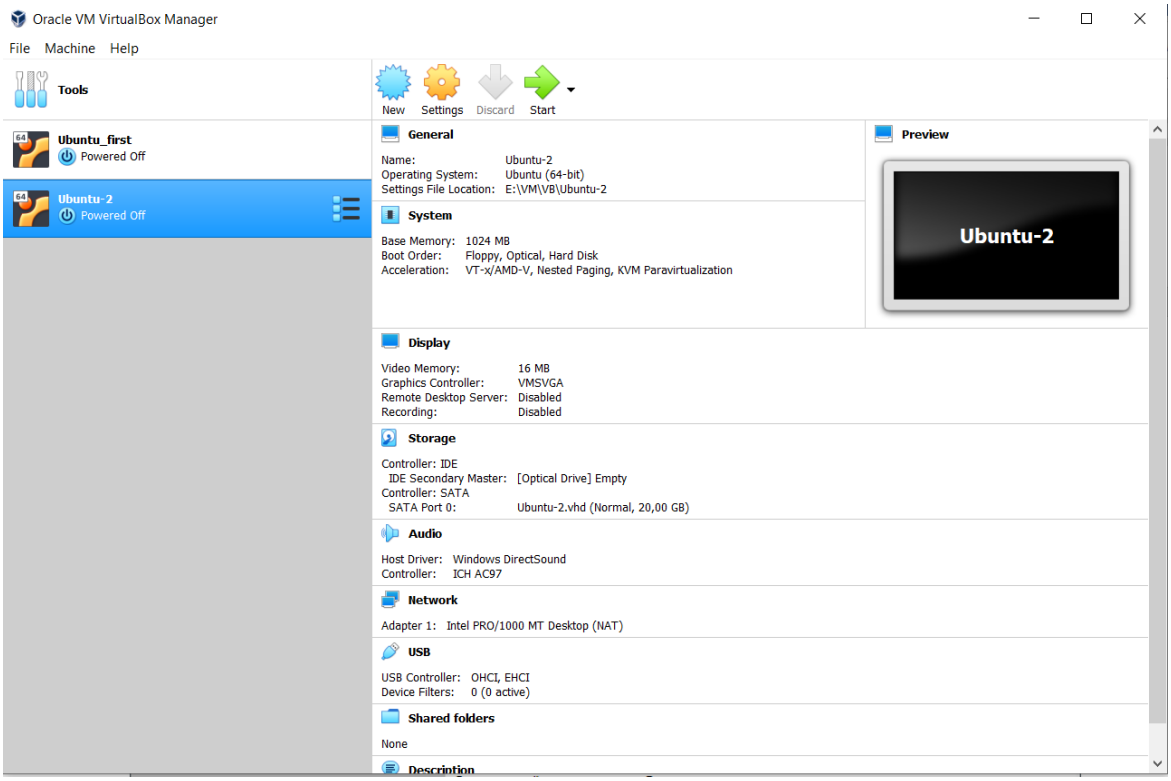


Рис. 1.3. Параметри створеної віртуальної машини

2) *Налаштування віртуальної машини в VirtualBox*

Клацніть правою кнопкою миші по значку імені віртуальної машини і виберіть пункт «Настроить»/«Settings» (Ctrl+S). У діалоговому вікні ліворуч вибираємо пункт «Общие»/«General». Праворуч вибираємо вкладку «Общие–Дополнительно» / «***General***–Advanced», встановити поля «Общий буфер обмена» та «Функция Drag'n'Drop» («Shared Clipboard» та «Drag'n'Drop») ***Двунаправленый/Disabled.*** Створити на диску папку для знімків Snapshots та вказати у полі «Папка для снимков»/«Snapshots Folder» шлях до цієї папки (рис. 1.4).

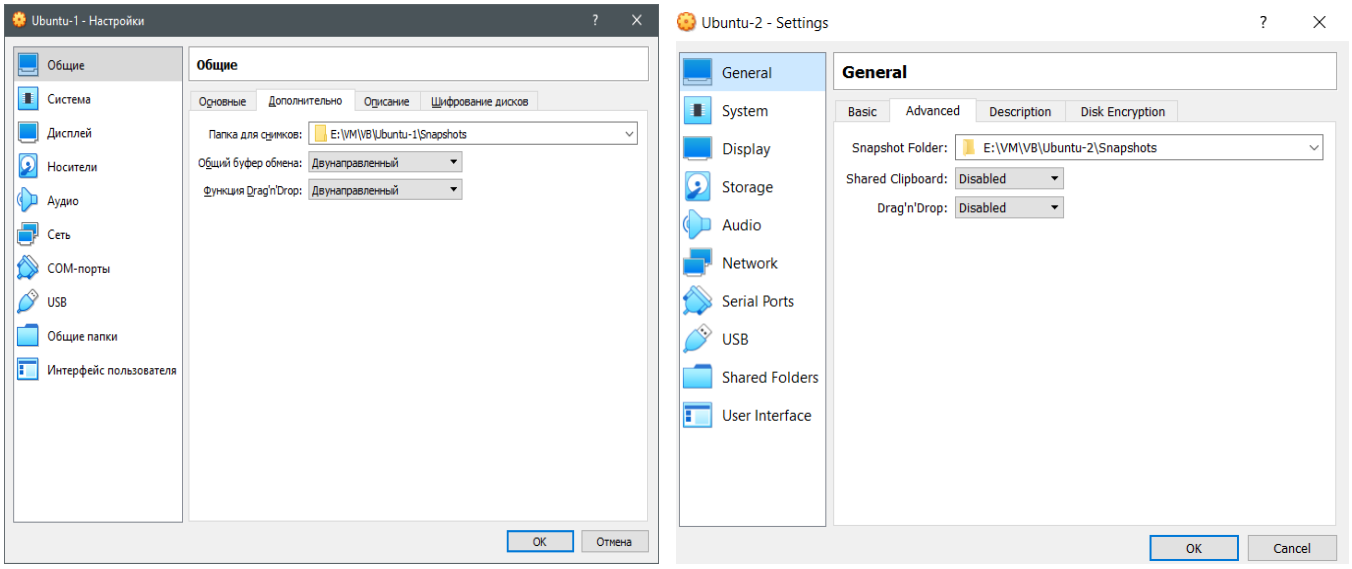


Рис. 1.4. Налаштування вкладки «Дополнительно» розділу «Общие»

У розділі «Система»/«System», вкладка «Процессор»/«Processor» за бажанням можна збільшити кількість процесорів, доступних віртуальній машині (бажано не менше 2-х процесорів) (рис. 1.5). Також бажано включити Додаткові можливості/Extended Features – Включить Nested VT-x/AMD-V / Enable Nested VT-x/AMD-V.

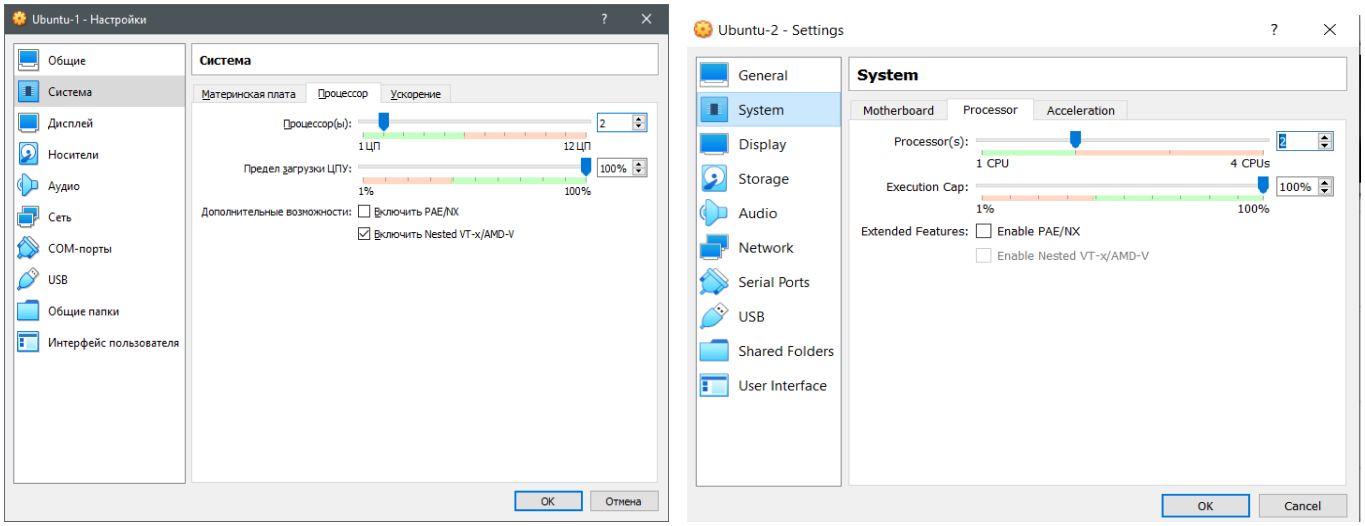


Рис. 1.5. Налаштування вкладки «Процессор» розділу «Система»

У розділі «Дисплей»/«Display» (рис. 1.6) і на вкладці «Екран»/«Screen» біля поля «Ускорение»/«Acceleration ставимо галочку «Включить 3D-ускорение»/«Enable 3D-Acceleration». Також можна збільшити обсяг відеопам'яті до 128 Мб («Видеопамять»/«Video Memory»).

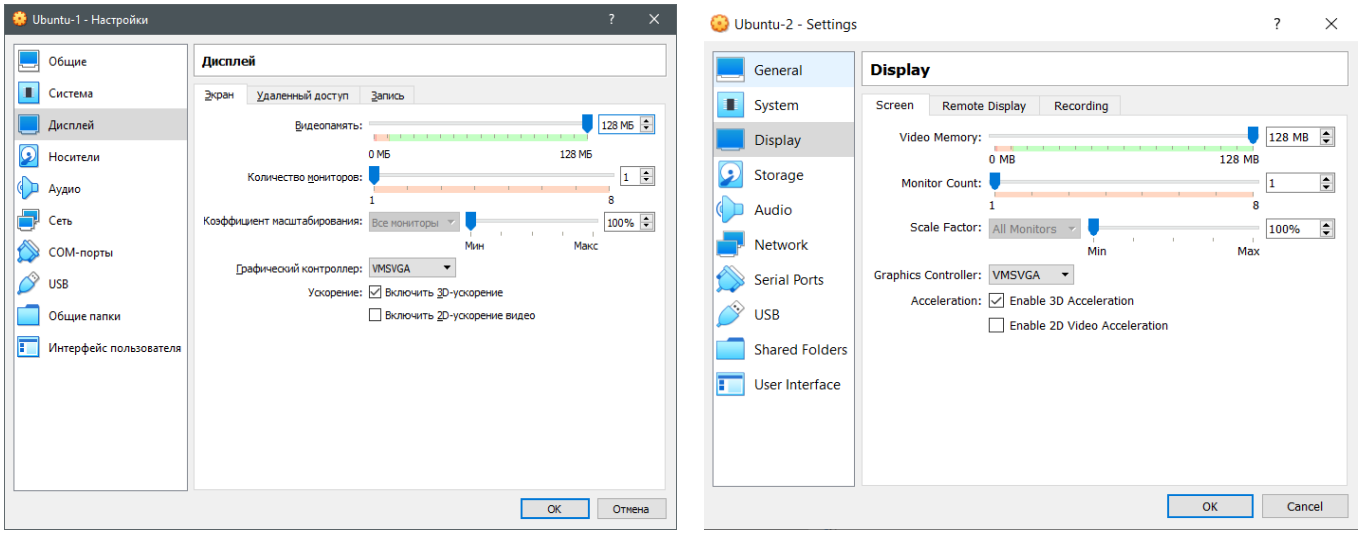


Рис. 1.6. Налаштування розділу «Дисплей»

У розділі «Общие папки»/«Shared Folders» праворуч натискаємо кнопку «+»  , щоб додати загальну папку, за допомогою якої можна буде копіювати файли з хост-системи на віртуальну машину і назад. В діалозі виберемо папку, яка буде спільною

для обох систем (Shared), в наступному полі автоматично з'явиться ім'я для спільної папки (рис. 1.7), яке буде використовуватися віртуальною машиною. Поставимо галочку *Auto mount* (Авто-підключення) та два рази натиснемо кнопку OK.

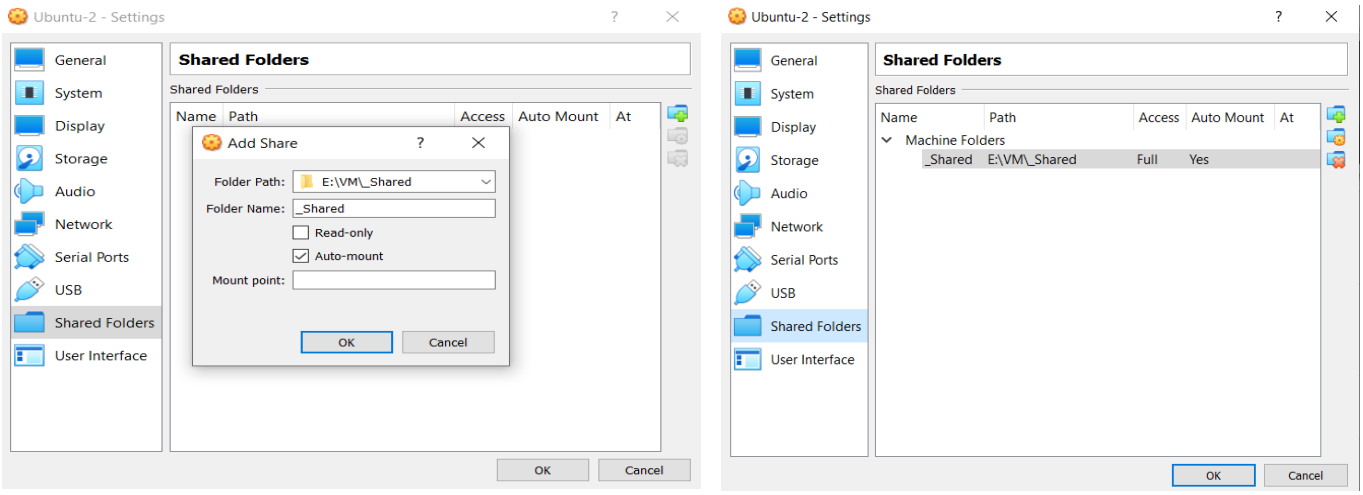


Рис. 1.7. Спільна папка

3) Встановлення ОС Ubuntu на створену віртуальну машину

Основні налаштування виконані. Запускаємо створену віртуальну машину Ubuntu-1/Ubuntu-2 (лівою кнопкою миші по назві нашої віртуальної машини і натискаємо на кнопці «Запустить»/«Start» (рис. 1.8).

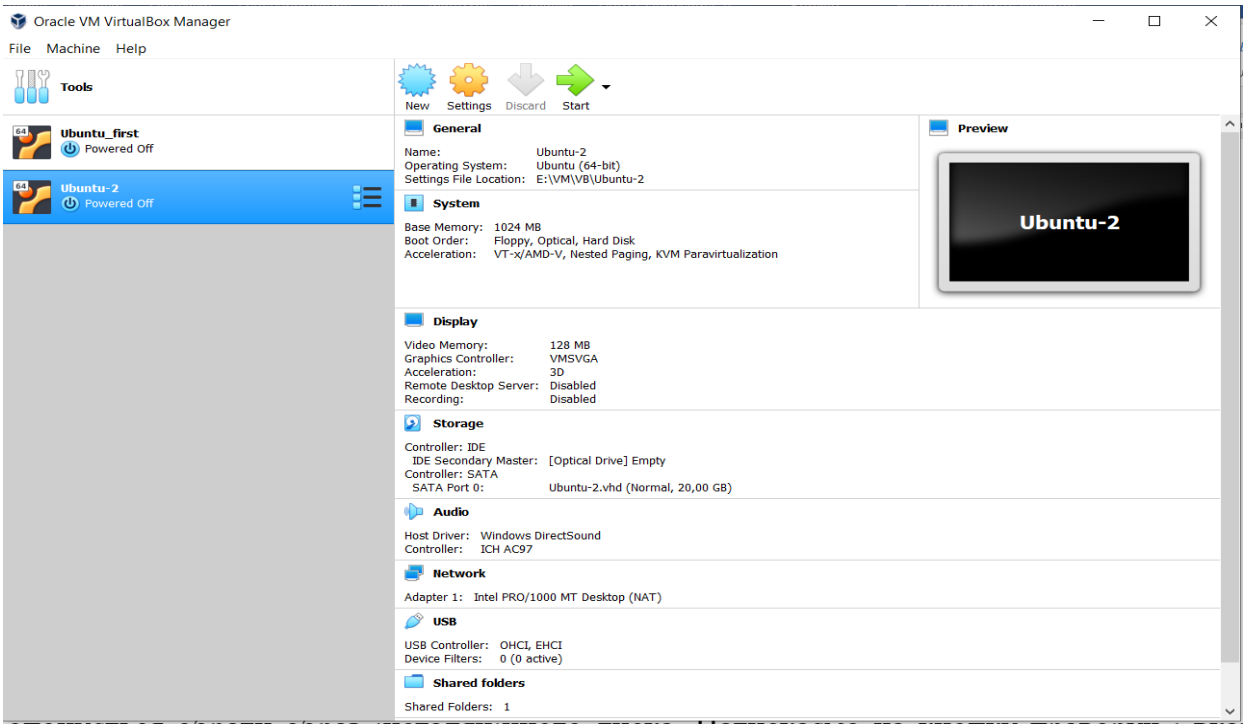


Рис. 1.8. Запуск віртуальної машини

Пропонується обрати образ інсталяційного диска. Натискаємо на кнопку праворуч і вказуємо шлях, де знаходиться ISO-файл, завантажений з сайту Ubuntu (рис. 1.9). Натискаємо кнопку «Start».

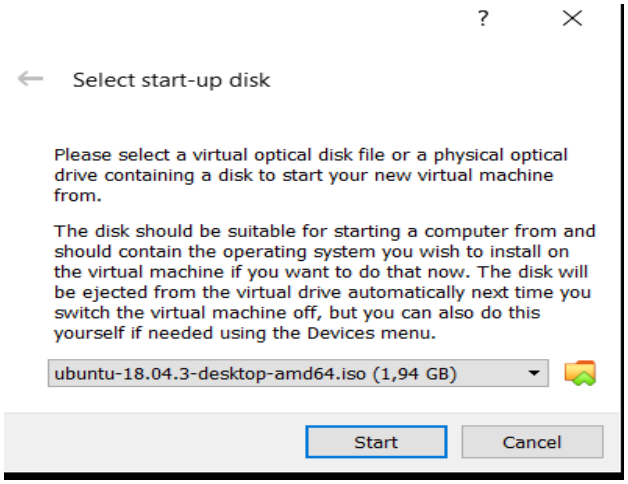


Рис. 1.9. Вибір образу інсталяційного диска при першому старті віртуальної машини

Після нетривалої паузи на екрані з'явиться зображення робочого столу віртуальної машини (рис. 1.10). Обираємо зі списку ліворуч мову (English) і натискаємо кнопку «Install Ubuntu».

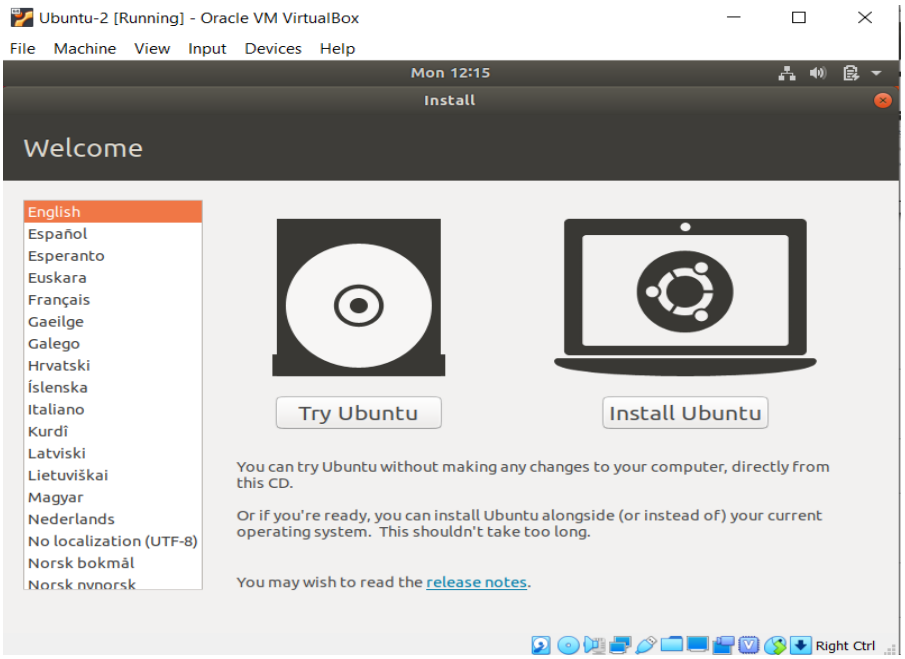


Рис. 1.10. Вибір мови та початок установки Ubuntu

Далі відображається вікно «Updates and other software» / Оновлення та інше програмне забезпечення (рис.1.11), в якому треба обрати «Normal Installation» / Звичайна установка та «Download updates while installing Ubuntu» / Завантажити оновлення при встановленні Ubuntu.

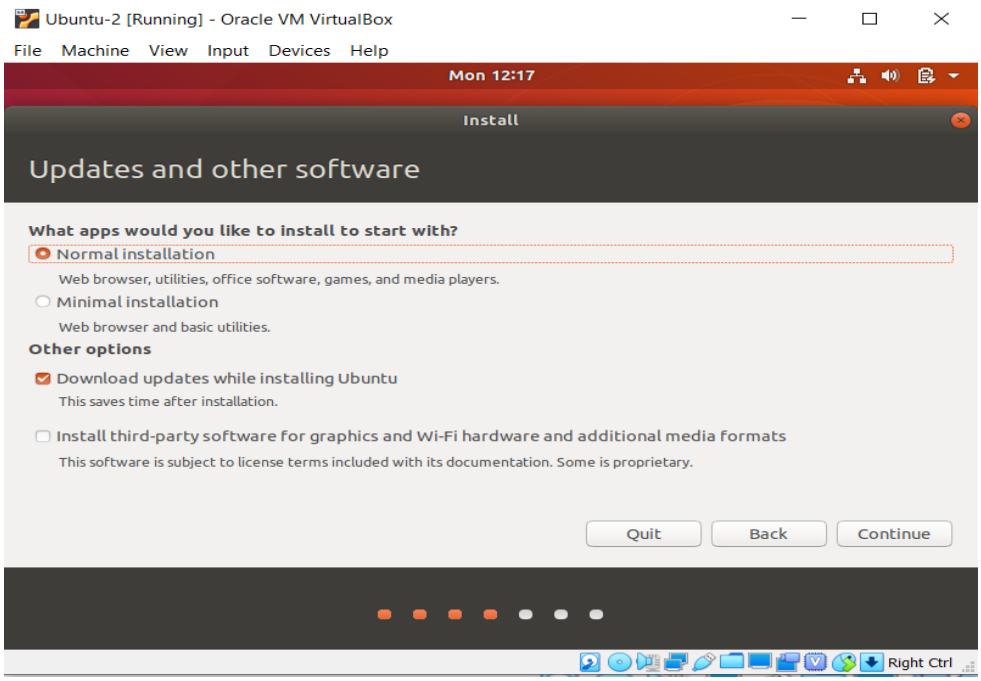


Рис. 1.11. Оновлення та інше програмне забезпечення

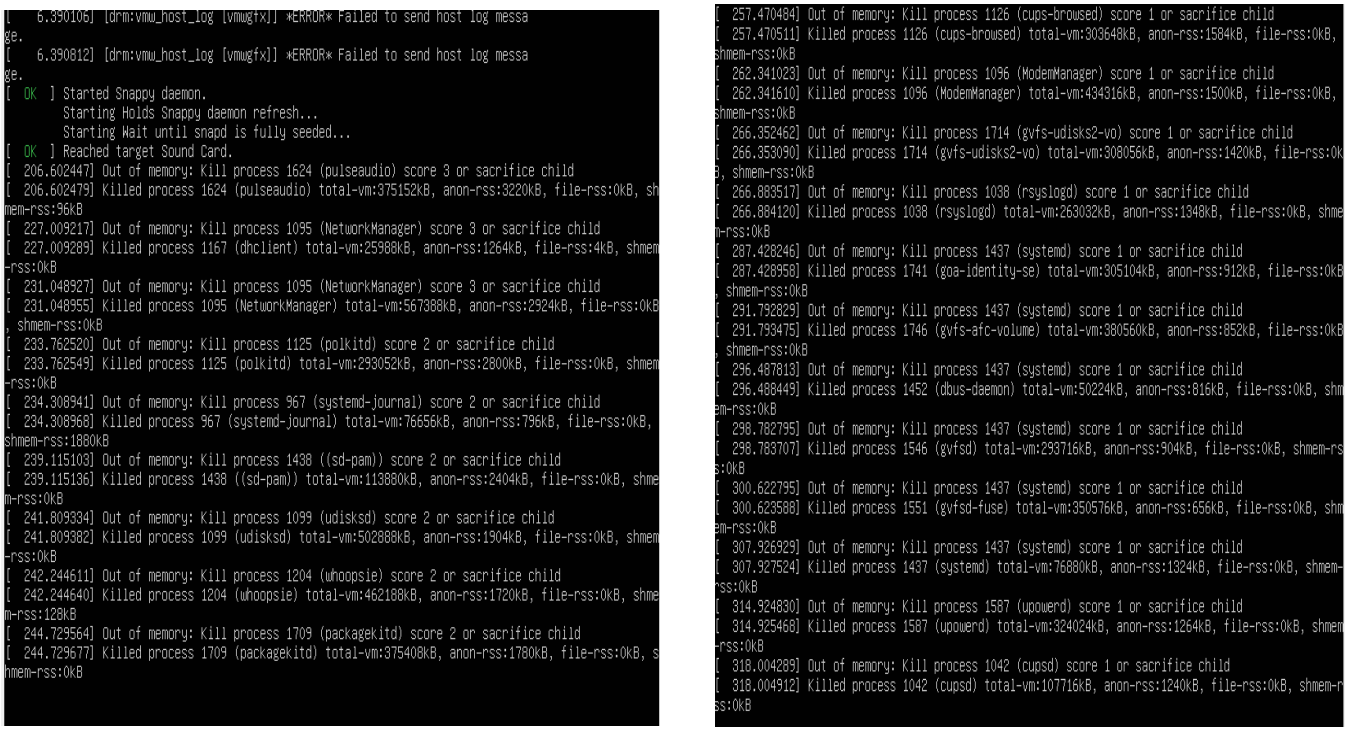


Рис. 1.12. Вибір оновлень

Пропонуються оновлення (рис. 1.12). Далі необхідно визначит тип установки (рис. 1.13) – Erase disk and install Ubuntu / Стерти диск і встановити Ubuntu. Натиснути Install Now.

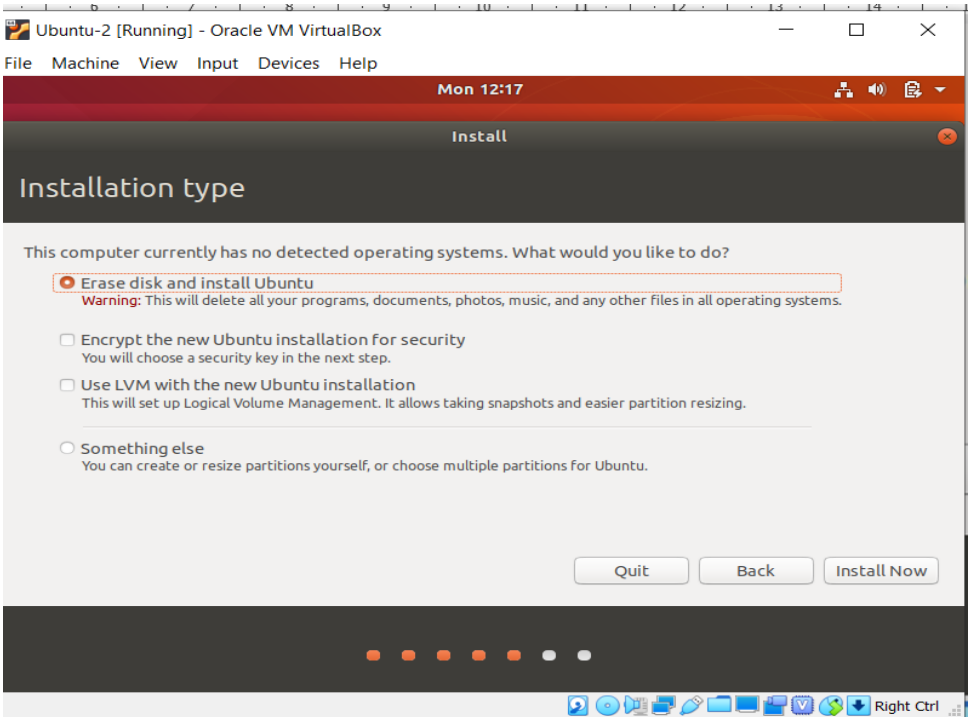


Рис. 1.13. Тип установки

Обираємо часовий пояс і місто, розкладку клавіатури.

Вводимо ім'я користувача, ім'я комп'ютера і двічі пароль. Пароль повинен містити великі і малі літери латинського алфавіту, цифри і спеціальні знаки, наприклад, крапку, кому або зірочку. Встановлюємо галочку «Log in automatically»/Входити до системи автоматично. Розпочинається установка (рис. 1.14).

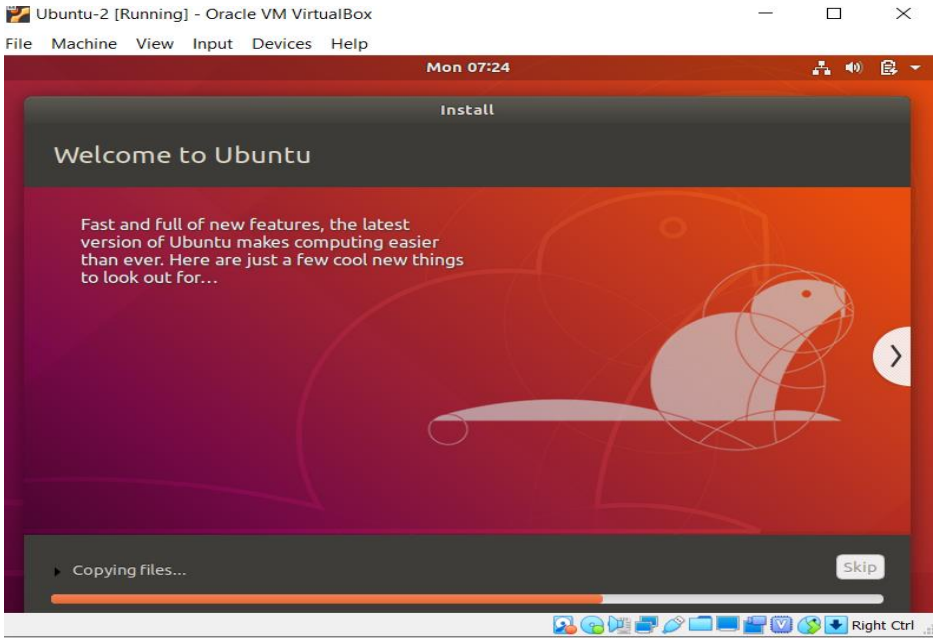


Рис. 1.14. Процес установки Ubuntu

По завершенні установки натиснемо кнопку «Restart Now» (рис. 1.15).

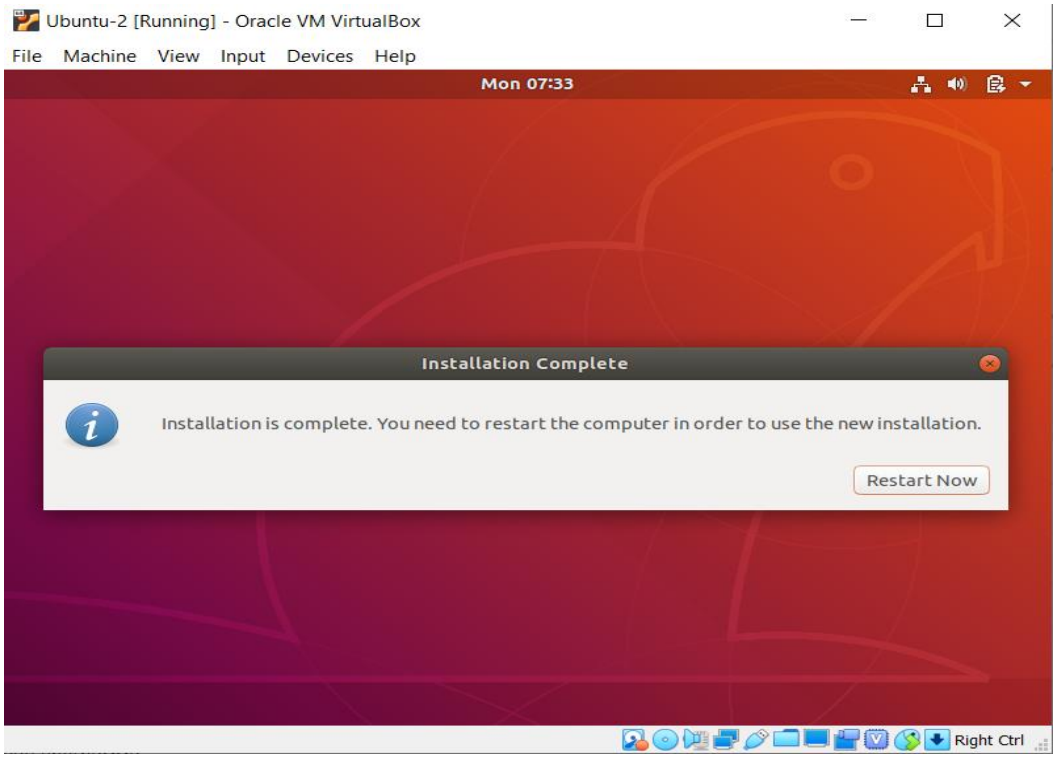


Рис. 1.15. Завершення установки Ubuntu

Після перезавантаження на екрані з’явиться робочий стіл Ubuntu (рис. 1.16).

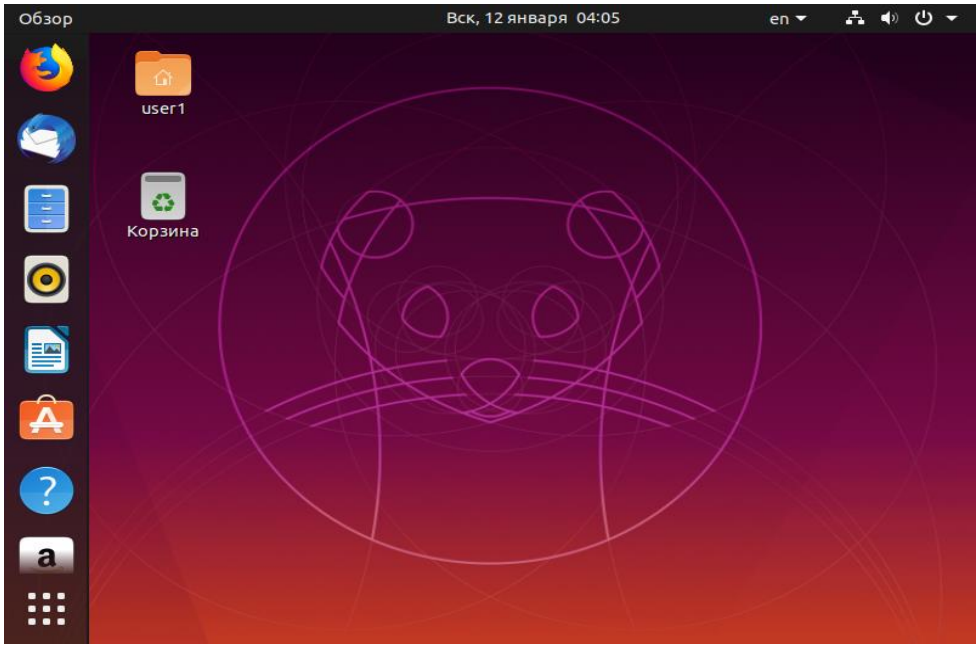


Рис. 1.16. Робочий стіл Ubuntu

4)Встановити доповнення гостьової операційної системи

Для забезпечення кращої інтеграції гостьової операційної системи (Ubuntu) з хост-машиною необхідно встановити доповнення гостьовій ОС. Для цього у вікні віртуальної машини треба вибрати пункт меню *Устройства ‣ Подключить образ диска Дополнений гостевой ОС* (рис. 1.17).

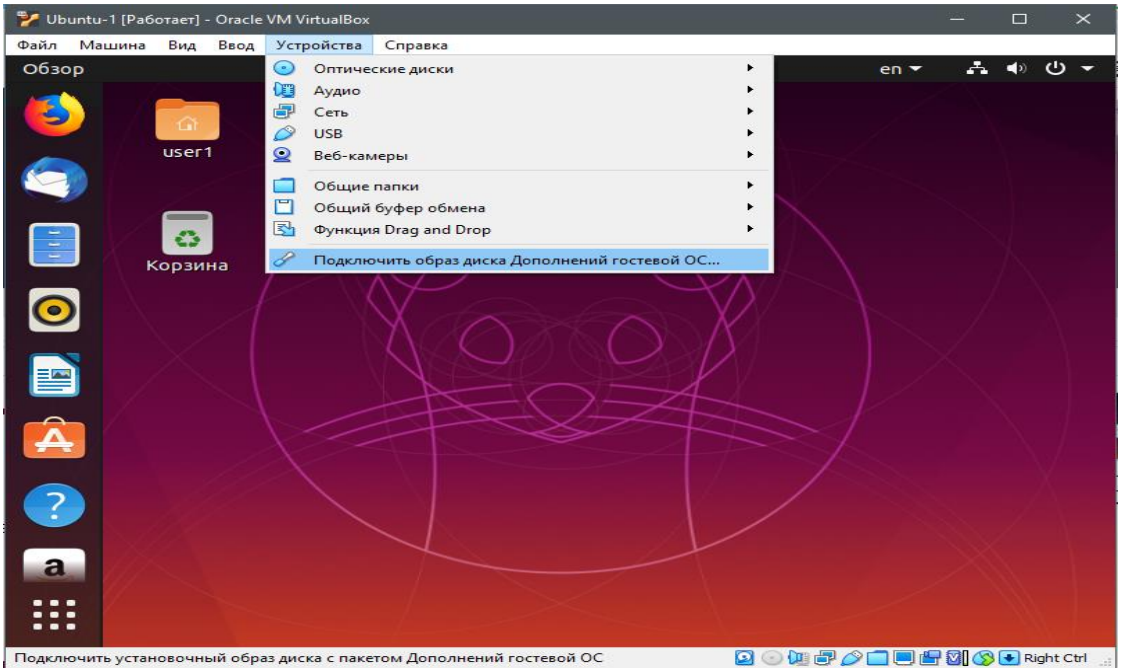


Рис. 1.17. Підключення образу диска гостьової ОС

Запуск установки доповнень гостьової ОС наведено на рис. 1.18

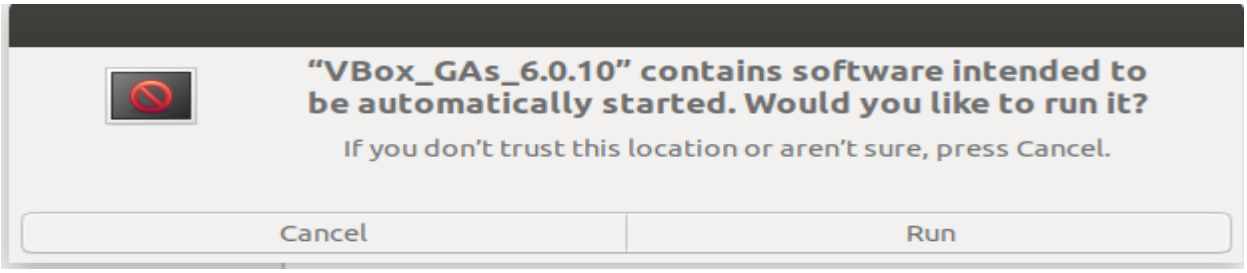


Рис. 1.18. Запуск установки доповнень гостьової ОС

По завершенні процесу установки (доведеться почекати пару хвилин) натиснемо клавішу «Enter» (рис. 1.19).

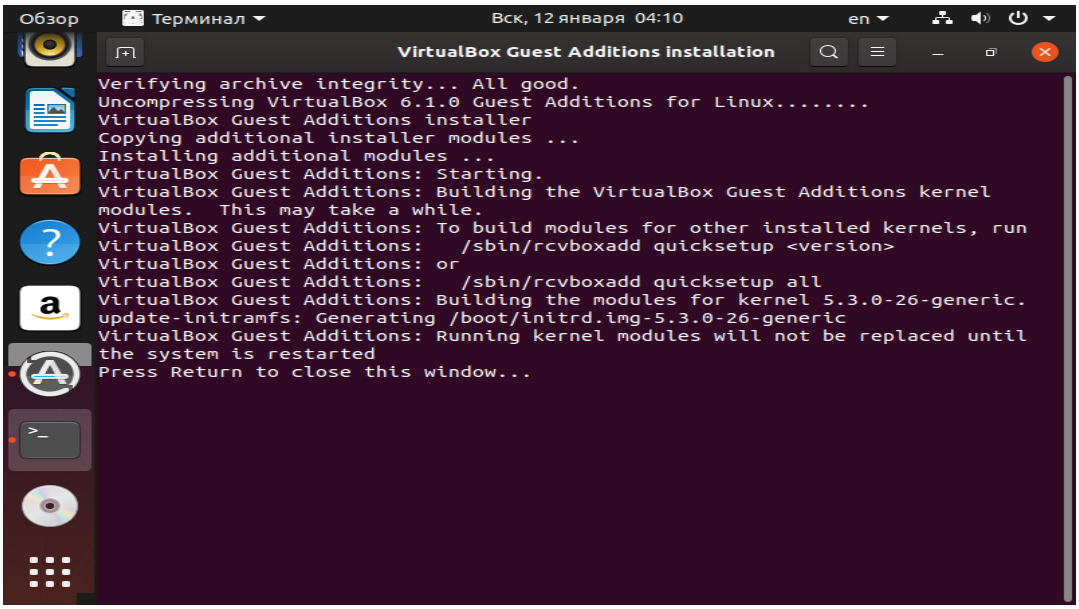


Рис. 1.19. Завершення установки доповнень гостьової ОС

Вимоги до звіту:

1. Оформити титульний лист згідно зразка кафедри.
2. Мета роботи.
3. Завдання.
4. Описати процес установки ОС Ubuntu.
5. Висновки по роботі.

Контрольні питання

1. Що таке Ubuntu?
2. Які існують версії ОС Ubuntu?
3. Що таке віртуальна машина?
4. Які операційні системи можна ставити на віртуальні машини?
5. У чому полягає концепція віртуальної машини?
6. Що таке VirtualBox? Для чого вона застосовується?
9. Назвіть основні особливості програми VirtualBox.
10. Які основні етапи установки ОС Ubuntu?
11. Як забезпечується інтеграція гостьової ОС с хост-машиною?

КОМП'ЮТЕРНИЙ ПРАКТИКУМ 2

Менеджери для роботи з пакетами програм в Linux

Мета роботи:

- набути навичок роботи з менеджерами пакетів програм Linux.

Теоретичні відомості

Управління пакетами або Менеджер пакетів - це програма, яка здійснює установку та підтримку (оновлення / за необхідності видалення) програмного забезпечення операційної системи. Пакети надаються користувачеві вже готовими до установки на операційну систему. Проте в Linux, завжди можна отримати початковий код того чи іншого програмного забезпечення для вивчення, поліпшення і компіляції. Також менеджер пакетів відстежує залежності між програмами та бібліотеками, зберігає систему в цілісності. У Linux пакети мають наступні характеристики:

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

- файли пакетів в Linux, на відміну від інсталяторів в 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 повинні бути встановлені для коректної роботи пакета; багато програмних пакетів залежать від бібліотечних пакетів; бібліотеки надають код, який й використовується багатьма програмами.

- пакети містять інформацію про версії, щоб пакетний менеджер міг визначити, який з двох пакетів новіший;

- пакети містять інформацію про архітектуру, щоб визначити тип центрального процесора (x86, x86-64, ARM і т. д.), для якого вони призначені; спеціальне позначення мають пакети, що не залежать від архітектури, наприклад, шрифти або теми робочого столу.

Формати пакетів програм Linux

Основними форматами пакетів, тобто форматами файлів, які використовуються системами управління пакетами операційних систем на основі Linux та GNU є:

- *бінарні (двійкові) пакети,* які є готовими відкомпільованими пакетами і закачані для конкретної системи, тобто це тільки виконувані файли;

- *пакети, що містять вихідні коди програм*, тобто це пакети, які потребують компіляції на локальній машині. Вони є більш універсальними, ніж бінарні, тому що можуть використовуватися для різних систем.

Бінарні пакети містять набагато більше інформації, яка полегшує роботу вашого менеджера пакетів, ніж просто скомпільовані файли.

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

При вирішенні різних завдань з управління пакетами програмного забезпечення, необхідно знати, що існують два типи утиліт: низькорівневі інструменти (здійснюють фактичну установку, оновлення та видалення файлів пакетів), і високорівневі інструменти (відповідають за виконання завдань по вирішенню залежностей і пошуку метаданих - так звані «дані про дані»).

Низькорівневі системи управління пакетами:

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

Високорівневі системи управління пакетів:

- 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 і dpkg-split.** Коротко про кожного з них:

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

Aptitude - інструмент для управління пакетами з командного рядка. По суті є зовнішнім інтерфейсом для інструменту *apt*, полегшує роботу з пакетами.

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

Tasksel - дозволяє користувачеві встановлювати всі відповідні пакунки пов'язані з певній задачі.

Deselect - менеджер пакетів працює через псевдо-меню, в даний момент замінений aptitude.

Dpkg-deb - працює з файлами архівів Debian.

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

Менеджери пакетів, засновані на Debian

Менеджер пакетів Dpkg

Ubuntu і Debian вважаються одними з найбільш широко використовуваних операційних систем на основі Linux. Їх менеджери пакетів є загальними, і належать до низькорівневої системи управління пакетів «*Dpkg*» скорочено від «Debian Package». Це скелет програмного забезпечення для управління 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: управління пакетами .deb

До корисних її опцій належать: ***--install, --remove*, а *-l*** перераховує пакети, інстальовані в системі.

1. Отримання короткої довідки # dpkg --help

2. Версія dpkg # dpkg --version

3. Dpkg, установка пакета, команда *dpkg --install,*

Для установки *.deb* пакета використовується ключ ***-i***:

 # dpkg -i flashpluginnonfree_2.8.2+squeeze1_i386.deb

Команда виконана по відношенню до пакету, який вже знаходиться в системі, перед інсталяцією видаляє попередню версію пакету.

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, для перегляду списку встановлених програм використовується ключ ***-l***: ***$ dpkg -l***

```
qwe@vb:~$ dpkg -l
Желаемый=неизвестно[u]/установить[i]/удалить[r]/вычистить[p]/зафиксировать[h]
| Состояние=не[n]/установлен[i]/настроен[c]/распакован[U]/частично настроен[F]/
           частично установлен[H]/trig-aWait/Trig-pend
|/ Ошибка?=(нет)/требуется переустановка[R] (верхний регистр
в полях состояния и ошибки указывает на ненормальную ситуацию)
||/ Имя                                          Версия
+++-==============================================-==============================
ii  accountsservice                               0.6.50-0ubuntu1
ii  acl                                           2.2.53-4
ii  acpi-support                                  0.143
ii  acpid                                         1:2.0.31-1ubuntu2
ii  adduser                                       3.118ubuntu1
ii  adwaita-icon-theme                            3.32.0-1ubuntu1
ii  aisleriot                                     1:3.22.8-1
ii  alsa-base                                     1.0.25+dfsg-0ubuntu5
ii  alsa-utils                                    1.1.8-1ubuntu1
ii  amd64-microcode                               3.20191021.1+really3.20181128.1ub
ii  anacron                                       2.3-27
ii  apg                                           2.2.3.dfsg.1-5
ii  app-install-data-partner                      19.04
ii  apparmor                                      2.13.2-9ubuntu6
```

Щоб дізнатися чи встановлена конкретна програма, потрібно вказати її ім'я:

$ dpkg -l nginx

За допомогою команди *$* **dpkg -L name package** виводиться список файлів пакета, наприклад: $ **dpkg -L finger**

```
qwe@vb:~$ dpkg -L finger
/.
/usr
/usr/bin
/usr/bin/finger
/usr/share
/usr/share/doc
/usr/share/doc/finger
/usr/share/doc/finger/BUGS
/usr/share/doc/finger/changelog.Debian.gz
/usr/share/doc/finger/copyright
/usr/share/man
/usr/share/man/man1
/usr/share/man/man1/finger.1.gz
```

5. Dpkg, видалити пакет, для видалення *.deb* пакета використовується ключ **-r (remove)** із зазначенням імені пакета, наприклад "flashpluginnonfree", повна назва "flashplugin-nonfree_3.2_i386.deb", вказувати не обов'язково.

$ **dpkg -r flashpluginnonfree**

Щоб видалити пакет разом з файлами конфігурації, замість -r, використовуйте ключ ***-P (purge):*** $ **dpkg -P flashpluginnonfree**

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

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

7. Dpkg, перевірити, встановлений пакет чи ні, використовується ключ -s (status) $ dpkg -s flashplugin-nonfree

8. Dpkg, відобразити місце розташування встановлених файлів пакетів, ключ -L: $ dpkg -L mysql-common

9. Dpkg, встановити всі пакети з конкретної директорії, використовуйте **ключі -R і --install**.

Наступна команда встановить всі * .deb файли з директорії debpackages:

$ **dpkg -R --install debpackages.**

Менеджер пакетів APT

Менеджер пакетів APT (скорочено від *Advanced Package Tool*), має інтерфейси: ***apt*** та ***aptitude***. *apt* набагато більш просунутий у функціональності у порівнянні з *dpkg*. Він також може встановлювати, видаляти і збирати пакети - однак його функціональність йде набагато далі. 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 операційних систем на основі Debian. Для роботи в командному рядку з дистрибутивами *Linux (Debian, Ubuntu)*, необхідно мати права адміністратора для використання *apt*. Для цього існують дві **утиліти - *Apt-get* і *Apt-cache.***

Утиліта *Apt-get* працює з бібліотекою APT (Advanced Packaging Tool) і використовується для установки нових пакетів програмного забезпечення, видалення та оновлення існуючих пакетів. Крім того *Apt-get* використовується для оновлення всієї операційної системи. Утиліта *Apt-cache* також використовується для пошуку пакетів програмного забезпечення в кеші *apt*, збору інформації про пакети, а також для пошуку готових пакетів для установки в операційних системах на базі Debian або Ubuntu. Як правило *apt-get і apt-cache* використовуються спільно: *apt-get* для маніпуляцій з пакетами, *apt-cache* для отримання інформації.

Для установки пакету використовується команда:

sudo apt-get install <package_name>

Однак, починаючи з Ubuntu 16.04 був доданий новий пакетний менеджер просто «***apt***». Для встановлення на комп'ютері програмного забезпечення, використовуючи менеджер ***apt,*** необхідно мати права адміністратора. Тому синтаксис установки пакета наступний

sudo apt install <package_name>

Для видалення пакета синтаксис наступний:

sudo apt remove <package_name>

У старого ***apt-**** існує досить ключів, в яких легко заплутатися.

Пакетний менеджер ***apt*** замінює старі ***apt-get і apt-cache***, в якому реалізовані найбільш використовувані команди по установці, видаленні пакетів, оновленню системи і пошуку пакетів.

Інтерфейс менеджера пакетів ***aptitude*** наступний:

```
qwe@vb:~$ apt-get help
apt 1.8.0 (amd64)
Использование: apt-get [параметры] команда
               apt-get [параметры] install|remove пакет1 [пакет2…]
               apt-get [параметры] source пакет1 [пакет2…]

apt-get — интерфейс командной строки для получения пакетов и
информации о них из доверенных источников, а также установки,
обновления и удаления пакетов вместе с их зависимостями.

Основные команды:
  update - получить новые списки пакетов
  upgrade - выполнить обновление
  install - установить новые пакеты (указывается имя пакета libc6, а не имя фай
ла libc6.deb)
  reinstall - переустановить пакеты (указывается имя пакета libc6, а не имя фай
ла libc6.deb)
  remove - удалить пакеты
  purge - удалить пакеты вместе с их файлами настройки
  autoremove - автоматически удалить все неиспользуемые пакеты
  dist-upgrade - обновить всю систему, подробнее в apt-get(8)
  dselect-upgrade - руководствоваться выбором, сделанным в dselect
  build-dep - настроить сборочные зависимости для пакета с исходным кодом
  clean - удалить скачанные файлы архивов
  autoclean - удалить старые скачанные файлы архивов
  check - проверить отсутствие нарушенных зависимостей
  source - скачать архивы с исходным кодом
  download - скачать двоичный пакет в текущий каталог
  changelog - скачать и показать журнал изменений заданного пакета
```

Інтерфейс менеджера пакетів ***apt*** наступний:

```
qwe@vb:~$ apt help
apt 1.8.0 (amd64)
Использование: apt [параметры] команда

apt — менеджер пакетов с интерфейсом командной строки. Он предоставляет
команды для поиска и управления, а также запросов информации о пакетах.
apt выполняет те же задачи, что и специализированные инструменты APT,
например apt-get и apt-cache, но по умолчанию задействует параметры,
которые больше подходят для интерактивного использования.

Основные команды:
  list - показать список пакетов на основе указанных имён
  search - искать в описаниях пакетов
  show - показать дополнительные данные о пакете
  install - установить пакеты
  reinstall - переустановить пакеты
  remove - удалить пакеты
  autoremove - автоматически удалить все неиспользуемые пакеты
  update - обновить список доступных пакетов
  upgrade - обновить систему, устанавливая/обновляя пакеты
  full-upgrade - обновить систему, удаляя/устанавливая/обновляя пакеты
  edit-sources - редактировать файл с источниками пакетов
```

Відмінність *apt* від *apt-get* полягає у тому, операції виконуються значно швидше (наприклад, install, remove)

Команда ***search*** аналогічна *apt-cache search*, використовується для пошуку пакета в репозиторіях. Різниця в тому, що ця команда виводить відсортований список за алфавітом.

Команда ***show*** аналогічна *apt-cache show*, виводить детальну інформацію про пакет. Різниця в тому, що тепер інформація більш коротка і по суті. Приховані більш технічні параметри, такі як хеш-кодування.

Команда ***update*** аналогічна *apt-get update*, оновлює інформацію про пакети в доданих репозиторіях в системі. Різниця в тому, що текст виконання став різнобарвним.

Команда ***list*** схожа на *dpkg list*, відображає список пакетів в залежності від додаткового ключа.

Ключі можуть бути наступними:

--upgradable - виведення списку пакетів, які можуть бути оновлені (є нові версії в репозиторіях);

--installed - виведення списоку усіх встановлених пакетів в системі;

--manual-installed - виведення списку пакетів, які встановлювалися користувачем в ручну;

--all-version – виведення усіх пакетів, описаних вище;

--verbose - відображення короткої інформації (що це за пакет програми) по всім пакетам.

Також можна вказувати частину імені пакетів, за якими потрібно отримати інформацію, вказавши зірочку на кінці, або безпосередньо повне ім'я пакета, наприклад: ***apt list zypper* --verbose***

Всі файли налаштувань *apt* зберігаються в директорії */etc/apt.*

apt.conf - */etc/apt/apt.conf* - основний файл налаштувань, який використовується усіма інструментами зі складу *apt*. Опис всіх можливих налаштувань і опцій можна прочитати в документації до нього: **man** apt.conf.

apt.conf.d - директорія містить в собі файли конфігурації, аналогічні по синтаксису *apt.conf*. За допомогою цієї директорії можна швидко і зручно

маніпулювати налаштуваннями *apt,* додаючи або видаляючи підготовлені файли з настройками.

auth.conf - файл, що містить ключі, для авторизації в репозиторіях. Наприклад, туди додаються логіни і паролі від репозиторіїв.

sources.list - файл з переліком репозиторіїв.

sources.list.d - директорія з файлами репозиториїв, за призначенням аналогічних *sources.list.* Кожний репозиторій описується в окремому файлі.

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

Git - це набір консольних утиліт, які відстежують і фіксують зміни в файлах як правило початкового коду проекту, дозволяють відкотитися на більш стару версію вашого проекту, порівнювати, аналізувати, зливати зміни і багато іншого. Цей процес і є контролем версій. Для встановлення Git -пакету треба виконати наступні команди:

sudo apt update **sudo apt install git**

Може виникнути помилка, що ресурс тимчасово недоступний. Такі помилки виникають, коли треба обновити систему. Тому для надійності треба перезавантажити систему, виконати три наступні команди:

sudo apt update оновлення даних, **sudo apt upgrade** оновлення системи,

sudo apt update оновлення даних,

а потім **sudo apt install git.**

Далі треба треба виконати дві команди (вказати ваше ім’я користувача та електронну пошту) **git config --global user.name "Your Name"**

git config --global user.email "youremail@domain.com"

Завдання:

1. Ознайомитися з теоретичними матеріалом по лабораторній роботі.
2. Опанувати команди для роботи з менеджерами пакетів.
3. Підготувати звіт для викладача про виконання лабораторної роботи і представити його.

Хід виконання роботи

1. Виконати команди управління менеджера пакетів *dpkg* з різними ключами.
2. Проаналізувати результат виконання команд.
3. Виконати команди менеджера *apt – help, show, update*, зробити знімки екрану.
4. Виконати перенаправлення введення-виведення команди *list* менеджера пакетів *apt* у файл.
5. Встановити пакет *finger* з використанням менеджера пакетів *apt.*
6. Вивести детальну інформацію про пакет *xcolors* для *apt.*
7. Віднайти за допомогою команди пошуку пакета утиліти apt назву пакета консольного файлового менеджера «Midnight Commander»/
8. Встановити консольний файловий менеджер «Midnight Commander» (mc).
9. Встановити команду *ifconfig / net-tools,* яка відображає стан поточної конфігурації мережі або ж налаштовує мережевий інтерфейс.
10. Встановити git – систему контролю версій.

Підготувати звіт

1. Описати хід виконання поставлених завдань, надаючи знімок екрану (screenshot) виконання команди.
2. Висновки по роботі.

Контрольні питання

1. Що таке менеджер пакетів?
2. Які особливості пакетів Linux?
3. Які вам відомі формати пакетів Linux?
4. Які вам відомі системи управління пакетами?
5. Які вам відомі пакети, засновані на Debian?
6. Які можливості команди *dpkg*?
7. Які особливості менеджер пакетів *apt*?
8. Які можливості пакету git?

КОМП'ЮТЕРНИЙ ПРАКТИКУМ 3

Робота в оболонці bash, середовище оточення

Мета роботи:

- набути навичок налаштування програмного середовища Linux.

Теоретичні відомості

Командні оболонки ОС Unix

У системах Unix використовуються різні командні оболонки (command shells), які називаються також командними процесорами або інтерпретаторами команд. Командна оболонка забезпечує взаємодію між користувачем та ядром системи.

Серед них найбільш відомі і поширені:

• *sh (Bourne shell) - оболонка Борна* (не дуже зручна в роботі);

• *csh (C-shell) - оболонка С* (зручніша у порівнянні з sh, але несумісна з нею по командній мові);

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

• *bash (Bourne-Again Shell)* - оболонка «Борна» (зручна для інтерактивної роботи, створена на основі sh і багато в чому з нею сумісна).

Тип оболонки, як правило, можна визначити за останнім символом запрошення:

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

- знак амперсанда «&» відповідає оболонці csh.

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

Основними функціями командних оболонок є:

• організація діалогу з користувачем (введення команд);

• виконання внутрішніх команд;

• запуск зовнішніх програм;

• виконання командних файлів.

Командні мови в різних оболонках розрізняються, *а стандартною прийнято вважати командну мову оболонки bash.*

В графічній оболонці програма «Термінал», яка запускається комбінацією клавіш «Ctrl+Alt+T» або ярлик із загального меню програм  , це важливий елемент операційної системи, який дозволяє запускати програми, створювати папки, копіювати і видаляти файли, встановлювати додатки і т.д. Системна утиліта, в яку ви передаєте ці команди, називається *Shell* або командна оболонка. За замовчуванням в Ubuntu використовується командна оболонка, яка називається *Bash*.

При вході в командний інтерпретатор відкривається вікно, в якому відображається:

qwe – ***ім'я облікового запису користувача***, *vb* – ***ім'я комп'ютера***, символ «:» - ***розділювач***,

«~» - каталог виконання команди (***домашній каталог***).

Синтаксис команд в Терміналі

Команди Терміналу, як правило, складаються з назви програми, ключа і значення. В загальному вигляді виглядають так: ***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 приймати виконувана програма;

[значення] - даний параметр може приймати в якості аргументу цифри, текст, спеціальні символи і навіть змінні.

Оболонка має свої налаштування та забезпечує доступ до ресурсів системи, користувач має свої налаштування при роботі з системою та своїми додатками. Для цього використовується середовище оточення.

Середовище оточення - це область, яка містить визначальні властивості системи у вигляді змінних, і це середовище оболонка будує при кожному запуску сесії. Такі змінні можуть містити загальні налаштування системи, параметри графічної або командної оболонки, дані про вподобання користувача, місце розміщення виконуваних файлів в системі, ім'я текстового редактора і багато іншого. Змінні

оточення дозволяють простим і надійним способом передавати налаштування відразу для безлічі додатків.

Таким чином, *змінні оточення* в Linux - це спеціальні змінні, які визначені оболонкою і використовуються програмами під час виконання. Вони можуть визначатися системою і користувачем. Системні змінні оточення Linux визначаються системою і використовуються програмами системного рівня.

Змінні оточення бувають трьох типів:

1. *Локальні змінні оточення*.

Це змінні, які визначені лише для поточної сесії. Вони будуть зразу стерті після завершення сесії, будь то віддалений доступ або емулятор терміналу. Вони не зберігаються ні в яких файлах, а створюються і видаляються за допомогою спеціальних команд.

2. *Змінні користувача в середовищі оболонки.*

Ці змінні оболонки в Linux визначаються для конкретного користувача і завантажуються кожний раз, коли він входить в систему за допомогою локального терміналу, або ж підключається віддалено. Такі змінні зберігаються в файлах конфігурації: ***.bashrc***, ***.bash_profile***, ***.bash_login***, ***.profile*** або в інших файлах, розміщених в директорії користувача. Ці файли є налаштуванням оболонки користувача. Вони складаються з команд *bash* і виконуються перед тим як запустити оболонку або завантажити систему.

Різниця цих файлів полягає у тому, коли вони виконуються, а саме:

.profile та *.bash_profile* виконуються при вході в систему, тобто один раз- це при введенні логін користувача. Файл *.bashrc* виконується кожний раз, коли користувач відкриваєте нове вікно терміналу (аналог автозавантаження в Windows).

Bash також працює зі *скриптами (сценаріями* - ***scripting language)***, тобто списком команд, які записані у файл. Сценарій – це програма, яка працює з готовими компонентами.

3. *Системні змінні оточення*

Ці змінні доступні у всій системі та для всіх користувачів. Вони завантажуються при старті системи з системних файлів конфігурації: ***/etc/environment***, ***/etc/profile***, ***/etc/profile.d/*.sh, /etc/bash.bashrc:***

- ***/etc/environment*** –системний файл конфігурації, який означає, що він використовується всіма користувачами. Він належить *root*, тому необхідно бути адміністратором і використовувати його *sudo* для зміни;

- ***~/.profile*** є одним із сценаріїв ініціалізації особистої оболонки вашого власного користувача. Кожний користувач має один такий файл і може його редагувати.

- ***/etc/profile ma /etc/profile.d/*.sh*** є сценаріями глобальної ініціалізації, які є еквівалентними ***~/.profile*** для кожного користувача. Глобальні сценарії виконуються раніше користувацьких сценаріїв; і *main* ***/etc/profile*** виконує усі **.sh* сценарії з директорії /***etc/profile.d/*** безпосередньо перед виходом;

- ***/etc/bash.bashrc*** - це загальносистемна версія файлу *~/.bashrc*. ОС Ubuntu налаштована за замовчуванням для виконання цього файлу, коли користувач вводить оболонку або середовище робочого столу.

Файл */etc/environment* зазвичай містить тільки наступний рядок:

PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games"

Він встановлює змінну PATH для всіх користувачів в системі в це значення за замовчуванням, яке не повинно бути змінено основним способом. Принаймні, ви не повинні видалити будь - який з таких важливих шляхів, як */bin, /sbin, /usr/bin* та /usr/sbin з нього.

Цей файл читається як один з перших файлів конфігурації кожною оболонкою кожного користувача, це не скрипт оболонки.

Середовище має вигляд рядка, що містить пари виду «*ключ-значення*».

Ключ – це ім’я змінної. Кілька значень, як правило, поділяються символом двокрапки (:). Кожна пара, в цілому, виглядає таким чином:

КЛЮЧ = значення1: значення2: ...

Якщо ж значення містить пробіли, потрібно використовувати подвійні лапки:

КЛЮЧ = "значення з пробілами".

В даному випадку під ключем маються на увазі змінні одного з двох існуючих видів: *змінні середовища або змінні оболонки*.

Змінні середовища - це змінні, які були визначені для поточної оболонки і успадковуються усіма дочірніми оболонками або процесами. Змінні середовища використовуються для передачі інформації процесам, які запущені з оболонки.

Змінні оболонки - це змінні, які містяться виключно в оболонці, в якій вони були встановлені або визначені. Вони часто використовуються для відстеження поточних даних (наприклад, поточного робочого каталогу).

Зазвичай такі змінні позначаються за допомогою *великих літер*. Це допомагає користувачам розрізняти змінні середовища в інших контекстах.

У сценаріях зазвичай використовуються оголошення виду:

ім'я_змінної="значення змінної",

але конкретний синтаксис залежить від інтерпретатора, який використовується.

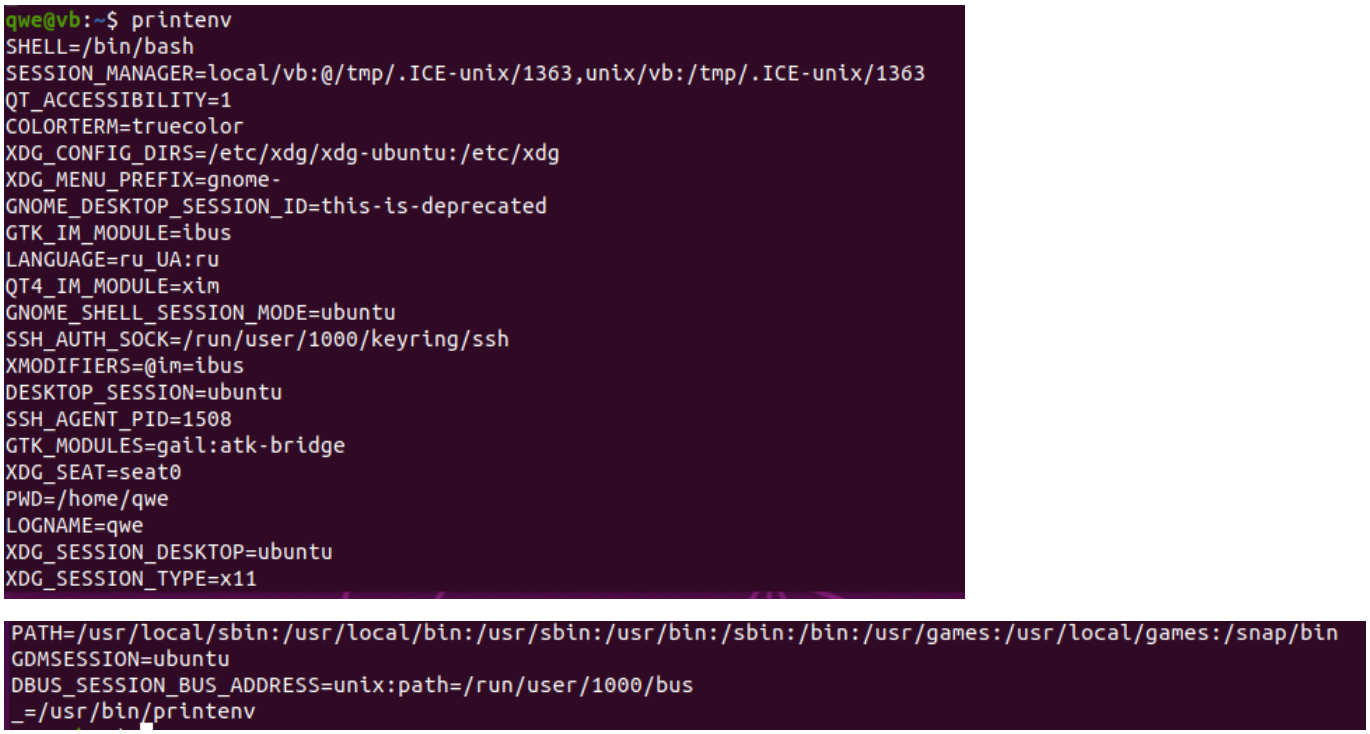
Для отримання значення змінної необхідно перед її ім'ям поставити символ долара. Також іноді потрібно її ім'я взяти у дужки (наприклад, в сценаріях утиліти *make*).

Виведення змінних оболонки (set) і середовища (env або *printenv)*

Кожна сесія відстежує свої змінні оболонки і середовища. Вивести їх можна кількома способами. Щоб переглянути *список всіх змінних середовища*, використовуйте команди *env* або *printenv*. За замовчуванням команди виведуть однаковий результат ***env***

```
qwe@vb:~$ env
SHELL=/bin/bash
SESSION_MANAGER=local/vb:@/tmp/.ICE-unix/1363,unix/vb:/tmp/.ICE-unix/1363
QT_ACCESSIBILITY=1
COLORTERM=truecolor
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg
XDG_MENU_PREFIX=gnome-
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
GTK_IM_MODULE=ibus
LANGUAGE=ru_UA:ru
QT4_IM_MODULE=xim
GNOME_SHELL_SESSION_MODE=ubuntu
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
XMODIFIERS=@im=ibus
DESKTOP_SESSION=ubuntu
SSH_AGENT_PID=1508
GTK_MODULES=gail:atk-bridge
XDG_SEAT=seat0
PWD=/home/qwe
LOGNAME=qwe
XDG_SESSION_DESKTOP=ubuntu
XDG_SESSION_TYPE=x11
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
GDMSESSION=ubuntu
DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus
_=/usr/bin/env
```

printenv



Це типовий приклад результату, який виведений командами *env та printenv.* Ці команди відрізняються лише кількома індивідуальними функціями. Наприклад, *printenv* може запитувати значення окремих змінних:

printenv SHELL

/bin/bash

Команда *env* дозволяє змінювати середовище, в якій запущені програми, передаючи набір значень змінних в команду:

env VAR1="blahblah" command_to_run command_options

Дочірні процеси зазвичай успадковують змінні середовища батьківського процесу, що дає можливість змінювати значення або вносити додаткові змінні для дочірніх процесів. При виведенні команди *printenv* багато змінних середовища створені за допомогою системних файлів і процесів без втручання користувача.

Для *перегляду змінних оболонки* використовується *команда* ***set.*** *При введенні без додаткових параметрів команда set виводить список всіх змінних оболонки, змінних середовища, локальних змінних і функцій оболонки:*

set

BASH=/usr/bin/bash

BASHOPTS=checkwinsize:cmdhist:expand_aliases:extglob:extquote:force_fignore:histappend:interactive_comments:login_shell:progcomp:promptvars: sourcepath

BASH_ALIASES=()

BASH_ARGC=()

BASH_ARGV=()

BASH_CMDS=()

. . .

Цей список досить довгий. Для виведення списку у більш зручному форматі, відкрийте його за допомогою програми-пейджера: **set | less.**

```
BASH=/usr/bin/bash
BASHOPTS=checkwinsize:cmdhist:complete_fullquote:expand_aliases:extglob:extquote
:force_fignore:globasciiranges:histappend:interactive_comments:progcomp:promptva
rs:sourcepath
BASH_ALIASES=()
BASH_ARGC=([0]="0")
BASH_ARGV=()
BASH_CMDS=()
BASH_COMPLETION_VERSINFO=([0]="2" [1]="8")
BASH_LINENO=()
BASH_SOURCE=()
BASH_VERSINFO=([0]="5" [1]="0" [2]="3" [3]="1" [4]="release" [5]="x86_64-pc-linu
x-gnu")
BASH_VERSION='5.0.3(1)-release'
CLUTTER_IM_MODULE=xim
COLORTERM=truecolor
COLUMNS=80
DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus
DESKTOP_SESSION=ubuntu
DIRSTACK=()
DISPLAY=:0
EUID=1000
GDMSESSION=ubuntu
```

Список містить величезну кількість додаткової інформації, яка в даний момент не потрібна (наприклад, деякі функції *bash*). Щоб у більш зручному форматі вивести результат, потрібно запустити команду *set* в *режимі POSIX, який пропускає функції оболонки.* Це потрібно виконати в субоболонці, щоб не змінити поточне середовище:

set -o posix; set

Команда виведе усі змінні середовища і оболонки.

Основні змінні середовища і оболонки

Деякі особливо корисні змінні середовища і оболонки використовуються дуже часто. Нижче наведено **список основних змінних середовища**:

• SHELL - описує оболонку, яка є інтерпретаторам введених команд. У більшості випадків за замовчуванням встановлена оболонка ***bash***, але це значення можна змінити в разі потреби. `SHELL=/bin/bash`

• TERM - вказує вид терміналу, який емулюється при запуску оболонки. Залежно від операційних вимог можна емулювати різні апаратні термінали.

```
TERM=xterm-256color
```

•LOGNAME - реєстраційне ім’я користувача. `LOGNAME=qwe`

• USER - поточний користувач. `USER=qwe`

• PWD - поточний робочий каталог. `PWD=/home/qwe`

• LS_COLORS - визначає кольорові коди, які використовуються для кольорового виведення результату команди *ls*. Таке виведення допомагає користувачеві швидше прочитати результат команди (наприклад, швидко розрізнити типи файлів).

```
LS_COLORS='rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:or=40;31;01:mi=00
:su=37;41:sg=30;43:ca=30;41:tw=30;42:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:*.arc=01;31:*.arj=01
```

• PATH - список каталогів, до яких звертається система при виконанні команд. Коли користувач запускає команду, система перевіряє ці каталоги у зазначеному порядку в пошуках виконуваного файлу.

```
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
```

• LANG - поточні налаштування мови та локалізації, в тому числі кодування символів. `LANG=ru_UA.UTF-8`

• HOME - ім’я домашнього каталогу поточного користувача. `HOME=/home/qwe`

• _: остання виконана команда `_=/usr/bin/printenv`.

Список основних змінних оболонки:

• BASHOPTS: список опцій, які використовуються при виконанні *bash*.

```
BASHOPTS=checkwinsize:cmdhist:complete_fullquote:expand_aliases:extglob:extquote:force_fignore:globasciirang
es:histappend:interactive_comments:progcomp:promptvars:sourcepath
```

• BASH_VERSION: запущена версія *bash* в зрозумілій формі.

```
BASH_VERSION='5.0.3(1)-release'
```

• BASH_VERSINFO: версія *bash* в машиночитаємому форматі.

```
BASH_VERSINFO=([0]="5" [1]="0" [2]="3" [3]="1" [4]="release" [5]="x86_64-pc-linux-gnu")
```

• COLUMNS: визначає ширину виведення у стовбцях. `COLUMNS=108`

• DIRSTACK: стек каталогів, доступних командам *pushd і popd.*

```
DIRSTACK=()
```

• HISTFILESIZE: максимальна кількість рядків, що міститься у файлі історії команд. `HISTFILESIZE=2000`

• HISTSIZE: Кількість команд, які необхідно запам'ятовувати в списку історії.

`HISTSIZE=1000`

• HOSTNAME: поточне ім'я хоста. `HOSTNAME=vb`

• IFS: Внутрішній розділювач полів введення у командному рядку. За замовчуванням встановлено пробіл. `IFS=' '`

• PS1: визначає рядок первинного запрошення - вид командного рядка при запуску сесії оболонки. Змінна PS2 встановлює рядок вторинного запрошення, якщо команда займає декілька рядків.

```
PS1='\[\e]0;\u@\h: \w\a\]${debian_chroot:+($debian_chroot)}\[\033[01;32m\]\u@\h\[\033[00m\]:\[\033[01;34m\]\
w\[\033[00m\]\$ '
```

• SHELLOPTS: параметри оболонки, які можна встановити за допомогою *set*.

```
SHELLOPTS=braceexpand:emacs:hashall:histexpand:history:interactive-comments:monitor:posix
```

• UID: унікальний ідентифікатор поточного користувача. `UID=1000`

Приклад ~/.profile

```
# ~/.profile: executed by the command interpreter for login shells.
# This file is not read by bash(1), if ~/.bash_profile or ~/.bash_login
# exists.
# see /usr/share/doc/bash/examples/startup-files for examples.
# the files are located in the bash-doc package.

# the default umask is set in /etc/profile; for setting the umask
# for ssh logins, install and configure the libpam-umask package.
#umask 022

# if running bash
if [ -n "$BASH_VERSION" ]; then
    # include .bashrc if it exists
    if [ -f "$HOME/.bashrc" ]; then
        . "$HOME/.bashrc"
    fi
fi

# set PATH so it includes user's private bin if it exists
if [ -d "$HOME/bin" ] ; then
    PATH="$HOME/bin:$PATH"
fi

# set PATH so it includes user's private bin if it exists
if [ -d "$HOME/.local/bin" ] ; then
    PATH="$HOME/.local/bin:$PATH"
fi
```

Команда ***date*** виводить поточну дату.

Утиліта ***grep*** використовується для пошук рядка у тексті за шаблоном. В якості шаблону використовуються регулярні вирази. Ключі цієї утиліти можна подивитися за допомогою команди ***man grep***. Наприклад, grep 'word' filename.

Облікові записи користувачів знаходяться у файлі /etc/passwd. Для пошуку свого облікового запису можна скористатися командою ***grep qwe /etc/passwd.***

```
qwe@vb:~$ grep qwe /etc/passwd
qwe:x:1000:1000:qwe,,,:/home/qwe:/bin/bash
```

Команда ***df*** показує обсяг вільного простору на диску для всіх файлових систем.

Приклад використання: ***df -h***

- h: показує об'єм диска в кіло-і мегабайтах, а не байтах.

Команда ***free*** показує обсяг вільної та зайнятої оперативної пам'яті.

Приклад використання: ***free -mt***

-m: показує обсяг в мегабайтах, а не кілобайтах;, **-g:** показує обсяг в гігабайтах;

-t: відображає підсумковий рядок.

У випадку, коли виникає потреба послідовно виконати декілька команд, є можливість записати виклики декількох команд в одному рядку. Роздільником команд є «;» (крапка з комою). При цьому для виконання наступної команди не треба чекати завершення попередньої.

Синтаксис наступний: **command_1; command_2; command_3**

У випадку, коли необхідно виконання декількох команд в одному рядку і умовою є виконання наступної команди лише в тому випадку, якщо успішно відпрацює попередня. Синтаксис наступний: **command_1 && command_2**

Наприклад, **sudo apt update && sudo apt upgrade**

Слід пам’ятати, що при випадковому натисканні комбінації «Ctrl+S» відбувається блокування терміналу. Для переведення терміналу в робочий стан треба натиснути комбінацію клавіш «Ctrl+Q».

За допомогою символів «**!!**» можна викликати всю попередню команду. Наприклад, для виконання команди з правами суперкористувача можна набрати **sudo !!**

Для призупинення виконання команди, яка виконується, необхідно натиснути «Ctrl+C».

Команда ***history*** виводить команди сеансу користувача.

Завдання:

1. Ознайомитися з теоретичними матеріалом по лабораторній роботі.
2. Опанувати команди по роботі зі змінними оболонки та середовища.
3. Підготувати звіт для викладача про виконання лабораторної роботи і представити його.

Хід виконання роботи

1. Вивести змінні оболонки.

2. Вивести змінні середовища.

3. Вивести поточну дату, обсяг вільного простору на диску, обсяг оперативної пам’яті.

4. За допомогою команди *history* виведіть команди, які ви використовували.

Підготувати звіт

1. Описати хід виконання поставлених завдань, надаючи знімок екрану (screenshot).
2. Висновки по роботі.

Контрольні питання

1. Що таке командна оболонка?
2. Які функції виконує командна оболонка?
3. Що таке середовище оточення?
4. Що таке змінні оточення?
5. Які типи змінних оточення вам відомі?
6. Які основні змінні середовища?
7. Які основні змінні оболонки?

КОМП'ЮТЕРНИЙ ПРАКТИКУМ 4

Робота з файловою системою ОС Linux

Мета роботи:

- набуття навичок налаштування облікових записів користувачів, створювання груп користувачів;

- набуття досвіду роботи з файлами і каталогами в ОС Linux, налаштування прав на доступ до файлів і каталогів.

Теоретичні відомості

Налаштування облікових записів користувачів

В Unix-системах реєстрація користувачів здійснюється в системному файлі */etc/passwd*. Вміст цього файлу - це послідовність текстових рядків. Кожний рядок відповідає одному зареєстрованому в системі користувачу і містить сім полів, розділених символами двокрапки, а саме:

- реєстраційне ім'я користувача;

- зашифрований пароль;

- значення UID (*user ID*);

- значення GID основної групи (*group ID*);

- коментар (може містити розширену інформацію прокористувача, наприклад, ім'я, посаду, телефони і т. п.);

- домашній каталог;

- командна оболонка користувача.

Файл */etc/passwd* повинен бути доступний для читання всім користувачам.

Інформація про групи користувачі, які є системі, міститься у файлі реєстрації груп користувачів */etc/group*. Файл */etc/group* являє собою набір рядків, по одній для кожної зареєстрованої групи користувачів. Кожний рядок містить чотири поля, розділених двокрапкою:

- реєстраційне ім'я групи;

- пароль групи (пусте поле, тому що групам не призначають паролі);

- значення GID, що відповідає даній групі;

- розділений комами список користувачів, які входять в групу (може бути порожнім).

В ОС Ubuntu введено особливий режим використання облікового запису суперкористувача з ім'ям *root*. Обліковий запис *root* є головною обліковим записом в Linux та інших Unix-подібних операційних системах. Цей обліковий запис має доступ до всіх команд і файлів в системі з повними дозволами на читання, запис і виконання. Він використовується для виконання будь-яких системних задач: створення / оновлення / отримання доступу / видалення облікових записів інших користувачів, установки / видалення / оновлення програмних пакетів і багато чого іншого. Оскільки користувач *root* має абсолютними повноваженнями, будь-які виконувані ним дії є критичними для системи. У зв'язку з цим будь-які помилки користувача *root* можуть мати величезний вплив на нормальну роботу системи. Тому рекомендується відключити доступ до аккаунту та створити обліковий запис адміністратора, який буде налаштований на отримання привілеїв користувача *root* за допомогою команди ***sudo*** для виконання критичних завдань на сервері.

Якщо необхідно мати обліковий запис суперкористувача root, її можна активувати за допомогою наступної команди: ***sudo passwd root.***

Зазначена команда ініціює стандартну діалогову процедуру призначення пароля користувача (в даному випадку - суперкористувача з ім'ям *root*). Відповідно *для відключення облікового запису root* слід використовувати наступну команду: ***sudo passwd -l root.***

Реально, зазначена команда не видаляє, а лише блокує обліковий запис.

Додавання користувача здійснюється наступною командою:

sudo useradd -m <ім'я користувача>, ключ ***«-m»*** означає створити домашній каталог для користувача.

Наприклад, ***sudo useradd -m user1***

Перевіряємо чи створився користувач ***ls -l /home.***

Перевіряємо чи є у нього пароль ***cat /etc/passwd.***

Переконуємося, що поки паролю немає у користувача **user1** ***sudo cat /etc/shadow .***

Вводимо пароль ***sudo passwd <ім'я користувача>, sudo passwd user1.*** Двічі вводимо пароль. Переконуємося, що у *user1* з'явився пароль ***sudo cat/etc/shadow.***

Переходимо до домашнього каталогу *user1* ***cd /home/user1***

```
qwe@vb:~$ cd /home/user1
qwe@vb:/home/user1$ ls -l
итого 12
-rw-r--r-- 1 user1 user1 8980 апр 16  2018 examples.desktop
qwe@vb:/home/user1$
```

Перевіряємо що в ньому є ***ls -l***

Виявляємо, що є файл *examples.desktop*. При створюванні користувача створюється папка скелет «skel». Переходимо у цю папку ***cd /etc/skel/.*** Все, що є в папці */etc/skel/*, при створенні користувача копіюється в його папку. Створимо папку ***Desktop (команда mkdir)*** і файл ***myfile.txt (команда touch)***:

sudo mkdir Desktop

sudo touch myfile.txt

Переглянемо каталог ***ls -l***

Змінити користувача ***su <ім'я користувача>,*** ***su user1*** та переглянути вміст ***ls -l***

```
qwe@vb:/etc/skel$ su user1
Пароль:
$ ls -l
итого 16
drwxr-xr-x 2 root root 4096 янв 17 19:40 Desktop
-rw-r--r-- 1 root root 8980 апр 16  2018 examples.desktop
$
```

Додаємо другого користувача ***sudo useradd -m user2***

Перевіряємо чи з'явився user2 ***ls -l /home***

```
$ su qwe
Пароль:
qwe@vb:/etc/skel$ sudo useradd -m user2
qwe@vb:/etc/skel$ ls -l /home
итого 12
drwxr-xr-x 21 qwe   qwe   4096 янв 17 18:25 qwe
drwxr-xr-x  2 user1 user1 4096 янв 17 18:34 user1
drwxr-xr-x  3 user2 user2 4096 янв 17 19:54 user2
qwe@vb:/etc/skel$
```

Створили папку Video і файл newfile.txt

```
qwe@vb:/etc/skel$ ls -l /home/user2
итого 16
drwxr-xr-x 2 user2 user2 4096 янв 17 19:40 Desktop
-rw-r--r-- 1 user2 user2 8980 апр 16  2018 examples.desktop
qwe@vb:/etc/skel$ sudo mkdir Video
qwe@vb:/etc/skel$ sudo touch newfile.txt
qwe@vb:/etc/skel$ ls -l
итого 20
drwxr-xr-x 2 root root 4096 янв 17 19:40 Desktop
-rw-r--r-- 1 root root 8980 апр 16  2018 examples.desktop
-rw-r--r-- 1 root root    0 янв 17 19:58 newfile.txt
drwxr-xr-x 2 root root 4096 янв 17 19:57 Video
qwe@vb:/etc/skel$
```

Видаляємо користувача ***sudo userdel <ім'я користувача>,*** ***sudo userdel user1,*** але його папки залишаються. Щоб повністю його видалити разом з папками треба вказати ключ ***-r (remove).***

sudo userdel -r user1.

Створюємо групу ***sudo groupadd <ім'я групи>,***

sudo groupadd Programer і ще одну

sudo groupadd Marketing

Перевіряємо створилися чи групи ***cat /etc/group***

```
qwe@vb:/etc/skel$ sudo groupadd Programer
qwe@vb:/etc/skel$ sudo groupadd Marketing
qwe@vb:/etc/skel$ cat /etc/group
root:x:0:
daemon:x:1:
bin:x:2:
```

```
qwe:x:1000:
sambashare:x:129:qwe
systemd-coredump:x:999:
user2:x:1002:
Programer:x:1003:
Marketing:x:1004:
qwe@vb:/etc/skel$
```

Видалити групу ***sudo groupdel <ім'я групи>,*** ***sudo groupdel Marketing***

Перевіряємо чи є запис ***cat /etc/group***

```
qwe@vb:/etc/skel$ sudo groupdel Marketing
qwe@vb:/etc/skel$ cat /etc/group
root:x:0:
daemon:x:1:
bin:x:2:
```

```
qwe:x:1000:
sambashare:x:129:qwe
systemd-coredump:x:999:
user2:x:1002:
Programer:x:1003:
qwe@vb:/etc/skel$
```

Видалилась група *Marketing*. Додаємо користувача ***user2*** в групу. ***usermod*** – (mod- modification), -a (addition), G (group)

sudo usermod -aG Programer user2

```
qwe@vb:/etc/skel$ sudo usermod -aG Programer user2
```

Перевіряємо де знаходиться користувач user2 ***id user2***

```
qwe@vb:/etc/skel$ id user2
uid=1002(user2) gid=1002(user2) группы=1002(user2),1003(Programer)
qwe@vb:/etc/skel$
```

Як видалити користувача user2 з групи Programer?

sudo deluser user2 Programer

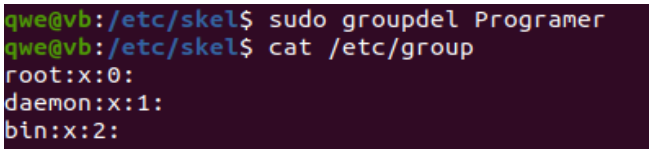
Перевіряємо id user2

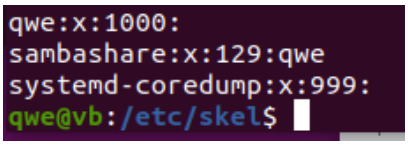
Видаляємо user2 ***sudo userdel -r user2***

```
qwe@vb:/etc/skel$ ls -l /home
итого 12
drwxr-xr-x 21 qwe   qwe   4096 янв 17 18:25 qwe
drwxr-xr-x  2  1001  1001 4096 янв 17 18:34 user1
drwxr-xr-x  3 user2 user2 4096 янв 17 19:54 user2
qwe@vb:/etc/skel$ sudo userdel -r user2
userdel: почтовый ящик user2 (/var/mail/user2) не найден
qwe@vb:/etc/skel$
```

```
qwe@vb:/etc/skel$ sudo userdel -r user2
userdel: почтовый ящик user2 (/var/mail/user2) не найден
qwe@vb:/etc/skel$ sudo userdel -r user1
userdel: пользователь «user1» не существует
qwe@vb:/etc/skel$ ls -l /home
итого 8
drwxr-xr-x 21 qwe  qwe  4096 янв 17 18:25 qwe
drwxr-xr-x  2 1001 1001 4096 янв 17 18:34 user1
qwe@vb:/etc/skel$
```

Видаляємо групу ***Programer***





Видалено

Команда *uname* виводить інформацію про операційну систему, яка встановлена

Команда *users* відображає короткий перелік користувачаів, які працюють в системі в даний момент

Визначення ідентифікаторів користувачів і груп

Щоб визначити UID користувача, GID та ім'я його основної групи, а також список інших груп, до якого включено користувача, використовується команда ***id.*** У разі її використання без аргументів, команда виведе інформацію про поточного користувача. Якщо ж вказати в якості аргументу ім'я зареєстрованого користувача, виведення команди буде відповідати зазначеному користувачеві.

Окремим випадком команди *id* є команда *groups*. Вона видає список імен всіх груп, в яких розташований поточний або вказаний користувач.

Введення команди ***who*** без аргументів дозволяє отримати список користувачів, які працюють в даний момент в системі. Якщо ж набрати ***whoami***, система виведе інформацію про поточного користувача. Додаткову інформацію про всіх перерахованих командах можна отримати за допомогою команди *man*, наприклад *$ man who.*

Файлова система

В 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), призначений для зберігання файлів. За замочуванням встановлюється файлова система *ext4fs,* яка є стандартом. При доступі до будь-якої файлової системи ОС Linux дані представляються у *вигляді ієрархії каталогів з розташованими в них файлами разом з ідентифікаторами власників і груп, бітами прав доступу та іншими атрибутами*. Вершиною ієрархічної структури файлової системи є каталог «/», який називається кореневим (рис. 1).

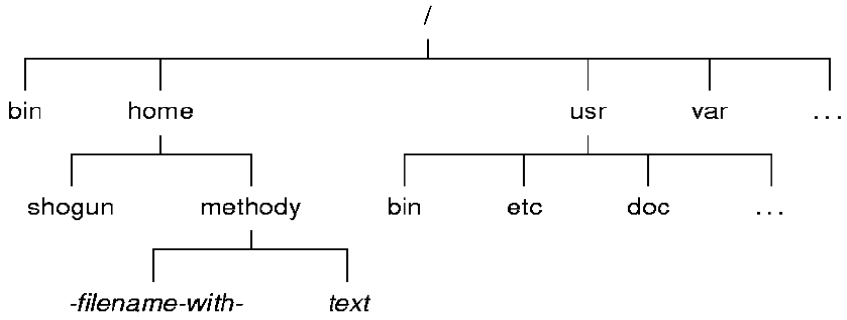


Рис. 1 Дерево каталогів файлової системи

Шлях від основи дерева файлової системи (кореня), який починається з символу «/», називається *повним* або *абсолютним:* `qwe@vb:~$ ls /home/qwe/G1`

Шлях, який починається від домашньої папки (вона позначається «~/» або шлях файлу відносно поточної папки, називається *відносним:* `qwe@vb:~/G2$ cat ss1.txt`

Для зазначення у відносному шляху поточного або батьківського каталогу використовуються символи «**.**» (крапка) і «**..**» (дві крапки) відповідно. Після авторизації користувача в системі його робота, як правило, починається з особистого каталогу користувача - *домашнього каталогу*. Для зазначення абсолютного шляху файлу, який знаходиться всередині домашнього каталогу користувача, можна використовувати спеціальний символ «**~**» (тильда). Каталог, в якому користувач знаходиться в даний момент часу називається *поточним* або *робочим* каталогом.

Імена файлів можуть мати практично будь-які символи (~! @ # $ &% * () [] {} ' "\:;> <, пробіл), крім роздільника директорія (/), точки довжиною до 255 символів. Рекомендується використовувати наступний *набір символів - латинські літери (великі і маленькі), цифри, знак підкреслення, дефіс (але не на початку), точка.* Не *варто також починати імена файлів з дефіса (-),* тому що багато програм, які працюють з файлами, приймають в командному рядку ключі (опції), які починаються з дефіса. Імена файлів є чутливими до регістру (*case sensitive*) - великі і маленькі букви в іменах розрізняються. Якщо ім'я файлу починається з точки, то цей файл вважається прихованим: деякі команди його «не бачать».

В залежності від структури і призначення файлу виділяють декілька типів файлів:

- звичайний файл (*regular file*);

- каталоги (*directories*);

- символьні посилання (*symbolic links*);

- жорсткі посилання (*hard links*)

- спеціальний файл пристрою (*special device file*),

- файли взаємодії між процесами - FIFO або іменований канал (*named pipe*);

- сокет (*socket*).

Звичайні файли - це іменовані набори даних з можливістю довільного доступу.

Каталоги - спеціальний тип файлів, який дозволяє групувати разом інші файли та каталоги. Вміст каталогу являє собою список файлів, які в в ньому знаходяться.

Операційна система Linux дозволяє створювати посилання на файли або каталоги, які дозволяють одним і тим же файлів мати декілька імен (один і той же файл розташовувати в декількох каталогах). На такий файл можна посилатися з будь-якого місця.

Посилання бувають двох типів: *жорсткі* та *символічні*. *Жорсткі посилання* є ім'ям файлу або каталогу. Поки існує хоча б одне жорстке посилання, існує і сам файл або каталог. При створенні файлу для нього обов'язково створюється одне жорстке посилання. *Символьне посилання* є файлом, який містить лише шлях, який вказує на інший файл або каталог. Головна відмінність від жорсткого посилання полягає в тому, що у разі видалення файлу, на який вказує символьне посилання, то посилання залишиться, але буде «недозволеним». І навпаки, якщо видалити символьне посилання, то файл, на який воно вказує залишиться недоторканим.

Жорсткі посилання реалізовані на більш низькому рівні файлової системи. Файл розміщено тільки в певному місці жорсткого диска, але на це місце можуть посилатися кілька посилань з файлової системи. Кожна з посилань - це окремий файл, але ведуть вони до однієї ділянки жорсткого диска. Файл можна переміщати між каталогами, і всі посилання залишаться робочими, оскільки для них неважливо ім'я.

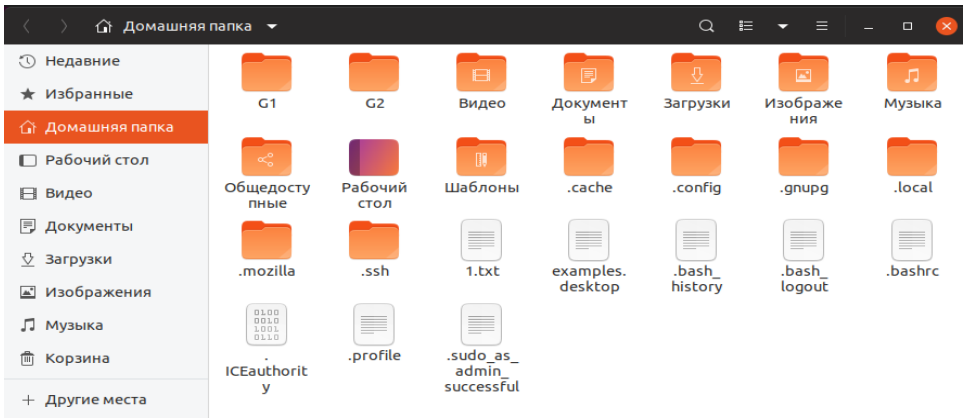
Файли пристроїв в Unix є засобом взаємодії прикладних програм з драйверами устаткування комп'ютера.

FIFO або іменований канал - це файл, який використовується для зв'язку між процесами.

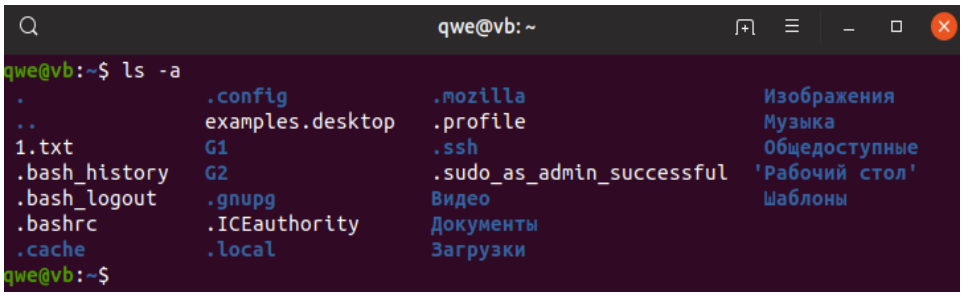
Сокети призначені для взаємодії між процесами. Інтерфейс сокетів часто використовується для взаємодії різних локальних і віддалених процесів в мережі TCP / IP.

Команди для роботи з каталогами та файлами

В Ubuntu за замовчуванням використовується файловий менеджер Nautilus. При вході в «Проводник», відкривши директорію «Домашняя папка» / «Home», натиснувши комбінацію клавіш «**Ctrl+H**», побачимо приховані файли.



Ввівши в своєму домашньому каталозі команду перегляду вмісту каталогу ***ls*** з ключем ***-a***, також будуть виведені ці папки і файли (ключ -a означає «показувати приховані файли»).



Файли ***.bashrc, .bash_profile і .profile*** - це файли налаштування нашої оболонки. Вони складаються з команд *bash* і виконуються перед тим як запустити оболонку або завантажити систему. Різниця цих файлів полягає в тому, коли вони виконуються: ***.profile і .bash_profile*** виконуються один раз при вході користувача в систему (логін користувача). Файл ***.bashrc*** виконується кожний раз, коли ви відкриваєте нове вікно терміналу (аналог автозавантаження в Windows).

В графічній оболонці програма «Термінал», яка запускається комбінацією клавіш «Ctrl+Alt+T» або ярлик із загального меню програм  , це важливий елемент операційної системи, який дозволяє запускати програми, створювати папки,

копіювати і видаляти файли, встановлювати додатки і т.д. Системна утиліта, в яку ви передаєте ці команди, називається *Shell* або командна оболонка. За замовчуванням в Ubuntu використовується командна оболонка, яка називається *Bash*.

При вході в командний інтерпретатор відкривається вікно, в якому відображається: *qwe* – ім'я облікового запису користувача, *vb* – ім'я комп'ютера, символ «:» -розділювач, «~» - каталог виконання команди (домашній каталог).

Синтаксис команд в Терміналі

Команди Терміналу, як правило, складаються з назви програми, ключа і значення. В загальному вигляді виглядають так: ***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.);

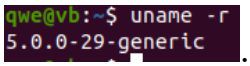
[ключ] - опції програми, які може приймати виконувана програма;

[значення] - даний параметр може приймати в якості аргументу цифри, текст, спеціальні символи і навіть змінні.

Наприклад, команда ***ls -l*** відображає вміст домашнього каталогу користувача 

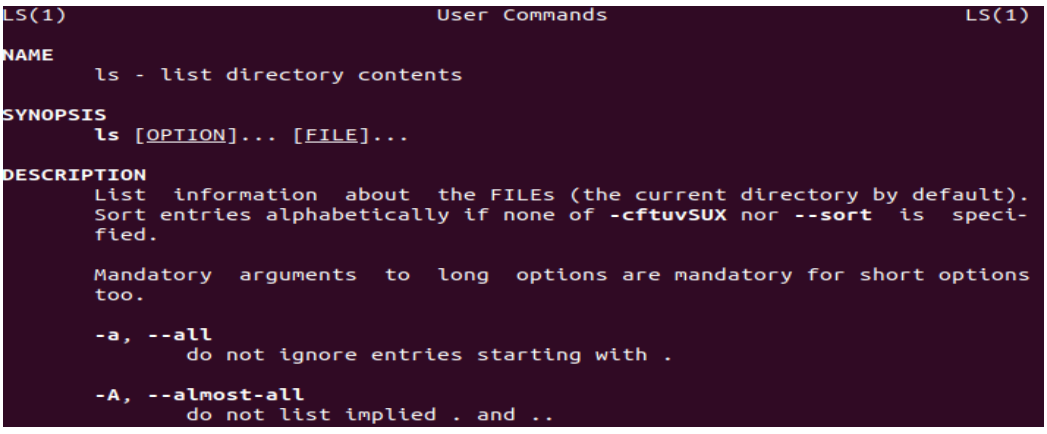
. Команда ***ls -la*** виводить усі каталоги і файли разом з прихованими.

Команда ***ls*** відображає вміст поточного каталогу.

Команда ***uname - r*** виводить версію ядра 

. Команда ***clear*** очищає екран. Команда ***pwd*** відображає каталог, в якому знаходиться користувач.

Для перегляду довідки про команду використовується команда ***man [ім'я_команди]***. Наприклад, ***man ls***.



З усіма ключами для команд, які описані нижче, можна ознайомитись з використанням команди *man*. Для перегляду введених команд використовується команда ***history***.

Команда *whoami* (від англ. *who am i* - хто я) призначена для отримання відомостей про ім'я користувача, під обліковим записом якого виконується дана команда. Вихід з Термінала «***Alt+F4***» або «***Ctrl+D***».

Для *створення файлу*, в ОС Ubuntu Linux існує команда ***touch [ключ] ... файл •.•*** Наприклад, команда ***touch myfile.txt*** у домашньому каталозі вашого облікового запису створить пустий файл myfile.txt.

```
qwe@vb:~$ touch myfile.txt
```

Часто *для створення файлів* використовується команда ***cat:***

cat>ім'я файла вводимо текста Ctrl+D (ознака кінця файлу)

```
qwe@vb:~$ cat>myfile.txt
The weather is wonderful
The day is clear and sunny
```

По команді *cat* виводиться ***вміст файлу*** або декількох файлів на стандартне виведення - на екран, якщо їхні імена послідовно задати як аргументи команди: ***cat /шлях/ім'я_файлу***

```
qwe@vb:~$ cat myfile.txt
The weather is wonderful
The day is clear and sunny
```

```
qwe@vb:~$ cat /home/qwe/myfile.txt
The weather is wonderful
The day is clear and sunny
```

По команді *cat* можна отримати копію якогось файлу, використовуючи перенаправлення у файл, тобто направляють дані зі стандартного введення - з клавіатури, а виведення команди - у новий файл: ***[user] $ cat file1> file2***

```
qwe@vb:~$ cat /home/qwe/myfile.txt>/home/qwe/youfile.txt
qwe@vb:~$ cat youfile.txt
The weather is wonderful
The day is clear and sunny
```

Виведення інформації про файл ***file шлях_до_файлу***

```
qwe@vb:~$ cd /home/qwe/G2
qwe@vb:~/G2$ file ZZ1.txt
ZZ1.txt: ASCII text
```

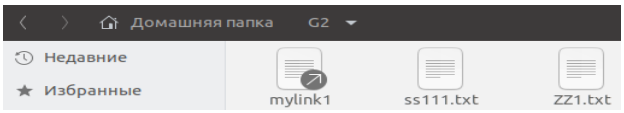
Перенаправлення виведення за допомогою символу ">>":

Дописати вміст файлу *newfile* у файл *myfile* ***cat newfile >>myfile***

Створення ***символьного посилання*** – команда ***ln*** з ключем «***-s»:***

ln -s цільовий_файл ім'я_символьного_посилання

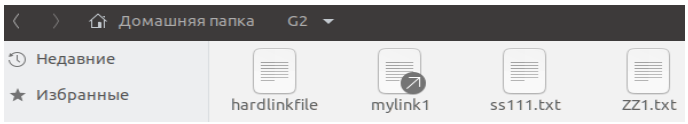
```
qwe@vb:~$ cd G2
qwe@vb:~/G2$ ln -s ZZ1.txt mylink1
```



Створення ***жорсткого посилання*** – команда ***ln*** без ключів

ln цільовий_файл ім'я_жорсткого_посилання

```
qwe@vb:~/G2$ ln ss111.txt hardlinkfile
```



Для *створення каталогу* в ОС Ubuntu Linux використовуєтьсяі команда mkdir (від англ. *make directory*- створити каталог). Синтаксис команди наступний:

mkdir [ключ] ... каталог. •.

Команда mkdir також дозволяє однією командою створювати відразу декілька каталогів в поточному каталозі, розділяючи їх пробілами: ***mkdir folderl folder2 folder3***

```
qwe@vb:~$ mkdir RR1 RR2 RR3
qwe@vb:~$ ls
 1.txt              G2    RR3          Загрузки      Общедоступные
 examples.desktop   RR1   Видео        Изображения   'Рабочий стол'
 G1                 RR2   Документы    Музыка        Шаблоны
qwe@vb:~$
```

Для *видалення пустих каталогів* призначена команда rmdir, синтаксис якої виглядає наступним чином: ***rmdir [ключ] ... каталог*** ... Ключ ***-p*** дозволяє видалити усі вкладені каталоги в 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. Для того щоб видалити каталог, потрібно команді *rm* передати ключ ***-r,*** який дозволить рекурсивно видалити каталог і його вміст.

Обов'язково слід розуміти, що запускаючи команду ***rm -rf /***, ви запускаєте процес видалення вмісту кореневої файлової системи, ***операційна система самознищується*** (особливо такі «жарти» пропонують на форумах).

Для переміщення між каталогами файлової системи застосовується команда *cd.*

cd [ключ] шлях_до_директорії

```
qwe@vb:~$ cd RR3/
```

Перейти в домашній каталог – «***cd***» без параметрів або «***cd ~***».

```
qwe@vb:~/RR3$ cd
```

Перейти на рівень вище «***cd ..***».

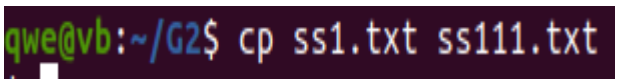
Перейти в директорію двома рівнями вище - «***cd ../..***»

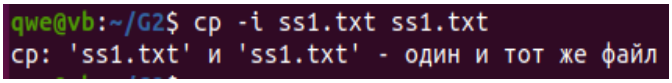
Перейти в кореневий каталог «***cd /***»

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 в заданий каталог. Синтаксис команди *cp*:

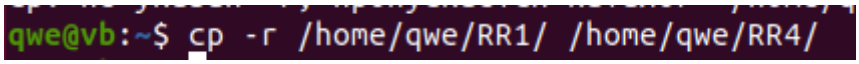
cp [ключ] ... джерело каталог_ (призначення)

При використанні команди *cp* рекомендується застосовувати опцію «***-i»*** для того, щоб отримати попередження, коли файл буде записуватись.

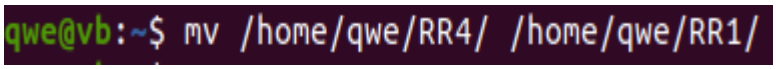




Ключ «**-r**» забезпечує рекурсивне копіювання каталогів:

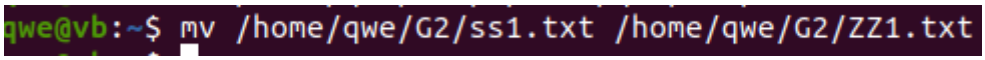


Для ***переміщення файлу з одного каталогу в інший***, ви можете скористатися командою ***mv.*** Синтаксис цієї команди аналогічний синтаксису команди *cp*. Команда працює наступним чином: спочатку копіює файл (чи каталог), а тільки потім видаляє вихідний файл (каталог).



Команда *mv* може використовуватися не тільки для переміщення, але і ***для перейменування файлів і каталогів*** (тобто переміщення їх всередині одного каталогу). Для цього треба просто задати як аргументи старе і нове ім'я файлу:

mv oldname newname

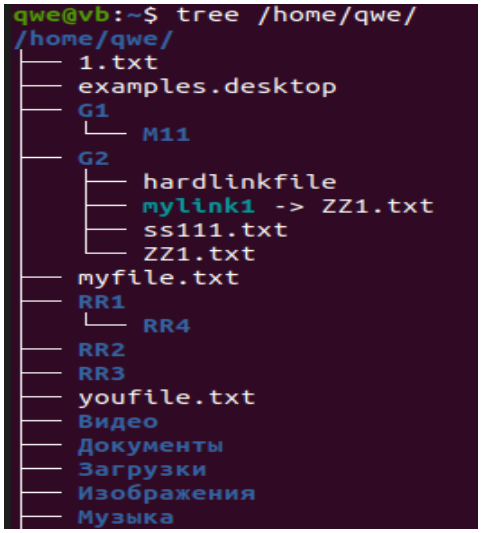


Команда *mv* не дозволяє перейменувати відразу декілька файлів (використовуючи шаблон імені), так що команда *mv *.xxx *.yyy* не працюватиме.

При використанні команди *mv*, також як і при використанні *cp*, рекомендовано застосовувати опцію ***-i***, щоб отримати попередження, коли файл буде записуватись.

Вивести дерево каталогів на термінал – команда ***tree***. Однак цю команду треба встановити, вибравши одну з команд: ***sudo apt install tree*** або ***sudo snap install tree.***

tree шлях_до_папки



Кожний сеанс роботи з ОС Unix повинен закінчуватися введенням команди *logout*. Також можна використовувати комбінацію клавіш Ctrl+D, яка дозволяє виконати команду завершення роботи з командною оболонкою, після чого система переходить в режим очікування реєстрації наступного користувача.

Завдання:

1. Ознайомитися з теоретичними матеріалом по лабораторній роботі. Набути навичок роботи в терміналі Linux.
2. Опанувати команди для роботи з файловою системою.
3. Навчитися створювати облікові записи користувачів.
4. Підготувати звіт для викладача про виконання лабораторної роботи і представити його

Хід виконання роботи

1. Авторизуйтесь в системі, запустіть Термінал.
2. Ознайомтесь з роботою команд.
3. Використовуючи команди оболонки, створіть у домашньому каталозі три файли, запишіть до них текст. Виведіть результати роботи. Перейменуйте один з файлів за вибором.
4. Створити у домашньому каталозі каталог, назва якого складається з першої літери вашого прізвища, імені, по батькові та 1 (умовно PIB1). Скопіюйте до нього усі три файли.
5. Створіть підкаталоги pib2, pib10, pib8. В підкаталозі pib2 створіть директорії pib3 та pib5, а підкаталозі pib10 – директорії pib4 та pib9. В директорії pib3

створіть директорії pib6, pib7. Директорія PIB1/pib8 містить символічне посилання на каталог PIB1/pib2/pib3/pib6.

6. Вивести дерево каталогу PIB1.
7. Перейменувати каталог pib9 у pib 99.
8. Видалити каталог pib5.
9. Скопіювати один з файлів, створених в каталозі PIB1, до директорії pib4.
10. Вивести дерево каталогу PIB1.
11. Створити три групи користувачів, 3 користувача, додати по одному користувачу в кожну групу. Переглянути результат.
12. Видалити одного користувача з будь-якої групи, показати результат.
13. За допомогою команди history виведіть команди, які ви використовували.

Підготувати звіт

1. Описати хід виконання поставлених завдань, надаючи знімок екрану (screenshot).
2. Висновки по роботі.

Контрольні питання

1. Що таке файлова система?
2. Що таке каталог?
3. Що таке шлях до файлу?
4. Абсолютний і відносний шлях?
5. Типи файлів які вам відомі?
6. Посилання. Типи посилань.
7. Команда створення посилання.
8. Команди для роботи з каталогами.
9. Команди для роботи з файлами.
10. Додавання/видалення користувачів.
11. Створення/видалення груп користувачів.
12. Додавання/видалення користувача у групу.

КОМП'ЮТЕРНИЙ ПРАКТИКУМ 5

Створення сценаріїв в оболонці Bash

Мета роботи - набути навичок створювання bash-скриптів в ОС Linux.

Теоретичні відомості

Bourne-again shell (GNU Bash) - це реалізація Unix shell, написана на C в 1987 році Брайаном Фоксом (Brian Fox) для GNU Project. Синтаксис мови Bash є надбудовою синтаксису мови Bourne shell. Переважна більшість скриптів для Bourne shell можуть бути виконані інтерпретатором Bash без змін, за винятком скриптів, які використовують спеціальні змінні або вбудовані команди Bourne shell. Також синтаксис мови Bash включає ідеї, запозичені з Korn shell (ksh) і C shell (csh): редагування командного рядка, історія команд, стек директорій, змінні $ RANDOM і $ PPID, синтаксис POSIX *для підстановки команд: $* (...).

Командна оболонка *BASH* дозволяє створювати скрипти за допомогою групування декількох команд, які виконують певну дію.

Скрипт - це звичайний текстовий файл, що містить системні або вбудовані команди оболонки. Такий файл може бути запущений на виконання наступним чином: ***$bash ім'я_файла.***

Оболонка послідовно інтерпретує і виконує команди, задані в сценарії. Ці ж команди можуть бути виконані простим послідовним викликом їх в командному рядку оболонки. Для файлів сценаріїв оболонки *bash* прийнято встановлювати розширення ***.sh***. Тобто, для виконання скрипту необхідно запустити командну оболонку, передавши їй як параметр ім'я відповідного файлу. Є другий варіант запуску скрипту - вказати його ім'я в командній оболонці (тобто зробити з нього якийсь вид програми). Для цього треба в параметрах доступу визначити файл як *виконуваний*, і в перших рядках цього файлу явно вказати оболонку, для якої призначений цей скрипт, в такий спосіб: **#!оболонка**

Будь-який сценарій для Bash починається з вказівки в першому рядку комбінації:

#!/bin/bash

Символи **#!** повідомляють системі про те, що наступний за ними аргумент – це програма, яка застосовується для виконання даного файлу. В даному випадку програма ***/bin/sh*** - командна оболонка, що застосовується за замовчуванням. Ця

послідовність вказує на програму, яка використовується для обробки сценарію в командну оболонку bash. У загальному випадку символ «**#**» в скрипті означає коментар, що означає ігнорування рядка. Але якщо він є першим символом файлу і за ним слідує символ «**!**» та шлях до файлу (наприклад, /bin/bash, /bin/perl, /bin/ sh і т.д), командна оболонка запускає відповідний файл і передає йому його ім'я в якості аргументу.

Створення сценарію

Створити файл, який містить команди, можна допомогою будь-якого текстового редактора. У даній роботі рекомендується використовувати вбудований в ***mc*** редактор. Для створення нового файлу в *mc* використовуйте комбінацію клавіш Shift + F4. Створіть файл з ім'ям *first* з таким вмістом:

```
#!/bin/sh
# first
# Цей файл переглядає всі файли в поточному каталозі для пошуку рядка
# POSIX, а потім виводить імена знайдених файлів в стандартне виведення.
for file in *
do
  if grep -q POSIX $file
  then
    echo $ file
  fi
done
exit 0
```

Виконання скрипту відбувається по рядкам.

Перетворення сценарію у виконуваний файл

Файл сценарію можна виконати двома способами. Перший - запустити оболонку з ім'ям файлу сценарію як параметром: **$ /bin/sh first.** Цей варіант працює.

Другий спосіб (більш доцільний) - запускати сценарій, ввівши його ім'я і тим самим присвоївши йому статус інших команд Linux. Зробити це можна за допомогою команди **chmod**, змінивши режим файлу (**file mode**) і зробивши його виконуваним для всіх користувачів:

$ chmod u+x <ім’я сценарія>

$ chmod u+x first

Додані режими: для власника (User), виконання (x - eXecutable).

Аналогічну функціональність реалізує наступна команда:

chmod 744 <ім’я_файла_сценарія>

Після цього можна виконувати файл за допомогою **команди $ first**

При цьому може з'явитися повідомлення про помилку, яке говорить про те, що команда не знайдена.

Запуск сценарію на виконання з поточного каталогу проводиться за допомогою наступної команди: **./ <ім’я_файла_сценарія>**

Виправити помилку можна запровадивши з клавіатури в **командному рядку ./first** в каталозі, що містить сценарій, щоб задати командній оболонці повний відносний шлях до файлу.

Зазначення шляху, який починається з символів «**./**», дає ще одну перевагу - ви випадково не зможете виконати іншу команду з тим же ім'ям, що і у вашого файлу сценарію. Після того як ви переконаєтеся в коректній роботі вашого сценарію, можете перемістити його в більш відповідне місце, ніж поточний каталог.

Синтаксис команд для створення сценарію

Змінні та підстановка їх значень. Всі змінні у мові – текстові. Їх імена повинні починатися з літери і складатися з латинських букв, цифр і знаку підкреслення (_). Щоб скористатися значенням змінної, треба перед нею поставити символ **$**. Використання значення змінної називається *підстановкою*.

Розрізняють два класи змінних: позиційні та з іменем.

Позиційні змінні - це аргументи командних файлів, їх іменами слугують цифри:

$0 - ім'я виконуваної команди (для скрипту - це шлях, вказаний при його виклику, для функції - ім'я оболонки).

$1 - перший аргумент, другий - $2 і т.д. - змінні, які відповідають аргументам, заданим при виклику сценарію (n - десяткове число, що відповідає номеру аргументу).

Значення змінній присвоюється наступним чином: **ім’я змінної=значення**.

Наприклад, X=1, або X=a, або X="f" і т.п. Але до і після знаку «=» немає пропуску!

Прийнято називати змінні буквами верхнього регістру, наприклад:

```
#!/bin/bash
TERM=vt100
CONTER=0
```

При виконанні команд використовується *підстановка змінних*. В команду «підставляється» будь що (змінна, виведення іншої команди і т.п.). Для підстановки використовується або символ «$», або вираз, розміщений у зворотні апострофи ***(`вираз`).***

Якщо в тексті команди зустрічається символ «$», то наступний за ним текст до пробілу або кінця команди інтерпретується як ім'я змінної, значення якої підставляється в текст команди.

Наприклад: `$FRUIT=Яблуко`

`$echo "Фрукт "$FRUIT`

Команда *echo* виведе на екран Фрукт Яблуко.

Команди `$DATE=`date``

`$echo $DATE`

Виведуть системну дату на екран.

Командна оболонка дозволяє виконувати арифметичні операції. Для цього вираз, яке необхідно інтерпретувати як арифметичний, записати у подвійні круглі дужки, і перед ними ставиться знак долара.

Наприклад:
```
$foo=$((((5+3*2)-4)/2))
echo $foo
```

Для отримання значення змінної використовуються наступний синтаксис:

```
ALFA=$BETA+$GAMMA
```

Виконувану команду необхідно записати у зворотні апострофи і присвоїти змінній: **`FILES=`/bin/ls -a/home/student``**

Користувач може надати значення змінній через командну оболонку за допомогою команди **read**, якої в якості аргументу передається ім'я необхідної змінної. Наприклад:

```
$read CHOICE
Привіт !!!
$echo "Ви ввели "${CHOICE}
Привіт !!!
$echo "Ви ввели "${CHOICE}
```

Умовний оператор if | then та else

При написанні скриптів часто виникає є необхідність у перевірці (розгалуженні) будь-яких процесів. Оператори **if | then** перевіряють код завершення переліку команд

на «успішне завершення (істина)», це означає «0». Якщо це так, то виконується одна або більше команд, які записані після оператора *then*. Якщо перевірка повертає «Не успішне завершення (невірно)», це означає «1», виконується оператор *else* «інакше» як того вимагає умова. На завершення умови обов'язково закриваємо його **«fi»!**

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 «[», є зарезервованим словом, а не командою, його bash виконує як один елемент з кодом повернення. Всередині «**[[....]]**» дозволяється виконання операторів **&&**, ||, які призводять до помилки в звичайних дужках «[....]» тим самим варіант з подвійною дужкою більш універсальний.

Круглі дужки «**((**»- є арифметичними виразами, яке так само повертають код 0. Тим самим такі вирази можна використовувати в операціях порівняння.

Список логічних операторів, які використовуються для ***if | then | else:***

«**-z**» - рядок порожній, «**-n**» - рядок не порожній,

«**=, (==)**» рядки рівні, «**!=**» - рядки нерівні,

«**-eq**» - дорівнює, «**-ne**» - не дорівнює, «**-lt, (<)**» - менше,

«**-le, (<=)**» - менше або дорівнює, «**-gt, (>)**» - більше,

«**-ge, (> =)**» - більше або дорівнює, «!» - запереченнНаприклад:я логічного виразу,

«-a, (&&)» - логічне «І», «**-o, (||)**» - логічне «АБО».

Ключі для роботи з файлами	
-d файл	True (істина), якщо файл - каталог
-e файл	True (істина), якщо файл існує.
-f файл	True (істина), якщо файл - звичайний
-r файл	True (істина), якщо файл доступний для читання
-s файл	True (істина), якщо файл ненульового розміру
-w файл	True (істина), якщо файл доступний для запису
-x файл	True (істина), якщо файл - виконуваний файл

Конструкції перевірки if\then

```
if умова; then
        команди
fi
```

Наприклад:
```
#!/bin/bash
if echo Тест; then
    echo 0
fi

#!/bin/bash
if [-f $HOME/.bashrc]; then
    echo "Файл існує!"
fi
```
де $HOME/.bashrc - шлях до файлу, echo - друкує повідомлення в консоль.

Конструкції перевірки if|then|else

if умова; then
команди 1
else
команди 2
fi

Наприклад:
```
if [-d /bin/bash]
then
   echo "/bin/bash  is a directory"
else
   echo "/bin/bash is NOT a directory"
fi
```

Умови для арифметичних виразів

Якщо оператор > використовувати всередині [[....]], він розглядається як оператор порівняння рядків, а не чисел. Тому для порівняння чисел треба використовувати оператори «<» або «>», які розташовані у круглі дужки.

Наприклад:
```
#!/bin/bash
if ((3 <6)); then
   echo Так
fi
```
Використання команди test, якою є квадратні дужки. Якщо «3» менше «6» друкуємо «Так».

```
#!/bin bash
if [3 -lt 6]; then
  echo Так
fi
```
Можна використовувати і подвійні квадратні дужки, це розширений варіант команди test ,еквівалентом якої є «[]». Якщо «3» менше «6» друкуємо «Так».

```
#!/bin/bash
if [[3 -lt 6]]; then
```

```
    echo Так
fi
```

Використовуємо подвійні квадратні дужки, тому що застосовуємо оператор «**&&**». Якщо перший вираз 2=2 (істина), тоді виконуємо друге, й якщо воно 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 перший вираз 2=2 (істина), тоді виконуємо друге, й якщо змінна «b» не дорівнює двом (нерпавда), друкуємо «Не вірно».

```
#!/bin/bash
a="2"
b="3"
if [[2="$a" && 2="$b"]]; then
    echo Вірно
else
    echo Не вірно
fi
```

Вкладання кількох перевірок

Bash дозволяє вкладати в 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 необхідно виконувати безліч перевірок краще використовувати ***elif*** - це коротка 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
```

Цикл ПОКИ

Якщо потрібно повторити виконання послідовності команд, але заздалегідь не відомо, скільки разів слід їх виконати, застосовується цикл ***while***:

```
while <умова> do
   <оператори>
done
```

До тих пір, поки код завершення останньої команди <списка1> є 0, виконуються команди <списка2>. При заміні службового слова while на until умова виходу з циклу змінюється на протилежне.

Приклад. Програма перевірки паролів

```
#!/bin/sh
echo "Enter password"
read trythis
while ["$trythis"!="Secret"]; do
   echo "Sorry, try again"
   read trythis
done
exit 0
```

Наступні рядки - приклад виведення даного сценарію:

```
Enter password
password
Sorry, try again
secret
$
```

Цикл until

Цикл виконується, поки умова не стане істинною (true):

```
until <умова> do
   <оператори>
done
```

Запис дуже схожа на синтаксис запису циклу while, але перевірка зі зворотною умовою.

Приклад. Обчислення суми цілих чисел, що вводяться з клавіатури. Ознака закінчення введення - число 0.

```
#!/bin/sh
sum=0
read num
until [$num -eq 0]; do
   sum = $((sum+num))
   read num
done
echo $ sum
exit 0
```

Як правило, якщо потрібно виконати цикл хоча б один раз, застосовують цикл while; якщо такої необхідності немає, використовують цикл until.

Оператор циклу for

Оператор циклу *for* призначений для обробки в циклі низки значень, які можуть являти собою будь-яка множина рядків. Рядки можуть бути перераховані в програмі або являти собою результат виконаної командною оболонкою підстановки імен файлів. Синтаксис оператора циклу:

```
for змінна in значення
do
   оператори
done
```

Приклад. Вивести на екран всі імена файлів сценаріїв в поточному каталозі, що починаються з літери "f", та імена всіх сценаріїв, які закінчуються символами *.sh*. Це можна зробити наступним чином:

```
#!/bin/sh
for file in $ (ls f*.sh); do
  echo $ file
done
echo $ file
exit 0
```

Функції

Синтаксис оголошення функції:

```
    function <ім'я> ()
  {
    <список>;
  }
```

Функція визначається іменем <ім'я>. Тіло функції - <список> записується між дужками { }.

Приклади обчислення факторіала.

Рекурсивне визначення факторіала:

```
function factorial
{
    typeset -i n=$1
    if [$n=0]; then
        echo 1
        return
    fi
    echo $((n*$(factorial $((n-1)))))
}
for i in {0..16}
do
   echo "$i!=$(factorial $i)"
done
```

Ітеративне визначення факторіала:

```
f=1
for ((n=1; $n<=17; $((n++)) );
do
  echo "$((n-1))!=$f"
  f=$((f*n))
done
```

Відладка сценаріїв

При виявленні помилки при виконанні сценарію командна оболонка виводить на екран номер рядка, що містить помилку. Якщо помилку відразу не видно, потрібно додати кілька додаткових команд *echo* для виведення значень змінних, протестувати фрагменти програмного коду, вводячи їх в командній оболонці в інтерактивному режимі. Основний спосіб відстеження помилок, які найбільш складно виявляються - використання опцій відладки командної оболонки.

Опції відладки командного рядка:

Опція	Призначення
sh -n <сценарій>	Тільки перевіряє синтаксичні помилки
sh -v <сценарій>	Виводить на екран команди перед їх виконанням

sh -x <сценарій>	Виводить на екран команди після обробки командного рядка
sh -u <сценарій>	Видає повідомлення про помилку при використанні евизначеної змінної

Завдання:

1. Ознайомитися з теоретичними матеріалом по лабораторній роботі.
2. Опанувати команди, які використовують при написанні сценарію.
3. Підготувати звіт для викладача про виконання лабораторної роботи і представити його.

Хід виконання роботи

Розробити скрипт, який виконує зазначені дії згідно варіанту по списку журналу групи. Створити файл з назвою власного прізвища, записати до нього ПІБ студента, групу навчання, за бажанням додаткову інформацію щодо ваших уподобань. Створити скрипт, який виводить ПІБ студента, групу навчання, коментарі щодо майбутніх дій, необхідно робити затримку після видачі результатів на екран. Скрипт повинен містити функції, які виконують зазначені дії, а також додатково записати дію відповідно до варіанту.

Варіанти:

1. Сценарій перейменування власного файлу.
2. Вивести коротку довідку щодо команди *read*.
3. Відобразити список процесів.
4. Вивести дерево вашого домашнього каталогу.
5. Знайти текстовий рядок у вашому файлі.
6. Вивести інформацію про систему.
7. Змінити поточний каталог.
8. Відобразити режим доступу до вашого файлу.
9. Змінити режим доступу до власного файлу групі користувачів для встановлення дозволу тільки на читання.
10. Виконати копіювання власного файлу зі збереженням атрибутів.
11. Створити каталог та виконати його копіювання.
12. Перемістити власний файл у новий каталог.
13. Створити каталог, вивести його назву на екран та видалити.

14.Вивести на екран вміст вашого файлу.

15.Скопіювати власний файл, видалити у ньому назву групи.

16.Вивести версію системи.

17.В каталозі віднайти виконуваний файл.

18.Вивести імена файлів, які починаються на буку «f».

19.Вивести на екран дату створення файлу (власного або будь-якого іншого).

20.Вивести довідку про команду *printf*.

21.Створити архів для вашого файлу або каталогу, де він зберігається.

22.Вивести список інстальованих програм менеджера пакетів ***dpkg.***

23.Вивести довідку про менеджер пакету ***aptitude.***

24. Відобразити стан поточної конфігурації мережі.

25.Вивести інформацію про поточного користувача.

26.Вивести інформацію про поточний каталог.

27.Створити новий файл, записати до нього вміст вашого файлу.

28.Створити новий каталог, скопіювати до нього ваш файл.

29.Ввести з командного рядка значення двох змінних, обчислити їх суму, записати у ваш файл.

30.Ввести з командного рядка вашу дату народження і записати її до вашого файлу.

Підготувати звіт

1. Описати хід виконання поставлених завдань, надаючи знімок екрану (screenshot).

2. Висновки по роботі.

Контрольні питання

1. Що таке скрипт?
2. Як перетворити скрипт у виконуваний файл?
3. Що означає символ «\», введений в командному рядку перед натисканням Enter?
4. Для чого використовується команда read?
5. Як працює умовний оператор if-fi?
6. Які конструкції застосовуються для організації циклу?

КОМП'ЮТЕРНИЙ ПРАКТИКУМ 6

Робота з процесами ОС Linux

Мета роботи:

- набуття навичок управління процесами в оболонці Bash;
- опанування команд *ps, top, pstree, bg, fg, nice, renice, kill, killall.*

Теоретичні відомості

Управління процесами в Linux

Процеси - це одна з найбільш фундаментальних абстракцій в системах UNIX після файлів. Від оптимального налаштування підсистеми управління процесами та числа одночасно виконуваних процесів залежить завантаження ресурсів процесора, що безпосередньо впливає на продуктивність системи в цілому. Задача ядра – управління процесами. Необхідно чітко розуміти відмінності між процесом і програмою.

Процес - це *середовище виконання завдання (оточення)*, яке містить виконуваний код, системні дані, дані користувача і, а також набір додаткових ресурсів, отриманих під час виконання (ресурси пам'яті, можливість доступу до пристроїв введення/виведення та різних системних ресурсів, включаючи послуги ядра. *Програма* - це файл, який містить виконуваний код, дані для ініціалізації та дані користувача.

Процес можна розглядати як сукупність даних ядра системи (*Kernel*), необхідних для опису образу програми в пам'яті і управління її виконанням, або як програму в стадії її виконання, тому що усі програми Unix представлені у вигляді процесів. 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. Процес складається з інструкцій, які виконуються процесором, даних та інформації про виконуване завдання, а саме: виділена пам'ять, відкриті файли і статус процесу.

Типи процесів

Системні процеси в Unix завжди розташовані в оперативній пам'яті. Їх виконувані інструкції та дані знаходяться 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, наприклад, *системний процес init,* який запускається ядром системи при завантаженні і є одним з ключових процесів для нормального функціонування системи. Приклади системних процесів системні - *vmdaemon, pagezero, bufdaemon, syncer*.

Демони – неінтерактивний процес, який працює у фоновому режимі і не прив'язаний ні до якого керуючого терміналу. Зазвичай демони запускаються при ініціалізації системи, однак після ініціалізації ядра забезпечують роботу різних підсистем Unix: системи термінального доступу, системи друку, системи мережевого доступу, мережевих послуг і т.п. Демони не пов'язані із жодним користувачем, тобто не мають ніякого відношення до користувацьких процесів. Як правило, демони знаходяться у стадії очікування, поки для певного процесу не виникне потреба виконати певну послугу (звернення до архіву файлу, друк документу). 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».

Усі інші *процеси*, які виконуються в системі, вважаються *прикладними або інтерактивними*. Практично це 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 рамках сеансу користувача. При виході з системи усі користувацькі процеси знищуються.

Процеси взаємодіють між собою засобами міжпроцесної взаємодії (*Interprocess Communication - IPC), а саме:*

- *канали* (*pipe*, конвейєри та іменовані канали *FIFO: First In First Out*),

- *сигнали* (це асинхронне повідомлення процесу про будь-яку подію. Коли сигнал посланий процесу, операційна система перериває його виконання. Якщо процес встановив власний обробник сигналу, операційна система запускає цей обробник,

передавши йому інформацію про сигнал. Якщо процес не встановив обробник, то виконується оброблювач за замовчуванням),

- *сокети*.

Сигнали посилаються наступними 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 наступним чином: один процес поміщає повідомлення в чергу за допомогою деяких системних викликів, а будь-який інший процес може прочитати його звідти, за умови, що і процес-джерело повідомлення і процес-приймач повідомлення використовують один і той же ключ для отримання доступу до черги).

Сокети представляють собою *віртуальний об'єкт*, який існує, поки на нього посилається хоча б один з процесів. *Сокети* UNIX бувають 2х типів: локальні і мережеві. *Локальному сокету* присвоюється UNIX-адреса і буде створений спеціальний файл (файл сокета) по заданому шляху, через який зможуть повідомлятися будь-які локальні процеси шляхом простого читання/запису з нього. При використанні *мережевого сокета* створюється абстрактний об'єкт, прив'язаний до порту операційної системи, який слухає, та мережевого інтерфейсу. Цьому типу сокета присвоюється INET-адреса, яка має адресу інтерфейсу і порту, який слухає.

Процеси можуть виконуватися на передньому плані (*foreground*) - режим за замовчуванням і 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 ви взаємодієте, він отримує інформацію з клавіатури (стандартне введення) і посилає результати на ваш екран (стандартне виведення).

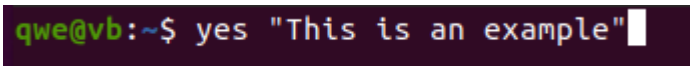
Фоновий процес після свого запуску завдяки використанню спеціальної команди командної оболонки відключається від клавіатури і екрану, тобто не очікує введення даних зі стандартного введення і не виводить інформацію на стандартне виведення, а

командна оболонка не очікує закінчення запущеного процесу, що дозволяє користувачеві негайно запустіть ще один процес.

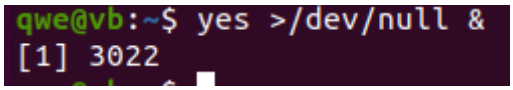
Зазвичай фонові процеси вимагають дуже великого часу для свого завершення і не потребують втручання користувача під час існування процесу. Наприклад, компіляцію програм або архівування великого обсягу інформації можна перевести у фоновий режим.

Для запуску програми в якості 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 для запуску програми ***yes*** у фоновому режимі з подавленням виведення має вигляд:



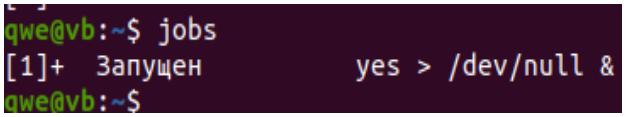
Команда *yes* використовується для відображення рядка декілька разів, поки вона не буде завершена за допомогою [**Ctrl + C**]. ***/home/user $ yes> /dev/null &***



Після команди виводиться повідомлення, що складається з двох чисел. Перше число в дужках означає номер запущеного фонового процесу для користувача в поточному сеансі, з його допомогою можна проводити маніпуляції з цим фоновим процесом. Друге число показує ідентифікаційний номер (PID) процесу. Відмінності цих двох чисел достатньо суттєві. Номер фонового процесу унікальний тільки для користувача, що запускає цей фоновий процес. Тобто, якщо в системі три користувача вирішили запустити фоновий процес (перший для поточного сеансу), то в результаті у кожного користувача з'явиться фоновий процес з номером 1. Навпаки, ідентифікаційний номер процесу (PID) є унікальним для всієї операційної системи і 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 в даний момент запущений один фоновий процес, і він виконується. Подавлення виведення команди здійснюється перенаправленням вихідного потоку на псевдопристрій */dev/null*. Все, що записується в цей файл, «зникає» назавжди.

Дві команди дуже корисні для перегляду п*рацюючих в системі процесів*, це ***ps*** (*process status,* показує моментальний *знімок процесів в системі*) і ***top*** (*table of processes*, дозволяє *переглядати інформацію про процеси в реальному часі).* Команда *ps використовується для отримання списку запущених процесів і може показати їх PID,* скільки пам'яті вони використовують, команду, якій вони були запущені і т.д. Команда *top показує запущені процеси і оновлює екран кожні* кілька секунд, що дозволяє спостерігати за роботою комп'ютера в реальному часі. Детальну інформацію про ці програмам можна отримати на відповідних сторінках довідки (*man ps і man top).*

З точки зору ядра процес являє собою запис в ***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 відкритих процесом файлів, які зберігається в його адресному просторі. *Запис в таблиці процесів і простір процесу разом складають контекст, або оточення, процесу*.

Контекст кожного процесу містить:

- ***pid*** (*process ID* - ідентифікатор процесу), примусово призначається планувальником при запуску процесу;

- ***ppid*** (*parent process identifier))* - ідентифікатор батьківського процесу, UID і GID - ідентифікатори прав процесу (UID - *user identifier, ідентифікатор користувача* і GID – *group identifier, ідентифікатор групи процесів*);

- ***tty*** - ім'я керуючого терміналу (термінал, з якого запущений процес);

- ***nice*** - пріоритет процесу або показник ввічливості;

- статус/стан процесу ***stat*** (*R* = R*unning,* виконується; *S=Sleeping* – у стану очікування; *D=Direct* процес очікує певного сигналу виключно від апаратної частини; *T=Tracing* – процес знаходиться у режимі тросування/відладка програми; *Z=Zombie*,

у стані зомбі, тобто процес закінчився, але з деяких причин не звільнений з ядра; «<» підвищений пріоритет; «+» знаходиться в інтерактивному режимі);

- таблиця відкритих (використовуваних) файлів процеса;

- - змінні оточення.

Перший процес, який ядро виконує під час запуску системи, називається *процесом ініціалізації* (*systemd).* Його *pid* завжди 1, *ppid* – 0. Зазвичай *systemd process* в Linux є програмою ініціалізації.

Ядро Linux перебирає чотири виконуваних модуля в наступному порядку:

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.

Після виконання ініціалізації системи запускаються різні сервіси та програми авторизації. Оскільки кожний процес належить певному користувачеві і групі, то ядро ОС Linux кожному процесу призначає числовий ідентифікатор (особистий номер) в діапазоні від 1 до 65535, тобто подвійне слово 2^{16}, *pid* та ідентифікатор батьківського процесу *ppid*. Усі процеси ОС Linux можна представити у вигляді дерева, в якому кореневим буде процес ініціалізації системи. Процеси, імена яких відображаються в квадратних дужках, наприклад, [keventd] - це процеси ядра. Такі процеси управляють складовими системи (менеджером пам'яті, планувальником часу процесора, менеджерами зовнішніх пристроїв тощо). Інші процеси це процеси користувача, які запущені або з командної оболонки (терміналу), або з графічної оболонки, або під час ініціалізації системи. Для отримання числового UID поточного користувача використовується наступна команда:

$ ***id -u*** . Можна також дізнатися UID будь-якого користувача системи: ***$ id -u USERNAMEUID.***

Суперкористувача (з ім'ям root в переважній більшості випадків) завжди дорівнює 0: ***$ id -u root.***

Для отримання *імені користувача* з числового UID застосовується бібліотечна функція *getpwuid(),* яка оголошена в заголовочному файлі ***pwd.h*** наступним чином:

struct passwd *getpwuid (uid_t UID);

В структурі *passwd* нас цікавить тільки одне поле, яке містить ім'я користувача:

```
struct passwd
{
    /* ... */
    char *pw_name;
 };
```

Якщо UID не відповідає жодному користувачеві системи, то *getpwuid ()* повертаєє NULL.

Приклад виконання процеса у фоновому режимі (знак «&» вказує про відсутність зв'язку з терміналом, це означає, що непотрібно виконувати операції введення-виведення):

```
qwe@vb:~$ sleep 2000 &
[1] 2310
```

Утиліта *sleep* виконує затримку на зазначений час, по закінченні цього часу завершується.

При створенні фонового процесу *d* оболонці *bash* еквадратних дужках вказується номер завдання в системі ([1]) та його його *pid* (2310).

Для виведення процесу із фонового режиму використовується команда ***fg*** і процесу надається функція керування (володіє потоками введення-виведення терміналу):

```
qwe@vb:~$ fg
bash: fg: выполнение задания прервано
[1]+  Завершён        sleep 2000
```

.

Моніторинг процесів

Життєвий цикл процесу. Для управління процесами в Linux використовується дві операції:

- створення нового процесу - виклик fork (),

- завершення поточного процесу, виклик exit() виконує основні кроки перед завершенням, а потім відправляє ядру команду припинити процес.

Системний виклик ***fork()*** - це операція, при якій процес копіює себе, і тим самим створює новий процес з унікальним ID, тобто запускає той же системний образ, що і поточний:

#include <sys/types.h>

#include <unistd.h>

pid_t fork(void);

У разі успішного звернення до *fork()* створюється новий процес, в усіх відношеннях ідентичний викликаючому. Обидва процеси виконуються від точки звернення до *fork(),* як ніби нічого не відбувалося.

Новий процес є дочірнім по відношенню до викликаючого, який, в свою чергу, називається батьківським. У дочірньому процесі успішний запуск *fork()* повертає *0*. В батьківському процесі *fork()* повертає *pid* дочірнього. Батьківський і дочірній процеси практично ідентичні, за винятком деяких особливостей:

- *pid* дочірнього процесу, призначається заново і відрізняється 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.

Стандартний виклик завершення поточного процесу ***exit()*** наступний:

#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 ніяких результатів.

Параметр *status* використовується для позначення статусу процесу завершення. Інші програми - як і користувач оболонки - можуть перевіряти цю величину. Зокрема, статус *status & 0377* повертається батьківському процесу.

Є інша можливість отримати інформацію про процес, а саме про його дані - ідентифікатор процесу (PID); ідентифікатор батьківського процесу (PPID); ідентифікатор користувача (UID).

Отримати PID, PPID і UID поточного процесу можна за допомогою наступних системних викликів, оголошених в заголовки unistd.h: ***pid_t getpid (void); pid_t getppid (void); uid_t getuid (void);*** Типи даних *pid_t* та *uid_t* є цілими числами, розмірність яких залежить від реалізації. Щоб використовувати ці типи, потрібно включити в програму заголовочний файл ***sys/types.h.*** Системний виклик ***getpid()*** повертає ідентифікатор поточного процесу, *getppid()* - батьківського, а *getuid()* - ідентифікатор користтувача, від імені якого виконується процес.

Для виведення списку усіх процесів в в поточній оболонці використовується **команда ps (Get-Process):** ***ps [PID] options.***

Команда *ps* має три типи стилів представлення опцій:

– стиль Unix98: ps [-опції] (використовується дефіс);

– в стилі BSD: ps [опції] (без дефіса);

– в стилі GNU-версії: ps [-- довге ім’я опції [--довге ім’я опції] ...] (використовується два дефіса).

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), *TTY* (ідентифікатор терміналу), *TIME* (час ЦП), CMD (ім’я команди).

Виведення усіх процесів в різних форматах

Кожний активний процес в системі Linux в загальному форматі (Unix / Linux) відображається з опціями «-A» або «-e»: ***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
    6 ?        00:00:00 kworker/0:0H-kblockd
    8 ?        00:00:00 mm_percpu_wq
    9 ?        00:00:00 ksoftirqd/0
   10 ?        00:00:03 rcu_sched
   11 ?        00:00:00 migration/0
   12 ?        00:00:00 idle_inject/0
   13 ?        00:00:05 kworker/0:1-events
   14 ?        00:00:00 cpuhp/0
   15 ?        00:00:00 cpuhp/1
```

Довідка про опції команди виводиться як ***man ps***.

Відображення усіх процесів у форматі BCD: ***ps au*** або ***ps axu***

```
qwe@vb:~$ ps au
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
qwe       1047  0.0  0.3 174068  6332 tty2     Ssl+ 17:40   0:00 /usr/lib/gdm3/g
qwe       1049  0.0  2.6 373164 54680 tty2     Sl+  17:40   0:10 /usr/lib/xorg/X
qwe       1081  0.0  0.7 584408 16248 tty2     Sl+  17:40   0:00 /usr/lib/gnome-
qwe       1373  0.8 15.1 2837208 308508 tty2   Sl+  17:41   1:32 /usr/bin/gnome-
qwe       1396  0.0  0.3 319104  8008 tty2     Sl   17:41   0:00 ibus-daemon --x
qwe       1401  0.0  0.3 244736  6860 tty2     Sl   17:41   0:00 /usr/lib/ibus/i
qwe       1405  0.0  1.3 296584 26720 tty2     Sl   17:41   0:00 /usr/lib/ibus/i
qwe       1409  0.0  1.1 219644 24032 tty2     Sl   17:41   0:00 /usr/lib/ibus/i
qwe       1521  0.0  0.4 323228  9848 tty2     Sl+  17:41   0:00 /usr/lib/gnome-
```

Відображення запущених користувацьких процесів: ***ps -x***

```
qwe@vb:~$ ps -x
  PID TTY      STAT   TIME COMMAND
  950 ?        Ss     0:00 /lib/systemd/systemd --user
  951 ?        S      0:00 (sd-pam)
 1042 ?        SLl    0:00 /usr/bin/gnome-keyring-daemon --daemonize --login
 1047 tty2     Ssl+   0:00 /usr/lib/gdm3/gdm-x-session --run-script env GNOME_SH
 1049 tty2     Sl+    0:11 /usr/lib/xorg/Xorg vt2 -displayfd 3 -auth /run/user/1
 1077 ?        Ss     0:00 /usr/bin/dbus-daemon --session --address=systemd: --n
 1081 tty2     Sl+    0:00 /usr/lib/gnome-session/gnome-session-binary --session
 1340 ?        Ss     0:00 /usr/bin/ssh-agent /usr/bin/im-launch env GNOME_SHELL
 1344 ?        Ssl    0:00 /usr/lib/at-spi2-core/at-spi-bus-launcher
 1349 ?        S      0:00 /usr/bin/dbus-daemon --config-file=/usr/share/default
 1351 ?        Sl     0:00 /usr/lib/at-spi2-core/at-spi2-registryd --use-gnome-s
 1373 tty2     Sl+    1:39 /usr/bin/gnome-shell
```

Команда ***ps -l*** у форматі BCD виводить розширений лістинг.

```
qwe@vb:~$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  2777  2764  0  80   0 -  4775 do_wai pts/0    00:00:00 bash
0 R  1000  2833  2777  0  80   0 -  5001 -      pts/0    00:00:00 ps
```

Для виведення усіх процесів, які виконуються з правами користувача root (реальний і ефективний ідентифікатор, Real і Effecitve ID), використовується команда ***ps -U root -u root***

```
qwe@vb:~$ ps -U root -u root
  PID TTY          TIME CMD
    1 ?        00:00:10 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
    9 ?        00:00:00 ksoftirqd/0
   10 ?        00:00:05 rcu_sched
   11 ?        00:00:00 migration/0
   12 ?        00:00:00 idle_inject/0
   13 ?        00:00:07 kworker/0:1-events
   14 ?        00:00:00 cpuhp/0
   15 ?        00:00:00 cpuhp/1
```

Відобразити процес за його *pid* : ***ps -fp 1351***

```
qwe@vb:~$ ps -fp 1351
UID          PID  PPID  C STIME TTY          TIME CMD
qwe         1351   950  0 17:41 ?        00:00:00 /usr/lib/at-spi2-core/at-spi2-r
```

Відобразити процес за його *ppid*: ***ps -f --ppid 950***

```
qwe@vb:~$ ps -f --ppid 950
UID          PID  PPID  C STIME TTY          TIME CMD
qwe          951   950  0 17:40 ?        00:00:00 (sd-pam)
qwe         1077   950  0 17:40 ?        00:00:00 /usr/bin/dbus-daemon --session
qwe         1344   950  0 17:41 ?        00:00:00 /usr/lib/at-spi2-core/at-spi-bu
qwe         1351   950  0 17:41 ?        00:00:00 /usr/lib/at-spi2-core/at-spi2-r
qwe         1382   950  0 17:41 ?        00:00:00 /usr/lib/gvfs/gvfsd
qwe         1387   950  0 17:41 ?        00:00:00 /usr/lib/gvfs/gvfsd-fuse /run/u
qwe         1397   950  0 17:41 ?        00:00:00 /usr/libexec/xdg-permission-sto
qwe         1412   950  0 17:41 ?        00:00:00 /usr/lib/ibus/ibus-portal
qwe         1423   950  0 17:41 ?        00:00:00 /usr/lib/gnome-shell/gnome-shel
qwe         1429   950  0 17:41 ?        00:00:00 /usr/libexec/evolution-source-r
```

Вивести інформацію про усі процеси ***ps -e***

```
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
    9 ?        00:00:00 ksoftirqd/0
   10 ?        00:00:06 rcu_sched
   11 ?        00:00:00 migration/0
   12 ?        00:00:00 idle_inject/0
   13 ?        00:00:08 kworker/0:1-cgroup_destroy
   14 ?        00:00:00 cpuhp/0
   15 ?        00:00:00 cpuhp/1
   16 ?        00:00:00 idle_inject/1
```

Вивести дерево 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 монітору і дозволяє виявити причини нестабільної роботи операційної системи та виявити процеси, які споживають більшість системних ресурсів.

```
qwe@vb:~$ top

top - 22:03:01 up  4:23,  1 user,  load average: 0,33, 0,14, 0,09
Tasks: 198 total,   1 running, 197 sleeping,   0 stopped,   0 zombie
%Cpu(s):  3,1 us,  2,2 sy,  0,0 ni, 94,7 id,  0,0 wa,  0,0 hi,  0,0 si,  0,0 st
МиБ Mem :   1990,8 total,    275,7 free,    842,2 used,    872,8 buff/cache
МиБ Swap:    947,2 total,    930,0 free,     17,3 used.    976,6 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 1373 qwe       20   0 2837736 302844 101860 S   6,0  14,9   3:29.73 gnome-sh+
  415 root      20   0   29448  13392   9188 S   1,7   0,7   4:59.30 plymouthd
 1049 qwe       20   0  373520  46340  34664 S   1,3   2,3   0:34.12 Xorg
 1070 root      20   0  381564  42272  23780 S   0,7   2,1   0:47.73 containe+
  350 root      20   0   19252   5176   2924 S   0,3   0,3   0:05.91 systemd-+
  604 root      20   0  246352   7140   6340 S   0,3   0,4   0:00.60 accounts+
  751 root      20   0  504740  72864  40028 S   0,3   3,6   1:03.65 dockerd
 3158 root      20   0       0      0      0 I   0,3   0,0   0:01.20 kworker/+
 3375 root      20   0       0      0      0 I   0,3   0,0   0:02.65 kworker/+
 3865 qwe       20   0   20456   3708   3272 R   0,3   0,2   0:00.10 top
    1 root      20   0  165284  10392   7740 S   0,0   0,5   0:13.75 systemd
    2 root      20   0       0      0      0 S   0,0   0,0   0:00.00 kthreadd
```

Перший рядок (*top*) – загальна інформація:

- 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) – статистика процесів:

- загальна кількість процесів в системі (198 *total*),
- кількість працюючих в даний момент процесів (1 *running*),
- кількість очікуючих подій процесів (197 *sleeping*),
- кількість зупинених процесів (0 *stopped*),
- кількість процесів, які очікують батьківський процес для передачі 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 з 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 віртуальній 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 повинно 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 і розділу підкачки відповідно. 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, яка 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 даним процесом,

%MEM - відсоток використання оперативної пам'яті даним процесом,

TIME+ - загальний час активності процесу,

COMMAND - ім'я процесу.

Пріоритети процесів

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 називаються в Linux *значеннями ввічливості (NI),* оскільки їх основна ідея - «бути ввічливим» по відношенню до інших процесів шляхом проходження процесної пріоритетності, дозволяючи іншим процесам споживати більше системного процесорного часу. Ядро Linux виділяє завданням з

більш високим пріоритетом більший квант часу, а завданням з меншим пріоритетам - менший квант часу, який в середньому становить 200 і 10 мс відповідно.

Діапазон значень ввічливості - 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 вище його пріоритет і більше квант часу.

Linux надає декілька викликів для отримання і установки значення ввічливості процесу. Найпростіший з них - ***nice():***

#include <unistd.h>

int nice(int inc);

Успішна робота *nice()* збільшує значення ввічливості процесу на значення ***inc*** і повертає оновлену величину. Тільки процес з характеристикою CAP_SYS_NICE (яким фактично володіє користувач *root*) може встановити негативну величину *inc*, зменшуючи його значення ввічливості і, таким чином, підвищуючи пріоритет процесу. Процеси без прав *root* можуть тільки знижувати свої пріоритети (збільшуючи значення ввічливості).

Кращим рішенням є застосування системних викликів ***getpriority() і setpriority(),*** які надають більше можливостей для управління 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);

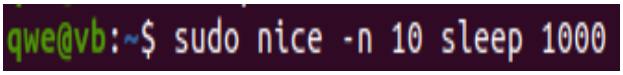
Ці виклики впливають на процес, групу процесів або користувачів, що визначено за допомогою *which і who*. Значення *which* повинно бути PRIO_PROCESS, PRIO_PGRP або PRIO_USER, в той час як *who* вказує ідентифікатор процесу, групи процесів або ідентифікатор користувача відповідно. Якщо значення *who* дорівнює 0, то виклик працює з поточними ідентифікатором процесу, групи процесів або ідентифікатором користувача відповідно.

Виклик *getpriority()* повертає найбільший пріоритет (найменшу чисельну величину значення люб'язності) кожного із зазначених процесів. Виклик *setpriority()* встановлює значення пріоритету кожного з зазначених процесів, яке дорівнює *prio*.

Як і *nice()*, тільки процес з властивістю CAP_SYS_NICE може збільшити пріоритет процесу (знизити чисельне значення люб'язності). Отже, тільки процес з 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 в команді, то використовується число 10.

sudo nice -10 sleep 1000



Для зміни пріоритету процесу використовується команда

renice -n <число> <список ідентифікаторів процесів>

Команда *renice* працює аналогічно *nice*, з тією різницею, що змінюється не пріоритет створюваного процесу, а пріоритети усіх запущених процесів, ідентифікатори яких входять у список, розділений пробілами та переданий команді в якості параметра. Число у складі ключа «-n» додається до поточного пріоритету процесу, що пришвидчує виконання процесу. Така зміна виконується виключно від імені адміністратора (root): *sudo renice -n -10 PID.*

Управління сигналами

Одним із способів взаємодії між процесами в Linux є сигнали; сигнали - це програмні переривання, які можуть бути послані процесу за допомогою системного виклику. Кожному виду сигналів в системі відповідає назва і номер. Багато сигнали мають спеціальний сенс. Зокрема, сигнал SIGTERM повідомляє процес про необхідність завершення, сигнал SIGTERM використовується для безумовного завершення процесу, сигнали SIGSTOP і SIGCONT, відповідно, зупиняють і відновлюють виконання процесу. Сигнали SIGKILL (примусове завершення процесу) і SIGSTOP неможливо ні перехопити, ні ігнорувати, тому що вони адресовані не процесу, а планувальнику ОС.

Для того щоб послати сигнал процесу з програми-оболонки:

kill [-s <назва сигналу>] <список ідентифікаторів процесів>

Для виведення усіх сигналів, які підтримує ця система, використовується 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 використовуємо команду ***kill -s SIGCONT 123.***

Для завершення процеса з ідентифікатором 123 використовуємо 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 всім процесам з іменем ***процес***. Якщо сигнал не вказаний, надсилається SIGTERM.

Команди оболонки для роботи у фоновому режимі: jobs, bg *(background) та* ***fg*** *(foreground)* Команда ***jobs*** виводить список процесів, які виконуються у фоновому режимі. Зі знаком «+» відображається процес, з якими працювали останнім, «-» - процес, з яким працювали передостаннім.

fg <номер_завдання> - переводить процес на передній план,
bg <номер_завдання> - переводить процес на задній план.
Номер завдання - це не PID, а число, яке команда *jobs* виводить в квадратних дужках. З ключем ***-l*** вона буде виводити, крім того, і PID процесу.

Оскільки перелічені команди - не самостійна утиліти, а підкоманди bash, довідку по ним потрібно запитувати так: *help <підкоманда>*.

Завдання:

1. Ознайомитися з теоретичними матеріалом по лабораторній роботі. Набути навичок по роботі з процесами.
2. Опанувати команди для управління процесами.
3. Підготувати звіт для викладача про виконання лабораторної роботи і представити його.

Хід виконання роботи

1. Ознайомтесь з роботою команд по управлінню процесами.

2. Вивести на екран лістинг характеристик (у довгому і короткому форматах) процесів, ініціалізованих з вашого терміналу. Записати їх у файл. Проаналізувати і пояснити вміст кожного поля повідомлення.

3. Вивести всю ієрархію процесів поточної оболонки разом полями *pid* та *ppid.*

4. Побудувати дерево процесів, які визначені у попередньому пункті. Результат виконання вивести на екран і дописати в файл.

5. Переглянути список процесів вашого користувача.

6. Вивести список процесів вашого користувача у вигляді дерева (команда pstree).

7. Переглянути список сигналів вашого користувача. Записати у окремий файл

8. За допомогою команди history виведіть команди, які ви використовували.

Підготувати звіт

1. Описати хід виконання поставлених завдань, надаючи знімок екрану (screenshot).
2. Висновки по роботі.

Контрольні питання

1. Що таке процес?
2. Які відмінності процесу та програми?
3. Які вам відомі типи процесів?
4. Що таке контекст процесу?
5. Що таке *pid*, *ppid*?
6. Як управляти пріоритетами процесу?
7. Які вам відомі команди для роботи з процесами та їх призначення?
8. Що таке сигнали?
9. Як управляти роботою сигналів?
10. Які вас відомі команди для роботи у фоновому режимі?

КОМП'ЮТЕРНИЙ ПРАКТИКУМ 7

Потоки, перенаправлення потоків

Мета роботи:

- ознайомлення з системними викликами для управління потоками в ОС Linux,
- перенаправлення потоків при роботі файлами і командами.

Теоретичні відомості

Управління потоками

Процес розбивається на виконувані одиниці - потоки (*threads*) (один і більше потоків).

Потік – набір послідовність виконуваних команд процесора, які використовують загальний адресний простір процесу, це елемент виконання всередині процесу: віртуальний процесор, стек або статус програми. У порівнянні з процесами взаємодія і синхронізація потоків вимагає менше часу, оскільки потоки одного процесу виконуються в одному адресному просторі.

Процес містить один або кілька потоків. Якщо процес містить тільки один потік, такий процес називається однопоточними. Це класичні процеси UNIX. Якщо процес містить більше одного потоку, такі процеси називаються багатопоточними.

Існує дві основні категорії реалізації потоків: *користувацькі потоки* - потоки, що реалізуються через спеціальні бібліотеки потоків і працюють в просторі користувача. *Потоки ядра* - потоки, що реалізуються через системні виклики і працюють в просторі ядра.

В ОС UNIX/Linux для потоків реалізований стандарт Р-потоків - POSIX (*Portable Operating System Interface*) - **pthreads** ("P" - от POSIX). Для написання багатопотокової програми API для роботи з Р-потоками надає біля 100 інтерфейсів. Кожна функція в API забезпечена префіксом **pthread_.**

Прототипи функцій роботи з потоками і необхідні типи даних містяться в заголовки <**pthread.h>.** Ці функції не включені в стандартну бібліотеку мови С, вони знаходяться в бібліотеці ***libthread***. Тому в командному рядку для ***gcc*** необхідно додати опцію ***-pthread:***

gcc -Wall -Werror -pthread beard.c -o beard

Кожний потік має свій ідентифікатор потоку, ID потоку. В програмах на C/C++ для ID потоків слід використовувати тип ***pthread_t*** з ***<sys/types.h>.***

При роботі з потоками використовуються основні функції:

- створення потоку;
- блокування роботи потоку в очікування завершення іншого;
- дострокове завершення потоків;
- завершення роботи потоків.

Створення потоку. Потоки створюються функцією ***pthread_create***, яка має наступну сигнатуру

```
#include <pthread.h>.
int pthread_create (pthread_t  *thread,
           const pthread_attr_t  *attr,
           void *(*start_routine) (void *),
           void  *arg);
```

Ця функція визначена в заголовки <*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*** , використовується для установки атрибутів потоку. Цей об'єкт управляє деталями взаємодії потоку з іншою програмою. Якщо параметр дорівнює NULL, то потік буде створений з атрибутами за замовчуванням.

Третім параметром функції *pthread_create* повинна бути адреса функції потоку – вказівник на функцію потоку. Ця функція відіграє для потоку ту ж роль, що функція *main* для головної програми. Функція потоку приймає один параметр типу покажчик на *void* і повертає значення типу вказівник на *void*.

Четвертий параметр функції *pthread_create* має тип void*. Цей параметр може використовуватися для передачі значення як аргумент у функцію потоку. Через нього можна передавати новому потоку параметри.

Після виклику *pthread_create* функція потоку буде запущена на виконання паралельно з іншими потоками програми.

Функція повертає 0 в разі успіху, код помилки – в разі невдачі.

Ідентифікатори потоків (TID *-Thread ID***)** для потоків є аналогами ідентифікаторів процесів (PID). У той час як *PID призначаються ядром Linux, TID*

призначаються лише бібліотекою P-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();***

Блокування роботи потоку. Приєднання дозволяє одному з потоків блокуватися в очікуванні завершення іншого:

```
#include <pthread.h>
int pthread_join(pthread_t  thread,  void  **retval);
```

Після успішного виконання викликаючий потік блокується до тих пір, поки потік, вказаний як *thread*, не завершиться (якщо потік *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 P-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.

В разі успіху функція *pthread_join()* повертає 0, в разі помилки - код помилки.

Дострокове завершення потоків. P-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*.

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);
```

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.

Функція повертає 0 в разі успіху, в разі невдачі – код помилки.

Завершення роботи потоків. Завершення роботи потоків дуже схоже на завершення роботи процесів, за винятком того, що, коли потік завершується, інші потоки в процесі продовжують виконуватися. Потоки можуть перериватися за наступних обставин:

- 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().*

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 в 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.

Потік завершується при звичайному поверненні з функції потоку, коли величина, яка повертається ***return***, буде значенням, яке повертається потоком.

Приклад роботи з потоком:

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
int i = 0;
void* thread_func(void *arg) {
         pthread_setcancelstate(PTHREAD_CANCEL_DISABLE,
NULL);
         for (i=0; i < 4; i++)  {
                          printf("I'm              still
running!\n");
                           }
          pthread_setcancelstate(PTHREAD_CANCEL_ENABLE,
NULL);
          pthread_testcancel();
          printf("YOU WILL NOT STOP ME!!!\n");
}
int main(int argc, char * argv[]) {
       pthread_t thread;
       pthread_create(&thread, NULL, thread_func, NULL);
       pthread_cancel(thread);
       printf("Requested to cancel the thread\n");
       pthread_join(thread, NULL);
       printf("The thread is stopped.\n");
return EXIT_SUCCESS;
}
```

Приклад багатопоточної програми

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#define NUM_THREADS 6
void *thread_function(void *arg);
int main() {
int res;
int lots_of_threads;
pthread_t a_thread[NUM_THREADS];
```

```
    void *thread_result;
    srand ((insigned)time(NULL));
    for (lots_of_threads=0; lots_of_threads < NUM_THREADS;
    lots_of_threads++)
              {
               res=pthread_create
(&(a_thread[lots_of_threads]),NULL,
                                          thread_function, (void
*)&lots_of_threads);
                if (res != 0) {
                                perror("Thread          creation
failed");
                                exit(EXIT_FAILURE);
                                }
              }
    printf("Waiting for threads to finish...\n");
    for (lots_of_threads = NUM_THREADS - 1; lots_of_threads >=
0;
    lots_of_threads--)
              {
                  res=pthread_join
(a_thread[lots_of_threads],&thread_result);
                  if (res == 0) printf("Picked up a thread\n");
                           else perror("pthread_join failed");
               }
    printf("All done\n");
    exit(EXIT_SUCCESS);
    }
    void *thread_function(void *arg) {
         int my_number = *(int *)arg;
         int rand_num;
         printf ("thread_function is running. Argument was
%d\n", my_number);

rand_num=1+(int)(9.0*rand()/(RAND_MAX+1.0));
                  printf ("Bye from %d\n", my_number);
                  pthread_exit(NULL);
    }
```

Синхронізація потоків

Усі потоки виконуються в одному адресному просторі. У зв'язку з цим постає проблема спільного використання загальних змінних, доступу до певного ресурсу, оскільки в один момент часу тільки єдиний потік повинен працювати з певним розділюваним ресурсом. Така задача має назву *забезпечення взаємовиключення*, а ділянки програмного коду, в яких потоки виконують операції з розділюваними

ресурсами, називаються *критичними секціями.* З іншого боку можлива ситуація, коли потоку для продовження своєї роботи потрібно результат виконання іншого потоку, що потребує синхронізації дій. Для узгодженості взаємодії потоків розроблені *засоби синхронізації потоків*:

м'ютекси (взаємні виключення), семафори і умовні змінні.

Синхронізація і взаємовиключення забезпечуються за рахунок атомарности виконуваних операцій над м'ютексів і семафора. Атомарної називають операцію, яка не може бути перервана в ході свого виконання.

М'ютекс дозволяє потокам управляти доступом до даних. При використанні м'ютекса тільки один потік в певний момент часу може заблокувати м'ютекс і отримати доступ до ресурсу (право на його використання). По завершенні роботи з ресурсом потік повинен повернути це право, розблокувавши м'ютекс. Якщо будь-який потік звернеться до вже заблокованого м'ютексу, то він буде змушений чекати розблокування м'ютекса потоком, який їм володіє.

Прототипи функцій для виконання операцій над м'ютексів описуються в файлі *pthread.h.* Нижче наводяться прототипи найбільш часто використовуваних функцій разом з поясненням їх синтаксису і виконуваних ними дій.

#include <pthread.h>

pthread_mutex_init (pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);

ініціалізує м'ютекс *mutex* із зазначеними атрибутами attr або з атрибутами за замовчуванням (при вказівці 0 в якості *attr*).

#include <pthread.h>

int pthread_mutex_destroy (pthread_mutex_t *mutex); знищує м'ютекс mutex.

#include <pthread.h>

int pthread_mutex_lock (pthread_mutex_t *mutex);

виконує блокування або замикання викликаючого потоку, поки м'ютекс, вказаний як mutex не стане доступним.

Якщо м'ютекс вже заблокований, то потік, який викликав, буде заблокований до розблокування мьютекса.

#include <pthread.h>

int pthread_mutex_unlock (pthread_mutex_t *mutex);

негайне розблокування або відмикання, або вивільнення м'ютекса mutex.

Приклад використання мьютекса для контролю доступу до змінної. У наведеному нижче коді функція *increment_count* використовує м'ютекс, щоб гарантувати атомарность (цілісність) модифікації розділюваної змінної *count*. Функція *get_count()* використовує м'ютекс, щоб гарантувати, що змінна *count* атомарному зчитується.

```
#include <pthread.h>
pthread_mutex_t count_mutex;
long count;
void increment_count() {
        pthread_mutex_lock(&count_mutex);
        count=count+1;
        pthread_mutex_unlock(&count_mutex);
}
long get_count() {
      long c;
      pthread_mutex_lock(&count_mutex);
      c = count;
      pthread_mutex_unlock(&count_mutex);
      return (c);
}
```

Приклад багатопотокової програми з синхронізацією з використанням м'ютексів

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <math.h>
#define SIZE_I 2
#define SIZE_J 2
float X[SIZE_I][SIZE_J];
float S[SIZE_I][SIZE_J];
int count = 0; // глобальный счетчик
struct DATA_ {
       double x;
       int i;
       int z;
};
typedef struct DATA_ DATA;
pthread_mutex_t lock; // Блокування
// Функция для вычислений
double f(double x) { return x*x; }
// Потокова функція для обчислень
void *calc_thr (void *arg) {
        DATA* a = (DATA*) arg;
```

```
            X[a->i][a->z] = f(a->x);
            // установка блокування
            pthread_mutex_lock(&lock);
            // зміна глобальної змінної
            count ++;
            // зняття блокування
            pthread_mutex_unlock(&lock);
            delete a;
            return NULL;
    }
    // Потокова функція для введення
    void *input_thr(void *arg) {
            DATA* a = (DATA*) arg;
            printf("S[%d][%d]:", a->i, a->z);
            scanf("%f", &S[a->i][a->z]);
            delete a;
            return NULL;
    }
    int main() {
    //масив ідентифікаторів потоків
    pthread_t thr[ SIZE_I * SIZE_J ];
    // ініціалізація м'ютекса
    pthread_mutex_init(&lock, NULL);
    DATA *arg;
    // Введення даних для обробки
    for (int i=0;i<SIZE_I; i++) {
    for (int z=0; z<SIZE_J; z++) {
                arg = new DATA;
                arg->i = i;
                arg->z = z;
    // створення потоку для введення елементів матриці
    pthread_create (&thr[ i* SIZE_J + z ], NULL, input_thr,
(void *)arg);
    }       // for (int z=0; z<SIZE_J; P ++z)
    }   // for (int i=0;i<SIZE_I; P ++i)
    // Очікування завершення усіх потоків введення даних
    // ідентифікатори потоків зберігаються у масиві thr
    for(int  i  =  0;  i  <  SIZE_I*SIZE_J;  i++)  pthread_join
(thr[i], NULL);
    // Обчислення елементів матриці
    pthread_t thread;
    printf("Start calculation\n");
    for (int i=0;i<SIZE_I; i++) {
           for (int z=0; z<SIZE_J; z++) {
                arg=new DATA;
                arg->i=i;
                arg->z=z;
```

```
                arg->x = S[i][z];
                // створення потоку для обчислень
                pthread_create  (&thread,  NULL,  calc_thr,
(void *)arg);
                // переведення потоку у режим від'єнання
        pthread_detach(thread);
                // for (int z=0; z<SIZE_J; z++)
    }      // for (int i=0;i<SIZE_I; i++)
    do {
    // Основний процес "засинає" на 1с
    // Перевірка стану обчислень
    printf("finished %d threads.\n", count);
    } while ( count < SIZE_I*SIZE_J);
    // Виведення результатів
    for (int i=0;i<SIZE_I; i++) {
           for (int z=0; z<SIZE_J; z++) {
                printf("X[%d][%d] = %f\t", i, z, X[i][z]);
    } printf("\n");
    }
    // видалення м'ютекса
    pthread_mutex_destroy(&lock);
    return 0;
    }
```

Семафор призначений для синхронізації потоків щодо дій та даних. Семафор - це захищена змінна, значення якої можна опитувати і міняти тільки за допомогою спеціальних операцій *P* і *V* і операції ініціалізації. Семафор може приймати ціле невід'ємне значення. При виконанні потоком операції *P* над семафором *S* значення семафора зменшується на 1 при S> 0 або потік блокується, «чекаючи на семафорі», при S=0. При виконанні операції *V(S)* відбувається пробудження одного з потоків, які очікують на семафорі *S*, а якщо таких немає, то значення семафора збільшується на 1. Як випливає з вищесказаного, при вході в критичну секцію потік повинен виконувати операцію *P*, а при виході з критичної секції операцію *V*.

Прототипи функцій для маніпуляції з семафора описуються у файлі ***semaphore.h.*** Нижче наводяться прототипи функцій разом з поясненням їх синтаксису і виконуваних ними дій.

int sem_init (sem_t *sem, int pshared, unsigned int value); - ініціалізація семафора *sem* значенням *value*. В якості *pshared* завжди необхідно вказувати 0.

int sem_wait (sem_t *sem); - «очікування на семафорі». Виконання потоку блокується до тих пір, поки значення семафора не стане позитивним. При цьому значення семафора зменшується на 1.

int sem_post (sem_t *sem); - збільшує значення семафора *sem*.

int sem_destroy (sem_t *sem); - знищує семафор *sem*.

int sem_trywait (sem_t *sem); - неблокуючий варіант функції *sem_wait*. При цьому замість блокування викликав потоку функція повертає управління з кодом помилки в якості результату роботи.

Приклад багатопотокової програми з синхронізацією семафора

```
#include "main.h"
#include <iostream.h>
#include <semaphore.h>
#include <fstream.h>
#include <stdio.h>
#include <error.h>
void* WriteToFile(void*);
int errno;
sem_t psem;
ofstream qfwrite;

int main(int argc, char **argv) {
pthread_t tidA,tidB;
int n;
char filename[]="./rezult.txt";
qfwrite.open(&filename[0]);
sem_init(&psem,0,0);
sem_post(&psem))
pthread_create(&tidA,NULL,&WriteToFile,(void*)100));
        pthread_create(&tidB,NULL,&WriteToFile,(void*)100));
        pthread_join(tidA,NULL));
        pthread_join(tidB,NULL));
sem_destroy(&psem));
qfwrite.close();
}
void* WriteToFile(void *f){
int max = (int)f;
for (int i=0; i<=max; i++)
{
          sem_wait(&psem);
          qfwrite<<pthread_self()<<"-writetofilecounter
i="<<i<<endl;
          qfwrite<<flush;
```

```
        sem_post(&psem);
    }
    return NULL;
    }
```

Умовна змінна дозволяє потокам очікувати виконання деякої умови (події), пов'язаної з розділюваними даними. Над умовними змінними визначені дві основні операції: *інформування* про настання події і *очікування* події. При виконанні операції «*інформування*» один з потоків, які очікують на умовну змінну, відновлює свою роботу. Умовна змінна завжди використовується спільно з м'ютексів. Перед виконанням операції «очікування» потік повинен заблокувати м'ютекс. При виконанні операції «очікування» зазначений м'ютекс автоматично розблокується. Перед відновленням очікує потоку виконується автоматичне блокування м'ютекса, що дозволяє потоку увійти в критичну секцію, після критичної секції рекомендується розблокувати м'ютекс. При подачі сигналу іншим потокам рекомендується так само функцію «сигналізації» захистити м'ютексом.

Прототипи функцій для роботи з умовними змінними містяться в файлі ***pthread.h.*** Нижче наводяться прототипи функцій разом з поясненням їх синтаксису і виконуваних ними дій.

pthread_cond_init (pthread_cond_t *cond, const pthread_condattr_t *attr); - ініціалізує умовну змінну *cond* із зазначеними атрибутами *attr* або з атрибутами за замовчуванням (при вказівці 0 в якості attr).

int pthread_cond_destroy (pthread_cond_t *cond); - знищує умовну змінну *cond*.

int pthread_cond_signal (pthread_cond_t *cond); - інформування про настання події потоків, які очікують на умовній змінної *cond*.

int pthread_cond_broadcast (pthread_cond_t *cond); - інформування про настання події потоків, які очікують на умовній змінної *cond*. При цьому відновлені будуть все очікують потоки.

int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mutex); - очікування події на умовну змінну *cond*.

Перенаправлення введення/виведення в Linux

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

Введення і виведення розподіляється між трьома стандартними потоками:

stdin - стандартне введення (клавіатура), номер потоку - 0;
stdout - стандартне виведення (екран), номер потоку - 1;
stderr - стандартна помилка (виведення помилок на екран), номер потоку - 2.

Зі стандартного введення команда може тільки зчитувати дані, а два інших потоки можуть використовуватися тільки для запису. Дані виводяться на екран і зчитуються з клавіатури, так як стандартні потоки за замовчуванням асоційовані з терміналом користувача. Потоки можна підключати до чого завгодно: до файлів, програм і навіть пристроїв. У командному інтерпретаторі *bash* така операція називається перенаправленням:

<file - використовувати файл як джерело даних для стандартного потоку введення,
>file - направити стандартний потік виведення в файл; якщо файл не існує, він буде створений, якщо існує - перезаписаний зверху;
2*>file* - направити стандартний потік помилок в файл; якщо файл не існує, він буде створений, якщо існує - перезаписаний зверху;
>>file - направити стандартний потік виведення в файл; якщо файл не існує, він буде створений, якщо існує - дані будуть дописані до нього в кінець;
2>>file - направити стандартний потік помилок в файл; якщо файл не існує, він буде створений, якщо існує - дані будуть дописані до нього в кінець;
&>file або> *&file* - направити стандартний потік виведення і стандартний потік помилок в файл; інша форма запису*:> file 2>&1*.

Стандартне введення - стандартний вхідний потік передає дані від користувача до програми.

cat> myfile - введення тексту з клавіатури у файл, після введення кожного рядка натискається *Enter*, по завершенні введення тексту натискається Ctrl + D, що означає кінець файлу EOF.

cat myfile - виведення інформації на екран з файлу.

cat file1 >> file2 - дописати вміст файлу file1 у файл file2.

Команда ***cat*** зазвичай використовується для об'єднання вмісту файлів.

cat file1 file2 file3 > file4 команда об’єднання трьох файлів в один файл file4.

Стандартне виведення - стандартний вихідний потік не перенаправляється в який-небудь файл, а виводить текст на дисплей терміналу. Команда ***echo*** виводить на

екран будь-який аргумент (текст або значення змінних), який передається йому в командному рядку: ***echo*** Example.

echo> file1 "текст" - перенаправлення виведення за допомогою символу ">", якщо файл із таким ім'ям вже існує, то він буде перезаписаний;

echo>>file2 "***текст додається***" - перенаправлення виведення за допомогою символу ">>", новий текст буде додано в кінець файлу;

echo "текст на принтер" | lp - передача стандартного виведення однієї команди на стандартний вхід іншої за допомогою символу "|", текст необхідно роздрукувати на принтері.

Канали. Канали використовуються для перенаправлення потоку з однієї програми в іншу.

Особливим варіантом перенаправлення виведення є організація програмного каналу (іноді називає трубопроводом або конвеєром, оператор «|»). Для цього дві або декілька команд, таких, як виведення попередньої слугує введенням для наступної та розділяються символом вертикальної риски – «|». При цьому стандартний вихідний потік команди, розташованої ліворуч від символу «|», направляється на стандартне введення програми, розташованої праворуч від символу «|», наприклад:

cat myfile | grep Linux | wc -l .

Команда ***grep*** відшукує слово ***Linux*** у файлі, команда ***wc -l*** обчислює кількість рядків у слові. Рядок означає, що виведення команди *cat*, тобто з файлу *myfile*, буде спрямовано на вхід команди *grep*, яка виділить тільки рядки, що містять слово *"Linux"*. Виведенняк команди *grep* буде, в свою чергу, спрямовано на вхід команди *wc -l*, яка підрахує кількість таких рядків.

Програмні канали використовуються для того, щоб скомбінувати кілька маленьких програм, кожна з яких виконує тільки певні перетворення над своїм вхідним потоком, для створення узагальненої команди, результатом якої буде якесь більш складне перетворення.

Завдання:

1. Опанувати команди по роботі з потоками.
2. Ознайомитися із засобами синхронізації потоків.

3. Ознайомитися із стандартними потоками введення/виведення.
4. Підготувати звіт для викладача про виконання лабораторної роботи і представити його.

Хід виконання роботи

1. Використати вихідні тексти для створення програм.
2. Пояснити результати роботи програм та їх особливості.
3. Створити файл з особистими даними (група, прізвище, ім’я, по-батькові), використовуючи стандартне введення з клавіатури. Вивести дані на екран. Додати у файл інформацію про ваше хоббі. Створити канал.

Підготувати звіт

1. Описати хід виконання поставлених завдань, надаючи знімок екрану (screenshot).
2. Висновки по роботі.

Контрольні питання

1. У чому полягає відмінність потоку і процесу?
2. Які вам відомі типи потоків?
3. Які вам відомі основні функції при роботі з потоками?
4. Які вас відомі засоби синхронізації потоків?
5. Які стандартні потоки введення/виведення?
6. Які види перенаправлення вам відомі?
7. Що таке канали?

КОМП'ЮТЕРНИЙ ПРАКТИКУМ 8

Установка Docker

Мета роботи - набути навичок встановлювати додаток Docker в ОС Linux, надати основи для роботи на постійній основі з образами та контейнерами, що дозволяє не засмічувати робочу машину локально встановленими різними версіями низки програмного забезпечення: *apache, mysql, virtualenv, python, mongodb, memchaced, redis, php,* а також подібного програмного забезпечення, яке використовується при розробці проектів та часто ще й конфліктує між собою від версії до версії.

Теоретичні відомості

Docker є найпопулярнішою платформою управління контейнерами. Це програмне забезпечення з відкритим кодом, принцип роботи якого найпростіше порівняти з транспортними контейнерами. Філософію Docker часто описують за допомогою метафори «доставки універсальних вантажних контейнерів», тобто стандартизованих розмірів контейнерів, які можна переміщувати між різними видами транспорту (вантажівками, поїздами, кораблями) з мінімумом ручної праці. Така ідея була перенесена на IT-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 т. д. Може мати місце така ситуація, коли додаток працює на вашому комп'ютері, але відмовляється працювати на комп'ютері іншого користувача. Ця проблема вирішується через створення програмного забезпечення, яке не залежить від системи.

Саме контейнери Docker спрощують перенесення програмних додатків.

Термінологія

Контейнери - це *технологія упаковки і запуску додатків* Windows, Linux, MacOS в різних локальних середовищах і в хмарі.

Контейнер - *це виконуваний екземпляр, який інкапсулює необхідну програмне забезпечення. Він складається з образів.* Його можна легко видалити і знову створити за короткий проміжок часу. Контейнери надають невимогливе до ресурсів ізольоване

середовище, яке спрощує розробку, розгортання, запуск програмного забезпечення, особливо в динамічних і розподілених середовищах та керування додатками.

Образ - *базовий елемент кожного контейнера*. Залежно від способу, може знадобитися деякий час для його створення.

Порт - це *порт TCP/UDP* (протоколи транспортного рівня для передачі пакетів між комп'ютерами) в своєму первинному значенні. Щоб все було просто, припустимо, що порти можуть бути відкриті в зовнішньому світі або підключені до контейнерів (доступні тільки з цих контейнерів і невидимі для зовнішнього світу).

Том - описується як *загальна папка*. Тома ініціалізуються при створенні контейнера і призначені для збереження даних, незалежно від життєвого циклу контейнера.

Реєстр - це *сервер, на якому зберігаються образи*. Порівняємо його з GitHub: ви можете витягнути образ з реєстру, щоб розгорнути його локально, і так само локально можете вносити в реєстр створені образи.

Docker Hub - *публічний репозиторій* з інтерфейсом, що надається Docker Inc. Він *зберігає безліч образів*. Ресурс є джерелом «офіційних» образів, зроблених командою Докер або створених у співпраці з розробником програмного забезпечення. Для офіційних образів перераховані їх потенційні уразливості. Ця інформація відкрита для будь-якого зареєстрованого користувача. Доступні як безкоштовні, так і платні акаунти.

Контейнери створюються на основі ядра операційної системи сервера, але не отримують необмежений доступ до ядра. Наприклад, контейнер може звертатися до віртуалізованої версії файлової системи і реєстру, але будь-які зміни стосуються тільки контейнера і видаляються при його зупинці. Контейнер збирається поверх ядра, але ядро не надає всі інтерфейси API і служби, необхідні для запуску програми. Більшість з 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 в якості основного рівня, на якому збирається контейнер, надаючи йому служби операційної системи, які не надаються ядром.

Таким чином, Docker використовує не віртуалізацію, а засоби ядра, які дозволяють створювати ізольовані групи процесів. При запуску Docker робить лише кілька системних викликів і ядро створює для нового процесу окремий простір PID-ів, окрему віртуальну мережу, окремий набір обмежень по ресурсах. Ядро асоціює Docker зі специфічним набором налаштувань.

На відміну від контейнера, віртуальна машина (ВМ) працює під управлінням повноцінної операційної системи, включаючи її власне ядро, і є повною емуляцією іншого програмного (операційного) середовища. Перевагами та метою створення контейнерів є прискорення розробки, інкапсуляція додатків (залежностей додатків, операційних систем) та переносимість програмного забезпечення.

Всі контейнери створюються з образів контейнерів. Образи контейнерів представляють собою набір файлів, організованих в стек шарів, розташованих на локальному комп'ютері або у віддаленому реєстрі контейнерів. Образ контейнера складається з файлів операційної системи режиму користувача, необхідних для підтримки додатку, будь-яких середовищ виконання або залежностей додатків, а також будь-якого іншого файлу конфігурації, необхідного для правильної роботи додатка.

Таким чином, Docker – це стандартизоване пакетне програмне забезпечення, призначене для розробки, розгортання проектів та використання розроблених додатків. Docker дозволяє відокремити ваш додаток від вашої інфраструктури і дозволяє запустити буд-який додаток, який безпечно ізольований у контейнері. Docker має особливі образи програмного забезпечення, що запускаються у віртуальному середовищі, не створюючи повну копію ОС).

Одним з найбільш поширених варіантів використання контейнерів є мікросервіси (microservices). Мікросервіси – це спосіб розробки та компонування програмних систем, при якому вони формуються з невеликих незалежних компонентів, що взаємодіють один з одним через мережу. 64-бітовий Linux-контейнер працює тільки на хості з встановленою 64-бітовою версією ОС Linux.

Docker доступний для будь-якої з операційних систем: Windows, Linux, Mac OS. Docker ставиться на версію Ubuntu 18.04, на Ubuntu 19.10 не ставиться. Docker

дозволяє запустити ОС Linux в ізольованому середовищі дуже швидко, протягом декількох хвилин. Платформа Docker складається з двох окремих компонентів:

- *Docker Engine*, механізму, що відповідає за створення і функціонування контейнерів,
- *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, були потрібні неабиякий запас спеціальних знань в цій області і великий обсяг ручної роботи. *Docker Hub* надає величезну кількість образів контейнерів з відкритим доступом для завантаження, дозволяючи користувачам швидко почати роботу з ними і уникнути рутинної роботи, раніше вже виконану іншими людьми.

Трохи пізніше були розроблені ***інструментальні засоби для Docker***:

- ***Swarm*** - менеджер кластерів,
- ***Kinematic*** - графічний користувальницький інтерфейс для роботи з контейнерами;
- ***Machine*** - утиліта командного рядка для підтримки роботи Docker-хостів.

Установка Docker

1). Дистрибутив Docker, доступний в офіційному репозиторії Ubuntu, не завжди є останньою версією програми. Доцільно встановити останню версію Docker, завантаживши її з офіційного репозиторію Docker. Для цього додаємо новий джерело дистрибутива, вводимо ключ GPG з репозиторія Docker, щоб переконатися, чи дійсна завантажена версія, а потім встановлюємо дистрибутив.

Спочатку оновлюємо існуючий перелік пакетів: ***sudo apt update***

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

Потім додаємо в свою систему ключ GPG офіційного репозиторію Docker:

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -

Додаємо репозиторій Docker в список джерел пакетів *APT*:

sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu bionic stable"

Далі оновимо базу даних пакетів інформацією про пакети Docker з знову доданого сховища: ***sudo apt update***

Слід переконатися, що ми встановлюємо Docker з репозиторію Docker, а не з репозиторію за замовчуванням Ubuntu: ***apt-cache policy docker-ce***

На екран буде виведена наступна інформація (номер версії Docker може бути іншим):

```
docker-ce:
  Installed: (none)
  Candidate: 18.03.1~ce~3-0~ubuntu
  Version table:
     18.03.1~ce~3-0~ubuntu 500
     500          https://download.docker.com/linux/ubuntu
bionic/stable amd64 Packages
```

Зверніть увагу, що *docker-ce* не встановлюється, але для установки буде використаний репозиторій Docker для Ubuntu 18.04 (*bionic*).

Далі встановлюємо Docker: ***sudo apt install docker-ce***

Docker встановлений, демон запущений, і процес буде запускатися при завантаженні системи. Переконаємося, що процес запущений:

sudo systemctl status docker

На екран виводиться наступна інформація, сервіс повинен бути запущений і активний:

```
Output
  docker.service - Docker Application Container Engine
   Loaded: loaded (/lib/systemd/system/docker.service;
   enabled; vendor preset: enabled)
   Active: active (running) since Thu 2018-07-05 15:08:39
UTC; 2min 55s ago
     Docs: https://docs.docker.com
Main PID: 10096 (dockerd)
   Tasks: 16
  CGroup: /system.slice/docker.service
     ├─10096 /usr/bin/dockerd -H fd://
     └─10113 docker-containerd --config
/var/run/docker/containerd/containerd.toml
```

При установці Docker ми отримуємо не тільки сервіс (демон) Docker, але і утиліту командного рядка docker або клієнт Docker. Використання утиліти командного рядка docker розглянуто нижче.

2). Використання команди Docker без sudo (необов'язково)

За замовчуванням команду ***docker*** може запустити користувач *root* або користувач з групи *docker*, яка автоматично створюється при установці *Docker*. Якщо ви хочете запустити команду *docker* без префікса *sudo* або від імені користувача, що не входять в групу *docker*, будуть виведені дані:

```
Output
docker: Cannot connect to the Docker daemon. Is the docker
daemon running on this host?.
See 'docker run --help'.
```

Щоб не вводити *sudo* кожний раз при запуску команди *docker*, додайте ім'я свого користувача у групу *docker*: ***sudo usermod -aG docker ${USER}.***

Для застосування цих змін у складі групи необхідно разлогінитися і знову залогінитися на сервері або задати наступну команду: ***su - ${USER}.***

Для продовження роботи необхідно ввести пароль користувача.

Щоб переконатися, що користувач доданий у групу *docker*, слід набрати команду: ***id -nG.*** На екран виведеться:

```
Output
username sudo docker
```

Якщо ви хочете додати довільного користувача в групу *docker*, можна вказати конкретне ім'я користувача: ***sudo usermod -aG docker username***

3). Використання команди *Docker*

Команда *docker* дозволяє використовувати різні опції, команди з аргументами. Синтаксис команди наступний: ***docker [option] [command] [arguments].***

Для перегляду усіх доступних підкоманд введіть: ***docker***

Повний список підкоманди Docker:

Output

attach Attach local standard input, output, and error streams to a running container,
build Build an image from a Dockerfile.
commit Create a new image from a container's changes,
cp Copy files/folders between a container and the local filesystem,
create Create a new container,
diff Inspect changes to files or directories on a container's filesystem,
events Get real time events from the server,
exec Run a command in a running container,
export Export a container's filesystem as a tar archive,
history Show the history of an image,

images List images,
import Import the contents from a tarball to create a filesystem image,
info Display system-wide information,
inspec Return low-level information on Docker objects,
kill Kill one or more running containers,
load Load an image from a *tar* archive or STDIN,
login Log in to a Docker registry,
logout Log out from a Docker registry,
logs Fetch the logs of a container,
pause Pause all processes within one or more containers,
port List port mappings or a specific mapping for the container,
ps List containers,
pull Pull an image or a repository from a registry,
push Push an image or a repository to a registry,
rename Rename a container,
restart Restart one or more containers,
rm Remove one or more containers,
rmi Remove one or more images,
run Run a command in a new container,
save Save one or more images to a tar archive (streamed to STDOUT by default),
search Search the Docker Hub for images,
start Start one or more stopped containers,
stats Display a live stream of container(s) resource usage statistic,
stop Stop one or more running containers,
tag Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE,
top Display the running processes of a container,
unpause Unpause all processes within one or more containers,
update Update configuration of one or more containers,
version Show the Docker version information
wait Block until one or more containers stop, then print their exit codes.

Для перегляду опцій використання певної команди введіть:

docker docker-subcommand --help

Для перегляду всієї інформації про *Docker* використовується наступна команда:

docker info

4). *Робота з образами 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 він підтримується компанією *Docker*. Будь-хто може створити і завантажити свої образи *Docker* в *Docker Hub*, тому для більшості додатків і дистрибутивів *Linux*, які можуть знадобитися вам для роботи, вже є відповідні образи в *Docker Hub*.

Для перевірки чи ви маєте доступ до образів і можете завантажувати образи з Docker Hub, введіть наступну команду: ***docker run hello-world***

Коректний результат роботи цієї команди, який означає, що Docker працює правильно, наведений нижче:

```
Output
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
9bb5a5d4561a: Pull complete
Digest:
sha256:3e1764d0f546ceac4565547df2ac4907fe46f007ea229fd7ef271
8514bcec35d
Status: Downloaded newer image for hello-world:latest
Hello from Docker!
This message shows that your installation appears to be working correctly.
```

Спочатку *Docker* не міг знаходити образ hello-world локально, тому завантажував образ з *Docker Hub*, який є репозиторієм за замовчуванням. Після завантаження образу *Docker* створював з образу контейнер і запускав додаток в контейнері, відображаючи повідомлення. Образи, доступні в Docker Hub, можна шукати за допомогою команди *docker* і підкоманди *search*. Наприклад, для пошуку образу *Ubuntu* вводимо: ***docker search ubuntu***

Скрипт переглядає *Docker Hub* і повертає список всіх образів, імена яких підходять під заданий пошук. Ми отримаємо наступний результат:

```
Output
NAME                                                DESCRIPTION
ubuntu                                              Ubuntu is a Debian-based Linux operating sys…  917
dorowu/ubuntu-desktop-lxde-vnc                      Ubuntu with openssh-server and NoVNC  193
rastasheep/ubuntu-sshd                              Dockerized SSH service, built on top of offi…  156
ansible/ubuntu14.04-ansible                         Ubuntu 14.04 LTS with ansible  93
ubuntu-upstart                                      Upstart is an event-based replacement for th…  87
neurodebian                                         NeuroDebian provides neuroscience research s…  50
ubuntu-debootstrap                                  debootstrap --variant=minbase --components=m…  38
1and1internet/ubuntu-16-nginx-php-phpmyadmin-mysql-5  ubuntu-16-nginx-php-
phpmyadmin-mysql-5                                  36
nuagebec/ubuntu                                     Simple always updated Ubuntu docker images w…  23
tutum/ubuntu                                        Simple Ubuntu docker images with SSH access  18
i386/ubuntu                                         Ubuntu is a Debian-based Linux operating sys…  13
ppc64le/ubuntu                                      Ubuntu is a Debian-based Linux operating sys…  12
1and1internet/ubuntu-16-apache-php-7.0              ubuntu-16-apache-php-7.0  10
```

1and1internet/ubuntu-16-nginx-php-phpmyadmin-mariadb-10 ubuntu-16-nginx-php-phpmyadmin-mariadb-10 6
eclipse/ubuntu_jdk8 Ubuntu, JDK8, Maven 3, git, curl, nmap, mc, 6
codenvy/ubuntu_jdk8 Ubuntu, JDK8, Maven 3, git, curl, nmap, mc, 4
darksheer/ubuntu Base Ubuntu Image -- Updated hourly 4
1and1internet/ubuntu-16-apache ubuntu-16-apache 3
1and1internet/ubuntu-16-nginx-php-5.6-wordpress-4 ubuntu-16-nginx-php-5.6-wordpress-4 3
1and1internet/ubuntu-16-sshd ubuntu-16-sshd 1
pivotaldata/ubuntu A quick freshening-up of the base Ubuntu doc
1and1internet/ubuntu-16-healthcheck ubuntu-16-healthcheck 0
pivotaldata/ubuntu-gpdb-dev Ubuntu images for GPDB development 0
smartentry/ubuntu ubuntu with smartentry 0
ossobv/ubuntu
...

Коли потрібний образ обраний, можна завантажити його на комп'ютер за допомогою підкоманди ***pull.*** Щоб завантажити офіційний образ *ubuntu* на комп'ютер, запускається наступна команда: ***docker pull ubuntu***

Отримуємо наступний результат:

```
Output
Using default tag: latest
latest: Pulling from library/ubuntu
6b98dfc16071: Pull complete
4001a1209541: Pull complete
6319fc68c576: Pull complete
b24603670dc3: Pull complete
97f170c87c6f: Pull complete
Digest:
sha256:5f4bdc3467537cbbe563e80db2c3ec95d548a9145d64453b06939
c4592d67b6d
Status: Downloaded newer image for ubuntu:latest
```

Після завантаження образу можна запустити контейнер із завантаженим образом за допомогою підкоманди ***run***. Як видно з прикладу *hello-world*, якщо при виконанні *docker* за допомогою підкоманди *run* образ ще не завантажений, клієнт *Docker* спочатку завантажить образ, а потім запустить контейнер з цим образом.

Для перегляду завантажених на комп'ютер образів потрібно ввести:

docker images

Отримаємо виведення на екран:

Output
REPOSITORY TAG IMAGE ID CREATED SIZE

ubuntu	latest	113a43faa138	4 weeks ago	81.2MB
hello-world	latest	e38bc07ac18e	2 months ago	1.85kB

Образи, які використовуються для запуску контейнерів, можна змінювати і застосовувати для створення нових образів, які, в свою чергу, можуть бути завантажені (технічний термін ***push***) в *Docker Hub* або інший *Docker*-реєстр.

5). ***Запуск контейнера Docker***

Контейнер *hello-world*, запущений на попередньому етапі, є прикладом контейнера, який запускається і завершує роботу після виведення тестового повідомлення. Контейнери можуть виконувати і більш корисні дії, а також можуть бути інтерактивними. Контейнери схожі на віртуальні машини, але є менш вимогливими до ресурсів.

Як приклад запустимо контейнер за допомогою останньої версії образу Ubuntu. Комбінація параметрів ***-i*** та ***-t*** забезпечує інтерактивний доступ до командного процесора контейнера: ***docker run -it ubuntu***

Командний рядок повинен змінитися, показуючи, що ми тепер працюємо в контейнері. Вона буде мати наступний вигляд:

Output

root @ d9b100f2f636: / #

В командному рядку відображається ідентифікатор контейнера. В даному прикладі це *d9b100f2f636.* Ідентифікатор контейнера потрібно нам пізніше, щоб вказати, який контейнер необхідно видалити.

Тепер можна запускати будь-які команди всередині контейнера. Наприклад, оновити базу даних пакета всередині контейнера. Перед командами не потрібно використовувати *sudo*, оскільки ви працюєте всередині контейнера як користувач з привілеями ***root***: ***apt update.*** Тепер в контейнері можна встановити будь-який додаток. Спробуємо встановити Node.js: ***apt install nodejs***

Ця команда встановлює *Node.js* в контейнер з офіційного репозиторію *Ubuntu*. Коли установка завершена, переконаємося, що *Node.js* встановлений: ***node -v***

У терміналі з'явиться номер версії:

Output
v8.10.0

Всі зміни, які ви здійснюєте всередині контейнера, застосовуються тільки для цього контейнера. Щоб вийти з контейнера, вводимо команду ***exitє.***

6). *Управління контейнерами Docker*

Через деякий час після початку використання *Docker* на вашій машині буде безліч активних (запущених) і неактивних контейнерів.

Перегляд ** активних контейнерів **: ***docker ps***

Результат перегляду:

```
Output
CONTAINER ID        IMAGE          COMMAND        CREATED
```

Було запущено два контейнери: один з образу *hello-world*, другий з образу *ubuntu*. Обидва контейнера вже не запущені, але існують в системі.

Щоб побачити і активні, і неактивні контейнери, запускаємо *docker ps* за допомогою параметра ***-a:*** ***docker ps -a***

Результат наступний:

d9b100f2f636 ubuntu "/bin/bash" About an hour ago Exited (0) 8 minutes ago sharp_volhard

01c950718166 hello-world "/hello" About an hour ago Exited (0) About an hour ago festive_williams

Для перегляду останнього створеного контейнерів, 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

Контейнер запускається. Тепер для перегляду його статусу можна використовувати ***docker ps***:

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

d9b100f2f636 ubuntu "/bin/bash" About an hour ago Up 8 seconds sharp_volhard

Для зупинки запущеного контейнера використовуємо команду ***docker stop***, потім вказуємо *ідентифікатор контейнера* або його *ім'я.* Цього разу ми використовуємо ***ім'я, яке призначив контейнеру Docker***, тобто ***sharp_volhard***:

docker stop sharp_volhard

Якщо вам ***контейнер*** більше не потрібен, ***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*, використовуйте 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 віддалиться після зупинки. Для більш докладної інформації про дані та інших опціях використовуйте команду ***docker run help***.

Контейнери можна перетворити в образи для побудови нових контейнерів. Розглянемо, як це зробити.

7). ***Збереження змін в контейнері в образ Docker***

При запуску контейнера з образу *Docker* ви можете створювати, змінювати і видаляти файли, як і на віртуальній машині. Внесені зміни застосовуються тільки для такого контейнера. Можна запускати і зупиняти контейнер, проте як тільки він буде знищений командою *docker rm*, всі зміни будуть безповоротно втрачені. В даному розділі показано, як зберегти стан контейнера у вигляді нового образу *Docker*.

Після установки *Node.js* в контейнері *Ubuntu* у вас буде працювати запущений з образу контейнер, але він буде відрізнятися від образу, який ви використовували для його створення. Однак вам може знадобитися такий контейнер *Node.js* як основа для майбутніх образів. Далі підтверджуємо зміни в новому образі *Docker* за допомогою наступної команди.

docker commit -m "What you did to the image" -a "Author Name" container_id repository /new_image_name

Параметр «**-m»** дозволяє задати повідомлення про підтвердження, щоб полегшити користувачам образу розуміння того, які зміни були внесені, а параметр «**-a»** дає змогу вказати автора. Ідентифікатор контейнера *container_id* - цей ідентифікатор, який використовувався раніше, коли починали інтерактивну сесію в контейнері *Docker*. Якщо ви не створювали додаткових репозиторіїв в *Docker Hub*, ім'я сховища (*repository*) зазвичай є вашим ім'ям користувача в *Docker Hub*.

Наприклад, для *користувача* ***sammy*** та *ідентифікатора контейнера* d9b100f2f636 команда виглядає наступним чином:

```
docker commit -m "added Node.js" -a "sammy" d9b100f2f636 sammy/ubuntu-nodejs
```

Після підтвердження (*commit*) образу новий образ зберігається локально на вашому комп'ютері. Для того щоб розмітити образ в реєстр *Docker* (наприклад, в *Docker Hub)* так, щоб він був доступний не тільки вам, а й іншим користувачам, необхідно виконати наступні дії. Для перегляду списку образів *Docker*, в ньому з'являться і новий образ, і початковий образ, на якому він був заснований виконаємо команду: ***docker images***

Результат буде наступним:

OutputREPOSITORY	*TAG*	*IMAGE ID*	*CREATED*	*SIZE*
sammy/ubuntu-nodejs	*latest*	*7c1f35226ca6*	*7 seconds ago*	*179MB*
ubuntu	*latest*	*113a43faa138*	*4 weeks ago*	*81.2MB*
hello-world	*latest*	*e38bc07ac18e*	*2 months ago*	*1.85kB*

У цьому прикладі *ubuntu-nodejs* - це новий образ, створений на основі існуючого образу *ubuntu* з *Docker Hub*. Різниця розмірів відображає внесені зміни. В даному прикладі зміна пов'язана з установкою *NodeJS*. У випадку, коли буде потрібно запустити контейнер *Ubuntu* з передвстановленим *NodeJS*, можна використовувати цей новий образ. Образи також можна створювати за допомогою файлу *Dockerfile*, який дозволяє автоматизувати установку програм в новому образі.

Новим образом можна поділитися з іншими користувачами, щоб вони могли створювати на його основі контейнери.

8). ***Завантаження контейнерів Docker в репозиторій Docker***

Для завантаження образів в *Docker Hub* або інший *Docker-реєстр*, до якого у вас є доступ, ви повинні мати у ньому обліковий запис.

Для *створення власного Docker-реєстр*у та його налаштування можна скористатися статтею *How To Set Up a Private Docker Registry on Ubuntu 14.04*: *https://www.digitalocean.com/community/tutorials/how-to-set-up-a-private-docker-registry-on-ubuntu-14-04*. Для того щоб завантажити свій образ *Docker у Docker Hub*, треба увійти в *Docker Hub*: ***docker login -u docker-registry-username***

Для входу у *Docker Hub* потрібно ввести пароль. Після введення правильного паролю ви будете успішно авторизовані. Якщо ім'я користувача в *Docker-реєстрі*

відрізняється від локального імені користувача, яке використовувалося для створення образу, необхідно прив'язати свій образ до імені користувача в реєстрі. Для цього вводимо команду з урахуванням попереднього приклад:

docker tag sammy/ubuntu-nodejs docker-registry-username/ubuntu-nodejs

Далі завантажуємо власний образ *Docker Hub*:

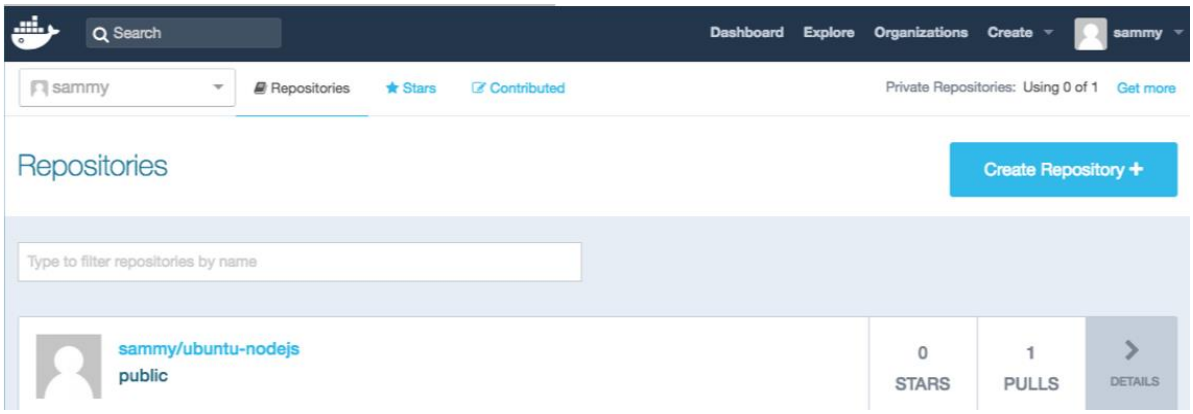
docker push docker-registry-username/docker-image-name

Команда для завантаження образу *ubuntu-nodejs* в репозиторій *sammy* виглядає наступним чином: ***docker push sammy/ubuntu-nodejs***

Для завантаження образу може знадобитися деякий час, але після завершення результат буде виглядати наступним чином:

Output
The push refers to a repository [docker.io/sammy/ubuntu-nodejs]
e3fbbfb44187: Pushed
5f70bf18a086: Pushed
a3b5c80a4eba: Pushed
7f18b442972b: Pushed
3ce512daaf78: Pushed
7aae4540b42d: Pushed

Після завантаження образу в реєстр його ім'я з'являється в списку панелі управління вашого профілю, як показано нижче:



Якщо при завантаженні з'являється наступна помилка (це означає, що не виконано вхід до реєстру):

Output
The push refers to a repository [docker.io/sammy/ubuntu-nodejs]
e3fbbfb44187: Preparing
5f70bf18a086: Preparing
a3b5c80a4eba: Preparing
7f18b442972b: Preparing
3ce512daaf78: Preparing

7aae4540b42d: Waiting
unauthorized: authentication required
необхідно повторити акторизацію.

Для авторизації в реєстрі повторюємо команду ***docker login*** та завантажуємо образ. Потім треба перевірити, що він з'явився на вашій сторінці в репозиторії *Hub*.

Далі за допомогою команди ***docker pull sammy ubuntu-nodejs*** можна завантажити образ на нову машину і використовувати його для запуску нового контейнера.

Завдання:

1. Ознайомитися з теоретичними матеріалом по лабораторній роботі.
2. Опанувати команди, які використовують при установці *Docker* та його встановити.
3. Підготувати звіт з описом процесу установки *Docker* з наведенням *screenshot*-ів екрану при виконанні кожної дії, надати його для викладача.

Хід виконання роботи

Встановити дистрибутив *Docker*.

Підготувати звіт

1. Описати хід виконання поставлених завдань, надаючи знімок екрану (*screenshot*).
2. Висновки по роботі.

Контрольні питання

1. Що таке Docker?
2. Що таке контейнер?
3. Що таке образ?
4. Що таке реєстр?
5. Що таке репозитарій?
6. Які відмінності між віртуальною машиною та контейнерами?
7. У яких операційних системах можна встановлювати Docker?
8. З яких компонентів складається платформа Docker?
9. Які вам відомі підкоманди Docker?
10. Як запустити команду *docker* без префікса *sudo?*
11. Як працювати з образами Docker?
12. Як запустити контейнер Docker, зупинити та видалити?
13. Як завантажити контейнер Docker в репозитарій Docker?

КОМП'ЮТЕРНИЙ ПРАКТИКУМ 9.1

Створення проекту для web-розробки, який складається наборів контейнерів з використанням Docker Compose та Dockerfile

Мета роботи – ознайомитися та набути навичок:

- написання скрипта **Dockerfile** для створення контейнеру;
- встановлення **Docker Compose в Ubuntu 18.04**;
- створення контейнерів (сервісів): **Nginx+Php-Fpm+ MySQL** для розроблення web-додатку для **Magento2**;

Теоретичні відомості

Для створення контейнерів необхідно вміти працювати з **Docker Compose** та **Dockerfile.**

Dockerfile та синтаксис для їх створення

Dockerfile - скрипт, який дозволяє автоматизувати процес побудови контейнерів шляхом виконання відповідних команд (дій) в *base* образі для формування нового образу.

Усі подібні файли починаються з позначення ***FROM*** так як і процес побудови нового контейнера, далі йдуть різні методи, команди, аргументи або умови, після застосування яких створиться Docker контейнер.

Розглянемо синтаксис *Dockerfile*. В Докер файлах міститься два типи основних блоків - *коментарі та команди з аргументами*. Причому для всіх команд передбачається певний порядок. Нижче наведено типовий приклад синтаксису, де перший рядок є коментарем, а другий - командою.

Print «Hello from User!»
RUN echo «Hello from User!!»

Розглянемо усі можливі команди. Усі команди в Докерфайлах прийнято вказувати великими літерами - наприклад **RUN**, **CMD** і т.д.

- Команда ***ADD*** - бере два аргументи, шлях звідки 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, використовується для виконання певних програм, але, на відміну від RUN дана команда зазвичай застосовується для запуску/ініціації додатків або команд вже після їх установки за допомогою RUN в момент побудови контейнера.

  Синтаксис команди: ***CMD %додаток% «аргумент», «аргумент», ..***

  CMD «echo» «Hello from User!»

- Команда ***ENTRYPOINT*** встановлює конкретний додаток за замовчуванням, який використовується кожний раз в момент побудови контейнера за допомогою образу. Наприклад, якщо ви встановили певний додаток всередині образу і ви збираєтеся використовувати даний образ тільки для цього додатка, ви можете вказати це за допомогою ENTRYPOINT, і кожний раз, після створення контейнера з образу, ваш 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

- Команда ***ENV*** використовується для установки змінних середовища (однієї або багатьох). Дані змінні виглядають наступним чином «*ключ = значення*» і вони доступні всередині контейнера скриптів і різних додатків. Даний функціонал Докера, по суті, дуже сильно збільшує гнучкість щодо різних сценаріїв запуску додатків.

  Синтаксис команди: ***ENV %ключ% %значення%***

  ENV BASH /bin/bash

- Команда ***EXPOSE*** використовується 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 Докерфайла. Вона визначає базовий образ для початку процесу побудови контейнера. Це може бути будь-який образ, в тому числі і створені вами до цього. Якщо вказаний вами образ не знайдений на хості, Докер спробує знайти і завантажити його. **Ця команда в Докерфайлі завжди повинна бути вказана першою**.

  Синтаксис команди: ***FROM %назва_образу% 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 при написанні Докерфайла. Вона бере команду як аргумент і запускає її з образу. На відміну від CMD дана команда використовується для побудови образу (можна запустити кілька RUN поспіль, на відміну від CMD).

  Синтаксис команди: RUN % імя_команди%

  RUN yum install -y wget
- Команда ***USER*** - використовується для установки *UID* або імені користувача, яке буде використовуватися в контейнері.

  Синтаксис команди: USER %ID_користувача%

  USER 751
- Команда ***VOLUME*** використовується для організації доступу вашого контейнера до директорії на хості (те ж 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 створення свого власного образу для встановлення Mongodb

MongoDB - найбільш популярна нереляційна база даних.

Створимо порожній файл і відкриємо його за допомогою редактора *nano*:

nano Dockerfile

Надалі ми можемо вказати коментарями для чого даний Докерфайл буде використовуватися (це не обов'язково), але може бути корисно в подальшому. Про всяк випадок нагадаю - всі коментарі починаються з символу #.

```
########
# Dockerfile to build MongoDB container images
# Based on Ubuntu
########
```

Далі, вкажемо базовий образ: ***FROM ubuntu***

Після чого оновимо репозиторії та встановимо *gnupg2 (*вільна програма для шифрування інформації і створення електронних цифрових підписів*)*:

RUN apt-get update && apt-get install -y gnupg2

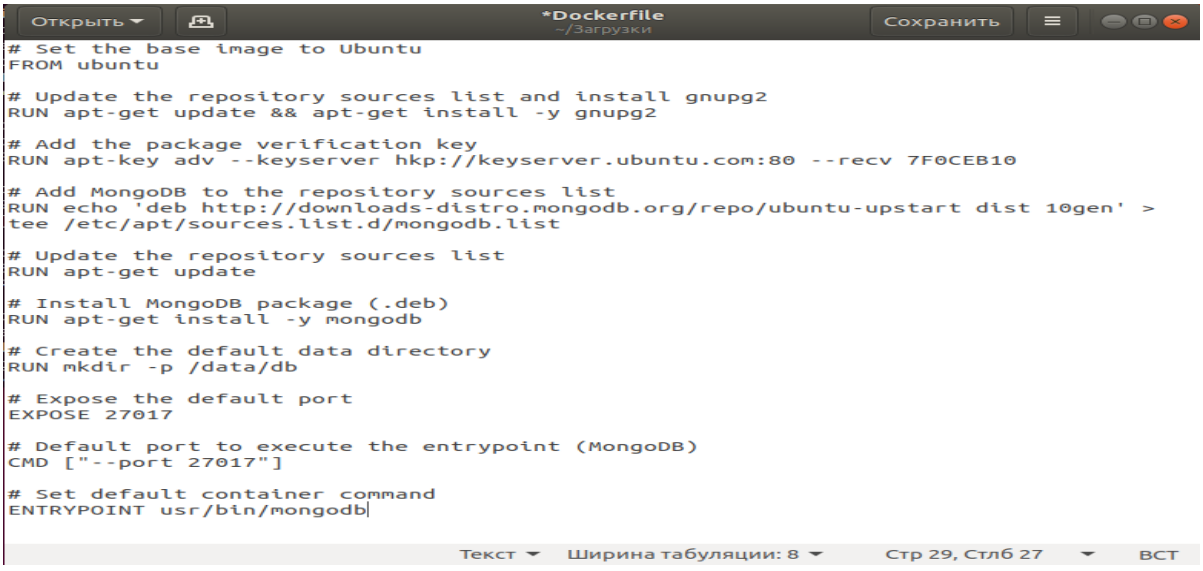
Після вкажемо команди і аргументи для скачування *MongoDB* (установку проводимо відповідно до плану на офіційному сайті):

```
RUN apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10
RUN echo 'deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart dist 10gen' > tee /etc/apt/sources.list.d/mongodb.list
RUN apt-get update
RUN apt-get install -y mongodb
RUN mkdir -p /data/db
```

Після чого вкажемо дефолтний порт для *MongoDB*: *EXPOSE 27017*

CMD [«--port 27017»]
ENTRYPOINT usr/bin/mongod

Ось як повинен виглядати у вас фінальний файл – перевірте, а потім можна зберегти зміни і закрити файл:



#########
 # Dockerfile to build MongoDB container images
 # Based on Ubuntu
#########
 # Set the base image to Ubuntu
FROM ubuntu
 # Update the repository sources list and install gnupg2
RUN apt-get update && apt-get install -y gnupg2
 # Add the package verification key
RUN apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10
 # Add MongoDB to the repository sources list
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
RUN apt-get update
 # Install MongoDB package (.deb)
RUN apt-get install -y mongodb
 # Create the default data directory
RUN mkdir -p /data/db
 # Expose the default port
EXPOSE 27017
 # Default port to execute the entrypoint (MongoDB)
CMD ["--port 27017"]
 # Set default container command
ENTRYPOINT usr/bin/mongodb

Запуск контейнера Docker

Створити наш перший *MongoDB* образ за допомогою Docker!

sudo docker build -t user_mongodb .

-t та ім'я тут використовується для присвоювання тега образу.

Для виведення всіх можливих ключів введіть ***sudo docker build -help***

А *точка в кінці* означає що Докерфайл знаходиться в тій же категорії, з якої виконується команда. *Запускаємо наш новий MongoDB* в контейнері!

sudo docker run -name UserMongoDB -t -i user_mongodb

Ключ ***-name*** використовується для *присвоєння простого імені контейнеру*, в іншому випадку це буде досить довга цифро-буквена комбінація. Після запуску контейнера для того, щоб повернутися в систему хоста натисніть ***CTRL+P,*** а потім ***CTRL+Q***.

Установка Docker Compose в Ubuntu 18.04

Docker - це інструмент для автоматизації розгортання додатків Linux всередині контейнерів програмного забезпечення, але для використання всіх його можливостей

необхідно, щоб кожний компонент додатка запускався у своєму власному контейнері. Для великих програм з великою кількістю компонентів, організація спільних - запуску, комунікації та зупинки всіх контейнерів може швидко стати дуже непростим і заплутаним завданням.

Спільнота Docker запропонувало популярне рішення, яке називається ***Fig*** і дозволяє вам використовувати єдиний файл з розширенням .***YAML або .YML*** для організації спільної роботи всіх ваших контейнерів і конфігурацій. Воно стало настільки популярним, що команда Docker вирішила створити Docker Compose на базі вихідного коду *Fig,* який в даний є застарілим інструментом і не підтримується.

Docker Compose спрощує організацію процесів контейнерів Docker, включаючи запуск, зупинку і настройку зв'язків і томів всередині контейнера. Це утиліта, яка полегшує збірку і запуск системи, що складається з декількох контейнерів, пов'язаних між собою.

1. ***Встановимо останню версію Docker Compose для управління додатками з декількома контейнерами.***

Можна встановити *Docker Compose* з офіційних репозиторіїв Ubuntu, але там не представлені найостанніші версії, тому ми будемо встановлювати *Docker Compose* зі сховищ *Docker на GitHub*. Команда нижче трохи відрізняється від команди, яку ви знайдете на сторінці Releases. Завдяки використанню прапора «***-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 необхідності оновимо її за допомогою наступної команди:

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

Налаштуємо дозволи: **sudo chmod +x /usr/local/bin/docker-compose**

Перевіримо чи установка пройшла успішно за допомогою перевірки версії:

docker-compose --version

В результаті повинна бути виведена встановлена нами версія:
Output
docker-compose version 1.25.5, build 8a1c60f6

Після встановлення Docker Compose можемо запустити приклад «Hello World».

2). ***Запуск контейнера за допомогою Docker Compose***

У загальнодоступному реєстрі *Docker, Docker Hu*b, міститься образ *Hello World,* який використовується 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***

Помістіть у файл наступні дані, збережіть його і закрийте текстовий редактор:

docker-compose.yml
my-test:
image: hello-world
Перший рядок файлу *YAML* використовується в якості частини імені контейнера. Другий рядок вказує, який образ використовується для створення контейнера.

При запуску команди ***docker-compose up*** вона буде шукати локальний образ за вказаним іменем, тобто *hello-world*. Після цього можна зберегти і закрити файл.

Ми можемо вручну переглянути образи в нашій системі за допомогою команди *docker images:* ***docker images***

Коли локальні образи відсутні, будуть відображені тільки заголовки стовпців:

Output
REPOSITORY TAG IMAGE ID CREATED SIZE
Далі, перебуваючи в директорії *«~/hello-world*», виконаємо наступну команду:

docker-compose up

При першому запуску команди, якщо локальний образ з ім'ям *hello-world* відсутній, *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, в 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 ми переглядаємо активні процеси, заголовки стовпців будуть з'являтися, але контейнер hello-world НЕ буде з'являтися в списку, оскільки він не запущений.

docker ps

Output

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

Переглянемо інформацію контейнера, яка нам 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

06069fd5ca23 hello-world "/hello" 35 minutes ago Exited (0) 35 minutes ago

PORTS NAMES

drunk_payne

Можемо отримати інформацію, яка нам буде потрібна для видалення контейнера, коли ми закінчимо працювати з ним.

3). ***Видалення способу (необов'язково)***

Щоб уникнути необов'язкового використання дискового простору, ми видалимо локальний образ. Для цього нам треба видалити всі контейнери, які містять образ, за допомогою команди ***docker rm***, після якої слідує CONTAINER ID або NAME. Нижче ми використовуємо CONTAINER ID з команди ***docker ps -a***, яку ми тільки що запустили. Не забувайте замінювати ідентифікатор на ідентифікатор вашого контейнера: ***docker rm 06069fd5ca23***

Після видалення всіх контейнерів, які містять образ, ми можемо видалити образ:

docker rmi hello-world

Створення наборів контейнерів для web-розробки
з використанням Docker Compose

Завдання:

Створити три контейнери для розроблення web-додатків:

- веб сервера ***Nginx*** з мінімальним налаштуванням для запуску проекту,
- сервера бази даних ***MySQL***,
- мови програмування для розробки web-додатків ***PHP7.0-fpm***.

В якості проекту будемо розгортати - беремо ***Magento2*** (система управління інтернет-магазинами).

Вмикаємо і запускаємо сервіс:

systemctl enable docker.service
systemctl start docker.service

Для створення однією командою нашої структури необхідно оновити *docker-compose.* Встановимо необхідні для нього компоненти:

sudo apt install epel-release
sudo apt install -y python-pip
sudo apt-get upgrade python

Створимо для цього проекту папку *mage* і перейдемо до неї:

mkdir /mage
cd /mage

Створюємо наступну структуру папок:

mkdir MySQL Nginx PHP

В папці *MySQL* будуть зберігатися бази. Зручно створювати резервні копії (*backup copy*) баз та їх переносити.
В папці *Nginx* будуть зберігатися лог-файли, файл конфігурації і наш проект.
В папку PHP будемо складати *Dockerfile* з налаштуваннями і ***php.ini.***
В корені (це папка */mage*) буде лежати файл *docker-compose.yml.*

Створюємо конфігураційний файл для *Nginx:*

cd /mage/Nginx/core
touch nginx.conf
nano nginx.conf

Можна використовувати будь-який інший редактор (наприклад, *mc*). Якщо його немає - можна встановити за допомогою:

sudo apt install nano

І додаємо до конфігураційного файлу *nginx.conf* наступний код:

```
server {
    listen 80;
    index index.php index.html index.htm;
    server_name magento2.dev;
    set $MAGE_ROOT /var/www/magento2;
    error_log  /var/log/nginx/error.log;
    access_log /var/log/nginx/access.log;
    root $MAGE_ROOT;
location ~*\.php$ {
    try_files $uri $uri/ /index.php last;
    fastcgi_split_path_info  (.+?\.php)(/.*)$;
    fastcgi_pass php:9000;
    fastcgi_index index.php;
    include fastcgi_params;
    fastcgi_param SCRIPT_FILENAME
$document_root$fastcgi_script_name;
    fastcgi_param PATH_INFO $fastcgi_path_info;
    }
location ~*.php/ {rewrite  (.*.php)/$1 last; }
}
```

Це мінімальна конфігурація для того, щоб все запрацювало.

У першому блоці описуємо який порт буде слухати, перелічуємо можливі *index* сторінки, називаємо та створюємо *alias* (псевдонім) для довгого шляху, де лежить *magento2*, пишемо – які потрібні логи і вказуємо де вони обов'язково повинні зберігаються, вказуємо папку там, де знаходиться *magento2* (у даному випадку наш *alias* $ MAGE_ROOT) .

У другому блоці прописуємо параметри *fastcgi*.

Третій блок потрібен для вирішення проблеми відображення, в проекті з'явилася пуста сторінка. З документації написано, що *magento2* вимагає реврайтинга (rewriting - обробка початкових текстових матеріалів з метою їх подальшого використання). (В інших проектах таких проблем не виникало).

В папці ***www*** створюємо каталог для нашого проекту:

cd /mage/Nginx/www
mkdir magento2

Скачуємо з офійційного сайту *magento2* https://magento.com/tech-resources/download

Та витягуємо з архіву у папку ***/mage/Nginx/www/magento2***
З налаштуваннями для Nginx ми закончили.

Переходимо до PHP:

Починаємо з *Dockerfile*

cd /mage/PHP
touch Dockerfile php.ini
nano Dockerfile

Збираємо самостійно:

```
FROM php:7.0-fpm
RUN apt-get update && apt-get install -y \
curl \
wget \
git \
libfreetype6-dev \
libjpeg62-turbo-dev \
libxslt-dev \
libicu-dev \
libmcrypt-dev \
libpng12-dev \
libxml2-dev \

&& docker-php-ext-install -j$(nproc) iconv mcrypt mbstring mysqli pdo_mysql zip \
&& docker-php-ext-configure gd --with-freetype-dir=/usr/include/ --with-jpeg-dir=/usr/include/ \
&& docker-php-ext-install -j$(nproc) gd

RUN docker-php-ext-configure intl
RUN docker-php-ext-install intl
RUN docker-php-ext-install xsl
RUN docker-php-ext-install soap
RUN curl -sS https://getcomposer.org/installer | php -- --install-dir=/usr/local/bin --filename=composer
ADD php.ini /usr/local/etc/php/conf.d/40-custom.ini
WORKDIR /var/www/magento2
CMD ["php-fpm"]
```

Налаштуємо *php.ini:* *nano php.ini*

```
memory_limit = 2G
always_populate_raw_post_data = -1
cgi.fix_pathinfo = 1
fastcgi_split_path_info = 1
max_execution_time = 18000
flag session.auto_start = off
zlib.output_compression = on
```

suhosin.session.cryptua = off
display_errors = Off

Налаштування PHP виконано.

Далі створюємо файл *docker-compose*, який нам усі складові збереже в одній зручній системі: ***cd /mage***
touch docker-compose.yml
nano docker-compose.yml

Розпишемо які сервіси і з якими налаштуваннями повинні запуститися:

```
# Пропишемо версію
version: '3.3'
# Перелічимо сервіси
services:
   nginx:
# Пропишимо який образ ми хочемо використати
      image: nginx:latest
# Дамо назву контейнеру
      container_name: nginx
# Перекидання портів
      ports:
         - "80:80"
         - "443: 443"
# Перекидання папок
      volumes:
         - ./Nginx/core:/etc/nginx/conf.d
         - ./Nginx/www:/var/www/
         - ./Nginx/Logs:/var/log/nginx/
         - ./Nginx/html:/usr/share/nginx/html/
# Вкажемо залежності
      links:
         - php
   mysql:
      image: mysql:latest
      ports:
         - "3306: 3306"
# Дамо назву контейнеру
      container_name: mysql
# Пропишемо налаштування, замість mypassword пропонується прописати більш складний пароль, який належить root
      environment:
         - MYSQL_ROOT_PASSWORD=mypassword
         - MYSQL_DATABASE=magento2
         - MYSQL_USER=magento2
         - MYSQL_PASSWORD=magento2
      volumes:
```

```
      - ./MySQL:/var/lib/mysql
  php:
# Будуємо з підтримкою dockerfile, вказавши директорію, де він знаходиться
    build: ./PHP
    container_name: php-fpm

    volumes:
      - ./Nginx/www:/var/www

    links:
      - mysql

  phpmyadmin:
    image: phpmyadmin/phpmyadmin

    container_name: phpmyadmin

    ports:
      - 8090: 80

    links:
      - mysql:db
```

На екрані з’являться рядки по ходу встановлення.

Після встановлення в папці *MySQL* створяться багато файлів і папок, з яких буде *Magento2*, а в папці Nginx/Logs з'являється 2 логи.

Відкривши браузер і набравши в ньому *localhost*, ви обов'язково побачите запрошення до установки *Magento2*.

Якщо щось у вас не вийшло, пропонується список рішень для усунення проблем:

1) Версія *docker-compose* - файлу не підійшла, тож потрібно поправити «версію: '3.3'», яка саме версія вам потрібна, перегляньте за посиланням:
   https://docs.docker.com/compose/compose-file/
2) Все нормально запустилось, але браузер відкриває чисту сторінку, без єдиної помилки - допоможе рядок в *nginx.conf:*

   "location ~ .php/ { rewrite (.*.php)/ $1 last; }"*
3) Якщо після встановлення самої *Magento2* (у браузері) у вас не прорисовуються фрейми та все виглядає як текстовий варіант сайту, вам потрібно зробити наступне:

3.1. Зайти в *SQL* через *phpmyadmin localhost:8090*, логін *root*, пароль *mypassword*, вибрати базу *magento2* і ввести *sql* запит:

insert into core_config_data (config_id, scope, scope_id, path, value) values (null, 'default', 0, 'dev/static/sign', 0)

3.2. Підключитися до контейнеру с *PHP (php-fpm)* і набрати:

php bin/magento cache:clean config

php bin/magento setup:static-content:deploy

Цей контейнер повинен перечитати і все перевірити. Після цього все повинно коректно відображатися.

Підготувати звіт

1. Описати хід виконання поставлених завдань, надаючи покроковий знімок екрану (*screenshot*).
2. Висновки по роботі.

Контрольні питання

1. Що таке Docker Compose?
2. Що таке Dockerfile?
3. Які вам відомі команди для роботи з Dockerfile?
4. У чому полягає алгоритм створення проекту для розроблення web-застосування?

КОМП'ЮТЕРНИЙ ПРАКТИКУМ 9.2

Створення проекту, що складається з контейтерів Django+PostgreSQL, з використанням Docker Compose та Dockerfile

Мета роботи – ознайомитися та набути навичок:

- написання скрипта **Dockerfile** для створення контейнеру;
- встановлення **Docker Compose в Ubuntu 18.04**;
- створення контейнерів (сервісів): **Django+PostgreSQL** для розроблення додатку.

Теоретичні відомості

Для створення контейнерів необхідно вміти працювати з **Docker Compose** та **Dockerfile.**

Dockerfile та синтаксис для їх створення

Dockerfile - скрипт, який дозволяє автоматизувати процес побудови контейнерів шляхом виконання відповідних команд (дій) в *base* образі для формування нового образу.

Усі подібні файли починаються з позначення ***FROM*** так як і процес побудови нового контейнера, далі йдуть різні методи, команди, аргументи або умови, після застосування яких створиться Docker контейнер.

Розглянемо синтаксис *Dockerfile*. В Докер файлах міститься два типи основних блоків - *коментарі та команди з аргументами*. Причому для всіх команд передбачається певний порядок.

Нижче наведено типовий приклад синтаксису, де перший рядок є коментарем, а другий - командою.

Print «Hello from User!»

RUN echo «Hello from User!!»

Розглянемо усі можливі команди. Усі команди в Докерфайлах прийнято вказувати великими літерами - наприклад **RUN**, **CMD** і т.д.

- Команда ***ADD*** - бере два аргументи, шлях звідки 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, використовується для виконання певних програм, але, на відміну від RUN дана команда зазвичай застосовується для запуску/ініціації додатків або команд вже після їх установки за допомогою RUN в момент побудови контейнера.

*Синтаксис команди: **CMD %додаток% «аргумент», «аргумент», ..***

CMD «echo» «Hello from User!»

- Команда ***ENTRYPOINT*** встановлює конкретний додаток за замовчуванням, який використовується кожний раз в момент побудови контейнера за допомогою образу. Наприклад, якщо ви встановили певний додаток всередині образу і ви збираєтеся використовувати даний образ тільки для цього додатка, ви можете вказати це за допомогою ENTRYPOINT, і кожний раз, після створення контейнера з образу, ваш 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 User!»***
ENTRYPOINT echo

- Команда ***ENV*** використовується для установки змінних середовища (однієї або багатьох). Дані змінні виглядають наступним чином «*ключ = значення*» і вони доступні всередині контейнера скриптів і різних додатків. Даний функціонал Докера, по суті, дуже сильно збільшує гнучкість щодо різних сценаріїв запуску додатків.

Синтаксис команди: ***ENV %ключ% %значення%***

ENV BASH /bin/bash

- Команда ***EXPOSE*** використовується 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 Докерфайла. Вона визначає базовий образ для початку процесу побудови контейнера. Це може бути будь-який образ, в тому числі і створені вами до цього. Якщо вказаний вами образ не знайдений на хості, Докер спробує знайти і завантажити його. **Ця команда в Докерфайлі завжди повинна бути вказана першою**.

Синтаксис команди: ***FROM %назва_образу% 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 при написанні Докерфайла. Вона бере команду як аргумент і запускає її з образу. На відміну від CMD дана команда використовується для побудови образу (можна запустити кілька RUN поспіль, на відміну від CMD).

Синтаксис команди: RUN % імя_команди% *RUN yum install -y wget*

- Команда ***USER*** - використовується для установки *UID* або імені користувача, яке буде використовуватися в контейнері.

Синтаксис команди: USER %ID_користувача% *USER 751*

- Команда ***VOLUME*** використовується для організації доступу вашого контейнера до директорії на хості (те ж 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 створення свого власного образу для встановлення Mongodb

MongoDB - найбільш популярна нереляційна база даних.

Створимо порожній файл і відкриємо його за допомогою редактора *nano*:

nano Dockerfile

Надалі ми можемо вказати коментарями для чого даний Докерфайл буде використовуватися (це не обов'язково), але може бути корисно в подальшому. Про всяк випадок нагадаю - всі коментарі починаються з символу #.

########
Dockerfile to build MongoDB container images
Based on Ubuntu
########

Далі, вкажемо базовий образ:

FROM ubuntu

Після чого оновимо репозиторії та встановимо *gnupg2* (вільна програма для шифрування інформації і створення електронних цифрових підписів):

RUN apt-get update && apt-get install -y gnupg2

Після вкажемо команди і аргументи для скачування *MongoDB* (установку проводимо відповідно до плану на офіційному сайті):

RUN apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10

RUN echo 'deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart dist 10gen' > tee /etc/apt/sources.list.d/mongodb.list

RUN apt-get update

RUN apt-get install -y mongodb

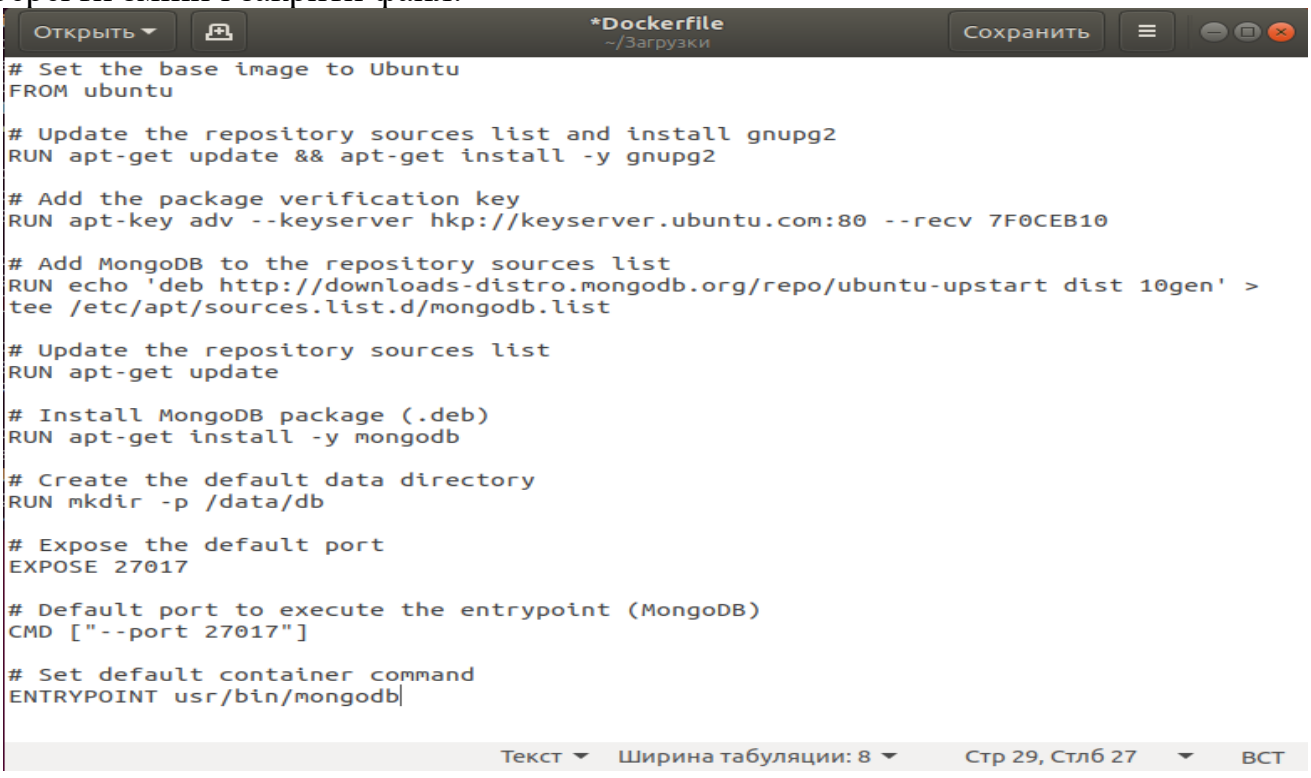
RUN mkdir -p /data/db

Після чого вкажемо дефолтний порт для *MongoDB*: *EXPOSE 27017*

CMD [«--port 27017»]

ENTRYPOINT usr/bin/mongod

Ось як повинен виглядати у вас фінальний файл – перевірте, а потім можна зберегти зміни і закрити файл:



```
#########
# Dockerfile to build MongoDB container images
# Based on Ubuntu
#########
```
 # Set the base image to Ubuntu
FROM ubuntu
 # Update the repository sources list and install gnupg2
RUN apt-get update && apt-get install -y gnupg2
 # Add the package verification key
RUN apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10
 # Add MongoDB to the repository sources list
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
RUN apt-get update
 # Install MongoDB package (.deb)
RUN apt-get install -y mongodb
 # Create the default data directory
RUN mkdir -p /data/db
 # Expose the default port
EXPOSE 27017
 # Default port to execute the entrypoint (MongoDB)
CMD ["--port 27017"]
 # Set default container command
ENTRYPOINT usr/bin/mongodb

Запуск контейнера Docker

Створити наш перший *MongoDB* образ за допомогою Docker!

sudo docker build -t user_mongodb .

-t та ім'я тут використовується для присвоювання тега образу.

Для виведення всіх можливих ключів введіть ***sudo docker build -help***

А *точка в кінці* означає що Докерфайл знаходиться в тій же категорії, з якої виконується команда.

Запускаємо наш новий MongoDB в контейнері!

sudo docker run -name UserMongoDB -t -i user_mongodb

Ключ ***-name*** використовується для *присвоєння простого імені контейнеру*, в іншому випадку це буде досить довга цифро-буквена комбінація. Після запуску контейнера для того, щоб повернутися в систему хоста натисніть ***CTRL+P,*** а потім ***CTRL+Q***.

Установка Docker Compose в Ubuntu 18.04

Docker - це інструмент для автоматизації розгортання додатків Linux всередині контейнерів програмного забезпечення, але для використання всіх його можливостей необхідно, щоб кожний компонент додатка запускався у своєму власному контейнері. Для великих програм з великою кількістю компонентів, організація спільних - запуску, комунікації та зупинки всіх контейнерів може швидко стати дуже непростим і заплутаним завданням.

Спільнота Docker запропонувало популярне рішення, яке називається ***Fig*** і дозволяє вам використовувати єдиний файл з розширенням ***.YAML або .YML*** для організації спільної роботи всіх ваших контейнерів і конфігурацій. Воно стало настільки популярним, що команда Docker вирішила створити Docker Compose на базі вихідного коду *Fig,* який в даний є застарілим інструментом і не підтримується.

Docker Compose спрощує організацію процесів контейнерів Docker, включаючи запуск, зупинку і настройку зв'язків і томів всередині контейнера. Це утиліта, яка полегшує збірку і запуск системи, що складається з декількох контейнерів, пов'язаних між собою.

1. ***Встановимо останню версію Docker Compose для управління додатками з декількома контейнерами.***

Можна встановити *Docker Compose* з офіційних репозиторіїв Ubuntu, але там не представлені найостанніші версії, тому ми будемо встановлювати *Docker Compose* зі сховищ *Docker на GitHub*. Команда нижче трохи відрізняється від команди, яку ви знайдете на сторінці Releases. Завдяки використанню прапора «***-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 необхідності оновимо її за допомогою наступної команди:

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

Налаштуємо дозволи:

sudo chmod +x /usr/local/bin/docker-compose

Перевіримо чи установка пройшла успішно за допомогою перевірки версії:

docker-compose --version

В результаті повинна бути виведена встановлена нами версія:

Output
docker-compose version 1.25.5, build 8a1c60f6

Після встановлення Docker Compose можемо запустити приклад «Hello World».

2). *Запуск контейнера за допомогою Docker Compose*

У загальнодоступному реєстрі *Docker, Docker Hu*b, міститься образ *Hello World*, який використовується 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

Помістіть у файл наступні дані, збережіть його і закрийте текстовий редактор:

docker-compose.yml

my-test:

image: hello-world

Перший рядок файлу *YAML* використовується в якості частини імені контейнера. Другий рядок вказує, який образ використовується для створення контейнера.

При запуску команди ***docker-compose up*** вона буде шукати локальний образ за вказаним іменем, тобто *hello-world*. Після цього можна зберегти і закрити файл.

Ми можемо вручну переглянути образи в нашій системі за допомогою команди *docker images:* ***docker images***

Коли локальні образи відсутні, будуть відображені тільки заголовки стовпців:
Output
REPOSITORY TAG IMAGE ID CREATED SIZE

Далі, перебуваючи в директорії *«~/hello-world*», виконаємо наступну команду:
docker-compose up

При першому запуску команди, якщо локальний образ з ім'ям *hello-world* відсутній, *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, в 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 ми переглядаємо активні процеси, заголовки стовпців будуть з'являтися, але контейнер hello-world НЕ буде з'являтися в списку, оскільки він не запущений.

docker ps
Output
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

Переглянемо інформацію контейнера, яка нам 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

06069fd5ca23 hello-world "/hello" 35 minutes ago Exited (0) 35 minutes ago

PORTS NAMES

drunk_payne

Можемо отримати інформацію, яка нам буде потрібна для видалення контейнера, коли ми закінчимо працювати з ним.

3). ***Видалення способу (необов'язково)***

Щоб уникнути необов'язкового використання дискового простору, ми видалимо локальний образ. Для цього нам треба видалити всі контейнери, які містять образ, за допомогою команди ***docker rm***, після якої слідує CONTAINER ID або NAME. Нижче ми використовуємо CONTAINER ID з команди ***docker ps -a***, яку ми тільки що запустили. Не забувайте замінювати ідентифікатор на ідентифікатор вашого контейнера: ***docker rm 06069fd5ca23***

Після видалення всіх контейнерів, які містять образ, ми можемо видалити образ:

docker rmi hello-world

Створення наборів контейнерів для web-розробки з використанням Docker Compose

Завдання:

Створити два контейнери для розроблення web-додатків:

- повнофункціонального серверного веб-фреймворка ***Django***, написаного на Python,
- вільної об'єктно-реляційної системи управління базами даних ***PostgreSQL***.

Визначення компонентів проекту

Перед початком повинен бути встановлений *Docker Compose*. Для цього проекту потрібно створити *Dockerfile*, файл залежностей *Python* та файл *docker-compose.yml*. (Для цього файлу можна використовувати розширення *.yml* або *.yaml*.)

1. Створіть порожній каталог для проекту.

Доцільно назвати каталог таким, що легко запам’ятовується. Цей каталог є контекстом для вашого образу додатка. Каталог повинен містити ресурси лише для створення цього образу.

2. Створіть новий файл під назвою Dockerfile у своєму проекті.

Dockerfile визначає вміст образу програми за допомогою однієї або декількох команд побудови, які налаштовують цей образ. Після побудови ви можете запустити образ в контейнер. Для отримання додаткової інформації про Dockerfile див. Посібник користувача Docker (https://docs.docker.com/get-started/) та посилання на Dockerfile (https://docs.docker.com/engine/reference/builder/).

3. *Додайте наступний вміст у файл Dockerfile.*

```
FROM python:3
ENV PYTHONUNBUFFERED 1
RUN mkdir /code
WORKDIR /code
COPY requirements.txt /code/
RUN pip install -r requirements.txt
COPY . /code/
```

Цей Dockerfile починається із батьківського образу *Python 3*. Батьківський образ модифікується шляхом додавання нового каталогу коду. Батьківський образ додатково модифікується шляхом встановлення вимоги *Python*, які визначені у файлі вимог *requirements.txt*.

4. Збережіть та закрийте *Dockerfile*.

5. Створіть файл вимог *requirements.txt* у вашому каталозі проекту.

Цей файл використовується командою ***RUN pip install -r requirements.txt*** у вашому *Dockerfile.*

6. Додайте необхідне програмне забезпечення у файл.

Django>=2.0,<3.0

psycopg2>=2.7,<3.0

7.*Збережіть та закрийте файл вимог requirements.txt.*

8. *Створіть файл з назвою docker-compose.yml* у вашому каталозі проекту.

Файл *Docker-compose.yml* описує сервіси, які створюють ваш додаток.

У цьому прикладі ці сервіси - це ***веб-сервер та база даних***.

Файл *compose* також описує які образи Докера ці сервіси використовують, як вони зв'язуються між собою, і які томи, які вони можуть знадобитися, встановлені всередині контейнерів. Нарешті, файл *docker-compose.yml* описує, які порти необхідні цим службам.

9. *Додайте у файл наступну конфігурацію*.

```
version: '3'
services:
  db:
    image: postgres
    environment:
      - POSTGRES_DB=postgres
      - POSTGRES_USER=postgres
```

```
    - POSTGRES_PASSWORD=postgres
  web:
   build: .
   command: python manage.py runserver 0.0.0.0:8000
   volumes:
    - .:/code
   ports:
    - "8000:8000"
   depends_on:
    - db
```

Цей файл визначає два сервіси: db-сервіс та веб-сервіс.

10. *Збережіть і закрийте файл docker-compose.yml.*

Створити проект Django

На цьому етапі створюємо проект для запуску *Django*, будуючи образ з контексту збірки, визначеної в попередній процедурі.

1. *Перейдіть до кореневого каталогу вашого проекту*.

2. Створіть проект Django, виконуючи команду run для docker-compose наступним чином.

sudo docker-compose run web django-admin startproject composeexample

Команда вказує *Compose* запустити *django-admin startproject composeexample* в контейнері, використовуючи образ та конфігурацію веб-сервіса. Оскільки веб-образу поки ще не існує, *Compose* створює його з поточного каталогу, як вказано у збірці: рядок в *docker-compose.yml.*

Після того, як побудовано образ веб-сервіса, *Compose* запускає його та виконує команду *django-admin startproject* у контейнері. Ця команда вказує *Django* створити набір файлів і каталогів, що представляють проект *Django*.

3. Після завершення команди Docker-Compose, складіть список вмісту вашого проекту.

```
$ ls -l
drwxr-xr-x 2 root   root   composeexample
-rw-rw-r-- 1 user   user   docker-compose.yml
-rw-rw-r-- 1 user   user   Dockerfile
-rwxr-xr-x 1 root   root   manage.py
-rw-rw-r-- 1 user   user   requirements.txt
```

Оскільки *Linux* при роботі з *Docker* створені файли *django-admin* належать користувачу *root* (це відбувається тому, що контейнер працює від імені користувача *root),* тому треба змінити власника на нові файли:

sudo chown -R $USER:$USER

Якщо ви працюєте з *Docker* на *MacOS* або *Windows*, ви вже повинні мати право власності на всі файли, включаючи файли, створені *django-admin*. Перерахуйте файли лише для того, щоб це підтвердити.

```
$ ls -l
total 32
-rw-r--r--  1 user  staff  145 Feb 13 23:00 Dockerfile
drwxr-xr-x  6 user  staff  204 Feb 13 23:07 composeexample
-rw-r--r--  1 user  staff  159 Feb 13 23:02 docker-compose.yml
-rwxr-xr-x  1 user  staff  257 Feb 13 23:07 manage.py
-rw-r--r--  1 user  staff   16 Feb 13 23:01 requirements.txt
```

Підключіть базу даних

У цьому розділі налаштуємо підключення до бази даних для *Django*.

1. Відредагуйте файл *composeexample/settings.py* у своєму каталозі проектів.

2. Замініть *DATABASES = ...* на наступне:

```
# setting.py

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'postgres',
        'USER': 'postgres',
        'PASSWORD': 'postgres',
        'HOST': 'db',
        'PORT': 5432,
    }
}
```

Ці налаштування визначаються образом Postgres Docker, який вказаний у *docker-compose.yml*.

3. Збережіть і закрийте файл.

4. Запустіть команду ***docker-compose up*** *з каталогу верхнього рівня для вашого проекту.*

```
$ docker-compose up
djangosample_db_1 is up-to-date
Creating djangosample_web_1 ...
Creating djangosample_web_1 ... done
Attaching to djangosample_db_1, djangosample_web_1
db_1   | The files belonging to this database system will be owned by user "postgres".
db_1   | This user must also own the server process.
```

db_1 |
db_1 | The database cluster will be initialized with locale "en_US.utf8".
db_1 | The default database encoding has accordingly been set to "UTF8".
db_1 | The default text search configuration will be set to "english".

. . .

web_1 | May 30, 2017 - 21:44:49
web_1 | Django version 1.11.1, using settings 'composeexample.settings'
web_1 | Starting development server at http://0.0.0.0:8000/
web_1 | Quit the server with CONTROL-C.

На даний момент ваш додаток *Django* повинен працювати в порту 8000 на хості Docker. На робочому столі Docker для Mac та Docker Desktop для Windows перейдіть на веб-браузер http://localhost:8000, щоб побачити сторінку вітання *Django*.

Якщо ви використовуєте *Docker Machine*, тоді IP MACHINE_VM *docker-machine* повертає IP-адресу хоста *Docker*, до якої ви можете додати порт (<Docker-Host-IP>: 8000).

На цьому установка закінчена.

Підготувати звіт

1. Описати хід виконання поставлених завдань, надаючи покроковий знімок екрану (*screenshot*).
2. Висновки по роботі.

Контрольні питання

1. Що таке Docker Compose?
2. Що таке Dockerfile?
3. Які вам відомі команди для роботи з Dockerfile?
4. У чому полягає алгоритм створення проекту для розроблення web-застосування?

СПИСОК ЛІТЕРАТУРИ

1. Уорд Б. Внутреннее устройство Linux. Санкт-Петербург : Питер. 2016. 384 с.
2. Негус К., Каэн Ф. Ubuntu и Debian Linux для продвинутых: более 1000 незаменимых команд. Санкт-Петербург : Питер. 2014. 384 с.
3. Керриск М. Linux API. Исчерпывающее руководство. Санкт-Петербург : Питер, 2019. 1248 с.
4. Шотт У. Командная строка Linux. Полное руководство. Санкт-Петербург : Питер, 2016. 480 с.
5. Колисниченко Д. Linux от новичка к профессионалу. Санкт-Петербург : БХВ-Питер, 2016. 672 с.
6. Тейлор Д., Перри Б. Сценарии командной оболочки. Linux, OS X и Unix. – СПб: Питер. 2017. 448 с.
7. Колисниченко Д. Н. Руководство по командам и shell-программированию в Linux. СПб.: БХВ-Петербург. 2011. 288 с.
8. Волох С. Ubuntu Linux с нуля. Санкт-Петербург : БХВ-Питер. 2016. 400 с.
9. Бреснахэн К., Блум Р. Linux на практике. Санкт-Петербург : Питер. 2017. 384 с.
10. Дейтел, Дейтел, Чофнес. Операционые системы. Основы и принципы. Москва : Бином, 2016. 1024 с.
11. Танебаум Э., Бос Х. Современные операционные системы. Санкт-Петербург : Питер, 2019. 1120 с.
12. Моэт Э. Использование Docker. Москва : ДМК Пресс, 2017. 354 с.
13. Парминдер Сингх Кочер. Микросервисы и контейнеры Docker. Москва : ДМК Пресс, 2019. 240 с.
14. Роббинс А. Bash. Карманный справочник системного администратора. СпБ.: ООО "Альфа-книга". 2017. 152 с.