МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ДНІПРОВСЬКА ПОЛІТЕХНІКА»

І.М. Гаркуша

ОПЕРАЦІЙНІ СИСТЕМИ

Методичні рекомендації до виконання лабораторних робіт
для здобувачів ступеня бакалавра освітньо-професійних програм
«Інформаційні системи та технології»
спеціальності 126 Інформаційні системи та технології
та «Комп'ютерна інженерія»
спеціальності 123 Комп'ютерна інженерія

Дніпро
НТУ «ДП»
2024

Гаркуша І. М.

Операційні системи [Електронний ресурс] : методичні рекомендації до виконання лабораторних робіт для здобувачів ступеня бакалавра освітньо-професійних програм «Інформаційні системи та технології» спеціальності 126 Інформаційні системи та технології та «Комп'ютерна інженерія» спеціальності 123 Комп'ютерна інженерія / І. М. Гаркуша ; М-во освіти і науки України, Нац. техн. ун-т «Дніпровська політехніка». – Дніпро : НТУ «ДП», 2024. – 78 с.

Автор
І. М. Гаркуша, канд. техн. наук, доц.

Затверджено науково-методичними комісіями спеціальностей 126 Інформаційні системи та технології та 123 Комп'ютерна інженерія (протокол № 3 від 27.11.2024) за поданням кафедри інформаційних технологій та комп'ютерної інженерії (протокол № 7 від 25.11.2024).

Надані теоретичні відомості та практичні завдання, а також рекомендації щодо їх виконання. Поданий перелік рекомендованих джерел.

Орієнтовано на активізацію навчальної діяльності здобувачів ступеня бакалавра спеціальностей «Інформаційні системи та технології» та «Комп'ютерна інженерія», закріплення практичних навичок у засвоєнні дисципліни «Операційні системи».

Відповідальний за випуск завідувач кафедри інформаційних технологій та комп'ютерної інженерії В. В. Гнатушенко, д-р техн. наук, проф.

ЗМІСТ

ВСТУП 4

Робота №1. Встановлення та налаштування ОС в середовищі Oracle VM VirtualBox 5

Робота №2. Система допомоги в Unix-подібних ОС 14

Робота №3. Основні консольні команди Unix-подібних ОС 19

Робота №4. Командний процесор Bash 25

Робота №5. Microsoft Windows PowerShell 38

Робота №6. Створення процесів в GNU/Linux-сумісних ОС 58

Робота №7. Створення процесів в MS Windows 66

Критерії оцінювання виконання робіт 73

Перелік рекомендованих джерел 74

Додаток А. Вимоги до оформлення звітності 75

ВСТУП

Операційна система (ОС; operating system, OS) – це комплекс програм для управління апаратним забезпеченням. ОС виконує роль інтерфейсу між пристроями обчислювальної системи та прикладними програмами. ОС призначена для керування пристроями та обчислювальними процесами з урахуванням ефективного розподілу обчислювальних ресурсів, організації надійних обчислень та захисту даних користувачів. ОС дозволяє абстрагуватися від деталей реалізації та функціонування пристроїв, надаючи певний набір функцій.

Множина існуючих сучасних ОС для комп'ютерів досить широка і, наприклад, у галузі серверних технологій Internet, за оцінками аналітиків та статистики, лідером є ОС на базі ядра Linux – множина GNU/Linux-сумісних ОС. Також безліч серверних систем побудовано на базі різних Unix-подібних ОС, наприклад, вільної FreeBSD або на базі корпоративних комерційних систем, таких, як: HP-UX, IBM AIX, SCO UnixWare, SCO OpenServer, Oracle Solaris та ін. Лідером серед ОС для створення десктопних технологій клієнтських робочих місць є сімейство ОС компанії Microsoft – MS Windows.

Найбільш відомими в сучасному світі для серверів та робочих станцій є ОС: MS Windows, Apple macOS, Debian GNU/Linux, Ubuntu, Linux Mint, Fedora, Arch Linux, Manjaro Linux, OpenSUSE, Gentoo Linux, Slackware, Kali Linux, Red Hat Enterprise Linux, AlmaLinux, Oracle Linux, FreeBSD, OpenBSD, NetBSD, GhostBSD, QNX та інші.

Сьогодні очевидним є той факт, що знання лише однієї ОС значно звужує професійні межі діяльності майбутніх фахівців і тому є актуальним вивчення архітектур різних ОС та інструментів у їх складі.

Методичні рекомендації спрямовано на вивчення початкових прийомів роботи з GNU/Linux-сумісними ОС, а також передових технологій Microsoft з управління ОС Windows.

В рекомендаціях пропонується опанувати певні команди та можливості командного процесора Bash, який найчастіше зустрічається у складі Unix-подібних та GNU/Linux-сумісних ОС. Розглянуто використання потужного засобу керування в ОС Microsoft – Windows PowerShell, який прийшов на зміну застарілому командному процесору Microsoft cmd.exe. Також певні роботи присвячені елементам системного програмування в GNU/Linux-сумісних ОС та MS Windows, зокрема, питанням створення процесів.

Наприкінці представленого практикуму з ОС надано перелік рекомендованих джерел, які рекомендуються для підвищення рівня знань здобувачів. Для виконання практикуму здобувач крім теоретичної складової кожної роботи зобов'язаний мати знання, отримані в процесі прослуховування лекційного курсу.

Робота №1.
Встановлення та налаштування ОС
в середовищі Oracle VM VirtualBox

Мета роботи: набути навичок встановлення та налаштування GNU/Linux-сумісної ОС в середовищі Oracle VM VirtualBox (або в UTM для платформи macOS за потреби).

Теоретично-практична частина

Oracle VM VirtualBox (далі скорочено VB) – програмний продукт віртуалізації для операційних систем (ОС) Microsoft Windows, GNU/Linux-сумісних, FreeBSD, macOS / Mac OS X, Solaris / OpenSolaris. Офіційні ресурси:

https://www.virtualbox.org/ та https://www.oracle.com/ua/virtualization/virtualbox/

Програма була створена компанією InnoTek з використанням source-коду проекту QEMU. Перша публічно доступна версія VB з'явилася 15 січня 2007 року. У лютому 2008 року InnoTek був придбаний компанією Sun Microsystems, модель поширення VB при цьому не змінилася. У січні 2010 року Sun Microsystems була поглинена корпорацією Oracle. Продукт VB все також залишається відкритим під певними ліцензіями.

Детальну інформацію про VB можна переглянути в документації користувача. Наприклад, для версії 7.0.18 за посиланням:

https://download.virtualbox.org/virtualbox/7.0.18/UserManual.pdf

Програма доступна для завантаження за посиланнями:

https://www.virtualbox.org/wiki/Downloads

та

https://www.oracle.com/ua/virtualization/technologies/vm/downloads/virtualbox-downloads.html

Розглянемо приклад встановлення та використання VB.

Примітка 1.1: системні вимоги для встановлення та роботи з віртуальною машиною (VM):

– **мінімальна** версія рекомендованої основної ОС:
MS Windows 8.1, Mac OS X 10.15 (Catalina), GNU/Linux-сумісні (Ubuntu 18.04 LTS, або Debian GNU/Linux 10, або RHEL 7, або Fedora 35, або openSUSE Leap 15.3, або подібні);

– розрядність процесору комп’ютера – 64 біт (старі версії VB, наприклад, 5.2.4, ще підтримували встановлення на 32-бітні архітектури);

– об’єм оперативної пам’яті (ОП) на комп’ютері – більше 1 Гб (рекомендується 8 Гб або більше для комфортної роботи);

– вільне місце на жорсткому диску комп’ютера – як мінімум 30 Гб (для роботи під платформою MS Windows потрібне розміщення на дисках з файловою системою NTFS);

– в BIOS/UEFI повинна бути активована опція апаратної віртуалізації VT-x/AMD-V або подібна (якщо такої немає, то VM не буде вірно працювати або буде працювати дуже повільно);

– пряме з’єднання з глобальною мережею Internet (мережа буде потрібна для встановлення додаткових пакунків та оновлень в гостьових ОС під керуванням VM).

Для завантаження з сайту Oracle (посилання дано вище) необхідно обрати варіант Platform – наприклад, Windows installer, а також Oracle VM VirtualBox Extension Pack – ExtPack. Станом на 03.10.2024 була доступна стабільна версія 7.0.18 (файл VirtualBox-7.0.18-162988-Win.exe) та відповідний їй пакет розширення (файл: Oracle_VM_VirtualBox_Extension_Pack-7.0.18.vbox-extpack) з офіційного ресурсу Oracle за посиланням:

https://www.oracle.com/ua/virtualization/technologies/vm/downloads/virtualbox-downloads.html

Примітка 1.2: в роботі використаний VB стабільної версії 7.0.18. Якщо вона відсутня на офіційній сторінці, можна її завантажити з архіву за посиланням: https://download.virtualbox.org/virtualbox/7.0.18/. Якщо робота відбувається в більш новішій версії VB, то можливі відхилення в описі роботи.

Після встановлення та запуску програми, оберіть в головному вікні пункт меню *Файл\Tools\Extension Pack Manager* (*File\Tools\Extension Pack Manager*) для встановлення пакету розширення програми VB. Використовуйте кнопку *Install* та оберіть extpack-файл для встановлення пакету розширення (при встановленні розширення підтвердить ліцензію, виконавши прокручування тексту у вікні вниз та натисніть на кнопку *Я згідний* (*I Agree*) щоб здійснити встановлення). За потреби введіть пароль адміністратора головної ОС, під керуванням якої працює VB.

При виконанні лабораторної роботи в **дистанційному режимі**, завантажте підготовлену VM з ОС Debian GNU/Linux, яка розміщена за посиланням: http://surl.li/tvoauq (для доступу потрібен попередній вхід за посиланням https://portal.office.com/).

Для завантаження обирайте більш свіжу версію. Наприклад, станом на 30.12.2022 була остання підготовлена версія, яка розміщена в підкаталозі: **Debian11_6_Console_IceWM_edu_2022_v4**.

Примітка 1.3: якщо використовується в якості головної ОС macOS та обладнання на процесорах Apple Silicon M series й програма VB не працює, або працює некоректно, то можна скористатися іншою програмою віртуалізації Наприклад, свіжою версією з ресурсу https://www.virtualbox.org/ або UTM за посиланням: https://mac.getutm.app/. В роботі розглянута версія UTM 4.0.9, яка є поточною станом на 03.01.2023. В цьому випадку можна використати спеціально підготовлену VM з ОС Debian GNU/Linux, наприклад з підкаталогу:

For_macOS_UTM_Debian11_6_Console_IceWM_edu_2022_v4

Багатотомні архіви із запропонованими VM потрібно розпакувати, отримавши або ova-файл (для програми VB), або utm-каталог (для програми UTM).

При виконанні лабораторної роботи **як в дистанційному режимі, так і в комп'ютерному класі**, оберіть варіант для встановлення ОС з iso-образу, який знаходиться за адресою:

https://cdimage.debian.org/cdimage/archive/11.6.0/amd64/iso-cd/

Наприклад, debian-11.6.0-amd64-netinst.iso або подібний з іншою версією.

Після завантаження iso-образу встановлювача, у VB оберіть пункт меню *Машина\Створити* (*Machine\New*). В діалозі вказати назву машини, наприклад, Debian, обрати версію – *Debian (64-bit)*, вказати розташування завантаженого ISO-образу (*ISO Image*), вказати опцію *Skip Unattended Installation* та натиснути на кнопку *Next*. На наступному кроці вказати об'єм ОП, який буде виділений з ОП основної системи для функціонування VM – наприклад, якщо основна система дозволяє, то вкажіть об'єм *1024 МБ* та натисніть кнопку *Next*.

На наступному кроці залишити активним першу опцію та вказати розмір віртуального жорсткого диску. Наприклад, *15 Гб* та натиснути кнопку *Next*.

Завершити останній крок, підтверджуючи раніше обрані параметри.

Після цього у VB потрібно провести додаткові налаштування – натиснути кнопку *Налаштування* (*Settings*) або обрати пункт меню *Машина\Налаштування* (*Machine \ Settings...*).

Зліва в діалозі, обрати пункт *Система* (*System*). У списку *Порядок завантаження* (*Boot Order*), зняти опцію з пункту *Дискета* (*Floppy*), встановити опцію *Ввімкнути I/O APIC* (*Enable I/O APIC*), перейти на закладку *Процесор*

(*Processor*) та вказати кількість процесорів – 1 або 2 (якщо це можливо), встановити опцію *Ввімкнути PAE/NX* (*Enable PAE/NX*).

Зліва в діалозі, обрати пункт *Екран* (*Display*) та встановити об'єм відео пам'яті (*Video Memory*) у розмірі *32 Мб*.

Зліва в діалозі, обрати пункт *Пам'ять* (*Storage*), обрати *Контролер IDE*, натиснути праву кнопку миші, та обрати пункт *Del*. В *Контролері SATA* (*Controller: SATA*) натиснути на кнопку у вигляді CD-диску. В діалозі додавання диску натиснути на кнопку *Add*, обрати завантажений раніше iso-образ та натиснути на кнопку *Choose*. Переведіть курсор миші та натисніть на пункт *Контролер: SATA* (*Controller: SATA*). Встановіть справа опцію *Використати кеш головного B/B* (*Use Host I/O Cache*).

Зліва в діалозі, обрати пункт *Мережа* (*Network*) та із списку *Під'єднаний до* (*Attached to*), обрати варіант *NAT*.

Після цього натиснути на кнопку *OK*.

Для початку роботи з VM переконатися, що головна система має з'єднання з мережею Internet, натиснути на кнопку *Запустити* (*Start*), або обрати пункт меню *Машина\Запустити\Звичайний запуск* (*Machine\Start\ Normal Start*).

Примітка 1.4: якщо використовується середовище UTM для macOS, то для імпортування utm-віртуальної машини потрібно виконати наступні кроки:

– переконатися, що macOS має з'єднання з мережею Internet;

– обрати пункт *Create a New Virtual Machine*;

– обрати варіант *Emulate*;

– обрати варіант *Other*;

– натиснути на кнопку *Browse...* та вказати завантажений iso-образ з встановлювачем Debian GNU/Linux;

– натиснути на кнопку *Continue*;

– обрати архітектуру *x86_64*;

– вказати об'єм ОП для виконання VM – *1024 MB*;

– в полі *CPU Cores* замість *default* вказати 1 та натиснути на кнопку *Continue*;

– вказати об'єм віртуального жорсткого диску – *15 GB* та натиснути на кнопку *Continue*;

– на кроці *Share Directory* натиснути на кнопку *Continue*;

– на кроці *Summary* вказати в полі *Name* назву нової VM – *Debian* та натиснути на кнопку *Save*;

– обрати зі списку VM *Debian* та відкрити контекстне меню й обрати пункт меню *Edit*;

– в діалозі властивостей VM зліва, обрати пункт *QEMU*;

– зняти опцію *UEFI Boot*, встановити опцію *Use Hypervisor* та натиснути на кнопку *Save*;

– запустити VM *Debian* по кнопці з трикутником-стрілкою.

Встановити ОС Debian GNU/Linux – після завантаження з iso-образу, обрати у вікні VM пункт *Graphical install*.

В процесі встановлення обрати мову *English*. Обрати варіант розташування *other* (далі *Continue*), *Europe* (далі *Continue*), *Ukraine* (далі *Continue*). В налаштуваннях локалі залишити *United States (en_US.UTF-8)*. Налаштування клавіатури – залишити *American English*.

Після встановлення основних компонент, виконайте налаштування:

– Hostname: *debian*;
– Domain name: залиште пустим;
– Root password: *studstud*;
– Re-enter password to verify: *studstud*;
– Full name for the new user: *Stud*;
– Username for your account: *stud*;
– Choose a password for the new user: *studstud*;
– Re-enter password to verify: *studstud*;
– Partitioning method: *Guided – use entire disk*;
– Select disk to partition: *SCSI1 . . .*;
– Partitioning scheme: *All files in one partition*;
– Write the changes to disks: *Yes*;
– Scan extra installation media: *No*;
– Debian archive mirror country: *Ukraine*;
– Debian archive mirror: *deb.debian.org*;
– HTTP Proxy information: залиште пустим;
– Participate in the package usage survey: *No*;
– Choose software to install: залиште тільки опції *SSH Server* та *standard system utilities*;
– Install the GRUB boot loader to your primary drive: *Yes*;
– Device for boot loader installation: */dev/sda*.

Примітка 1.5: після встановлення та перезавантаження VM в середовищі UTM в macOS, натиснути на кнопку вимикання VM. Відкрити вікно властивостей VM та перетягнути мишею перший пункт *IDE Drive* за другий. Таким чином змінюється порядок звернення до диску при завантаженні. Після цього натиснути на кнопку *Save* та запустити знову VM.

Після перезавантаження VM здійснити логін під користувачем ***stud***. При введені паролю в терміналі консолі (***studstud***), він не відображається на екрані! Після введення паролю натисніть на клавіатурі клавішу Enter. Якщо після перезавантаження VM процес запуску на певному кроці зупиняється, то потрібно спробувати закрити вікно VM (обираючи у VB опцію *Power off the machine*) та повторити запуск VM по кнопці *Запустити* (*Start*). Приклад вікна

VM з встановленою та завантаженою ОС Debian GNU/Linux представлено на рис. 1.1.

Для припинення роботи VM, натисніть кнопку закриття вікна VM. В діалозі оберіть опцію або збереження стану машини або надсилання сигналу завершення (рис. 1.2). Не вимикайте машину пунктом примусового вимикання!

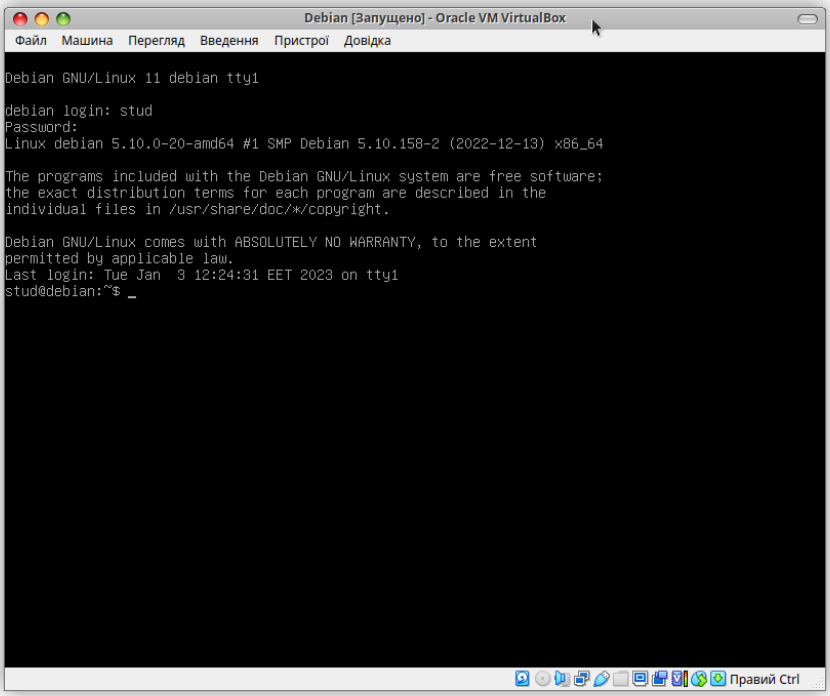


Рис. 1.1. Вікно VM з ОС Debian GNU/Linux у середовищі VB

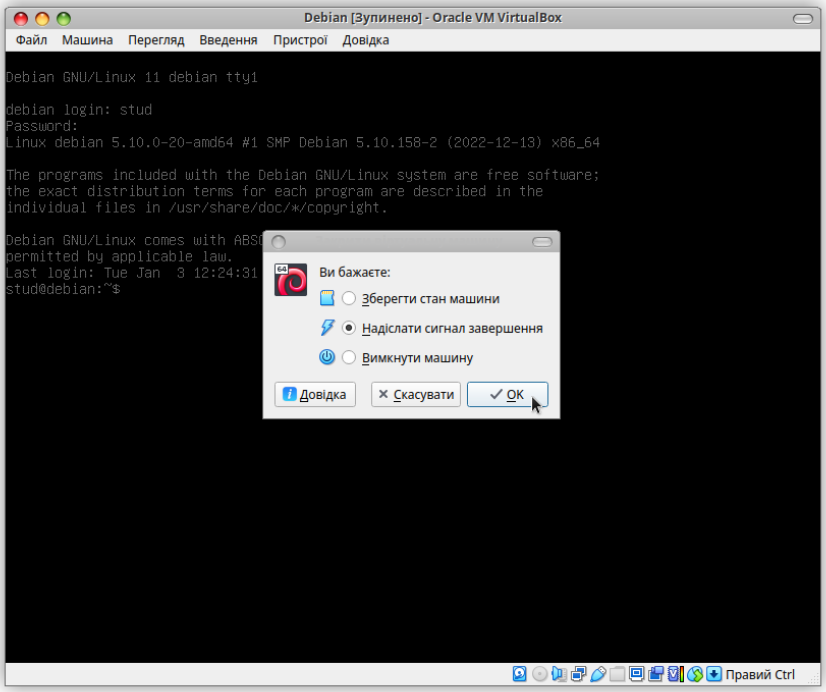


Рис. 1.2. Обрання варіанту надсилання сигналу завершення роботи VM у середовищі VB

Примітка 1.6: у разі встановлення з iso-образу Debian GNU/Linux в середовищі UTM в macOS, для вимикання VM, що працює, потрібно виконати наступні кроки в консолі:

– при виконаному вході в систему під користувачем *stud*, ввести команду *su* та натиснути на клавіатурі клавішу Enter;

– в рядку *Password:* ввести пароль *studstud* та натиснути на клавіатурі клавішу Enter;

– ввести команду */sbin/init 0* та натиснути на клавіатурі клавішу Enter – це призведе до негайного завершення роботи ОС у VM;

– закрити вікно VM.

При виконанні лабораторної роботи в **дистанційному режимі**, переконайтеся, що у VB вірно налаштований головний мережевий хост-адаптер. Для цього оберіть пункт меню *Файл\Tools\Network Manager* (*File\Tools\Network Manager*), та якщо в діалозі управління мережею (закладка *Host-only Networks*) буде відсутнє ім'я головного адаптеру, натисніть на кнопку *Create*. Після цього можна переходити до роботи з готовою VM (ova-файлом).

Для імпортування ova-файлу у VB скористайтеся пунктом меню *Файл \ Імпортувати образ віртуальної машини* (*File \ Import Appliance*) та в діалозі використайте відповідну кнопку для вказівки розташування розпакованого ova-файлу.

Примітка 1.7: для імпортування в програмі UTM в macOS використайте кнопку *Create a New Virtual Machine*, далі в діалозі оберіть посилання *Open...* та вкажіть розпакований utm-каталог.

Запустить імпортовану VM (рис. 1.3). У середовищі VB машина налаштована таким чином, що підтримує роботу через два мережеві інтерфейси (рис. 1.3): через перший VM буде мати змогу виходити до мережі Internet, через другий адаптер можна буде підключатися до VM з головної системи (через *ssh*-консоль, або через Web-браузер).

Примітка 1.8: у середовищі UTM в macOS VM має спеціально налаштований єдиний мережевий інтерфейс як для виходу в Internet, так і для звернення до VM з головної системи.

Імпортований варіант VM з ОС Debian GNU/Linux містить графічний інтерфейс користувача (Graphical User Interface, GUI) на базі віконного менеджера (Window Manager) IceWM, який за потреби викликається командою *startx* або *start-icewm* (рис. 1.4).

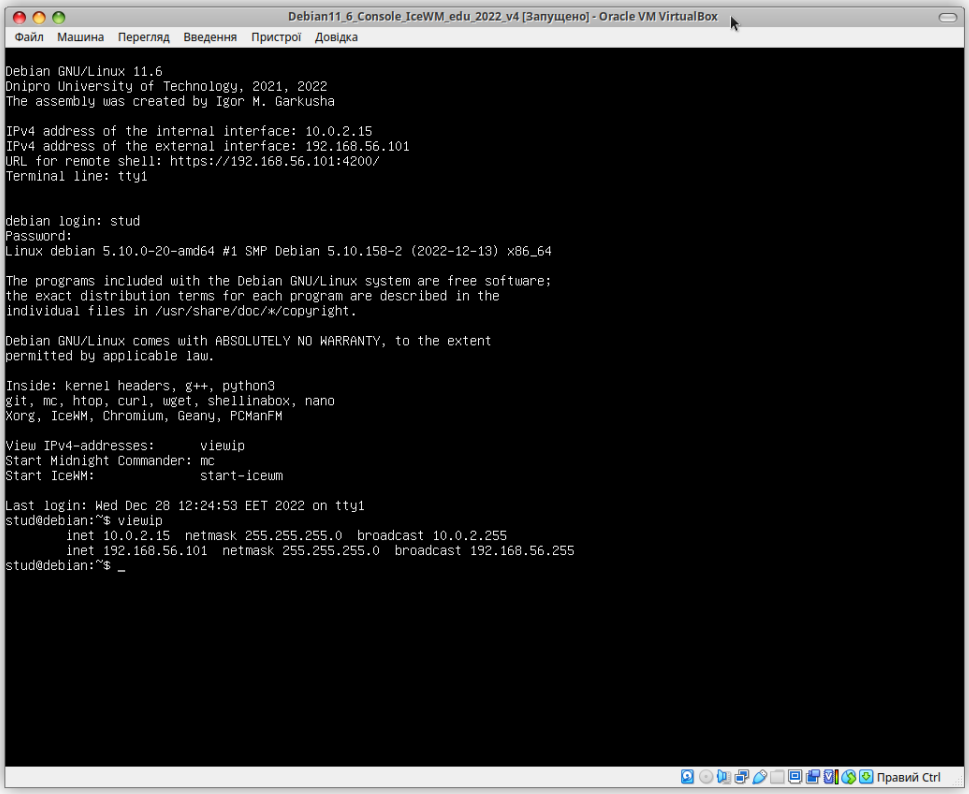


Рис. 1.3. Вікно VM з імпортованою та запущеною ОС Debian GNU/Linux в середовищі VB

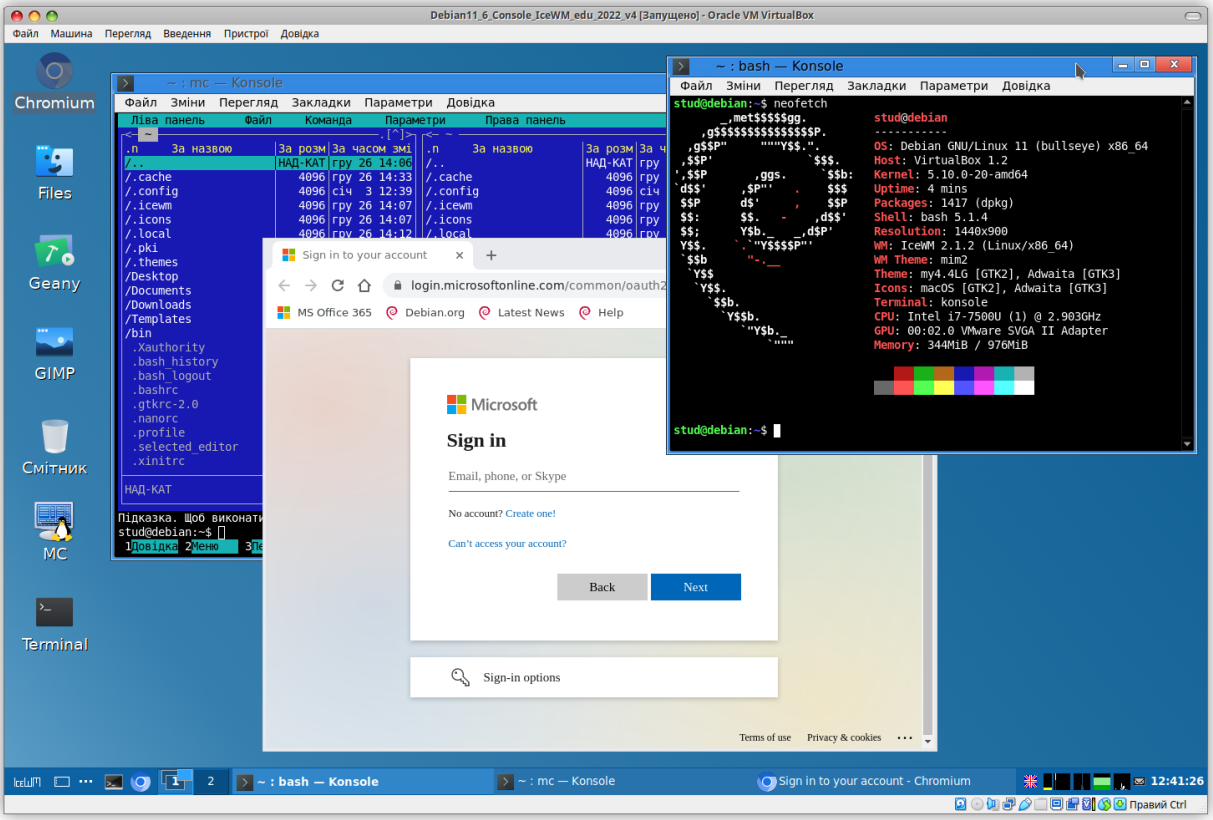


Рис. 1.4. GUI на базі віконного менеджера IceWM

При захисті роботи представити встановлені ОС в середовищі VB (або в UTM за потреби) та дати вірні відповіді на наступні питання.

1. Що таке віртуальна машина?
2. Яким чином у VB здійснюється управління чергою завантаження з носіїв (у разі використання UTM студент також повинен знати, як це відбувається та повинен також пояснити, як змінити порядок завантаження в UTM)?
3. Як здійснюється імпортування готової VM в різних середовищах (VB та UTM)?
4. Як зберігати стан віртуальної машини у VB?
5. Що являє собою ОС Debian GNU/Linux та що про неї відомо?
6. Які налаштування при встановленні ОС Debian GNU/Linux були здійснені?

Робота №2.
Система допомоги в Unix-подібних ОС

Мета роботи: вивчити основні можливості команди *man* по отриманню довідникової інформації в Unix-подібних ОС.

Теоретична частина

Для отримання детальної довідникової інформації з якої-небудь команди (програми) в Unix-подібній ОС, призначена команда *man* (скорочення від "manual"). У найпростішому вигляді її виклик виглядає наступним чином:

`man` *команда/функція/програма*

де *команда/функція/програма* – назва будь-якої команди, функції або програми ОС.

Якщо довідка по вказаній інформації встановлена в одному з розділів документації (як правило, це каталоги /usr/share/man, /usr/local/man, /usr/local/share/man, /usr/man, /usr/share/man та ін.), то користувачеві буде виданий результат.

За допомогою наступних команд користувач може завжди дізнатися, в яких каталогах файлової системи (ФС) розташована документація команди *man*:

```
man --path
```

або

```
manpath
```

Якщо інформація, що виводиться командою `man`, не поміщається у вікні консолі, то, як правило, її подальше виведення переривається і може бути продовжене натисканням клавіші пробілу. Вихід з програми man здійснюється клавішею *q* (скорочення від ***q****uit*).

Весь довідниковий матеріал розбитий на розділи, порядок і назви яких різні для різних версій ОС. У таблиці 2.1 перераховані традиційні розділи та їх назви для двох основних гілок Unix: BSD та System V. Часто в багатьох ОС ці розділи збігаються. Як правило, список розділів та їх номери, а також всю інформацію про команду man можна дізнатися увівши команду:

```
man man
```

При вивченні документації з використанням команди man, зустрічаються посилання на інші команди, програми, опис файлів конфігурацій, функції мови програмування С та розділи в наступному форматі: *команда(номер)*. Наприклад:

fs(5). В даному випадку це означає, що можливе отримання довідки по файловим системам в розділі 5 в такий спосіб:

man *номер_розділу команда/функція/програма*

Наприклад:

```
man 5 fs
```

Таблиця 2.1

Назва розділів довідки (доступні за командою: `man man`)

Зміст розділу	*BSD UNIX*	*UNIX System V*
Прикладні утиліти	1	1
Системні виклики	2	2
Бібліотечні функції	3	3
Спеціальні файли, драйвери пристроїв та апаратне забезпечення	4	7 або 4
Формати різних конфігураційних та системних файлів, а також протоколів	5	4 або 5
Ігри	6	6
Різні матеріали, наприклад, про типи файлових систем, визначення типів даних та т.ін.	7	5 або 7
Адміністративні утиліти	8	8
Kernel routines – підпрограми ядра (не стандартизований)	9	9

У більшості програм є ключі (параметри, опції), які можна вказати під час їх запуску. Так, вельми корисними ключами самої команди *man* є ключі *-f* та *-k*. Багатьом програмам з ключами відповідають готові командні файли (командні скрипти або просто скрипти). Так, наприклад, команді *man -f* відповідає програма *whatis*, команді *man -k* – програма *apropos* (в певних ОС *apropos* може посилатися на *whatis*).

Команда *whatis* або *man -f* здійснює пошук короткого опису в базі даних *whatis* по кожному із зазначених ключових слів та відображення одного рядка опису у вікні терміналу для кожної відповідності. Ця команда здійснює пошук лише слів, *які повністю збігаються*.

Якщо користувачеві необхідно *висвітлити всі сторінки* довідки, які містять вказане їм слово, наприклад, будь-яку команду, то для цих цілей в Unix-подібних ОС призначена команда *apropos* або *man -k*. Команда здійснює пошук перерахованих рядків в записах бази даних програми *whatis* та видачу результатів у вікно терміналу. Тобто, команда *apropos* здійснює *пошук рядків, а не окремих слів*.

Приклади використання:

```
whatis chown
whatis passwd
```

```
man -f chown passwd
apropos passwd
man -k passwd single splash
```

Графічним різновидом команди *man* є команда *xman* (за замовчуванням може бути відсутньою в системі). В Debian-подібних ОС (Debian, Ubuntu, Linux Mint та ін.) її можна встановити за допомогою команди:

```
sudo apt install x11-apps
```

Після встановлення в середовищі раніше імпортованої VM Debian, скористайтеся командою `startx`, запустіть графічне вікно консольного терміналу та введіть команду *xman*, після чого можна обрати *Manual Page*, а по пункту *Sections* обрати потрібний розділ довідки й перейти до потрібної команди (рис. 2.1) .

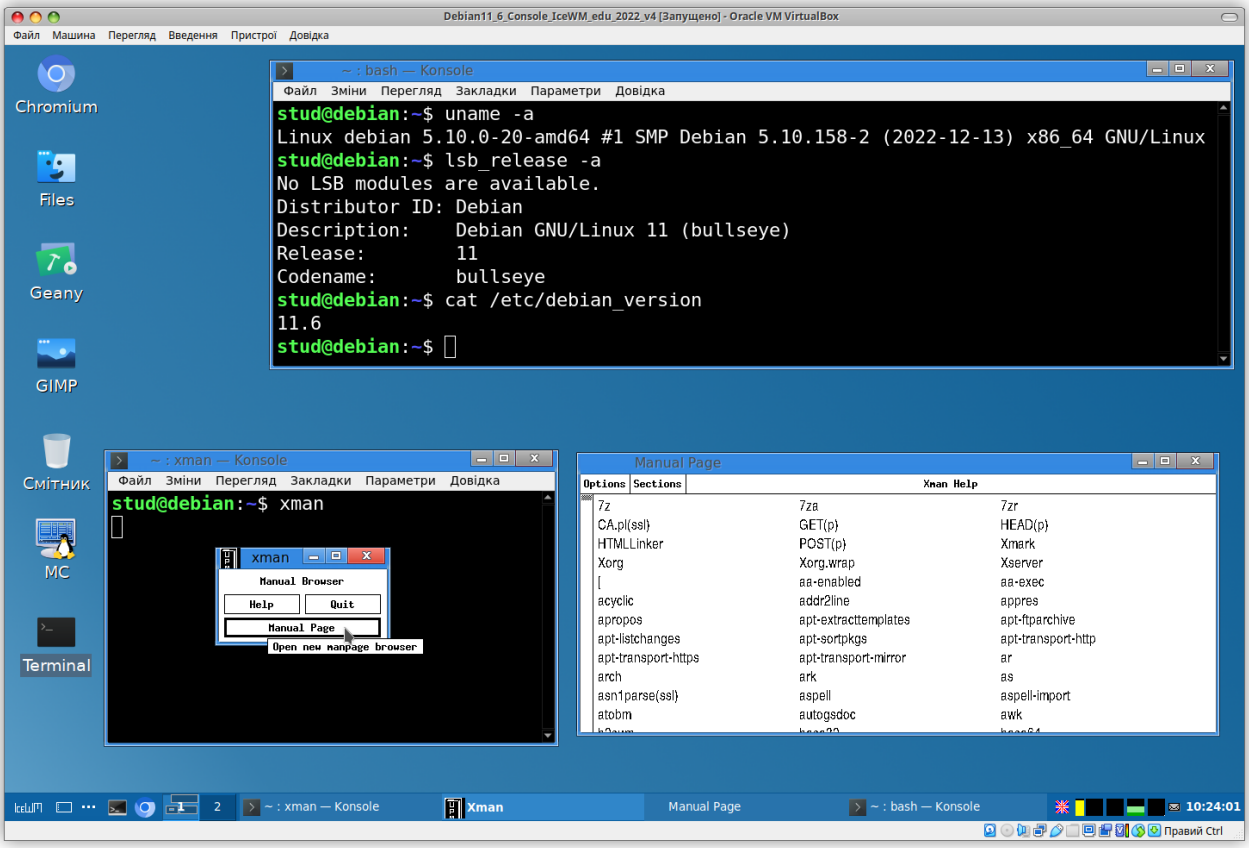


Рис. 2.1. Приклад виклику *xman* з консолі у графічному режимі

Отримати коротку довідку по будь-якій Unix-подібній команді можливо при використанні ключа `--help` у складі цієї команди. Наприклад для команди `mkdir`:

```
mkdir --help
```

Команда *info* є GNU-засобом перегляду гіпертексту: відображає оперативну документацію, попередньо створену з вихідних файлів *Texinfo*. Інформаційні файли (info-файли) мають ієрархічну структуру, містять меню та підтеми. Якщо команда *info* відсутня, тоді, наприклад, в Debian-подібних ОС її можна встановити командою:

```
sudo apt intall info
```

Примітка 2.1: в ОС на базі ядра Linux велика частина програм надана проєктом GNU, що розвивається під управлінням Фонду вільно поширюваного програмного забезпечення. GNU – це рекурсивна абревіатура: «GNU's not Unix» (GNU – це не Unix). Часто щоб підкреслити цю особливість кажуть або пишуть не "Linux", а "GNU/Linux".

Примітка 2.2: існують й інші програмні рішення для відображення довідникової документації, які входять до складу графічних оболонок KDE, GNOME, Xfce та ін. (наприклад, *Gman*). Окрім того, можна скористатися окремою програмою *man2html* для конвертування потрібної man-сторінки документації в html-сторінку. Наприклад, для відкриття у браузері *chromium* сторінки документації по команді `ls` (після встановлення *man2html* та *chromium*), виконують команду:

```
man2html $(man --path ls) > ./ls.html && chromium ./ls.html
```

Примітка 2.3: при встановленні додаткового пакету *man2html* може бути встановлений Web-сервер Apache2. Для його зупинки та деактивації в Debian-подібних ОС виконують команду:

```
sudo systemctl disable --now apache2
```

Крім наведених вище можливостей отримання довідки, користувач також може використовувати сторінки документації, які в більшості Unix-подібних ОС розташовуються в наступних каталогах:

```
/usr/doc
/usr/doc/Linux-HOWTOs
/usr/doc/Linux-mini-HOWTOs
/usr/doc/Linux-FAQs
/usr/src/linux/Documentation
/usr/local/doc
/usr/share/doc
/usr/share/info
/usr/local/share/doc
/usr/local/share/info
```

Можливе й інше розташування в системі. Крім того, сторінки довідки HOWTO (від англійського how to – як зробити) та/або FAQ (Frequently Asked Question(s), що часто задаються) можуть бути відсутні. Це залежить від того встановив їх адміністратор чи ні.

Практична частина

1. Запустити VirtualBox, а в ньому, встановлену раніше ОС.
2. Увійти в систему під ім'ям/паролем: *stud / studstud*.
3. Ознайомитися з різними способами отримання довідникової інформації на прикладах, розглянутих в теоретичній частині.

При захисті роботи дати вірні відповіді на наступні питання:

1. Які способи отримання довідки існують в Unix-подібних ОС?
2. Як отримати довідку, використовуючи команду *man*?
3. Як отримувати довідку з певного розділу?
4. Які вивчені ключі команди *man* розширюють можливості пошуку та яке призначення цих ключів?
5. Як отримати коротку довідку по певній команді або програмі?
6. У яких каталогах Unix-подібних ОС слід шукати документацію?

Робота №3.
Основні консольні команди Unix-подібних ОС

Мета роботи: вивчити консольні команди Unix-подібних ОС, що мають найбільше практичне використання: *pwd, ls, ls -l, ls -la, mkdir, rm, mv, cd, vim, cat, cp, chmod, chown, chgrp, ps, top, free, df, who, uname, echo, less, nano*.

Теоретична частина

Короткий опис команд, які надані в лабораторній роботі, представлено в таблиці 3.1.

Таблиця 3.1

Консольні команди Unix-подібних ОС, які найчастіше використовують

Команда	*Короткий опис*
`pwd`	Виводить повне ім’я поточного робочого каталогу
`ls`	Відображення списку файлів та каталогів
`mkdir`	Створення каталогу
`rm`	Видалення файлів або каталогів
`cd`	Перехід в заданий каталог (зміна каталогу)
`cat`	Конкатенація файлів або відображення їх вмісту
`cp`	Копіювання файлів/каталогів
`mv`	Переміщення/перейменування файлів або каталогів
`chmod`	Зміна прав доступу до файлів/каталогів
`chown`	Зміна власника-користувача файлу/каталогу
`chgrp`	Зміна власника-групи файлу/каталогу
`ps`	Виведення інформації про процеси в системі
`top`	Виведення інформація з оновленням в часі про процеси в системі
`free`	Виведення інформації про використання оперативної пам’яті
`df`	Виведення інформації про розподіл дискового простору
`who`	Виведення інформація про користувачів, які працюють в системі
`uname`	Виведення інформації про поточну систему (ядро, архітектуру та ін.)
`echo`	Виведення на консоль сформованого рядку
`vi`	Виклик консольного текстового редактору vi
`vim`	Виклик покращеної версії консольного текстового редактору vi
`nano`	Виклик консольного текстового редактору nano
`less`	Перегляд заданого файлу з можливістю прокручування

Для отримання більш детальної довідки, наприклад, по ключам (опціям) команд, використовуйте команду *man*. Наприклад: `man cp`.

Консольний текстовий редактор vim

До складу Unix-подібних ОС, входить велика кількість різноманітних текстових редакторів, в тому числі консольних. Вони особливо корисні в тих випадках, коли графічна підсистема з певних причин недоступна. Класичним консольним текстовим редактором є *vi*. Він володіє великим функціональним потенціалом. Всі дії виконуються через комбінації клавіш та команди, які вводить користувач. Існує велика кількість його поліпшених версій, однією з яких є редактор *vim*. У багатьох Unix-подібних ОС, команда *vi* є лише посиланням на одну з поліпшених версій редактору.

Для встановлення редактору *vim* у Debian-подібній ОС виконайте команду:

```
sudo apt install vim
```

або, якщо *sudo* не встановлено:

```
su
apt install vim
```

Розглянемо основні кроки створення та редагування текстових файлів за допомогою консольного редактора *vim*. Для відкриття текстового файлу в редакторі вводять команду:

```
vim ./filename
```

Якщо файл *filename* відсутній в поточному каталозі, то буде відкрито новий порожній файл. Зазначений файл відкривається в режимі перегляду. Для входу в режим вставки/редагування тексту, у *vim* необхідно натиснути на клавіатурі клавішу *i*.

Після завершення редагування потрібно вийти з режиму вставки, натиснувши клавішу *Esc*.

Операції виходу з редактора або збереження здійснюються з рядку команд. Для переходу в режим введення команд, потрібно ввести з клавіатури символ ":" (двокрапка), за ним потрібно вводити команди. Наприклад, щоб зберегти набраний текст та вийти з редактору, вводиться команда *wq* (команда запису змін (**w**rite) та виходу (**q**uit)):

```
:wq
```

Для виконання команди, після її введення, натискають клавішу *Enter*.

Примітка 3.1: всі команди, що починаються з ":", повинні завершуватися натисканням клавіші *Enter*.

Якщо користувач не робив ніяких змін, то вихід з редактору здійснюється командою *q*:

```
:q
```

Якщо з яких-небудь причин неможливо вийти з редактору у звичайному режимі, то застосовують примусове завершення роботи без збереження змін за допомогою команди:

```
:q!
```

Якщо редагований файл потрібно зберегти під іншим іменем, то вводять команду:

```
:w ім'я_файлу
```

Наприклад:

```
:w ./test.txt
:w /home/stud/test.txt
```

Варіант із збереженням та подальшим виходом:

```
:wq ./test.txt
:wq /home/stud/test.txt
```

Якщо редагований файл має атрибут "тільки для читання" та потрібно примусовий запис в нього інформації, то застосовують наступну команду:

```
:w!
```

Слід пам'ятати, що для виходу з командного режиму (для команд, які починаються з ":"), потрібно натискати клавішу *Esc*.

Таблиця 3.2

Команди *vim*, що часто використовуються
(в таблиці тільки дві команди, які вводяться після символу ":")

Команда	*Опис*
dw	Видалення ділянки тексту до кінця слова в буфер редактору
dd	Видалення рядку в буфер редактора
u	Скасування результату роботи попередньої команди
p	Вставка фрагмента тексту з буфера редактора
shift+g	Швидкий перехід в кінець файлу
/слово	Пошук слова у прямому напрямку (в бік кінця файлу)
?слово	Пошук слова у зворотному напрямку (до початку файлу)
n або shift+n	Повтор пошуку
:%s/було/стало/g	Швидка заміна слова "було" на слова "стало" в усьому файлі

ctrl+g	Інформація про поточний стан курсору в файлі
:!зовнішня_команда	Виконання зовнішньої команди (наприклад – **:!pwd**)

Консольний текстовий редактор nano

Більш привабливішим для користувача при роботі в командному рядку є редактор *nano*. Останнім часом цей редактор включається до переважної більшості дистрибутивів ОС за замовчуванням. Приклад його інтерфейсу представлений на рис. 3.1.

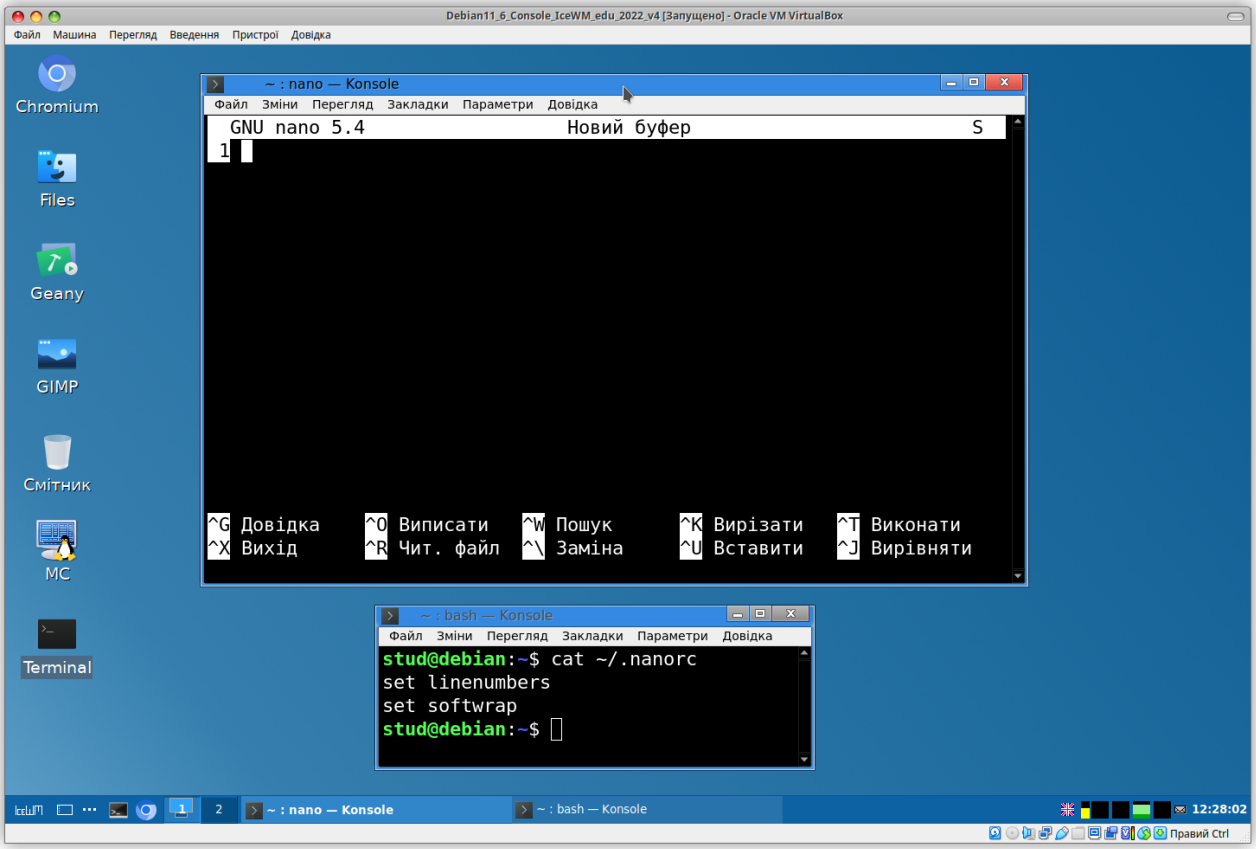


Рис. 3.1. Консольний редактор *nano* та приклад вмісту файлу його конфігурації

Тобто всі підказки основних комбінацій гарячих клавіш (є регістронезалежними) представлені в нижній частині вікна. Наприклад: Ctrl+G (виклик довідки по роботі з редактором nano), Ctrl+O (запис змін у файл), Ctrl+K (видалення поточного рядку в буфер), Ctrl+U (вставка з буфера), Ctrl+X (завершення роботи з nano або з певним режимом, наприклад, довідкою) та інші. Є й інші корисні комбінації. Наприклад, коментування рядку (Alt+3), швидке переходження до першого рядку (Ctrl+Home) або до останнього рядку

(Ctrl+End), початок позначення тексту (Ctrl+6), копіювання позначеного тексту в буфер (Alt+6) та інші.

Для більш зручного орієнтування по вмісту файлу, що редагується, існують додаткові опції редактора, які можна вказувати або з командного рядку, або зафіксувавши у прихованому файлі конфігурації – `.nanorc`, що розміщується у домашньому каталозі користувача (рис. 3.1).

Практична частина

Використовуючи матеріал теоретичної частини та команди отримання довідки, вивчені в попередній роботі, виконайте наступні пункти практичного завдання.

1. Виконайте команди: *pwd, ls, ls -l, ls -lh, ls -la*. Запротоколюйте та поясніть отриманий результат.

2. У домашньому каталозі з консолі створити каталог Text та зробити його поточним.

3. Використовуючи редактор *vim*, створити в каталозі Text невеликий текстовий файл вітання з ім'ям hello.txt.

4. За допомогою команди *cat* вивести вміст файлу hello.txt у вікно консолі. Запротоколювати результат виведення команди *cat*.

5. Створіть копії файлу hello.txt в поточному каталозі за допомогою команд *cp* та *cat*, назвавши нові файли hello_cp.txt та hello_cat.txt відповідно. Перевірити вміст поточного каталогу за допомогою команди *ls -l*. Виведіть у вікно консолі вміст нових файлів-копій. Запротоколювати результати.

6. Вивчіть та поясніть у звіті довідку по командам *echo, rm, chown* та *chgrp*, їх призначення та приклади використання.

7. Скасуйте у власника-користувача право на запис у файлі hello_cp.txt, а всім іншим користувачам скасуйте право на запис у файл hello_cat.txt. Перевірте та запротоколюйте результат.

8. Послідовно за допомогою однієї команди допишіть до кінця файлу hello_cat.txt вміст всіх файлів з каталогу */etc*, імена яких закінчуються словом "release" або "version". Виведіть вміст файлу hello_cat.txt на консоль за допомогою команди *less*. Запротоколюйте результат виконання команд.

9. У каталозі Text створіть підкаталоги backup та temp.

10. Виконати копіювання однією командою відразу всіх раніше створених текстових файлів в каталог backup. Перевірте та запротоколюйте результат.

11. Виконати копіювання каталогу backup з усім його вмістом в каталог temp. Видалити файл hello_cp.txt з каталогу backup, розташованого в каталозі temp. Перевірити та запротоколювати результати всіх виконаних команд.

12. Виконайте команду *ps -ef*, при цьому перенаправте її виведення у файл ps_result.txt, який повинен бути розміщений в каталозі Text. Надайте у звіті вміст файлу за допомогою певної команди.

13. Ознайомтеся з виведенням команд *top, free, df* та поясніть їх.

14. Вивчіть ключі команд *who* та *uname*. Опишіть їх у звіті.

15. Перейменуйте файл ./Text/hello.txt на ./Text/h.txt. Запротоколюйте виконання команди.

16. Залучіть можливості редактора nano для правки файлу-протоколу виконання роботи. У звіті вкажіть вивчені особливості роботи з редактором nano.

Робота №4.
Командний процесор Bash

Мета роботи: вивчити основні команди та інструкції командного процесору bash. Оволодіти технікою створення командних скриптів в середовищі GNU/Linux-сумісній ОС.

Теоретична частина

Всі командні процесори, встановлені в системі, перелічені у файлі */etc/shells*. В Unix-подібних ОС найбільш часто можна зустріти такі командні процесори: *sh* (Bourne shell), *bash* (Bourne Again SHell), *ksh* (KornSHell), *ash* (Almquist SHell), *dash* (Debian Almquist SHell), *zsh*, *csh*, *tcsh*, *fish* та ін.

Кожному новому користувачеві в Unix-подібних ОС призначається командна оболонка певного командного процесору за замовчуванням. Дізнатися, який командний процесор надає командну оболонку для конкретного користувача можна за допомогою команди (ім'я командного процесору вказується в кінці виведеного рядку-результату):

```
cat /etc/passwd | grep ім'я_користувача
```

або просто для поточного користувача:

```
echo $SHELL
```

sh – рання командна оболонка Unix, яка розроблена Стівеном Борном з Bell Labs в 1978 році, була у складі Unix Version 7 та є стандартом де-факто для Unix-подібних ОС. У 1987 році вона вдосконалена Брайаном Фоксом і існує під назвою *bash*. *bash* є командним процесором в Unix-подібних ОС, що найбільш частіше використовується.

Проєкт *bash* розвивається за підтримки Free Software Foundation в рамках проєкту GNU. Станом на жовтень 2024 року, актуальною є версія 5.2.37. Дізнатися версію встановленого *bash* можна увівши команду:

```
bash --version
```

або

```
echo $BASH_VERSION
```

Основне призначення *bash* – надати оболонку (shell) для виконання команд або сценаріїв (командних скриптів) автоматизації користувача. Команду, яка вводиться, *bash* сприймає як деяку програму, розміщену в одному з каталогів, зазначених у змінній оточення *PATH*, або в зазначеному каталозі, або

як внутрішню команду *bash*. В іншому випадку виводиться повідомлення про неможливість виконання команди.

Послідовність команд для автоматизації виконання певних завдань можна групувати *в сценарії*. При запуску оболонки *bash* виконується спеціальний сценарій *.bashrc*, що знаходиться в домашньому каталозі користувача. У ньому вказують команди, які необхідно виконати відразу після входу користувача в систему. Файл є необов'язковим і може бути відсутнім. Файл *.bash_history* зберігає історію команд, які вводяться користувачем в оболонці *bash*.

Команди інтерпретатора *bash* можуть бути об'єднані в сценарії (командні скрипти) та використані спільно із іншими скриптами або програмами ОС. Для створення командних скриптів можуть бути використані будь-які редактори простого тексту, наприклад, nano, vim, vi, emacs, mcedit, mousepad, leafpad, gedit, kwrite, kate, geany, MS Visual Studio Code або подібні. Невеликі сценарії можуть бути набрані навіть за допомогою команд *cat* або *echo* та перенаправлення виведенням у файл. Наприклад:

```
echo '#!/bin/bash' > ./myscript.sh
echo 'mkdir ./`date +%d.%m.%y_%H%M%S`' >> ./myscript.sh
```

або:

```
$ cat << EOF > ./myscript.sh
> #!/bin/bash
> mkdir ./`date +%d.%m.%y_%H%M%S`
> EOF
$
```

В останньому фрагменті прикладу символ `$` є привітанням для звичайного користувача у командному процесорі *bash*. За цим символом вводять команди *bash*. Символом привітання для суперкористувача (користувача з іменем *root*, кореневого користувача) є символ `#` .

У командному сценарії *bash* першим рядком обов'язково повинен бути рядок, який не містить пробілів, та має наступний формат:

```
#!/bin/bash
```

За цим рядком повинен йти текст коду сценарію. Для запуску файлу сценарію йому необхідно встановити право на виконання. Наприклад, якщо є файл скрипту *myscript.sh*, то для його виконання вводять команду:

```
chmod +x ./myscript.sh
```

Файли сценаріїв можуть мати розширення **.sh**, **.bash** або не мати його взагалі.

Примітка 4.1: встановлення атрибуту на виконання можливо у випадках, коли файл розташований в розділах GNU/Linux-сумісних або Unix-подібних файлових систем (ФС), наприклад, таких як: ext2, ext3, ext4, ufs, zfs, btrfs та їм подібним. Однак у випадках з ФС Microsoft (FAT, FA32, NTFS, exFAT), файли вже мають атрибут на виконання, але їх виконання з цих ФС може бути неможливим, якщо не будуть виконані певні системні налаштування при монтуванні подібних ФС.

Можна виконати скрипт і без права на виконання. Для цього треба викликати безпосередньо командний процесор та передати йому в якості аргумента ім'я файлу командного скрипту. Наприклад:

```
bash ./myscript.sh
```

Часто в GNU/Linux-подібних ОС команда *sh* є символічним посиланням (symbolic link) на *bash*. Тому виклик *sh* буде відповідати виклику *bash*. Перевірити, чи є *sh* просто посиланням на *bash* або це справжній командний процесор *sh*, можна за допомогою команди:

```
ls -l /bin/*sh
```

Коментарі в *bash*-скриптах починаються з символу # та пробілу.

Якщо потрібно здійснити виведення будь-якого повідомлення (рядку) на консоль, то використовують команду `echo`.

Для оголошення змінної (використовується тільки всередині файлу-скрипту) є конструкція: `змінна=значення`. Пробільні символи між змінною та символом присвоєння та між символом присвоєння і значенням, відсутні!

Приклад командного скрипту:

```
#!/bin/bash
# This is my test bash-script
MYPROG=$HOME/myprograms/myprog1
echo 'HOME directory: '$HOME
echo 'My program:'
echo $MYPROG
echo 'Enter program name:'
read MYPROG
echo $MYPROG
CURDIR=`pwd`
echo 'Current directory: '$CURDIR
```

У прикладі показані деякі команди *bash* та робота із змінними. Команда `read` здійснює введення з клавіатури. Конструкція `змінна=`команда`` встановлює в якості значення результат виведення зазначеної в похилих лапках (` `) команди.

В GNU/Linux-подібних ОС існують спеціальні системні змінні оточення, що містять службові дані, наприклад:

HOME – домашній каталог користувача, який запустив сценарій;

PATH – шляхи пошуку програм для запуску;

PWD – поточний каталог;

UID – ID користувача, що запустив сценарій;

USER – ім’я користувача, що запустив сценарій;

RANDOM – випадкове число в діапазоні від 0 до 32767;

LANG – поточні мовні налаштування терміналу;

HOSTNAME – ім’я комп’ютеру (хосту);

HISTFILE – ім’я файлу журналу команд, з яким відбувається робота;

PS1 – первинне запрошення командного рядку;

PS2 – вторинне запрошення нового рядку для незакінченої команди.

Регістр в іменах змінних має значення. Наприклад, такі команди виводять на консоль абсолютно різні результати:

```
echo $PATH
echo $path
```

Для встановлення власної змінної оточення використовується команда `export` наступного вигляду (в цьому випадку змінна буде доступна всім дочірнім процесам з *bash*):

```
export MYVAR
```

або

```
export MYVAR=значення
```

Обробка аргументів сценаріїв здійснюється за допомогою звернення до спеціальних змінних:

`$0` – містить ім’я сценарію;

`$n` – містить значення `n`-го аргументу сценарію (наприклад: `$1` – перший аргумент, `$9` – останній аргумент);

`$#` – містить кількість переданих скрипту аргументів в командному рядку.

`$@` – містить рядок із всіма аргументами скрипту, який можна окремо розібрати на частини (доречно використовувати при великій кількості переданих аргументів – див. приклад нижче).

Кількість аргументів скрипту, до яких можна звернутися напряму через змінні `$1`, `$2` і так далі, є обмеженою 9 (тобто `$9`). Якщо треба продовжити звертатися до наступних аргументів, то потрібно використати команду `shift` для зсуву всіх параметрів вліво (буде втрата перших аргументів). Тобто, якщо маємо десятий аргумент, то після команди `shift` до нього можна буде звернутися через змінну `$9`, а змінна `$1` буде містити вже другий аргумент скрипту.

Приклад доступу до великої кількості вхідних аргументів сценарію із залученням оператору циклу *for*:

```
#!/bin/bash
arguments=($@)
for (( i=0; i<$#; i++ )); do
    echo "${arguments[$i]}"
done
```

Інструкції та команди bash

В *bash* доступні два умовних оператора – *if* та *case*. синтаксис:

```
if умова_1 then
      команди_1
elif умова_2 then
      команди_2
...
elif умова_N then
      команди_N
else
      команди_N+1
fi
```

Кількість блоків *elif* необмежена. Формат умов (символи пробілів обов'язкові, завершальна крапка з комою – обов'язкова!):

```
[ Змінна вираз значення | змінна ];
```

де виразом якась з конструкцій:

`==` – дорівнює
`! =` – не дорівнює
`-lt` – менше
`-gt` – більше
`-le` – менше або дорівнює
`-ge` – більше або дорівнює
`-eq` – дорівнює
`-e` *ім'я_файлу* – умова істинна, якщо файл існує
`-f` *ім'я_файлу* – умова істинна, якщо файл існує і він регулярний
`-d` *ім'я_каталогу* – умова істинна, якщо каталог існує
`-x` *ім'я_каталогу* – умова істинна, якщо файл є на виконання
`!` – заперечення умови
`&&` – логічне І (ТА)
`||` – логічне АБО

Приклади умов:

```
# Змінна MAX дорівнює 30
[ $MAX == 30 ]
# Змінна MAX не дорівнює 30
[ $MAX != 30 ]
# MAX менше 30
[ $MAX -lt 30 ]
# MAX менше або дорівнює N
[ $MAX -le $N ]
```

Повний приклад сценарію з оператором *if*:

```
#!/bin/bash
MAX=10
N=5

if [ $MAX -lt $N ]; then
    echo 'MAX < N'
else
    echo 'MAX > N'
fi

if [ -f ./"$1" ]; then
    echo " $1 - This is regular file!"
else
    echo " $1 - This is NOT regular file!"
fi

A=10
B=20

if [ $A -gt 0 ] && [ $B -gt 0 ]; then
    echo 'A>0 та B>0'
fi
```

Оператор *case* має наступний синтаксис:

```
case  змінна  in
      значення_1) команди_1 ;;
      ...
      значення_N) команди_N ;;
      *) команди_за_замовчуванням ;;
esac
```

Значення зазначеної змінної по черзі порівнюються із наведеними значеннями (`значення_1`, ..., `значення_N`). Якщо є збіг, то будуть виконані команди, які відповідають вказаним значенням. Якщо збігів немає, то будуть виконані команди за замовчуванням. Приклад:

```
#!/bin/bash

case "$1" in
    start)
        echo "Запускаємо процес"
    ;;

    stop)
        echo "Зупиняємо процес"
    ;;

    *)
        # Виводимо довідникове повідомлення
        echo "Використовуйте: $0 [start | stop]"
        # Залишаємо сценарій з кодом 1
        exit 1
    ;;
esac
```

Циклічні операції в *bash* реалізуються за допомогою операторів: *for*, *while*, *until*, *select*. Синтаксис *for*:

```
for змінна in список ;
do
    команди ;
done
```

або

```
for (( вираз_1 ; вираз_2 ; вираз_3 )); do
      команди ;
done
```

Приклади:

```
for i in 1 2 3 4 5;
do
    echo "Start loop iteration -->";
    echo $i;
done

for (( i=0; i<10; i++ )); do
    echo "Loop iteration #"$i
done
```

Синтаксис *while*:

```
while умова; do
    команди ;
done
```

Приклад:

```
i=0
while [ $i -lt 10 ]; do
    echo "Loop iteration # $i";
    i=$(( $i+1 ));
done
```

Оператор виду `$((арифметичний_вираз))` повертає результат обчислення виразу. У виразах можуть використовуватися різні операції, по синтаксису відповідні мові програмування С. Наприклад:

`++` – операція інкремент;

`--` – операція декремент;

`+=` – операція складання з присвоєнням;

`%` – операція взяття залишку від ділення і т.п.

Наприклад, той же цикл з операцією `++` буде виглядати так:

```
i=0
while [ $i -lt 10 ]; do
    echo "Loop iteration # $i";
    ((i++));
done
```

Інший приклад циклу, тіло якого виконується десять разів, починаючи від нуля. При цьому у термінал консолі буде виведений рядок з номером ітерації тільки за умови, що залишок від ділення номеру ітерації на 2, буде дорівнювати нулю:

```
for (( i=0; i<10; i++ )); do
    if [ $((i%2)) -eq 0 ]; then
        echo "Loop iteration #"$i
    fi
done
```

Команда *let* дозволяє виконувати цілочисельні операції. Наприклад:

```
i=4
let "count=0" "i=i+1" "num=4"
# виведе 0
echo $count
# виведе 5
echo $i
# виведе 4
echo $num
```

В *bash* вбудовано багато команд, відомості про яких можна отримати за допомогою команди *man bash*. Наприклад, вбудована команда `exit` з кодом повернення виконує завершення командного скрипту та повертає код завершення для подальшого аналізу в операційне середовище. Наприклад:

```
exit 1
```

Команда `test` з умовою повертає 0 або 1 в залежності від виконання умови. Наприклад, перевірка існування файлу та його виконання у разі успіху:

```
test -e ./myscript.sh && bash ./myscript.sh
```

Для з'ясування чи маємо справу з внутрішньою командою оболонки, чи з окремою програмою, яка реалізує команду, можна звернутися до команди `type`. Опція `-a` надасть додаткову інформацію. Приклади:

```
type type
type exit
type test
type for
type -a read
type -a ls
type -a pwd
```

В результаті виконання останньої команди бачимо, що `pwd` є і як внутрішня команда оболонки, і як окрема програма в певному каталогу (або каталогах) ОС.

Псевдоніми команд

Команда *alias* без аргументів виводить на екран список визначених псевдонімів команд. Псевдоніми можуть бути корисні, коли користувач часто виконує якусь довгу команду або ж потрібно модифікувати виконання команди за замовчуванням, наприклад, ввести попередні запити. Також, якщо певна команда виконується часто із заданим ключем, їй також можна призначити псевдонім. Це відбувається за наступним синтаксису:

```
alias псевдонім="команда_з_аргументами"
```

Наприклад, введемо нові три команди: `myps`, `ps_mod` та `killp`, які дозволять вивести докладно інформацію про всі процеси поточного користувача (*myps*), топові процеси, які витрачають багато пам'яті (*ps_mod*) та команду видалення процесу з пам'яті, посилаючи йому сигнал *SIGKILL* (*killp*). Псевдоніми опишемо в кінці файлу *$HOME/.bashrc* рядками:

```
alias myps="ps -ef | grep $USER "
alias ps_mod='ps -eo pid,ppid,cmd:50,%mem,%cpu --sort=-%mem | head'
alias killp="kill -s SIGKILL "
```

Приклад виконання нових команд в терміналі:

```
$ nano &
$ myps | grep nano
$ killp $(pidof nano)
$ ps_mod
```

Виконання декількох команд

При формуванні команди можна використовувати одночасно кілька команд, застосовуючи спеціальні роздільники.

Якщо між командами використовується символ '|' , то це говорить про формування так званого конвеєра команд, коли другій команді в якості аргументів передається результат першої команди, або на вхід третьої команди передається результат другої, яка обробляла аргумент від першої і т.д. Наприклад, наступний конвеєр дозволяє вивести на консоль інформацію про всі процеси, потім відфільтрувати тільки ті, які мають відношення до активного користувача, а потім, якщо список виведення більший за висоту вікна терміналу або екрану, забезпечити прокручування командою `more`:

```
ps -ef | grep $USER | more
```

Якщо між командами використовується символи '||', то друга команда, що йде за першою, виконається тільки у разі невиконання першої. Наприклад, видалити файл *abc.sh*, розташований в поточному каталозі і якщо його немає, вивести детальний список файлів:

```
rm ./abc.sh || ls -l
```

Якщо між командами використовується символи '&&', то друга команда, що йде за першою, виконається тільки у разі успішного завершення виконання першої. Наприклад, створити каталог та перейти до нього із виведенням шляху:

```
mkdir ./test && cd ./test && pwd
```

Якщо між командами використовується символ ';' , То формується так званий список команд, які просто повинні виконуватися одна за одною як є, незалежно від результатів виконання попередньої команди в списку. Наприклад, вивести список всіх користувачів, що працюють на даний момент в системі, а потім список останніх 10-ти входжень в систему поточного користувача (списки відокремити один від іншого символом нового рядку):

```
who; echo; last $USER | head
```

Якщо потрібно перенаправити виведення із стандартного потоку виведення (stdout) команд(и) в файл, використовують символи перенаправлення:

– ‘>’ – створює новий файл (якщо файл існує, то його вміст буде втрачено!);

– ‘>>’ – додає виведення команди в кінець файлу (якщо файлу немає, то він буде створений).

Якщо вказуються комбінації ‘2>’ або ‘2>>’, то маємо справу з перенаправленням стандартного потоку виведення помилок (stderr).

Наприклад, перенаправити виведення всіх перерахованих команд в файл *logfile.txt*:

```
(date; echo; free -mh; echo; who; echo; last $USER | head) > ./logfile.txt
```

Ще приклади.

Вивести в файл *find.txt* результати пошуку та перенаправити повідомлення про помилки в файл *find_err.txt*:

```
find /etc -iname '*.conf' > ./find.txt 2> ./find_err.txt
```

Вивести в файл *find.txt* результати пошуку та перенаправити повідомлення про помилки на нульовий пристрій:

```
find /etc -iname '*.conf' > ./find.txt 2> /dev/null
```

Робота з архівами

У різних ОС існує безліч програм архівування даних. В Unix-подібних ОС отримала широке поширення утиліта архівування *tar* (tape archive). Спочатку ця програма розроблялася для створення архівів на магнітній стрічці. Зараз вона використовується для зберігання кількох файлів всередині одного файлу, а також для поширення ПЗ та створення архіву ФС.

Одним з переваг формату *tar* при створенні архівів є те, що в архів записується інформація про структуру каталогів, інформація про власника та групи окремих файлів, а також тимчасові мітки файлів.

Однією з особливостей *tar* є те, що вона не створює стислих архівів, а використовує для стиснення зовнішні утиліти, наприклад, такі як *gzip*, *bzip2, xz* або аналогічні.

Файли, що містять архіви *tar* мають, як правило, розширення *.tar*. Якщо *tar*-файл стиснутий зовнішньої утилітою, то до початкового розширення додається ще одне, яке свідчить про те, яка утиліта стиснення використовувалася, або скорочення. Наприклад, у разі *gzip* – це розширення:

```
.tar.gz, .tar.gzip, .tgz
```

У випадку з *bzip2* – це розширення:

```
.tar.bz2, .tar.bzip2, .tbz2, .tbz, .tb2.
```

У випадку з *xz* – це розширення:

```
.tar.xz, .txz, .lzma, .tlz, .lz
```

Для отримання інформації щодо залучення опція, яка буде вказувати, який алгоритм стиснення та відповідну утиліту потрібно використовувати, можна використати команду отримання швидкої довідки:

```
tar --help | grep gzip
tar --help | grep bzip2
 tar --help | grep xz
```

Нехай, наприклад, існує каталог */home/stud/data*, вміст якого має бути піддано архівації. Ім'я файлу-архіву: *data.tar.gz*. Тоді необхідно ввести наступну команду:

```
cd /home/stud ; tar -czvf ./data.tar.gz data
```

Опція *-c* каже про створення архіву, *-z* – встановлює використання утиліти *gzip*, а за опцією *-f* необхідно вказати ім'я файлу-архіву, який створюється.

Якщо необхідно вивести список файлів та каталогів, розміщених в архіві без розпакування, то використовують опцію *-t*. Наприклад:

```
tar -tf ./data.tar.gz
```

Розпакування даних відбувається при вказівці опції *-x*. Якщо задана опція *-v*, то користувач побачить виведення на екран файлів архіву при їх обробці, а при використанні опції *-C* можна вказати каталог для розпакування. Якщо каталог не заданий, то розпакування буде зроблено в поточний каталог. Приклади команд розпакування:

```
tar -zxf ./data.tar.gz
tar -zxvf ./data.tar.gz -C /home/stud/tmp/
```

Практична частина

1. Написати командний *bash*-скрипт, який дозволяє:
– створити в $HOME каталог *Images*;

– створити однією командою в *Images* підкаталоги *JPG*, *PNG*, *GIF*;

– здійснити пошук всіх файлів з розширеннями *JPG, jpg* та *jpeg* в підкаталогах каталогу */usr* і знайдені файли відразу скопіювати в каталог *Images/JPG*;

– виконати ті ж дії з файлами, які мають розширення *PNG, png, GIF* та *gif*, копіюючи їх до каталогів відповідно *Images/PNG* та *Images/GIF*;

– після того як файли знайдені та скопійовані, створити архів вмісту каталогу *Images* за допомогою програми *tar* із стисненням *gzip.* Ім'я файлу-архіву: *images.tar.gz*;

– повторити операцію та створити архів із залученням стиснення *bzip2*. Ім'я файлу-архіву: *images.tar.bz2*;

– здійснити рекурсивне видалення каталогу *Images*.

– створити однією командою підкаталоги *$HOME/Result1* та *$HOME/Result2*.

– здійснити розпакування архівів *images.tar.gz* та *images.tar.bz2* відповідно у каталоги *$HOME/Result1* та *$HOME/Result2*.

Шляхи до створюваних каталогів повинні зберігатися у відповідних змінних: *DATADIR*, *JPGDIR*, *PNGDIR*, *GIFDIR*. Після завершення створення архіву, видалення каталогу *Images* має бути здійснено тільки за умови успішного створення файлу-архіву. При цьому, в разі успіху, скрипт повертає в середовище виконання код нуля, в разі невдачі – код одиниці.

При підготовці *bash*-скрипту скористатися наявними можливостями відомих команд ОС та *bash*-операторів.

2. Створити у своєму оточенні псевдоніми для полегшення розпакування *tar.gz*- та *tar.bz2*-архівів. Продемонструвати роботу псевдонімів на довільному тимчасовому каталогу для розпакування та файлах-архівах *images.tar.gz* і *images.tar.bz2*.

3. Запропонувати власний командний *bash*-скрипт для знаходження в каталозі *$HOME/Result1 png*-файлів певного розміру або більше за вказаний. Каталог та граничне значення розміру файлів для пошуку, передаються як аргументи скрипту в командному рядку. При знаходженні таких файлів формувати окремо список. На виході забезпечити виведення списку у відсортованому вигляді у вікно терміналу консолі.

При підготовці *bash*-скрипту скористатися наявними можливостями команд *find, sort* та *bash*-операторів.

У звіт включити код скриптів з коментарями та запропоновані псевдоніми команд. Продемонструвати роботу скриптів та результати.

Робота №5.
Microsoft Windows PowerShell

Мета роботи: набути навичок роботи в командній оболонці Microsoft Windows PowerShell, вивчивши основні команди та практики.

Теоретична частина

Початок роботи

Windows PowerShell – це засіб автоматизації від Microsoft, який складається з оболонки із інтерфейсом командного рядку та супутньої мови сценаріїв. Windows PowerShell побудований на базі технології Microsoft .NET Framework та інтегрований з нею (як мінімум вимагає встановлену в ОС бібліотеку .NET Framework версії 2.0 або вище). В нових версіях ОС MS Windows цей засіб включено за замовченням.

Додатково, PowerShell надає зручний доступ до технологій COM (Component Object Model – об'єктна модель компонентів), WMI (Windows Management Instrumentation – інструментарій управління Windows) та ADSI (Active Directory Service Interfaces – інтерфейси служби Active Directory), так само як і дозволяє виконувати звичайні команди командного рядку для створення єдиного оточення, в якому адміністратори змогли б виконувати різні завдання на локальних та віддалених системах.

Версія 1.0 випущена в 2006 році та доступна для установки для середовищ Windows XP SP2, SP3, Windows Server 2003, Windows Vista.

Версія 2.0 інтегрована у вигляді компоненту ОС Windows 7, Windows Server 2008 R2 та поставляється з графічною оболонкою Windows PowerShell Integrated Scripting Environment (ISE), що містить вбудований відладчик, підсвічування синтаксису, автоматичне завершення команд.

Версія 3.0 інтегрована у вигляді компоненту систем Windows 8, Windows Server 2012 і також містить ISE.

Версія 4.0 інтегрована у Windows 8.1 та Windows Server 2012 R2, а також доступна для Windows 7 SP1, Windows Server 2008 R2 SP1, Windows Server 2012.

Версія 5.0 стала доступною з п'ятою версією продукту Windows Management Framework 5.0 (WMF 5) у 2014 році. Ця версія була перевипущена з корекцією різних помилок у 2016 році.

Версія 5.1 була випущена влітку 2016 року для Windows 10, Windows Server 2016. Оновлення стали доступними для Windows 7, Windows Server 2008/2008 R2/2012/2012 R2. Вона стала першою версією, яка

випускається в двох редакціях: Desktop та Core. Desktop-редакція працює над стеком .NET Framework, а Core-редакція використовує .NET Core у складі Windows Server 2016 Nano Server.

Версія 6.0 була анонсована влітку 2016 року. Ця версії та наступні є кросплатформними та засновані на .NET Core. При її встановлені на різні платформи версій 6.0 та 6.1 можна використовувати в MS Windows, macOS, CentOS, RHEL, Debian, Ubuntu, OpenSUSE.

Версія 7.0 вийшла у березні 2020 року та знаменує перехід до використання .NET Core 3.1.

В листопаді 2020 року у MS Windows 10.0.19042.508 (версія 20H2) за замовчанням доступний MS PowerShell версії 5.1.19041.1 та новіша версія ISE.

У версії MS Windows 11 21H2 доступний MS PowerShell версії 5.1.22000.65.

У версії MS Windows 11 22H2 доступний MS PowerShell версії 5.1.22621.2506.

Windows PowerShell 5.1 залишається передвстановленим в ОС MS Windows 10, Windows 11 та Windows Server 2022.

Також існують певні редакції PowerShell для Unix-подібних ОС, зокрема, для GNU/Linux-сумісних.

В роботі розглянута MS Windows PowerShell версії 5.1.

Для встановлення більш новіших версій PowerShell, можна скористатися документацією за посиланням:

https://learn.microsoft.com/en-us/powershell/scripting/install/installing-powershell-on-windows

Для запуску *Windows PowerShell ISE*, в рядку пошуку програм, введіть `powershell` та оберіть для запуску *Windows PowerShell ISE*. Це середовище можна запустити з командного рядку Windows командою:

```
powershell_ise
```

На рис. 5.1 представлено головне вікно середовища ISE.

Щоб дізнатися версію встановленої Windows PowerShell (далі скорочено PS), у вікні терміналу PS-консолі введіть одну з команд:

```
host
```

або

```
$host
```

або

```
$host.version
```

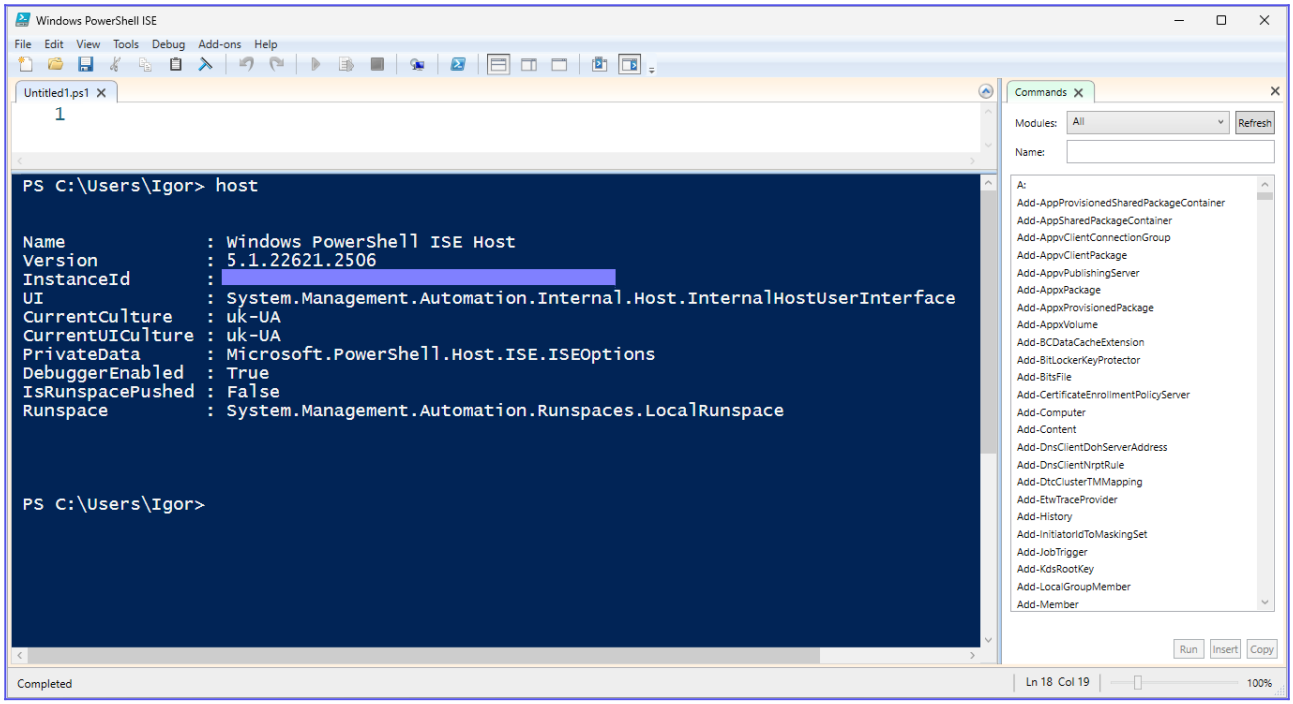


Рис. 5.1. MS Windows PowerShell ISE в ОС MS Windows 11 22H2

PS містить велику кількість команд, які називаються командлетами (cmdlets). Наприклад у показаній на рис. 5.1 версії маємо 681 командлет. Для виведення загальної кількості можна залучити PS-команду:

```
(Get-Command -CommandType Cmdlet).Count
```

або

```
(gcm -CommandType Cmdlet).Count
```

або

```
(gcm -co cmdlet).count
```

або

```
gcm -co cmdlet | measure
```

Примітка 5.1: `Get-Command` (псевдонім `gcm`) також є командлетом.

Примітка 5.2: загальна кількість командлетів в різних версіях PowerShell відрізняється.

Командлети слідують певним правилам іменування.

1. Команди PS складаються з дієслова та іменника, розділених тире, що записуються англійською мовою. Наприклад, команда (вона також є командлетом) виклику довідки: `Get-Help` (регістр символів значення не має).

2. Перед параметрами ставиться символ тире. Наприклад, команда більш детального виведення довідкової інформації:

```
Get-Help -Detailed
```

3. В PS багатьом командам відповідають псевдоніми. Наприклад, команді `Get-Help` відповідають псевдоніми `help` (класичний Windows) та `man` (класичний Unix). Для виведення псевдонімів можна скористатися, наприклад, командами:

```
gcm -co alias
```

та

```
help -category alias
```

Для автоматичного завершення команд в ISE використовується клавіша TAB.

Команда виведення довідки для всіх команд PS:

```
help *
```

Перенаправити всі назви команд та командлетів у файли для детального розгляду (каталог *C:\Temp* повинен існувати на диску):

```
help * > C:\temp\ps_commands.txt

gcm -co cmdlet > C:\temp\ps_cmdlets.txt
```

Параметр `-Detailed`, як правило, використовують спільно з певною командою. Наприклад, наступні команди виводять користувачеві скорочену та детальну інформацію про команду `Get-Process` (псевдоніми `ps` та `gps`):

```
help Get-Process
help Get-Process -Detailed
help ps -full
```

Команда виведення повної довідки з прикладами (якщо вони встановлені), наприклад, для команди Get-Process:

```
help ps -examples
```

Можна також отримати довідку з прикладами за посиланням:

https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.management/get-process?view=powershell-5.1

При використанні команд може формувати конвеєр (формується через символ '|'). Так, наприклад, наступна команда виведе повний список команд PS, а потім перенаправить цей список у зазначений файл за допомогою командлета `Out-File`:

```
help * | Out-File c:\temp\help.txt
```

Для перенаправлення у файл можна також використовувати спеціальні символи ‘>’ або ‘>>’. Microsoft радить використовувати `Out-File` замість символів перенаправлення, коли потрібно скористатися можливостями параметрів цього командлету. Це може значно розширити перенаправлення.

Для запуску одного або декількох процесів на локальному комп’ютері використовують команду `Start-Process` або її псевдоніми `saps` або `start`. Наприклад, для відкриття збереженого файлу *help.txt* можна використовувати програму WordPad:

```
Start-Process wordpad c:\temp\help.txt
```

або

```
saps wordpad c:\temp\help.txt
```

або

```
start wordpad c:\temp\help.txt
```

Виконання скриптів

Для виконання скриптів, PS необхідні певні налаштування політик системи безпеки Windows. Перевірити, яка діє політика виконання для поточного сеансу, можна за допомогою командлета `Get-ExecutionPolicy`:. Просто виконання цієї команди пропонує ефективну політику:

```
Get-ExecutionPolicy
```

Поточний стан для різних рівнів подаються командою:

```
Get-ExecutionPolicy -List
```

Для зміни політики використовують командлет `Set-ExecutionPolicy`.

Перед зміною політики захисту дуже рекомендується після роботи із скриптами в процесі навчання, повертати політику в попереднє становище!!! Тому поточний стан завжди треба запам’ятовувати! Або встановити політику жорсткого захисту.

Зміну політики виконують із сесії Windows PowerShell (ISE) або в терміналі PS-консолі, які запускається <u>в режимі адміністратора</u>!

Допустимі значення політики виконання:
– *Restricted* – не завантажує файли конфігурації і не виконує скрипти;

– *AllSigned* – вимагає, щоб всі скрипти і файли конфігурації були підписані довіреним видавцем, в тому числі скрипти, підготовлені на локальному комп'ютері;

– *RemoteSigned* – вимагає, щоб всі скрипти та файли конфігурації, завантажені з Internet, були підписані довіреним видавцем;

– *Unrestricted* – завантажує всі файли конфігурації та виконує всі скрипти. Якщо запущено непідписаний скрипт, завантажений з Internet, то програма просить ввести дозвіл перед запуском;

– *Bypass* – нічого не блокується і ніякі попередження та інші запити не з'являються;

– *Undefined* – видаляє поточну призначену політику виконання з поточної області (параметр не видаляє політику виконання, задану в області групової політики).

Приклади команд від адміністратора:

```
Get-ExecutionPolicy -list
Set-ExecutionPolicy -scope CurrentUser -ExecutionPolicy Unrestricted -force
Set-ExecutionPolicy -scope LocalMachine -ExecutionPolicy Restricted -force
Get-ExecutionPolicy -list
```

Після встановлення відповідних політик, можна запускати скрипти від користувача.

Командні скрипти мають розширення **.ps1**.

Наприклад, нехай потрібно в корені каталогу *C:* створити каталог *Temp2*, надати всім користувачам та процесам повний доступ, створити в каталозі текстовий файл з переліком команд PS та після того як він буде створений, запустити на виконання процес WordPad, який відкриє цей файл. Приклад такого скрипту представлений на рис. 5.2 та рис. 5.3.

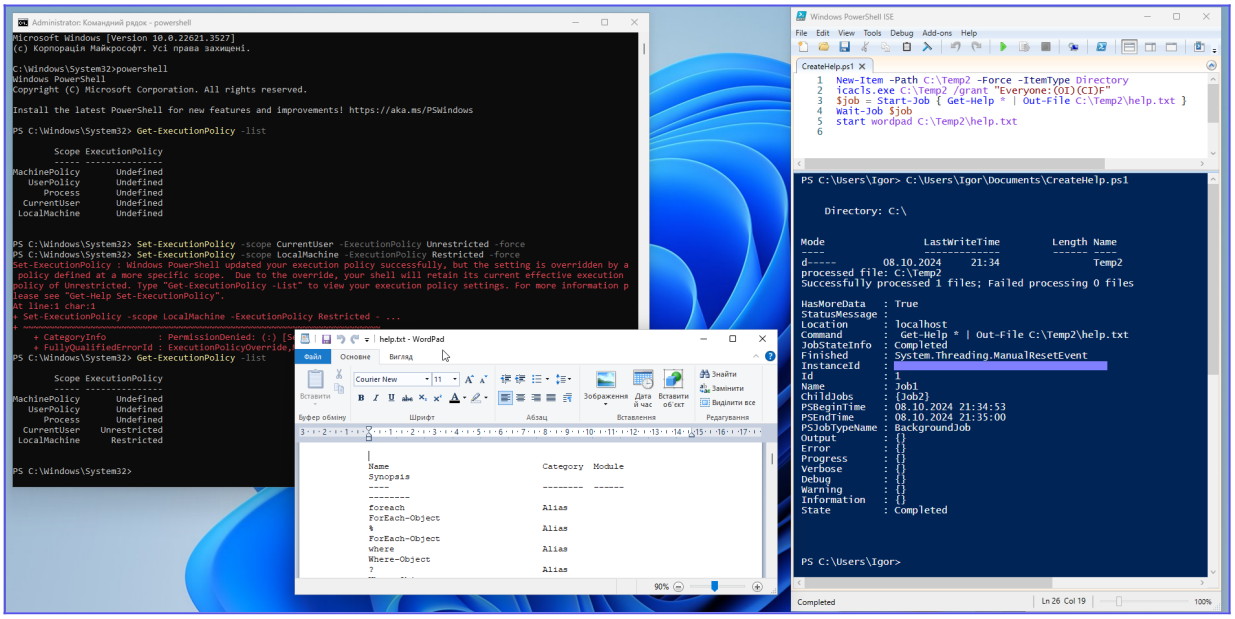


Рис. 5.2. Приклад скрипту Windows PowerShell

У наведеному прикладі для спрощення вказівки прав доступу до *C:\Temp2*, використовується системна утиліта `icacls`. Для прискореного створення нового каталогу використовується командлет `New-Item`. Для очікування завершення процесу формування файлу *help.txt*, використовуються командлети `Start-Job` та `Wait-Job`.

Робота з процесами

Для виведення списку всіх працюючих в системі процесів використовується команда `Get-Process` (псевдонім в стилі Unix-подібних: `ps`). Для сортування виведеного списку-результату (за замовчуванням по зростанню) використовують інший командлет `Sort-Object` або псевдонім `sort` (для сортування по спаданню використовують параметр `-Descending`). Як параметр, вказують властивість об'єкта, наприклад, відсоток завантаження процесора (рис. 5.3):

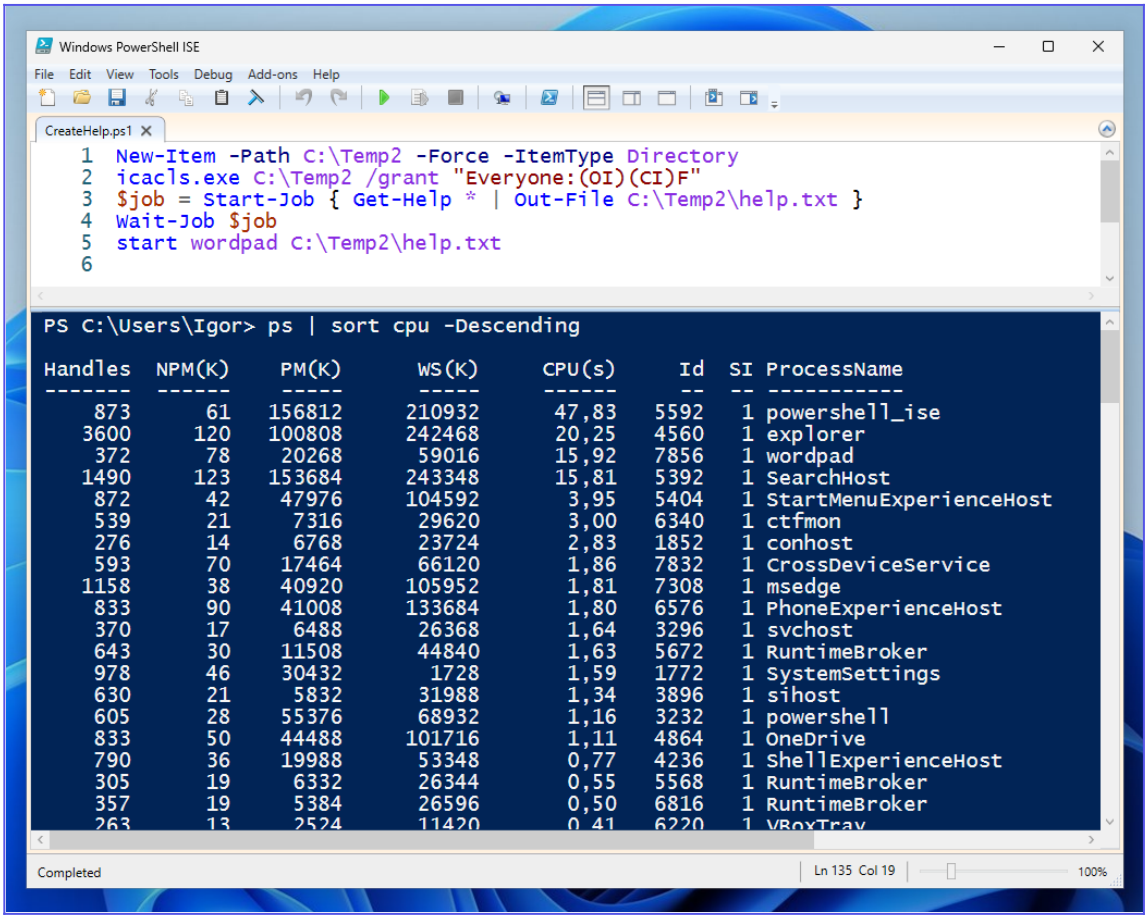


Рис. 5.3. Виведення процесів із сортуванням по завантаженню процесора

Вивести процеси поточного користувача (команда потребує повноважень адміністратору – рис. 5.4):

```
ps -IncludeUserName | Where-Object UserName -m "$env:USERNAME"
```

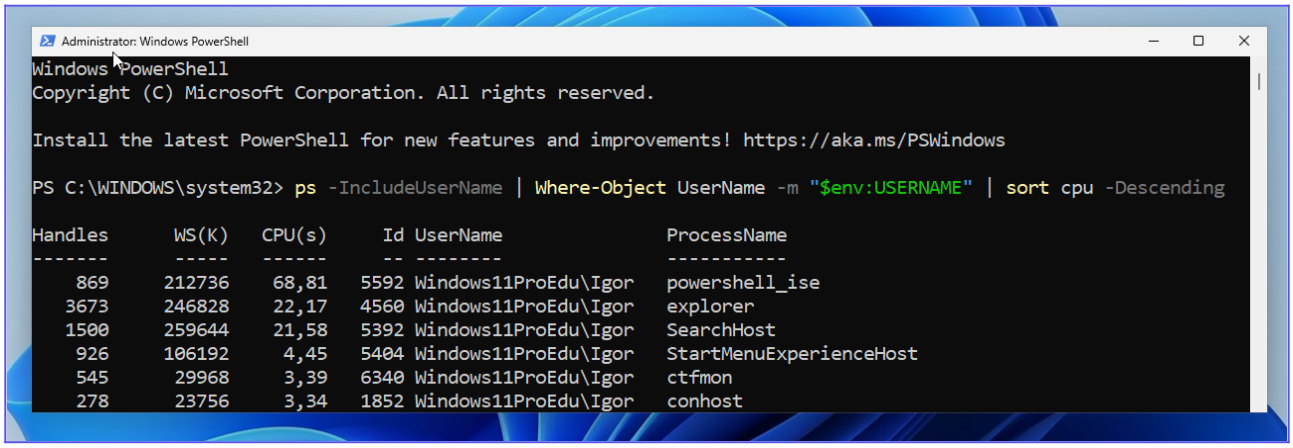


Рис. 5.4. Виведення процесів поточного користувача Igor

За допомогою параметра `-FileVersionInfo` (псевдонім `fvi`) отримують інформацію про версію файлу-процесу і його розміщенні на диску. Наприклад (команда може видавати помилки, якщо не може виконатися для певного процесу):

```
ps -fileversioninfo
```

або

```
ps -fvi
```

Для виведення інформації про процеси з віддаленого комп'ютера, використовують команду з параметром `ComputerName` (псевдонім `cn`). При цьому користувач, що виконує команду, повинен мати відповідний доступом до віддаленого комп'ютера. Наприклад:

```
ps -ComputerName comp23
```

або

```
ps -cn comp23
```

Для отримання дуже докладної інформації про процеси, використовують додатковий командлет `Format-List` (псевдонім `fl`). Наприклад, перша команда перенаправляє у файл *process.txt* докладні відомості про всі завантажені процеси локального комп'ютера, а друга – виводить інформацію тільки про процес *explorer*:

```
ps | Format-List * | Out-File c:\temp\process.txt

ps explorer | fl *
```

При виконанні команди `ps` можна вказувати різні умови, які обмежують виведення інформації. Це досягається, наприклад, за допомогою командлету `Where-Object` (псевдоніми `where` або символ “?”), який створює фільтр. Фільтр визначає, які об’єкти будуть передані по командному конвеєру. Один з прикладів був наведений вище (рис. 5.4). Інший приклад – необхідно вивести всі процеси локального комп’ютера, які споживають в поточний момент часу більше 20 Мбайт пам’яті:

```
ps | ?{$_.WorkingSet -gt 20*1024*1024}
```

або

```
ps | ?{$_.WS -gt 20*1024*1024} | sort WS -Descending
```

де `gt` – означає умову “більше”. Інші можливі варіанти:
`eq` – дорівнює,
`lt` – менше,
`ne` – не дорівнює,
`le` – менше або дорівнює,
`ge` – більше або дорівнює.

Якщо проводиться порівняння рядків та потрібно враховувати перевірку регістра символів, то перед потрібним позначенням операції використовують символ "*c*". Наприклад, наступна команда порівнює два рядки та повертає значення *True* або *False* залежно від результату:

```
# повертає True
"this" -eq "THIS"

# повертає False
"this" -ceq "THIS"
```

Примітка 5.3: в PS коментар в один рядок починається з символу ‘`#`’. Для створення коментаря на декілька рядків, використовують пари з символів: ‘`<#`’ та ‘`#>`’.

Умовні вирази та перехоплення виключень

За допомогою інструкції `if` можна виконувати певні блоки коду, коли задана умова має значення *True*. Можна задавати одну або кілька додаткових умов. Синтаксис інструкції `if` (дужки [ ] позначають необов’язкову частину):

```
if (<умова_1>)
{ <список_інструкцій_1> }
[   elseif (<умова_2>)
    { <список_інструкцій_2> }
]
```

```
[   else
    { <список_інструкцій_3> }
]
```

Приклад 5.1. Виведення повідомлення у разі, якщо значення змінної *a* більше значення 2:

```
$a=3
if ( $a -gt 2 ) { echo "Значення $a більше ніж 2" }
```

Приклад 5.2. Визначення існування вказаного файлу:

```
$f="c:\temp\commands.txt"
if ( test-path -path $f )
{
    echo "Файл $f існує"
}
else
{
    help * > $f
    if ( test-path -path $f ) { echo "Створено файл $f" }
    else { echo "Помилка створення файлу $f" }
}
```

Часто при виконанні будь-яких операцій необхідно реагувати на виняткові ситуації. Наприклад, неможливо створити файл, якщо вказано невірний шлях до нього. У прикладі 5.2 буде видана помилка, якщо каталог *C:\Temp* відсутній. Тому коректніше використовувати також спеціальні конструкції обробки виключень: блоки `try, catch, finally`. Наприклад:

```
$f="c:\temp\commands.txt"
if ( test-path -path $f )
{
    echo "Файл $f існує"
}
else
{
    try
    {
        help * > $f
    }
    catch [System.IO.IOException]
    {
        echo " Помилка створення файлу $f!"
    }
    finally
    {
        if ( test-path -path $f ) { echo " Створено файл $f" }
```

```
    }
}
```

В таблиці 5.1 представлені деякі класи виключень, які часто використовуються для виявлення помилок виконання.

Таблиця 5.1

Приклади класів виключень

Ім'я класу виключення	*Опис*
`System.Exception`	Вказує на помилки, що відбуваються під час виконання програми

`System.ArithmeticException`	Виключення виробляється для помилок арифметичних дій, а також операцій приведення до типу та перетворення.
	Нащадки класу:
	`System.DivideByZeroException`
	`System.NotFiniteNumberException`
	`System.OverflowException`

`System.IO.IOException`	Виключення, яке виникає при помилках введення/виведення.
	Нащадки класу:
	`System.IO.DirectoryNotFoundException`
	`System.IO.EndOfStreamException`
	`System.IO.FileNotFoundException`
	`System.IO.FileLoadException`
	`System.IO.PathTooLongException`

`System.FormatException`	Виключення, яке виникає, якщо формат аргументу не відповідає специфікаціям параметра методу, який викликається
`System.IndexOutOfRangeException`	Виключення виникає при спробі звернення до елемента масиву з індексом, значення якого виходить за межі розміру масиву.

У випадках, коли список альтернативних умов в інструкції `if` великий, вдаються до інструкції `switch`. Вона по черзі зіставляє вираз з кожною умовою. Якщо виявлено збіг, то виконується дія, яка пов'язана із цією умовою. Спрощена форма інструкції `switch`:

```
switch( "string" | number | variable )
{
  "string" | number | variable | {<вираз>}  { <список_інструкцій> }
  [ default { <список_інструкцій> } ]
}
```

Наприклад:

```
$day = "day3"
switch ($day)
{
```

```
    day1 {"Понеділок"; break}
    day2 {"Вівторок"; break}
    day3 {"Середа"; break}
    day4 {"Четвер"; break}
    day5 {"П’ятниця"; break}
    day6 {"Субота"; break}
    day7 {"Неділя"; break}
    default {"Дуже багато днів"}
}
```

Циклічні операції

Досить часто при обробці масивів даних виникає необхідність циклічної обробки. Для цього використовують інструкції `for`, `foreach`, `while`, `do-while`, `do-until`.

Синтаксис інструкції `for`:

```
for ( <ініціалізація>; <умова>; <повторення> )
{ <список_інструкцій> }
```

Синтаксис інструкції `foreach`:

```
foreach ( $<елемент> in $<колекція> )  { <список_інструкцій> }
```

Синтаксис інструкції `while`:

```
while ( <умова> )  { <список_інструкцій> }
```

Синтаксис інструкцій `do-while` (виконується, поки умова залишається істиною) та `do-until` (виконується, поки умова хибна):

```
do { <список_інструкцій> } while ( <умова> )
do { <список_інструкцій> } until ( <умова> )
```

Приклад 5.3. Цикл виведення через пробіл значень від 1 до 10 (без переведення на новий рядок):

```
for( $i=1; $i -le 10; $i++) { Write-Host -NoNewline $i" " }
```

Приклад 5.4. Цикл виведення значень від 1 до 10 з переведенням на новий рядок:

```
for( $i=1; $i -le 10; $i++) { echo $i }
```

Приклад 5.5. Нескінченний цикл виведення у вікно терміналу PS-консолі. Перервати його користувач може натиснувши комбінацію клавіш Crtl+C:

```
for(;;) { echo "i" }
```

Приклад 5.6. Виведення кількості елементів масиву, які розташовані до нульового елементу:

```
$x = 1, 2, 78, 0
$count = $i = 0
do {
    $count++;
    $i++;
} while ($x[$i] -ne 0)
$count
```

Приклад 5.7. Виведення всіх елементів масиву:

```
$data = 1, 2, 78, 0
foreach( $v in $data ) { Write-Host -NoNewline $v" " }
echo ""
```

Приклад 5.8. Скрипт виведення на консоль всіх імен файлів з каталогу *C:\Temp*, чий розмір більше або дорівнює 3 Мбайт:

```
# $max - filter value size in MB
echo "======================================"
$max = 3
$dir = "c:\temp"
Write-Host "Find in directory" $dir
$count = 0
$size = 0
foreach ($file in Get-ChildItem -path $dir)
{
    if ($file.length -ge $max*1024*1024)
    {
      $fname = "File Name: "+$file.Name
      $size += $file.length
      $fsize = ($file.length / 1024 / 1024).ToString("F0")
      $fsize = "File size: "+$fsize+" MB"
      Write-Host ($fname, $fsize, "") -separator "`n"
      $count++
    }
}
Write-Host "Total Founds:"$count
Write-Host "Total size:"($size / 1024 / 1024).ToString("F0") "MB"
echo "======================================"
```

При формуванні рядка в останньому прикладі використовується спецсимвол переведення на новий рядок: "`n".

При виклику методу перетворення `ToString()` використовується формат `"F0"` – виводиться ціла частина значення з плаваючою комою. Більш детальніше про формати з .Net можна почитати, наприклад, за посиланням:

https://learn.microsoft.com/en-us/dotnet/standard/base-types/standard-numeric-format-strings

У PS розпізнаються наступні спеціальні символи (вводяться з урахуванням регістра!):

- `` `0 `` – Null;
- `` `a `` – попередження (звуковий сигнал);
- `` `b `` – повернення курсору;
- `` `f `` – переведення сторінки;
- `` `n `` – новий рядок;
- `` `r `` – повернення каретки;
- `` `t `` – горизонтальна табуляція;
- `` `v `` – вертикальна табуляція.

Об'єкти користувача в PS

Для створення екземпляра об'єкту .NET Framework, в PS використовується командлет `New-Object`. Командлету повідомляється тип об'єкта, який створюється. Наприклад, `PSObject`, який забезпечує узгоджене уявлення будь-якого об'єкту в середовищі PS.

У параметрі `Property` командлету необхідно описати так звану хеш-таблицю – пари "ключ-значення" (скорочений синтаксис – @{}). Пари "ключ-значення", які визначені в хеш-таблиці та передані через параметр `Property`, перетворюються у властивості та значення в новому `PSObject`.

Наприклад, необхідно здійснити виведення у вигляді таблиці інформації про файл, яка містить ім'я та розмір файлу в кілобайтах. Тоді за допомогою командлетів `New-Object` та `Format-Table` це можна реалізувати наступним чином:

```
help * > C:\temp\commands.txt
$f = Get-ChildItem -Path C:\temp\commands.txt
$NewObject = New-Object PsObject -Property @{
      FileName=$f.Name; FileSize=($f.Length/1024).ToString("F0")+" KB" }
$NewObject | Format-Table -auto
```

Спочатку отримуємо інформацію про файл за допомогою командлету `Get-ChildItem`. Потім створюємо новий об'єкт за допомогою командлету `New-Object`, який містить дві рядкових властивості: `FileName` та `FileSize`. Передаємо цим властивостям необхідні значення, виходячи із поставленої

задачі, а потім за допомогою командлету `Format-Table` виводимо значення властивостей у вигляді таблиці у вікно терміналу PS-консолі.

Для спрощення створення об'єкту можна також використовувати командлет `Select-Object`. Наприклад, всі зазначені вище дії можна виконати так:

```
help * > C:\temp\commands.txt
$f = Get-ChildItem -Path C:\temp\commands.txt

$obj = 0 | Select-Object `
    @{name="FileName";expression={$f.Name}},
    @{name="FileSize";expression={($f.Length/1024).ToString("F0")+" KB"}}
$obj | Format-Table -auto
```

Об'єкти, що створюються, можна накопичувати у масив за допомогою операції `+=` .

Ось як буде виглядати описаний вище приклад 5.8 із застосуванням технології створення об'єктів та виведення інформації у табличному вигляді (нові рядки коду виділені жирним курсивом):

```
# $max - filter value size in MB
echo "====================================="
$max = 3
$dir = "c:\temp"
Write-Host "Find in directory" $dir
$count = 0
$size = 0
$NewObject =@()
foreach ($file in Get-ChildItem -path $dir)
{
    if ($file.length -ge $max*1024*1024)
    {
     $fname = "File Name: "+$file.Name
     $size += $file.length
     $fsize = ($file.length / 1024 / 1024).ToString("F0")
     $fsize = "File size: "+$fsize+" MB"
     $NewObject += New-Object PSObject -Property @{FileName=$fname;FileSize=$fsize}
     $count++
    }
}

$NewObject | Format-Table -auto
Write-Host "Total Founds:"$count
Write-Host "Total size:"($size / 1024 / 1024).ToString("F0") "MB"
echo "====================================="
```

Для виведення інформації про методи та властивості об'єкту використовують командлет `Get-Member` (псевдонім `gm`). Наприклад:

```
$f = Get-ChildItem -Path C:\temp\commands.txt
echo "`nDecription File Object:"
$f | Get-Member

$obj = 0 | Select-Object @{n="FileName";e={""}}, @{n="FileSize";e={""}}
echo "`nDecription My Object:"
```

```
$obj | gm
```

Робота з файлами та каталогами

У PS є кілька командлетів, які керують файлами та каталогами. Наприклад:

– `Get-ChildItem` – витягує елементи та їх нащадки із заданого розташування (псевдонім `ls`);

– `Get-Item` – отримує елемент, що знаходиться в заданому місцезнаходженні (псевдонім `gi`);

– `New-Item` – створює новий елемент і задає його значення (псевдонім `ni`);

– `Copy-Item` – копіює елемент з одного місцезнаходження до іншого всередині одного простору імен (псевдонім `cp`);

– `Move-Item` – переміщує елемент з одного місцезнаходження до іншого (псевдонім `mv`);

– `Remove-Item` – видаляє задані елементи (псевдонім `rm`);

– `Rename-Item` – перейменовує елемент в просторі імен постачальника PS (псевдонім `ren`);

– `Get-Content` – витягує вміст елементу, що знаходиться за заданим місцезнаходженням, наприклад, текст з файлу (псевдонім `cat`).

Приклад 5.9: створення в каталозі *C:\Temp* файлів *testfile1.txt, testfile2.txt* та каталогу *scripts*. Другий текстовий файл містить короткий текст. Після створення здійснюється виведення у вікно терміналу PS-консолі вмісту каталогу та файлу з текстом:

```
clear
if (test-path c:\temp\testf*.txt) { rm c:\temp\testf*.txt }
if (test-path c:\temp\scripts) { rm c:\temp\scripts }
new-item -path c:\temp -name testfile1.txt -type "file"
new-item c:\temp\testfile2.txt -type "file" -value "This is a text string."
new-item -type directory -path c:\temp\scripts
ls c:\temp -include testf*.txt, c:\temp\sc*
cat c:\temp\testfile2.txt
```

Різниця між командлетами `Get-Item` та `Get-ChildItem`: `Get-Item` не має параметра `Recurse`, так як він витягує тільки елемент, а не його вміст. Для рекурсивного вилучення вмісту елементу використовується командлет `Get-ChildItem`.

Якщо необхідно провести в текстовому файлі пошук рядків за певним шаблоном, то використовують командлет `Select-String`. Наприклад, нехай

необхідно вивести на екран тільки ті рядки із файлу *C:\Temp\commands.txt*, в яких міститься слово *about*:

```
ls C:\temp\commands.txt | Select-String about
```

або

```
ls C:\temp\commands.txt | Select-String -pattern "about"
```

Ускладнимо завдання. Нехай, наприклад, необхідно сформувати масив, із номерами знайдених рядків у вказаному файлі. Це можна виконати, наприклад, наступним чином:

```
$lines = (ls C:\temp\commands.txt | Select-String -pattern "about")
$mas = 0..($lines.length-1)
for($i = 0; $i -le $lines.length-1; $i++) { $mas[$i] = $lines[$i].LineNumber }
Write-Host $mas
```

Використання математичних функцій

Для використання в PS апарату математичних функцій частіше використовують клас `Math` з простору імен `System` бібліотеки .NET Framework. Наприклад, в класі існують такі методи:

– *Abs*(Decimal), *Abs*(Double), *Abs*(Int16), *Abs*(Int32), *Abs*(Int64), *Abs*(SByte), *Abs*(Single) – повертають абсолютну величину значення відповідного типу;

– *Acos*(double), *Asin*(double), *Atan*(double), *Atan2*(double y, double x), *Cos*(double), *Cosh*(double), *Sin*(double), *Sinh*(double), *Tan*(double), *Tanh*(double) – тригонометричні функції;

– *Pow*(double x, double y) – зведення значення *x* в ступінь *y*;

– *Sqrt*(double) – корінь квадратний числа;

– *Log*(double) – натуральний логарифм (логарифм за основою *e*);

– *Log*(double a, double newBase) – обчислення логарифма за основою *newBase* числа *a*;

– *Log10*(double) – обчислення логарифма за основою 10.

– *Exp*(double d) – зведення *e* в ступінь *d*.

У класі існують перевантажені методи *Min*, *Max*, *Round*, *Floor*, *Truncate*. Існують і інші методи.

В клас `Math` введені дві константи:

– `E` (2.7182818284590452354);

– `PI` (3.14159265358979323846).

Таким чином, клас `Math` містить досить багато математичних методів для розрахунку будь-яких складних математичних виразів. У PS це виглядає, наприклад, так:

```
$Num = 49.68124
$res1=[System.Math]::Round($Num, 2)
$res2=[Math]::Truncate($Num)
$res3=[Math]::Sqrt($Num)
echo ""
$res1, $res2, $res3
```

Практична частина

Створити в каталозі користувача PS-скрипт, що виконує наступні дії.

1. У домашньому каталозі користувача створює каталог *Data*.
2. В каталозі *Data* формує файл *commands.txt* з переліком всіх команд PS.
3. Робить вибірку з файлу *commands.txt* всіх частин *about*-інструкцій та формує в каталозі *Data* окремі файли з повним описом кожної *about*-інструкції. При вибірці рядків використовувати властивість `Line` та її метод `Split` із передачею в якості параметру символу-роздільника, наприклад, пробілу.

В каталозі користувача створити другий скрипт для обчислення математичного виразу відповідно до варіанту завдання, яке було виданого викладачем (таблиці 5.2 – 5.4).

У звіт включити коди написаних PS-скриптів з коментарями, а також результати їх роботи.

Вміти дати відповіді викладачу на наступні питання.

1. Яким чином визначити версію Windows PowerShell?
2. Що таке командлет?
3. Як викликати довідку по командлету або інструкції?
4. Які умови необхідні для виконання скриптів на Windows PowerShell?
5. Як отримати відомості про завантажені процеси?
6. Які в PS існують інструкції умов?
7. Як побудувати умовні вирази на PS?
8. Коли і як необхідно застосовувати інструкцію `switch`?
9. Як обробляти виняткові ситуації?
10. Які існують інструкції для побудови циклів? Привести приклади.
11. Які бувають спецсимволи в PS?
12. Як здійснювати виведення кількох рядків через роздільник?
13. Як здійснювати виведення даних в табличному вигляді?
14. Як створювати свої об’єкти .NET Framework в PS?
15. Які відомі командлети для керування файлами та каталогами?
16. Як викликати математичні функції в PS-скриптах?

Таблиця 5.2

Варіанти завдань

Варіант завдання	*Варіант виразу*	*Варіант вхідних даних*	*Варіант завдання*	*Варіант виразу*	*Варіант вхідних даних*
1	1	1	11	1	2
2	2	2	12	2	1
3	3	1	13	3	2
4	4	2	14	4	1
5	5	1	15	5	2
6	6	2	16	6	1
7	7	1	17	7	2
8	8	2	18	8	1
9	9	1	19	9	2
10	10	2	20	10	1

Таблиця 5.3

Вхідні дані

Варіант вхідних даних	a_i
1 (тип даних int)	5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55
2 (тип даних double)	0.3, 0.7, 0.9, 1.3, 1.7, 1.9, 2.3, 2.7, 2.9, 3.3, 3.7

Таблиця 5.4

Варіанти математичних виразів

Варіант	*Математичний вираз*
1	$S=\dfrac{\sum_{i=1}^{5}\sin\lvert 1-\ln a_i\rvert}{\sum_{i=1}^{11}\sin^2 18\,a_i^3}$
2	$S=\dfrac{\sum_{i=1}^{11}\sin\left(1-3*\sin^3 a_i\right)}{3*\sum_{i=1}^{11}\cos^2 a_i^5}$

3	$S=\frac{\sum_{i=1}^{11}\cos\lvert 1-\ln^2 a_i\rvert}{\sum_{i=1}^{11}\sin^2 18\,a_i^{a_i}}$
4	$S=\frac{\ln\left[\sum_{i=1}^{11}\left(\lvert\sin a_i\rvert^{\sin a_i}+\lvert\cos a_i\rvert^{\cos a_i}\right)\right]}{3*\sum_{i=1}^{11}\sin^2 a_i^{\sin a_i}}$
5	$S=\frac{\sin^2\left[\sum_{i=1}^{11}\ln\left(\lvert 1-\cos^2 a_i\rvert\right)\right]}{\sum_{i=1}^{11}\cos\left(\lvert 1-\sin a_i\rvert\right)}$
6	$S=\sum_{i=1}^{11}\frac{1-2\cos^2 a_i}{\lvert\sin a_i\rvert*\lvert\cos a_i\rvert}$
7	$S=\sum_{i=1}^{11}\left(1+\cos a_i+\cos\frac{a_i}{2}\right)$
8	$S=\sum_{i=1}^{11}\frac{\cos 2a_i}{1-\sin 2a_i}$
9	$S=\sqrt{\frac{\sum_{i=1}^{11}\left(a_i-\tilde{a}\right)^2}{10}},\tilde{a}=\frac{1}{11}\sum_{i=1}^{11}a_i$
10	$S=\sum_{i=1}^{11}\left(1+\sin a_i+\sin\frac{a_i}{2}\right)$

Робота №6.
Створення процесів в GNU/Linux-сумісних ОС

Мета роботи: вивчити особливості створення процесів в GNU/Linux-сумісних ОС.

Теоретична частина

Найпростішим способом породити процес в GNU/Linux-сумісній ОС є виклик бібліотечної функції `system()`. Ця функція передає команду в оболонку */bin/sh*. Функція оголошена в файлі *stdlib.h* наступним чином:

```
int system(const char* command);
```

Функція повертає статус завершення процесу нащадка – ціле число, що містить код повернення та деяку іншу інформацію про те, як завершився процес. Для з'ясування цієї інформації існують спеціальні макроси:

– `WIFEXITED()` – повертає ненульове значення, якщо нащадок завершиться за допомогою повернення з функції *main( )* або через виклик *exit( )*;

– `WEXITSTATUS()` – повертає код завершення процесу. Цей макрос викликається в тому випадку, якщо `WIFEXITED()` повернув ненульове значення;

– `WIFSIGNALED()` – повертає ненульове значення, якщо процес був завершений за допомогою отримання сигналу;

– `WTERMSIG()`, `WCOREDUMP()`, `WIFSTOPPED()`, `WSTOPSIG()` та `WCONTINUED()` – мають відношення до сигналів.

Сигнал – це системне повідомлення, яке один процес направляє іншому. При цьому процес-приймач може або проігнорувати сигнал та продовжити своє виконання, або перервати всі свої дії і негайно почати обробляти отриманий сигнал. Деякі типи сигналів призначені для негайного завершення процесу.

Наступний приклад демонструє виведення на консоль відомостей про ОС командою `uname`, при цьому використовується ключ *-a*, що дозволяє отримати всі відомості.

```
#include <stdlib.h>
int main()
{
    system("uname -a");
    return 0;
}
```

У ряді випадків функція `system()` безумовно корисна, однак повноцінне керування процесами в GNU/Linux-сумісних здійснюється за допомогою системних викликів.

При роботі з процесами часто буває необхідно знати їх ідентифікатори. Отримання PID, PPID та UID поточного процесу можливо за допомогою наступних системних викликів:

```
pid_t getpid(void);
pid_t getppid(void);
uid_t getuid(void);
```

Для їх використання в код програми необхідно включити header-файли *unistd.h* та *sys/types.h*.

Якщо необхідно отримати ім'я користувача за його ідентифікатором UID, то застосовується бібліотечна функція `getpwuid()`, яка оголошена в файлі *pwd.h* наступним чином:

```
struct passwd* getpwuid(uid_t UID);
```

Після її успішного виклику, ім'я користувача буде зберігатися в полі `char* pw_name` структури `passwd`. Якщо UID не відповідає жодному користувачеві системи, то функція `getpwuid()` поверне `NULL`.

Типи даних `pid_t` та `uid_t` є цілими числами, розмірність яких залежить від реалізації.

Приклад отримання PID, PPID, UID та імені користувача в програмі:

```
#include <sys/types.h>
#include <unistd.h>
#include <pwd.h>
#include <stdio.h>

int main()
{
   struct passwd* pwdp = getpwuid(getuid());
   if (pwdp != NULL)
   {
      printf("PID:      %d\n", getpid());
      printf("PPID:     %d\n", getppid());
      printf("UID:      %d\n", getuid());
      printf("Username: %s\n", pwdp->pw_name);
   }
}
```

Для породження нового процесу в Unix-подібних ОС призначений системний виклик `fork()`, який оголошений в *unistd.h* наступним чином:

```
pid_t fork(void);
```

Він породжує процес методом “клонування”. Це означає, що новий процес є точною копією свого батька та виконує ту ж саму програму.

Якщо необхідно, щоб батьківська програма та програма-нащадок, наприклад, виводили на консоль різну інформацію, в тому числі й стан деякої змінної, то можна скористатися наступним прикладом:

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main()
{
      int a = 0;
      printf("Execute parent...\n");
      pid_t pid = fork();

      if (-1 == pid)
      {
          perror("FORK");
          return 1;
      }

      if (!pid)
      {
          printf("FORK. Execute child...\n");
          a += 5;
          printf("CHILD: A = %d\n", a);
          exit(0);
      }

      int status = 0;
      wait(&status);   // або в даному прикладі можна так: wait(NULL);

      printf("Continue execute parent...\n");
      a += 10;
      printf("PARENT: A = %d\n", a);

      return 0;
}
```

Послідовність виконання цієї програми така.

1. Запускається основна (батьківська) програма.

2. Виконується `fork()`, що призведе до клонування батьківського процесу.

3. Аналізується результат повернення `fork()` – якщо він дорівнює -1, то виникла якась системна помилка, яку виведе функція `perror()` та програма завершиться.

4. Якщо результат повернення `fork()` дорівнює нулю, то це означає, що програма виконується за клонованим сценарієм (тобто це процес-нащадок). Код завершення процесу-нащадку передаємо через функцію `exit()`.

5. Далі весь код може бути віднесений до батьківського процесу. За допомогою системного виклику `wait()` очікуємо на завершення процесу-нащадка і після цього виконуємо далі батьківський процес.

Схема з `fork()` може бути й інша. Наприклад:

```
. . .
pid_t pid = fork();
if (-1 == pid)
{
  ...
  return 1;
}

if (!pid)
{
   ... // child
}
else
{
   ... // parent
}
...
```

Або скорочена схема:

```
...
if (!fork()) {
    ... // child
}
... // parent
```

Системний виклик `wait()` оголошений в файлі *sys/wait.h* наступним чином:

```
pid_t wait(int* wstatus);
```

Для аналізу системний виклик `wait()` повертає ідентифікатор потомку, що завершився, а також через покажчик повертає статус його завершення.

У стандартній бібліотеці мови С є спеціальні функції, які називаються сімейством `exec`. Ці функції оголошені в файлі *unistd.h* наступним чином:

```
int execl(const char* path, const char* arg, ...);
int execlp(const char* file, const char* arg, ...);
int execle(const char* path, const char* arg, ..., char* const envp[]);
int execv(const char* path, char* const argv[]);
int execvp(const char* file, char* const argv[]);
int execvpe(const char* file, char* const argv[], char* const envp[]);
```

Ім’я функції формується додаванням до *exec* обов’язкового наступного символу та необов’язкового наступного за ним другого символу. Призначення символів в *exec*-імені представлено в таблиці 6.1.

exec-виклики замінюють поточний образ процесу новим, за допомогою завантаження в пам’ять програми, на яку вказує перший аргумент.

Таблиця 6.1

Призначення символів в імені *exec*-функції

Символ	*Наявність або відсутність*	*Опис*
`l`	+	Аргументи програми передаються не у вигляді масиву `argv`, а у вигляді окремих аргументів *exec*-функції, список яких закінчується аргументом `NULL`.
	–	Набір аргументів для програми, що передається у вигляді масиву рядків `argv`.
`v`	+	Аргументи програми передаються у вигляді єдиного масиву рядків `argv`, останнім елементом якого є `NULL`.
	–	Замість «`v`» присутній символ «`l`», який задає список програми у вигляді окремих аргументів-рядків, список яких закінчується аргументом `NULL`.
`e`	+	В якості останнього аргументу використовується масив рядків `envp` (оточення майбутньої програми).
	–	Початковим оточенням для програми, яка запускається, буде оточення, що пов’язане з поточною діяльністю.
`p`	+	В якості першого аргументу використовується ім’я файлу програми, пошук якого буде проводитися в каталогах, які перелічені у змінній оточення `PATH`.
	–	В якості першого аргументу виступає повний шлях до файлу програми (відносно поточного та кореневого каталогу).

Наступний приклад демонструє запуск процесу з подальшим його очікуванням та обробкою статусу завершення.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>

int main()
{
  pid_t pid = fork();
  if (!pid) execlp("ls", "ls", "-l", "/", NULL);

  int exit_status = 0;
```

```
  pid_t childpid = wait(&exit_status);

  if (WIFEXITED(exit_status))

      printf("Процес ls с PID=%d завершився з кодом %d\n",
             childpid, WEXITSTATUS(exit_status));

  if (WIFSIGNALED(exit_status))
      printf("Процес ls с PID=%d завершився по сигналу %d\n",
             childpid, WEXITSTATUS(exit_status));

  return 0;
}
```

Спрощений варіант запуску процесу з очікуванням може бути представлений у вигляді функції:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

void execute(const char* command, const char * const args[]);

int main()
{
    const char * const args[] = {"ls", "-l", "/", nullptr};
    execute("/bin/ls", args);
    printf("Процес завершено\n");
    return 0;
}

void execute(const char* command, const char * const args[])
{
    if (!fork()) execv(command, const_cast<char* const*>(args));
    wait(nullptr);
}
```

В останньому прикладі показано, яким чином треба обходити сповіщення компілятора про використання покажчиків та рядків. Це стосується версій C++ починаючи зі стандарту C++11.

На мові C рядкові літерали мають тип `char[]` і можуть бути присвоєні безпосередньо (неконстантному) `char*`. C++03 також це дозволяє, але є застарілим. C++11 більше не дозволяє такі призначення без приведення типів. Як наслідок, в коді присутня конструкція `const_cast<>` для вирішення питання передачі оголошеного константного масиву до `execv()`.

Примітка 6.1: якщо, з яких-небудь причин, розробнику потрібно призупинити виконання обчислювального потоку, то використовують функцію `sleep()`, яка оголошена в файлі *unistd.h*. Функція приймає значення в секундах та затримує виконання.

Практична частина

В GNU/Linux-сумісній ОС (наприклад, Debian або Ubuntu Desktop), розробити мовою C++ програму-меню, яка надає користувачеві простий варіант запуску нового процесу та виконання дій відповідно до варіанту з таблиці 6.2. Варіант видається викладачем.

Запуск нового процесу повинен відбуватися для пунктів меню 1-3 з використанням функцій *fork( )*, *exec...( )*, *system( )* відповідно.

Різновид *exec*-функції для виконання завдання конкретизує викладач.

При виконанні завдання використовувати знання, які отримані з лекцій, лабораторних робіт № 2 – 4, а також довідникової літератури.

У звіт включити код програми з коментарями та результати роботи з поясненнями.

Таблиця 6.2

Варіанти виконання завдання у роботі №6

Варіант	*Пункти меню*
1	1. Вміст поточного каталогу 2. Інформація про стан оперативної пам’яті 3. Інформація про запущені процеси 4. Вихід
2	1. Вміст каталогу, зазначеного користувачем 2. Інформація про вільний дисковий простір файлових систем 3. Назва версії ОС 4. Вихід
3	1. Список змонтованих файлових систем 2. Останні системні повідомлення 3. Версія ядра ОС 4. Вихід
4	1. Список файлових систем, які підтримуються 2. Зміст bash-профайлу користувача 3. Назва процесора 4. Вихід
5	1. Список користувачів, що працюють з системою в поточний момент 2. Редактор nano 3. Редагувати профайл користувача 4. Вихід
6	1. Моніторинг запущених процесів 2. Завершення певного процесу 3. Інтерактивний перегляд файлу 4. Вихід
7	1. Лістинг каталогу тимчасових файлів 2. Файловий менеджер MC

	3. Список монтованих файлових систем 4. Вихід
8	1. Інформація про процес реєстрації користувача в системі 2. Редактор nano 3. Створення backup-копії bash-профайлу користувача 4. Вихід
9	1. Інформація про системне оточення 2. Інформація про зареєстровані шляхи пошуку 3. Інформація про мовні локальні налаштування консолі 4. Вихід
10	1. Створення backup-копії файлів конфігурації користувача 2. Список користувачів системи 3. Список груп користувачів 4. Вихід
11	1. Зміна прав доступу до файлу/каталогу 2. Перейменування файлу/каталогу 3. Компіляція cpp-файлу 4. Вихід
12	1. Вивести процеси-демони поточного користувача 2. Запуск програми 3. Довідка по команді монтування 4. Вихід

Робота №7.
Створення процесів в MS Windows

Мета роботи: вивчити особливості створення процесів в середовищі MS Windows.

Теоретична частина[1]

Для створення нового процесу в MS Windows використовуються три API-функції: `WinExec()`, `ShellExecute()`, `CreateProcess()`. Прототипи функцій наступні (функції із суфіксом “W” – це функції з підтримкою Unicode):

```
UINT WinExec(LPCSTR lpCmdLine, UINT uCmdShow);

HINSTANCE ShellExecuteW(HWND hwnd, LPCWSTR lpOperation,
      LPCWSTR lpFile, LPCWSTR lpParameters,
      LPCWSTR lpDirectory, INT nShowCmd);

BOOL CreateProcessW(LPCWSTR lpApplicationName,
      LPWSTR lpCommandLine,
      LPSECURITY_ATTRIBUTES lpProcessAttributes,
      LPSECURITY_ATTRIBUTES lpThreadAttributes,
      BOOL bInheritHandles, DWORD dwCreationFlags,
      LPVOID lpEnvironment, LPCWSTR lpCurrentDirectory,
      LPSTARTUPINFOW lpStartupInfo,
      LPPROCESS_INFORMATION lpProcessInformation);
```

Для створення нового процесу використовується стандартний системний виклик `CreateProcessW()`. Щоб запустити нову програму на виконання, простіше використовувати функцію `WinExec()`. Але вона є застарілою та не рекомендується Microsoft до використання!

Фактично всередині `WinExec()` відбувається звернення до функції `LoadModule()` (ще один застарілий системний виклик). Вона, в свою чергу, викликає `CreateProcess()` із значеннями аргументів за замовчуванням.

Функція `WinExec()` є найпростішим інструментом створення нового процесу. При зверненні до неї необхідно повідомити ім’я програми (параметр *lpCmdLine* – повний шлях або коротке ім’я файлу для виконання, розташованого в шляху пошуку за системною змінною PATH), а також спосіб відображення вікна програми (параметр *uCmdShow* – наприклад, одна з констант SW_SHOW, SW_HIDE та ін.). Програма може проігнорувати цей аргумент та запуститися так, як вона налаштована.

[1] В описі теоретичної частини в певних абзацах цитується матеріал з книги: Al Williams. Windows 2000 Systems Programming Black Book. – Coriolis Group Books, 2000. – 615 p. ISBN: 1576102807, 9781576102800.

У випадку якщо сталася помилка, функція `WinExec()` повертає ціле беззнакове значення, яке менше 32. Якщо програма успішно запущена, то функція повертає *дескриптор* нової програми (який не може бути менше 32).

Примітка 7.1: в документації Microsoft зазначено, що функція `WinExec()` надається у сучасному складі Win32 API тільки для сумісності з 16-тирозрядними версіями MS Windows. Окрім того, Microsoft пропонує використовувати замість неї саме функцію `CreateProcess()` або `CreateProcessW()`. Тобто не варто сподіватися, що вона буде відпрацьовувати вірно в сучасних ОС MS Windows. За документацією, після запуску нової програми, функція `WinExec()` **негайно передає керування** програмі, з якої був здійснений її виклик, тобто вона ***не очікує*** моменту, поки запущена програма завершить роботу. Але у версії MS Windows 11 це може бути невірно! Тобто можна бачити очікування завершення роботи запущеного процесу і тільки потім буде продовжено виконання основної програми! Цей ефект був підтверджений у версіях 10.0.22621.3527 та 10.0.22621.3880 MS Windows 11. У версії 10.0.17763.379 ця функція працювала ще вірно за своїм сценарієм. Поточна версія виводиться у вікно терміналу консолі при його старті або командою `ver`. Значення на кшталт 22621.3527 у номері версії є номером збірки (OS Build).

Примітка 7.2: номер версії MS Windows можна дізнатися за посланням:

https://learn.microsoft.com/en-us/windows/win32/sysinfo/operating-system-version

Дізнатися номери збірок різних версій MS Windows, можна за відповідними посиланнями зі сторінки, яка розміщена за адресою:

https://learn.microsoft.com/en-us/windows/release-health/

Іншим простим способом, що використовується для запуску програм, є системний виклик `ShellExecuteW()`. Цей виклик багато в чому нагадує `WinExec()`, однак він підтримує обробку типів файлів, зареєстрованих графічною оболонкою MS Windows. Наприклад, якщо за допомогою `ShellExecuteW()` спробувати запустити файл із розширенням **.txt**, то буде запущена програма Notepad або будь-яка інша програма, яка використовується в системі для перегляду текстових файлів.

В якості аргументів, функція `ShellExecuteW()` приймає дескриптор вікна (параметр *hwnd*) та операційний рядок (параметр *lpOperation*), такий як `"open"` (відкрити), `"print"` (роздрукувати) або `"explore"` (дослідити), або `NULL`, що за замовчуванням призведе до тієї ж дії, що і рядок `"open"`. Також функції необхідно повідомити ім’я файлу (параметр *lpFile*) та будь-які параметри командного рядка або `NULL` для них (параметр *lpParameters*). Останні два аргументи – це поточний каталог (параметр *lpDirectory*) та константа функції

ShowWindow() (параметр *nShowCmd*). Значення, що повертається функцією, таке ж, як і для WinExec(), але іншого типу (див. приклад).

Для використання у власній програмі функції WinExec() досить включення файлу *windows.h*, а для використання ShellExecuteW() – файлу *shellapi.h*.

Приклад програми OpenTextFile, яка призводить до створення нового процесу для відкриття текстового файлу:

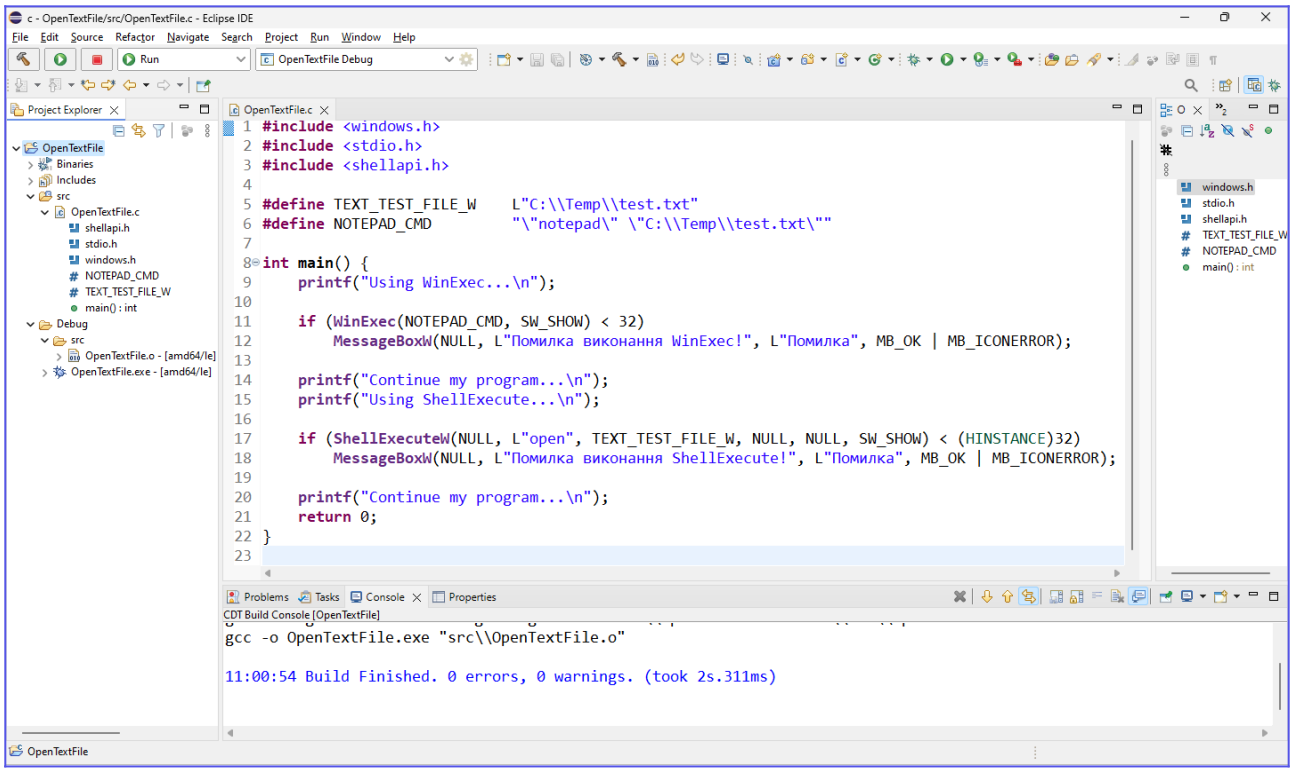


Рис. 7.1. Приклад використання застарілої функції WinExec() та функції ShellExecuteW()

Виклики ShellExecuteW() та WinExec() дуже зручно використовувати для виконання найпростіших дій, таких як відкриття файлів або запуск програм без очікування їхнього завершення. Якщо ж необхідно створити новий процес, використовуючи при цьому деякі додаткові можливості, то не обійтися без системного виклику CreateProcessW(). Опис аргументів, прийнятих цим викликом, наведено в таблиці 7.1.

Поля структури STARTUPINFOW (https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/ns-processthreadsapi-startupinfow) містять заголовок консолі, початковий розмір та позицію нового вікна, направлення стандартних потоків введення/виведення.

Якщо немає потреби використовувати які-небудь із цих полів, то тоді необхідно передати функції CreateProcessW() коректний вказівник на

порожню структуру STARTUPINFOW. При цьому необхідно вказати розмір структури в полі cd та привласнити полю dwFlags значення 0.

Таблиця 7.1

Опис аргументів системного виклику CreateProcessW()

Аргумент	*Опис*
lpApplicationName	Ім'я програми (або NULL, якщо ім'я програми зазначене в командному рядку)
lpCommandLine	Командний рядок
lpProcessAttributes	Атрибути безпеки для дескриптора процесу, що повертаються функцією
lpThreadAttributes	Атрибути безпеки для дескриптора потоку, що повертаються функцією
bInheritHandles	Вказує, чи успадковує новий процес дескриптори, які належать поточному процесу
dwCreationFlags	Параметри створення процесу
lpEnvironment	Значення змінних оточення (або NULL, якщо успадковується поточне оточення)
lpCurrentDirectory	Поточний каталог (або NULL, якщо використовується поточний каталог поточного процесу)
lpStartupInfo	Структура STARTUPINFOW, яка містить інформацію про запуск процесу
lpProcessInformation	Дескриптори, що повертаються функцією та ідентифікатори (ID) процесу і потоку

Параметр lpProcessInformation повертає вказівник на структуру типу PROCESS_INFORMATION (https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/ns-processthreadsapi-process_information), в якій містяться ідентифікатор та дескриптор нового процесу, а також ідентифікатор та дескриптор першого потоку, що належить новому процесу. Ці відомості можуть використовуватися для того, щоб повідомити про новий процес іншим програмам, а також для того, щоб контролювати новий процес.

При створенні процесу з використанням виклику CreateProcessW(), можна передати новому процесу у спадщину деякі ресурси, зокрема дескриптори відкритих файлів. Це буває корисно, коли необхідно об'єднати стандартні потоки введення/виведення декількох консольних програм.

Маючи дескриптор процесу, можна управляти процесом за допомогою системних викликів управління. Деякі з них наступні:

– CreateRemoteThread() – створює потік в іншому процесі;

– GetExitCodeProcess() – повертає код завершення процесу;

– SetProcessPriorityBoost() – дозволяє або забороняє MS Windows динамічно змінювати пріоритет процесу;

– GetProcessPriorityBoost() – повертає статус зміни пріоритету процесу;

– TerminateProcess() – коректне завершення роботи процесу;

– ExitProcess() – негайне завершення процесу;

– `GetProcessTimes()` – повертає ступінь використання CPU процесом;

– `GetStartUpInfoW()` – повертає структуру `STARTUPINFOW`, яка передана процесу при звернені до `CreateProcessW()`.

Щоб визначити момент завершення процесу, можна скористатися одним з декількох способів. По-перше, можна використовувати виклик `GetExitCodeProcess()`. Цей виклик повертає або значення `STILL_ACTIVE` (якщо процес усе ще продовжує роботу), або код завершення процесу (якщо процес завершений).

Дізнатися ***дескриптор*** **поточного процесу** можна за допомогою функції `GetCurrentProcess()`.

Дізнатися ***ідентифікатор*** **поточного процесу** можна за допомогою функції `GetCurrentProcessId()`.

Щоб управляти процесом з іншого процесу, необхідно звернутися до функції `OpenProcess()`, якій необхідно передати ідентифікатор ID-процесу. Щоб відкрити процес для доступу, потрібно мати необхідні дозволи на доступ до процесу. Процес, що створює новий процес за допомогою функції `CreateProcess()`, вже має дескриптор нового процесу (цей дескриптор міститься в структурі `PROCESS_INFORMATION`).

Ще один спосіб визначення поточного стану процесу – використання функції `WaitForSingleObject()` або `WaitForSingleObjectEx()`. Основне призначення цих викликів – визначити, чи перебуває деякий дескриптор у ***сигнальному стані***. Дескриптор процесу переходить у сигнальний стан тоді, коли процес завершує свою роботу.

При звернені до функції `WaitForSingleObject()` необхідно вказати дескриптор процесу та інтервал часу в *мс*. Якщо інтервал дорівнює 0, функція завершує роботу негайно, повертаючи поточний стан процесу. Якщо інтервал часу дорівнює константі `INFINITE`, то функція **буде чекати** доти, поки процес, що цікавить, **не завершить роботу**.

Щоб визначити стан процесу, необхідно порівняти значення, що повернула функція `WaitForSingleObject()`, із значеннями `WAIT_OBJECT_0` (сигнальний стан) або `WAIT_TIMEOUT` (процес продовжує функціонувати). У випадку помилки функція поверне значення `WAIT_FAILED`.

У наступному прикладі[2] наведені дві простих консольних програми. Перша програма ParentProcess (рис. 7.2) запускає другу програму ChildProcess (рис. 7.3) та переходить у режим очікування. Програма ChildProcess читає ідентифікатор процесу (PID, *Process Identifier*) програми, яка її запустила з командного рядку та очікує завершення роботи програми ParentProcess. У

[2] Приклади побудовані із використанням додаткових прикладів із книги: Al Williams. Windows 2000 Systems Programming Black Book. – Coriolis Group Books, 2000. – 615 p. ISBN: 1576102807, 9781576102800.

командному рядку програми ParentProcess можна вказати повний шлях до файлу програми ChildProcess (рис. 7.4).

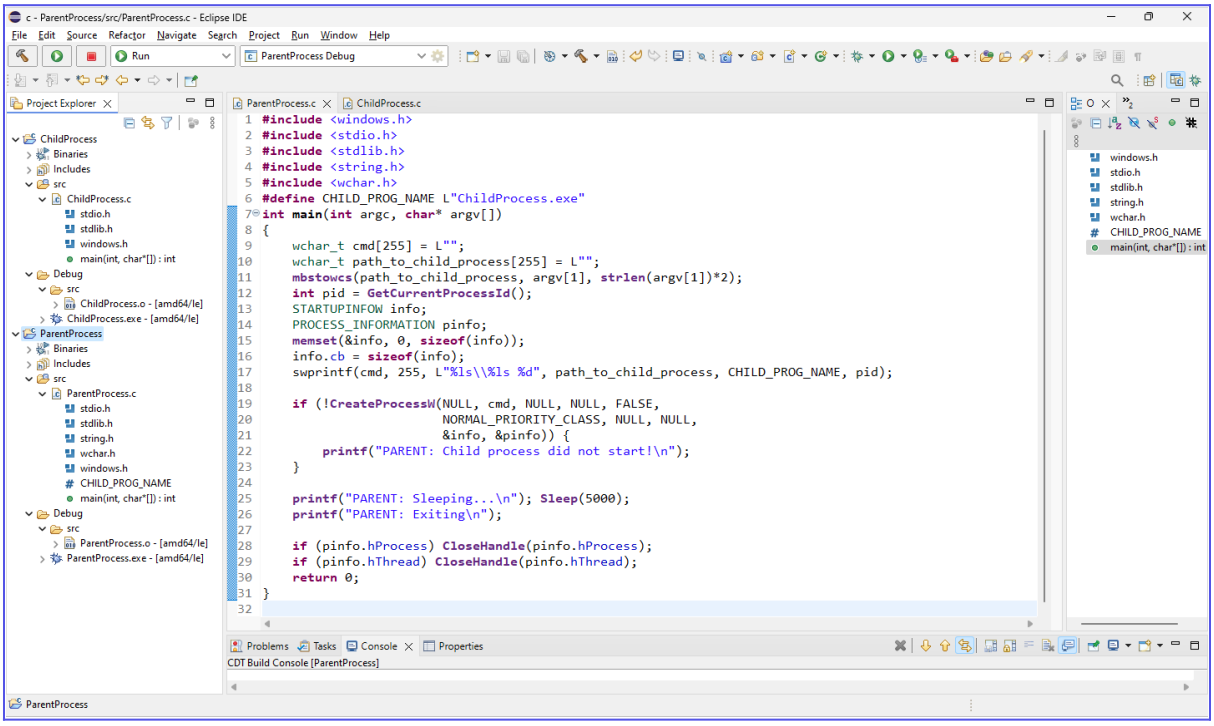


Рис. 7.2. Код програми ParentProcess

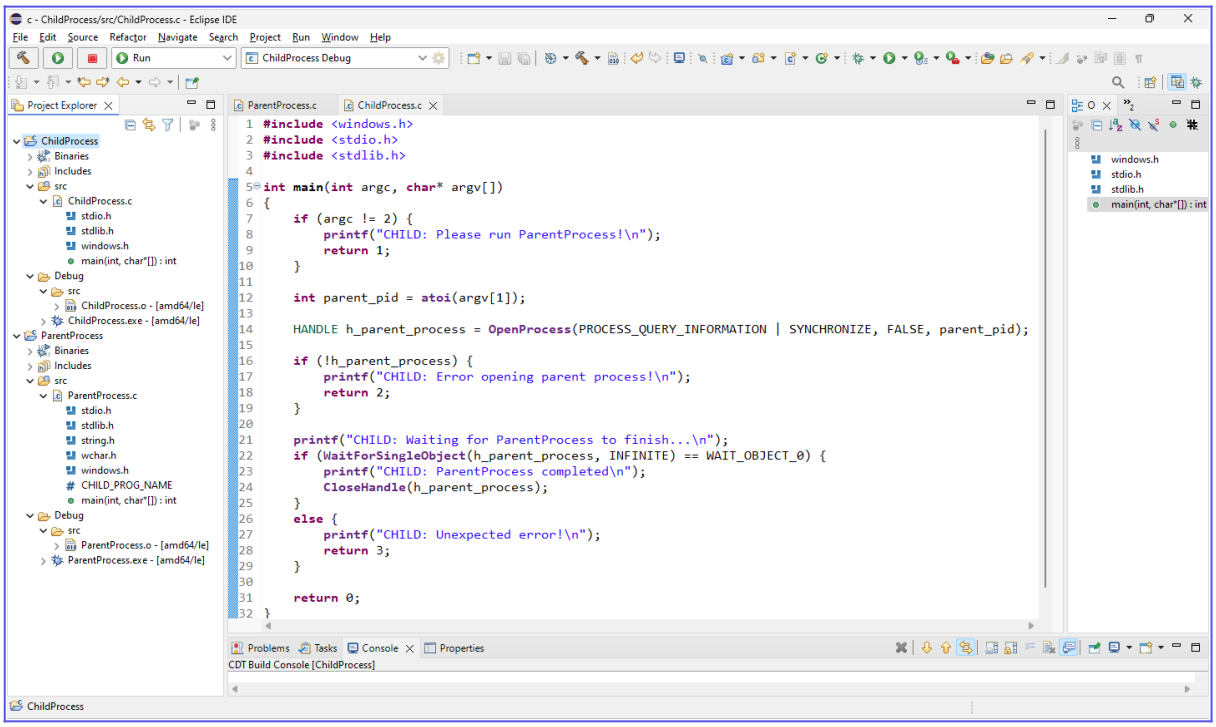


Рис. 7.3. Код програми ChildProcess

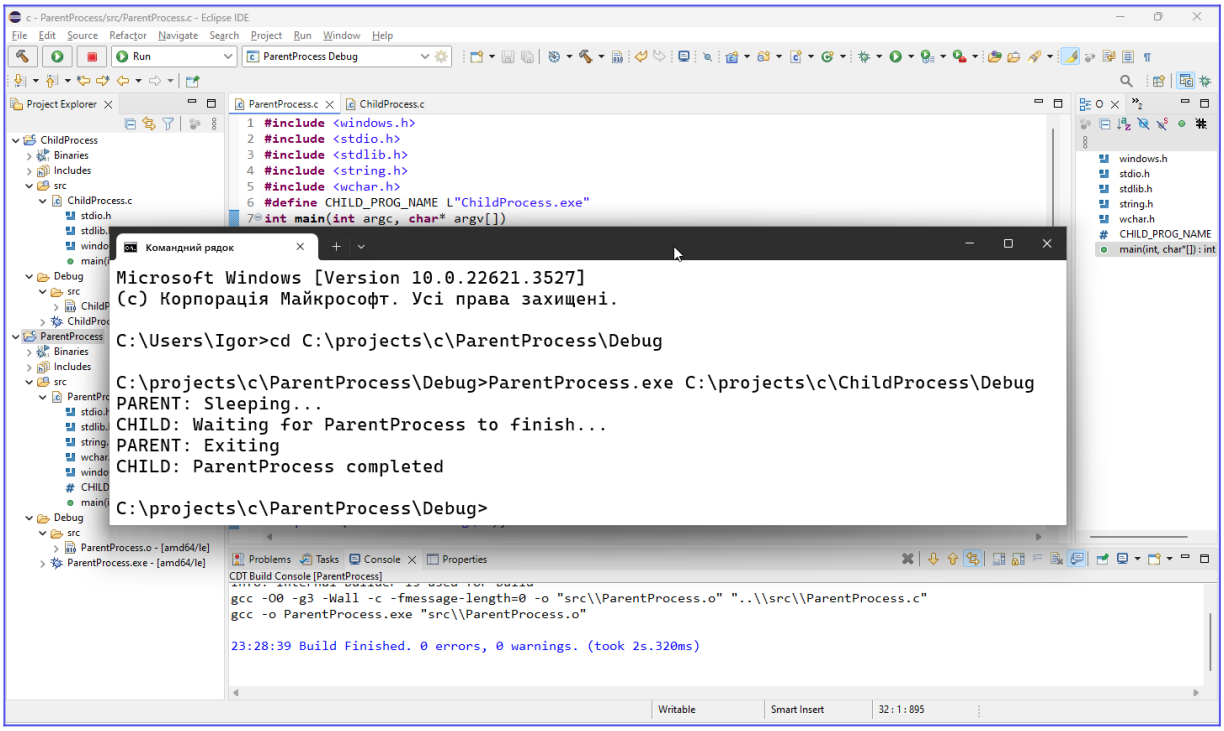


Рис. 7.4. Результат виконання програм ParentProcess та ChildProcess

Практична частина

Використовуючи компілятор GNU GCC та Eclipse IDE, в ОС MS Windows створити програму мовою С, яка породжує новий процес та виконує дії, що зазначені у варіанті завдання (таблиця 7.2), виданого викладачем.

При виконанні завдання використовувати знання, що отримані з лекцій, теоретичної частини роботи, а також з довідникової літератури та ресурсів Microsoft.

В звіт включити код програми з коментарями та результати роботи з поясненнями.

Таблиця 7.2

Варіанти виконання завдання в середовищі MS Windows

Варіант	*Дії, які виконує програма*	*Функції, які потрібно залучити в програмі*
1	Запуск редактора Notepad	`WinExec()`
	Запуск редактора WordPad з очікуванням	`CreateProcess(), WaitForSingleObject()`
2	Відкриття txt-файлу	`ShellExecute()`
	Запуск та завершення роботи редактору WordPad	`CreateProcess(), TerminateProcess()`

Критерії оцінювання виконання робіт

Виконання завдань, які описані в практичних частинах робіт, та захист результатів, що подані здобувачем у звітах, є обов'язковим. При наявності в роботі декількох варіантів завдання, конкретний варіант для виконання призначає викладач.

В залежності від складності практичної частини роботи, на виконання завдань відводиться певна кількість годин, обсягом, що представлений в робочій програмі дисципліни.

Після виконання практичної частини роботи, здобувач складає звіт, в якому повинен описати хід виконання завдання та надати висновки щодо отриманих результатів роботи. Вимоги по оформленню звітності подані в додатку А.

Захист звіту з виконаної роботи проводиться під час проведення заняття у встановлений розкладом основний час або у додатковий час, якій погоджений із викладачем.

Узагальнені критерії оцінки виконаної роботи наведені у таблиці:

Критерії оцінювання виконання та захисту роботи здобувачем	*Рейтингова оцінка*	*Інституційна оцінка*
Результати виконання роботи вірні згідно вимог, описаних в завданні та наданих викладачем в ході лекційних занять. Звіт оформлений згідно вимог, описаних в додатку А. На всі поставлені питання викладача при захисті роботи надані вірні відповіді. Робота захищена.	90 – 100	відмінно
Результати виконання роботи вірні згідно вимог, описаних в завданні та наданих викладачем в ході лекційних занять. Звіт оформлений згідно вимог, описаних в додатку А. На 80% поставлених питань викладача при захисті роботи були надані вірні відповіді. Робота захищена.	80 – 89	добре
Результати виконання роботи вірні згідно вимог, описаних в завданні та наданих викладачем в ході лекційних занять. Звіт оформлений згідно вимог, описаних в додатку А. На 70% поставлених питань викладача при захисті роботи були надані вірні відповіді. Робота захищена.	74 – 79	
Результати виконання роботи містять помилки. Є відхилення від поставлених вимог оформлення звітності. Під час захисту здобувач надав 50% вірних відповідей на поставлені питання викладача. Робота захищена.	60 – 73	задовільно
Результати виконання роботи невірні або звіт відсутній, або здобувач не може під час захисту результатів роботи відповісти на питання викладача. Робота не була захищена.	0 – 59	незадовільно

Перелік рекомендованих джерел

1. Andrew S. Tanenbaum, Herbert Bos. Modern Operating Systems, 5th Edition. – Pearson, 2024. – 1879 p. ASIN: B0D7QDMNFL.

2. Pavel Yosifovich. Windows 10 System Programming, Part 1. – Independently published, 2020. – 528 p. ISBN-13: 979-8634170381.

3. Richard Blum, Christine Bresnahan. Linux Command Line and Shell Scripting Bible, 4th Edition. – Wiley, 2021. – 832 p. ISBN-10: 1119700914, ISBN-13: 978-1119700913.

4. Chris Dent. Mastering PowerShell Scripting, 5th Edition. – Packt Publishing, 2024. – 826 p. ISBN-10: 1805120271, ISBN-13: 978-1805120278.

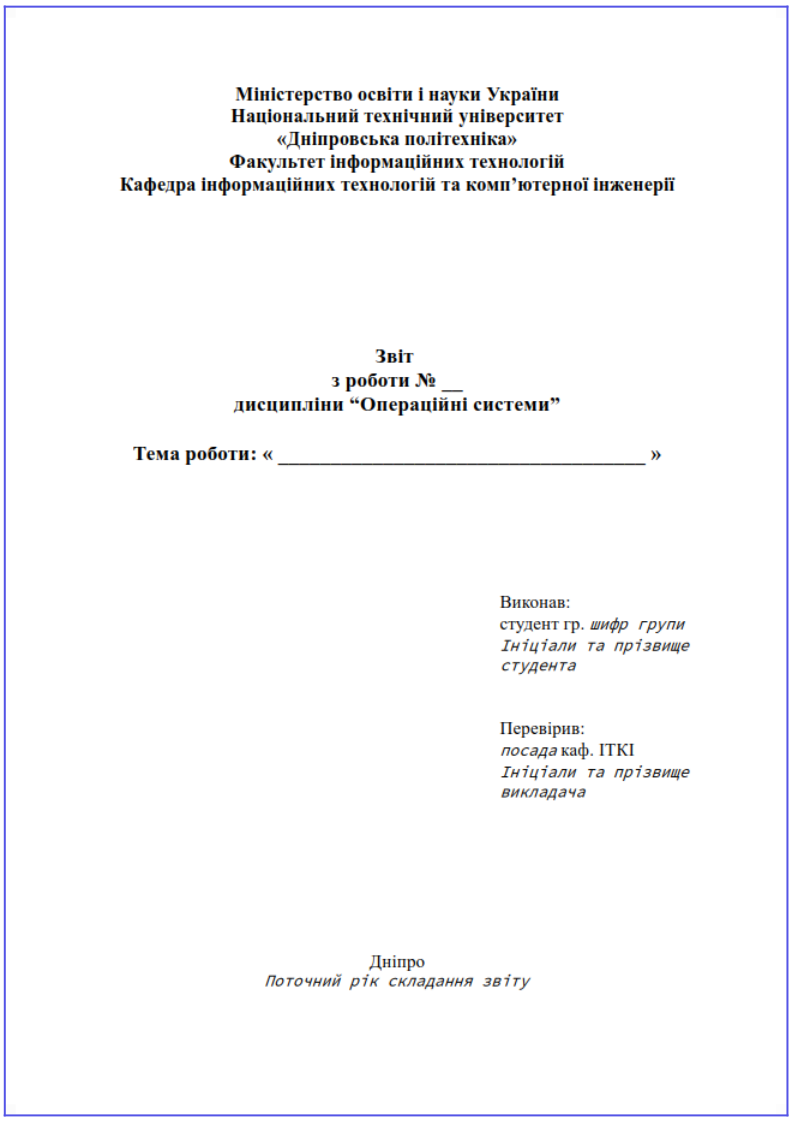
5. Гаркуша І.М. Конспект лекцій з дисципліни “Операційні системи” для студентів галузі знань 12 “Інформаційні технології” спеціальності 126 “Інформаційні системи та технології”. Дніпро : НТУ «ДП», 2020. 73 с.

Додаток А.
Вимоги до оформлення звітності

Звіт оформлюється на аркушах формату А4 та включає титульний аркуш (зразок оформлення представлений на рис. А.1) та опис роботи за представленим нижче зразком.

У разі захисту звіту з виконання роботи при аудиторній формі навчання, звіт друкується та надається викладачу на розгляд.

Для захисту виконаної роботи у дистанційній (online) формі навчання, звіт подається до захисту в електронному форматі PDF (Portable Document Format) та надсилається викладачеві через засоби online-спілкування для розгляду.



Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»
Факультет інформаційних технологій
Кафедра інформаційних технологій та комп'ютерної інженерії

Звіт
з роботи № __
дисципліни "Операційні системи"

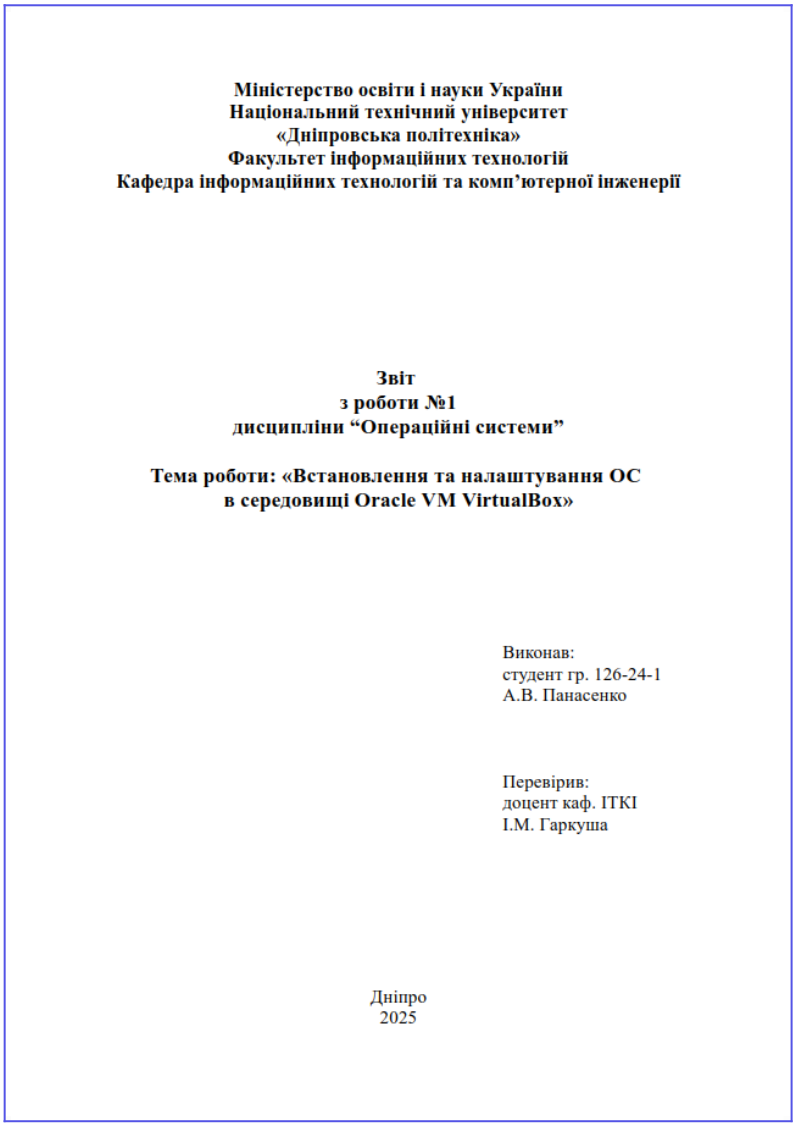
Тема роботи: « _____ »

Виконав:
студент гр. *шифр групи*
Ініціали та прізвище студента

Перевірив:
посада каф. ІТКІ
Ініціали та прізвище викладача

Дніпро
Поточний рік складання звіту

Рис. А.1. Форма оформлення титульного аркуша звіту


Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»
Факультет інформаційних технологій
Кафедра інформаційних технологій та комп'ютерної інженерії

Звіт
з роботи №1
дисципліни "Операційні системи"

Тема роботи: «Встановлення та налаштування ОС
в середовищі Oracle VM VirtualBox»

Виконав:
студент гр. 126-24-1
А.В. Панасенко

Перевірив:
доцент каф. ІТКІ
І.М. Гаркуша

Дніпро
2025

Рис. А.2. Приклад оформленого титульного аркуша звіту

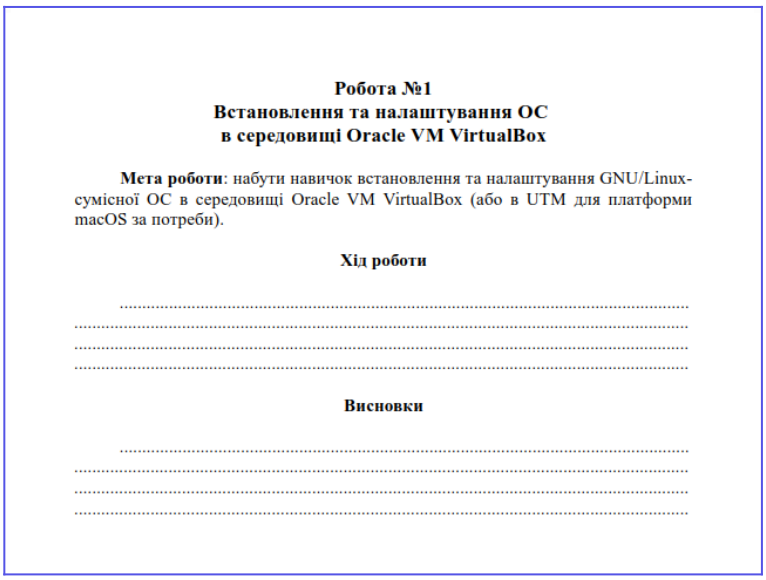
Опис виконаної роботи в звіті розпочинається з наступного аркуша. Вгорі сторінки, по центру, вказується номер роботи та її тема. Після цього вказується мета роботи. Через рядок по центру сторінки йдуть слова *Хід роботи* і далі через рядок розпочинається основний текст опису виконаної роботи.

Після завершення опису роботи, через рядок, йде частина, яка присвячена висновкам по проведеній роботі. Вона починається зі слова *Висновки*, розміщеного по центру листа та через рядок починається текст з висновками по роботі.

Приклад загального вмісту опису роботи представлений на рис. А.3.

Написання основного тексту ходу роботи та висновків йде **від третьої особи**!!! Тобто, є невірним використання у звіті висловів, на кшталт: "В ході роботи я розробив алгоритм розгалуження". Замість такого

вислову вірно буде написати: “В ході роботи розроблений алгоритм розгалуження”. Або замість вислову, на кшталт: “Під час виконання роботи ми опанували використання оператору розгалуження”, вірно буде написати: “Під час виконання роботи опановано використання оператора розгалуження”.



Робота №1
Встановлення та налаштування ОС
в середовищі Oracle VM VirtualBox

Мета роботи: набути навичок встановлення та налаштування GNU/Linux-сумісної ОС в середовищі Oracle VM VirtualBox (або в UTM для платформи macOS за потреби).

Хід роботи

.....
.....
.....
.....

Висновки

.....
.....
.....
.....

Рис. А.3. Приклад загального подання опису роботи

Текст звіту оформляється шрифтом Times New Roman з розміром 14, міжрядковий інтервал 1 – 1,5. Абзацний відступ – 5 символів.

Шрифтом з розміром 10 оформлюють тексти кодів програм або командних скриптів, якщо такі є в роботі. При цьому бажано використовувати один з наступних моноширинних шрифтів: Source Code Pro, Monospace, Consolas, DejaVu Sans Mono, Courier New, Monospaced, Menlo, SF Mono або подібні.

Вирівнювання тексту в звіті – по ширині сторінки.

Вирівнювання рисунків та підписів до них – по центру сторінки.

Вирівнювання таблиць (якщо такі будуть) – по центру сторінки.

Якщо код програми не є великим за обсягом, то його можна подати у вигляді знімку з екрану, причому фон повинен бути білого кольору, а текст коду програми повинен добре виділятися. Приклад такого знімку з екрану представлений на рис. А.4. Такий рисунок потрібно розмістити по центру сторінки звіту та надати по центру підпис для нього. Наприклад:

Рисунок 1 – Код програми за пунктом 1 завдання роботи №3

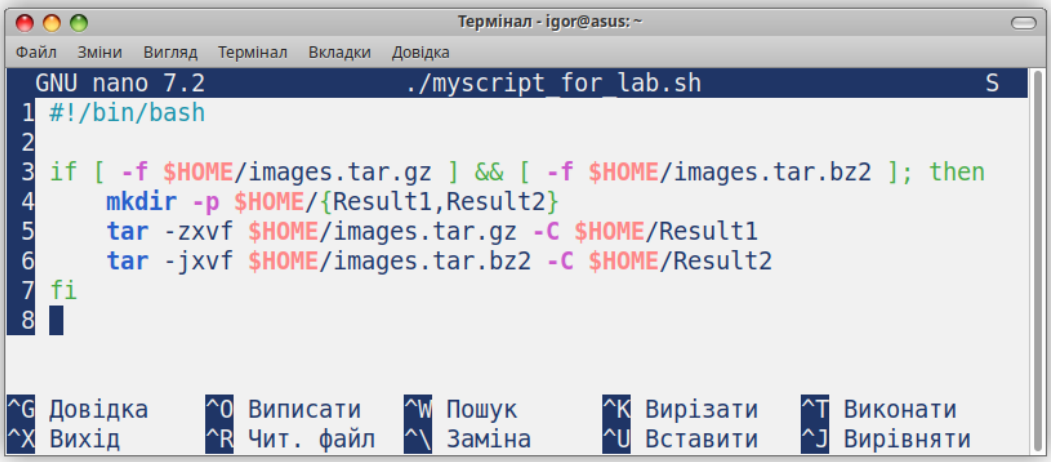


Рис. А.4. Приклад знімка з екрану вмісту
командного *bash*-скрипту для звіту

Якщо в тексті звіту наводиться рисунок, то на нього обов'язково повинно бути посилання з тексту та пояснення. Якщо поданий текст коду програми або командного скрипту, то при описі вказують номер рядку, коментуючи дії, які виконані в рядку під цим номером.

В роботах є завдання, які містять варіанти. При отриманні варіанту виконання від викладача, цей номер варіанту повинен бути відображений на початку опису ходу роботи. Якщо варіант завдання містить математичний вираз для підрахунку, то такий вираз також повинен бути присутнім у звіті. Для створення математичного виразу потрібно скористатися певними інструментами. Математичний вираз розміщують по центру сторінки та, при необхідності, нумерують. Наприклад:

$$S=\frac{3,7\,x}{\sqrt{5\,z^3}}+\frac{21\,x-3,4\,y^2+7\,z}{\sqrt{\sin^2 2\,x^3}} \qquad (1)$$

При складанні тексту висновків, потрібно вказати на результати щодо досягнення поставленої мети роботи. Вказати особливості виконання роботи та складнощі. При цьому доречно залучати вислови на кшталт: "В роботі розглянуті...", "...виконані...", "Отриманий результат дозволив...", "Використання умов дозволило..." та т.ін.

Навчальне видання

Гаркуша Ігор Миколайович

ОПЕРАЦІЙНІ СИСТЕМИ

Методичні рекомендації до виконання лабораторних робіт
для здобувачів ступеня бакалавра освітньо-професійних програм
«Інформаційні системи та технології»
спеціальності 126 Інформаційні системи та технології
та «Комп'ютерна інженерія»
спеціальності 123 Комп'ютерна інженерія

Видано в авторській редакції.

Електронний ресурс.
Підписано до видання 10.12.2024. Авт. арк. 5,6.

Національний технічний університет «Дніпровська політехніка».
49005, м. Дніпро, просп. Дмитра Яворницького, 19.