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

Фізико-технічний факультет

Голота В. І.

“СИСТЕМНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ. МОВА СЦЕНАРІЇВ BASH.”

Методичні вказівки до виконання лабораторних робіт

Електронне мережеве навчальне видання

Івано-Франківськ, 2024

УДК 004.432.2
Г 61

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

Рецензенти:

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

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

Методичні вказівки до виконання лабораторних робіт з дисципліни "Системне програмне забезпечення. Мова сценаріїв Bash. [Електронний ресурс] / Прикарпатський національний університет імені Василя Стефаника; уклад. Голота В. І. – Електронні текстові дані (8,7 файл: 1 Мбайт). – Івано-Франківськ: Прикарпатський національний університет імені Василя Стефаника, 2024. – 226 с. – Назва з екрану.

Електронне мережеве навчальне видання

У методичних вказівках викладено теоретичні відомості та практичні завдання для виконання лабораторних робіт з дисципліни "Системне програмне забезпечення. Мова сценаріїв Bash". Методичні вказівки розроблено з метою вивчення команд ОС Linux, консольних і потокових редакторів, операторів оболонки Bash та отримання практичних навичок написання консольних та інтерактивних сценаріїв.

Лабораторні роботи містять теоретичну частину, контрольні запитання, завдання та приклади програм. Методичні вказівки призначені для здобувачів ступеня бакалавра галузі знань 12 "Інформаційні технології" спеціальності 123 "Комп'ютерна інженерія".

УДК 004.432.2

ЗМІСТ

1. Команди Linux 4
2. Консольні редактори Vim, Nano.....22
3. Команди галуження.....48
4. Команди циклів.....64
5. Параметри і ключі командного рядка.....87
6. Перенаправлення потоків введення-виведення.....104
7. Задачі і сигнали.....117
8. Функції.....135
9. Текстові і графічні списки вибору.....155
10. Потокові редактори Sed, Gawk.....170
11. Регулярні вирази.....191
12. Мережеві команди.....200
13. Автоматизація сценаріїв.....217
Список літератури.....225

Лабораторна робота 1

Команди Linux

Мета роботи: вивчення основних команд Linux.

1. Теоретична частина

1.1. Список всіх команд Linux

Поекранний список команд

```
compgen -c | sort -d | less
whatis $(compgen -c) | sort -d | less
```

Список команд по стовпцях

```
ls $(echo ${PATH//:/ })
```

1.2. Інформації про команди і їх пошук

Інформація про команди:

help, man –h довідкова система man

help, man –h довідкова система help

mount –h показати опції команди mount

man mount знайти man сторінки з описання команди mount

man hier виведення на екран описання ієрархії файлової системи Linux

info mount показати інформацію про команду mount

apropos mount знайти man сторінок, які мають відношення до команди mount

card ls --output=ls.ps вивести інформацію про команду ls у ps-форматі

Пошук окремих команд:

type mount показати першу команду mount в PATH

whereis mount показати binary, source і man сторінки для mount

which umount знайти umount в любому місці файлової системи

whatis ls вивести однорядкове описання команди

rpm –gal | grep umount знайти umount в любих інстальованих пакетах

rpm –g --whatprovides tar знайти пакет, який підтримує команду tar

type <command> - визначення належності команди до Linux чи shell

1.3. Історія команд

Оболонка зберігає впорядкований список усіх введених команд. Кожній команді присвоюється номер відповідно до порядку її введення. Історію нумеровних команд виводить команда

```
history
```

Можна повторно виконати команди з історії команд

```
!!      # виконання останньої команди
!-3     # виконання третьої з останніх команд
!1084   # виконання 1084-ої команди
!5      # виконання п'ятої команди списку команд
```

!ps # виконання команди, яка починається з ps

Bash зберігає історію команд у файлі `.bash_history`

```
> ls -al | grep .bash*
```

1.4. Робота з каталогами і файлами

1.4.1. Вміст каталогу (ls)

При першому входженні у систему, поточним робочим каталогом є домашній каталог користувача. Домашній каталог користувача має таке ж ім'я, як ім'я користувача при вході у систему, наприклад

/home/user

Щоб дізнатися, що знаходиться у каталозі користувача використовується команда

ls

Команда ls виводить у консоль вміст поточного робочого каталогу. У каталозі можуть зберігатися підкаталоги, звичайні файли і скриті файли, назви яких починаються із символу '.' Список всіх файлів, включно з тими що починаються з символу ".", виводить команда

ls -a

Команда ls має багато ключових параметрів (опцій), які можна переглянути командою

man ls

1.4.2. Створення каталогів, файлів (mkdir, touch)

Для створення підкаталогу або ієрархії підкаталогів у робочому каталозі використовується команда

mkdir *subdir1*
mkdir *subdir2*
mkdir *subdir1/subdir12/subdir13*

Створення файлу

touch *file*

Створення файлу

cat > *file*
// введення вмісту файла

"CTRL+D" - збереження файлу

1.4.3. Перехід в інший каталог (cd)

Для переходу до іншого каталогу використовується команда

cd *subdir1*
cd *subdir2*
cd *subdir1/subdir12/subdir13*

Для посилання на домашній каталог (наприклад, /home/user) використовується символ ~. Так посилання на підкаталог /home/user/subdir1 можна записати як

~/subdir1

Перехід у домашній каталог з підкаталогу любої вкладеності

cd
cd ~

Перехід у попередній каталог

cd -

1.4.4. Абсолютний шлях до каталогу (pwd)

Абсолютний шлях до поточного каталогу дає команда

pwd

наприклад, /home/user/subdir1

1.4.5. Каталоги . і ..

Якщо виконати команду **ls -a** то можна побачити два спеціальних каталоги, які позначаються як **.** і **..**

Символ **.** позначає поточний каталог. Команда

cd .

не змінює поточний каталог.

Символ **..** позначає батьківський каталог для поточного каталогу. Тому команда

cd ..

здійснює перехід у батьківський каталог для поточного каталогу (вверх на один рівень по ієрархії каталогів).

Використання команди ../../ дозволяє задавати *відносний шлях до каталогу* (вверх на два рівні по ієрархії каталогів)

cd ../../*downloads*

1.4.6. Копіювання файлів і каталогів (cp)

Копіювання файлу у поточному каталозі

cp *file1 file2*

Копіювання файлу із заданого каталогу у точний

cp */home/user2/bash/1.bash* **.**

Копіювання файлу із заданого каталогу у одним рівнем вище у ієрархії каталогів

cp *file1 ../file2*

Копіювання каталогу із вкладеними підкаталогами (рекурсивне копіювання)

cp -R *dir1 dir2*

1.4.7. Переміщення файлів і каталогів (mv)

Переміщення файлу з поточного каталогу у заданий каталог

mv *file1 dir1/*

Переміщення файлів з поточного каталогу у заданий каталог

mv *file1 file2 file3 dir1/*

Перейменування файлу у поточному каталозі

mv *file1 file2*

Перейменування каталогу у поточному каталозі

mv *dir1/ dir2/*

Перейменування файлів з розширення *html* у розширення *htm*

rename *html htm *.html*

Переміщення і перейменування файлу за один крок

mv *file1 dir1/file2*

1.4.8. Вилучення файлів і каталогів (rm, rmdir)

Вилучення файлу
rm *file*
Вилучення файлу з підтвердженням
rm -i *file*
Вилучення порожнього каталогу
rmdir *dir*
Вилучення каталогу з файлами
rm -r *dir*
Порівняння вмісту каталогів
diff dir1 dir2

1.4.9. Виведення вмісту файлу на екран (cat, tee, less, head, tail, od, wc)

Вивести вміст файлу починаючи з початку на екран
cat *filename*
Вивести вміст файлу починаючи з початку на екран, а також записати у файл
tee *filename*
Вивести вміст файлу починаючи з кінця на екран
tac *filename*
Вивести вміст файлу починаючи з заданого розділювача і заданої кількості стовпчиків
cut -d*розділювач* -f*кількість-стовпчиків filename*
По екранне виведення вмісту файлу на екран. Гортання сторінок вперед/назад **PgUp/PgDn**. Вихід з перегляду **q**.
less *filename*
Виведення перших 10 рядків з файлу на екран
head *filename*
Виведення перших 20 рядків з файлу на екран
head -20 *filename*
Вивести 10 найновіших файлів у поточному каталозі
ls -lt | head
Виведення останніх 10 рядків з файлу на екран
tail *filename*
Виведення останніх 20 рядків з файлу на екран
tail -20 *filename*
Виведення вмісту файлів у вісімковому, шістнадцятковому або інших форматах.
od -td *filename*
Лічильник кількості символів "\n" (рядків), слів, байтів
wc *filename*

1.4.10. Статус і класифікація файлів

Статус файлу або всієї файлової системи
stat *filename*
Для класифікації іменованих файлів за типом їх даних використовується команда
file *filename*
Команда розпізнає наступні типи файлів:

- ASCI текст;
- Bash сценарій;
- ELF 64-біт LSB executable (виконуваний);
- ELF 64-bit LSB shared object (спільно використовуваний об'єкт);
- GNU tar archive (архів);
- gzip compressed data (стиснені дані);
- HTML document text (текстовий документ);
- JPEG image data (зображення);
- PostScript document text (PostScript файл);
- Zip archive data (архів даних).

1.4.11. Пошук даних у файлах (less, grep, uniq), пошук файлів (find, locate), заміна символів (tr)

Пошук з використанням команди **less:**

less *file.txt*

тепер, знаходячись у less, задати через похилу слово (шаблон) пошуку

/програмування

знайдені слова будуть підсвічені. Для продовження пошуку ввести ***n***.

Команда **grep** виконує пошук за заданими словами або шаблонами у файлі і виводить на екран рядки, які задовільняють умови пошуку.

grep *word filename*

grep *pattern filename*

Для ігнорування регістра великих/малих букв вказується опція **-i**

grep -i *word filename*

Для пошуку фраз або шаблонів їх необхідно задати в одинарних апострофах

grep *'мова програмування Java' filename*

Пошук рядків які повторюються

uniq -c *filename*

Команда **find** виконує пошук файлів і підкаталогів у файловій системі

Пошук файлів у ієрархії каталогів

find ./subdir1 -name *.txt

Пошук у поточному каталозі всіх підкаталогів

find . -type d

Вивести загальне число файлів у поточному каталозі і всі підкаталоги

find . -type f -print | **wc** -l

Пошук файлів за іменами у фоновому режимі (у файловій системі або базах даних)

locate *filename*

locate -i *filename*

locate -r *filename*

Перетворення символів із стандартного входу заданих шаблоном великі/малі букви

cat *filename* | **tr** *ABCxyz abcXYZ*

Вилучення або повідомлення про рядки, які повторюються у стандартному вході

sort *filename* | **uniq**

1.4.12. Завантаження файлів із віддалених сайтів (wget)

Команда wget є web-клієнт, яка дозволяє завантажувати файли з web і ftp сайтів
wget *URL*
wget http://www.ee.surrey.ac.uk/Teaching/Unix/science.txt
wget -O sci.txt http://www.ee.surrey.ac.uk/Teaching/Unix/science.txt

1.4.13. Стиснення і розтиснення файлів

Стиснення файлів

```
zip archivename.zip filename1 filename2 filename3
```

Розтиснення архіву файлів

```
unzip archivename
```

Стиснення файлів у .tar формат

```
tar –cvf tar-filename source-folder-name
```

Розтиснення .tar архіву

```
$ tar –xvf tar-filename
```

1.4.14. Сортування файлів і вмісту файлів

Сортування файлів

```
ls -l | sort
```

Сортування вмісту файлу

```
sort flag filename
```

1.5. Права доступу до файлів і каталогів

Якщо у поточному каталозі ввести команду

```
ls -l
```

то можна побачити детальну інформацію про файли і підкаталоги наступного виду

```
drwxr-xr-x 3 user user 4096 Apr  3 19:53 cpp
-rw-r--r-- 1 user user 3952 Apr 17 12:14 module
```

Ліва частина кожного рядка довжиною в 10 символів може містити наступний перший символ:

`d` – каталог;
`-` – файл;
`c` – символьний пристрій;
`b` – блоковий пристрій;
`l` – символьне посилання;
`p` – іменований канал;
`s` – сокет.

Наступні дев'ять символів вказують на права доступу до файлу або каталогу користувача.

Права доступу до файлів:

- r (або -) дозвіл (або відсутність дозволу) на читання файлу;
- w (або -) дозвіл (або відсутність дозволу) на зміну файлу;
- x (або - дозвіл (або відсутність дозволу) на виконання файлу.

Права доступу до каталогів:

- r (або -) дозвіл (або відсутність дозволу) на читання файлів каталогу;

• w (або -) дозвіл (або відсутність дозволу) на вилучення або добавлення файлів каталогу;
• x (або -) дозвіл (або відсутність дозволу) на доступ до файлів.

Дев'ять символів прав доступу розбиті на три групи по три символи:

• власник каталогу/файлу (user, u);
• група людей, які є власниками каталогу/файлу (group, g);
• усі інші, яким надається доступ до каталогу/файлу (other, o).

1.5.1. Зміна прав доступу

Міняти права доступу до каталогів/файлів може тільки власник файлів. Для зміни прав доступу служить команда chmod. Команда сприймає як абсолютне, так і символьне подання прав. У абсолютному поданні права користувача (u), групи (g) та інших (o) записуються вісімковими цифрами

```
u   g    o
rwxrwxrwx – 777   (вісімкові числа)
rw-rw-rw- - 666
r—r--r--  - 444
```

Вісімкові числа записують двійковими тетрадами:

```
rwx          7   = 111
rw-          6   = 110
r--          4   = 100
```

Зміна прав доступу в абсолютному поданні:

```
                    u   g   o
                    rw- r-- r--
>chmod 644 myfile   # 110_100_100
>chmode –R 644 /tmp/test   # рекурсивна зміна прав доступу до каталогу і всіх
його файлів
```

У символьному поданні можна явно вказати, яке право треба змінити. Формат символьного подання:

```
chmod <категорія><дія><набір_прав>< файл >
```

Категорія:

u – власник
g – група власника
o – інші
a – всі користувачі (all), аналогічно до ugo.

Дія:

+ – додати набір прав
- – відняти набір прав
= – назначити набір прав

Право:

r – на читання
w – на запис
x – на виконання
s – зміни ідентифікатора власника або групи

`t` – біт прилипання (sticky-біт)

`u` – таке як у власника

`g` – таке як у групи

`o` – таке як у інших

Приклад символьного задання прав групі та іншим на виконання файлу:

```
$ chmod go+x myfile
$ ls -1 myfile
-rw-r-x--x 1 user users 0 Feb 14 19:08 /home/user/myfile
```

Sticky біт це спеціальний тип прав доступу до каталогів та файлів. Встановлений прапор відображається символом t у полі прав доступу. Коли sticky біт каталогу встановлено, файлова система трактує файли в такому каталозі таким чином, що лише власник файлу, власник (owner) каталогу або root можуть перейменувати або вилучити файл. Якщо ж sticky біт скинуто, то будь-який користувач з правами на запис і виконання для цього каталогу може перейменовувати або вилучати файли, що містяться в ньому, незалежно від власника файлу. Звичайно його встановлюють на каталог з тимчасовими файлами (/tmp), щоб завадити звичайним користувачам вилучати або переміщати файли інших користувачів.

```
$ ls -ld /tmp
drwxrwxrwxt 1 roor root 2472 Oct 20 12:15
```

Встановити Sticky біт:

```
$ chmod  +t /home/user/1
```

або

```
$ chmod  1777 /home/user/1
```

Скинути Sticky біт:

```
$ chmod  +t /home/user/1
```

або

```
$ chmod  1777 /home/user/1
```

1.5.2. Зміна власника

Право зміни користувача (`SUID`) і групи (`SGID`) означає наступне. Звичайно виконуваний файл (програма або командний сценарій) отримує ті ж права на доступ до файлів, що і користувач, який запустив його на виконання. Але у цього файлу є ще й власник, права якого можуть бути іншими. Наявність одного із цих бітів дозволяє виконуваній програмі користуватися правами власника файлу або члена його групи.

1.5.2.1. Налаштування Setuid

Setuid – це біт дозволу, який дозволяє користувачеві запускати виконуваний файл із правами власника цього файлу. Іншими словами, використання цього біта дозволяє нам підняти привілеї користувача у разі, якщо це необхідно. Класичний приклад використання цього біта в операційній системі – це команда sudo.

```
$ which sudo /usr/bin/sudo
$ ls -l /usr/bin/sudo
-rwsr-xr-x 1 root root 166056 Jan 19  2021 /usr/bin/sudo
```

Як бачимо на місці, де звичайно знаходиться класичний біт x (на виконання), виставлений спеціальний біт s. Це дозволяє звичайному користувачеві системи виконувати команди з підвищеними привілеями без необхідності входу в систему як root, знаючи пароль користувача

root.

Встановити біт setuid:

```
$ chmod u+s <filename>
```

Скинути біт setuid:

```
$ chmod u-s <filename>
```

Для прикладу створимо файл myfile1 і перевіримо дозволи для виконання користувача, групи і інших:

```
$ touch myfile1
$ ls -l myfile1
-rw-r--r-- 1 user user 0 Sep 13 20:34 myfile
```

Як бачимо, файл не має дозволів на виконання для користувача, групи та інших. Додамо біт setuid:

```
$ chmod u+s myfile1
$ ls -l myfile1
-rwSr--r-- 1 user user 0 Sep 13 20:34 myfile
```

Зазначимо, що замість очікуваної літери «s», бачимо велику літеру «S». Чому? Це трапляється, якщо setuid встановлено, але сам власник файлу не має права на його виконання. Додамо цей дозвіл за допомогою команди chmod u+x.

```
$ chmod u+x myfile1
$ ls -l myfile1
-rwsr--r-- 1 user user 0 Sep 13 20:34 myfile
```

1.5.2.2. Налаштування Setgid

Принцип роботи setgid дуже подібний на setuid з відмінністю у тому, що файл запускатиметься користувачем від імені групи, яка володіє файлом. Ілюструє роботу цього біту команда crontab:

```
$ which crontab
/usr/bin/crontab
$ ls -l /usr/bin/crontab
-rwxr-sr-x 1 root 18 43720 Feb 13  2020 /usr/bin/crontab
```

Для прикладу налаштування setgid створимо файл myfile2 і перевіримо дозволи для виконання користувача, групи і інших:

```
$ touch myfile2
$ ls -l myfile2
-rw-r--r-- 1 user user 0 Sep 13 21:03 myfile2
```

Як бачимо, файл не має дозволів на виконання для користувача, групи та інших. Додамо біт setgid:

```
$ chmod g+s myfile2
$ ls -l myfile2
-rw-r-Sr-- 1 user user 0 Sep 13 21:03 myfile2
```

Зазначимо, що замість очікуваної літери «s», бачимо велику літеру «S». Чому? Це трапляється, якщо setgid встановлено, але група не має права на виконання файлу. Додамо цей дозвіл за допомогою команди chmod g+x.

```
$ chmod g+x myfile2
```

```
$ ls -l myfile2
-rw-r-sr-- 1 user user 0 Sep 13 21:03 myfile2
```

1.6. Процеси (processes) і завдання (jobs), апаратура, сервіси

Процес – це програма, яка виконується в оперативній пам'яті та має унікальний ідентифікатор процесу (PID). Інформацію про процеси та пов’язані з ними PID, статус і тип дає команда

```
$ ps
```

Виведення всіх поточних процесів у деревоподібному форматі

```
$ pstree
```

Інтерактивний перегляд процесів та керування ресурсами комп'ютера безпосередньо з терміналу забезпечує команда

```
$ htop
```

Команда моніторингу всіх поточних процесів у системі Linux з іменем користувача, рівнем пріоритету, унікальним ідентифікатором процесу та спільною пам’яттю для кожного завдання

```
$ top
```

```
top - 22:42:52 up  2:07,   users, load average: 0.00, 0.00, 0.00
Tasks: 6 total, 1 running, 5 sleeping, 0 stopped, 0 zombie
%Cpu(s):  0.0 us,  0.0 sy,  0.0 ni,100.0 id, 0.0 wa, 0.0 hi,  0.0 si, 0.0 st
MiB Mem : 7880.8 total,  07.9 free, 280.1 used, 92.7 buff/cache
MiB Swap: 2048.0 total, 2048.0 free, 0.0 used. 7367.3 avail Mem

PID USER    PR  NI  VIRT  RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   1 root   20  0  2324   1504   1404 S   0.0   0.0   0:00.04 init(Ubuntu-20.
  4 root    20  0  2324      4      0 S   0.0   0.0   0:00.00 init
 133 root  20   0  2328    108      0 S   0.0   0.0   0:00.00 SessionLeader
 134 root  20   0    2344    112      0 S   0.0   0.0   0:00.20 Relay(135)
  135 user 20   0   10060   5120   3352 S   0.0   0.1   0:00.37 bash
 1610 user 20   0   10876   3624   3108 R   0.0   0.0   0:00.00 top
```

Команда відображення інформації про архітектуру ЦП, таку як потоки, сокети, ядра та кількість ЦП

```
$ lscpu
```

```
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   36 bits physical, 48 bits virtual
CPU(s):                          4
On-line CPU(s) list:             0-3
Thread(s) per core:              2
Core(s) per socket:              2
Socket(s):                       1
Vendor ID:                       GenuineIntel
CPU family:                      6
Model:                           58
Model name:                      Intel(R) Core(TM) i5-3320M CPU @ 2.60GHz
Stepping:                        9
```

```
CPU MHz:                         2594.106
BogoMIPS:                        5188.21
Hypervisor vendor:               Microsoft
Virtualization type:             full
L1d cache:                       64 KiB
L1i cache:                       64 KiB
L2 cache:                        512 KiB
L3 cache:                        3 MiB
```

Команда розрахунку контрольної суми файлу, потоку або даних

```
$ cksum 11
1541341405 79 11
```

Детальна інформація про апаратне забезпечення на якому функціонує Linux

```
$ lshw
```

Інформація про блокові пристроїв

```
$ lsblk
NAME MAJ:MIN RM   SIZE RO TYPE MOUNTPOINT
sda    8:0    0 363.3M  1 disk
sdb    8:16   0     2G  0 disk [SWAP]
sdc    8:32   0     1T  0 disk /mnt/wslg/distro
```

Інформація про USB пристроїв

```
$ lsusb
Bus 002 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
```

Інформація про жорсткий диск та інші пристрої

```
$ hdparm
```

Процес може може виконуватися на передньому плані, у фоновому режимі або бути призупиненим. В загальному оболонка не повертає символ командного рядка до завершення виконання поточного процесу. Деякі процеси виконуються довго і захоплюють. Запуск таких процесів у фоновому режимі звільнює термінал і дозволяє запускати інші команди.

Для запуску процесу в фоновому режимі потрібні в кінці командного рядка задати символ &

```
$ sleep 10 &
[1] 164
```

де 1 – номер завдання, 164 – PID процесу.

Процес, який виконується на передньому плані можна перевести у фоновий режим. Для цього процес потрібно призупинити <Ctrl>+z і перевести у фоновий режим командою bg:

```
$ sleep 100 &
^Z[1]   Done                    sleep 10
[2]+  Stopped                   sleep 100
$ bg
[2]+ sleep 200 &
[2]+  Done                      sleep 100
```

Список завдань, затриманих і виконуваних (на передньому плані і у фоновому режимі),

виводить команда

```
jobs
[1]+  Stopped                 sleep 2000
[2]-  Running                 sleep 100 &
```

Затриманий процес запускається на виконання командою

```
fg %номер-завдання
```

Завершити завдання, яке виконується на передньому плані можна командою

```
<Ctrl>+c
```

Завершити завдання, яке затримане або виконується у фоновому режимі можна командою

```
$ kill %номер-завдання
```

Статус процесів можна отримати командою

```
$ ps
 PID TTY          TIME CMD
    9 pts/0    00:00:00 bash
  171 pts/0    00:00:00 sleep
  172 pts/0    00:00:00 ps
```

Завершити процес з використанням його PID можна командою

```
$ kill <PID>
$ kill 172
```

Якщо процес не завершується то можна використати опцію -9

```
$ kill 172 -9
```

Список запущених сервісів і їх статус

```
$ service --status-all
```

[-] apparmor
[?] apport
[-] atd
[-] console-setup.sh
[-] cron
[?] cryptdisks
[?] cryptdisks-early
[-] dbus
[?] hwclock.sh
[+] irqbalance
[-] iscsid
[-] keyboard-setup.sh
[?] kmod
[-] lvm2

Карта пам'яті процесу із заданим PID

```
$ pmap <pid>
```

1.7. Інформація про файлову систему (df, du)

Команда df повідомляє інформацію про вільний простір у файловій системі

```
df
$ df .
Filesystem      1K-blocks    Used  Available Use% Mounted on
/dev/sdc       1055762868 1640844 1000418552   1% /
$ df -h .
Filesystem      Size  Used Avail Use% Mounted on
/dev/sdc       1007G  1.6G  955G   1% /
```

Команда du виводить кількість кілобайтів, які використовуються кожним підкаталогом

```
$ du
4       ./.config/mc/mcedit
12      ./.config/mc
16      ./.config
4       ./zip
12      ./cpp/minheap
672     ./cpp
$ du *
432     DGP
164     DGP_arx/arx
192     DGP_arx
8       c
12      cpp/minheap
672     cpp
```

Визначення обсягу вільної пам'яті

```
$ free
              total        used        free      shared  buff/cache   available
Mem:        8069896      287000     7176092        2268      606804     7543936
Swap:       2097152           0     2097152
```

Використання системної віртуальної пам'яті

```
$ vmstat
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 0  0      0 7170124  21520 592244    0    0    10    13    2   11  0  0 99  0  0
```


2. Практична частина

Контрольні запитання.

1. Команда для отримання вмісту каталогу.
2. Команди роботи з каталогами.
3. Команда роботи з файлами.
7. Абсолютний і відносний шлях до каталогу.
8. Комбінації клавіш для консольного рядка.
9. Копіювання файлів і каталогів.

10. Переміщення файлів і каталогів.
11. Вилучення файлів і каталогів (rm, rmdir).
12. Виведення вмісту файлу на екран.
13. Пошук у файлі (less, grep).
14. Перенаправлення стандартних потоків.
15. Канали.
16. Символи підставлення * і ?

Завдання.

1. Виконати команди:

1.1. Описати функції команди ls з наступними ключами

ls -l
ls -lt
ls -ls
ls -lrS
ls -lrt

1.2. Створити в домашньому каталозі (/home/user) підкаталоги subdir1/subdir2/subdir3.

1.3. Перейти у підкаталог subdir3. Пояснити, що виведуть команди

ls ~
ls ~/..
ls ~/../..

1.4. Перейти з підкаталогу subdir3 зразу у каталог /home/user.

1.5. Виконати команду head -5 file.txt і пояснити результат

1.6. Вивести останні 15 рядків з файлу file.txt на екран.

1.7. Виконати команду grep -ivc і пояснити результат.

1.8. Виконати команду grep -ivn і пояснити результат.

1.9. Виконати команду wc file.txt і пояснити результат.

1.10. Виконати команди sort -d file.txt і sort -n file.txt і пояснити результат.

1.11. За допомогою команди less знайти у файлі file.txt всі слова, які містять символ "a".

1.12. За допомогою команди grep знайти у файлі file.txt всі слова, які містять символ "b".

1.13. З використанням каналів записати список файлів поточного каталогу у файл file.txt, відсортувати файл, вивести на екран перші 15 рядків.

1.14. Вивести на екран список файлів поточного каталогу, префікси імен яких містять тільки цифри.

1.15. Вивести на екран список файлів поточного каталогу, суфікси імен яких містять тільки цифри.

1.16. Вивести на екран список файлів поточного каталогу, префікси імен яких містять тільки букви.

2. Записати приклади типового використання 5 команди з 3-4 ключами для заданої групи:

2.1. Системна інформація;

2.2. Встановлення системи;

2.3. Файли і каталоги;

2.4. Пошук файлів;

2.5. Монтування файлових систем;

2.6. Дисковий простір;

2.7. Користувачі і групи;
2.8. Встановлення/зміна повноважень на файли;
2.9. Спеціальні атрибути файлів;
2.10. Архівування і стискання файлів.

3. Приклади

1. Створення сценарію:

```
# 01.sh
#!/bin/bash
# Сценарій висвічує дату і користувачів
echo ”1.sh”
echo ”Час і дата:”
date
echo ”Користувачі:”
who

>chmod u+x 01.sh
>./01.sh
```

2. Запуск сценарію 1.sh на виконання із 2.sh:

```
# 02.sh
echo ”2.sh”
bash 01.sh

>chmod u+x 02.sh
>./02.sh
```

3. Виконання сценарію з трасуванням команд

```
# 03.sh
#!/bin/bash
# bad.bash: A simple script to list files
shopt -o -s nounset
shopt -o -s xtrace
declare -i RESULT
declare -i TOTAL=3
while [ $TOTAL -ge 0 ] ; do
let ”TOTAL--”
let ”RESULT=10/TOTAL”
printf ”%d\n” ”$RESULT”
done
```

4. Виведення повідомлень на екран

```
# 04.sh
#!/bin/bash
# Сценарій висвічує дату і користувачів
echo -n Час і дата:
date
echo Користувачі:
who
```

5.Виведення змінних середовища

```
# 05.sh
```

```
# Виведення змінних shell середовища
$printenv
```

6. Виведення інформації про користувачів

```
# 06.sh
#!/bin/bash
# Інформація про користувача із системи
echo "Інформація про користувача із системи: $USER"
echo UID: $UID
echo HOME: $HOME
```

7. Використання змінних користувача

```
# 07.sh
#!/bin/bash
# Тестування змінних
days=10
guest="Катя"
echo "$guest входила в систему $days днів назад"
days=5
guest="Оля"
echo "$guest входила в систему $days днів назад"
```

8. Присвоєння значень іншій змінній

```
# 08.sh
#!/bin/bash
var1=10
printf ”%s\n” $var1
var2=$var1
echo var2=$var2
```

9. Виведення блоків символів із текстового файлу examples.txt

```
$ cat examples.txt
N,A,B,C,D,E
1,A,B,C,D,9
2,AA,BB,CC,DD,99
3,AAA,BBB,CCC,DDD,999

# 09.sh
#!/bin/bash
# вивести 3-й і 8-й символи
cut -b 3,8 examples.txt
# вивести символи з 3-го по 8-й
cut -b 3-8 examples.txt
# вивести всі символи починаючи з 4-го
cut -b -4 examples.txt
# вивести всі символи до 7-го
cut -b -7 examples.txt
# вивести 2-й стовпчик
cut -d ',' -f 2 examples.txt
# вивести 2-й і 4-й стовпчики
cut -d ',' -f 2,4 examples.txt
```

10. Виконання команд у сценарії

```
# 10.sh
#!/bin/bash
# Використання зворотного апострофу для
```

```
# присвоєння змінним результату виконання команди або сценарію
test=$(date)
echo "Дата і час:" $test
# запис дати у файл реєстрації
today=$(date +%d.%m.%y)
ls /usr/bin -al > log.$today
```

11. Перенаправлення виведення

```
# 11.sh
#!/bin/bash
# Перенаправлення виведення команда->файл
echo "*****************"
date > 10out
cat 10out
echo "******************"
# створення нового файлу
who > 10out
cat 10out
echo "*****************"
# дописування в існуючий файл
date > 10out
who >> 10out
cat 10out
```

12. Перенаправлення введення

```
# 12.sh
#!/bin/bash
# Перенаправлення даних команда<-файл
#  wc - число ліній, число слів, число байтів у вхідному файлі
echo "*******************************"
f1=10out
echo "Число рядків, слів, байтів у вхідному файлі $f1"
wc < $f1
rm $f1
# введення з консолі
echo "*******************************"
echo "Число рядків, слів, байтів при введенні з консолі"
wc << EOF
1
12
123
1234
12345
EOF
```

13. # перенаправлення потоку даних (piping) між командами

```
# 13.sh
#!/bin/bash
# rpm -qa > rpm.list
# sort < rpm.list
#rpm -qa | sort > rpm.list
 rpm -qa | sort | more
```

14. # перенаправлення із Java програми у Сі-програму у середовищі bash

```
// MyClass.java
public class MyClass {
   static void MyMethod() {
      System.out.println(”Hello world”);
   }
}

> javac MyClass.java
> java MyClass

// MyC.c
#include <stdio.h>
void main(int argc, char *argv[]) {
   char str[16];
   char str1[16];
   scanf(”%s %s”,str,str1);
   printf(”% %s”,str,str1);
}

> gcc MyC.c –o MyC

// 14a.sh
#!/bin/bash
echo ”14a.sh”
echo ”Java програма”
java MyClass

>chmod u+x 14a.sh

// 14b.sh
#!/bin/bash
./14a.sh
echo ”14b.sh”
echo ”C програма”
java MyClass | ./MyC

>chmod u+x 14b.sh
> ./14b.sh
14a.sh
Java програма
14b.sh
C програма
Hello world
```

Лабораторна робота 2

Консольні редактори Vim, Nano

Мета роботи: навчитися працювати у редакторах *Vim, Nano,* створювати Bash сценарії і запускати їх на виконання з командного рядка.

1. Короткі теоретичні відомості

1.1. Редактор Vim

Для роботи із сценаріями (скриптами, scripts) на мові Bash можуть використовуватися редактори *vim, nano, kwrite, kate, pluma, pico*. Найгнучкішим із них є редактор *vim,* а найпростішим *nano*.

Встановлення редактора

```
sudo apt install vim
```

Запуск редактора на виконання

```
vim
```

Редактор *vim* має специфічний набір команд і можливості налаштування під специфічні потреби користувача, тому з ним переважно працюють програмісти. Редактор *vim* працює із даними, які розміщуються у буфері пам'яті, тому він є незалежним від платформ. Сучасним розгалуженням редактора *vim* є *Neovim*.

Основна відмінність *vim* від інших редакторів в тому, що в нього декілька режимів:

- NORMAL – звичайний режим застосовується для швидкого переміщення по тексту і редагування: в ньому можна копіювати, вставляти, переміщати, вилучати і змінювати текст.
- INSERT – режим введення тексту дозволяє редагувати текст, як у будь-якому іншому редакторі.
- REPLACE – режим заміни тексту дозволяє редагувати текст, як у будь-якому іншому редакторі.
- VISIUL – візуальний режим виділення тексту.
- COMMAND – командний режим. Він створений для керування програмою та її інтерфейсом – переміщення по коду, поділу екрана на кілька частин, відкриття нових документів, роботи з файлами, налаштування редактора і т.п.

1.1.1. Звичайний режим

При відкритті нового або існуючого файлу редактор працює у звичайному режимі.

Перехід у всі інші режими здійснюється тільки через *звичайний режим*.

Перехід з інших режимів у звичайний здійснюється натисненням клавіші «ESC» або <CTRL+[>.

Для набору тексту нормальний режим не використовується. Редагування тексту виконується за допомогою команд. Вони дуже ефективні в комбінації з командами переміщення. Будь-яка команда з розділу переміщення може задати напрямок. Наприклад, можна видалити або скопіювати одне слово, вказавши "w" за відповідною командою.

У звичайному режимі можна вилучати символи і рядки, замінювати текст, копіювати і вставляти текст.

1.1.1.1. Вилучення тексту

x – вилучити символ на позиції курсора;

d – вилучити у заданому командою переміщення напрямку. Наприклад, “d1” вилучає один символ праворуч;

dd – вилучити рядок у поточній позиції курсора;

D – вилучити всі символи від положення курсора до закінчення рядка

dw – вилучити слово у поточній позиції курсора;

d$ – вилучити від поточної позиції до кінця рядка;

J, Shift+j – вилучити розрив рядка у поточній позиції курсора (об’єднує два рядки);

a – додати дані після поточної позиції курсора (перемикає у режим Insert). Вихід з режиму Insert натисканням клавіші `Esc`.

R, Shift+r – перезаписати дані починаючи від поточної позиції курсора (перемикає у режим Replace). Вихід з режиму Replace натисканням клавіші «**esc**».

1.1.1.2. Заміна тексту

r char – замінити символ на позиції курсора заданим після команди символом char;

c – замінити в заданому командою переміщення напрямку. Наприклад, “cb” замінює попереднє слово. Для набору тексту на заміну запускається режим вставки.

C – замінити до закінчення рядка. Текст на заміну набирається в режимі вставки.

1.1.1.3. Копіювання і вставка

y – копіювати в заданому далі напрямку;

yy – копіювати рядок;

yw - копіювати слово;

Y – копіювати до кінця рядка;

p – вставити вміст буфера після курсору;

P – вставити вміст буфера перед курсором;

gp – вставити вміст буфера після курсору і перемістити курсор у кінець вставленого тексту;

gP – вставити вміст буфера перед курсором і перемістити курсор у кінець вставленого тексту;

1.1.1.4. Додаткові команди редагування

u – скасувати останню дію;

U – відновити (скасувати вилучення);

<ctrl>r – повернути останню скасовану дію;

. повторити останню команду

J – приєднати рядок нижче до поточного.

1.1.2. Режим вставки

Для набору тексту потрібно перейти в режим "вставки". Він подібний на інтерфейс набору тексту багатьох інших програм. Для перемикання в режим вставки використовуються наступні команди:

i – вставити текст з позиції курсора, символ під курсором буде замінений;

I – вставити текст на початок рядка;

a – додати текст починаючи від позиції курсора;

A – додати текст в кінці поточного рядка;

o – вставити новий рядок після цього та почати редагування;

O – вставити новий рядок перед цим та почати редагування;

S – очистити рядок і почати введення;

R – замінити кілька символів.

Вихід з режимів вставки (Insert) натисненням клавіші «ESC».

1.1.3. Режим заміни

Для заміни тексту потрібно перейти в режим "заміни". Для перемикання в режим заміни використовується команда:

R – замінити декілька символів.

Вихід з режимів заміни (Replace) натисненням клавіші «ESC».

1.1.4. Режим візуального виділення

Дозволяє виділяти текстові фрагменти підсвічуванням. Потім до них можна застосовувати команди нормального режиму для редагування або форматування. Для перемикання у візуальний режим використовуються команди:

v – режим звичайного (посимвольного) візуального виділення переміщенням курсора вгору, вниз, вліво або вправо

V , <**Shift**>+**v**>– режим порядкового візуального виділення. Рядки виділяються цілком переміщенням вгору або вниз, потім виконати команду у-копіювання.

<Ctrl>+V, <Ctrl>+v – режим блокового візуального виділення. Виділяється прямокутна ділянка, яку можна збільшувати і зменшувати. Дозволяє вибирати фрагменти декількох рядків.

Щоб вийти з візуального режиму потрібно повторно натиснути клавішу **“v”, "V" або <Ctrl>+v** відповідно до режиму входу.

1.1.5. Командний режим

Використовується для просунутого редагування, зміни параметрів і керування. У ньому виконується збереження, вихід з програми, просунутий пошук і багато іншого. Активується клавішею двокрапки “**:**”.

1.1.5.1. Команди керування

Команди керування задаються введенням “:” перед кожною командою.

:q – вихід з програми. Дія не буде виконана, якщо не зберегти зміни;

:q! – вихід з програми з скасуванням усіх змін, які не були збережені;

:w – зберегти зміни. При первинному збереженні або збереженні в інший файл вказати ім'я через пробіл;

:e – редагувати зазначений далі файл;

:bn – редагувати наступний файл (якщо відкрито кілька файлів);

:bp – редагувати попередній файл (якщо відкрито кілька файлів);

:qw – зберегти файл і вийти.

1.1.5.2. Об'єднання команд

Як згадувалося вище, ефективність vim обумовлена можливістю будувати ланцюжки різних дій. Найкраще цей принцип ілюструється, якщо представляти команди `vim` у вигляді мови. Клавіші в нормальному режимі грають роль різних частин мови.

Наприклад, для копіювання 5 слів можна уявити цю фразу як команди, які розуміє `vim`. Це буде виглядати так:

:y5w (y – копіювати, 5 – кількість, w –слів)

Вилучити текст від поточної позиції до закінчення файлу:

:dG (d – вилучити, G – перехід в кінець файлу)

Також корисно засвоїти ряд використовуваних в діях `vim` принципів. Наприклад, здвоєний символ звичайно поширює дію команди на весь рядок. Наступна команда скопіює не один символ, а рядок цілком:

:yy

З іншого боку, відповідна команді велика буква часто діє від положення курсора до закінчення рядка. У випадку зі зміною тексту це буде наступна команда:

:C

Щоб виконати операцію з декількома символами або рядками, потрібно вказати перед командою число. Наприклад, така команда замінить весь текст від поточної позиції курсора до закінчення наступного рядка:

:2C

1.1.5.3. Нумерація рядків у командному режимі

Vim підтримує три режими нумерації рядків: абсолютний, відносний, гібридний.

Абсолютна нумерація рядків:

1. Натиснути клавіші Esc і перейти в командний режим.
2. Натиснути «:» і курсор переміститься у лівий нижній кут екрану.
3. Ввести команду

`:set number` (або `:set nu`)

і натиснути Enter. Номери рядків відобразяться у лівій частині екрану.

Для відключення абсолютної нумерації рядків ввести команду

`:set nonumber` (або `:set number!` або `:set nu!`)

1.1.5.4. Відносна нумерація рядків

Коли включена відносна нумерація рядків, поточний рядок відображається як 0, а рядки вище і нижче поточного рядка нумеруються з приростом (1, 2, 3 ... і так далі). Режим відносної адресації рядків зручний, так як операцій в *Vim*, такі як переміщення вверх/вниз і вилучення

рядків, працюють з відносними номерами рядків. Наприклад, щоб вилучити наступні десять рядків під курсором, потрібно використати команду d10j.

Щоб включити відносну нумерацію рядків потрібно перейти в командний режим і ввести

`:set relativenumber` або `:set rnu`

Для відключення відносної нумерації рядків ввести команду

`:set no relativenumber` (або `:set nornu`)

1.1.5.5. Гібридна нумерація рядків

Гібридна нумерація рядків така ж, як і відносна нумерація рядків, з тією лише різницею, що поточний рядок замість показу 0, показує абсолютний номер рядка. Щоб включити гібридну нумерацію рядків потрібно перейти в командний режим і ввести

```
:set number relativenumber
```

Щоб відключити гібридну нумерацію потрібно відключити абсолютну і відносну нумерацію.

1.1.5.6. Постійні налаштування

Якщо потрібно, щоб номери рядків з'являлися при кожному запуску *Vim*, потрібно додати відповідну команду у `.vimrc` (файл конфігурації *Vim*). Наприклад, щоб включити абсолютну нумерацію рядків, потрібно відкрити конфігураційний файл:

```
vim ~/.vimrc
```

і додати наступне

```
set number
```

1.1.6. Переміщення по документу

Для переміщення по документу використовуються клавіші **`hjkl`** (←↓↑→) або комбінації клавіш:

`h`, ← – переміщення на одну позицію вліво;

`j`, ↓ – переміщення на одну позицію вниз;

`k`, ↑ – переміщення на одну позицію вверх;

`l`, → – переміщення на одну позицію вправо;

`Ctrl+f` – переміщення на один екран вперед;

`Ctrl+b` – переміщення на один екран назад.

Також є додаткові функції навігації (найбільш важливі):

`gg` – перехід на початок документа;

`G` – перехід в кінець документа. Якщо попередньо ввести номер – перемістится на рядок з цими номером.

w – переміщення до наступного слова. Якщо попередньо ввести кількість - перемістится на цю кількість слів;

b – переміщення до попереднього слова. Якщо попередньо ввести кількість – перемістится на цю кількість слів назад;

e – перехід у кінець слова. Якщо попередньо вказати кількість – перемістится на вказану кількість слів;

0 – перехід на початок рядка;

$ – перехід у кінець рядка.

1.1.7. Пошук тексту

Для пошуку потрібно натиснути клавішу **"/"**. Курсор переміститься у командний рядок, де вводиться потрібне слово для пошуку.

`/word`

Для виходу з режиму пошуку потрібно перейти у режим вставки.

Заміна тексту. Для заміни фрагменту тексту потрібно перейти у командний рядок «**Shift + :**» і ввести команду `s` з параметрами:

`:s/old/new/` – замінити один фрагмент тексту `old` на `new`

`:n` – продовжити пошук вперед;

`:N` – продовжити пошук назад;

`:s/old/new/g` – замінити всі фрагменти тексту `old` на `new` у рядку

`:number1, number2s/old/new/g` – замінити всі фрагменти тексту `old` на `new` між рядками `number1` і `number2`

`:%s/old/new/g` – замінити всі фрагменти тексту `old` на `new` у всьому файлі;

`:%s/old/new/gc` – замінити всі фрагменти тексту `old` на `new` у всьому файлі з виведенням повідомлень.

1.1.8. Одночасне редагування декількох файлів

Щоб відкрити декілька файлів, потрібно просто передати їх як параметри для запуску `vim`:

```
vim файл1 файл2 файл3
```

Редактор `vim` відкриває перший файл, щоб переключитись до другого, використовується команда :**n**, щоб повернутися назад **:N**.

За допомогою команди **:buffers** можна переглядати всі відкриті файли, а команда **:buffer 3** перемикає на третій файл.

1.1.9. Буфер обміну

Для того щоб скопіювати блок тексту і вставити в інше місце, виконується наступна послідовність команд:

- натиснути «**esc**», щоб перейти в командний режим;
- виділити текст за допомогою команди **v**;
- за допомогою команди **y** скопіювати виділений текст у буфер;
- перемістити курсор у місце, де потрібно вставити цей текст;
- натиснути **p** для вставки тексту з буфера.

1.1.10. Довідка по командах

Нормальний режим – **:help x**
Режим вставки – **:help i_**
Візуальний режим – **:help v_u**
Командний рядок – **:help :quit**

1.2. Редактор Nano

Встановлення редактора

```
sudo apt install nano
```

Запуск редактора на виконання:

```
nano
```

Позначення клавіш в меню редактора:

^ - клавіша Ctrl

M (метасимвол "M") – клавіша Alt або Esc.

Комбінація клавіш	Описання
Ctrl-G	Показати довідку по командах
Файлові операції	
Ctrl-R File to itsert [from ./]:	Прочитати *файл* у файловий буфер
Ctrl-R потім **Alt-f**: File to read into new buffer [from ./]:	Відкрити новий файловий буфер і прочитати в нього файл
Alt-→ і **Alt-←** **Alt->** і **Alt-<**	Перемикання між файловими буферами
Ctrl-O	Записати зміни у файл
Ctrl-x	Вийти з редактора nano
Переміщення по тексту	
Ctrl-a	Перейти на початок рядка
Ctrl-e	Перейти на кінець рядка
Ctrl-v	Перейти на одну сторінку вниз
Ctrl-y	Перейти на одну сторінку вверх
Alt-\ і Alt-/	Перейти на початок або кінець файлу
Alt-g	Перейти на заданий рядок [, стовпчик]
Alt-]	Перехід між символами групування ([...], {...}, (...)
Копіювання/вставлення тексту	
Alt-a	Виділити стрілками блок. Повторне Alt-a зняти виділення блоку.
Ctrl-k	Вирізати поточний рядок і записати у буфер обміну
Ctrl-k	Вирізати виділений блок у буфер обміну
Alt-a потім **Alt**-^ **Alt-a** потім **Alt**-6	Скопіювати виділений блок у буфер обміну
Ctrl-u	Вставити з буфера обміну у поточну позицію курсора
Alt-Del	Вилучити текст від початку курсору до кінця тексту Вилучити виділений блок

Пошук і заміна	
Ctrl-w	Пошук заданого рядка у тесті
Alt-w	Повторити пошук заданого рядка
Alt-r	Пошук і заміна
Нумерація рядків	
Alt-Shift-# **Alt-N**	Включити/виключити нумерація рядків (повтор команди).
Відміна і відновлення операцій	
Alt-u	Відміна (undo) останньої операції
Alt-e	Відновлення (redo) останньої відміненої операції

1.3. Командні оболонки

Командна оболонка (англ., Shell) не тільки забезпечує інтерфейс взаємодії між користувачем та ядром операційної системи, але й своєрідна мова програмування, в якій присутні такі конструкції, як оператори умовного розгалуження, цикли, змінні та багато іншого.

Операційна система (ОС) запускає командну оболонку для кожного користувача, коли той входить у систему чи відкриває вікно терміналу. Першим, що користувач побачить у вікні терміналу, буде запрошення оболонки — воно, як правило, складається з імені користувача та імені хоста, відокремлених один від одного символом @, слідом за ними йде шлях поточної робочого каталогу та один із двох символів: **$** або **#**.

Якщо користувач не має особливих прав, то як запрошення для вводу команд у терміналі відображатиметься символ **$**. Якщо ж було виконано вхід під обліковим записом привілейованого користувача, то в терміналі відображатиметься символ **#.** Наприклад,

ivan@ubuntu:~# (поточний каталог ~ - /home/ivan/)

root@ubuntu:~# (поточний каталог ~ - /root/)

Після запрошення користувач вводить різні команди в термінал, оболонка запускає програми для користувача, а потім відображає в терміналі результат їх виконання. Команди можуть бути введені безпосередньо самим користувачем, або зчитані з файлу, який називається shell-сценарієм або shell-програмою.

Команди Linux та сценарії запускаються в командній оболонці. Команди, що вводяться користувачем, діляться на два типи:

- Внутрішні – це команди, які вбудовані в оболонку.
- Зовнішні – це команди, які не вбудовані в оболонку. За своєю суттю вони є скоріше невеликими окремими програмами, які розташовані десь у файловій системі (зазвичай, у каталогах /bin або /usr/bin).

Щоб визначити тип команди, достатньо у вікні терміналу ввести

type *ім'я_команди*

Використовуються наступні командні оболонки типу Bourne shell:

- sh (Bourne Shell);
- bash (Bourne-Again shell);
- ksh (Korn shell);
- zsh (Z Shell);

- csh (C shell);
- tcsh (TENEX/TOPS C Shell);

sh (скор. від «Bourne shell») – це найстаріша (серед розглянутих) оболонка, написана Стівеном Борном з AT&T Bell Labs для ОС UNIX v7. Оболонка доступна практично в будь-якому *nix-дистрибутиві. Багато інших командних оболонок походять саме від sh. Завдяки своїй швидкості роботи та компактності, ця оболонка є кращим варіантом для написання shell-сценаріїв. До її недоліків можна віднести відсутність функцій для використання оболонки в інтерактивному режимі, а також відсутність вбудованої обробки арифметичних та логічних виразів.

bash (скор. від «Bourne-Again shell») – це вдосконалений та доповнений варіант командної оболонки sh, який є одним з найпопулярніших сучасних командних оболонок *nix-систем. Властивості:

- Сумісний з sh.
- Об’єднує в собі корисні фішки оболонок ksh та csh.
- Підтримує навігацію за допомогою стрілочок, завдяки чому можна переглядати історію команд та виконувати редагування прямо у командному рядку.

Характерні риси bash:

Повний шлях до оболонки: /bin/bash.

Запрошення для звичайного користувача: ім'я_користувача@ім'я_хоста:~$ (де ~ домашній каталог поточного користувача, наприклад, ivan@mypc:~$).

Запрошення для суперкористувача (root): root@ім'я_хоста:~#.

ksh (скор. від «Korn shell») – це командна оболонка, яка розроблена Девідом Корном з AT&T Bell Labs в 1980-х роках. Властивості:

- Є розширенням sh.
- Має зворотну сумісність з sh.
- Має інтерактивний функціонал, як в csh.
- Підтримує зручні для програмування функції, такі як: вбудована підтримка арифметичних виразів/функцій, Сі-подібний синтаксис скриптів та засоби для роботи з рядками.
- Працює швидше за csh.
- Може запускати написані для sh скрипти.

csh (скор. від «C shell») — це командна оболонка, створена Біллом Джоєм (автором редактора vi) з метою удосконалення стандартно] оболонки Unix (sh). Властивості:

- Має вбудовані функції для інтерактивного використання, наприклад, псевдоніми (aliases) та історію команд.
- Підтримує зручні для програмування функції, такі як: вбудована підтримка арифметичних виразів та Сі-подібний синтаксис скриптів.

tcsh (скор. від «TENEX C shell») – це командна оболонка, створена Кеном Гріром, яка позиціонується як покращена версія оболонки csh.

Має повну сумісність з csh. Властивості:

- Саме в цій оболонці вперше з’явилася функція автодоповнення команд та шляхів.
- Зручна для інтерактивної роботи.
- Підтримує редактор командного рядку у стилі vi або emacs.

• Є стандартною оболонкою у FreeBSD.

zsh (скор. від «Z shell») – це командна оболонка, створена Паулем Фалстадом під час його навчання у Принстонському університеті, яка позиціонується як вільна сучасна sh-сумісна командна оболонка. Властивості:

• Серед стандартних оболонок найбільше схожа на ksh, але включає безліч покращень.

• Вбудована підтримка програмного автодоповнення команд, імен файлів та іншого.

• Підтримка перевірки орфографії та синтаксичних помилок.

• Розділена історія команд для одночасної роботи з декількома запущеними оболонками.

Перевірка командної оболонки, встановленої за замовчуванням

```
$ echo $SHELL
/bin/bash
```

Щоб переглянути всі доступні командні оболонки у системі, необхідно звернутися до вмісту файлу `/etc/shells`:

```
$ cat /etc/shells
/bin/sh
/bin/bash
/usr/bin/bash
/bin/rbash
/usr/bin/rbash
/bin/dash
/usr/bin/dash
```

Зміна оболонки

```
$ sudo chsh --shell /bin/sh host
$sudo grep host /etc/passwd
```


1.4. Оболонка Bash

Ключові слова Bash:

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

Команда `help` виводить перелік команд, які підтримує поточна реалізація Bash.

1.4.1. Запуск Bash на виконання

Bash може працювати в інтерактивному режимі або запускатися на виконання, як бінарний файл з консолі. Для цього послідовність Bash-команд (сценарій) записується у текстовий файл. Текстовий файл сценарію можна запустити на виконання двома способами:

– змінити права доступу до файлу виконання (execute) і запустити на виконання, наприклад

```
> chmod u+x myscript
> ./myscript
```

– запустити на виконання викликом Bash:

```
> bash myscript
```

Bash запускається для роботи в інтерактивному режимі командою `sh`. Вийти з Bash можна командою `exit`.

```
> sh
Sh-4.2$
...
$exit
```

Таким чином, признак роботи в консолі символ “**>**”, а роботи у Bash – “**$**”.

У командному рядку можна вводити декілька команд, розділивши їх крапкою з комою:

```
> pwd ; whoami
/home/internet
internet
```

Для виведення дати і часу служить команда

```
$ date
$ date "+%H:%M"
```

1.4.2. Змінні і стандартне середовище Bash

Існує два типи змінних, які можна використовувати у сценаріях:

– змінні середовища, наприклад:

`$USER` – ім’я користувача, який виконує сценарій;

`$HOSTNAME` – hostname машини на якій виконується сценарій;

`$SECONDS` – число секунд з початку виконання сценарію;

`$RANDOM` – випадкове число;

`$LINENO` – номер поточного рядка сценарію;

– змінні користувача, наприклад var1=10; var2=”Іван”.

Коли користувач входить у систему, запускається програма Bash і читає серію сценаріїв конфігурації, які називаються файлами запуску. Вони визначають стандартне середовище для всіх користувачів. Після цього з’являються додаткові файли запуску в нашому домашньому каталозі, які визначають наше особисте середовище. Точна послідовність залежить від типу сеансу оболонки, який запускається. Є два види: сеанс оболонки входу з реєстрацією та сеанс оболонки входу без реєстрації. Сеанс оболонки входу з реєстрацією – це сеанс, під час якого запитується ім’я користувача та пароль. Сеанс оболонки входу без реєстрації зазвичай виникає, коли запускається термінальний сеанс у графічному інтерфейсі користувача.

Сеанс оболонки з реєстрацією читає один або декілька наступних файлів:

`/etc/profile` – глобальні конфігурації, що стосуються всіх користувачів

`~/.bash_profile` – особистий файл запуску користувача. Можна використовувати для розширення або заміни налаштувань у сценарії глобальної конфігурації.

`~/.bash_login` – якщо ~/.bash_profile не знайдено, bash намагається прочитати цей сценарій.

`~/.profile` – якщо ні ~/.bash_profile, ні ~/.bash_login не знайдено, bash намагається прочитати цей файл. Це типове значення в дистрибутивах на основі Debian, наприклад Ubuntu.

Сеанс оболонки без реєстрацією читає один або декілька наступних файлів:

`/etc/bash.bashrc` – сценарій глобальної конфігурації, який застосовується до всіх користувачів.

`~/.bashrc` – особистий файл запуску користувача. Можна використовувати для розширення або заміни налаштувань у сценарії глобальної конфігурації.

Оболонки входу читають один або кілька файлів запуску, як показано нижче:

Окрім читання вказаних вище файлів запуску, оболонки входу без реєстрації також успадковують середовище від свого батьківського процесу, зазвичай оболонки входу з реєстрацією. Потрібно подивитися, які з цих файлів запуску у є системі. Оскільки більшість імен файлів, наведених вище, починаються з крапки (це означає, що вони приховані), потрібно буде використовувати параметр «-a» під час використання команди `ls`. Файл `~/.bashrc` є, мабуть, найважливішим файлом запуску з точки зору звичайного користувача, оскільки він майже завжди читається. Оболонки входу без реєстрації читають його за замовчуванням, і більшість файлів запуску для оболонок входу з реєстрацією написані таким чином, щоб також читати файл `~/.bashrc`.

Якщо переглянути типовий `.bash_profile` (у разі відсутності `.profile`) системи Ubuntu), то він виглядає приблизно так:

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

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

У сценарії перевіряється чи файл `$HOME/.bashrc` існує, якщо так то читається і виконується у поточній оболонці файл `$HOME/.bashrc` (команда `"."` синонім команди `source`). Наступним кроком до змінної `PATH` додається шлях до каталогу `$HOME` і шлях `$PATH`

```
PATH="$HOME/bin:$PATH"
```

Оболонка підтримує список каталогів, де зберігаються виконувані файли (програми) у змінній середовища `$PATH`, і шукає лише каталоги в цьому списку. Якщо програма не знайде програму після пошуку в кожному каталозі в списку, видасть повідомлення про помилку «Команда не знайдено».

Цей список каталогів називається шляхом `$PATH`. Список каталогів можна переглянути за допомогою такої команди:

```
$ echo $PATH
```

В результаті буде повернено список каталогів, розділених двокрапками, у яких здійснюватиметься пошук, якщо під час введення команди не вказано конкретне ім’я шляху.

Можна створити змінну `PATH`, додати потрібні каталоги `$HOME/directory` до шляху `$PATH` і експортувати змінну:

```
$ PATH=$PATH:$HOME/directory
$ export PATH
```

Звичайно виконувані програми користувача зберігаються у каталозі `$HOME/bin`. Перевірити чи такий каталог існує і створити можна командою

```
$ [ ! -d "$HOME/bin" ] && mkdir $HOME/bin
```

Якщо він існує то можна перемістити виконуваний сценарій у каталог `$HOME/bin`

```
$ mv ./runme $HOME/bin
```

Також потрібно перевірити чи каталог `$HOME/bin` є у списку `$PATH`

```
$ [[ $PATH =~ $HOME/bin ]] && echo 'yes' || echo 'no'
no
```

Якщо не має, то потрібно добавити у шлях `$PATH`

```
$ export PATH=$HOME/bin:$PATH
```

Тепер виконуваний сценарій можна запускати на виконання з любого каталогу

```
user@host:~/bash$ runme
```

Кожний дистрибутив Liux має свої підходи до зміни змінної `$PATH`, однак можна завжди змінити або створити файл `$HOME/.bashrc` і в ньому визначити змінну `$PATH`

```
$ PATH=$PATH:$HOME/directory
$ export PATH
```

1.4.3. Призначення і виведення значення змінних

Bash розрізняє імена змінних і значення змінних. Для використання значення змінної перед її іменем ставиться знак $.

Використання змінної середовища Bash для виведення поточного каталогу користувача

```
echo ”Поточний каталог: $HOME”
Поточний каталог: /home/internet
```

```
$ num=5
$ myfile=”info.txt”
$ echo $num $myfile
$ echo ”$num $myfile”
$ printf  ”%s %s\n” ”$num $myfile”
```

Нульова команда «:»

```
$ :
$ : $(date)   перенаправлення виводу у пристрій /dev/null аналогічно як
$ date > /dev/null
```

Змінній може бути присвоєний результат виконання команди Linux:

– за допомогою конструкції `$()`;

```
$ DATE=$(date)
$ printf “%s\n” “$DATE”
```

Якщо команди розділені крапкою з комою “;” то вони виконуються послідовно.
A; B запускається A, а потім B, незалежно від успіху або невдачі A.

```
$ printf ”%s\n” “Повідомлення 1” ; printf ”%s\n” “Повідомлення 2”
Повідомлення 1
Повідомлення 2
```

Якщо команди розділені подвійним амперсандом
A && B (логічне and) то виконуються B, тільки успішного виконання A

```
$ date 'abcd!' && printf "%s\n" "Помилка в команді date"
date: invalid date 'abcd!'
```

Якщо команди розділені подвійними вертикальними рисками
A || B (логічне або) то виконуються B, тільки після помилки в A

```
$ date 'abcd!' || printf "%s\n" "Помилка в команді date"
date: invalid date 'abcd!'
"Помилка в команді date"
```

Знаки ; || && \ (новий рядок) можуть змішуватися в одному рядку

```
$ date 'adbcd!' || printf "%s\n" "Помилка в команді date" && \
printf "%s\n" "printf виводиться"
date: bad conversion
Помилка в команді date
printf виводиться
```

Оператор NOT (!). Вилучити всі файли у каталозі за винятком *.html

```
$ rm -r !(*.html)
```

1.4.4. Об'єднання команд в групи

Є два способи згрупувати список команд, які потрібно виконати як одиницю. Коли команди згруповано, переспрямування можна застосувати до всього списку команд. Наприклад, виведення усіх команд у списку може бути перенаправлене до одного потоку. При розміщенні списку команд у круглих дужках () оболонка створює підоболонку і кожна з команд у списку виконується у середовищі підоболонки.

```
$ ( command1; command2; ... commandN )
$ ( echo "Привіт світ"; ls; pwd; date ) > outputfile
```

Оскільки список виконується у підоболонці, то призначення змінних не залишаються в силі після завершення підоболонки.

```
$ VAR="1"; ( VAR="2"; echo "Всередині групи: VAR=$VAR" ); echo "Зовні: VAR=$VAR"
Всередині групи: VAR=2
Зовні: VAR=1
```

Розміщення списку команд між фігурними дужками {} призводить до того, що список буде виконано в поточному контексті оболонки. Підоболонка не створюється. Крапка з комою після списку є обов'язковою

```
$ { command1; command2; ... commandN; }
```

```
$ VAR="1"; ( VAR="2"; echo "Всередині групи: VAR=$VAR" ); echo "Зовні: VAR=$VAR"
Всередині групи: VAR=2
Зовні: VAR=2
```

Фігурні дужки є зарезервованими словами, тому вони повинні бути відокремлені від списку пробілами або іншими метасимволами оболонки. Дужки є операторами, і розпізнаються оболонкою як окремі токени, навіть якщо вони не відокремлені від списку пробілами. Статус

виходу обох цих конструкцій є статусом виходу списку.

Оператор об'єднання довгих команд в декількох рядках \:

```
$ nano test\(1\).txt
```

1.4.5. Поточний стек каталогів

Інформація про поточний стек каталогів. Команда `dirs` виводить список каталогів у стеку.

```
$ dirs
~/поточний каталог
```

Команда `pushd` добавляє каталог у список і змінює поточний каталог на новий

```
$ pwd
/home/user1
$ pushd /home/user1/new
~/new ~
$ pwd
/home/user1/new
```

Команда `popd` вилучає перший каталог зі списку і змінює каталог на наступний у списку

```
$ popd
~
```

Показати список каталогів із відносними шляхами

```
$ dirs –v
```

Показати список каталогів із абсолютними шляхами

```
$ dirs –l
```

Очистити список каталогів

```
$ dirs -c
```

Перемістити 1-ий (n-ий) каталог зліва у списку на початок списку

```
$ dirs –1
```

Перемістити 1-ий (n-ий) каталог справа у списку на початок у списку

```
$ dirs +1
```

Помістити каталог у список без зміни поточного каталогу

```
$ dirs –n
```

1.4.6. Використання лапок і символу екранування

В сценаріях використовуються два типи лапок: подвійні ”…”, одинарні ‘…’ і символи екранування.

Подвійні лапки. Якщо поміщаємо текст *у* подвійні лапки, усі спеціальні символи, які використовує оболонка, втрачають своє спеціальне значення та розглядаються як звичайні символи. Винятком є «$», «\» (зворотна коса риска) і «`» (зворотні лапки). Це означає, що розбиття слів, розширення імені шляху, розширення тильди та розширення фігурних дужок придушено, але розширення параметрів, арифметичне розширення та заміна команд все одно виконуються. Використовуючи подвійні лапки, можемо використовувати назви файлів, які містять вбудовані пробіли. Уявіть, що є файл під назвою two words.txt. Якщо його використати в

командному рядку, розділення слів призвело б до того, що це розглядалося б як два окремих аргументи, а не як єдиний бажаний аргумент:

```
$ ls -l two words.txt
ls: cannot access two: No such file or directory
ls: cannot access words.txt: No such file or directory

$ ls -l "two words.txt"
-rw-rw-r-- 1 me me 18 2020-02-20 13:03 two words.txt
```

Приклад створення двох каталогів:

```
$ mkdir hello world
$ ls -F
hello/ world/
```

Приклад створення імені одного каталогу з двох слів:

```
$ mkdir ”hello world”
$ ls -F
hello world/
```

За замовчуванням функція поділу слів шукає наявність пробілів, знаків табуляції та нового рядка (символів переходу рядка) і розглядає їх як роздільники між словами. Це означає, що пробіли без лапок, табуляції та нові рядки не вважаються частиною тексту. Вони служать лише роздільниками. Оскільки вони розділяють слова на різні аргументи, наш приклад командного рядка містить команду, за якою слідують чотири різні аргументи.

```
$ echo this is a     test
this is a test
```

Якщо додати подвійні лапки то поділ слів відміняється, а вбудовані пробіли не розглядаються як роздільники, а стають частиною аргументу. Після додавання подвійних лапок командний рядок містить команду, за якою слідує один аргумент:

```
$ echo "this is a     test"
this is a     test
```

Той факт, що символи нового рядка вважаються роздільниками у механізмі поділу слів, спричиняє цікавий, хоч і непомітний вплив на заміну команд:

```
$ echo $(cal)
September 2023 Su Mo Tu We Th Fr Sa 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
18 19 20 21 22 23 24 25 26 27 28 29 30

$ echo "$(cal)"
   September 2023
Su Mo Tu We Th Fr Sa
                1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
```

У першому випадку підставлення команди без лапок призвела до командного рядка, що містив тридцять вісім аргументів. У другому – командний рядок з одним аргументом, який містить вбудовані пробіли та символи нового рядка.

Одинарні лапки. Коли потрібно відмінити всі розширення, використовуються одинарні лапки. Приклад виведення тексту без лапок, з подвійними та одинарними лапками:

```
$ echo text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER
text /home/user/*.txt a b foo 4 user

$ echo "text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER"
text ~/*.txt {a,b} foo 4 user

$ echo 'text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER'
text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER
```

Змінну можна створити і присвоїти їй значення використовуючи знак ”=”.

```
$ FILENAME=”info.txt”
$ printf ”%s\n” ”$FILENAME”
info.txt
```

Якщо змінна знаходиться у подвійних лапках, то до неї можна звернутися через `$ім’я_змінної` або `${ім’я_змінної}`. Якщо змінна знаходиться у одинарних лапках – то до неї звернутися неможливо.

```
#!/bin/bash
x=5 # присвоюємо x значення 5
echo ”x=$x”   # буде виведено x=5
echo ’x=$x’   # буде виведено x=$x
```

Для присвоєння змінній результату виконання команди Linux або арифметичного виразу використовується `$(вираз)`.

```
# присвоєння змінній вмісту каталогу
$ x=$(ls -l)
$ echo $x
# присвоєння змінній арифметичного виразу
$ x=1
$ x=$(expr $x + 1)
$ x=$((x+1))
```

Символ відміни (екранування). Іноді потрібно вивести лише один символ з особливим значенням. Для цього ми можемо перед символом поставити зворотну косу риску, яка в цьому контексті називається символом відміни. Часто це робиться в подвійних лапках, щоб вибірково запобігти розширенню:

```
$ echo "The balance for user $USER is: \$500.00"
The balance for user user is: $500.00
```

Також часто використовується екранування, щоб усунути особливе значення символу в назві файлу. Наприклад, у іменах файлів можна використовувати символи, які зазвичай мають особливе значення для оболонки. Серед них «$», «!», «&», « » та інші. Обробити спеціальний символ в назві файлу, можна так:

```
$ mv bad\&filename good_filename
```

Зворотна коса риска використовується для продовження опцій довгих команд з нових рядків. У цьому випадку вона відміняє дію символу новий рядок (\n).

```
$ ls -l \
   --reverse \
   --human-readable \
   --full-time
```

Зворотні косі риски також використовуються для вставки спеціальних символів у текст. Вони називаються керуючими символами зі зворотною косою рискою:

`\n` – вставлення порожнього рядка;

`\t` – вставлення символу горизонтальної табуляції у текст;

`\a` – виведення звукового сигналу;

`\\` – вставлення зворотної косої риски;

`\f` – символ надсилається друкарці для виштовхування аркуша.

```
$ echo -e "Вставлення декількох порожніх рядків\n\n\n"
```

```
$ echo -e "Слова\tвідокремлені\tгоризонтальною\tтабуляцією"
Слова   відокремлені    горизонтальною  табуляцією
```

```
$ echo -e "DEL C:\\WIN2K\\LEGACY_OS.EXE"
DEL C:\WIN2K\LEGACY_OS.EXE
```

1.4.7. Налагодження сценаріїв і опції

Для виведення результатів виконання команд потрібно добавити для bash опцію -x:

```
# date-time.bash
#!/bin/bash -x
i=0
let i=$((i+1))
echo $i
test=$(date)
echo "Дата і час:" $test
# запис дати у файл реєстрації
today=$(date +%d.%m.%y)
ls /usr/bin -al > log.$today
```

```
./date-time.bash
+ i=0
+ let i=1
+ echo 1
1
++ date
+ test='Fri Sep 29 22:59:46 EEST 2023'
+ echo 'Дата і час:' Fri Sep 29 22:59:46 EEST 2023
Дата і час: Fri Sep 29 22:59:46 EEST 2023
++ date +%d.%m.%y
+ today=29.09.23
+ ls /usr/bin -al
```

або використати команду `set -x`

```
# date-time.bash
#!/bin/bash
i=0
set  -x
let i=$((i+1))
echo $i
test=$(date)
echo "Дата і час:" $test
# запис дати у файл реєстрації
today=$(date +%d.%m.%y)
ls /usr/bin -al > log.$today
set +x
```

```
$ bash date-time.bash
```

`bash –n` виявлення помилок сценарію без його виконання

```
$ bash -n date-time.bash
```

Можна задати опції Bash командою `shopt` на початку сценарію

```
#!/bin/bash
shopt -so errexit # завершити сценарій якщо команда завершилася з помилкою
shopt -so nounset # завершити сценарій якщо зустрілася невизначена змінна
shopt –so xtrace  # вивести на консоль кожну команду перед її виконанням
```

```
#!/bin/bash
shopt -so xtrace
for i in $(seq 0 2 10); do
echo $i
done
```

```
 ./2.sh
++ seq 0 2 10
+ for i in $(seq 0 2 10)
+ echo 0
0
+ for i in $(seq 0 2 10)
+ echo 2
2
+ for i in $(seq 0 2 10)
+ echo 4
4
+ for i in $(seq 0 2 10)
+ echo 6
6
+ for i in $(seq 0 2 10)
+ echo 8
8
+ for i in $(seq 0 2 10)
+ echo 10
10
```


1.4.8. Експорт змінних

Кожний сценарій виконується у своєму власному процесі, який створює область видимості для всіх змінних сценарію. Сценарій може запускати на виконання інші сценарії. Якщо потрібно щоб змінні сценарію були доступні у іншому запущеному підсценарії, то ці змінні потрібно експортувати командою `export`.

1.4.9. Псевдоніми (aliases) і функції оболонки

Псевдонім – це простий спосіб створити нову команду, яка діє як абревіатура довшої. Він має такий синтаксис:

```
alias name=value
```

де `name` – це ім'я нової команди, а `value` - це текст, який буде виконуватися щоразу, коли ім'я вводиться в командний рядок.

Приклад створення нової команди today у командному рядку

```
$ alias today='date +"%A, %B %-d, %Y"'
```

Якщо записати її у кінець фалу `$HOME/.bashrc` то це створить нову вбудовану (builtin) команду `today`.

Псевдоніми підходять для дуже простих команд, але щоб створити щось складніше, потрібні функції оболонки. Функції оболонки або підсценарії в сценарії є невеликими функціями, які записують у файл `$HOME/.bashrc`. Наприклад, записавши функцію `today()`

```
today() {
    echo -n "Today's date is: "
    date +"%A, %B %-d, %Y"
}
```

у файл `$HOME/.bashrc` отримуємо вбудовану функцію.

2. Практична частина

Контрольні запитання.

1. Які є режими роботи редактора Vim і як здійснюється перехід між ними?
2. Які команди використовуються в нормальному режимі роботи редактора Vim?
3. Які команди використовуються в режимі вставки редактора Vim?
4. Які команди використовуються в командному рядку?
5. Які режими нумерації рядків є у Vim і як їх включити і відключити?
6. Редактор Nano, основні комбінації клавіш.
7. Командні оболонки.
8. Оболонка Bash. Як виконати сценарій з любого каталогу?
9. Присвоєння і виведення значення змінних.
10. Об'єднання команд у групи дужками (), {}.
11. Поточний стек каталогів.
12. Використанні подвійних і одинарних лапок, символу екранування.
13. Налагодження сценаріїв, трасування команд.
14. Розділення команд символами ;, ||, &&, \.
15. Псевдоніми і функції оболонки.

Завдання.

1. Написати сценарій який виводить з html файлу усі теги з номерами рядків в яких вони знаходяться. Назву каталогу вводити з командного рядка.

2. Написати сценарій для отримання інформації (назви і розміри) окремо про файли і окремо про каталоги (з вкладеними) у заданому каталозі. Назву каталогу вводити з консолі.

3. Написати сценарій для створення у заданому каталозі заданої кількості файлів з іменами user-1-date,...,user-n-date. Назву каталогу і кількість файлів n вводити з командного рядка.

4. Написати сценарій для створення у заданому каталозі заданої кількості вкладених каталогів з іменами user-1-dir-1, user-1-dir-1/dir-2, user-1-dir-1/dir-2/dir-3,... Назву каталогу і кількість підкаталогів n вводити з командного рядка.

5. Написати сценарій для переміщення файлів з каталогу з вкладеними підкаталогами в один новий каталог. Ім’я каталогу вводити з командного рядка.

6. Написати сценарій для переміщення кожного файлу у підкаталог заданого каталогу. Підкаталоги іменувати цифрами 1, 2, 3, ... В каталоги з меншими номерами записувати файли з меншим розміром.

7. Написати сценарій, який у каталозі $USER знаходить файли, які містять два заданих слова. Задані слова вводити з командного рядка.

8. Написати сценарій, який у каталозі $USER знаходить файли, які містять однакову кількість слів.

9. Написати сценарій для виконання арифметичного виразу (a+b)*(c-d)/(a*d) з використання сценаріїв sum.sh, sub.sh, mul.sh. Значення змінних a, b, c, d вводити як параметри командного рядка.

10. Написати сценарій для виконання арифметичного виразу (a+b)*(c-d)/(a*d) з використання функцій оболонки sum(), sub(), mul(). Значення змінних a, b, c, d вводити як параметри командного рядка.

11. Написати сценарій який визначає кількість файлів і підкаталогів у заданому каталозі, які створені у поточному році по місяцях. Ім'я каталогу вводити з командного рядка.

12. Написати сценарій, який виводить на екран імена файлів каталогу $HOME згруповані за типами.

13. Написати сценарій, який виводить на екран імена файлів каталогу $HOME із датою створення і датою останньої модифікації.

14. Написати сценарій, який виводить на екран імена файлів каталогу $HOME е яких дозволений запис (write) групі "інші".

15. Написати сценарій, який виводить із заданої інтернет сторінки усі номери телефонів у відсортованому порядку . Адресу інтернет сторінки вводити з командного рядка

16. Написати сценарій, який виводить із заданої інтернет сторінки усі e-mail адреси у відсортованому порядку . Адресу інтернет сторінки вводити з командного рядка

17. Написати сценарій, який записує у стек підкаталоги каталогу, а потім виводить на екран їх абсолютні шляхи.

Примітка. Номер варіанту завдання вибирається за порядковим номером студента у журналі групи.

Звіт з лабораторної роботи має містити в одному word файлі:

- назву групи, П.І.Б. студента, завдання до роботи;
- короткий опис використаних команд;
- текст Bash сценарію з коментарями;
- роздрук екранів із результатом виконання Bash сценаріїв.

3. Приклади

1. Друк тексту

```
#!/bin/bash
echo "Printing text with newline"
echo -n "Printing text without newline"
echo -e "\nRemoving \t backslash \t characters\n"
```

2. Створення HTML сторінки, яка не містить лапок

```
#!/bin/bash
echo "<html>
 <head>
```

```
   <title>
   Назва сторінки
   </title>
 </head>

 <body>
   Вміст сторінки
 </body>
 </html>"
$ bash 2.bash > 2.html
```

3. Створення HTML сторінки, яка містить лапки (<<)

```
cat << _EOF_
<html>
<head>
    <title>
    Назва сторінки
    </title>
</head>

<body>
    Вміст сторінки
</body>
</html>
_EOF_
$ bash 3.bash > 3.html
```

4. Створення HTML сторінки, яка містить лапки і символи табуляції (<<-)
(<<- ігнорування символів табуляції)

```
cat <<- _EOF_
<html>
<head>
    <title>
    Назва сторінки
    </title>
</head>

<body>
    Вміст сторінки
</body>
</html>
_EOF_
$ bash 4.bash > 4.html
```

5.Додавання двох чисел

```
#!/bin/bash
((sum=25+35))
#друк результату
echo $sum
```

6. Багаторядковий коментар

```
#!/bin/bash
: '
Обчислення
значення 5 у степені 2'
```

```
((a=5*5))
echo $a
```

7. AND логіка

```
#!/bin/bash
echo "Введіть username"
read username
echo "Введіть password"
read password
if [[ ( $username == "admin" && $password == "secret" ) ]]; then
echo "дійсний user"
else
echo "недійсний user"
fi
```

8. АБО логіка

```
#!/bin/bash
echo "Введіть любе число"
read n
if [[ ( $n -eq 15 || $n  -eq 45 ) ]]
then
echo "Ви виграли"
else
echo "Ви програли"
fi
```

9. Сума двох чисел

```
#!/bin/bash
echo "Введіть перше число"
read x
echo "Введіть друге число"
read y
(( sum=x+y ))
echo "Сума=$sum"
```

10. Вилучення файлу

```
#!/bin/bash
echo "Введіть ім'я файлу для вилучення"
read fn
rm -i $fn
```

11. Поточна дата і час

```
#!/bin/bash
Year=$(date +%Y)
Month=$(date +%m)
Day=$(date +%d)
Hour=$(date +%H)
Minute=$(date +%M)
Second=$(date +%S)
echo $(date)
echo "Поточна дата: $Day-$Month-$Year"
echo "Поточний час: $Hour:$Minute:$Second"
```

12. Очікування на завершення усіх дочірніх процесів

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

```
echo "Очікування команди" &
process_id=$!
wait $process_id
echo "Завершено із статусом $?"
```

13. Друк змінних середовища

```
printtenv
  або
set
  або
env
```

14. Логічні оператори

```
if (1 == 1 && 2 == 2)
   echo "test;"
if (1 == 1 && 2 == 3)
   echo "test;"
if (1 == 1 || 2 == 2)
   echo "test;"
if (1 == 1 || 2 == 3)
   echo "test;"
```

15. Операції цілочисельної математики

```
#!/bin/bash
var1=10
var2=20
var3=$(expr $var1 + $var2)
echo var1=$var1 var2=$var2
echo var1+var2=$var3
var3=$(expr $var1 \* $var2)
echo var1*var2=$var3
var3=$[$var1 + $var2]
echo var1+var2=$var3
var3=$[$var1 * ( $var1 + $var2)]
echo "var1*(var1+var2)="$var3
```

16. Операції арифметики з плаваючою крапкою

```
# Набрати в консолі
$ bc -q
3.44/5
0
scale=4
3.44/5
.6880
quit
$
```

17. Арифметика з плаваючою крапкою. Виклик калькулятора bc зі сценарію

```
#!/bin/bash
var1=`echo " scale=4; 3.44 / 4" | bc`
echo Результат=$var1
```

18. Арифметика з плаваючою крапкою

```
#!/bin/bash
var1=100
var2=45
var3=$(echo "scale=4; $var1 / $var2" | bc)
```

```
echo Результат: $var3
```

19. Арифметика з плаваючою крапкою

```
#!/bin/bash
var1=20
var2=3.14159
var3=$(echo "scale=4; $var1 * $var2" | bc)
var4=$(echo "scale=4; $var3 * $var2" | bc)
echo Результат:$var4
```

20. Вихід із сценаріїв командою exit

```
#!/bin/bash
#Перевірка статусу  exit
var1=10
var2=30
var3=$[ $var1 + $var2 ]
echo var3 = $var3
exit 5

$echo $?
```

21. Перевірка статусу exit

```
#!/bin/bash
var1=10
var2=30
var3=$[ $var1 + $var2 ]
echo var3 = $var3
exit $var3

$echo $?
```

22. Перевірка статусу exit

```
#!/bin/bash
var1=10
var2=30
var3=$[ $var1 * $var2 ]
echo var3 = $var3
exit $var3

$echo $?
```

23. Перевірка exit статусу виконання команди

```
#!/bin/bash
mkdir tmp
cd tmp
touch myfile
err=&?
echo ”err=” $err
if [ $err ]
then
rm myfile
cd ..
rm –d tmp
fi
```

```
$ ./23.sh
err= 0
```

24. Експорт змінних із одного сценарію в інший

```
#!/bin/bash
var1=1;var2=2
echo $0: $var1 , $var2
export var1
./25.sh
echo $0: $var1 , $var2

# 25.sh
#!/bin/bash
var1=5
echo $0: $var1 , $var2
var1=10;var2=20

$./24.sh
24.sh: 1 , 2
$./25.sh: 5 ,
24.sh: 1 , 2
```

Лабораторна робота 3

Команди галуження

Мета роботи: навчитися застосовувати вкладені команди if-then-else, test, let, case.

1. Короткі теоретичні відомості

Команди порівняння.

В *командах порівняння* використовуються значення істина (`true`) і фальш (`false`):

```
$ true
$ printf ”%d\n” ”$?”
0
$ false
$ printf ”%d\n” ”$?”
1
```

Команда **`if-then`** перевіряє чи є нульовим код завершення команди або послідовності команд і якщо так, то виконує послідовність команд за словом `then`.

```
if команда1
then echo ”Команду виконано”
fi
```

```
if команда1; команда2
then echo ”Всі команди виконано”
else echo ”Не всі команди виконано”
fi
```

```
if  cmp a b  &> /dev/nul
then echo ”файли ідентичні”
else echo ”файли різні”
fi
```

Команда `if-then` може мати вкладені перевірки.

Команда **`case`** підтримує багатозначний вибір для однієї змінної. Синтаксис оператора `case`:

```
case EXPRESSION in

  PATTERN_1)
    STATEMENTS
    ;;

  PATTERN_2)
    STATEMENTS
    ;;

  PATTERN_N)
    STATEMENTS
    ;;

  *)
    STATEMENTS
    ;;
esac
```

```
case $змінна in
     шаблон11 | шаблон12)
          команда1;;
     шаблон2)
          команда 2;;
     шаблон3)
          команда 3;;
     …
     *) команди за замовчуванням;;
     esac
```

Приклад:

```
#!/bin/bash

echo -n "Enter the name of a country: "
read COUNTRY

echo -n "The official language of $COUNTRY is "

case $COUNTRY in

  Lithuania)
    echo -n "Lithuanian"
    ;;

  Romania | Moldova)
    echo -n "Romanian"
    ;;

  Italy | "San Marino" | Switzerland | "Vatican City")
    echo -n "Italian"
    ;;

  *)
    echo -n "unknown"
    ;;
esac
```

Команда **`test`** – це вбудована команда Bash, яка сприймає свої аргументи як вирази порівняння чисел, символьних рядків або файлів і повертає у відповідності з результатами перевірки нуль – істина або 1 – фальш.

```
$ var=abc
$ test -n "$var"
$ echo $?
1
```

Команда перевірки умов: `test умова`. Команда може об'єднуватися з наступними варіантами оператора `if`: `if-then-fi, if-then-else-fi, if-then-elif-then-fi`

`if test умова`
`then`
`  команди`
`fi`
	`if test умова`
	`then`
	`  команди`
	`else`
		`if test1 умова`
		`then`
		`  команди`
		`elif test2 умова`

	команди fi	then команди else команди fi

Оператор `If` може мати вкладені оператори `if`.

Використання фігурних {} і круглих () дужок .

Фігурні дужки дозволяють *групувати команди і виконувати в поточній оболонці*.

Круглі дужки дозволяють *групувати команди і виконувати в підоболонці*.

Можна перевірити наявність файлу `orders.txt`, якщо він існує, то виводиться про нього інформація і файл вилучається:

```
$ test -f orders.txt && { ls -l orders.txt ; rm orders.txt; } \
|| printf ”no such file”
```

Вихід команди `ls` стає входом для команд у фігурних дужках:

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

Фігурні дужки також використовуються для зміни параметрів:

• обрізання змінної:

```
$ var="abcde"; echo ${var%d*}
abc
```

• замін, як у sed:

```
var="abcde"; echo ${var/de/12}
abc12
```

• для використання значень за замовчуванням:

```
$ default="hello"; unset var; echo ${var:-$default}
hello
```

• для створення списків:

```
$ echo f{oo,ee,a}d
food feed fad
```

```
Перейменування error.log в error.log.old
$ mv error.log{,.OLD}
```

```
$ for num in {000..2}; do echo "$num"; done
000
001
002

$ echo {00..8..2}
00 02 04 06 08

$ echo {D..T..4}
D H L P T
```

Команда test

Bash використовує синонім команди `test` – одинарні квадратні дужки […] (в реалізації тільки ліву дужку)

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

`if [ умова ]; then`
`  команди`
`fi`
	`if [ умова ]; then`
	`  команди`
	`else`
	`  команди`
	`fi`
		`if [ умова1 ]; then`
		`  команди`
		`elif [ умова2 ]; then`
		`  команди`
		`...`
		`else`
		`  команди`
		`fi`

```
$ cat greetings
# Program to print a greeting
hour=$(date | cut -c12-13)
if [ "$hour" -ge 0 -a "$hour" -le 11 ]; then
    echo "Good morning"
elif [ "$hour" -ge 12 -a "$hour" -le 17 ]; then
       echo "Good afternoon"
else
       echo "Good evening"
fi
```

Над умовами можуть виконуватися *логічні операції*:

`!` – унарне заперечення

`-a` – логічне AND

`-o` – логічне АБО

```
[ ! -f "$mailfile" ]
[ "$x1" != "$x2" ]
[ -f "$mailfile" -a -r "$mailfile" ]
[ "$count" -ge 0   -a   "$count" -lt 10 ]
[ -n "$mailopt" -o -r $HOME/mailfile ]
```

В умовах можуть порівнюватися:

- числові вирази, **`[ n1 -x n2 ]`**, де `-x`:

`-eq`, `-ge`, `-gt`, `-le`, `-lt`, `-ne`;

- символьні рядки, **`[ str1 -x str2 ]`**, де `-x`:

=, !=, <, >, `-n` (довжина більше нуля), `-z` (довжина нуль);

- файли; умови порівняння **`[ -x file ]`**, де `-x`::

`-b` *file* — True якщо файл є блоковим пристроєм

`-c` file — True якщо файл є символьним пристроєм

`-d` *file* — True якщо файл є каталогом

`-e` *file* — True якщо файл існує

f1 `-ef` *f2* — (еквівалентні файли) True якщо файл *f1* є жорстким посиланням на файл *f2*

n1 `-eq` *n2* — (однакові) True якщо *n1* однаковий з *n2*

`-f` *file* — True якщо файл існує і є дійсно файлом

n1 `-ge` *n2* — True якщо *n1* є більше або рівне *n2*

n1 `-gt` *n2* — True якщо *n1* є більше *n2*

`-g` *file* — True якщо файл має встановлені права доступу групи

`-G` *file* — True якщо файлом володіє ваша група

-h *file* (or -L *file*) — True якщо файл є символьним посиланням
-k *file* — True якщо файл має встановлений sticky біт доступу
n1 -le *n2* — True якщо *n1* є менше або рівне *n2*
n1 -lt *n2* — True якщо *n1* є менше *n2*
-n *s* (or just *s*) — (*not null*) якщо стрічка не порожня
-N *file* — True якщо файл має новий вміст (з часу останньої операції read)
n1 -ne *n2* — True якщо *n1* не дорівнює *n2*
f1 -nt *f2* — True якщо файл *f1* новіший від файлу *f2*
-O *file* — True якщо користувач є (ефективним) власником файлу
f1 -ot *f2* — (*older than*) True якщо файл *f1* старіший від файлу *f2*
-p *file* — True якщо файл є каналом (pipe)
-r *file* — True якщо файл можна читати (сценарієм користувача)
-s *file* — True якщо файл існує і не порожній
-S *file* — True якщо файл є сокет
-t fd — True якщо файловий дескриптор відкритий в терміналі
-u *file* — True якщо файл має встановлені права доступу користувача
-w *file* — True якщо у файл можна записати (сценарієм користувача)
-x *file* — True якщо файл можна виконати (сценарієм користувача)
-z *s* — (*zero length*) True якщо стрічка порожня

`[ file1 -nt file2 ]`, `file1` новіший від `file2`;
`[ file1 -ot file2 ]`, `file1` старіший від `file2`;

Для створення складних умов перевірки використовується булева логіка:

`[ умова 1 ] && [ умова 2 ]`
`[ умова 1 ] || [ умова 2 ]`

Розширений варіант команди `test` – подвійні квадратні дужки `[[ умова ]]` забезпечують *розширені умови порівняння символьних рядків*, наприклад порівняння із шаблонами регулярних виразів

```
[[ $USER == r* ]]
```

Обчислення арифметичних виразів.

Арифметичні вирази обчислює вбудована команда `let`. Команда `let` сприймає стрічку, яка містить ім’я змінної, знак дорівнює і вираз для обчислення.

```
$ let ”SUM=5+5”
$ printf ”%d” $SUM
10

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

Якщо змінна декларована як integer (ключ `-i`), то `let` команда виконується як опція

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

Приклад округлення до найближчого 10

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

Якщо змінна декларована як символьна, то їй будуть назначатися символьні значення

```
$ unset SUM
$ declare SUM=0
$ SUM=SUM+5
$ printf ”%s\n” $SUM
SUM+5
```

Команда `let` дозволяє виконувати наступні операції:

```
-, + — унарний мінус і плюс
!, ~ — логічну і бітову інверсію
*, /, % — ділення, множення, залишок від ділення
+, - — додавання, віднімання
<<, >> — лівий, правий бітовий зсув
<=, >=, <, > - порівняння
==, != — рівність, нерівність
& — побітове I (AND)
^ — побітове виключальне АБО ( XOR )
| — побітове АБО ( OR )
&& — логічне AND
|| — логічне OR
expr ? expr1 : expr2 – інструкція умови
=, *=, /=, %= — присвоєння
+=, -=, <<=, >>=, &=, ^=, |= — операції самопосилання
```

Команда `let` може обробляти вісімкові і шістнадцяткові числа

```
$ declare -i OCTAL=0
$ let ”OCTAL=0775”
$ printf ”%i\n” ”$OCTAL”
509
```

Подвійні круглі дужки (( )).

Команда `let` має синонім – подвійні круглі дужки `(( … ))`. Вони використовуються для того, щоб вбудувати `let` вираз як параметр у інші команди.

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

Подвійні круглі дужки `(( вираз ))` дозволяють *обчислювати і використовувати в умовах складні арифметичні вирази* і опускати знаки долара для цілочислових змінних, змінних масивів, а також вставляти пропуски перед арифметичними операціями для зручності читання, наприклад

```
$ let ”var1=(( 2 ** 5 ))”
$ echo $var1
256
```

```
if  (( $var1 ** 2 > 32 )) ; then
echo ”більше 32”
fi
```

```
$ ((b=1,c=2))
$ ((a=b))
$ ((a++))
$ for ((i=0;i<5; i++)) echo $((a + b + c + i)))
6
7
8
9
```

Змінну можна задекларувати командою `typeset` як integer (ключ `-i`), то `let` команда виконується як опція:

```
$ typeset -i i=99
$ echo $i
99
$ i=0x80
$ echo $i
128
$ i=2#1101001
$ echo $i
105
$ (( i = 16#a5 + 16#120 ))
$ echo $i
453
```

В подвійних круглих дужках можуть використовуватися наступні командні символи: `var++`, `var--`, `++var`, `--var`, `!` (логічна інверсія), `~` (побітова інверсія), `**` (експонента), `<<`, `>>` (зсув бітів вліво, вправо), `&` (бітове AND), `|` (бітове OR), `&&` (логічне AND), `||` (логічне OR).

```
#!/bin/bash
a=1;b=2
sum1=$((1+2)); echo "sum1=$sum1"
((a++))
sum2=$(( a + b )); echo "sum2=$sum2"
((a+=1))
sum3=$(( $a + $b)); echo "sum3=$sum3"
((a++))
let "sum4 = $(( a + b ))"; echo "sum4=$sum4"
((a+=1))
let sum5=$((a+b)); echo "sum5=$sum5"
((a++))
sum6=$( expr $a + $b ); echo "sum6=$sum6"
((a++))
((sum7=$a + $b)); echo "sum7=$sum7"
sum1=3
sum2=4
sum3=5
```

```
sum4=6
sum5=7
sum6=8
sum7=9
```

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

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

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

Значення змінним присвоюються як символи:

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

Перетворення символьного виразу в арифметичний:

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

Подвійні круглі дужки використовуються для виведення чисел у 2-й, 8-й, 10-й та 16-й системах числення:

```
$ echo ((2#101))
3
$ echo ((8#0x101))
65
$ echo ((10#101))
101
$ echo ((16#101))
257
```

Для перетворення між системами числення може використовуватися `bc` калькулятор з параметрами `ibase` (вхідна основа), `obase` (вихідна основа):

```
$ echo "ibase=10;obase=2;9" | bc
1001
```

Бітове подання чисел із знаком та без знаку:

```
#!/bin/bash
shopt -so xtrace
twos() {
  n=$(getconf LONG_BIT) # визначення архітектури машини 32bit чи 64bit
  printf 'obase=2; 2^%d+%d\n' "$n" "$1" | bc | sed -E "s/.*(.{$n})$/\1/"
}
twos $1

$./twos.sh 100
00000000000000000000000000000000000000000000000000000001100100

$./twos.sh -100
11111111111111111111111111111111111111111111111111111110011100
```

Побітові операції:

~ NOT

```
$ echo $(( ~ -3))
2
```

```
11111111111111111111111111111111111111111111111111111111111111101
------------------------------------------------------------------ NOT
00000000000000000000000000000000000000000000000000000000000000010
```

& AND

```
$ echo $((1 & 0))
0
```

```
$ echo $(( -1000 & 20 ))
16
```

| OR

```
$ echo $(( 123 | -321 ))
-257
```

```
$ A=123;
$ (( A |= -321 ))
$ echo $A
-257
```

^ XOR

```
$ echo $(( -314 ^ 537  ))
-801
```

```
$ A=-314;
$ (( A ^= 537 ))
$ echo $A
-801
```

<< Зсув вліво

```
$ n=3
$ x=-1000
$ echo $(( x << n ))
-8000
```

>> Зсув вправо

```
$ x=1030
$ n=3
$ echo $(( x >> n ))
128
```

При використання різних побітових операцій краще явно вказувати послідовність операцій у круглих дужках:

```
$ echo $(( 8 | (((~ 4) >> 1) ^ (2 & 6)) ))
-1
```


2. Практична частина

Контрольні запитання.

1. Який синтаксис інструкцій `if-then-fi`, `if-then-else-fi`?
2. Яке призначення команди `test` і який її синонім?
3. Які вирази можуть використовуватися в умовах порівняння?

4. Як обчислюються прості і складні арифметичні вирази?
5. Як порівнюються символьні вирази?
6. Розширені умови порівняння символьних виразів.
7. Як порівнюються файли?
8. Які ключі використовуються в умовах порівняння файлів?
9. Які утворюються складні умови перевірки?
10. Команда `let` і її синонім, особливості застосування?
11. Арифметичні команди і варіанти їх застосування.
12. Який синтаксис інструкції `case-esac`?
13. Як здійснити групування команд?
14. Яке логічне значення для символів `0, 1, -1`.
15. Яка різниця між командами `&` (бітове AND), `|` (бітове OR) та командами `&&` (логічне AND), `||` (логічне OR).

Завдання.

1. Написати сценарій, який використовує *curl* функція для пошуку веб сторінок в інтернеті які містять задане слово, введене з командного рядка. Вивести у відсортованому порядку *url* перших десяти сайтів і кількість використань заданого слова.

2. Написати сценарій, який використовує *curl* функція для пошуку заданої веб сторінок в інтернеті, введеної з командного рядка. Вивести кількість каталогів і файлів веб сайту.

3. Написати сценарій, який використовує *curl* функція для пошуку заданої веб сторінки в інтернеті і заданої теми, введених з командного рядка. Вивести імена файлів в яких задана тема згадувалася у поточному році.

4. Написати сценарій, який сприймає з командного рядка назву каталогу і виводить розміри каталогів і файлів як поточного, так і всіх вкладених каталогів.

5. Написати сценарій, який сприймає з командного рядка любе число імен файлів і виводить їх розміри для поточного каталогу.

6. Написати сценарій, який сприймає з командного рядка любе число імен каталогів і виводить їх розміри для поточного каталогу.

7. Написати сценарій, який сприймає з командного рядка шлях до каталогу та розмір і визначає які файли і каталоги є більшими від заданого розміру.

8. Написати сценарій, який сприймає з командного рядка шлях до каталогу та дату і визначає які файли і каталоги є старішими від заданої дати.

9. Написати сценарій, який сприймає з командного рядка шлях до каталогу та дату і визначає які файли і каталоги є молодшими від заданої дати і хто є їх власником.

10. Написати сценарій, який сприймає з командного рядка шлях до каталогу та дві дати і визначає які файли і каталоги були створені між цими датами і коли вони були модифіковані.

11. Написати сценарій, який сприймає з командного рядка шлях до каталогу і друкує кількість файлів і каталогів у всіх підкаталогах.

12. Написати сценарій який виводить на екран файли поточного каталогу, створені у минулому місяці з 9.00 до 12.00 години.

13. Написати сценарій, який виводить на екран відсортовані файли каталогу `/tmp`, вибрані за шаблоном `[a-xA-X]`, введеним з консольного рядка.

14. Написати сценарій, який зчитує з консолі числа і підраховує окремо суми парних і непарних чисел.

15. Написати сценарій, який зчитує з консолі день народження у форматі ДД.ММ.РР і визначає який це був день тижня і який це буде тиждень у поточному місяці.

Примітка. Номер варіанту завдання вибирається за порядковим номером студента у журналі групи.

Звіт з лабораторної роботи має містити:
- назву групи, П.І.Б. студента, завдання до роботи;
- короткий опис теоретичної частини;
- текст програми із коментаріями;
- роздрук екранів із результатом запуску програми.

Приклади

1. Перевірка користувача у системі

```
# 01.sh
#!/bin/bash
user=student
if grep $user /etc/passwd
then
echo ”Користувач $user відсутній”
fi
```

2. Тестування каталогів, опція -d

```
# 02.sh
#!/bin/bash
if [ -d $HOME ]
then
echo "Каталог HOME існує"
cd $HOME
ls -al
else
echo "Каталог HOME відсутній"
fi
```

3. Перевірка наявності об’єкту, опція -e

```
# 03.sh
#!/bin/bash
if [ -e $HOME ]
then
echo "Каталог" $HOME "існує, перевіримо наявність файлу test"
# Перевірка наявності файлу test
if [ -e $HOME/test ]
then
# якщо файл існує, то поповнити його даними
echo "Поповнення файлу даними"
date >> $HOME/test
cat $HOME/test
else
# якщо файл відсутній, створимо новий файл
echo "Створення нового файлу test"
date > $HOME/test
cat $HOME/test
fi
```

```
else
echo "В системі відсутній каталог" $HOME
fi
```

4. Перевірка наявності файлу, опція -f

```
# 04.sh
#!/bin/bash
if [ -e $HOME ]
then
echo "Об'єкт існує, а чи це файл?"
if [ -f $HOME ]
then
echo "Так, це файл!"
else
echo "Ні, це не файл!"
if [ -f $HOME/.bash_history ]
then
echo "Bash_history є файл!"
fi
fi
else
echo "Об'єкт не існує"
fi
```

5. Перевірка права на читання файлу, опція -e

```
# 05.sh
#!/bin/bash
pwfile=/etc/shadow
# перевірка чи файл існує
if [ -f $pwfile ]
then
# перевірка права на читання, опція -r
if [ -r $pwfile ]
then
tail $pwfile
else
echo "Ви не маєте права на читання файлу $pwfile"
fi
else
echo "Файл $pwfile не існує"
fi
```

6. Перевірка чи файл порожній, опція -s

```
# 06.sh
#!/bin/bash
file=test6
touch $file
if [ -s $file ]
then
echo "Файл $file існує і має дані"
else
echo "Файл $file існує, але порожній"
fi
date > $file
if [ -s $file ]
```

```
then
echo "Файл $file має дані"
else
echo "Файл $file все ще порожній"
fi
```

7. Перевірка чи в файл можна записувати (is writeable), опція -w

```
# 07.sh
#!/bin/bash
logfile=$HOME/test7
touch $logfile
chmod u-w $logfile
now=`date +%Y%m%d-%H%M`
if [ -w $logfile ]
then
   echo "Програма виконана: $now" > $logfile
   echo "Перша спроба успішна"
else
   echo "Перша спроба невдала"
fi

chmod u+w $logfile
if [ -w $logfile ]
then
   echo "Програма виконана: $now" > $logfile
   echo "Друга спроба вдала"
else
   echo "Друга спроба невдала"
fi
```

8. Перевірка чи файл можна виконувати, опція –w

```
# 08.sh
# створити файл сценарій test8.sh
#!/bin/bash
if [ -x test8.sh ]
then
echo "Ви можете виконати сценарій:"
./test8
else
echo "Вибачте, ви не можете виконати сценарій"
fi
```

9. Перевірка власника файлу, опція -O

```
# 09.sh
#!/bin/bash
if [ -O /etc/passwd ]
then
echo "Ви власник файлу  /etc/passwd file"
else
echo "Ви не власник файлу /etc/passwd file"
fi
```

10. Перевірка володіння файлу групою, опція -G

```
# 10.sh
# створити файл test10
```

```
#!/bin/bash
if [ -G $HOME/test10 ]
then
echo "Ви в такій групі як і файл"
else
echo "Ваша група не є власником файлу"
fi
```

11. Перевірка дати файлів, опції –nt, -ot

```
# 11.sh
# створити файл test11
#!/bin/bash
if [ ./test11 -nt ./test10 ]
then
echo "Файл test11 новіший від test10"
else
echo "Файл test10 новіший від test11"
fi
if [ ./test10 -ot ./test11 ]
then
echo "Файл test10 є старіший від test11"
fi
```

12. Складені порівняння

```
# 12.sh
#!/bin/bash
if [ -d $HOME ] && [ -w $HOME/test11 ]
then
echo "Файл існує і в нього можна записувати"
else
echo "У файл не можна записувати"
fi
```

13. Використання команди declare

```
# 13.sh
#!/bin/bash
# 13.bash: Convert Fahrenheit to Celsius
# CVS $Header$
shopt -s -o nounset
declare -i FTEMP # температура Фаренгейта
declare -i CTEMP # температура Цельсія
printf ”%s\n\n” ”Перетворення з Фаренгейта в Цельсій”
read -p ”Введіть температуру у Фаренгейтах: ” FTEMP
# let ”CTEMP=(5*(FTEMP-32))/9”
CTEMP=”(5*(FTEMP-32))/9”
printf ”Температура Цельсія %d\n” ”$CTEMP”
exit 0
```

14. Використання подвійних дужок для степені **

```
# 14.sh
#!/bin/bash
val1=10
if (( $val1 ** 2 > 90 ))
then
(( val2 = $val1 ** 2 ))
```

```
echo "$val1 в степені 2 буде $val2"
fi
```

15. Використання шаблону регулярного виразу * в команді [[ ]] (test)

```
# 15.sh
#!/bin/bash
if [[ $USER == r* ]]
then
echo "Вітаю $USER"
else
echo "На жаль, я не знаю Вас"
fi
```

16. Пошук необхідного значення в команді [ ] (test)

```
# 16.sh
#!/bin/bash
if [ $USER = "user" ]
then
echo "Запрошую $USER"
echo " до роботи"
elif [ $USER = ”user1” ]
then
echo "Запрошую  $USER"
echo " до роботи "
elif [ $USER = ”user2” ]
then
echo " до роботи"
elif [ $USER = ”user” ]
then
echo "Не забудьте вийти з системи (logout) по закінченню роботи"
else
echo "На жаль, Ви не маєте прав доступу"
fi
```

 # 17. Використання команди case

```
# 17.sh
#!/bin/bash
case $USER in
user | user1)
echo "Запрошуємо, $USER"
echo " до роботи";;
test)
echo "Перевірка конкретного користувача";;
user2)
echo "Не забудьте вийти із системи (log off) по закінченню роботи";;
*)
echo "На жаль Ви не маєте прав доступу";;
esac
```

 # Використання команди case з діапазонами значень

```
# 18.sh
#!/bin/bash
space_free=$( df –h | awk ’{ print $5 }’ | sort –n | tail –n 1 | sed ’s/%//’ )
echo $space_free
case $space_free in
```

```
  [1-20]*)
      echo ”Вільного простору HDD < 20%”
      ;;
  [21-40]*)
      echo ”Вільного простору HDD < 40%”
      ;;
  [41-60]*)
      echo ”Вільного простору HDD < 60%”
      ;;
  [61-80]*)
      echo ”Вільного простору HDD < 80”
      ;;
  *)
      echo ”Заповненість HDD > 81%”
;;
esac

$ ./18.sh
52
Вільного простору HDD < 60%
```

Лабораторна робота 4

Команди циклів

Мета роботи: вивчення команд циклів.

Теоретичний матеріал: лекції, технічна література, інтернет ресурси.

1. Короткі теоретичні відомості

Цикл

```
for in 1,2 .. N
do
  command
done
```

використовується, коли потрібно перебрати декілька значень змінної.

```
#!/bin/bash
echo -n "Друк в рядок десяти крапок"
for i in 1 2 3 4 5 6 7 8 9 10
do
   echo -n "."
done
..........
```

Цикл в діапазоні значень:

```
#!/bin/bash
for value in {1..5}
do
   echo -n ”$value ”
done
1 2 3 4 5
```

Цикл в діапазоні значень із кроком:

```
#!/bin/bash
for value in {10..0..2}
do
   echo -n $value
done
10 8 6 4 2 9

# seq {start..end[..increment]}
for i in $(seq 1 2 4)
do
 echo "skip by 2 value $i"
done
skip by 2 value 1
skip by 2 value 3

echo {0000..0010..2}
0000 0002 0004 0006 0008 0010

seq 2 2 4
2
```

```
4

# -s separator
seq -s ’ ’ 2 2 6
2 4 6

seq -s , 2 2 6
2,4,6
```

Цикл по словах в стрічці символів:

```
str="aaa bbb ccc"
for i in $str
do
   echo $i
done
aaa
bbb
ccc
```

Використання команди seq:

```
$ seq -s ' ' 1 5
1 2 3 4 5

$ var="$(seq -s ' ' 1 5)"
$ echo "$var"
1 2 3 4 5
```

Цикл по символах слова:

```
word="abcd"
for i in $(seq 1 ${#word})
do
 echo "${word:i-1:1}"
done
a
b
c
d
```

Цикл по результату виконання команди:

```
for f in $(ls /user/*.pdf)
do
   print "File $f"
done
```

Добавлення розширення .txt для всіх файлів поточного каталогу:

```
#!/bin/bash
for file in *
do
  echo "Добавлення розширення .txt для файлу $file ..."
  mv $file $file.txt
  sleep 1
done
```

Кількість елементів у масиві:

```
Array=("Фіолетовий" "Темно синій" "Голубий")
```

```
echo "n@ = ${#Array[@]}"
echo "n* = ${#Array[*]}"
n@ = 3
n* = 3
```

Кількість індексів у масиві:

```
echo "n@ = ${!Array[@]}"
echo "n* = ${!Array[*]}"
n@ = 0 1 2
n* = 0 1 2
```

Довжина першого елементу масиву (нумерація з нуля)

```
echo ${#arr[0]}
10
```

Цикл по елементах масиву

```
Array=("Фіолетовий" "Темно синій" "Голубий")
for i in "${Array[@]}"
do
  echo $i
done
Фіолетовий
Темно синій
Голубий

for i in "${Array[*]}"
do
  echo $i
done
Фіолетовий Темно синій Голубий
```

Цикл по індексах елементів масиву

```
Array=("Фіолетовий" "Темно синій" "Голубий")
for i in "${!Array[@]}"
do
  echo $i " : " "${Array[$i]}"
done
0  :  Фіолетовий
1  :  Темно синій
2  :  Голубий
```

Вплив лапок на розгортання елементів масиву:

```
array=("first item" "second item" "third" "item")

echo "Число елементів в оригінальному масиві: ${#array[*]}"
for ix in ${!array[*]}
do
    printf "   %s\n" "${array[$ix]}"
done
echo

arr=(${array[*]})
```

```
echo " * після розширення без лапок: ${#arr[*]}"
for ix in ${!arr[*]}
do
    printf "   %s\n" "${arr[$ix]}"
done
echo

arr=("${array[*]}")
echo " * після розширенням з лапками: ${#arr[*]}"
for ix in ${!arr[*]}
do
    printf "   %s\n" "${arr[$ix]}"
done
echo

arr=(${array[@]})
echo " @ після розширення без лапок: ${#arr[*]}"
for ix in ${!arr[*]}
do
    printf "   %s\n" "${arr[$ix]}"
done
echo

arr=("${array[@]}")
echo " @ після розширення з лапками: ${#arr[*]}"
for ix in ${!arr[*]}
do
    printf "   %s\n" "${arr[$ix]}"
done

#Створення копії файлів каталогу $HOME
$ files=($HOME/*.bash); cp "${files[@]}" ./backups
```

```
Число елементів в оригінальному масиві: 4
   first item
   second item
   third
   item

 * після розширення без лапок: 6
   first
   item
   second
   item
   third
   item

 * після розширенням з лапками: 1
   first item second item third item

 @ після розширення без лапок: 6

 first
   item
```

```
second
item
third
item

@ після розширення з лапками: 4
first item
second item
third
item
```

Вкладені цикли `for`:

```
#!/bin/bash
# IFS – internal field separator
IFS=:
for dir in $PATH
do
  echo ”$dir:”
  for myfile in $dir/*
  do
   if  [ -x $myfile ]
   then
     echo ”$myfile”
   fi
  done
done
```

Цикли у стилі мови програмування С:

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

Інструкція `while умова` використовується для організації циклів. Вона виконується поки `умова` істинна:

```
#!/bin/bash
while true
do
  echo "Натисніть CTRL-C для виходу."
  sleep 1
done
```

`true` – програма, яка запускає тіло циклу на повторення. Використання `true` вважається повільним, так як сценарій має її запускати в кожній ітерації. У альтернативному варіанті використовується вбудована функція Bash “:”

```
#!/bin/bash
while :
do
  echo " Натисніть CTRL-C для виходу."
sleep 1
done
```

Приклад організації циклу з умовою більше:

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

```
read -p "Введіть число: " num
while [ "$num" -gt 5 ]; do
echo "$num"
num=$(( "$num" - 1 ))
done
```

Приклад організації циклу з умовою менше дорівнює:

```
#!/bin/bash
x=0;
while [ ”$x” -le 5 ]
do
  echo "Поточне значення x: $x"
  # Збільшуємо значення x:
  # x=$(expr $x + 1)
  # ((x++))
  x=$(( $x + 1 ))
done
```

Запис циклу в один рядок:

```
x=0; while  [ $x -le 5 ]; do echo "Поточне значення x: $x" $(( x++ )); done
```

Обчислення факторіалу:

```
#!/bin/bash
counter=$1
factorial=1
while [ $counter -gt 0 ]
do
   factorial=$(( $factorial * $counter ))
   counter=$(( $counter - 1 ))
done
echo "$factorial"
```

While для читання файлу і друку (повільно):

```
while read -r line; do
    echo "$line changed" >> output.txt
done < 1m_lines.txt
```

Readarray читання файлу і друку (швидко):

```
readarray -t lines < 1m_lines.txt
for i in "${!lines[@]}"; do
    lines[$i]="${lines[$i]} changed"
done
printf "%s\n" "${lines[@]}" > output.txt
```

Інструкція `until умова` - виконується до того часу поки `умова` не стане істинною. Так в прикладі цикл зупиниться, коли величина x досягне значення 10.

```
#!/bin/bash
x=0
until [ ”$x” -ge 10 ]
do
   echo ”Поточне значення x дорівнює $x”
   # x=$(expr $x + 1)
```

```
    x=$(( $x + 1 ))
    sleep 1
done
```

Для виходу із тіла циклу використовується інструкція `break [ n ]`, де `n` – номер вкладеності зовнішнього циклу.

Для продовження циклу використовується інструкція `continue [ n ]`, де `n` – номер вкладеності зовнішнього циклу.

Перенаправлення даних із циклу:

- перенаправлення у файл `do … done > out.txt`
- перенаправлення в іншу команду `do … done | sort`

Перенаправлення даних у цикл:

```
#!/bin/bash
#usage: друкування рядків файлу
# ./print_file.sh <file.txt>
while read line; do
    echo "$line"
done < $1
```

```
#!/bin/bash
#usage: створення каталогів із списку заданого у файлі
while read line; do
    mkdir "$line"
done < "$1"
```

Процес підставлення `<(…)` дозволяє трактувати вихід одної або декількох команд як файл та направити його у цикл:

```
<(команда)
<(команда | команда | … | команда)
```

Приклад ітерації по кожну рядку виходу команди `ls`:

```
#!/bin/bash
# читання і друк відсортованого виходу команди ls
while read line; do
echo "$line"
done < <(ls -l $HOME | sort)
```


2. Практична частина

Контрольні запитання.

1. Як працює інструкція `for` з перебором серії значень?
2. Цикл for у стилі C.
3. Використання у циклі `for` різних змінних і перенаправлення виведення у файл.
4. Як прочитати дані із файлу за допомогою інструкції `for`?
5. Вкладені цикли `for`.
6. Як вивести список каталогів і файлів за допомогою інструкції `for`?
7. Інструкція циклів `while`. Команди `true`, «`:`».
8. Організація циклу `while` з використанням змінних.
9. Організація циклу `until` з використанням команд і змінних.
10. Цикл по файлах каталогу.

11. Інструкції для виходу і продовження циклу.

12. Як перенаправляються дані із циклу?

13. Як перенаправляються дані у циклу?

14. Операція підставлення `<(…)`.

Завдання.

1. Написати сценарій, який використовує цикл для виведення переліку вкладених файлів і каталогів каталогу $HOME.

2. Написати сценарій, який використовує цикл для об'єднання двох масивів `mas1=(2,4,6,8,10), mas2=(1,3,7,9,11)` в один і виводить його у відсортованому порядку.

3. Написати сценарій, який використовує цикл для запису у масив окремих слів із стрічкової змінної `str="Як ваше здоров'я? Якщо погано, то перестаньте дихати димом електронних цигарок, який є канцерогеном і підвищує ризики хвороб серця, раку хронічного обструктивного захворювання легень, і пити енергетики, які короткочасно мобілізують резерви організму, що в результаті спричиняє підвищену втому, порушення сну, нервові розлади та депресію"` і виводить їх у відсортованому порядку за спаданням.

4. Написати сценарій який перетворює довільний асоціативний масив у інший асоціативний масив, у якого ключі стають значеннями, а значення - ключами.

5. Написати сценарій який об'єднує два асоціативні масиви в один звичайний масив без повторення слів.

6. Написати сценарій для посимвольного читання значень двох стрічкових змінних x="1234", y="abcd" і запису пар символів <ключ> <значення> в асоціативний масив.

7. Написати сценарій, який у циклі обчислює значення e^x, як степеневого ряду Тейлора, а значення `x` вводиться з консолі.

8. Написати сценарій, який використовує цикли у стилі C для матричного множення елементів двох двовимірних масивів

```
mas1=1 2 3 4 5
     1 3 5 7 9
mas2=6 7 8 9 10
     5 4 3 2 1
```

9. Написати сценарій, який в циклі в діапазоні 1-100 виводить прості числа і їх суму.

10. Написати сценарій, який у циклі обчислює суму ряду Маклорена $(1-x)^{-1}$, значення `x` вводиться з консолі. Вивести значення суми для 10, 50 і 100 елементів.

11. Написати сценарій, який зчитує з файлу текст з кириличними буквами, в циклі шифрує його буквами англійської абетки із заданим зміщенням. Ім'я файлу, величина зміщення і режим шифрування вводиться з консолі. Зашифрований файл записати у файл. Сценарій має мати режими шифрування і дешифрування.

12. Написати сценарій, який зчитує з файлу довільну послідовність чисел і виводить на екран окремо прості, парні і непарні числа.

13. Написати сценарій, який зчитує з файлу довільний англійський текст і виводить його на екран як "поросячу латину". https://uk.wikipedia.org/wiki/Поросяча_латина

14. Написати сценарій, який із заданого каталогу зчитує файли і записує <ім'я власника> <назва файлу> в асоціативний масив. Вивести з асоціативного масиву імена файлів згруповані за їх власниками.

15. Написати сценарій, який із заданого каталогу зчитує файли і записує <розширення файлу> <кількість файлів> в асоціативний масив. Вивести з асоціативного масиву відсортовані за кількістю файлів розширення файлів.

16. Написати сценарій, який зчитує з файлу довільний символьний рядок і виводить на екран окремо малі/великі голосні і приголосні букви та цифри.

Примітка. Номер варіанту завдання вибирається за порядковим номером студента у журналі групи.

Звіт з лабораторної роботи має містити:

- назву групи, П.І.Б. студента, завдання до роботи;
- короткий опис теоретичної частини;
- текст сценарію із коментаріями;
- роздрук екранів із результатом запуску сценарію.

Приклади

1. Команда циклу

```
# 01.sh
#!/bin/bash
# basic for command
for test in Alabama Alaska Arizona Arkansas California Colorado
do
   echo The next state is $test
done
```

$ bash 01.sh

```
The next state is Alabama
The next state is Alaska
The next state is Arizona
The next state is Arkansas
The next state is California
The next state is Colorado
```

2. Цикл читання масиву

```
# 02.sh
#!/bin/bash
languages=("Bash PERL Python PHP")
for language in $languages
do
  if [ $language == 'PHP' ]
  then
    echo "$language є мова web програмування"
  else
    echo "$language є мова сценаріїв"
  fi
done
```

$ bash 02.sh

```
Bash є мова сценаріїв
PERL є мова сценаріїв
Python є мова сценаріїв
PHP є мова web програмування
```

3. Слова з апострофами

```
# 03.sh
#!/bin/bash
for test in I don ’t know if this’ll work
do
  echo "слово:$test"
done
```

$ bash 03.sh

```
слово:I
слово:don
слово:’t
слово:know
слово:if
слово:this’ll
слово:work
```

4. Слова з екранованими апострофами

```
# 04.sh
#!/bin/bash
for test in I don\’t know if "this’ll" work
do
   echo "word:$test"
done
```

 # bash 04.sh

```
word:I
word:don’t
word:know
word:if
word:this’ll
word:work
```

5. Читання діапазону значень

```
# 05.sh
#!/bin/bash
for num in {10..30..5}
do
  # Перевірка чи число ділиться на 10
  if (( $num%10==0 ))
  then
    echo "$num ділиться на 10"
  fi
done
```

$ bash 05.sh

```
10 ділиться на 10
20 ділиться на 10
30 ділиться на 10
```

6. Читання діапазону значень у зворотному порядку

```
# 06.sh
#!/bin/bash
```

```
echo "Наступні числа діляться на 5 і 10"
for num in {50..30..10}
do
  # перевірка чи число ділиться на 5 і 10t
  if (( $num%5==0 && $num%10==0 ))
  then
    echo -n "$num "
  fi
done
```

$ bash 06.sh

```
Наступні числа діляться на 5 і 10
50 40 30
```

7. Цикл у стилі Сі

```
# 07.sh
#!/bin/bash
sum=0
for (( n=1; n<=50; n++ ))
do
  # додавання до суми
  ((sum=$sum+$n))
done
echo "Сума від 1 до 50 є $sum"
```

$ bash 07.sh

```
Сума від 1 до 50 є 1275
```

8. Цикл з двома умовами

```
# 08.sh
#!/bin/bash
#!/bin/bash
for (( x=5,y=25; x<=20 && y>5; x+=5,y-=5 ))
do
  echo "Поточне значення x=$x і y=$y"
done
```

$ bash 08.sh

```
Поточне значення x=5 і y=25
Поточне значення x=10 і y=20
Поточне значення x=15 і y=15
Поточне значення x=20 і y=10
```

9. Нескінченний цикл

```
# 09.sh
#!/bin/bash
# нескінченний цикл
for (( ; ; ))
do
  # введення даних
  echo "Введіть число між 1 і 20"
  read n
  # умова завершення циклу
  if [ $n == "quit" ]
  then
```

```
    echo "Програма завершена"
    exit 0
  fi
  # перевірка діапазону числа
  if [[ $n -lt 1 || $n -gt 20 ]]
  then
    echo "Число за межами діапазону"
  else
    echo "Число є $n"
  fi
done
```

$ bash 09.sh

```
Введіть число між 1 і 20
5
Число є 5
Введіть число між 1 і 20
25
Число за межами діапазону
Введіть число між 1 і 20
quit
Програма завершена
```

10. Ітерація по файлах каталогу

```
# ім'я каталогу і файлу взято в лапки бо в імені може бути символ пропуску
# 10.sh
#!/bin/bash
for file in /home/user1/*
do
  if [ -d "$file" ]
  then
     echo "$file : каталог"
  elif [ -f "$file" ]
  then
     echo "$file : файл"
  fi
done
```

$ bash 10.sh

```
1.sh : файл
2.sh : файл
3.sh : файл
4.sh : файл
```

11. Читання з виходу команди find

```
# 11.sh
#!/bin/bash
# Встановлення поля розділювача
IFS=$'\n';
# Читання файлів каталогу
for file in $(find "*.bash"); do
  echo $file
done
# відміна поля розділювача
unset IFS;
```

```
$ bash 11.sh
./test1.sh
./test2.sh
./test3.sh
./test4.sh
```

12. Рекурсивне читання файлів заданого каталогу командою find

```
# 12.sh
#!/bin/bash
# Встановлення поля розділювача
IFS=$'\n';
# Читання файлів каталогу
for file in $(find "$HOME"); do
  echo $file
done
# відміна поля розділювача
unset IFS;
```

```
$ bash 12.sh
/home/user/bash/test1.sh
/home/user/bash/test2.bash
/home/user/bash/test3.bash
/home/user/1.txt
/home/user/2.txt
```

13. Рекурсивне читання файлів заданого каталогу командою find з умовою

```
# 13.sh
#!/bin/bash
# Встановлення поля розділювача
IFS=$'\n';
# Читання файлів каталогу
for file in $(find "$HOME" -iname '*.txt'); do
  echo $file
done
# відміна поля розділювача
unset IFS;
```

```
$ bash 13.sh
/home/user/bash/example.txt
/home/user/DGP/input.txt
/home/user/DGP_arx/input.txt
/home/user/DGP_arx/arx/tpoint.txt
```

14. Поелементне читання списку файлів

```
# 14.sh
#!/bin/bash
# Встановлення поля розділювача
IFS=$'\n';
#  поелементне читання списку файлів
for line in $(apt list --installed)
do
  echo "$line"
done
```

```
# Відміна поля розділювача
unset IFS;
```

$ bash 14.sh

```
ubuntu-keyring/focal-updates,now 2020.02.11.4 all [installed,automatic]
ubuntu-minimal/focal-updates,now 1.450.2 amd64 [installed]
ubuntu-release-upgrader-core/now   1:20.04.37   all   [installed,upgradable   to:
1:20.04.41
...
```

15. Порядкове читання файлу

```
# 15.sh
#!/bin/bash
# Встановлення поля розділювача
IFS=$'\n';
#  порядкове читання файлу
for line in $(cat temp.txt)
do
  echo "$line"
done
# Відміна поля розділювача
unset IFS;
```

$ bash 15.sh

```
Рядок 1
Рядок 2
...
```

16. Команда while

```
# 16.sh
#!/bin/bash
var1=5
while [ $var1 -gt 0 ]
do
  echo $var1
  var1=$[ $var1 - 1 ]
done
```

$ bash 16.sh

```
5
4
3
2
1
```

17. Мультикоманда while

```
# 17.sh
#!/bin/bash
var1=5
while echo $var1
[ $var1 -gt 0 ]
do
  echo "В середині циклу"
  var1=$(( $var1 - 1 ))
```

```
done
```

$ bash 17.sh

```
5
В середині циклу
4
В середині циклу
3
В середині циклу
2
В середині циклу
1
В середині циклу
0
```

18. Команда until

```
# 18.sh
#!/bin/bash
var1=100
until [ $var1 -eq 0 ]
do
  echo $var1
  var1=$(( $var1 - 25 ))
done
```

$ bash 18.sh

```
В середині циклу 100
В середині циклу 75
В середині циклу 50
В середині циклу 25
```

19. Мультикоманда until

```
# 19.sh
#!/bin/bash
var1=100
until echo $var1
[ $var1 -eq 0 ]
do
  echo В середині циклу: $var1
  var1=$(( $var1 - 25 ))
done
```

$ bash 19.sh

```
100
В середині циклу: 100
75
В середині циклу: 75
50
В середині циклу: 50
25
В середині циклу: 25
0
```

20. Вбудовані цикли

```
# 20.sh
```

```
#/!bin/bash
for (( a=1; a<=3; a++ ))
do
  echo "Зовнішній цикл $a:"
  for (( b=1; b<=3; b++ ))
  do
    echo "Внутрішній цикл: $b"
  done
done
```

$ bash 20.sh

```
Зовнішній цикл 1:
Внутрішній цикл: 1
Внутрішній цикл: 2
Внутрішній цикл: 3
Зовнішній цикл 2:
Внутрішній цикл: 1
Внутрішній цикл: 2
Внутрішній цикл: 3
Зовнішній цикл 3:
Внутрішній цикл: 1
Внутрішній цикл: 2
Внутрішній цикл: 3
```

21. Розміщення циклу for всередині циклу while

```
# 21.sh
#!/bin/bash
var1=5
while [ $var1 -ge 0 ]
do
  echo "Зовнішній цикл: $var1"
  for (( var2=1; $var2<3; var2++ ))
  do
    var3=$[ $var1 * $var2 ]
    echo " Внутрішній цикл: $var1 * $var2 = $var3"
  done
  var1=$[ $var1 - 1 ]
done
```

$ bash 21.sh

```
Зовнішній цикл: 5
 Внутрішній цикл: 5 * 1 = 5
 Внутрішній цикл: 5 * 2 = 10
Зовнішній цикл: 4
 Внутрішній цикл: 4 * 1 = 4
 Внутрішній цикл: 4 * 2 = 8
Зовнішній цикл: 3
 Внутрішній цикл: 3 * 1 = 3
 Внутрішній цикл: 3 * 2 = 6
Зовнішній цикл: 2
 Внутрішній цикл: 2 * 1 = 2
 Внутрішній цикл: 2 * 2 = 4
Зовнішній цикл: 1
 Внутрішній цикл: 1 * 1 = 1
 Внутрішній цикл: 1 * 2 = 2
```

```
Зовнішній цикл: 0
 Внутрішній цикл: 0 * 1 = 0
 Внутрішній цикл: 0 * 2 = 0
```

22. Цикли until і while

```
# 22.sh
#!/bin/bash
var1=3
until [ $var1 -eq 0 ]
do
  echo "Зовнішній цикл: $var1"
  var2=1
  while [ $var2 -lt 5 ]
  do
     var3=`echo "scale=4; $var1 / $var2" | bc`
     echo "Внутрішній цикл: $var1 / $var2 = $var3"
     var2=$(( $var2 + 1 ))
  done
  var1=$(( $var1 - 1 ))
done
```

$ bash 22.sh

```
Зовнішній цикл: 3
Внутрішній цикл: 3 / 1 = 3.0000
Внутрішній цикл: 3 / 2 = 1.5000
Внутрішній цикл: 3 / 3 = 1.0000
Внутрішній цикл: 3 / 4 = .7500
Зовнішній цикл: 2
Внутрішній цикл: 2 / 1 = 2.0000
Внутрішній цикл: 2 / 2 = 1.0000
Внутрішній цикл: 2 / 3 = .6666
Внутрішній цикл: 2 / 4 = .5000
Зовнішній цикл: 1
Внутрішній цикл: 1 / 1 = 1.0000
Внутрішній цикл: 1 / 2 = .5000
Внутрішній цикл: 1 / 3 = .3333
Внутрішній цикл: 1 / 4 = .2500
```

23. Зміна IFS значення

```
# 23.sh
#!/bin/bash
IFS_OLD=$IFS
IFS=$'\n'
for entry in `cat /etc/passwd`
do
  echo "Значення в $entry -"
  IFS=:
  for value in $entry
  do
    echo " $value"
  done
done
IFS=$IFS_OLD
```

$ bash 23.sh

```
Значення в at:x:25:25:Batch jobs daemo -
 at
 x
 25
 25
 Batch jobs daemo
Значення в :/var/spool/atjobs:/bi -

 /var/spool/atjobs
 /bi
Значення в /bash
avahi:x:481:481:User for Avahi:/ru -
 /bash
 ...
```

24. Виконання фонового процесу у циклі

```
# 24.sh
#!/bin/bash
# ініціалізація лічильника
cnt=1
# умова завершення
until [ $cnt -ge 20 ]
do
  echo "Фоновий процес виконується";
  # чекати 1 секунду
  sleep 1;
  ((cnt++))
done > output.txt &
# чекати 3 секунди
echo "Затримка..."
sleep 3
```

$ bash 24.sh

```
Затримка...
user@DELL:~/bash/$ cat output.txt
Фоновий процес виконується
Фоновий процес виконується
Фоновий процес виконується
Фоновий процес виконується
Фоновий процес виконується
```

25. Вихід (breaking) з циклу for

```
# 25.sh
#!/bin/bash
for var1 in 1 2 3 4 5 6 7 8 9 10
do
  if [ $var1 -eq 5 ]
  then
    break
  fi
  echo "Номер ітерації: $var1"
done
echo "Закінчення for циклу"
```

$ 25.sh

```
Номер ітерації: 1
Номер ітерації: 2
Номер ітерації: 3
Номер ітерації: 4
Закінчення for циклу
```

26. Вихід (breaking) з while циклу

```
# 26.sh
#!/bin/bash
var1=1
while [ $var1 -lt 10 ]
do
  if [ $var1 -eq 5 ]
  then
    break
  fi
  echo "Iteration: $var1"
  var1=$(( $var1 + 1 ))
done
echo "Вихід з циклу loop"
```

$ bash 26.sh

```
Ітерація: 1
Ітерація: 2
Ітерація: 3
Ітерація: 4
Вихід з циклу loop
```

27. Вихід (breaking) з внутрішнього циклу

```
# 27.sh
#!/bin/bash
for (( a=1; a < 4; a++ ))
do
 echo "Зовнішній цикл: $a"
 for (( b=1; b<100;  b++ ))
 do
   if [ $b -eq 5 ]
   then
     break
   fi
   echo "Внутрішній цикл: $b"
 done
done
```

$ bash 27.sh

```
Зовнішній цикл: 1
Внутрішній цикл: 1
Внутрішній цикл: 2
Внутрішній цикл: 3
Внутрішній цикл: 4
Зовнішній цикл: 2
Внутрішній цикл: 1
Внутрішній цикл: 2
Внутрішній цикл: 3
Внутрішній цикл: 4
```

```
Зовнішній цикл: 3
Внутрішній цикл: 1
Внутрішній цикл: 2
Внутрішній цикл: 3
Внутрішній цикл: 4
```

28. Вихід (breaking) із зовнішнього циклу

```
# 28.sh
#!/bin/bash
for (( a = 1; a < 4; a++ ))
do
echo "Зовнішній цикл: $a"
  for (( b = 1; b < 100; b++ ))
  do
    if [ $b -gt 4 ]
    then
      break 2
    fi
    echo "Внутрішній цикл: $b"
  done
done
```

$ bash 28.sh

```
Зовнішній цикл: 1
Внутрішній цикл: 1
Внутрішній цикл: 2
Внутрішній цикл: 3
Внутрішній цикл: 4
```

29. Використання команди continue

```
# 29.sh
#!/bin/bash
for (( var1 = 1; var1 < 15; var1++ ))
do
  if [ $var1 -gt 5 ] && [ $var1 -lt 10 ]
  then
    continue
  fi
  echo "Номер ітерації: $var1"
done
```

$ bash 29.sh

```
Номер ітерації: 1
Номер ітерації: 2
Номер ітерації: 3
Номер ітерації: 4
Номер ітерації: 5
Номер ітерації: 10
Номер ітерації: 11
Номер ітерації: 12
Номер ітерації: 13
Номер ітерації: 14
```

30. Продовження зовнішнього циклу

```
# 30.sh
```

```
#!/bin/bash
for (( a = 1; a <= 5; a++ ))
do
  echo "Ітерація $a:"
  for (( b = 1; b <  3; b++ ))
  do
    if [ $a -gt 2 ] && [ $a -lt 4 ]
    then
      continue 2
    fi
    var3=$(( $a * $b ))
    echo "Результат $a * $b =  $var3"
  done
done
```

$ bash 30.sh

```
Ітерація 1:
Результат 1 * 1 = 1
Результат 1 * 2 = 2
Ітерація 2:
Результат 2 * 1 = 2
Результат 2 * 2 = 4
Ітерація 3:
Ітерація 4:
Результат 4 * 1 = 4
Результат 4 * 2 = 8
Ітерація 5:
Результат 5 * 1 = 5
Результат 5 * 2 = 10
```

31. Перенаправлення виходу в файл

```
# 31.sh
#!/bin/bash
for (( a = 1; a < 10; a++ ))
do
  echo "Число $a"
done > /tmp/29.sh.out
echo "stdout з /tmp.sh.29.out"
cat /tmp/sh.29.out
rm  /tmp/sh.29.out
```

$ bash 31.sh

```
stdout з /tmp.sh.29.out
Число 1
Число 2
Число 3
Число 4
```

32. Направлення (piping) даних циклу в іншу команду

```
# 32.sh
#!/bin/bash
for  i  in  "Івано-Франківська"  "Львівська"  "Донецька"  "Київська"  "Чернівецька"
"Луганська"
do
echo $i
```

```
done | sort
echo "Області відсортовані за абеткою"
```

; bash 32.sh

```
Івано-Франківська Львівська Донецька Київська Чернівецька Луганська
Донецька
Івано-Франківська
Київська
Луганська
Львівська
Чернівецька
Області відсортовані за абеткою
```

33. Направлення (piping) даних циклу в іншу команду

```
# 33.sh
#!/bin/bash
obl=("Івано-Франківська"    "Львівська"    "Донецька"    "Київська"    "Чернівецька"
"Луганська")
echo ${obl[@]}
for i in  ${obl[@]}
do
echo $i
done | sort -r
echo "Області реверсно відсортовані за абеткою "
```

$ bash 33.sh

```
Івано-Франківська Львівська Донецька Київська Чернівецька Луганська
Чернівецька
Львівська
Луганська
Київська
Івано-Франківська
Донецька
Області реверсно відсортовані за абеткою
```

34. Отримання списку виконуваних процесів

```
# 34.sh
!/bin/bash
# 5 ітерацій виконання команди top
for ((i=0; i<5; i++))
do
  top -b -i -n2 | tail -1 >> output.txt
done
cat output.txt
```

$ bash 34.sh

```
PID   USER PR  NI  VIRT   RES   SHR   S  %CPU   %MEM    TIME+    COMMAND
3274  user 20  0   43564  4224  3592  R  6.250  0.106   0:00.01  top
```

35. Процес підставлення <(…)

```
# 35.sh
#!/bin/bash
# читання і друк відсортованого виходу команди ls
while read line; do
```

```
   echo "$line"
done < <(ls -l $HOME | sort)
```

```
$ bash 35.sh
-rw-r--r-- 1 user user    79 Sep 11 19:50 11
-rw-r--r-- 1 user user   123 Sep 11 16:14 112
-rw-r--r-- 1 user user   199 Feb 28  2023 22
drwxr-xr-x 2 user user  4096 Feb  4  2023 zip
```

Лабораторна робота 5

Параметри і ключі командного рядка

Мета роботи: навчитися працювати із параметрами і ключами командного рядка.

Теоретичний матеріал лекції, технічна література, ресурси Інтернету.

1. Короткі теоретичні відомості

1.1. Параметри

В Bash використовуються наступні типи параметрів і ключів:

- позиційні;
- параметри ключі;
- параметри ключі команди `getopts`;
- параметри ключі команди `getopt`;

Оболонка Bash назначає передані параметри спеціальним позиційним змінним `$0, $1,… $9`, де:

```
$0 – шлях до сценарію та ім’я сценарію;
$1 – перший параметр;
…
$9 – дев’ятий параметр.
```

Якщо параметрів більше як 9, то вони розміщуються у спеціальних змінних `${10}, ${11}, … .`

Найпростіше передати дані у сценарій за допомогою командного рядка, наприклад

```
# test.sh
#!/bin/bash
echo $0 $1 $2
```

```
$ ./test.sh 10 20
test.sh 10 20
```

В оболонці Bash використовуються також додаткові спеціальні змінні для роботи з параметрами:

`$0` – ім’я Bash сценарію.

`$1-$9` – перші 9 аргументів Bash сценарію.

`$#` - число переданих аргументів в сесії Bash або в сценарії.

`$@, $*` повертають всі параметри. Інтерпретація параметрів залежить від того, чи взяті спеціальні змінні у подвійні дужки чи ні. У цьому випадку:

`$@, $*` - завжди розділяє параметри символом пропуску і *розбиває параметри на окремі слова*, навіть якщо вони взяті *в подвійні лапки*.

`$@` часто використовується для передачі набору ключів іншій команді, наприклад `ls $@`.

`# test.sh`
`#!/bin/bash`
`echo $@`
`touch $@`
`$ ./test.sh ″a b″ ″c d″`
	`# test.sh`
	`#!/bin/bash`
	`echo $*`
	`touch $*`
	`$ ./test.sh ″a b″ ″c d″`

Будуть створені файли `a, b, c, d.`	Будуть створені файли `a, b, c, d.`

"$@" - не розбиває параметри у подвійних лапках на окремі слова і розділяє параметри символом пропуску.

"$" - не розбиває параметри у подвійних лапках на окремі слова* і об'єднує параметри в один параметр символом IFS (internal field separator, змінна Bash середовища `$IFS`).

```
# test.sh
#!/bin/bash
echo "$@"
touch "$@"
```

$ ./test.sh "a b" "c d"

Будуть створені файли

'a b', 'c d'

```
# test.sh
#!/bin/bash
echo "$*"
touch "$*"
```

$ ./test.sh "a b" "c d"

Буде створений файл

'a b c d'

```
# test.sh
#!/bin/bash
IFS=,;echo "$*"
touch "$*"
```

$ ./test.sh "a b" "c d"

Буде створений файл

'a b,c d'

Спеціальна змінна *"$*"* використовується для створення CSV (comma separated value) файлів:

```
# test.sh
#!/bin/bash
IFS=,
echo "$*"
```

```
./test.sh "1 2" 3 "4 5" 6 > out
cat out
1 2,3,4 5, 6
```

Якщо перший символ `$IFS` є символом пропуску – то розділювач пропуск, якщо символ невстановлений (`unset`) – то розділювач відсутній.

Розглянемо різницю між цими спеціальними змінними на прикладах:

```
# test.sh
#!/bin/bash
echo "Метод \$@: $@"
echo "Метод \$*: $*"
```

```
$ ./test.sh 1 2 3
Метод \$@: 1 2 3
Метод \$*: 1 2 3
```

Як видно, при виведенні отримано однаковий результат. Тепер виведемо значення параметрів у циклах, щоб побачити різницю:

```
# test.sh
#!/bin/bash
count=1
for param in "$@"
do
echo "Параметр \$@: $count = $param"
count=$(( $count + 1 ))
done

count=1
for param in "$*"
```

```
do
echo ”Параметр \$*: $count = $param”
count=$(( $count + 1 ))
done

# "$@" розбиває параметри без дужок на окремі слова
# "$* об'єднує параметри без дужок в одне слово
```

```
./ test.sh 1 2 aa bb
”Параметр $@: 1 = 1”
”Параметр $@: 2 = 2”
”Параметр $@: 3 = aa”
”Параметр $@: 4 = bb”
”Параметр $*: 1 = 1 2 aa bb”
```

```
# test1.sh
#!/bin/bash
fruits=(apple pear plumm peach melon)
vegetables=(carrot tomato cucumber potatoe onion)

printf "Fruits:\t%s\n" "${fruits[*]}"
printf "Fruits:\t%s\n" "${fruits[@]}"
echo " ------------------------------------"
printf "Vegetables:\t%s\n" "${vegetables[*]}"
printf "Vegetables:\t%s\n" "${vegetables[@]}"
```

```
>bash test1.sh

Fruits: apple pear plumm peach melon
Fruits: apple
Fruits: pear
Fruits: plumm
Fruits: peach
Fruits: melon
 ------------------------------------
Vegetables:     carrot tomato cucumber potatoe onion
Vegetables:     carrot
Vegetables:     tomato
Vegetables:     cucumber
Vegetables:     potatoe
Vegetables:     onion
```

Як видно з результату, спеціальна змінна ”$@” містить параметри як окремі слова, а ”$*” – як одне слово.

Спеціальна змінна $_ має різні варіанти використання:

- містить шлях розміщення Bash або сценарію, якщо змінна $_ використовується перед іншими командами:

```
!/bin/bash
echo $_
ls *.sh | sort
```

```
$ ./test.sh
/usr/bin/bash
test1.sh
test2.sh
test3.sh
```

- виводить останній аргумент попередньої виконаної команди в сценарії Bash

```
#!/bin/bash
a=10
b=12
echo "$a" "$b"
echo $_
```

```
$ ./test.sh
10 12
12
```

- виводить останній аргумент попередньої виконаної команди в терміналі

```
# test.sh
#!/bin/bash
$ ls *.sh; echo $_
1.sh
2.sh
3.sh
3.sh
```

Команда `shift` вилучає параметр `$1` і зсовує на його місце наступний параметр `$2`.

```
# test.sh
#/bin/bash
count=1
while [ -n ”$1” ]
do
echo ”Параметр $count = $1”
count=$(( $count + 1 ))
shift
done
```

```
$ ./test.sh 10 20 30
Параметр 1 = 10
Параметр 2 = 20
Параметр 3 = 30
```

1.2. Параметри ключі

Параметри ключі, або просто ключі, виглядають як букви, перед якими стоїть знак тире. Вони служать для керування сценаріями.

```
# test.sh
#/bin/bash
echo
while [-n ”$1” ]
do
case ”$1” in
-a) echo ”Знайдено ключ -a”
-b) echo ”Знайдено ключ -b”
-c) echo ”Знайдено ключ -c”
 *) echo ”$1 не ключ”
esac
shift
done
```

```
$ ./test.sh –a –b –c –d
Знайдено ключ –a
Знайдено ключ –b
```

```
Знайдено ключ -c
-d не ключ
```

Для роботи з *ключами і ключами з параметрами* отриманими з командного рядка у Bash є дві вбудовані команди – `getopts, getopt`.

Команда **`getopts`** видобуває і перевіряє ключі і ключі з параметрами без використання спеціальних позиційних параметрів. Однак `getopts` не отримує всі параметри відразу, а отримує лише наступний ключ з командного рядку під час запуску. Тому вона використовується у команді while, для перевірки того, що всі ключі оброблені.

Команда використовує спеціальні змінні середовища:

`$OPTSTRING` – зберігає список букв ключів (наприклад `для myscript -h -c  OPTSTRING` міститиме букви `hc`);

`$WITCH` – при перевірці аргументів містить поточне ім'я ключа;

`$OPTIND` – вказівник на наступний аргумент, який буде оброблятися командою `getopts`. Це дозволяє продовжити обробку решти параметрів командного рядка після завершення роботи команди `getopts`.

`$OPTARG` – аргумент, який використовується з ключем.

Синтаксис команди при виклику у сценарії:

```
getopts "optstring" variable
```

`optstring` – список букв ключів (після букви ставиться двокрапка, якщо ключ з аргументом). Для подавлення повідомлень про помилки перед списком букв ставиться двокрапка.

`variable` – змінна, яка містить значення поточного параметра командного рядка.

Звичайно команда `getopts` вставляється у цикл `while` і в кожному проході циклу видобувається черговий ключ (опція) і його аргумент (якщо є), обробляється, потім зменшується на 1 спеціальна змінна `$OPTIND` і здійснюється перехід на нову ітерацію.

Ключам, які передаються у сценарій, має передувати символ «-» або «+». Ці префікси дозволяють відрізнити ключі від інших аргументів.

```
# test.sh
#!/bin/bash
declare SWITCH
getopts hc: SWITCH
printf "Перший ключ SWITCH=%s OPTARG=%s OPTIND=%s\n" \
"$SWITCH" "$OPTARG" "$OPTIND"
$ ./test.sh -h -c 1111
Перший ключ SWITCH=h OPTARG= OPTIND=2
$ ./test.sh -c 1111 -h
Перший ключ SWITCH=c OPTARG=111 OPTIND=2
```

Приклад таймера з ключами хвилин і секунд:

```
#!/bin/bash
if [ $# -lt 2 ]; then
echo "./timer.sh <-m | -s>  <number>"
exit
fi
total_seconds=""
while getopts "m:s:" opt; do
   case "$opt" in
```

```
      m) total_seconds=$(( $total_seconds + $OPTARG * 60 )) ;;
      s) total_seconds=$(( $total_seconds + $OPTARG )) ;;
   esac
done

while [ $total_seconds -gt 0 ]; do
    echo "$total_seconds"
    total_seconds=$(( $total_seconds - 1 ))
    sleep 1s
done
echo "Time's up!"
```

Приклад перетворення температур в градусах Цельсія і Фаренгейта з ключами -c і -f:

```
#!/bin/bash
# перетворення температури Цельсія і Фаренгейта
# usage: ./celsuim.sh <-c | -f> <number>
while getopts "c:f:" opt; do
    case $opt in
        c)
            result=$(echo "scale=2; ($OPTARG * (9 / 5)) + 32" | bc)
            ;;
        f)
            result=$(echo "scale=2; ($OPTARG - 32) * (5/9)" | bc)
            ;;
        *)
            echo "$opt"
            ;;
   esac

   echo "$result"
done
```

Команда Bash `getopts` не підтримує стандарт `Linux` по обробці аргументів і їх ключів (так не підтримується подвійне тире, наприклад `--help`).

В Linux є своя команда обробки ключів і аргументів **`/usr/bin/getopt`**. Так як `getopt` є зовнішньою командою, вона не може зберігати ключі у змінних середовища так як `getopts`. Вона також не може експортувати змінні середовища назад у сценарій. Тому `getopt` обробляє всі параметри за один раз як одну групу. Команда обробляє ключі і їх параметри у любій послідовності і повертає їх у впорядкованому виді – ключі і ключі з параметрами, подвійне тире, параметри без ключів. Формат команди:

```
getopt optstring parameters
getopt [options] [--]optstring parameters
getopt [options] -o|--options [--] optstring [options] [--] parameters
```

де `options` – список букв ключів (після букв ключів, які мають параметри ставиться двокрапка);

```
options – список опцій
parameters – список параметрів
```

Ключі (опції) останньої версії команди `getopt` можна отримати командою

```
    $ getopt –help
 -a, --alternative            Allow long options starting with single -
```

```
-h, --help                    This small usage guide
-l, --longoptions <longopts> Long options to be recognized
-n, --name <progname>         The name under which errors are reported
-o, --options <optstring>     Short options to be recognized
-q, --quiet                   Disable error reporting by getopt(3)
-Q, --quiet-output            No normal output
-s, --shell <shell>           Set shell quoting conventions
-T, --test                    Test for getopt(1) version
-u, --unquote                 Do not quote the output
-V, --version                 Output version information
```

Приклад:

```
$ getopt ab:cd -a -b param1 -cd arg1 arg2
-a -b param1 -c -d -- arg1 arg2
```

Для заміни опцій і параметрів командного рядка відформатованою версію команди `getopt` використовується команда `set`.

```
set -- `getopt -q ab:c "$@"`
```

Ключі (опції) можуть передаватися команді `getopt` з використанням ключа з подвійним тире `--`, `(--options (-o) )`. Ключ `--name (-n)` задає ім'я сценарію для оброблення любих помилок. Для задання ключів цілими словами використовується ключове слово `longoptions`, наприклад

```
--longoptions ”help, other”
RESULT=`getopt --name ”$SCRIPT” --options “-h, -c:” --longoptions ”help” -- ”$@”`
```

Признак кінця всіх ключів `-- ”$@”`.

Приклад передачі параметрів в `getopt ( getopt.sh )`

```
declare -rx SCRIPT=${0##*/}   # SCRIPT - ім'я цього сценарію
declare RESULT
RESULT=`getopt --name ”$SCRIPT” --options ”-h, -c:” -- ”$@”`
printf ”status code=$? result=\”$RESULT\”\n”
```

```
$ bash getopt.sh -h
status code=0 result=” -h --”
```

```
$ bash getopt.sh -c
getopt.sh: option requires an argument -- c
status code=1 result=” --”
```

```
$ bash getopt.sh -x
getopt.sh: invalid option -- x
status code=1 result=” --”
```

Довгі опції використовуються з ключем `–longoption` (або `–l`)

```
RESULT=`getopt --name ”$SCRIPT” --options ”-h, -c:” \
--longoptions ”help” -- ”$@”`
```

Крім використання параметрів дані в сценарій можна також ввести командою `read`. Вбудована команда `read` зупиняє сценарій і чекає на введення з клавіатури

```
$ printf ”Введіть кількість днів? ”
$ read days
```

Ключ `–p` дозволяє об'єднати команди `printf` і `read`:

```
$ read –p ”Введіть кількість днів? ” days
```

Ключ `–r` відміняє екранування символом \ спеціальних символів, наприклад \n.

```
$ read -p ”Введіть шлях: ” -r MS_PATH
```

Введення ліміту часу на виконання команди

```
$ read -t 5 FILENAME  # очікування 5 сек на виконання команди filename
```

Введення ліміту на число зчитуваних символів

```
$ read -n 10 FILENAME # прочитати не більше 10 символів
```

Контрольні запитання.

1. Які типи параметрів і ключів використовуються в Bash?
2. Як отримати параметри, передані через командний рядок?
3. Призначення спеціальних змінних Bash – `$#, $@, $*, $_`?
4. Яка різниця між спеціальними змінними Bash `$@, $*` при отриманні параметрів?
5. Як отримати параметри, передані через командний рядок, без використання змінної, яка містить їх число?
6. Чим відрізняються ключі від параметрів і як їх розпізнати у сценарії?
7. Призначення команди `getopts` і змінні середовища оболонки Bash `$OPTSTRING, $SWITCH, $OPTIND, $OPTARG`.
8. Особливості виклику і роботи команди `getopts` у Bash сценарії.
9. Можливості і формат команди Linux `getopt?`
10. Введення даних у сценарій за допомогою команди `read`.

2. Практична частина

Завдання.

1. Написати сценарій, який використовує команду `getopts` для контролю ключів і параметрів командного рядка `–a par_a –b –c`.
2. Написати сценарій, який використовує команду `getopts` для контролю ключів і параметрів командного рядка `–a par_a –b –c par_c –d`.
3. Написати сценарій, який використовує команду `getopts` для контролю ключів і параметрів командного рядка `–a par_a –b –c par_c –d par_d`.
4. Написати сценарій, який використовує команду `getopts` для контролю ключів і параметрів командного рядка `–a par_a –b –c par_c –d par_d par4`.
5. Написати сценарій, який використовує команду `getopts` для контролю ключів і параметрів командного рядка `–x par_x –y –c par_c –y par_y par4 par5`.
6. Написати сценарій, який використовує команду `getopts` для контролю ключів і параметрів командного рядка `–a –b par_b –d par3 par4 –h help`.
7. Написати сценарій, який використовує команду `getopts` для контролю ключів і параметрів командного рядка `–a –b –c –d pard par1 par2 –e par_e`.
8. Написати сценарій, який використовує команду `getopts` для контролю ключів і параметрів командного рядка `–a par_a –b –d par_d par1`.
9. Написати сценарій, який використовує команду `getopt` для контролю ключів і параметрів командного `–a –b par1 –c –d par2 -- par3 par4`.

10. Написати сценарій, який використовує команду `getopt` для контролю ключів і параметрів командного `-a -b par_b -c -d par_d -- par1 par2`.

11. Написати сценарій, який використовує команду `getopt` для контролю ключів і параметрів командного `-a -b -c -e par_e -- par1`.

12. Написати сценарій, який використовує команду `getopt` для контролю ключів і параметрів командного `-- par1 par1`.

13. Написати сценарій, який використовує команду `getopt` для контролю ключів і параметрів командного `-a par_1 -c -d -e -- par1`.

14. Написати сценарій, який використовує команду `getopt` для контролю ключів і параметрів командного рядка `-a -b -c par_c -- par1 par2 par3`.

15. Написати сценарій, який використовує команду `getopt` для контролю ключів і параметрів командного рядка `-a -b -c par_c -- par1 par2`.

Примітка. Номер варіанту завдання вибирається за порядковим номером студента у журналі групи.

Звіт з лабораторної роботи має містити:

- назву групи, П.І.Б. студента, завдання до роботи;
- короткий опис теоретичної частини;
- текст програми із коментаріями;
- роздрук екранів із результатом запуску програми.

Приклади

```
# 1. Параметри командного рядка: $1,...,$9
# 1.sh
#/bin/bash
echo "Перший параметр" $1
echo "Другий параметр" $2
total=$[ $1 * $2 ]
echo "Добуток" $total
```

```
# 2. Використання параметрів командного рядка
# 2.sh
#!/bin/bash
factorial=1
for (( number=1; number<=$1; number++ ))
do
   factorial=$[ $factorial * $number ]
done
echo Факторіал від $1 = $factorial
```

```
# 3. Символьний рядок в командному рядку
# 3.sh
#!/bin/bash
echo Привіт $1, радий тебе бачити.
```

```
# 4. Обробка параметрів при їх кількості більше 9
# 4.sh
#!/bin/bash
# виконання
# ./test4 1 2 3 4 5 6 7 8 9 10 11 12
```

```
total=$[ ${10} * ${11} ]
echo Десятий параметр ${10}
echo Одинадцятий параметр ${11}
echo The total is $total
```

 # 5. Читання параметру $0

```
# 5.sh
#!/bin/bash
echo "Введена команда:" $0
```

 # 6. Використання імені сценарію

```
# 6.sh
#!/bin/bash

# виконання:
# chmod u+x test6
# cp test6 test6_a
# ln -s test6 test6_m
# ls -l
# ./test6_a 2 5
# ./test6_m 2 5

name=`basename $0`
if [ $name = "test6_a" ]
   then
     total=$[ $1 + $2 ]
   elif [ $name = "test6_m" ]
   then
     total=$[ $1 * $2 ]
fi
echo Обчислене значення $total
```

 # 7. Перевірка наявності параметрів в командному рядку

```
# 7.sh
#!/bin/bash
if [ -n "$1" ]
then
  echo Параметр 1 наявний
else
  echo "Параметри відсутні"
fi
```

 # 8. Визначення кількості параметрів в командному рядку

```
# 8.sh
#!/bin/bash
echo Є $# параметрів
```

 # 9. Порівняння кількості параметрів

```
# 9.sh
#!/bin/bash
if [ $# -ne 2 ]
then
  echo Виконання програми: test9 a b
else
  total=$[ $1 + $2 ]
```

```
  echo Результат: $total
fi
```

 # 10. захоплення останнього параметра

```
# 10.sh
#!/bin/bash
params=$#
echo Число параметрів $params
echo Останній параметр ${!#}
```

 # 11. Тестування параметрів командного рядка $* and $@

```
# 11.sh
#!/bin/bash
echo "Використання \$* методу: $*"
echo "Використання \$@ методу: $@"
```

 # 12. Перевірка параметрів командного рядка $* і $@

```
# 12.sh
#!/bin/bash
count=1
for param in "$*"
do
  echo "\$* Параметри #$count = $param"
  count=$[ $count + 1 ]
done
  count=1
for param in "$@"
do
  echo "\$@ Параметри #$count = $param"
  count=$[ $count + 1 ]
done
```

 # 13. Команда зсуву (shift) параметрів командного рядка

```
# 13.sh
#!/bin/bash
count=1
while [ -n "$1" ]
do
  echo "Параметр #$count = $1"
  count=$[ $count + 1 ]
  shift
done
```

 # 14. Зсув параметрів командного рядка на декілька позицій

```
# 14.sh
#!/bin/bash
echo "Початкові параметри: $*"
shift 2
echo "Зсунуті параметри: $1"
```

 # 15. Генерування 8-символьної стрічки з випадкових символів

```
# 15.sh
#!/bin/bash
if [ -n "$1" ]  #  якщо є аргумент командного рядка
then            #  то він є стартовою стрічкою
```

```
  str0="$1"
else            #  інакше стартовою стрічкою є PID
  str0="$$"
fi
POS=2  # почати з 2 позиції у стрічці
LEN=8  # видобути 8 символів
str1=$( echo "$str0" | md5sum | md5sum ) #  подвійне обчислення mm5sum
randstring="${str1:$POS:$LEN}"
# параметризації   ^^^^ ^^^^
echo "$randstring"
exit $?
```

```
$ ./15.sh my-password
1bdd88c4
```

 # 16. Видобування опцій з параметрів командного рядка

```
# 16.sh
#!/bin/bash
while [ -n "$1" ]
do
 case "$1" in
  -a) echo "Знайдено опцію -a" ;;
  -b) echo "Знайдено опцію -b" ;;
  -c) echo "Знайдено опцію -c" ;;
   *) echo "$1 не параметр";;
 esac
 shift
done
```

 # 17. Видобування опцій і параметрів з командного рядка

```
# 17.sh
#!/bin/bash
# Виконання:
# ./test16 -c -a -b test1 test2 test3
# ./test16 -c -a -b -- test1 test2 test3

while [ -n "$1" ]
do
 case "$1" in
   -a) echo "Знайдено опцію -a" ;;
   -b) echo "Знайдено опцію -b" ;;
   -c) echo "Знайдено опцію -c" ;;
   --) shift
   break ;;
   *) echo "$1 не опція";;
 esac
 shift
done

count=1
for param in $@
do
  echo "Параметр #$count: $param"
  count=$[ $count + 1 ]
done
```

 # 18. Видобування опцій і значень аргументів командного рядка

```
# 18.sh
#!/bin/bash
# виконання:
# ./test17 -a  -b 2 -c  -d  --- 1 3 4
while [ -n "$1" ]
  do
    case "$1" in
      -a) echo "Знайдено опцію -a";;
      -b) param="$2"
          echo "Знайдено опцію -b з параметром $param"
          shift;;
      -c) echo "Знайдено опцію -c";;
      --) shift
           break;;
       *) echo "$1 не опція";;
    esac
    shift
  done
  count=1
  for param in "$@"
  do
    echo "Parameter #$count: $param"
    count=$[ $count + 1 ]
  done
```

 # 19. Використання команди getopts

```
while getopts ae:f:hd:s:qx: option
do
   case "${option}"
      in
         a) ALARM="TRUE";;
         e) ADMIN=${OPTARG};;
         d) DOMAIN=${OPTARG};;
         f) SERVERFILE=$OPTARG;;
         s) WHOIS_SERVER=$OPTARG;;
         q) QUIET="TRUE";;
         x) WARNDAYS=$OPTARG;;
         \?) usage
         exit 1;;
   esac
done
```

 # 19. Використання команди getopts

```
# 19.sh
#!/bin/bash
while getopts :ab:c opt
do
  case "$opt" in
   a) echo "Знайдено опцію -a" ;;
   b) echo "Знайдено опцію -b з параметром $OPTARG";;
   c) echo "Знайдено опцію -c" ;;
   *) echo "Невідома опція: $opt";;
  esac
```

```
done
```

$ ./19.sh –a –b 1111 –c

```
Знайдено опцію –a
Знайдено опцію –b з параметром 1111
Знайдено опцію -c
```

 # 20. Обробка опцій командою getopts

```
# 20.sh
#!/bin/bash
while getopts :ab:cd opt
do
   case "$opt" in
    a) echo "Знайдено опцію -a" ;;
    b) echo "Знайдено опцію -b option з параметром $OPTARG";;
    c) echo "Знайдено опцію -c option";;
    d) echo "Знайдено опцію -d option";;
    *) echo "Невідома опція: $opt";;
   esac
done
shift $[ $OPTIND - 1 ]
count=1
for param in "$@"
  do
    echo "Параметр $count: $param"
    count=$[ $count + 1 ]
  done
```

$ ./20.sh –a –b 1111 –c –d

```
Знайдено опцію –a
Знайдено опцію –b з параметром 1111
Знайдено опцію -c
Знайдено опцію -d
```

 # 21. Обробка опцій командою getopts

```
# 21.sh
#!/bin/bash
NO_ARGS=0
usage () {
  echo "Сценарій `basename $0` для демонстрації можливостей getopts."
  echo ""
  echo "Використання: `basename $0` -abef -c C -d D"
  echo -e "    \033[1mОпції:\033[0m"
  echo "    -a | -b Дві опції для однієї дії"
  echo "    -c    Ключ (опція) з аргументом"
  echo "    -d    Ще ключ з аргументом"
  echo "    -e    Ключ без аргументу"
  echo "    -f    Ще ключ без аргументу"
}

if [ $# -eq "$NO_ARGS" ]  # Сценарій викликаний без аргументів?
then
  usage                   # Якщо запущений без аргументів - вивести довідку
  exit $E_OPTERROR        # і вийти з кодом помилки
fi

while getopts "abc:d:ef" Option
```

```
do
  case $Option in
      a | b ) echo "Дія 1: ключ - $Option. Номер ключа: $OPTIND. Аргумент:
$OPTARG";;
      c       ) echo "Дія 2: ключ - $Option. Номер ключа: $OPTIND. Аргумент:
$OPTARG";;
      d       ) echo "Дія 3: ключ - $Option. Номер ключа: $OPTIND. Аргумент:
$OPTARG";;
      e       ) echo "Дія 4: ключ - $Option. Номер ключа: $OPTIND. Аргумент:
$OPTARG";;
      f       ) echo "Дія 5: ключ - $Option. Номер ключа: $OPTIND. Аргумент:
$OPTARG";;
    *     ) echo "Вибрано недопустимий ключ."
            usage
            exit $E_OPTERROR;;    # За замовчуванням
  esac
done
shift $(($OPTIND - 1))
exit 0
```

 # 22. Видобування опцій і параметрів командного рядка командою getopt

```
# 22.sh
#!/bin/bash
set -- `getopt -q ab:c "$@"`
while [ -n "$1" ]
do
   case "$1" in
    -a) echo "Знайдено опцію -a" ;;
    -b) param="$2"
       echo "Знайдено опцію -b з параметром $param"
       shift ;;
    -c) echo "Знайдено опцію -c option" ;;
    --) shift
        break;;
    *) echo "$1 не опція";;
   esac
   shift
done
   count=1
   for param in "$@"
   do
     echo "Parameter #$count: $param"
     count=$[ $count + 1 ]
   done
```

$ ./22.sh –a –b 111 –c

```
Знайдено опцію –a
Знайдено опцію –b з параметром 111
Знайдено опцію -c
```

 # 23. Тестування команди read

```
# 23.sh
#!/bin/bash
echo -n "Введіть ім'я: "
read name
echo "Привіт  $name, запрошуємо в нашу програму."
```

 # 24. Тестування команди read з опцією -p

```
# 24.sh
#!/bin/bash
read -p "Скільки годин:" god
sec=$[ $god * 3600 ]
echo Введені години мають $sec секунд
```

 # 25. Введення декількох змінних

```
# 25.sh
#!/bin/bash
read -p "Введіть ім'я, по батькові: " first last
echo "Вітаю Вас $first $last,..."
```

 # 26. Тестування REPLAY змінної середовища

```
# 26.sh
#!/bin/bash
read -p "Enter a number: "
factorial=1
for (( count=1; count <= $REPLY; count++ ))
do
factorial=$[ $factorial * $count ]
done
echo "Факторіал $REPLY дорівнює $factorial"
```

 # 27. Час на введення даних

```
# 27.sh
#!/bin/bash
if read -t 5 -p "Введіть ім'я: " name
then
  echo "Привіт $name, запрошую вивчати bash"
else
  echo
echo "Вибачайте, Ви запізнилися!"
fi
```

 # 28. Обмеження вводу одним символом

```
# 28.sh
#!/bin/bash
read -n1 -p "Ви бажаєте продовжити [Y/N]? " answer
case $answer in
  Y | y) echo
         echo "добре, продовжимо...";;
  N | n) echo
         echo OK, Бувайте здорові
         exit;;
esac
echo "Кінець сценарію"
```

 # 29. Читання даних з файлу test28.txt

```
# 29.sh
#!/bin/bash
count=1
cat test28.txt | while read line
do
```

```
  echo "Рядок $count: $line"
  count=$[ $count + 1]
done
echo "Кінець файлу"
```

Лабораторна робота 6

Перенаправлення потоків введення-виведення

Мета роботи: навчитися перенаправляти дескриптори файлів.

Теоретичний матеріал лекції, технічна література, інтернет ресурси.

1. Короткі теоретичні відомості

1.1. Перенаправлення потоків виведення

В Linux всі пристрої є файлами і зберігаються в каталозі `/dev`.

`/dev/stdin` – пристрій стандартного введення (дескриптор STDIN, 0)
`/dev/stdout` – пристрій стандартного виведення (дескриптор STDOUT, 1)
`/dev/stderr` – пристрій стандартного виведення помилок (дескриптор STDERR, 2)
`/dev/null` – пристрій утилізації любих даних які направлені на його вхід
`/dev/zero` – пристрій для створення файлів, які повністю заповнені нулями
`/dev/tty` – термінал або консоль в якій виконується програма
`/dev/dsp` – інтерфейс до пристроїв які відтворюють звук або звукової карти
`/dev/fd0` – перший гнучкий диск
`/dev/hda1` – перший розділ IDE диска
`/dev/sda1` – перший розділ SCSI диска
…

Коли Bash стартує відкриваються три стандартні файлові дескриптори : `stdin` (файловий дескриптор 0), `stdout` (файловий дескриптор 1), `stderr` (файловий дескриптор 2). Можна відкрити і більше файлових дескрипторів (3, 4, 5, ...), потім їх закрити. Можна копіювати файлові дескриптори. Можна в них записувати і потім читати з них. Файлові дескриптори завжди вказують на деякі файли, якщо вони не закриті. Звичайно, коли Bash стартує всі три дескриптори `stdin`, `stdout`, `stderr` вказують на термінал `dev/tty0`.

Перенаправлення `stdin` і `stderr` у файл:

```
command > file
(command1; command2) > file
```

У загальному випадку можна перенаправити файловий дескриптор `n` у файл:

```
commanf n> file
```

Перенаправлення `stderr` у файл:

```
command 2> file
```

Перенаправлення `stdout` одного процесу у `stderr` іншого процесу:

```
command > >(stdout_cmd) 2> >(stderr_cmd)
command > /dev/fd/50 2> /dev/fd/61
```

В Linux оператор `&0` використовується для посилання на потік стандартного введення, `&1` – на потік стандартного виведення, а `&2` – на потік виведення помилок.

Перенаправлення `stdout` і `stderr` у файл:

```
command &> file
```

```
command >& file
command > file 2>&1
```

Однак порядок перенаправлень важливий. У прикладі нижче, команда перенаправляє тільки `stdout` у файл:

```
 command 2>&1 > file
```

Приклади перенаправлень результатів команд:

```
printf ”Sales are up” > results.txt # направити у файл на диску
printf ”Sales are up” > /dev/tty # направити у консоль (безпосередньо)
printf ”Sales are up” # направити у консоль через стандартний вивід
printf ”Sales are up >&1 #  -”- стандартний stdin
printf ”Sales are up > /dev/stdout # -”- стандартний stdout

$ ls –l home/user  > listing.txt # записано у listing.txt
$ ls -l home/user  1>&1 # записано у listing.txt
$ ls -l home/user  > /dev/tty # виведено на екран
$ printf ”$SCRIPT:$LINENO: Відсутні файли для оброблення” >&2
```

Перенаправляти результати виконання команд можна у файл з його очищенням або з добавленням:

```
command > file  # результат виконання команди записати у файл з очищенням
command >> file # результат виконання команди добавити у файл
```

Результати виконання команд можна очистити за допомогою спеціального пристрою */dev/null*

```
command > /dev/null
command > /dev/null 2>&1
command &> /dev/null
```

Перенаправлення стандартних дескрипторів (символ “>” на вихід):

`1>  file` – перенаправлення стандартного вихідного потоку `stdout` для запису у файл

`1>> file` – перенаправлення стандартного вихідного потоку `stdout` для добавлення у файл

`2>  file` – перенаправлення потоку помилок `stderr` для запису у файл

`2>> file` – перенаправлення потоку помилок `stderr` для добавлення у файл

`&>, >& file` - перенаправлення потоків `stdout` і `stderr` для запису у файл

`&>> file` - перенаправлення потоків `stdout` і `stderr` для добавлення у файл

Файлові дескриптори можуть бути перенаправлені:

- тимчасово для одноразового виконання команди;
- постійно для всіх команд сценарію.

Для тимчасового перенаправлення дескриптора файлу перед ним ставиться знак `&`:

```
echo ”Привіт”  >&1
echo ”Помилка” >&2
```

Для постійного перенаправлення дескриптора файлу використовується команда `exec`, яка запускає нову оболонку сценаріїв (`shell`) і перенаправляє файловий дескриптор на час дії сценарію з командою `exec`.

```
exec 1> out  # перенаправлення stdout, stderr для запису у файл out
```

```
exec 2> err  # перенаправлення stderr для запису у файл err
```

1.2. Перенаправлення потоків введення

Перенаправлення даних з файлу на вхід `stdin` команди:

```
command < file
```

```
$ wc < myfile
2 3 6  # число рядків число слів число байтів
```

```
read -r line < file # порядкове читання stdin з файлу
```

Можна перенаправити вхідний потік `stdin` з клавіатури на файл:

```
exec 0< file
```

Перенаправлення фрагменту тексту у `stdin` команди

```
command << унікальні_символи_початку_даних
>дані
…
>дані
>Унікальні_символи кінця даних (мають співпадати із символами початку)
```

Приклад:

```
$ wc << EOL
>1
>2
>3
>EOL
3 3 6
```

Замість передачі стрічки у команду, наприклад `$ echo "Привіт світ!" | command`,

можна використати оператор перенаправлення <<< . Тоді замість коду

```
$ echo '5*6' | bc
30
```

можна записати

```
$ bc <<< 5*6
30
```

Стрічка, яка перенаправляється оператором <<<, створюється як тимчасовий файл у форматі `/tmp/sh-thd.<random string>`

```
ls -l /proc/self/fd/ <<< "TEST"
total 0
lr-x------ 1 victor victor 0 Oct 21 12:41 0 -> '/tmp/sh-thd.sPL5P7 (deleted)'
lrwx------ 1 victor victor 0 Oct 21 12:41 1 -> /dev/tty1
lrwx------ 1 victor victor 0 Oct 21 12:41 2 -> /dev/tty1
lr-x------ 1 victor victor 0 Oct 21 12:41 3 -> /proc/148/fd
```

За допомогою трасування syscalls можна побачити, що тимчасовий файл спочатку відкривається як fd 3, у нього записуються дані, потім він перевідкривається з прапором O_RDONLY як fd 4 і пізніше від'єднується, потім застосовується dup2() до fd 0, який пізніше успадковується командою `cat`.

```
$ strace -f -e open,read,write,dup2,unlink,execve bash -c 'cat <<< "TEST"'
```

```
    execve("/bin/bash", ["bash", "-c", "cat <<< \"TEST\""], 0x7fffd4407d38 /* 18
vars */) = 0
    read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\3\0>\
0\1\0\0\0\220\311\0\0\0\0\0\0"..., 832) = 832
    read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\16\0\0\0\0\0\0"...,
832) = 832
    read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\20\35\2\0\0\0\0\0"...,
832) = 832
    read(3, "# Locale name alias data base.\n#"..., 4096) = 2995
    read(3, "", 4096)                       = 0
    strace: Process 152 attached
    [pid   152] write(3, "TEST", 4)         = 4
    [pid   152] write(3, "\n", 1)           = 1
    [pid   152] unlink("/tmp/sh-thd.lE5m6U") = 0
    [pid   152] dup2(4, 0)                  = 0
    [pid   152] execve("/bin/cat", ["cat"], 0x7fffd19f51a0 /* 18 vars */) = 0
    [pid               152]     read(3,     "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\
0\1\0\0\0\20\35\2\0\0\0\0\0"..., 832) = 832
    [pid   152] read(3, "# Locale name alias data base.\n#"..., 4096) = 2995
    [pid   152] read(3, "", 4096)           = 0
    [pid   152] read(0, "TEST\n", 131072)   = 5
    [pid   152] write(1, "TEST\n", 5TEST
    )       = 5
    [pid   152] read(0, "", 131072)         = 0
    [pid   152] +++ exited with 0 +++
    --- SIGCHLD {si_signo=SIGCHLD, si_code=CLD_EXITED, si_pid=152, si_uid=1000,
si_status=0, si_utime=0, si_stime=0} ---
    +++ exited with 0 +++
```

1.3. Конвеєризація команд

Для передачі даних з виходу `stdout` однієї команди на вхід `stdin` іншої команди використовуються конвеєри:

```
command1 | command2
```

Так, виконання двох команд можна замінити одною командою

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

Передача `stdout i stderr` однієї команди на вхід `stdin` іншої команди:

```
command1 |& command2
command1 2>&1 |command2
```

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

```
who | tee testfile
```

Реєстраційні файли можна також створити з використанням команди `script`:

```
$ script -c 'bash hello.bash' `date '+%Y-%m-%d-%H:%M:%S'`.log

Script started, file is 2024-11-12-11:39:37.log
Hello word!
```

```
hello.bash
Script done, file is 2024-11-12-11:39:37.log
```

1.4. Підставлення процесів

Перенаправлення `stdout` одної команди на `stdin` іншої команди є потужною технікою. Але бувають випадки коли потрібно перенаправити `stdout` багатьох команд. У цьому випадку використовується підставлення (заміна) процесів. Підставлення процесів направляє вихід процесу (або процесів) на вхід іншого процесу. Список команд поміщається у круглі дужки:

```
>(список_команд)
<( список_команд)
```

Підставлення процесів використовує файл `/dev/fd/<n>` для надсилання результатів процесу (або процесів) у круглих дужках іншому процесу.

Оболонка створює дескриптор тимчасового файлу **`/dev/fd/63`** куди записується вихід

```
$ echo >(echo bar)
/dev/fd/63

$ echo <(echo bar)
/dev/fd/63


$ wc <(echo one; echo two)
2   2       8 /dev/fd/63

$ (echo one; echo two) | wc
2   2       8 /dev/fd/63
```

Підставлення процесів може порівнювати виходи двох різних команд або виходи одної команди з різними ключами:

```
$ comm <(ls -l | sort) <(ls -al | sort)
```

Різниця у виконанні команд:

```
$ diff <(ls -l | sort) <(ls -al | sort)
```

Сортування файлів у трьох різних каталогах:

```
$ sort -k 9 <(ls -l /bin) <(ls -l /usr/bin) <(ls -l /usr/X11R6/bin)
```

Перевірка результату розтискання файлу:

```
$ gzip -d<mydata.gz | md5sum -c mydata-orig.md5)
```

Підставлення процесів може порівнювати вміст двох каталогів

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

1.5 Користувацькі дескриптори файлів

Оболонка сценаріїв може мати до *дев'яти* відкритих файлових дескрипторів. Дескриптори файлів з *третього по дев'ятий* може використати користувач для перенаправлення як `stdin` так і `stdout`.

Виведення у файл із дескриптором користувача:

```
file="test"
exec 4> $file
```

```
echo ”Привіт світ” >&4
exec 4>&-    # закрити файловий дескриптор 4
```

Можна постійно перенаправляти стандартні дескриптори файлів альтернативним і навпаки.

```
file=”test”
exec 3>&1
echo ”Постійно перенаправити дескриптора 3 в 1” > $file
exec 1>&3
echo ”Постійно перенаправити дескриптора 1 в 3" >> $file
cat $file
```

```
Постійно перенаправити дескриптор 3 в 1
Постійно перенаправити дескриптор 1 в 3
```

Аналогічно можна перенаправити `stdin` у файловий у дескриптор користувача 6.

```
file=”test”
exec 6< $file
read -u 6 line
echo $line
exec 6<&-
```

Для призначення файловим дескрипторам можливостей введення/виведення даних використовується команда `exec` з відповідним номером файлового дескриптора і символом “<>”:

```
echo ”Привіт світ” > file
exec 5<> file
read line <&5
echo ”Test” >&5
```

При створенні нових файлових дескрипторів оболонка Bash закриває їх при завершенні сценарію. Для ручного закриття файлових дескрипторів їх необхідно перенаправити у `&-`:

```
exec 5>&-
```

Приклад читання і запису в задану позицію файлу:

```
file=”test”
exec 5<> $file          # зв’язування файлу з дескриптором 5
echo ”12345”            # запис у файл стрічки
read –n 2 var <&5       # читання 2-х символів з файлу
echo $var
echo –n ”+” >&5         # запис в наступну позицію символу ”+”
exec 5>&-               # закриття дескриптора 5
cat  $file               # результат ”12+45”
```

Отримати інформацію про відкриті файлові дескриптори всієї системи Linux дозволяє команда `lsof`. Для зменшення обсягу інформації для виведення використовують ключі:

`-d 0,1,2,3` # вказують номери потрібних файлових дескрипторів

`-p PID1 -p PID2` # вказують PID потрібних процесів

`-p $$` # PID поточного процесу

1.6. Виконання команд через файл

Для виконання команд через файд потрібно дві оболонки. В оболонці 1 виконуються команди:

```
mkfifo fifo
exec < fifo
```

В оболонці 2 виконуються команди:

```
exec > fifo
echo 'echo Привіт світ'
```

1.7. Доступ до вебсайтів через Bash:

Bash розглядає `/dev/tcp/host/port` як спеціальний файл для відкриття `tcp` з'єднань.

```
exec 3<> /dev/tcp/pnu.edu.ua/80
echo -e "GET / HTTP/1.1\n\n" >&3
cat <&3
```

Контрольні запитання.

1. Які основні пристрої використовуються у Linux?
2. Як тимчасово перенаправити файлові дескриптори?
3. Яка команда постійно перенаправляє файлові дескриптори?
4. Як постійно перенаправити `stdin` з клавіатури на файл?
5. Як перенаправити стандартні файлові дескриптори альтернативним і навпаки?
6. Як надати файловим дескрипторам можливість введення/виведення даних?
7. Як закриваються файлові дескриптори?
8. Як подавити виведення команд у Linux?
9. Для чого призначена команда tee?
10. Якою командою можна отримати інформацію про відкриті файлові дескриптори системи Linux?
11. Обмін даними через канал FIFO.
12. Для чого призначений порт `/dev/tcp/host/port`.
13. Підставлення процесу (процесів).

2. Практична частина

Завдання.

1. Переслати повідомлення "Привіт світ" введене з консолі в іншу консоль через канал FIFO з використанням дескриптора 5.
2. Знайти у заданому каталозі файли створені у поточному місяці, відсортувати їх за розміром. Мінімальний і максимальний розмір файлів переслати в іншу консоль через канал FIFO з використанням дескриптора 6.
3. Вивести вміст текстового файлу у іншу консоль через канал FIFO з використанням дескриптора 7.
4. Виконати команду `ls -l` у іншій консолі через канал FIFO з використанням дескриптора 3.
5. За допомогою команд `ls, test, wc` визначити кількість каталогів у поточному каталозі результат перенаправити у іншу консоль через канал FIFO з використанням дескриптора 8.

6. Переписати файл з одної консолі у іншу консоль з використанням каналу FIFO з використанням дескриптора 3.

7. Перенаправити повідомлення про помилки при виконанні команди `ls xxx` через канал FIFO у іншу консоль з використанням дескриптора 5.

8. Порівняти два файли і однакові рядки записати в `stdout`, а неоднакові – у `stderr`.

9. Використовуючи перенаправлення дескрипторів користувача вивести у консоль вміст тестового файлу.

10. Створити дескриптор 5 з можливістю читання і запису файлу. Записати у файл декілька рядків даних, прочитати їх і перезаписати непарні рядки з символом “*” на початку.

11. Створити дескриптор 6 з можливістю читання і запису файлу. Записати у файл декілька рядків даних, прочитати їх і перезаписати парні рядки символом “+” в кінці.

12. Створити дескриптор 7 з можливістю читання і запису файлу. Записати у файл стрічку “Привіт світ”, прочитати файл і замінити букви “і” на букву “а”, записати змінену стрічку у файл.

13. Створити тимчасовий файл у каталозі `/tmp`, записати туди дані про поточну дату і час у форматі dd.mm/yy-hh:mm:ss. Закрити файлові дескриптори, вивести вміст файлу командою `cat` у консоль.

14. Створити тимчасовий файл у тимчасовому каталозі `/tmp`, записати туди дані про процеси, які виконуються в системі.

15. Створити за допомогою команди `script` і `tee` реєстраційний файл `log.txt` куди записувати виведення усіх сценаріїв консольної сесії.

Примітка. Номер варіанту завдання вибирається за порядковим номером студента у журналі групи.

Звіт з лабораторної роботи має містити:

- назву групи, П.І.Б. студента, завдання до роботи;
- короткий опис теоретичної частини;
- текст програми із коментарями;
- роздрук екранів із результатом запуску програми.

Приклади

1. Перенаправлення стандартних дескрипторів

```
# 01.sh
# дескриптор файлу: 0 - STDIN, 1 – STDOUT, 2 - STDERR
# направлення виходу в STDOUT
ls --l > 1a.txt
# добавлення виходу до файлу
ls -l>> 1b.txt
who >> 1a.txt
# направлення виходу помилок в STDERR
ls -a -l xyz 2> err
# направлення STDERR і STDOUT в один файл
ls -a -l 1a.txt xyz &> 1b.txt
# направлення STDERR в 1a, SRDOUT в 1b
ls –a -l xyz 2> 1a.txt 1a.txt 1> 1b.txt
```

2. Тимчасове перенаправлення повідомлення у файловий дескриптор STDERR

```
# 02.sh
#!/bin/bash
```

```
echo "Це повідомлення про помилку для STDERR" >&2
echo "Це звичайне повідомлення для STDOUT"
```

3. Постійне перенаправлення всіх виходів у файл

```
# 03.sh
#!/bin/bash
exec 1> 3.txt
echo "Тест на перенаправлення всіх виходів"
echo "з сценарію в інший файл"
echo "без необхідності перенаправляти кожний файл"
```

4. Постійне перенаправлення виведення у різні дескриптори

```
# 04.sh
#!/bin/bash
# перенаправлення виводу у файл
exec 2>4.err
echo "Старт сценарію"
echo "перенаправлення всіх виходів"
exec 1>4.txt
echo "Це   повідомлення буде виведене у файл 4.txt"
echo "а це повідомлення буде виведене у файл 4.err" >&2
```

5. Перенаправлення введення з файлу за допомогою команди exec

```
# створити файл 5.txt і ввести в нього 3-4 рядки даних
# 05.sh
#!/bin/bash
echo ”aaa” >  5.txt
echo ”bbb” >> 5.txt
echo ”ccc” >> 5.txt
exec 0< 5.txt
count=1
while read line
do
echo "Рядок #$count: $line"
count=$[ $count + 1 ]
done
`rm 5.txt`
```

6. Призначення альтернативних файлових дескрипторів (скрипт може мати до дев’яти відкритих файлових дескрипторів)

```
# 06.sh
#!/bin/bash
exec 3>6.out
echo "Це повідомлення буде направлено на екран"
echo "це буде збережене у файл" >&3
echo "а це буде знову направлено на екран"
```

7. Зміна номера дескриптора для STDOUT, а потім його відновлення

```
# 07.sh
#!/bin/bash
exec 3>&1
echo " Призначення дескриптору 1 значення 3" >&3
exec 1>7.out
echo "Перенаправлення STDOUT в файл"
exec 1>&3
```

```
echo " Призначення дескриптора 3 значення 1" >&1
```

8. Перенаправлення дескрипторів вхідного файлу

```
#!/bin/bash
FILE=$1
# read $FILE using the file descriptors
exec 3<&0
exec 0<$FILE
while read line
do
        # use $line variable to process line
        echo $line
done
exec 0<&3
```

9. Перенаправлення дескрипторів вхідного файлу

```
# 09.sh
#!/bin/bash
echo "Перенаправлення дескриптора 0 в 6"
exec 6<&0
echo "aaa" > 9.txt
echo "bbb" >> 9.txt
echo "ccc" >> 9.txt
exec 6<&0
exec 0< 9.txt
count=1
  while read line
  do
    echo "Рядок #$count: $line"
    count=$[ $count + 1 ]
  done
echo "Перенаправлення дескриптора 6 в 0"
exec 0<&6
read -p "Ви закінчили роботу (y/n)? " answer
case $answer in
  Y|y) echo "Бувайте здорові";;
  N|n) echo "Вибачте, це кінець.";;
esac
rm 9.txt
```

10. Створення дескрипторів файлу з можливістю введення/виведення даних

```
# 10.sh
#!/bin/bash
exec 3<> 10.txt
read line <&3
echo "Читання: $line"
echo "Це тестовий рядок" >&3
```

11. Закриття дескриптора файлу

```
# 11.sh
#!/bin/bash
exec 3> 11.txt
echo "Це тестовий рядок даних" >&3
exec 3>&-
cat 11.txt
```

```
exec 3> 11.txt
echo "Це буде невірно" >&3
```

12. Використання команди lsof для отримання інформації про відкриті файлові дескриптори всієї системи Linux

```
# 12.sh
#!/bin/bash
exec 3> 12.1
exec 6> 12.2
exec 7< 11.txt
#/usr/sbin/lsof -a -p $$ -d 0,1,2,3,6,7
lsof -a -p $$ -d 0,1,2,3,6,7
```

13. Подавлення виведення команд

```
# 13.sh
#/bin/bash
echo Подавлено вивід на STDOUT
ls -al xyz  1> /dev/null
echo Подавлено вивід на SDTERR
ls -al xyz  2> /dev/null
```

14. Створення і використання тимчасового локального файлу

```
# 14.sh
#!/bin/bash
tempfile=`mktemp 14.XXXXXX`
exec 3>$tempfile
echo "Скрипт записує дані в тимчасовий файл $tempfile"
echo "Перший рядок" >&3
echo "Другий рядок" >&3
echo "Третій рядок" >&3
# занулення дескриптора
3>&-

echo "Створено тимчасовий файл. Його вміст:"
cat $tempfile
# вилучення файлу
rm -f $tempfile 2> /dev/null
```

15. Створення тимчасового файлу в /tmp

```
# 15.sh
#!/bin/bash
tempfile=`mktemp -t tmp.XXXXXX`
echo "Це тестовий файл" > $tempfile
echo "Другий рядок тесту" >> $tempfile
echo "Тимчасовий файл розміщений у каталозі: $tempfile"
cat $tempfile
# знищення тимчасового файлу
rm -f $tempfile
```

16. Використання тимчасових каталогів

```
# 16.sh
#!/bin/bash
tempdir=`mktemp -d dir.XXXXXX`
cd $tempdir
tempfile1=`mktemp temp.XXXXXX`
```

```
tempfile2=`mktemp temp.XXXXXX`
exec 7> $tempfile1
exec 8> $tempfile2
echo "Надсилання даних у каталог $tempdir"
echo "Тестовий рядок для файлу  $tempfile1" >&7
echo "Тестовий рядок для файлу  $tempfile2" >&8
```

17. Команда tee

```
# 17.sh
#!/bin/bash
# команда tee одночасно перенаправляє вхід у STDOUT і заданий файл
# використання команди tee для ведення журналу реєстрації (logging)
tempfile=test22.txt
echo "Початок тесту" | tee $tempfile
echo "Перший рядок тесту" | tee -a $tempfile
echo "Кінець тесту" | tee -a $tempfile
```

18. Оброблення конвеєризованих даних

```
# 18.sh
#!/bin/bash
echo ”Підсумок продаж”
cat /dev/stdin | cut –d’ ’ –f 2,3 | sort

>cat sales.data
Іван  яблука  5 Травень  12
Степан молоко 2 Червень  21
Василь пиво  4 Липень   11
Ігор риба 7 Вересень 5

>cat sales.data | ./18.sh
молоко 2
пиво 4
риба 7
яблука 5
```

19. Іменований канал FIFO для передачі даних

```
# 19a.sh
#!/bin/bash
pipe=/tmp/testpipe

trap "rm -f $pipe" EXIT

if [[ ! -p $pipe ]]; then
    mkfifo $pipe
fi

while true
do
    if read line <$pipe; then
        if [[ "$line" == 'quit' ]]; then
            break
        fi
        echo $line
    fi
done
```

```
echo "Reader exiting"

#19b.sh
#!/bin/bash

pipe=/tmp/testpipe
if [[ ! -p $pipe ]]; then
    echo "Reader not running"
    exit 1
fi

if [[ "$1" ]]; then
    echo "$1" >$pipe
else
    echo "Hello from $$" >$pipe
fi
```

Сценарії запускаються в різних оболонках:

```
Оболонка 1                Оболонка 2
$ ./19a.sh                $ ./19b.sh
Hello from 13278          $ ./19b.sh
Hello from 13281          $ ./19b.sh
Hello from 13284
```

Лабораторна робота 7

Задачі і сигнали

Мета роботи: вивчення команд керування задачами і оброблення сигналів ОС.

1. Короткі теоретичні відомості

1.1. Керування задачами

Для запуску *задачі* на виконання у фоновому режимі потрібно після команди добавити знак &. Люба команда або сценарій, які виконуються у фоновому режимі, є задачами. Для керування задачами створюється список задач. В командному режимі керування задачами доступне за замовчуванням. Для активізації керування задачами в сценаріях потрібно задати опцію `shopt –s -o monitor`.

Перелік активних задач виводить команда `jobs`.

```
$ sleep 60 &
[1] 28875                    # 28875 – PID задачі
$ jobs
[1]+ Running sleep 60 &      # [1] – номер задачі у списку задач,
                                +  - задача виконується
$ jobs –l                    #  -l – вивести PID задачі
[1]+ 28875 Running sleep 60 &
```

Завершити виконання задачі можна командою `kill` із заданням `PID` задачі, номера задачі у черзі або імені задачі

```
$ kill %1
[1]+ Terminated sleep 60

$ kill 28875
[1]+ Terminated sleep 60

$ sleep 10 &
[1] 13692
$ kill %?sleep
$
[1]+ Terminated sleep 10
```

Команда `jobs –x` дозволяє підставити замість номера задачі у черзі її `PID`

```
$ printf ”%d\n” %1
bash: printf: %1: invalid number
$ jobs -x printf ”%d\n” %1
6259
```

Команда `jobs –p` висвічує тільки PID задачі.

```
$ jobs -p
6259
```

Сценарій використовує змінну `$$` для отримання ідентифікатора процесу `PID`, який Linux назначив сценарію:

```
echo "PID сценарію $$"
```

Запуск на виконання зупинених задач у в основному і фоновому режимі здійснюється командами `fg` і `bg`:

```
fg номер_задачі_у_списку
bg номер_задачі_у_списку
```

```
$ jobs
[1]+ Stopped ./test1
$ fg 1
[1]+ ./test2
```

```
$ jobs
[2]+ Stopped ./test2
$ bg 2
[2]+ ./test2 &
```

Фонову задачу можна вилучити із списку задач `bash` командою `disown`. Вилучена із списку задача продовжує виконуватися (наприклад MP3 players), але поза контролем `bash`.

```
$ disown %1
$ disown –a вилучення із списку всіх задач
$ disown –r вилучення із списку активних задач
```

Можна зупинити сценарій до завершення вказаної задачі у списку командою `wait`

```
$ wait %3  # чекати завершення третьої задачі у списку
$ wait     # чекати завершення всіх задач
```

1.2. Сигнали

Сигнал є різновидом програмного переривання. Сигнал є запитом на призупинення поточної задачі і перемикання на іншу термінову задачу. Коли bash отримує сигнал він завершує поточну команду, ідентифікує отриманий сигнал і здійснює необхідну дію. Якщо можливо, то по завершенню обробки сигналу bash продовжує виконання наступної команди.

Сигнали посилаються сценаріям командою `kill`. Команда `kill` не тільки завершує задачі сигналом `SIGTERM`, а й виконує інші дії, наприклад зупиняє і продовжує роботу задачі.

```
$ { sleep 60; echo ”DONE”; } &    # запуск команд у фоновому режимі
[1] 7613
```

```
#$ kill -SIGSTOP 7613              # зупинка виконання команди
$ kill -19 7613
[1]+ Stopped { sleep 60; echo ”DONE”; }
```

```
#$ kill -SIGCONT 7613  # продовження виконання команди з точки переривання
$ kill -18 7613
$ DONE
[1]+ Done { sleep 60; echo ”DONE”; }
```

Всього в ОС Linux визначено 64 різних сигнали. Числове значення і короткі символьні імена сигналів, визначених у мові Сі, показані в таблиці 1.

Для виконання сценарію у фоновому режимі без консолі використовується команда nohup. Команда nohup запускає іншу команду або сценарій та блокує сигнал SIGHUP, який посилається процесу. Це захищає процес від завершення при закритті консольної сесії. Так як команда вилучає асоціативний зв’язок процесу з терміналом, то процес втрачає зв’язок із стандартними

потоками STDIN та STDOUT. Тому при запуску команди nohup в основному і фоновому режимі ігнорується стандартний ввід, а стандартний вихід додається у файл nohup.out:

```
$ nohup ./test1 &
[1] 30797
$ nohup: ignoring input and appending output to `nohup.out'
```

Таблиця 1 – **Сигнали**

Числове значення	Символьне ім'я (мова Сі)	Описання
1	SIGHUP	З'єднання встановлено (hung up)
2	SIGINT (CTRL+C)	Перервати процес (interrupt) не виділяючи процесорний час (сигнал блокується і перехоплюється)
3	SIGQUIT (CTRL+D)	Завершити процес (як SIGTERM) і вивести дамп пам'яті
4	SIGILL	Недійсна інструкція (not reset)
5	SIGTRAP	Трасування пастки (trace trap)
6	SIGABRT	Вихід (abort)
7	SIGBUS	Помилка шини (Bus error)
8	SIGFPE	Виняткова ситуація з плаваючою крапкою
9	SIGKILL	Безумовно завершити процес (не блокується і не перехоплюється)
10	SIGUSR1	Визначено користувачем
11	SIGSEGV	Порушення сегментації
12	SIGUSR2	Визначено користувачем
13	SIGPIPE	Запис у канал з неможливістю зчитування
14	SIGALRM	Генерується таймером, встановленим функцією alarm()
15	SIGTERM	При можливості коректно завершити процес (блокується і перехоплюється)
16	SIGSTKFLT	
17	SIGCHLD	Зміна статусу нащадка
18	SIGCONT	Продовжити зупинений процес
19	SIGSTOP	Безумовно зупинити, але не завершувати процес
20	SIGTSTP (CTRL+Z)	Запит користувача на зупинку процесу (stop) без завершення від tty
21	SIGTTIN	Спроба фонового tty на читання
22	SIGTTOU	Спроба фонового tty на запис
23	SIGURG	Термінова умова на каналі вводу/виводу
24	SIGXCPU	Вичерпано ліміт часу ЦП
25	SIGXFSZ	Перевищено ліміт розміру файлу
26	SIGVTALRM	Вичерпаний час віртуального таймера
27	SIGPROF	Вичерпаний час таймера профілю
28	SIGWINCH	Змінено розмір вікна

29	SIGIO	Можливе введення/виведення
30	SIGPWR	Помилка живлення
31	SIGSYS	Помилковий системний виклик

1.3. Команда `suspend`

Команда `suspend` в сценарії зупиняє його виконання. Вона є аналогом команди `kill -SIGSTOP $$` (сценарій зупиняє сам себе).

`kill -SIGCONT` (продовження виконання зупиненого сценарію).

```
#!/bin/bash
# demo.sh - очікування сигналу SIGCONT і друк повідомлення
shopt -s -o nounset
shopt -s -o monitor # дозвіл контролю задач
# очікування сигналу SIGCONT
suspend
printf "%s\n" "Відновлено виконання задачі"
exit 0
```

```
$ bash demo.sh &
[1] 9185
$
[1]+ Stopped bash suspend_demo.sh
$ kill -SIGCONT 9185
$ Відновлено виконання задачі
[1]+ Done bash demo.sh
$
```

1.4. Пастки

Пастки. Коли bash виконується інтерактивно він обробляє більшість сигналів від імені користувача. При виконанні сценаріїв потрібні спеціальні дії для оброблення відповідних сигналів. Дії, виконувані bash, залежать від отримуваних сигналів:

- `SIGQUIT` завжди ігнорується
- `SIGTTIN` ігнорується, якщо контроль задач дозволений, інакше сценарій призупиняється;
- `SIGTTOU` ігнорується, якщо контроль задач дозволений, інакше сценарій призупиняється;
- `SIGTSTP` ігнорується, якщо контроль задач дозволений, інакше сценарій призупиняється;
- `SIGHUP` посилається всім фоновим задачам, запущеним сценарієм, запускаючи при необхідності на виконання зупинені задачі.

Наприклад, bash звичайно обробляє сигнал `SIGHUP` передаючи його всім підсценаріям. Таке саме спрацювання bash можна досягти і у сценарії, встановивши опцію `huponexit`.

Інші сигнали завершують роботу сценарію, незважаючи на створення обробника (пастки) сигналів. Вбудована команда `trap` керує обробником сигналів сценарію. Перелік всіх сигналів Linux з їх номерами:

```
$ trap -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
```

```
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL      10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM      15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU      25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO        30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1   36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11  46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14  51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9   56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

Не всі сигнали захоплюються пастками. Деякі сигнали, такі як `SIGKILL`, завжди завершують сценарій, навіть якщо встановлений обробник сигналів. Обробник сигналів створюється командою `trap`. Формат команди `trap`

```
trap команда_обробник_сигналу Сигнал_1 [ … Сигнал_N ]
```

Сигнали можна задавати назвою – `SIGTERM` або кодом 2.

Приклад сценарію, у якому цикл буде виконуватися до значення змінної `$count`=100 або до отримання сигналу `SIGINT`, формується натисканням клавіш CTRL+C (`interrupt`).

```
# trap.sh
#!/bin/bash
count=0
trap 'echo "Exit"; exit 1' 2
while [ $count -lt 100 ]
do
sleep 1
(( count++ ))
echo $count
done
```

```
./trap.sh
1
2
3
Exit
```

Після отримання сигналу `SIGINT`, `trap` виконає команди `echo "Exit"` і `exit 1` та завершить виконання. Сигнал, який потрібно перехопити заданий кодом 2.

Як команди `trap` може приймати функції, наприклад функцію `answer`:

```
# trap1.sh
#!/bin/bash
count=0

answer () {
while read response; do
echo
case $response in
[yY])
return 0
break
;;
```

```
[nN])
return 1
break
;;
*)
printf "Please, enter Y(yes) or N(no)! "
esac
done
}

trap 'printf "Are you sure to skip? [Y/n] "; answer && printf "Skipping..." &&
exit 1 ' SIGINT

while [ $count -lt 100 ]
do
sleep 1
(( count++ ))
echo $count
done
```

./trap1.sh

```
1
2
3
Are you sure to skip? [Y/n] n

4
5
6
Are you sure to skip? [Y/n] y

Skipping...
```

Можна заставити сценарій ігнорувати отримані сигнали (крім `kill -9`)

```
$ trap3.sh
#!/bin/bash
count=0
trap '' 2
while [ $count -lt 100 ]
do
sleep 1
(( count++ ))
echo $count
done
```

$./trap-2.sh

```
1
2
3
...
19
20
Terminated
```

Сценарій не можна перервати командою `kill` (за замовчуванням це `SIGTERM`, код 15). Якщо добавити код

```
trap  '' 2 15
```

То перервати виконання сценарію можна буде тільки за допомогою `kill -9`.

Приклад встановлення обробника сигналів для сигналу `SIGWINCH` (сигнал генерується при зміні розміру вікна)

```
$ trap 'printf "Рядків %s\n" "$LINES"' SIGWINCH
Рядків 28
Рядків 29
Рядків 30

$ trap -p SIGWINCH    # поточний стан пастки
$ trap SIGWINCH       # відновлення стану пастки
```

`SIGUSR1` і `SIGUSR2` – сигнали, які визначаються користувачем у сценаріях. Приклад сценарію, який отримує сигнал `SIGUSR1`, чекає завершення наступної команди і тоді друкує `SIGUSR1`

```
#!/bin/bash
# trap.sh

# Нескінчене очікування SIGUSR1, друк повідомлення, якщо його отримають

shopt -s -o nounset
trap 'printf "SIGUSR1\n"' SIGUSR1
# нескінчений цикл
while true ; do
sleep 1
done
printf "%s\n" "Це повідомлення ніколи не друкується!" >&2
exit 192

$ bash trap.sh &
[1] 544
$ kill -SIGUSR1 544
SIGUSR1
$ kill 544
[1]+ Terminated bash trap.sh
```

Обробники сигналів використовуються для запуску призупинених сценаріїв і тимчасового блокування сигналів для деяких критичних команд, які не повинні перериватися. Для тимчасового блокування сигналів використовуються порожні лапки "" або нульова команда ":", як обробник сигналів. Після виконання команд, які не повинні перериватися, відновлюється початковий обробник сигналів.

```
trap : SIGINT SIGQUIT SIGTERM # Блокування обробників сигналів
trap   SIGINT SIGQUIT SIGTERM # Відновлення обробників сигналів
```

Обробник виходу є набором команд, які виконуються при завершенні сценарію. Як обробник виходу використовується неіснуючий сигнал `EXIT`. Обробник виходу може видати повідомлення або викликати функцію.

```
#!/bin/bash
# Виводиться повідомлення ”Goodbye” при виході з сценарію
shopt -s -o nounset
trap ’printf ”Goodbye\n”’ EXIT
sleep 5
exit 0
```

1.5. Команда Linux `killall`

Команда Linux `killall` завершує програми за їх іменами, а не за ідентифікатором процесу PID.

```
$ sleep 15 &
[1] 1225

$ killall sleep
[1]+ Terminated sleep
```

1.6. Керування пріоритетами

У Linux всі задачі виконуються із пріоритетами від -20 (найбільший) до 19 (найменший). Linux команда `nice` змінює номер пріоритету команд і сценаріїв. Звичайний користувач може тільки зменшувати пріоритет, а суперкористувач може зменшувати і збільшувати пріоритет любої задачі. За замовчуванням `nice` понижує пріоритет до 10. При запуску команд або сценаріїв у фоновому режимі пріоритет також автоматично понижується. Формат команди

```
nice  -номер_пріоритету команда | сценарій &
nice  --adjustment=номер_пріоритету команда | сценарій &
$ nice –n 20 ./test1 &

$ nice –n 20 sleep 60 &

$ nice --adjustment=20 sleep 60 &

$ nice --adjustment=20 my_script.sh &
```

Команда `renice` змінює пріоритет фонової задачі, якою володіє користувач.

```
$ bash my_script.sh &
[1] 22516

$ renice 10 –p 22516
22516: old priority 0, new priority 10
```

1.7. Статус процесу

Команда `jobs` надає основну інформацію про задачі фонового режиму. Повну інформацію про процеси основного і фонового режимів надає команда `ps (process status)`. Команда має багато ключів і аргументів. Найбільш корисну інформацію про процеси надає команда `ps -l` (або `ps al`). Перегляд зупинених задач `ps au`.

```
$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1001  4717  4715  0  80   0 -  5678 wait   pts/1    00:00:00 bash
0 R  1001 16278  4717  0  80   0 -  2649 -      pts/1    00:00:00 ps
```

S – стан процесу:

```
     D – очікування введення/виведення
     R – виконання або очікування виконання
     S – очікування (sleeping)
     T – трасується або зупинений
     Z – непрацюючий процес, який очікує вилучення ( zombie )
UID – ідентифікатор користувача
PID – ідентифікаційний номер процесу
PPID - ідентифікаційний номер батьківського процесу
PRI – значення Linux пріоритету
NI - nice значення пріоритету
TTY - tty пристрій, який використовує програма (який повертає команда tty)
TIME—CPU час, використаний процесом
CMD – ім'я команди
```

2. Запуск завдань за таймером і календарем

2.1. Команди at і batch

Команда `at` дозволяє планування *разового* виконання завдання у заданий час і дату. В залежності від дистрибутиву Linux команда може бути не встановленою. Встановлення команди

в Ubunty

```
sudo apt update      # оновлення репозиторію
sudo apt install at  # інсталяція команди
```

в Fedora

```
sudo yum -y update   # оновлення репозиторію
sudo yum install at  # інсталяція команди
```

Після встановлення програми `at` потрібно активувати сервер планування і його запуск при перезавантаженні системи:

```
sudo systemctl enable --now atd
```

Синтаксис команди:

```
at [опції] time
```

Команда має багато опцій і основні з них:

`-q [queue]`, використати чергу, де queue задається буквами від `a` до `z`. За замовчування `a` черга для `at`, `b` черга для `batch`.

`-f [filename]`, *`filename`* – команда або сценарій, який потрібно запустити на виконання;

`-m` надіслати на e-mail користувача повідомлення про завершення завдання.

`-b` керувати і виконувати завдання у `batch` черзі, якщо рівень середнього завантаження системи менше 1.5.

`-d` вилучення завдань із черги за їх номером.

`-l` список усіх незавершених завдань.

`-t [time_arg]` – час запуску завдання (ГГ.ХХ, наприклад 20.45).

Час можна задати в абсолютних одиницях:

- `YYMMDDhhmm[.ss]`. Абревіатури рік, місяць, день, хвилини і опційно секунди.
- `CCYYMMDDhhmm[.ss]`. Повний рік, місяць, день, хвилини і опційно секунди.
- `now`. Поточний день і час, негайне виконання.
- `midnight`. Задає 00:00 AM.

- `noon`. Задає 12:00 PM (після 12.00).
- `teatime`. Означає 4 PM (16.00).
- `AM`. Означає час до 12:00 PM.
- `PM`. Означає час після 12:00 PM.
- `today`. Поточний день.
- `tomorrow`. День, після поточного.

```
echo "Привіт світ" | at 5PM
```

або у відносних одиницях, добавленням символу '+':

```
• minutes
• hours
• days
• weeks
• months
• years
```

```
echo "hello" | at now +5 minutesat -l
```

Команда `at` дозволяє переглядати, які завдання все ще перебувають у черзі та очікують на виконання. Є два способи переглянути незавершені завдання:

```
sudo atq
```

Завдання можна вилучити із черги командою `atrm`. Перегляд вмісту попередньо запланованого завдання:

```
at -c [job_number],
```

`job_number` можна отримати командою `atq`.

Команда `at` надсилає завдання у чергу в спеціальний каталог (`/var/spool/at`), з якого bash буде їх запускати. Команда `atd` перевіряє каталог із завданнями кожні 60 сек і якщо поточний час співпадає із часом завдань, `atd` запускає їх на виконання. Команда `atd` запускається у режимі суперкористувача: `atd start`. По закінченню команди `at` генерується e-mail повідомлення створене виводом сценарію у `/var/spool/mail/user`.

Приклад виконання команди у командному рядку

```
$ at 18:00
at> echo "hello word"
```

Натиснути клавіші Enter, а потім Ctrl+D

```
at><EOT>
You have new mail in /var/spool/mail/user
```

Приклад виконання сценарію

```
$ at 18:00 -f /user/mysh/1.sh
$ at -f /user/mysh/1.sh now +2 minutes
```

Приклад використання конвеєра

```
echo "echo 'Hello word'" | at now +5 minutes
```

Приклад використання блоку введення

```
$ at 14:00 <<END
> echo "Hello word"
> END
```

Приклад виконання команди без надсилання e-mail повідомлення:

```
echo "shutdown -h now" | at -M 00:00
```

Команда batch. Запускати задачі на виконання при зменшенні навантаження на систему можна командою `batch`. Команда `batch` перевіряє середнє завантаження системи і якщо воно менше 1.5, вона запускає на виконання любу задачу із черги задач або вказану задачу.

Запуск з командного рядка

```
$ batch
at> 1.sh
at <EOT>
```

`<EOT>` – натискання клавіші Ctrl+D.

Запуск з використанням конвеєра

```
echo "2.sh" | batch
echo "echo 'Hello word'" | batch
```

Приклад використання блоку введення

```
$ batch <<END
> echo "Hello word"
> END
```

За замовчуванням завдання, створені за допомогою `at`, поміщаються у чергу з назвою `a`, а завдання, створені за допомогою `batch`, поміщаються у чергу `b`. Черги можуть мати назви від `a` до `z` і від `A` до `Z`. Черги з меншими літерами виконуються з нижчим пріоритетом. Можна вказати чергу за допомогою опції `-q`. Приклад встановлення завдання у чергу L:

```
$ at monday +2 hours -q L
```

2.2. Команда cron

Для регулярного запуску завдань (за календарем, дата, час) призначена програма **`cron`**. Cron є одним із найкорисніших інструментів в Linux. Звичайно він використовується для завдань системного адміністратора, таких як резервне копіювання або очищення каталогів /tmp/ тощо. Служба `cron` (демон) працює у фоновому режимі та постійно перевіряє файл `/etc/crontab` і каталоги `/etc/cron.*/`. Вона також перевіряє каталог `/var/spool/cron/`. Кожний користувач може мати власний конфігураційний файл `crontab` у каталозі `/var/spool/cron/crontabs`. Файл можна редагувати за допомогою команди `crontab`.

Типи конфігураційних файлів:

1. Системний `crontab` Linux звичайно використовується системними службами та критичними завданнями, для яких потрібні права `root`. Можна налаштувати виконання команд від імені звичайного користувача.

2. Користувацький `crontab` дозволяє користувачу встановлювати власні завдання `cron` за допомогою команди `crontab`.

Для створення `cron` таблиці потрібні права суперкористувача (root):

```
$ crontab [-u user] [ -e -l -r ]
# -e створити крон таблицю
# -l список крон таблиць
# -r знищити крон таблицю
```

Описання полів файлу `crontab`:

```
1-хвилини 2-години 3-день_місяця 4-місяць 5-день_тижня команда
1 2 3 4 5 /path/to/command arg1 arg2
```

де 1 – хвилини (0-59), 2 – години (0-23), 3 – день (0-31), 4 – місяць (0-12), 5 – день тижня (0-7, 0 або 7 неділя).

Завдання `cron` для системних завдань виглядає так:

```
1 2 3 4 5 USERNAME /path/to/command arg1 arg2
1 2 3 4 5 USERNAME /path/to/script.sh
```

Параметри, які не використовуються, вказуються ”*”. Наприклад, для запуску команду або сценарію в 20 год 30 хв кожного дня, запис матиме наступний вид:

```
30 20 * * *  /home/user1/test1.sh > test1.out
```

Оператор для задання декількох значень у полі:

* – всі можливі значення;
, – список значень (1,5,10,15,20);
- – діапазон значень (5-10 => 5,6,7,8,9,10);
/ – значення із кроком (у полі годин 0-23/ означає виконання кожну годину. Кроки також дозволені після *, наприклад */2 означає кожні дві години).

За замовчуванням результат команди або сценарію буде надіслано на локальний обліковий запис електронної пошти. Щоб припинити отримання електронної пошти від `crontab`, потрібно додати `>/dev/null 2>&1`. Наприклад:

```
23 0-23/2 * * * /root/scripts/perl/perlscript.pl >/dev/null 2>&1
```

Замість перших п'яти полів можна використовувати будь-який з восьми спеціальних рядків:

`@reboot` виконати один раз при старті
`@yearly` виконати один раз в рік (0 0 1 1 *)
`@annually` так само як `@yearly`
`@monthly` виконати один раз в місяць (0 0 1 * *)
`@weekly` виконати один раз в тиждень (0 0 * * 0)
`@daily` виконати один раз в день (0 0 * * *)
`@midnight` так само як `@daily`
`@hourly` виконати один раз в годину (0 * * * *)

Для отримання повідомлень від `crontab` на потрібну адресу електронної пошти потрібно визначити змінну MAILTO:

```
MAILTO: student@gmail.com
23 0-23/2 * * * /root/scripts/perl/perlscript.pl >/dev/null 2>&1
```

Список усіх `cron` завдань:

```
crontab -l
crontab -u username -l
```

Вилучення усіх завдань із

```
crontab -r
crontab -u username -r
```

2.2.1. Файл `/etc/crontab` і каталог `/etc/cron.d/*`

`/etc/crontab` - системний файл. Зазвичай використовується лише користувачем `root` або демонами для налаштування системних завдань. Кожен окремий користувач повинен використовувати команду `crontab` для встановлення та редагування своїх завдань, як описано вище. `/var/spool/cron/` або `/var/cron/tabs/` - це каталог для особистих файлів `crontab` користувача. Це має бути резервна копія з домашнім каталогом користувача.

Типовий файл `/etc/crontab`:

```
SHELL=/bin/bash
PATH=/sbin:/bin:/usr/sbin:/usr/bin
MAILTO=root
HOME=/

# run-parts
01 * * * * root run-parts /etc/cron.hourly
02 4 * * * root run-parts /etc/cron.daily
22 4 * * 0 root run-parts /etc/cron.weekly
42 4 1 * * root run-parts /etc/cron.monthly
```

Крім того, cron читає файли з каталогу `/etc/cron.d/`. Зазвичай системний демон, такий як `sa-update` або `sysstat`, розміщує тут свій `cronjob`. Користувач `root` або суперкористувач може використовувати наступні каталоги для налаштування завдань `cron`. Користувач може безпосередньо перекинути свої сценарії сюди. Команда `run-parts` запускає сценарії або програми в каталозі через файл `/etc/crontab`:

```
/etc/cron.d  помістити усі сценарії сюди і викликати їх з /etc/crontab файлу
/etc/cron.daily/  виконати усі сценарії один раз за день
/etc/cron.hourly/  виконати усі сценарії один раз за годину
/etc/cron.monthly/ виконати усі сценарії один раз за місяць
/etc/cron.weekly/  виконати усі сценарії один раз за тиждень
```

Приклад сценарію, який буде очищати кеш кожні 10 днів. Сценарій з іменем `clean.cashe` створюється безпосередньо в каталозі `/etc/cron.daily/`:

```
#!/bin/bash
# Простий сценарій очищення кешу веб сервера lighttpd
CROOT="/tmp/cachelighttpd/"
# Очищення файлів кожні $DAYS
DAYS=10
# Ім'я користувача веб сервера і ім'я групи
LUSER="lighttpd"
LGROUP="lighttpd"
# Запуск очищення кожні $DAYS
/usr/bin/find ${CROOT} -type f -mtime +${DAYS} | xargs -r /bin/rm
# Failsafe
# якщо каталог вилучено іншими сценаріями то відновити його
if [ ! -d $CROOT ]
then
        /bin/mkdir -p $CROOT
        /bin/chown ${LUSER}:${LGROUP} ${CROOT}
fi
```

Зробити сценарій виконуваним:

```
$ chmod +x /etc/cron.daily/clean.cache
```

Перегляд журналу виконання:

```
$ cat /var/log/cron
sudo systemctl status cron
sudo journalctl -u cron
sudo journalctl -u cron | grep backup-script.sh
```

2.3. Команда anacron

Команда `cron` не контролює виконання завдань у випадку відмови системи. Цей недолік усуває команда `anacron`, яка ставить часові позначки для визначення виконання задачі. Якщо задача у заданий час не було виконана, то `anacron` запустить її на виконання при першій можливості. Команда `anacron` має свою таблицю, розміщену в `/etc/anacrontab`. Формат записів таблиці `anacron`:

`період затримка ідентифікатор команда`

`період` – частота виконання команди в днях;

`затримка` – затримка в хвилинах на запуск команди;

`ідентифікатор` – унікальний ідентифікатор задачі в журналі повідомлень і e-mail помилок.

Контрольні запитання.

1. Як запустити задачу в основному і фоновому режимі, перевести із режиму в режим?
2. Яке призначення команд `jobs, disown` і їх ключі?
3. Яке призначення команд `kill, killall, suspend` і їх основні ключі?
4. Яке призначення команди `wait` і її ключі?
5. Що таке сигнал, які функції він виконує і як надсилається?
6. Що таке пастка, які функції вона виконує?
7. Яке призначення команди `trap`, обробника сигналів і обробника виходу?
8. Для чого існують пріоритети, як їх змінити командою `nice, renice`?
9. Яку інформацію виводить команда `ps` і її основні ключі?
10. Яке функціональне призначення команд `at, atd, batch`?
11. Команда `at` і налаштування завдань на виконання.
12. Яке функціональне призначення команд `cron, anacron`?
13. Команда `cron` і налаштування завдань на виконання.

2. Практична частина

Завдання.

1. Написати сценарій, який у фоновому режимі, без консолі, записуватиме у файл статистику запусків: номер запуску, номер задачі, `PID` сценарію, час і дату запуску на виконання, час і дату закінчення виконання, загальний час виконання.
2. Написати сценарій, який щодня архівує поточний каталог.
3. Написати сценарій, який щодня архівує тільки змінені файли поточного каталогу.

4. Написати сценарій, який підраховує кількість файлів у поточному каталозі, а при закінченні, з використанням сигналу `EXIT`, виводить підраховану кількість файлів на екран.

5. Написати сценарій, який щодня робить архів каталогу Documents.

6. Написати сценарій, який виводить повідомлення “Привіт” через через 2 хвилин після завантаження комп’ютера (формат `Now + 5 minutes`).

7. Написати сценарій, який раз в місяць корегує та оновлює пакети.

8. Написати сценарій, який на електронну пошту на початку місяця надсилає поточний курс валют.

9. Написати сценарій, який кожного тижня завантажує розклад занять у поточний каталог.

10. Написати сценарій, який запускається у консолі і продовжує свою роботу після закриття консолі.

11. Написати сценарій, який у 24.00 виводить на екран повідомлення “Кінець робочого дня. Добраніч”.

12. Використати команду `batch` для запуску завдання із черги задач на архівацію сирцевих кодів поточного каталогу при зменшенні завантаження на систему.

13. Написати сценарій, який раз в місяць надсилає не електронну пошту розклад поїздів.

14. Написати сценарій, який надсилатиме на електронну пошту повідомлення про перехід на літній/зимовий час.

15. Написати сценарій, який кожного дня зранку надсилатиме повідомлення на електронну пошту про необхідність виконання певних дій із тижневого журналу.

Примітка. Номер варіанту завдання вибирається за порядковим номером студента у журналі групи.

Звіт з лабораторної роботи має містити:

- назву групи, П.І.Б. студента, завдання до роботи;
- текст програми із коментарями;
- роздрук екранів із результатом запуску програми.

Приклади

1. Перехоплення сигналу CTRL+C і виконання команди echo

```
# 1.sh
#!/bin/bash
trap "echo Ха-ха!" SIGINT SIGTERM
echo "Це тестова задача"
count=1
while [ $count -le 5 ]
do
echo "Цикл #$count"
sleep 3
count=$[ $count + 1 ]
done
echo "Кінець програми"
```

```
$ ./1.sh
Цикл #1
Цикл #2
Ха-Ха!
Цикл #3
Цикл #4
```

Цикл #5
Кінець програми

2. Перехоплення команди exit і виконання команди echo

```
# 2.sh
#!/bin/bash
trap "echo Перехоплення команди EXIT" EXIT
count=1
while [ $count -le 5 ]
do
echo "Цикл #$count"
sleep 3
count=$[ $count + 1 ]
done
```

```
$ ./2.sh
```
Цикл #1
Цикл #2
Цикл #3
Цикл #4
Цикл #5
Перехоплення команди EXIT

3. Вилучення встановленої пастки опцією trap "-"

```
# 3.sh
#!/bin/bash
trap "echo захват EXIT" EXIT
count=1
while [ $count -le 5 ]
do
echo "Цикл #$count"
sleep 3
count=$[ $count + 1 ]
done
trap - EXIT
echo "Пастка вилучена"
```

$ ./3.sh
Цикл #1
Цикл #2
Цикл #3
Цикл #4
Цикл #5
Пастка вилучена

3а. Запуск задачі в фоновому режимі

```
# 3a.sh
#!/bin/bash
./1.sh &
```

```
./3a.sh
```
Цикл #1
Цикл #2
<ENTER>
ps au
Цикл #3
Цикл #4

Цикл #5
Кінець програми

3b. Запуск декількох фонових задач

```
# 3b.sh
#!/bin/bash
echo "Запуск першої фонової задачі"
./1.sh &
echo "Запуск другої фонової задачі"
./1.sh &
```

```
> ./3b.sh
```
Цикл #1
Цикл #2
<ENTER>
ps au
…
Цикл #5
Цикл #5
Кінець програми
Кінець програми

4. Тестування команди jobs

```
# 4.sh
#!/bin/bash
# команда $$ виводить PID сценарію
# після першого циклу зупинити програму - CTRL+Z
# визначити номер зупиненого завдання командою jobs
# відновити роботу програми в основному режимі fg N (де N - номер задачі)
# або фоновому режимі bg N
echo "Це тестова програма PID=$$"
count=1
while [ $count -le 5 ]
do
echo "Цикл #$count"
sleep 5
count=$[ $count + 1 ]
done
echo "Кінець тестової програми"
```

```
$./4.sh
```
Це тестова програма 4155
Цикл #1
Цикл #2
Цикл #3
Цикл #4
Цикл #5
Кінець тестової програми

5. Зміна пріоритетів

```
# 5.sh
#!/bin/bash
# пріоритети -10(самий високий),  0(сценарій),  10 (самий низький)
# зміна пріоритету задачі
nice -n 10 ./5 > 5.txt &
```

```
# список задач які виконуються
ps al
# зміна пріоритету задачі під час її виконання
# renice 10 -p  PID

$ ./5.sh
```

6. Виконання сценарію без консолі (у фоновому режимі)

```
# 6.sh
#!/bin/bash
nohup ./1.sh &
cat nohup.out

$ ./6.sh
```

Цикл #1
Цикл #2
Цикл #3
Цикл #4
Цикл #5
Кінець програми

7. Виконання сценарію без консолі (у фоновому режимі)

```
# 7.sh
#!/bin/bash
# Функція для виконання роботи
do_work() {
    while true
    do
        echo "$(date): Виконується робота..." >> /tmp/background.log
        sleep 60  # Пауза на 60 секунд
    done
}

# Експорт функції, щоб вона була доступна для subshell
export -f do_work

# Запуск функції у фоновому режимі, використовуючи bash -c
nohup bash -c do_work &

echo "Сценарій запущено у фоновому режимі. Результат у лог-файлі /tmp/background_script.log"

$ ./7.sh
```

8. Тестування команди at

```
#Для виконання команди at має бути запущений демон atd із правами root
# 8.sh
#!/bin/bash
time=$(date +%T)
echo "Сценарій виконується в $time"
echo "Кінець сценарію" >&2

# ввести відповідний час
$ at -f 8.sh 14:00
```

Лабораторна робота 8

Функції

Мета роботи: навчитися створювати і використовувати функції Bash.

Теоретичний матеріал лекції, технічна література, ресурси інтернету.

1. Середовище програм і сценаріїв

Коли запускається Linux програма, вона не виконується у пустому середовищі, а успадковує всі змінні експортовані попередньою програмою. Експортовані змінні називаються змінними середовища, так як вони є частиною середовища в якому виконується програма. Середовище виконання *програми* містить:

- відкриті файли;
- поточний робочий каталог;
- режим створення файлів `umask`;
- обробники сигналів встановлені програмою `trap`;
- експортовані змінні середовища;
- експортовані функції оболонки bash;
- опції доступні за замовчуванням або встановлені командою `set`;
- опції bash `shopt`;
- bash синоніми (`aliases`) команд;
- ідентифікатори (`ID`) різних процесів, включно з фоновими задачами, значення `$$`, і значення `$PPID`.

Середовище виконання *сценарію* містить:

- відкриті файли (можливо модифіковані або їх дескриптори перенаправлені підсценаріями);
- поточний робочий каталог;
- значення `umask`;
- значення змінних bash, позначені для експорту;
- сигнали пасток (`traps`).

Підсценарії успадковують своє середовище від батьківського сценарію.

2. Створення функцій

Існує два формати описання функцій. У першому форматі функція оголошується командою `function name`, а у другому задається тільки ім'я функції `name()`:

```
function name {
  <commands>
}

function name { <commands>; }
```

```
name() {
  <commands>
}

name() { <commands>; }
```

Функції викликаються за своїми іменами:

```
$ name
```

Імена функцій мають бути унікальними, а визначення функцій – розміщуватися *на початку* сценарію. Для запобігання перевизначення функцій використовується команда

```
>readonly -f my_fun
```

Змінні `$SCRIPT` і `$FUNCNAME` містять імена сценарію і функції. Вони використовуються при визначенні місця виникнення помилок `echo $SCRIPT $FUNCNAME`.

Так як функція не є сценарієм вона повертає код завершення і декларовані змінні по іншому. У функції команда `exit` завершує сценарій. Для повернення коду без завершення функції використовується команда `return`. Команда `return` може повертати як код фіксоване значення `return 100` або обчислюване значення `return $[ $value * 2 ]`.

При завершенні, без команд `exit` і `return`, функція повертає значення останньої виконаної команди, наприклад команди `echo "msg"`.

Значення `exit` коду останнього завершеного процесу містить спеціальна змінна `$?`.

Код повернення функції можна присвоїти змінній, для подальшого його використання:

```
my_fun() {
  echo Hello
  return 100
}

result=$(myfun)
```

Для того щоб функція була доступною за межами її визначення іншим сценаріям, потрібно використати команду `export`

```
$export -f my_fun
```

Визначення, в якому файлі оголошена функція:

```
$ bash -debugger
$ declare -F my_fun
my_fun 115 /home/.basrc
```

Виведення вмісту функції:

```
$ declare -f my_fun

export -f my_fun
my_fun()
{
   echo "Привіт Світ"
}
```

Звільнення простору імен від імені функції:

```
unset my_fun
```

2.1. Змінні функцій

Функції використовують два типи змінних:

– глобальні;

– локальні.

Змінні оголошені в основному сценарії і функціях є *глобальними*. Для оголошення локальних змінних всередині функцій і присвоєння їм значень може використовуватися команда `local`, але вона не може встановлювати атрибути змінних. Тому рекомендується для

оголошення локальних змінних замість команди `local` використовувати команду `declare` або її аналог `typeset`. Якщо змінна явно не оголошена (тобто є глобальною), то вона не знищується при завершенні функції, що може спричиняти неочікувані результати. Якщо глобальна і локальна змінні мають однакове ім'я, локальна змінна перекриває глобальну. Локальні змінні знищуються при завершенні роботи функції.

```
$ function f { COMPANY="Ivan and brothres Inc" ; }
$ f
$ printf "%s\n" "$COMPANY"
Ivan and brothres Inc

$ function f { declare COMPANY="Ivan and brothres Inc" ; }
$ f
$ printf "%s\n" "$COMPANY"
$
```

Всі змінні Bash зберігаються як *символьні рядки*. Кожна змінна має атрибути, які можна встановити командою `declare`, приклад оголошення цілочислової змінної `nomer`

```
declare –i nomer=15
printf "%d\n" $i
nomer="Hello" # помилка: присвоєння цілочисловій змінній значення стрічки
```

Змінну можна оголосити константою `declare -r` (`read-only`).

Атрибути змінної або функції можна вивести командою

```
declare –i my_var=1
declare –p my_var
declare my_fun
declare –p my_fun
```

Якщо змінна оголошена командою `declare` в основному сценарії, то вона є *глобальною*, а якщо всередині функції, то вона є *локальною*.

```
# Global Declarations
declare -rx SCRIPT=${0##*/} # SCRIPT є ім'я цього сценарію
declare -rx who="/usr/bin/who" # команда who
echo $SCRIPT
```

Змінні оголошені командою `declare` існують до завершення функції або сценарію, або до їх руйнування командою `unset`. Для того щоб змінні були доступними за межами їх визначення іншим сценаріям, потрібно оголосити їх як експортовані (атрибут `-x`, `export`), наприклад:

```
declare -x CVSROOT="/home/cvs/cvsroot".
```

```
#outer.sh
declare -rx COMPANY_NAME="Ivan and brothres Inc"
bash inner.sh
printf "%s\n" "$COMPANY_NAME"
exit 0

#inner.sh
declare -p COMPANY_NAME
COMPANY_NAME="Ivan and brothres Inc"
printf "%s\n" "$COMPANY_NAME"
```

Наявність локальних змінних дозволяє створювати рекурсивні функції. Рекурсивні функції використовують локальні змінні і тільки ті зовнішні змінні, які передаються як аргументи функції у командному рядку.

```
#!/bin/bash
# factorial.sh: рекурсивна функція
shopt –s –o nounset
declare -rx SCRIPT=${0##*/}
declare -i REPLY
# FACTORIAL : обчислення факторіалу
# $1 – число, для якого обчислюється факторіал

function factorial {
declare -i RESULT=1 # використовується функцією factorial1
function factorial1 {
declare -i FACT=$1 # поточне число
let ”FACT--” # декремент
if [ $FACT -gt 1 ] ; then # більше ніж 1?
factorial1 $FACT # повторити
let ”RESULT=RESULT*FACT” # множення результату
else # інакше
RESULT=1 # factorial of 1 is 1
fi
return # вихід з функції
}

factorial1 $1 # старт із параметром 1
printf ”%d\n” $RESULT
}
readonly -f factorial
declare -t factorial
printf ”Факторіал якого числа? –> ”
read REPLY
factorial $REPLY
exit 0
```


2.2. Аргументи функцій

Функція може використовувати стандартні аргументи змінних середовища. Ім’я функції визначено у змінній `$0`, аргументи функції – у змінних `$1,…,$9`, число аргументів функції – `$#`, розширення списку аргументів до окремих слів (`”$1” ”$2” ...`) - `$@`, розширення списку аргументів в одну символьну стрічку (`”$1 $2 ... ”`) - `$*`. Так як функція використовує змінні середовища для своїх аргументів, то вона не має доступу до аргументів сценарію, які передаються через командний рядок.

2.3. Команда `source`

Вбудована у bash команда `source` читає і виконує вміст сценарію/файлу. Вона має наступні особливості:

• шукає сценарій у каталогах описаних у змінній PATH поточної оболонки;

• розпізнає абсолютні або відносні шляхи вказані у сценарії;

• сценарій не обов’язково має бути виконуваним;

• повертає код завершення останньої виконаної команди сценарію;

• якщо у сценарії є команди `exit`, `source` припиняє роботу.

Якщо вмістом файлу є сценарій, то він виконується. Наприклад файл `lib` має наступний вміст:

```
#!/bin/bash
echo ”Привіт світ”
```

Тоді, результатом виконання команди `$ source lib` буде

```
Привіт світ
```

Команда `source` дозволяє імпортувати набір змінних середовища у сценарій. Так можна читати і встановлювати змінні із `.profile` користувача, завдання `cron`.

```
#!/bin/bash
source /etc/profile
source ~/.bash_profile
source ~/.bashrc
# do something useful
```

Команда `source` дозволяє створювати бібліотеку повторно використовуваних функцій. Наприклад файл `lib` містить наступні функції:

```
my_name(){
  whoami
}

my_disk(){
  du -sh ~ | cut -f1
}
```

Сценарій `test.sh`, який використовує функції з бібліотеки:

```
#!/bin/bash
source /home/user/mylibs/lib
echo ”Моє ім’я $(my_name), мій диск $(my_disk)
```

```
$ ./test.sh
Моє ім’я user, мій диск 100 G
```

Якщо сценарій і бібліотека знаходяться в одному каталозі, то шлях можна явно не вказувати:

```
source $(dirname ”$0”)/lib
```

Функції основного сценарію можна зібрати і записати в один бібліотечний файл. Команда `source` підключає бібліотечний файл всередині основного сценарію. Це дає доступ основному сценарію до функцій бібліотечного файлу. Команда `source` має синонім (aliases) `.` (крапка). Так, сценарій `source1.sh`

```
#!/bin/bash
# source1.sh
shopt -s -o nounset
declare -rx SCRIPT=${0##*/}
declare -r source2=”source2.sh”
if test ! -r ”$source2” ; then
  printf ”SCRIPT: the command $source2 is not available\n” >&2
  exit 1
fi
```

```
# включення файлу source2.sh
source $source2
printf "The variable YEAR is %s\n" "$YEAR"
exit 0
```

можна виконати командою

```
$. ./source1.sh
```

Файл `source2` є фрагментом, а не окремим сценарієм і буде вставлений в файл `source1.sh`:

```
declare -r YEAR='date '+%Y''
```

Для того щоб функції бібліотечного файлу були доступні при завантаженні bash їх можна помістити у файл `.bashrc` або у каталог `/home/user/.local`, який описаний у змінній `$PATH`.

При виконанні зовнішнього сценарію або вставленні фрагменту командою `source`, керування завжди повертається до наступного рядка основного сценарію. Команда `exec` передає керування новому сценарію без повернення до старого

```
#!/bin/bash
# файл exec1.sh
shopt -s -o nounset
declare -rx SCRIPT=${0##*/}
declare -r exec2="./exec2.sh"
if test ! -x "$exec2" ; then
printf "$SCRIPT:$LINENO: скрипт $exec2 недоступний\n" >&2
exit 192
fi
printf "$SCRIPT: передача керування $exec2 без повернення...\n"
exec "$exec2"
printf "$SCRIPT:$LINENO: помилка виконання exec!\n" >&2
exit 1

#!/bin/bash
# файл exec2.sh
declare -rx SCRIPT=${0##*/}
printf "$SCRIPT: тепер виконується exec2.sh"
exit 0

$ bash exec1.sh
exec1.sh: передача керування exec2.sh без повернення...
exec2.sh: тепер виконується exec2.sh
```

3. Сценарій демон

Демони (більшість серверів є демонами) – це програми, які виконуються незалежно від bash сесії і постійно виконують деяку задачу. Команда Linux `nohup` виконує команду так, що вона не завершується після від'єднання від сесії bash. Команда `nohup` понижує пріоритет виконуваної команди і перенаправляє стандартний потік виведення у файл `nohup.out` (так як сесія bash завершиться). Щоб уникнути створення файлу `nohup.out` потрібно закрити стандартний потік виведення або перенаправити його із кінцевого сценарію у сценарій демон.

Так як `nohup` не запускає команди у фоновому режимі, то це має роботи кінцевий сценарій командою `&`.

Сценарій демон має нескінчений цикл в якому він перевіряє виконувану роботу. Для перевірки роботи демона використовується два методи: опитування або блокування. Демон виконується аж поки його не зупинять командами `suspend` або `kill`.

Контрольні запитання.

1. Що містить середовище виконання програми і підсценарію?
2. Область видимості змінних у Bash сценарії. Як оголошуються локальні і глобальні змінні.
3. Як задати і перевірити атрибути змінних і функцій?
4. Формати оголошення функцій. Що місять змінні середовища `$SCRIPT i $FUNCNAME?`
5. Які значення може повертати функція і як їх отримати?
6. Які змінні середовища використовує функція для передачі аргументів?
7. Як передати аргументи командного рядка сценарію, як аргументи у функцію.
8. Як передати вектор змінних у функцію?
9. Як отримати повернути вектор змінних із функції?
10. Як змінити значення вектора змінних у функції?
11. Яка різниця між виконанням сценарію командами `source` і `exec`?
12. Які особливості команди `source`?
13. Які можливості команди `source`?
14. Створення бібліотечного файлу функцій і підключення його до сценарію?
15. Як забезпечити доступ до бібліотечного файлу функцій з різних каталогів?
16. Реалізація рекурсивних функцій.
17. Сценарій демон, як він запускається, функціонує і завершується.

4. Практична частина

Завдання.

1. Для довільних двох рядків тексту з файлу визначити віддаль Хеммінга (кількість символів які попарно не співпадають), наприклад

```
GAGCCTACTAACGGGAT
CATCGTAATGACGGCCT
^  ^ ^  ^   ^      ^^
1  2 3    4   5      67
```

Віддаль Хеммінга 7

2. Для довільного числа, введеного з консолі, визначити чи воно є числом Армстронга. В числі Армстронга сума окремих розрядів у степені кількості цифр дорівнює самому числу.

- 9 є число Армстронга, так як 9 = 9^1 = 9
- 10 не є число Армстронга, так як 10 != 1^2 + 0^2 = 1
- 153 є число Армстронга, так як:
- 153 = 1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153
- 154 не є число Армстронга, так:
- 154 != 1^3 + 5^3 + 4^3 = 1 + 125 + 64 = 190

3. Написати програму для перевірки достовірності номерів кредитних карточок. Номер карточки вводити з консолі.

Перевірка номерів кредитних карточок (Luhn algorithm).

Задати номер карточки 4539 3195 0343 6467

Подвоїти кожну другу цифру справа і якщо результат більший від 9

то відняти від нього 9 8569 6195 0383 3437.

Порахувати суму всіх цифр 8+5+6+9+6+1+9+5+0+3+8+3+3+4+3+7 = 80

Якщо сума ділиться на 10, то кредитний номер достовірний.

4. Написати програму для перевірки у текстовому файлі коректність парності і вкладеності круглих (), квадратних [], фігурних {} і кутових <> дужок.

5. Реалізувати розбір речень і виконати математичні операцію для цілих чисел a, b. Значення a, b вводити з консолі. Речення для розбору записати у файл.

Ввести a, b

Додати a, b

Відняти a, b

Помножити a, в

6. Реалізувати алгоритм стискання і розтискання тексту на основі довжини повторюваних символів.

"AABCCCDEEEE" -> "2AB3CD4E" -> "AABCCCDEEEE"

7. За заданими довжинами сторін трикутника визначити, чи він є рівностороннім, рівнобедреним чи косокутним. У рівностороннього трикутника всі три сторони однакові. Рівнобедрений трикутник має принаймні дві сторони однакової довжини. Косокутний трикутник має всі сторони різної довжини. Довжини сторін трикутника вводити з консолі.

8. Перетворити число a_b (у системі b) в число c_d (у системі d). Числа a, b і системи числення b, d вводити з консолі.

42_{10} = (4 * 10^1) + (2 * 10^0)

101010_2 = (1 * 2^5)+(0 * 2^4)+(1 * 2^3)+(0 * 2^2)+(1 * 2^1)+(0 * 2^0)

1120_3 = (1 * 3^3) + (1 * 3^2) + (2 * 3^1) + (0 * 3^0)

9. Перевірити дійсність ISBN-10, яке використовується для ідентифікації книжок. Формат ISBN-10 складається з 9 цифр (від 0 до 9) плюс один контрольний символ (лише цифра або X). Якщо перевірочним символом є X, це означає значення «10». Цифри можуть записуватися з дефісами або без них, і їх можна перевірити на дійсність за такою формулою:

$(d_1 * 10 + d_2 * 9 + d_3 * 8 + d_4 * 7 + d_5 * 6 + d_6 * 5 + d_7 * 4 + d_8 * 3 + d_9 * 2 + d_{10} * 1) \bmod 11 == 0$

Якщо результат 0, це дійсний ISBN-10, інакше він недійсний.

10. Написати програму, яка реалізує обмін ключами Діффі-Хеллмана. Іван та Петро обмінюються ключами Діффі-Хеллмана, щоб поділитися секретами. Вони починають із простих чисел, вибирають приватні ключі, генерують і діляться відкритими ключами, а потім генерують спільний секретний ключ.

Крок 0.

Програма надає прості числа p і g.

Крок 1.

Іван вибирає закритий ключ a, більший за 1 і менший за p.

Петро робить те саме, щоб вибрати закритий ключ b.

Крок 2.

Іван обчислює відкритий ключ A.

$A = g^a \bmod p$

Використовуючи ті самі p і g, Петро аналогічно обчислює відкритий ключ B зі свого закритого ключа b.

Крок 3.

Іван і Петро обмінюються відкритими ключами. Іван обчислює секретний ключ s.

$s = B^a \bmod p$

Петро обчислює

$s = A^b \bmod p$

Розрахунки дають той самий результат! Тепер Іван і Петро діляться секретами.

11. Написати програму, яка перевіряє гіпотезу Коллатца. Гіпотезу Коллатца або проблему 3x+1 можна підсумувати таким чином: взяти будь-яке натуральне число n. Якщо n парне, поділити n/2, якщо непарне, помножте n на 3 і додайте 1, щоб отримати 3n + 1. Повторюйте процес нескінченно. Гіпотеза стверджує, що незалежно від того, з якого числа почати, завжди досягається 1.

Для заданого число n, введеного з консолі, визначити кількість кроків, необхідних для досягнення 1.

n = 12

1. 12
2. 6
3. 3
4. 10
5. 5
6. 16
7. 8
8. 4
9. 2
10. 1

12. Піфагорова трійка — це набір із трьох натуральних чисел {a, b, c}, для яких

$a^2 + b^2 = c^2$

$a < b < c$

Наприклад:

$3^2 + 4^2 = 5^2$.

Написати програму, яка для заданого цілого числа N, введеного з консолі, знаходить всі трійки Піфагора, для яких

$a + b + c = N$.

Наприклад:

$a + b + c = 1000$: {200, 375, 425}

13. Реалізувати бібліотеку функцій з основними операціями зі списком (без використання існуючих):

append (за наявності двох списків, додати всі елементи з другого списку в кінець першого списку);

concatenate (за наявності серії списків об’єднати всі елементи в усіх списках в один зведений список);

filter (за наявності предикату та списку повертає список усіх елементів, для яких предикат(елемент) має значення True);

length (за наявності списку повертається загальна кількість елементів у ньому);

map (за наявності функції та списку повертає список результатів застосування функції(елемента) до всіх елементів);

foldl (враховуючи функцію, список і початковий накопичувач, згорнути (зменшити) кожен елемент у накопичувач зліва);

foldr (враховуючи функцію, список і початковий накопичувач, згорнути (зменшити) кожен елемент до накопичувача справа);

reverse (за наявності списку повертається список із усіма вихідними елементами, але у зворотному порядку).

Примітка. Порядок, у якому аргументи передаються до функцій згортання (foldl, foldr), є суттєвим.

Це завдання передбачає використання змінних *nameref*. Для цього потрібна версія bash принаймні 4.0. Bash Namerefs — це спосіб передачі змінних функції за посиланням. Таким чином, змінну можна змінити у функції, а оновлене значення стане доступним у області виклику, наприклад:

```
prependElements() {
local -n __array=$1
shift
__array=( "$@" "${__array[@]}" )
}

my_array=( a b c )
echo "before: ${my_array[*]}"    # => before: a b c

prependElements my_array d e f
echo "after: ${my_array[*]}"     # => after: d e f a b c
```

14. Написати програму, яка визначає чи ціле число є ідеальним, недостатнім чи надлишковим.

Для ідеального числа сума множників, не включаючи число, дорівнює самому числу:

6: (1+2+3)=6 == 6

Для недостатнього числа числа сума множників, не включаючи число, менша самого числа:

8: (1+2+4)=7 < 8

Для надлишкового числа числа сума множників, не включаючи число, більша самого числа:

12: (1+2+3+4+6)=16 > 12

15. Замінити у сценарії `18.sh` команду `sleep` командою `suspend`, що спричинить його зупинку. Запустити сценарій на виконання `./18.sh`. Показати його стан в іншій консолі командою `ps au`. Записати у каталог `/home/user/ftp/incoming` нові файли. Командою `kill -SIGCONT` продовжити роботу сценарію. Показати, як реагує сценарій на появу нових файлів.

16. Модифікувати сценарій `18.sh` так щоб при добавленні або вилученні у поточний каталог файлів записувалася про це інформація у файл `change.log`.

Примітка. Номер варіанту завдання вибирається за порядковим номером студента у журналі групи.

Звіт з лабораторної роботи має містити:

- назву групи, П.І.Б. студента, завдання до роботи;
- короткий опис теоретичної частини;
- текст програми із коментаріями;
- роздрук екранів із результатом запуску програми.

Приклади

1. Використання функцій в сценаріях

```
# 01.sh
#!/bin/bash
function func1 {
  echo "Функція func1"
}
count=1
while [ $count -le 5 ]
do
  func1
  count=$((count+1))
done
echo "Кінець циклу"
func1
echo "Кінець сценарію"

$ ./01.sh
Функція func1
Функція func1
Функція func1
Функція func1
Функція func1
Кінець циклу
Функція func1
Кінець сценарію
```

2. Розміщення функції в середині сценарію

```
# 02.sh
#!/bin/bash
count=1
echo "Рядок перед визначенням функції"
function func1 {
  echo "Функція 1"
}
while [ $count -le 5 ]
do
  func1
  count=$((count+1))
done
echo "Кінець циклу"
func2
echo "Кінець сценарію"
function func2 {
  echo "Функція 2"
}
```

```
$ ./02.sh
Рядок перед визначенням функції
Функція 1
Функція 1
Функція 1
Функція 1
Функція 1
Кінець циклу
$ ./02.sh: line 14: func2: command not found
Кінець сценарію
```

3. Дублювання імені функції

```
# 03.sh
#!/bin/bash
function func1 {
  echo "Перше визначення функції func1"
}
func1
function func1 {
  echo "Повторне визначення функції func1"
}
func1
echo "Кінець сценарію"
0
$ ./3.sh
Перше визначення функції func1
Повторне визначення функції func1
Кінець сценарію
```

4. Тестування exit статусу функції

```
# 04.sh
#!/bin/bash
func1() {
  echo "спроба звернення до неіснуючого файлу"
  ls -l badfile
}
echo "тестування функції:"
func1
echo "exit статус: $?"

$ ./04.sh
тестування функції:
спроба звернення до неіснуючого файлу
ls: cannot access 'badfile': No such file or directory
exit статус: 2
```

5. Використання команди return у функції

```
# 05.sh
#!/bin/bash
function func1 {
  read -p "Ввести значення (не більше 255): " value
  echo "значення в функції" $value
  return $value
}
func1
```

```
echo "значення повернене командою return $?"

$ ./05.sh
Ввести значення (не більше 255): 122
значення в функції 122
значення повернене командою return 122
```

6. Використання echo для повернення значення

```
# 06.sh
#!/bin/bash
function func1 {
  read -p "Введіть значення: " value
  echo $value
}
result=$(func1)
echo "Повернене значення $result"

$ ./6.sh
Введіть значення: 333
Повернене значення 333
```

7. Використання echo для повернення значення

```
# 07.sh
#!/bin/bash
Lines_in_file() {
  cat $1 | wc -l
}
num_lines=$( lines_in_file $1 )
echo "Файл $1 має $num_lines рядків"

$ ./07.sh 06.sh
Файл 06.sh має 8 рядків
```

8. Доступ до параметрів сценарію всередині функції

```
# 08.sh
#!/bin/bash
# $# - кількість параметрів, які передаються у функцію
function func7 {
   echo $[ $1 * $2 ]
}

if [ $# -eq 2 ]
then
   value=`func7 $1 $2`
   echo "Результат множення $value"
else
   echo "Виконання: ./7.sh <число1> <число2>"
fi

$ ./08.sh 3 4
Результат множення 12
```

9. Використання глобальних змінних для передачі значень

```
# 09.sh
#!/bin/bash
```

```
function f1 {
  value=$((value*2))
}
read -p "Введіть число: " value
f1
echo "Результат: $value"

$ ./09.sh
Введіть число: 5
Результат: 10
```

10. Використання локальних змінних

```
# 10.sh
#!/bin/bash
function func1 {
   local temp=$((value+5))
   echo "значення у функції temp=$temp"
   result=$((temp*2))
}

temp=4
echo "значення у сценарії temp=$temp"
value=6
func1
echo "Результат $result"
if [ $temp -gt $value ]
then
   echo "temp є більше"
else
   echo "temp є менше"
fi

$ ./10.sh
значення у сценарії temp=4
значення у функції temp=11
Результат 22
temp є менше
```

11. Передача вектора змінних у функцію

```
# 11.sh
#!/bin/bash
# передача вектора змінних у функцію
function testit {
  local newarray
  newarray=($(echo "$*"))
  for index in ${!newarray[*]}
  do
      newarray[$index]=$[ ${newarray[$index]}*2 ]
  done

  echo "Новий вектор змінних: ${newarray[*]}"
}

myarray=(1 2 3 4 5)
echo "Початковий вектор змінних ${myarray[*]}"
```

```
testit ${myarray[*]}

$ ./11
```
Початковий вектор змінних 1 2 3 4 5
Новий вектор змінних: 2 4 6 8 10

12. Присвоєння значень вектору змінних у функції

```
# 12.sh
#!/bin/bash
function addarray {
  local sum=0
  local newarray
  newarray=($(echo "$@"))
for value in ${newarray[*]}
do
   sum=$((sum+value))
done
   echo $sum
}

myarray=(1 2 3 4 5)
echo "Початковий вектор: ${myarray[*]}"
arg1=$(echo ${myarray[*]})
result=$(addarray $arg1)
echo "Результат $result"

$ ./12.sh
```
Початковий вектор: 1 2 3 4 5
Результат 15

13. Повернення вектора значень із функції

```
# 13.sh
#!/bin/bash
function arraydblr {
  local origarray
  local newarray
  local elements
  local i
  origarray=($(echo "$@"))
  newarray=($(echo "$@"))
  elements=$(($#-1))
  for (( i = 0; i <= $elements; i++ ))
  {
     newarray[$i]=$((${origarray[$i]} * 2))
  }
  echo ${newarray[*]}
}

myarray=(1 2 3 4 5)
echo "Початковий вектор: ${myarray[*]}"
arg1=$(echo ${myarray[*]})
result=($(arraydblr $arg1))
echo "Новий вектор: ${result[*]}"

$ ./13
```

```
Початковий вектор: 1 2 3 4 5
Новий вектор: 2 4 6 8 10
```

14. Використання рекурсії при обчисленні факторіалу, повернення значення - echo

```
# 14.sh
#!/bin/bash
function factorial {
if [ $1 -eq 1 ]
then
  echo 1
else
  local temp=$[ $1 - 1 ]
  local result=$(factorial $temp)
  echo $((result*$1))
fi
}

read -p "Введіть значення: " value
if [ $value -gt 25 ]
then
   # для 26! переповнення
   echo "$value > 25"
fi
result=$(factorial $value)
echo "Факторіал від $value : $result"

$ ./14.sh
Введіть значення: 6
Факторіал від 6 : 720
```

15. Використання рекурсії при обчисленні факторіалу, повернення значення - return

```
# 15.sh
#!/bin/bash
# bash обчислює факторіал коректно тільки для 6!
#+ так як ret повертає 8-розрядне значення

# переповнення стеку залежить від розміру пам'яті
MAX_ARG=5  # Segmentation fault (core dumped)

E_WRONG_ARGS=65
E_RANGE_ERR=66

if [ -z "$1" ]
then
  echo "Запуск: `basename $0` число"
  exit $E_WRONG_ARGS
fi

if [ "$1" -gt $MAX_ARG ]
then
  echo "Максимальне значення $MAX_ARG"
  exit $E_RANGE_ERR
fi

fact ()
```

```
{
  local number=$1
  #  змінна "number" має бути оголошена як local,
  #+ інакше рекурсія не працюватиме
  if [ "$number" -eq 0 ]
  then
    factorial=1    # Факторіал 0 = 1
  else
    let "decrnum = number - 1"
    fact $decrnum  # рекурсивний виклик функції.
    let "factorial = $number * $?"
  fi

  return $factorial
}

fact $1
echo "Факторіал $1 is $?"

exit 0

$ ./15.sh
Факторіал 5 is 120
```

16. Використання функцій визначених у бібліотечному файлі поточного каталогу

```
# Створити файл myfuncs з наступними функціями addem, multem, divem:

function addem {
 echo $(($1 + $2))
}

function multem {
 echo $(( $1 * $2 ))
}
function divem {
 echo $(( $1 / $2 ))
}

# створити сценарій:
# 16.sh
#!/bin/bash
. ./myfuncs
value1=10
value2=5
result1=$(addem $value1 $value2)
result2=$(multem $value1 $value2)
result3=$(divem $value1 $value2)
echo "Результат додавання: $result1"
echo "Результат множення: $result2"
echo "Результат ділення: $result3"

$ ./16.sh
Результат додавання: 15
Результат множення: 50
Результат ділення: 2
```

17. Добавлення функцій у файл /home/user/.bashrc

```
# .bashrc
# ...
# Source global definitions
if [ -r /etc/bashrc ]; then
. /etc/bashrc
fi

function addem {
 echo $(($1 + $2))
}

function multem {
 echo $(( $1 * $2 ))
}
function divem {
 echo $(( $1 / $2 ))
}
```

 # тепер функції будуть доступні сценаріям з різних каталогів

```
# 17.sh
#!/bin/bash
value1=10
value2=5
result1=$(addem $value1 $value2)
result2=$(multem $value1 $value2)
result3=$(divem $value1 $value2)
echo "Результат додавання: $result1"
echo "Результат множення: $result2"
echo "Результат ділення: $result3"

$ ./17.sh
```

Результат додавання: 15
Результат множення: 50
Результат ділення: 2

18. Програма демон з опитуванням каталогів ftp

```
# 18.sh - демон використовує опитування для виявлення нових файлів
#!/bin/bash
shopt -s -o nounset
declare -rx SCRIPT=${0##*/}
declare -rx INCOMING_FTP_DIR="/home/internet/ftp/incoming"
declare -rx PROCESSING_DIR="/home/internet/ftp/processing"
declare -rx stat="/usr/bin/stat"
declare FILE
declare FILES
declare NEW_FILE
printf "$SCRIPT стартував у %s\n" "$(date)"
# перевірка наявності
if test ! -r "$INCOMING_FTP_DIR" ; then
  printf "%s\n" "$SCRIPT:$LINENO: помилка каталогу incoming - aborted" >&1
  exit 1
fi
if test ! -r "$PROCESSING_DIR" ; then
```

```
  printf "%s\n" "$SCRIPT:$LINENO: помилка каталогу processing - aborted" >&1
  exit 1
fi
if test ! -r "$stat" ; then
   printf "%s\n" "$SCRIPT:$LINENO: неможливо знайти або виконати $stat - aborted"
>&1
  exit 1
fi
# опитування нових FTP файлів
cd $INCOMING_FTP_DIR
while true; do
# Перевірка на незмінність нових файлів кожні 30 сек
FILES=$(find . -type f -mmin +30 -print)
echo "FILES=$FILES"
# Якщо нові файли існують перемістити їх у каталог processing
if [ ! -z "$FILES" ] ; then
    printf "$SCRIPT: нові файли поступили в %s\n" "$(date)"
    printf "%s\n" "$FILES" | {
    while read FILE ; do
        # Remove leading ”./”
        FILE="${FILE##*/}"
        echo "file=$FILE"
        # Перейменувати файли з поточним розміром і датою
        NEW_FILE=$($stat -c "%n %s %x" "$FILE")
        if [ -z "$NEW_FILE" ] ; then
            printf "%s\n" "$SCRIPT:$LINENO: помилка stat при створенні атрибутів"
        else
            # Переміщення файлів у каталог processing
            printf "%s\n" "$SCRIPT: переслано $FILE у $PROCESSING_DIR/$NEW_FILE"
            mv "$FILE" "$PROCESSING_DIR/$NEW_FILE"
        fi
    done
    }
fi
sleep 30
done

printf "$SCRIPT закінчився неочікувано в %s\n" "$(date)"
exit 1

$./18.sh
18.sh стартував у Thu Nov  1 23:12:22 EET 2019
FILES=./14.sh
./10.sh
./11.sh
./12.sh
./13.sh
17.sh: нові файли поступили в Thu Nov  1 23:12:22 EET 2019
file=14.sh
17.sh: переслано 14.sh у /home/internet/ftp/processing/14.sh 1060 2019-11-01
23:09:50.047997364 +0200
file=10.sh
17.sh: переслано 10.sh у /home/internet/ftp/processing/10.sh 537 2019-11-01
22:34:24.000000000 +0200
file=11.sh
```

```
17.sh: переслано 11.sh у /home/internet/ftp/processing/11.sh 410 2019-11-01
21:22:31.000000000 +0200
file=12.sh
17.sh: переслано 12.sh у /home/internet/ftp/processing/12.sh 555 2019-11-01
23:05:46.000000000 +0200
file=13.sh
17.sh: переслано 13.sh у /home/internet/ftp/processing/13.sh 461 2019-11-01
23:05:46.000000000 +0200
```

Демон можна запустити так, щоб він виконувався у фоновому режимі і автоматично перезапускав себе. При старті демона запускається два процеси. Зовнішній процес перенаправляє стандартні потоки входу, виходу і помилок у пристрій `/dev/null`. Внутрішній процес виконує сценарій у циклі `while`. Виконання сценарію можна побачити за допомогою команд `ps`. Можна опустити одну з команд групування використавши команду `exec` для закриття стандартних потоків вводу, виводу і помилок перед циклом `while`.

```
${ { while true ; do nohup bash polling.sh ; done ; } \
>/dev/null 2>&1 </dev/null & } &
```

Лабораторна робота 9

Текстові і графічні списки вибору

Мета роботи: навчитися створювати тестові і графічні списки вибору, керувати кольорами тексту.

Теоретичний матеріал лекції, технічна література, ресурси інтернету.

1. Короткі теоретичні відомості

Перед створенням текстових списків вибору (меню) потрібно очистити екран командою `clear`. За замовчуванням команда `echo` може виводити тільки друковані символи. При створенні списків вибору використовуються і недруковані керуючі символи, наприклад `"\n", "\t"`. Включити такі символи у команду `echo` дозволяє опція `-e`. З використання декількох команд `echo` можна створити список вибору. Остання команда `echo` містить опцію `-en`, яка пропускає символ нового рядка `"\n"`. Це дозволяє наступній команді зчитати дані з поточного рядка. Для зчитування даних з однієї позиції без натискання клавіші `Enter` використовується команда `read -n 1`. Послідовність списку для вибору можна оформити як одну функцію `function menu`.

При виборі кожного пункту із списку має викликатися відповідна функція, наприклад `diskspace, whoseon, memusage`. Логіку такого вибору можна організувати на основі команди `case`. Для забезпечення повторюваності виборів команда `case` поміщається у нескінчений цикл `while [ 1 ] do … done`.

1.2. Використання команди select

Команда `select` дозволяє створити список вибору з одного командного рядка

```
select variable in list
do
  commands
done
```

Параметр `list` є список стрічок меню розділених символом пропуску. Команда `select` висвічує кожний елемент списку, як перенумеровані опції. В кінці списку виводиться текст, заданий у змінній середовища PS3, наприклад PS3=”Виберіть опцію”. Так як команда `select` зберігає в елементах списку стрічки, то і у команді `case` потрібно використовувати стрічки.

Команда `select` дозволяє створювати меню в одному рядку і автоматично вводити і обробляти відповіді. Формат команди `select`:

```
select variable in ”choise 1” ”choise 2” ”choise 3”
do
  commands
done
```

де після `in` елементи меню відокремлені символами пропуску, `PS3` – змінна середовища, значення якої виводиться у кінці перенумерованого списку елементів меню.

Приклад команди `select` для створення меню:

```
#!/bin/bash

# using select in the menu
```

```
function diskspace {
  clear
  df -k
}

function whoseon {
  clear
  who
}

function memusage {
  clear
  cat /proc/meminfo
}
PS3=”Enter option: ”
select option in ”Display disk space” ”Display logged on users”
”Display memory usage” ”Exit program”
do
   case $option in
   ”Exit program”)
       break ;;
   ”Display disk space”)
       diskspace ;;
   ”Display logged on users”)
       whoseon ;;
   ”Display memory usage”)
       memusage ;;
   *)
       clear
       echo ”Sorry, wrong selection” ;;
  esac
done
clear
```

При виконанні сценарію появиться наступне меню:

```
$ ./smenu1
1) Display disk space    3) Display memory usage on users
2) Display logged        4) Exit program
Enter option:
```

1.3. Керування кольорами із сценаріїв

Більшість програм емуляторів терміналів розпізнають керуючі ANSI `Esc`-символи. ANSI `Esc`-символи починаються з керуючої послідовності CSI (control sequence introducer), яка вказує, що за нею розміщуються параметри, які встановлюють режим відображення дисплею та задають колір тексту і фону.

Код CSI складається з послідовності двох символів: Esc-символу (**`^[`**) і символу **`[`**. Esc-символ у більшості редакторів використовується для інших потреб. Тому для генерування Esc-символу в консолі або таких редакторах використовується послідовність натискання клавіш `Ctrl-v`, а потім `Esc`. В результаті появиться символ **`^[`** до якого потрібно додати **`[`**, що дає **`^[[`**.

У командах `echo -e` і `printf` Esc-символи **`^[[`** можна задати послідовністю символів **`` `\e[` ``** або **`` `\033[` ``**.

Після керуючої послідовності CSI задаються параметри SGR (Select Graphic Rendition), які керують відображенням дисплею. Синтаксис параметрів SGR:

```
CSI n[;k]m
```

`n` - параметри, які визначають коди кольорів;

`k` - параметри, які визначають режими відображення;

`m` – признак кінця SGR параметрів.

Значення кодів кольорів показані в табл. 1.

Таблиця 1 – Коди ANSI кольорів

Код	Описання
0	Чорний
1	Червоний
2	Зелений
3	Жовтий
4	Голубий
5	Бордовий
6	Бірюзовий
7	Білий
8	Зміна кольору

При заданні кольору тексту і фону використовують подвійні цифри. Для тексту вказується перша цифра 3, а для фону – 4. Друга цифра задає код кольору.

Приклади:

CSI31m – червоний текст

CSI47m – білий фон

Можна об’єднати колір тексту і фону:

CSI31;47m – текст червоний, фон – білий.

```
$echo -e '\e[31m Це червоний текст на чорному фоні \e[0m'
$echo -e '\e[31;47m Це червоний текст на білому фоні \e[0m'
```

Зміна RGB кольору:

Встановити 8-бітовий RGB колір тексту (код 5) Red=15; Green=194; Blue=40:

```
echo -e "\n\e[38;5;35;194;40m RGB 8-біт \e[0m встановлено! \n"
```

Встановити 8-бітовий RGB колір фону (код 5) Red=15; Green=194; Blue=40:

```
echo -e "\n\e[48;5;35;194;40m RGB 8-біт \e[0m встановлено! \n"
```

Встановити 24-бітовий RGB колір тексту (код 2) Red=15; Green=194; Blue=40:

```
echo -e "\n\e[38;2;35;194;40m RGB 8-біт \e[0m встановлено! \n"
```

Встановити 24-бітовий RGB колір фону (код 2) Red=15; Green=194; Blue=40:

```
echo -e "\n\e[48;2;35;194;40m RGB 8-біт \e[0m встановлено! \n"
```

Значення кодів режимів відображення показані в табл. 2.

Таблиця 2 – Коди режимів відображення

Парам.	Описання
0	Скинути у нормальний режим
1	Встановити інтенсивність `bold`
2	Встановити інтенсивність `faint`
3	Використати шрифт `italic`
4	Використати одиночне підкреслення
5	Встановити повільне блимання
6	Встановити швидке блимання
7	Інвертувати кольори тексту/фону
8	Встановити колір основного тексту у колір фону

```
echo -e '\n\e[31;1m Червоний жирний \e[0m \n'
echo -e "\n\e[31;4m Червоний підкреслений \e[0m \n"
echo -e "\n\e[31;47;5m Блимаючий червоний на білому фоні \e[0m \n"
```

Для зміни кольорів тексту і фону можна використати команду `tput`:

`tput setaf color` – задати колір тексту;

`tput setab color` – задати колір фону;

`tput smul` – режим підкреслення;

`tput usmul` – відмінити режим підкреслення;

`tput bold` – режим жирного тексту;

`tput dim` – режим зменшеної яскравості;

`tput sgr0` – скинути всі атрибути і режими.

```
tput setaf 1
tput setab 3
echo "Друк червоного тексту на жовтому фоні"
tput sgr0
```

Зміна кольору повідомлення командного рядка:

```
'\e[x;ym; $PS1 \e[0m'
```

де `x,y` - режими відображення.

Виведення PS1:

```
user@HP:~$ echo $PS1
```

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

Зміна кольору повідомлення з використанням команди `export`:

```
$ export PS1='\e[1;31m[\d \t \u@\h \w]\$ \e\0m'
$ export PS1=’\e[1;31m\u@h:\w\e[0m\$’
```

де `\e[1;31m` - встановлення яскравого червоного кольору тексту;

`\d` – дата;

`\t` - час;

`\u` – користувач;

`\h` – хост;

`\w` – повний шлях (`\W` – остання частина повного шляху до `/home/user`).

визначення режимів курсора:

```
CSI n q
```

n - код режиму курсора, табл. 3.

Таблиця 3 – Коди режимів курсора

Код.	Описання
1	Блимаючий блок
2	Нерухомий блок (за замовчуванням)
3	Блимаюче нижнє підкреслення
4	Нерухоме нижнє підкреслення
5	Блимаючий вертикальний прямокутник
6	Нерухомий вертикальний прямокутник

```
$ echo -e '\e[4 q'
```

1.4. Створення меню з використанням команди dialog

Для створення меню з використання графічних віджетів (діалогових графічних об'єктів) служить команда `dialog`. Команда запускається з командного рядка і викликає заданий віджет `dialog --widget parameters`.

Кожний діалоговий віджет підтримує виведення у двох формах:

- у стандартний потік помилок STDERR;
- з використанням статусного коду команди `exit`.

Статусний код команди `exit` визначає кнопка, яку натиснув користувач. Якщо натиснуто кнопки YES, OK, то команда `dialog` повертає 0, якщо кнопки CANCEL, NO – то 1. Для визначення, яка кнопка булла натиснута використовується стандартна змінна `$?`.

Довідку по стандартних віджетах поточної команди `dialog` можна отримати командою `man dialog` або `dialo --help`.

Формат команди: `dialog <загальні опції> <опції віджетів>`

Загальні опції дозволяють змінити вид і поведінку стандартних віджетів:

```
[--ascii-lines] [--aspect <ratio>] [--backtitle <backtitle>]
[--begin <y> <x>] [--cancel-label <str>] [--clear] [--colors]
[--column-separator <str>] [--cr-wrap] [--default-item <str>]
[--defaultno] [--exit-label <str>] [--extra-button]
[--extra-label <str>] [--help-button] [--help-label <str>]
[--help-status] [--ignore] [--input-fd <fd>] [--insecure]
[--item-help] [--keep-tite] [--keep-window] [--max-input <n>]
[--no-cancel] [--no-collapse] [--no-kill] [--no-label <str>]
[--no-lines] [--no-ok] [--no-shadow] [--nook] [--ok-label <str>]
[--output-fd <fd>] [--output-separator <str>] [--print-maxsize]
[--print-size] [--print-version] [--quoted] [--separate-output]
[--separate-widget <str>] [--shadow] [--single-quoted] [--size-err]
[--sleep <secs>] [--stderr] [--stdout] [--tab-correct] [--tab-len <n>]
[--timeout <secs>] [--title <title>] [--trace <file>] [--trim]
[--version] [--visit-items] [--yes-label <str>]
```

Опції віджетів:

Календар для вибору дати:

```
--calendar  <text> <height> <width> <day> <month> <year>
```

Вибір із багаторядкового списку:

```
--checklist  <text> <height> <width> <list height> <tag1> <item1> <status1>...
```

Вибір каталогу:

```
--dselect  <directory> <height> <width>
```

Редагування тексту з фалу:

```
--editbox      <file> <height> <width>
```

Форма з надписами і полями даних:

```
 --form  <text> <height> <width> <form height> <label1> <l_y1> <l_x1> <item1>
<i_y1> <i_x1> <flen1> <ilen1>...
```

Вікно для вибору файлів:

```
--fselect  <filepath> <height> <width>
```

Індикатор стану виконання (в процентах):

```
--gauge        <text> <height> <width> [<percent>]
```

Виведення інформаційного повідомлення:

```
--infobox  <text> <height> <width>
```

Форма для вводу тексту:

```
--inputbox  <text> <height> <width> [<init>]
```

Список вибору (меню) з можливостями редагування:

```
--inputmenu  <text> <height> <width> <menu height> <tag1> <item1>...
```

Список вибору

```
--menu  <text> <height> <width> <menu height> <tag1> <item1>...
```

Форма з надписами і полями даних різних типів (1 – невидимі, 2 – readonly):

```
 --mixedform  <text> <height> <width> <form height> <label1> <l_y1> <l_x1>
<item1> <i_y1> <i_x1> <flen1> <ilen1> <itype>...
```

Індикатор стану виконання (в процентах і значеннях):

```
--mixedgauge  <text> <height> <width> <percent> <tag1> <item1>...
```

Інформаційне повідомлення з кнопкою OK:

```
--msgbox  <text> <height> <width>
```

Поле для введення невидимого тексту:

```
--passwordbox  <text> <height> <width> [<init>]
```

Форма з надписами і скритими тестовими полями:

```
 --passwordform <text> <height> <width> <form height> <label1> <l_y1> <l_x1>
<item1> <i_y1> <i_x1> <flen1> <ilen1>...
```

Індикатор залишку затримки (в сек):

```
--pause        <text> <height> <width> <seconds>
```

Висвітлення тексту з потоку вводу у вікні:

```
--progressbox  <height> <width>   < file
```

Список вибору елементів радіокнопкою (одиничний вибір):

```
--radiolist  <text> <height> <width> <list height> <tag1> <item1> <status1>...
```

Висвітлення тексту з файлу у вікні з можливістю горизонтального переміщення:

```
--tailbox  <file> <height> <width>
```

Аналогічно до --tailbox, але функціонує у фоновому режимі:

```
--tailboxbg  <file> <height> <width>
```

Висвітлення вмісту файлу у вікні з прокруткою:

```
--textbox  <file> <height> <width>
```

Вікно для вибору годин, хвилин і секунд:

```
--timebox  <text> <height> <width> <hour> <minute> <second>
```

Вікно з кнопками YES, NO:

```
--yesno  <text> <height> <width>
```

Розміри вибираються автоматично при `height` і `width = 0` і встановлюються максимальними при `height` і `width = -1`. Глобальні розміри вибираються автоматично, якщо `menu_height/list_height = 0`.

Команда `dialog` може використовуватися у сценаріях з виконанням наступних вимог:

- перевірка `exit` статусу команди `dialog` на наявність кнопок YES або NO;
- перенаправлення стандартного потоку STDERR для отримання вихідного значення.

1.5. Створення меню з використанням команди dialog середовища KDE X Window

Графічне середовище KDE містить пакет `kdialog` для створення стандартних вікон. Пакет містить команду `kdialog`, яка дозволяє запускати різні типи віджетів із сценаріїв. Формат команди `kdialog`

```
kdialog [Qt-options] [KDE-options] [options] [arg]
```

```
Загальні опції:
  --help                    показати help про опції
  --help-qt                 показати опції специфічні для Qt
  --help-kde                показати опції специфічні для KDE
  --help-all                показати всі опції
  --author                  показати інформацію про автора
  -v, --version             показати інформацію про версію
  --license                 показати інформацію про ліцензію
  --                        кінець опцій

Опції:
  --yesno <text>            Вікно повідомлення з кнопками yes/no
  --yesnocancel <text>      Вікно повідомлення з кнопками yes/no/cancel
  --warningyesno <text>     Вікно попередження з кнопками yes/no
  --warningcontinuecancel <text> Вікно попередження з кнопками continue/cancel
  --warningyesnocancel <text> Вікно попередження з кнопками yes/no/cancel
  --yes-label <text>        Використати текст як позначку в кнопці Yes
  --no-label <text>         Використати текст як позначку в кнопці No
  --cancel-label <text>     Використати текст як позначку в кнопці Cancel
  --continue-label <text>   Використати текст як позначку в кнопці Continue
  --sorry <text>            Вікно повідомлення 'Sorry' (вибачте)
  --error <text>            Вікно повідомлення 'Error' (помилка)
  --msgbox <text>           Вікно діалогу Box
  --inputbox <text> <init>  Вікно діалогу Input Box
  --password <text>         Вікно діалогу Password
```

```
  --textbox <file> [width] [height] Вікно діалогу Text Box
  --textinputbox <text> <init> [width] [height] Вікно діалогу Text Input Box
  --combobox <text> item [item] [item] ... Діалог ComboBox
  --menu <text> [tag item] [tag item] ... Діалог Menu
  --checklist <text> [tag item status] ... Діалог Check List
  --radiolist <text> [tag item status] ... Діалог Radio List
  --passivepopup <text> <timeout> Пасивне випадаюче меню Passive Popup
  --getopenfilename [startDir] [filter] Діалог для відкриття існуючих файлів file
  --getsavefilename [startDir] [filter] Діалог для збереження файлів
  --getexistingdirectory [startDir] Діалог для вибору існуючих каталогів
  --getopenurl [startDir] [filter] Діалог для відкриття існуючих URL
  --getsaveurl [startDir] [filter] Діалог для збереження URL
  --geticon [group] [context] діалог вибору іконок
   --progressbar <text> [totalsteps] Діалог стану виконання, повертає посилання на
D-Bus для комунікації
  --getcolor                Діалог вибору кольору
  --title <text>            Діалог задання заголовку
   --default <text>           Точку входу (за замовчуванням) для використання для
combobox, menu і color
  --multiple                 дозвіл на  --getopenurl і --getopenfilename опції для
повернення множини файлів
   --separate-output           Повернення елементів списку в окремих рядках (для
checklist опції і файлу відкритого - опцією --multiple)
  --print-winid             Виведення winId для кожного діалогу
  --dontagain <file:entry>  Конфігураційний файл і ім’я опції для збереження стану
"do-not-show/ask-again"
    --slider <text> [minvalue] [maxvalue] [step] Діалог вікна повзунка, який
повертає вибране значення
  --calendar <text>         Діалог Calendar, який повертає вибрану дату
  --attach <winid>  Зробити діалог transient для застосування X заданого як winid

Аргументи:
  arg    Аргументи – залежать від основних опцій
```

Команда `kdialog` направляє своє виведення у стандартний потік STDOUT.

Контрольні запитання.

1. Яка послідовність створення тестових списків вибору (меню)?
2. Яка особливість створення тестових меню з використанням команд `while-case` і `select-case`?
3. Якими засобами можна керувати кольором текстових меню?
4. Як можна ввести керуючі послідовності символів CSI в консолі та сценаріях?
5. Формат і SGR параметри для керування режимом відображення дисплею і кольором тексту та фону?
6. Керуючі послідовності символів CSI для зміни рядка запрошення і форми курсору?
7. Як створити меню вибору з використанням команди `dialog`?
8. Які стандартні віджети можна створити командою `dialog`?
9. Як створити меню вибору з використанням команди `kdialog` середовища KDE X WINDOW?
10. Які стандартні віджети можна створити командою `kdialog`

2. Практична частина

Завдання.

1. Написати сценарій виведення текстового меню синього кольору на білому фоні з використанням команд `while` і `case`.

2. Написати сценарій виведення текстового меню жовтого кольору на чорному фоні з використанням команд `select` і `case`.

3. Написати сценарій виведення тестового меню червоного кольору на синьому фоні з використанням команд `while` і `case`.

4. Написати сценарій виведення тестового меню зеленого кольору на білому фоні з використанням команд `while`, `case` і `printf`.

5. Написати сценарій виведення тестового меню зеленого кольору на білому фоні з використанням команд `select` і `printf`.

6. Написати сценарій виведення тестового меню червоного кольору на синьому фоні з використанням команд `while, case` і `tput`.

7. Написати сценарій виведення тестового меню червоного кольору на синьому фоні з використанням команд `select` і `tput`.

8. Написати сценарій виведення тестового меню червоного кольору на синьому фоні з використанням команд `dialog`.

9. Написати сценарій виведення тестового меню червоного кольору на синьому фоні з використанням команд `kdialog`.

10. Написати сценарій з використанням пакету `dialog` середовища KDE для створення нового файлу за заданим іменем і атрибутами доступу `rwx`.

11. Написати сценарій з використанням пакету `dialog` середовища KDE для створення нового каталогу за заданим іменем і атрибутами доступу `rwx`.

12. Написати сценарій з використанням пакету `dialog` середовища KDE для вилучення існуючого файлу або каталогу.

13. Написати сценарій з використанням пакету `dialog` середовища KDE для створення архіву існуючого файлу або каталогу за заданим іменем.

14. Написати сценарій з використанням пакету `dialog` середовища KDE для створення меню з можливістю створення і вилучення заданих файлів і каталогів.

15. Написати сценарій з використанням пакету `dialog` середовища KDE для запуску на виконання вказаних Bash сценаріїв і висвітлення результату їх роботи.

Примітка. Номер варіанту завдання вибирається за порядковим номером студента у журналі групи.

Звіт з лабораторної роботи має містити:

- назву групи, П.І.Б. студента, завдання до роботи;
- короткий опис теоретичної частини;
- текст програми із коментаріями;
- роздрук екранів із результатом запуску програми.

Приклади

1. 1.sh - текстове меню з використанням команд while - case

```
function diskspace {
  clear
```

```
  df -k
}

function whoseon {
 clear
 who
}

function memusage {
 clear
 cat /proc/meminfo
}

function menu {
  clear
  echo
  echo -e "\t\t\tДовідка для  адміністратора\n"
  echo -e "\t1. Показати використання дисків"
  echo -e "\t2. Показати зареєстрованих користувачів"
  echo -e "\t3. Показати використання пам'яті"
  echo -e "\t0. Вийти\n\n"
  echo -en "\t\tВведіть опцію: "
  read -n 1 option
}

while [ 1 ]
do
 menu
 case $option in
 0)
      break ;;
 1)
     diskspace ;;
 2)
     whoseon ;;
 3)
     memusage ;;
 *)
    clear
    echo "Невірний вибір";;
 esac
 echo -en "\n\n\t\t\tНатисніть любу клавішу для продовження"
 read -n 1 line
done
clear
```

$ ./1.sh

Довідка для адміністратора

1. Показати використання дисків
2. Показати зареєстрованих користувачів
3. Показати використання пам'яті
0. Вийти

Виберіть пункт:

2. 2.sh - текстове меню з повідомленням жовтого кольору на синьому фоні

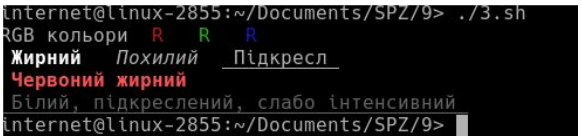
```
#!/bin/bash
# меню з кольором
function diskspace {
 clear
 df -k
}

function whoseon {
 who
}

function memusage {
 clear
 cat /proc/meminfo
}

function menu {
  clear
  echo
  echo -e "\t\t\tДовідка для адміністратора\n"
  echo -e "\t1. Показати дискову пам'ять"
  echo -e "\t2. Показати зареєстрованих користувачів"
  echo -e "\t3. Показати використання пам'яті"
  echo -e "\e[1m\t0. Вихід\n\n\e[0m\e[44;33m"
  echo -en "\t\tВведіть вибір: "
  read -n 1 option
}
echo "\e[44;33m"
while [ 1 ]
do
   menu
   case $option in
   0)
      break ;;
   1)
      diskspace ;;
   2)
      whoseon ;;
   3)
      memusage ;;
   *)
   clear
   echo -e "^[[5m\t\t\tНевірний вибір^[[0m^[[44;33m";;
   esac
   echo -en "\n\n\t\t\tНатисніть любу клавішу для продовження"
   read -n 1 line
done
echo "\e[0m"
clear
```

3. 3.sh – використання команди printf

```
#!/bin/bash
printf ”RGB кольори \e[31m R \e[0m \e[32m R \e[0m \e[34m R \e[0m \n”
```

```
printf ”\e[1m Жирний \e[0m \e[3m Похилий \e[0m \e[4m Підкресл \e[0m \n”
printf ”\e[31;1m Червоний жирний \e[0m \n”
printf ”\e[37;4;2m Білий, підкреслений, слабо інтенсивний \e[0m \n”
```

$ ./3.sh



4. Зміна кольору стрічки повідомлення командного рядка:

```
$ PS1=’\e[1;34m\u@h:\w\e[0m\$’
```

5. 5.sh Використання команди tput:

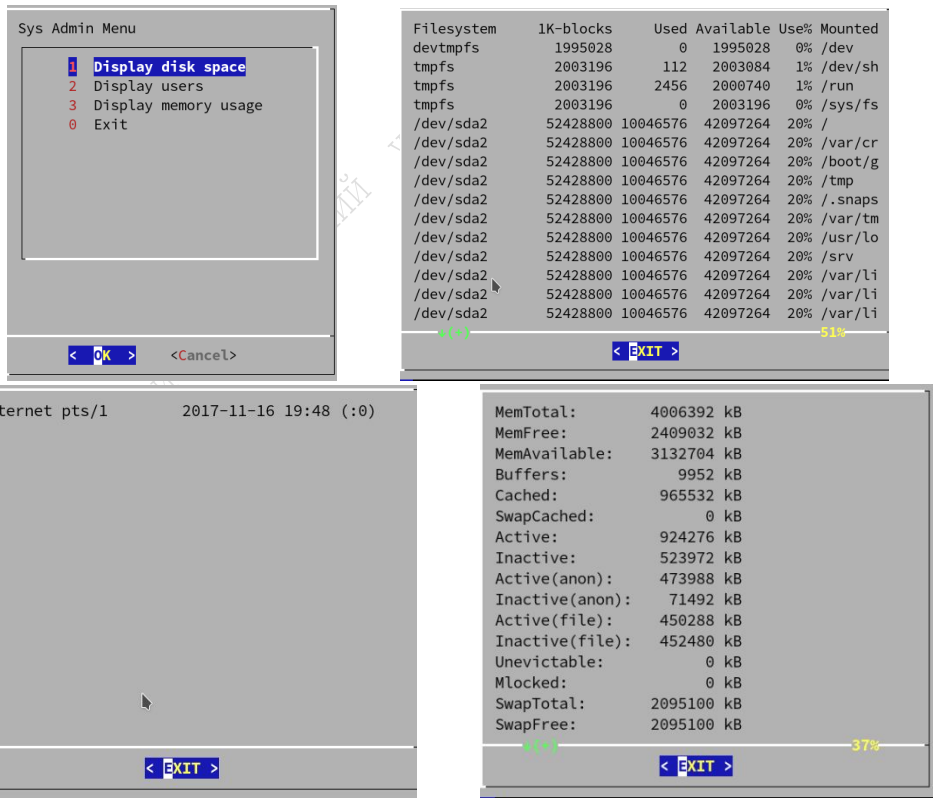
```
#!/bin/bash
tput setaf 1
echo ”Червоний”
tput setaf 2
echo ”Зелений”
tput setaf 4
echo ”Синій”
tput sgr0
```

6. 6.sh - використання команди dialog для створення графічних віджетів

```
#!/bin/bash
#temp= `mktemp -t test.XXXXXX`
#temp2=`mktemp -t test2.XXXXXX`

‘touch mypemp`
‘touch mytemp2`
temp=’mytemp’
temp2=’mytemp2’

function diskspace {
 df -k > $temp
 dialog --textbox $temp 20 60
}

function whoseon {
who > $temp
dialog --textbox $temp 20 50
}

function menusage {
 cat /proc/meminfo > $temp
 dialog --textbox $temp 20 50
}

while [ 1 ]
do

dialog --menu "Sys Admin Menu" 20 40 10 1 "Display disk space" 2 "Display users" 3
"Display memory usage" 0 "Exit"  2> $temp2

#dialog --menu "Довідка сисадміна" 20 40 1 "Дискова пам'ять" 2 "Зареєстровані
#користувачі" 3 "Оперативна пам'ять" 0 "Вихід" 2> $temp2
```

```
 if [ $? -ne 0 ]
   then
   break
 fi

 selection=`cat $temp2`

 case $selection in
  1)
   diskspace ;;
  2)
   whoseon ;;
  3)
   menusage ;;
  0)
   break ;;
 *)
   dialog --msgbox "Невірний вибір" 10 30
 esac
done
rm -f $temp  2> /dev/null
rm -f $temp2 2> /dev/null
```

$ ./6.sh

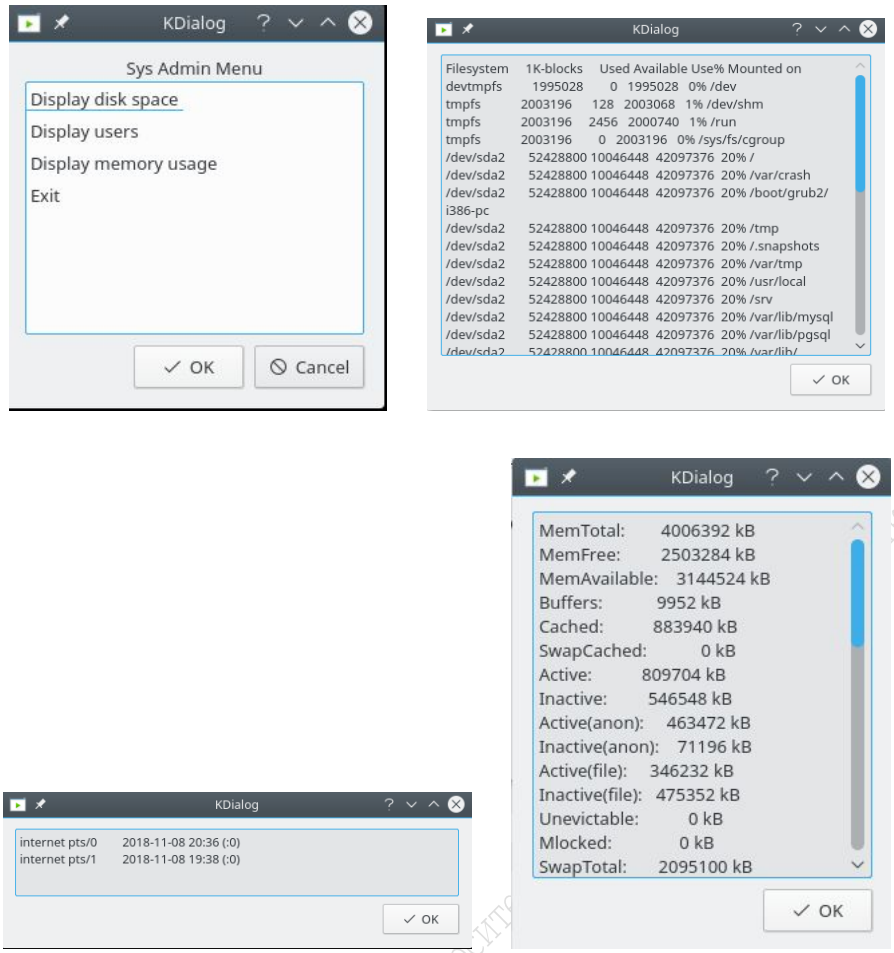


7. 7.sh - використання команди kdialog для створення віджетів у графічному середовищі KDE)

```
#!/bin/bash
# створення меню командою kdialog

temp=`mktemp -t temp.XXXXXX`
```

```
temp2=`mktemp -t temp2.XXXXXX`

function diskspace {
  df -k > $temp
  kdialog --textbox $temp 1000 10
}

function whoseon {
  who > $temp
  kdialog --textbox $temp 500 10
}

function memusage {
  cat /proc/meminfo > $temp
  kdialog --textbox $temp 300 500
}


while [ 1 ]
do
  kdialog --menu "Sys Admin Menu" "1" "Display disk space" "2" "Display users" "3"
"Display memory usage" "0" "Exit" > $temp2

if [ $? -eq 1 ]
  then
break

fi
selection=`cat $temp2`

case $selection in
 1)
   diskspace ;;
 2)
   whoseon ;;
 3)
   memusage ;;
 0)
 break ;;
 *)
 kdialog --msgbox "Sorry, invalid selection"
esac
done
```

$ ./7.sh



Лабораторна робота 10

Потокові редактори Sed, Gawk

Мета роботи: вивчити основи роботи з потоковими редакторами `sed` і `gawk`.

Теоретичний матеріал: лекції, технічна література, ресурси інтернету.

1. Короткі теоретичні відомості

Крім звичайний інтерактивних редакторів тексту існують і потокові редактори. Потокові редактори редагують текст на основі попередньо записаних правил. В Linux набули поширення два потокових редактори: `sed` (**s**`tream` **ed**`itor`) і `gawk` (GNU реалізація мови програмування `awk`).

1.2. Потоковий редактор `sed`

SED – це потоковий редактор тексту, за допомогою якого можна виконувати з файлами різні операції, наприклад пошуку та заміни, вставки чи видалення. При цьому найчастіше він використовується саме для пошуку та заміни.

Потоковий редактор `sed` обробляє дані у потоці даних на основі команд введених у командному рядку або у командному текстовому файлі. Він читає один рядок даних із

стандартного входу STDIN, порівнює дані із заданими командами, змінює дані згідно заданих команд і виводить новий рядок у STDOUT. Таким чином обробляються усі рядки даних і редактор завершує роботу.

Формат виклику потокового редактора sed:

```
sed options... [command...] datafile
sed [-n][-e] 'command' file(s)
sed [-n] -f cmdfile(s)
```

options – опції команди sed:

-e 'command' – команди задані в командному рядку (якщо більше одної), додаються до команд з файлу;

-f cmdfile – команди зчитуються з файлу;

-n – не виводить вихід кожної команди, а виводиться командою print.

[command ...] – задаються окремі команди, які застосовуються до одної адреси і вводяться у командному рядку з нового рядка.

datafile - файл, до якого застосовуються команди.

Sed не модифікує дані у вхідному файлі, а застосовує команди до даних вхідного потоку STDIN, який направляється на вивід. Це дозволяє направляти дані через канал на вхід редактору sed

```
$ echo "This is a test" | sed 's/test/big test/'
```

У прикладі, дані направляються через канал на вхід потокового редактора де до них застосовується команда s, яка замінює перший рядок другим test->big test.

Для запуску sed з читанням команд із файлу потрібно команди записати в один файл cmd1, а дані – у інший файл data1

```
$ cat cmd1
s/test/big test/

$ cat data1
This is a test

$ sed -f cmd1 data1
$ cat data1
This is a big test
```

Потоковий редактор sed має велику кількість команд і форматів ($ sed --help) для редагування даних.

1.2.1. Команда підставлення або заміни даних s (substitude):

[address]**s**/*pattern*/*replacement*/*flags*

підставляє замість шаблону *pattern* заміну *replacement*,

прапор *flags:*

N - число, яке вказує на кількість підставлень;

g – виконати всі підставлення з **g**-го входження;

p – друк вмісту файлу шаблона;

w file – вивести у файл результат підставлень.

1. Заміна входження входження учень на студент починаючи з 3-го входження:

```
$ cat data1
учень, учень, учень, учень, учень, учень
$ sed 's/учень/студент/3g' data1
$ cat dtat1
учень, учень, студент, студент, студент, студент
```

2. Дублювання змінених рядків починаючи з g-го входження:

```
$ cat data1
майстер, майстер
учень, учень
студент, студент
$ sed 's/учень/студент/gp' data1
$ cat dtat1
майстер, майстер
студент, студент
студент, студент
```

3. Замість розділювача стрічок "/" можна використовувати "!" (у випадку наявності в тексті символів /):

```
$ sed 's!/bin/bash!/bin/csh!' /etc/passwd
```

4. Підставлення, в 2-му рядку, в 2- і 3-му рядку, з 2-го рядка і до кінця тексту:

```
$ sed '2s/dog/cat/' data1
$ sed '2,3s/dog/cat/' data1
$ sed '2,$s/dog/cat/' data1
```

5. Два варіанти запису декількох команд у командному рядку:

```
$ sed -e 's/brown/green/; s/dog/cat/' data1
$ sed -e '
  >s/brown/green/
  >s/dog/cat/' data1
```

6. Використання шаблону *pattern* для фільтрування тексту. Команда підставлення буде застосована тільки до тих рядків, які містять текст шаблону */pattern*/command.

Команда буде застосована до тих рядків файлу, які містять слово rich:

```
$ sed '/rich/s/bash/csh/' /etc/passwd
```

7. Команди підставлення можна згрупувати, використовуючи фігурні дужки n,m{} і застосувати до рядків від n до m:

```
$ sed '2,7{
> s/fox/elephant/
> s/dog/cat/
> }' data1
```

8. Обгортання перших символів кожного слова дужками. Наступна команда виведе перший символ кожного слова обгорнутий дужками:

```
echo "Welcome To The Geek Stuff" | sed 's/\(\b[A-Z]\)/\(\1\)/g'
```

Результат:

```
(W)elcome (T)o (T)he (G)eek (S)tuff
```

1.2.2. Команда вилучення рядків

```
sed 'nd' datafile
```

n – номер рядка.

1. Рядки із заданими номерами n,m вилучаються командою n,m**d** (delete). Вилучення 2-го рядка, 2- і 3-го рядка, 3-го рядка і до кінця тексту ($):

```
sed '2d'   data2
sed '2,3d' data2
sed '3,$d' data2
```

2. Рядки із заданим шаблоном pattern вилучаються командою /pattern/**d**. Вилучення порожніх рядків:

```
sed /^$/d file > new_file
```

3. Вилучення рядків, які починаються або закінчуються на xxx:

```
sed '/^xxx/d; /xxx$/d' file > new_file
```

1.2.3. Команда вставлення рядків

Команда **i** (insert) вставляє новий рядок перед заданим рядком, а команда **a** (append) – після заданого рядка:

```
sed '[address]command\new line'  file

$ sed '3i\Цей рядок вставляється перед 3-м рядком' data1
$ sed '3a\Цей рядок вставляється після 3-го рядка' data1
```

1.2.4. Команда заміни всього рядка за номером або за шаблоном c (change):

```
sed '[address]c\new line'  file

$ sed '3c\Цей рядок замінить третій рядок' data1
$ sed '/abcd/c\Цей рядок замінить рядок за шаблоном' data1
```

1.2.5. Команда y (translate) для заданих рядків послідовно замінює символи з набору inchars у набір outchars

```
[address]y/inchars/outchars/
$ sed 'y/123/789/' data7
$ echo "ABCDEFGH" | sed 'y/ABCDEFGH/12345678/'
```

1.2.6. Виведення (друку) даних у консоль

Для виведення (друку) інформації з потоку даних використовуються наступні команди:
p – друк рядків тексту;
= – друк номерів рядків;
l – друк тексту і недрукованих (невидимих) символів ASCII як двоцифрових кодів.

1. Команда **p** друкує задані рядки (-n подавити вивід решти рядків):

```
$ echo "Тестовий рядок" | sed 'p'
$ sed -n '/number 3/p' data1
$ sed -n '2,3p' data1
$ sed -n '/3/{p s/line/test/p }' data1
```

2. Друк усіх рядків з номерами і друк рядків за шаблоном

```
$ sed '=' data1
$ sed -n '/number 4/{ = p }' data2
```

3. Друк рядків з недрукованими ASCII символами

```
$ sed -n 'l' data2
```

1.2.7. Команди запису `w` (write) і читання `r` (read) файлів

```
[address]w filename
[address]r filename
```

1. Записати у файл 1-й і 2-й рядок із вхідного потоку:

```
$ sed '1,2w test' data1
```

2. Прочитати дані з одного файлу і вставити їх в інший файл:

```
$ sed '3r data1' data2   # після 3-го рядка файлу data2 вставити файл data1
sed '/number 2/r data1' data2 # після рядків з шаблоном вставити файл data1
sed '$r data1' data2    # вставити файл data1 в кінці файлу data2

$ cat letter
Наступним працівникам:
LIST
надіслати повідомлення

$ cat data3
Іваненко І.І.
Петренко П.П.
Степаненко С.С

$ sed '/LIST/{
> r data3
> d               # вилучення шаблону LIST
> }' letter
```

Або в один рядок

```
$ sed /LIST/'r data' letter | sed /LIST/d

Наступним працівникам:
Іваненко І.І.
Петренко П.П.
Степаненко С.С
надіслати повідомлення
```


1.3. Розширений потоковий редактор `awk/gawk`

Розширений потоковий редактор `gawk` призначений для пошуку рядків (або стрічок) у файлах, які співпадають із заданим шаблоном, і виконання заданих дій із такими рядками. Редактор `gawk`, використовує для роботи з текстом не команди редактора, а мову програмування. Ця мова є орієнтована на дані, тобто спочатку необхідно вказати, які дані потрібно вибрати з файлу, а потім над знайденими даними виконати необхідні дії. Тому програма `gawk` складається з послідовності операторів (правил), які записуються в окремих рядках і мають вид:

```
шаблон  { дія }
шаблон  { дія }
…
```

Окремі випадки:

`шаблон` – коли виводяться рядки із заданим шаблоном;

`{ дія }` – коли дія виконується для всіх рядків. Дія може складатися із послідовності операторів розділених “;” або символом нового рядка (“\n”).

В `gawk` можливі дії:

- присвоєння виразів;
- оператори керування;
- оператори виведення;
- вбудовані функції.

Програму `gawk` можна запускати різними способами. Якщо програма коротка – її можна записати в командному рядку:

```
awk ’program’ input-file1 input-file2 ...
```

Якщо програма велика – її записують в окремий файл, з якого вона зчитується на виконання:

```
awk -f program-file input-file1 input-file2 ...
```

Програму `gawk` можна оформити як автономний сценарій:

```
# test1
#!/bin/gawk –f
BEGIN { print “Привіт від gawk” }
```

і запустити на виконання

```
$ chmod +x test1
$ ./test1
Привіт від gawk
```

Зарезервовані змінні qawk:

NR – номер поточного рядка.
NF – число полів в поточному рядку.
RS – розділювач рядків при введенні (за замовчуванням “\0”).
FS – розділювач полів при введенні (пропуск і/або табуляція).
ORS – розділювач рядків при виведенні (за замовчуванням RS).
ORS – розділювач полів при виведенні (за замовчуванням FS).
OFMT – формат виведення числа (“%.6g”).
FILENAME – ім’я вхідного файлу.
В рамках цієї мови програмування можна:

- визначити змінні для зберігання тексту (`$0, $1,…`);
- використати арифметичні і стрічкові операції для маніпулювання даними;
- писати структуровані програми;
- генерувати форматовані звіти з вхідного файлу у вихідний в іншому порядку і форматі.

Програма `gawk` використовує **внутрішні змінні** для роботи з полями даних одного рядка:

`$0` – містить увесь текст одного рядка;
`$1` – містить перше поле рядка тексту;
`$2` – містить друге поле рядка тексту;
…
`$n` – містить `n`-е поле рядка тексту;

Команда **print** використовується для виведення тексту. Вивід першого поля з кожного рядка тексту

```
$ cat data3 або qawk '{print}' data3
One line of test text.
Two lines of test text.
Three lines of test text.
$ gawk '{print $1}' data3
One
Two
Three
```

При читанні файлів можна використовувати різні розділювачі полів:

```
$ gawk -F: '{print $1}' /etc/passwd
at
avahi
bin
daemon
dnsmasq
ftp
ftpsecure
games
lp
```

В програмі можна використовувати змінні, вирази, команди:

```
$ echo "My name is Petro" | gawk '{$4="Ivan"; print $0}'
My name is Ivan
```

Читання програми з файлу:

```
$ cat script2
{ print $5 "'s userid is " $1 }
$ gawk -F: -f script2 /etc/passwd
Batch jobs daemon user id is at
User for Avavhi user id is avahi
bin user id is bin
Daemon user id is daemon
Dns dnsmasq user id is dnsmasq
FTP account user id is ftp
```

Якщо є декілька команд у сценарії, то команди починаються з нового рядка (без ;)

```
$ cat script3
{
text="'s userid is "
print $5 text $1
}
$ gawk -F: -f script3 /etc/passwd | more
root's userid is root
bin's userid is bin
```

Існує два спеціальних оператори **BEGIN { дія }** і **END** { дія }.

Виконання дій перед (BEGIN) і після (END) оброблення даних

```
$ echo ”1111” | gawk ‘BEGIN { print “Hello word!”} { print $0 }’
Hello word!
1111

$ echo ”1111” | gawk ’BEGIN {print "Hello World!"} {print $0} END {print
"byebye"}’
Hello word!
1111
byebye
```

Читання команд програми `gawk` з файлу:

```
$ cat script4
BEGIN {
print "The latest list of users and shells"
print " Userid Shell"
print "-------- -------"
FS=":"
}
{
print $1 " " $7
}
END {
print "This concludes the listing"
}

$ gawk -f script4 /etc/passwd
Userid Shell
------ -----
at /bin/bash
avahi /bin/false
daemon /bin/bash
...
This concludes the listing
```

1.3.1. Пошук за шаблоном

Програму `qawk` можна записати у файл, наприклад у `test1`:

```
# програма test1, яка знаходить лексеми integer, letter, blank line
/[0-9]+/    { print "That is an integer" }
/[A-Za-z]+/ { print "This is a string" }
/^$/        { print "This is a blank line." }
```

Приклад використання програми, яка обробляє вхід з консолі:

```
$ gawk -f test1
4
That is an integer
t
This is a string
4T
That is an integer
```

```
This is a string
RETURN
This is a blank line.
44
That is an integer
CTRL-D
$
```

1.3.2. Використання регулярних виразів

Регулярні вирази можна використовувати як шаблон між двома похилими. Наступний приклад друкує з файлу `file` 2-ге поле кожного запису, який містить слово `'Hello'`:

```
$ gawk '/Hello/ { print $2 }' file
```

Регулярні вирази можуть використовуватися у порівняннях. Для порівняння з регулярним виразом використовуються оператори `'~', '!~'`. Наступні приклади вибирають всі записи, у яких перше поле починається з букви *J*, не починається із букви *J*, містить цифри

```
$ gawk '$1 ~ /J/' file
$ gawk '$1 !~ /J/' file
$ gawk 'BEGIN {digit = "[0-9]+" } $1 ~ digit { print }' file
```

1.3.3. Виведення записів і полів даних

`Gawk` при введенні даних розбиває їх на записи і поля. Кількість прочитаних записів із вхідного файлу зберігається у вбудованій змінній FNR. Записи розділюються символом розділювачем, який задається у вбудованій змінній RS. Приклад розділення на записи з використанням символу розділювача ‘/’.

```
gawk 'BEGIN { RS = '/' } { print }' file
```

Записи розділюються на поля з використання символу пропуску. Символ розділювач полів задається у вбудованій змінній `FS`. Для посилання на поля використовуються змінні `$1`, `$2`, … . Змінна `$0` посилається на весь запис.

Значення полів можна корегувати

```
gawk '{ S1 = $1 - 10; $2 = $3 + $4 } { FS = ':'; print }' file
```

Для виведення даних використовується команда `print`:

```
gawk 'BEGIN { print "A     B"
              print "-------"}
            { print  $1, $2 } file
```

Кожна команда `print` створює вихідний запис. Розділювачем вихідних записів є вбудована змінна `ORS` (за замовчуванням `ORS='\n'`). Розділювачем вихідних полів є вбудована змінна `OFS`.

Для форматованого виведення значень полів використовується команда printf:

```
printf "format" var1, var2, …
```

1.3.4. Вирази, змінні і операції

Вирази є базовими будівельними блоками шаблонів і дій. Вирази будуються із констант, змінних і операцій над ними. Найпростішим типом виразів є константи: числа (всі числа в `gawk` є числами з *плаваючою крапкою*), стрічки, регулярні вирази. Для зберігання значень їх присвоюють змінним *variable = text*.

Перетворення типів стрічка-число виконується в контексті `gawk` програми.

```
two =2; three =3
print (two three) + 4
```

Результатом виконання програми буде 27.

Логічним типом є `true` і `false`. У `gawk` любі ненульові числові значення і непорожні стрічки мають значення `true`. Всі інші значення є `false`.

У виразах використовуються арифметичні і стрічкові операції. Арифметичні операції:

```
x ^ y, x ** y x в ступені y
-x – унарний мінус
+x – унарний плюс
x*y - множення
x/y – ділення без округлення ( ’3/4’ має значення 0.75 )
x%y – залишок від ділення.
x+y - додавання.
++x - інкремент
x-y - віднімання.
--x - декремент
```

Є тільки одна стрічкова операція – зчеплення. Для цього один вираз записується безпосередньо біля другого:

```
$ awk ’{ print "Field number one: " $1 }’ file
```

Операція порівняння:

```
x < y True якщо x менше y.
x <= y True якщо x менше або дорівнює y.
x > y True якщо x більше y.
x >= y True якщо x більше або дорівнює y.
x == y True якщо x рівне y.
x != y True якщо x не рівне y.
x ~ y True якщо x співпадає з regexp позначеним як y.
x !~ y True якщо x не співпадає з regexp позначеним як y.
subscript in array True якщо array має елемент subscript.
```

Конструкції галуження:

```
if (x % 2 == 0)
  print "x парне"
else
  print "x непарне"

$ echo 1e2 3 | awk ’{ print ($1 < $2) ? "true" : "false" }’
false

switch (NR * 2 + 1) {
case 3:
case "11":
    print NR - 1
    break
```

```
case /2[[:digit:]]+/:
    print NR

default:
    print NR + 1

case -1:
    print NR * -1
}
```

Конструкції циклів:

```
gawk '{
  i = 1
  while (i <= 3) {
    print $i
    i++
  }
}' file
```

```
gawk '{
  i = 1
  do {
    print $0
    i++
  } while (i <= 10)
}' file
```

```
awk '{
for (i = 1; i <= 3; i++)
    print $i
}' file


for (i in mas) { … }
```

Всередині конструкцій циклів можуть використовуватися інструкції `break`, `continue`, `next`, `nextfile`, `exit`.

1.3.5. Масиви

Масиви `gawk` є асоціативними, тобто елемент масиву складається з індексу і значення. Індексом може бути як число, так і стрічка.

```
Index 3 Value 30
Index 1 Value "foo"
Index 0 Value 8
Index 2 Value ""
```

```
Index "dog" Value "chien"
Index "cat" Value "chat"
Index "one" Value "un"
Index 1 Value "un"
```

Доступ до елементів масиву:

```
{
  if ($1 > max)
    max = $1
    arr[$1] = $0
}

END {
for (x = 1; x <= max; x++)
  if (x in arr)
    print arr[x]
}
```

```
{
  for (i = 1; i <= NF; i++)
  used[$i] = 1
}

END {
  for (x in used) {
     if (length(x) > 10) {
        ++num_long_words
        print x
     }
  }
  print num_long_words, " > 10 chars"
}
```

1.3.6. Функції

В gawk використовуються вбудовані і **визначені користувачем функції**. Вбудовані числові функції: `atan2(y, x), cos(x), exp(x), int(x), log(x), rand(), sin(x), sqrt(x), srand([x]), printf(fmt,…), substr(str,n,len), getline(), index(s1,sd2), split(str,M,FS)`

Функції для маніпуляції стрічками: `asort(source [, dest [, how]]), asorti(source [, dest [, how]]), gensub(regexp, replacement, how [, target]), gsub(regexp, replacement [, target]), index(in, find), length([string]), match(string, regexp [, array]), patsplit(string, array [, fieldpat [, seps]]) , split(string, array [, fieldsep [, seps]]), sprintf(format, expression1, ...), strtonum(str), sub(regexp, replacement [, target]), substr(string, start [, length]), tolower(string), toupper(string).`

Функції введення/виведення: `close(filename [, how]), fflush([filename]), system(command).`

Функції роботи з часом: `mktime(datespec), strftime([format [, timestamp [, utc-flag]]]), systime().`

Функції маніпуляції з бітами: `and(v1, v2 [, . . . ]), compl(val), lshift(val, count), or(v1, v2 [, . . . ]), rshift(val, count), xor(v1, v2 [, . . . ]).`

Функції визначені користувачем використовують ключове слово **`function`** :

```
function name([parameter-list])
{
   body-of-function
}
```

```
function myprint(num)
{
   printf "%6.3g\n", num
}
if ($3 > 0) { myprint($3) }
```

Контрольні запитання.

1. Яка різниця між інтерактивним і потоковим редактором?
2. Який формат виклику редактора `sed`?
3. Які особливості запуску команд редактора `sed` з командного рядка і з файлу?
4. Який формат команди підставлення даних у редакторі `sed`?
5. Який формат вилучення даних у редакторі `sed`?
6. Який формат вставлення рядків у редакторі `sed`?
7. Який формат команд заміни рядків і окремих символів у редакторі `sed`?
8. Які можливості і формат команд друку у редакторі `sed`?
9. Який формат команд запису і читання даних у файл у редакторі `sed`?
10. Який формат команд заміни символів з одного набору на інший у редакторі sed?
11. Який формат запуску на виконання програми `gawk`?
12. Як прочитати окремі поля рядків тексту командами програми `gawk`?
13. Які правила запису команд програми `gawk`?
14. Як виконати команди програми `gawk` до і після обробки даних?
15. Як виконати пошук даних за шаблоном ц програмі `gawk`?
16. Як виконати пошук даних з використання регулярних виразів у програмі `gawk`?
17. Як вивести записи і поля даних в `gawk`?
18. Вирази, змінні і операції `gawk`.
19. Масиви в `gawk`.
20. Функції в `gawk`.

2. Практична частина

Завдання.

1. Використати команди редактора `sed` для перетворення довільної фразу, введеної з консолі, в акронім, наприклад:

Рідко кристалічний дисплей => РКД.

2. Використовуючи команди редактора `sed` визначити чи задане речення є панграма (містить усі букви абетки). Речення вводити з консолі.

3. Використовуючи команди редактора `sed` порахувати кількість балів для довільної фрази, введеної з консолі.

Буква	Бал
A, E, I, O, U, L, N, R, S, T	1
D, G	2
B, C, M, P	3
F, H, V, W, Y	4
K	5
J, X	8
Q, Z	10

4. Використовуючи команди редактора `sed` зашифрувати текст з використання реверсованої абетки

Звичайна абетка: abcdefghijklmnopqrstuvwxyz

Реверсована абетка: zyxwvutsrqponmlkjihgfedcba

Зашифрований текст розбивається на групи по 5 символів.

Приклад

Кодування test дає gvhg

Кодування x123 дає c123b vh

Декодування gvhg дає test

Декодування gsvjf rxpyi ldmul cqfnk hlevi gsvoz abwlt дає
thequickbrownfoxjumpsoverthelazydog

5. Використовуючи команди редактора `gawk` реалізувати транспонування блоків довільного тексту. Блоки довільного тексту вводити з файлу.

ABC	AD
DEF	BE
	CF
AB	AD
DEF	BE
	F
ABCDE	AF
FGHKLM	BG
	CH
	DK
	EL
	M

6. Використовуючи команди редактора `gawk` зашифрувати і дешифрувати текст за допомогою шифру Цезаря із заданим ключем. Це простий шифр із зсувом всіх літер алфавіту на задану величину (ключ) від 0 до 26. Буква зсувається на стільки значень, яке значення ключа. Загальна нотація для ротаційних шифрів - ROT + <key>. Найбільш часто використовуваним ротаційним шифром є ROT13.

ROT13 над англійською абеткою дає:

Звичайний текст: abcdefghijklmnopqrstuvwxyz

Зашифрований текст: nopqrstuvwxyzabcdefghijklm

7. Використовуючи команди редактора `gawk` підрахувати кількість повторів кожної букви абетки у любому текстовому файлі.

8. Використовуючи команди редактора `gawk` реалізувати кодування та декодування для шифру огорожі. У шифрі огорожі повідомлення записується вниз на послідовних «планках» уявної огорожі, а потім рухається вгору, коли ми досягається дно (як зигзаг).

W . . . E . . . C . . . R . . . L . . . T . . . E

. E . R . D . S . O . E . E . F . E . A . O . C .

. . A . . . I . . . V . . . D . . . E . . . N . .

Нарешті повідомлення зчитується в рядках.

WECRLTEERDSOEEFEAOCAIVDEN

Щоб розшифрувати повідомлення, потрібно взяти зигзагоподібну форму та заповнити зашифрований текст уздовж рядків.

? . . . ? . . . ? . . . ? . . . ? . . . ? . . . ?

. ? . ? . ? . ? . ? . ? . ? . ? . ? . ? . ? .

. . ? . . . ? . . . ? . . . ? . . . ? . . . ? . .

Заповнюється перший рядок

W . . . E . . . C . . . R . . . L . . . T . . . E

. ? . ? . ? . ? . ? . ? . ? . ? . ? . ? . ? .

. . ? . . . ? . . . ? . . . ? . . . ? . . . ? . .

Заповнюється другий рядок

W . . . E . . . C . . . R . . . L . . . T . . . E

. E . R . D . S . O . E . E . F . E . A . O . C .

. . ? . . . ? . . . ? . . . ? . . . ? . . . ? . .

Заповнюється третій рядок

W . . . E . . . C . . . R . . . L . . . T . . . E

. E . R . D . S . O . E . E . F . E . A . O . C .

. . A . . . I . . . V . . . D . . . E . . . N . .

9. Використовуючи команди редактора `gawk` обчислити трикутник Паскаля до заданої кількості рядків. Кількість рядків вводити з консолі. У трикутнику Паскаля кожне число обчислюється додаванням чисел праворуч і ліворуч від поточної позиції в попередньому рядку.

```
   1
  1 1
 1 2 1
1 3 3 1
1 4 6 4 1
# ... etc
```

10. Використовуючи команди редактора `gawk` реалізувати кодування та декодування із змінної довжиною. Метою цього кодування є кодування цілих значень таким чином, щоб заощадити байти. Лише перші 7 бітів кожного байта є значущими (як ASCII байт). Отже, 32-бітове значення потрібно розпакувати у серію 7-бітових байтів. Звичайно, буде змінна кількість байтів залежно від цілого числа. Щоб вказати, який байт є останнім у ряді, біт 7 встановлюється в 0. У всіх попередніх байтах ви біт 7 встановлюється в 1. Отже, якщо ціле число в межах 0-127, воно може бути представлене одним байтом. Ось приклади отриманих цілих 32-розрядних чисел як значень і величин змінної довжини:

NUMBER	Змінна довжина
00000000	00
00000040	40
0000007F	7F
00000080	81 00
00002000	C0 00
00003FFF	FF 7F
00004000	81 80 00
00100000	C0 80 00
001FFFFF	FF FF 7F
00200000	81 80 80 00
08000000	C0 80 80 00
0FFFFFFF	FF FF FF 7F

11. Використовуючи команди редактора `gawk` написати програму, яка шифрує і розшифровує текстовий файл за алгоритмом Цезаря.

12. Використовуючи команди редактора `gawk` написати програму, яка шифрує і розшифровує текстовий файл за алгоритмом абетки в'язниці.

13. Іноді організм людини виробляє занадто багато певного білка, що може спричинити різного роду захворювання. Але можна створити дуже специфічну молекулу (називається мікро-РНК), це може перешкодити виробленню білка. Цей метод називається РНК-інтерференцією.

Визначити комплемент РНК даної послідовності ДНК. Ланцюги ДНК і РНК є послідовністю нуклеотидів.

Чотири нуклеотиди, знайдені в ДНК, це аденін (A), цитозин (C), гуанін (G) і тимін (T).

Чотири нуклеотиди, знайдені в РНК, це аденін (A), цитозин (C), гуанін (G) і урацил (U).

Дано ланцюг ДНК GGCTTCTTAGG, його транскрибований ланцюг РНК утворюється шляхом заміни кожного нуклеотиду своїм комплементом:

G -> C

C -> G

T -> A

A -> U

Використовуючи команди редактора `gawk` написати програму для перетворень довільного ланцюга ДНК у ланцюг РНК.

14. Використовуючи команди редактора `gawk` написати програму, яка для заданого файлу підраховує циклічний надлишковий код CRC-16.

15. Використовуючи команди редактора `gawk` написати програму, яка для заданого файду обчислює хеш код SHA-1.

Примітка. Номер варіанту завдання вибирається за порядковим номером студента у журналі групи.

Звіт з лабораторної роботи має містити:
- назву групи, П.І.Б. студента, завдання до роботи;
- короткий опис теоретичної частини;
- текст програми із коментаріями;
- роздрук екранів із результатом запуску програми.

Приклади

1. Підставлення слова у другому рядку

```
$ cat data1
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
$ sed '2s/dog/cat/' data1
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy cat.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
```

2. Декілька команд у командному рядку

```
$ sed -e 's/brown/green/; s/dog/cat/' data1
```

3. Підставлення слова у двох рядках

```
$ sed '2,3s/dog/cat/' data1
```

4. Підставлення слова від заданого рядка (2) і до кінця

```
$ sed '2,$s/dog/cat/' data1
```

5. Підставлення слів за шаблонами від 3-го рядка і до кінця

```
$ sed '3,$ {
 s/brown/green/
 s/lazy/active/
 }' data1
```

6. Вилучення рядків

```
# '2d' - другого рядка
# '2,3d' – другого, третього
# '2,$s' - від другого і до кінця
# '/1/,/3/d' - від першого до третього
```

7. Друк першого поля кожного рядка

```
$ cat data3
one line of test text
two line of test text
three line of test text
gawk '{print $1}' data3
one
two
```

```
three
```

8. Читання файлу з заданим розділювачем полів рядка і друк 1-го поля

```
$ gawk -F: '{print $1}' /etc/passwd
```

9. Запис підстановок у новий файл

```
# друкування тільки підстановок (-n подавити вивід інших рядків)
$  echo 'This is a test line' | sed -n 's/test/trial/p'
This is a trial line
# зберігання підстановок у новий файл
$   echo 'This is a test line' | sed -n 's/test/different/w tmp'; cat tmp; rm tmp
This is a different line.
```

10. Заміна символів у файлі

```
# заміна символів з набору 1->7, 2->8, 3->9
$ sed 'y/123/789/' data5
```

11. Нумерування рядків файлу

```
$ sed '=' data5
```

12. Друкування тексту і невидимих ASCI-символів файлу

```
$ sed -n 'l' data5
```

13. Вставлення даних з файлу data5 у файл data5a після 3-го рядка

```
$ sed '3r data5a' data5

This is line number 1
This is line number 2
This is line number 3
This is line number 4
This is line number 1 again
This is yet another line
This is the last line in the file
```

15. Введення команди підставлення s редактора sed з файлу

```
$ cat script
s/brown/green/
s/fox/elephant/
s/dog/cat/

# файл data1
# запуск на виконання
$ sed -e -f script1 data6
```

16. Друк програмою gawk повідомлення в STDOUT (вихід CTRL+D)

```
$ gawk '{print "Hello John!"}'
```

17. Програма gawk записана у файл друкує 5-те і 1-поле рядка. Поля розділені ":".

```
    $ cat script2
{ print $5 "'s userid is " $1 }
$ gawk -F: -f script2 /etc/passwd
```

18. Команди програми gawk записані окремими рядками у файлі

```
$ cat script3
```

```
{
text="'s userid is "
print $5 text $1
}
# виконання
> awk -F: -f script3 /etc/passwd | more
```

19. Виконання команд gawk до і після обробки даних

```
$ cat script4
BEGIN {
print "Останній список користувачів і оболонок"
print " Userid      Shell"
print "--------    -------"
FS=":"
}

{
 print $1 "            "$7
}

END {
print "Кінець списку"
}
$ gawk -f script4 /etc/passwd
```

20. Декілька команд у командному рядку програми gawk

```
# echo "My name is Rich" | gawk '{$4="Dave"; print $0}'
```

Приклади практичних програм

1. Gawk програма перетворення стрічок у числа

```
# mystrtonum --- перетворення стрічок у числа
function mystrtonum(str, ret, chars, n, i, k, c)
{
   if (str ~ /^0[0-7]*$/) {
      # octal
      n = length(str)
      ret = 0
      for (i = 1; i <= n; i++) {
         c = substr(str, i, 1)
         if ((k = index("01234567", c)) > 0)
               k-- # adjust for 1-basing in awk

         ret = ret * 8 + k
      }
   } else if (str ~ /^0[xX][[:xdigit:]]+/) {
     # hexadecimal
     str = substr(str, 3) # lop off leading 0x
     n = length(str)
     ret = 0
     for (i = 1; i <= n; i++) {
        c = substr(str, i, 1)
        c = tolower(c)
        if ((k = index("0123456789", c)) > 0)
             k-- # adjust for 1-basing in awk
```

```
        else if ((k = index("abcdef", c)) > 0)
             k += 9
        ret = ret * 16 + k
      }
   } else if (str ~ \
/^[-+]?([0-9]+([.][0-9]*([Ee][0-9]+)?)?|([.][0-9]+([Ee][-+]?[0-9]+)?))$/) {
      # decimal number, possibly floating point
      ret = str + 0
    } else
         ret = "NOT-A-NUMBER"
    return ret
}
# BEGIN { # gawk test harness
# a[1] = "25"
# a[2] = ".31"
# a[3] = "0123"
# a[4] = "0xdeadBEEF"
# a[5] = "123.45"
# a[6] = "1.e3"
# a[7] = "1.32"
# a[7] = "1.32E2"
#
# for (i = 1; i in a; i++)
# print a[i], strtonum(a[i]), mystrtonum(a[i])
# }
```

2. Gawk програма округлення чисел

```
# round.awk --- округлення чисел
function round(x, ival, aval, fraction)
{
   ival = int(x) # integer part, int() truncates
   # see if fractional part
   if (ival == x) # no fraction
      return ival # ensure no decimals

   if (x < 0) {
      aval = -x # absolute value
      ival = int(aval)
      fraction = aval - ival
      if (fraction >= .5)
         return int(x) - 1 # -2.5 --> -3
      else
         return int(x) # -2.3 --> -2
   } else {
      fraction = x - ival
      if (fraction >= .5)
         return ival + 1
      else
         return ival
   }
}

# test
{ print $0, round($0) }
```

3. Генератор випадкових чисел

```
# cliff_rand.awk --- генератор випадкових чисел
BEGIN { _cliff_seed = 0.1 }
function cliff_rand()
{
   _cliff_seed = (100 * log(_cliff_seed)) % 1
   if (_cliff_seed < 0)
      _cliff_seed = - _cliff_seed
   return _cliff_seed
}
```

4. Трансляція між символами і числами

```
# ord.awk --- трансляція символи - числа
# Global identifiers:
# _ord_: numerical values indexed by characters
# _ord_init: function to initialize _ord_
BEGIN { _ord_init() }

function _ord_init( low, high, i, t)
{
   low = sprintf("%c", 7) # BEL is ascii 7
   if (low == "\a") { # regular ascii
      low = 0
      high = 127
   } else if (sprintf("%c", 128 + 7) == "\a") {
      # ascii, mark parity
      low = 128
      high = 255
   } else { # ebcdic(!)
      low = 0
      high = 255
   }
   for (i = low; i <= high; i++) {
      t = sprintf("%c", i)
      _ord_[t] = i
   }
}
```

5. Реалізація на `gawk` можливостей команди Linux `cut`. Команда `cut` використовується для виведення заданих полів або записів вхідного файлу на стандартний вивід.

```
$ who | cut -c1-8 | sort | uniq
```

```
# cut.awk --- реалізація cut на awk
# Опції:
# -f list Cut fields
# -d c Field delimiter character
# -c list Cut characters
#
# -s Suppress lines without the delimiter
#
# Requires getopt() and join() library functions
function usage( e1, e2)
{
```

```
        e1 = "usage: cut [-f list] [-d c] [-s] [files...]"
        e2 = "usage: cut [-c list] [files...]"
        print e1 > "/dev/stderr"
        print e2 > "/dev/stderr"
        exit 1
}
```

6. Реалізація на `gawk` можливостей команди Linux `split`. Команда `split` використовується для розбиття великих текстових файлів на менші частини.

```
split [-count] file [ prefix ]
```

```
# split.awk --- реалізація split на awk
#
# Requires ord() and chr() library functions
# usage: split [-num] [file] [outname]
BEGIN {
    outfile = "x" # default
    count = 1000
    if (ARGC > 4)
       usage()

    i = 1
    if (ARGV[i] ~ /^-[[:digit:]]+$/) {
        count = -ARGV[i]
        ARGV[i] = ""
        i++
    }
   # test argv in case reading from stdin instead of file
   if (i in ARGV)
      i++ # skip data file name
      if (i in ARGV) {
          outfile = ARGV[i]
          ARGV[i] = ""
      }

      s1 = s2 = "a"
      out = (outfile s1 s2)
}
```

7. Реалізація на `gawk` можливостей команди Linux `uniq`. Команда `uniq` вилучає з тексту рядки, які повторюються.

```
# histsort.awk --- вилучення рядків, які повторюються з файлу history
{
   if (data[$0]++ == 0)
      lines[++count] = $0
}
END {
   for (i = 1; i <= count; i++)
     print lines[i]
}
```

8. Визначення частотності слів у текстовому файлі.

```
# wordfreq.awk --- друк списку частотності слів
{
   $0 = tolower($0) # remove case distinctions
   # remove punctuation
   gsub(/[^[:alnum:]_[:blank:]]/, "", $0)
   for (i = 1; i <= NF; i++)
      freq[$i]++
}

END {
    for (word in freq)
        printf "%s\t%d\n", word, freq[word]
}

$ awk -f wordfreq.awk file1 | sort -k 2nr
```

9. Програма друку сигнатури автора.

```
$ echo awk.sh
awk 'BEGIN{O="~"~"~";o="=="=="==";o+=+o;x=O""O;while(X++<=x+o+o)c=c"%c";
printf c,(x-O)*(x-O),x*(x-o)-o,x*(x-O)+x-O-o,+x*(x-O)-x+o,X*(o*o+O)+x-O,
X*(X-x)-o*o,(x+X)*o*o+o,x*(X-x)-O-O,x-O+(O+o+X+x)*(o+O),X*X-X*(x-O)-x+O,
O+X*(o*(o+O)+O),+x+O+X*o,x*(x-o),(o+X+x)*o*o-(x-O-O),O+(X-x)*(X+O),x-O}'
$ bash awk.sh
```

dave_br@gmx.com

Лабораторна робота 11

Регулярні вирази

Мета роботи: вивчення регулярних виразів і отримання практичних навичок їх використання у Bash сценаріях.

Теоретичний матеріал: лекції, технічна література, ресурси інтернету.

1. Короткі теоретичні відомості

Регулярний вираз є шаблоном, який використовують програми або Linux утиліти (sed, gawk) для фільтрування тексту. Регулярні вирази також використовують мови програмування (Java, Perl, Python, C/C++) і бази даних (MySQL, PostgreSQL). Регулярні вирази реалізовуються з використанням рушіїв регулярних виразів. У світі Linux набули поширення два рушії регулярних виразів:

- базовий рушій регулярних виразів POSIX (the POSIX Basic Regular Expression (BRE) engine);
- розширений рушій регулярних виразів POSIX (the POSIX Extended Regular Expression (ERE) engine).

2. Базові регулярні вирази

Найбільш поширеним шаблоном базових регулярних виразів (РВ) є збіг символів шаблону і тексту, наприклад пошук послідовності символів `test` у вхідному потоці і їх друк у всіх випадках збігу

```
$ echo "This is a test" | sed -n '/test/p'
$ echo "This is a test" | gawk '/test/{print $0}'
```

Для запису регулярних виразів використовуються спеціальні символи

```
.*[]^${}\+?|().
```

Якщо у шаблонах РВ потрібно використати ці символи, то їх необхідно “екранувати” символом “\”, наприклад

```
$ cat data1
Ціна $1
Ціна 2
Ціна $3
$ sed -n '/\$/p' data1
Ціна $1
Ціна $3

$ cat data2
Ціна 1/2
Ціна 1:2
$ sed -n '/\//p' data2
Ціна 1/2
```

2.1. Якірні елементи

Є спеціальні (якірні) символи, які дозволяють закріпити (заякорити) шаблон на початку або вкінці рядка тексту вхідного потоку даних. Символ “^” закріплює шаблон на початку рядка, а символ “$” - в кінці рядка.

```
$ echo ”Книжка на столі”  | sed –n ‘/^Книжка/p’
Книжка на столі
$ echo ”Замовлена книжка” | sed –n ‘/книжка$/p’
Замовлена книжка
```

2.2. Спеціальний символ “.”

Спеціальний символ ”.” використовується для позначення любого символу в заданій позиції, крім символу нового рядка “\n”.

```
$ echo ”Книжка на столі” | sed –n ‘/.a/p’
Книжка на столі
```

2.3. Класи символів […]

Для позначення символів у заданій позиції з набору символів використовуються класи символів, які задаються у квадратних дужках

```
$ cat data3
aat
cat
hat
xat
$sed -n ’/[ch]at/p’ data3
cat
hat
```

Для позначення символів у заданій позиції за винятком символів з класу символів використовується знак ”^”, який ставиться на початку символів класу

```
$sed -n ’/[^ch]at/p’ data3
aat
xat
```

Класи символів можуть задаватися інтервалами

```
$ echo ”147”sed -n ’/^[0-3][3-6][6-9]/p’
147
$ echo The ”The cat is sleeping in hat” | sed -n ’/[c-h]at/p’
cat
hat
```

Крім класів символів користувача є класи спеціальних символів, табл. 1.

Таблиця 1 – **Класи спеціальних символів**

Клас	Описання
[[:alpha:]]	збіг любих алфавітних символів [a-zA-Z]
[[:alnum:]]	збіг любих число-алфавітних символів [a-zA-Z0-9]

[[:blank:]]	збіг забілу або символу табуляції
[[:digit:]]	збіг цифр від 0 до 9 [0-9]
[[:lower:]]	збіг малих алфавітних символів [a-z]
[[:print:]]	збіг любих друкованих символів
[[:punct:]]	збіг розділових символів
[::space:]]	збіг символів забіл, Tab, NL, FF, VT, CR
[[:upper:]]	збіг великих алфавітних символів [A-Z]

```
$ echo "abc" | sed -n '/[[:alpha:]]/p'
abc
$ echo "abc123" | sed -n '/[[:digit:]]/p'
abc123
$ echo "This is, a test" | sed -n '/[[:punct:]]/p'
This is, a test
```

2.4. Символ “*”

Розміщення символу ”*” після любого символу рядка вказує на його повторюваність нуль або більше разів.

```
$ echo "ac" | sed -n '/ab*c/p'
ac
$ echo "abbbc" | sed -n '/ab*c/p'
abbbc
```

Для задання любого числа любих символів використовується шаблон ”.*” .

```
$ echo "abbbc" | sed -n '/.*/p'
abbbc
```

3. Розширені регулярні вирази

Редактор `sed` підтримує тільки базові регулярні вирази, а редактор `gawk` підтримує як базові так і розширені регулярні вирази.

3.1. Символ “?”

Символ “?” вказує, що попередній символ може повторитися нуль або один раз.

```
$ echo "bt" | gawk '/be?t/{print $0}'
bt
$ echo "bet" | gawk '/be?t/{print $0}'
bet
$ echo "bat" | gawk '/b[ae]?t/{print $0}'
bat
$ echo "bot" | gawk '/b[ae]?t/{print $0}'
$
$ echo "bet" | gawk '/b[ae]?t/{print $0}'
bet
```

3.2. Символ “+”

Символ “+” вказує, що попередній символ може повторитися один або більше разів.

```
$ echo "beeet" | gawk '/be+t/{print $0}'
beeet
$ echo "beet"  | gawk '/be+t/{print $0}'
beet
$ echo "bet"   | gawk '/be+t/{print $0}'
bet
$ echo "bt"    | gawk '/be+t/{print $0}'
```

3.3. Задання кількості повторень `{m}, {m,n}`

`{m}` – кількість повторень попереднього символу `m`;

`{m,n}` - кількість повторень попереднього символу найменша `m`, а найбільша `n`.

```
$ echo "bt"   | gawk --re-interval '/be{1}t/{print $0}'
$
$ echo "bet"  | gawk --re-interval '/be{1}t/{print $0}'
bet
$ echo "beet" | gawk --re-interval '/be{1}t/{print $0}'

$ echo "bt" | gawk --re-interval '/b[ae]{1,2}t/{print $0}'
$
$ echo "bat" | gawk --re-interval '/b[ae]{1,2}t/{print $0}'
bat
$ echo "bet" | gawk --re-interval '/b[ae]{1,2}t/{print $0}'
bet
$ echo "beat" | gawk --re-interval '/b[ae]{1,2}t/{print $0}'
beat
$ echo "beet" | gawk --re-interval '/b[ae]{1,2}t/{print $0}'
beet
$ echo "beeat" | gawk --re-interval '/b[ae]{1,2}t/{print $0}'
$
$ echo "baeet" | gawk --re-interval '/b[ae]{1,2}t/{print $0}'
$
$ echo "baeaet" | gawk --re-interval '/b[ae]{1,2}t/{print $0}'
```


3.4. Об’єднання шаблонів логічним АБО (OR)

Об’єднання шаблонів логічним виразом `expr1 | expr2` дозволяє здійснювати пошук на збіг за одним або іншим виразом

```
$ echo "The cat is asleep"   | gawk '/cat|dog/{print $0}'
The cat is asleep
$ echo "The dog is asleep"   | gawk '/cat|dog/{print $0}'
The dog is asleep
$ echo "The sheep is asleep" | gawk '/cat|dog/{print $0}'
```

3.5. Об’єднання шаблонів у групи (…)

Шаблони можуть бути об’єднані в групу за допомогою круглих дужок. Група розглядається як окремий символ і тому до неї можуть бути застосовані регулярні вирази.

```
$ echo "123" | gawk ’/123(456)?/{print $0}’
123
$ echo "123456" | gawk ’/123(456)?/{print $0}’
123456
```

Звичайно групи символів об’єднують логічними символами АБО

```
$ echo "cat" | gawk ’/(c|b)a(b|t)/{print $0}’
cat
$ echo "cab" | gawk ’/(c|b)a(b|t)/{print $0}’
cab
$ echo "bat" | gawk ’/(c|b)a(b|t)/{print $0}’
bat
$ echo "bab" | gawk ’/(c|b)a(b|t)/{print $0}’
bab
$ echo "tab" | gawk ’/(c|b)a(b|t)/{print $0}’
$
$ echo "tac" | gawk ’/(c|b)a(b|t)/{print $0}’
```

Контрольні запитання.

1. Що таке регулярний вираз і його призначення?
2. Базові регулярні вирази?
3. Розширені регулярні вирази?
4. Об’єднання шаблонів регулярних виразів?

Завдання.

1. Написати Bash сценарій з використанням регулярних виразів, який підраховує у заданому файлі кількість букв, цифр, розділових знаків (.;:’”/\()[]{}@#$), знаків арифметичних операцій (+-*.*^~!=) та інших символів і виводить результат у консоль.

2. Написати Bash сценарій з використанням регулярних виразів, який визначає відсоток використання слів у заданому файлі порівняно з файлом еталонних базових слів.

3. Написати Bash сценарій з використанням регулярних виразів, який зчитує з довільного сайту всі електронні адреси і записує у файл. а потім виводить їх за групами розширення `edu, org, gov, com` у консоль.

4. Написати Bash сценарій з використанням регулярних виразів, який зчитує маршрут серверів до сайту Google і виводить їх на екран за класами мереж.

5. Написати Bash сценарій з використанням регулярних виразів, який знаходить файли з розширенням *.pdf або *.djvu на сайтах університетів України, які містять в назвах “Системне програмне забезпечення” і записує посилання на них у файл.

6. Написати Bash сценарій з використанням регулярних виразів, який генерує пароль довжиною 8 символів, що містить чергування випадкових двох символів і двох цифр.

7. Написати Bash сценарій з використанням регулярних виразів, який генерує пароль довжиною 8 символів, що містить чергування великих букв, малих букв, цифр, розділових знаків, арифметичних операцій.

8. Написати Bash сценарій з використанням регулярних виразів, який читає заданий файл і виводить слова, які містять два або більше повторів одного і того ж символу.

9. Написати Bash сценарій з використанням регулярних виразів, який читає заданий файл і виводить стрічки, які починаються і закінчуються однаковим символом.

10. Написати Bash сценарій з використанням регулярних виразів, який читає вміст поточного каталогу і виводить окремо списки файлів з доступом rwx, rw-, r--.'

11. Написати Bash сценарій з використанням регулярних виразів, який читає вміст поточного каталогу і виводить окремо списки файлів створених із заданою датою.

12. Написати Bash сценарій з використанням регулярних виразів, який читає вміст поточного каталогу і виводить окремо списки файлів відкорегованих із заданою датою.

13. Написати Bash сценарій з використанням регулярних виразів, який обробляє файл екзаменаційних оцінок студентів. Списки студенів за університетською шкалою оцінювання 1-24, 25-49, 50-59, 60-69, 70-79, 80-89, 90-100 записати у окремі файли.

14. Написати Bash сценарій з використанням регулярних виразів, який для заданого файлу міняє всі букви на початку і в кінці слів на великі букви.

15. Написати Bash сценарій з використанням регулярних виразів, який для заданого файлу знаходить букви з діапазону [а-д] і замінює на символ '*', а з діапазону [р-с] замінює на '+'.

16. Написати Bash сценарій з використанням регулярних виразів, який для заданого файлу виводить у консоль слова, які містять на початку слів букв [а-д], а в кінці слів - [т-я].

Примітка. Номер варіанту завдання вибирається за порядковим номером студента у журналі групи.

Звіт з лабораторної роботи має містити:

- назву групи, П.І.Б. студента, завдання до роботи;
- блок схему алгоритму згідно ДСТУ виконану у графічній САПР;
- текст програми із коментаріями;
- роздрук екранів із результатом запуску програми.

Приклади

1. Регулярний вираз "*" 0 або більше повторень

```
$ ls –al da*
```

2. Друк командою p за шаблоном /this/ і /This/

```
$ echo "This is a test" | sed -n '/this/p'
$ echo "This is a test" | sed -n '/This/p'
This is a test
```

2.1. Друк за шаблоном /ber 1/

```
$ echo "This is line number 1" | sed -n '/ber 1/p'
This is line number 1
```

2.2. Друк за шаблоном / /

```
$ cat data1
This is a normal line of text.
This is a line with too many spaces.
$ sed -n '/ /p' data1
This is a line with too many spaces.
```

3. Базові і розширені РЕ використовують символи .*[]^${}\+?|()

```
# в текстах пошуку ці символи потрібно екранувати символом "\".
```

```
$ echo "\ це спеціальний символ sed" | sed -n '/\\/p'
\ це спеціальний символ sed
$ echo "3 / 2" | sed -n '/\//p'
3 / 2
```

3.1. Екранування символу "$"

```
$ cat data2
The cost is $4.00
$ sed -n '/\$/p' data2
The cost is $4.00
```

4. Символи "^" і "$" закріплення шаблону до початку і кінця рядка

```
$ cat data3
This is a test line.
This is a test.
this is another test line.
A line that tests this feature.
Yet more testing of this.
```

4.1. Пошук рядків, які починаються зі слова this

```
$ sed -n '/^this/p' data3
```

4.2. Пошук рядків, які закінчуються словом line.

```
$ sed -n '/line.$/p' data3
```

4.3. Пошук рядків, які починаються словом This і закінчуються словом test.

```
$ sed -n '/^This is a test.$/p' data3
```

5. Символ "." дозволу у заданій позиції любих символів крім "\n" (новий рядок).

```
$ cat data4
This is a test of a line.
The cat is sleeping.
That is a very nice hat.
This test is at line four.
at ten o'clock we'll go home.

# пошук слів, які закінчуються на est
$ sed -n '/.est/p' data4
```

6. Задання букв і чисел, які належать […] і не належать [^…] до класу символів

```
# пошук слів cat, hat
$ sed -n '/[ch]at/p' data4

$ echo "Yes" | sed -n '/[Yy]es/p'
Yes
$ echo "yes" | sed -n '/[Yy]es/p'
yes

$ cat data5
Цей рядок не містить чисел
Цей рядок містить число 1
Цей рядок містить число 2
Цей рядок містить число 8
```

```
1234
$ sed -n '/[0123]/p' data5
$ sed -n '/[^01234]/p' data5
```

7. Задання класу символів інтервалом [0-9], [a-z]

```
$ sed -n '/[а-я]/p' data5
$ sed -n '/[0-9]/p' data5
```

8. Спеціальні класи символів:

```
# [[:alpha:]], [[alnum:]],  [[:blank:]], [[:digit:]], [[:lower:]],
# [[:print:]], [[:punct:]], [[:space:]], [[:upper:]]
$ echo "abc" | sed -n '/[[:digit:]]/p'
$ echo "abc" | sed -n '/[[:alpha:]]/p'
$ echo "abc123" | sed -n '/[[:digit:]]/p'
$ echo "Це є, тест" | sed -n '/[[:punct:]]/p'
$ echo "Це є тест"  | sed -n '/[[:punct:]]/p'
```

9. Символ "*" повторення попереднього символу нуль або більше разів

```
$ sed -n '/12*/p' data5

# []* - відсутність або довільне повторення символів із класу
$ sed -n '/1[23]*4/p' data5
```

10. Використання шаблонів для фільтрування електронної пошти

```
>cat email_list
user@abc.now
user1@abc-now
user1@pu.if.ua
user2@gmail.com
user@abc@abc.org

#!/bin/bash
# isemail.sh - фільтрування електронної пошти
# запуск на виконання
# cat email_list | ./isemail.sh
gawk          --re-interval          '/^([a-zA-Z0-9          \-\.\+]+)@([a-zA-Z0-9
\-\.]+)\.([a-zA-Z]{2,5})/{print $0}'
```

11. Фільтрування номерів телефонів

```
>cat phone_list
000-000-0000
123-456-7890
212-555-1234
(317)555-1234
(202) 555-9876
33523
1234567890
234.123.4567

# isphone
#!bin/bash
# isphone.sh - фільтрування номерів телефонів
# запуск на виконання
# cat phone_list | ./isphone.sh
```

```
gawk --re-interval '/^\(?[2-9][0-9]{2}\)?(| |-|\.)[0-9]{3}( |-|\.)[0-9]{4}/{print
$0}'
```

12. Gawk, символ "?" – повторення попереднього символу нуль або один раз

```
$ echo "bt" | gawk '/be?t/{print $0}'
bt
```

13. Gawk, символ "+" – повторення попереднього символу один або більше число разів

```
$ echo "12333" | gawk '/123+/{print $0}'
12333
```

14. Задання кількості повторень попереднього символу {n}, {n,m)

```
$ echo "bat" | gawk --re-interval '/b[ae]{1}t/{print $0}'
$ echo "baat" | gawk --re-interval '/b[ae]{1,2}t/{print $0}'
baat
```

15. Gawk, об'єднання шаблонів логічним виразом АБО "|"

```
$ echo "Продаються двері" | gawk '/двері|вікна/{print $0}'
The cat is asleep
```

16. Об'єднання шаблонів у групи (…)

```
> echo "cat" | gawk '/(c|b)a(b|t)/{print $0}'
cat
> echo "cab" | gawk '/(c|b)a(b|t)/{print $0}'
cab
```

17. Використання шаблонів для підрахунку кількості файлів у каталогах змінної середовища PATH

```
#!/bin/bash
# заміна ":" на " "
mypath=`echo $PATH | sed 's/:/ /g'`
count=0
for directory in $mypath
do
  check=`ls $directory`
  for item in $check
  do
   count=$[ $count + 1 ]
  done
  echo "$directory - $count"
  count=0
done
```

```
>bash 17.sh
/usr/local/bin - 79
/bin - 86
/usr/bin - 1502
/usr/X11R6/bin – 175
```

Лабораторна робота 12

Мережеві команди

Мета роботи: вивчити використання основних мережевих команд в сценаріях Bash

1. Команди для роботи з мережею

Для отримання даних і роботи з мережею використовуються команди `ip`, `netstat`, `ss`, `netcat`, `iptables`, `lsof`, `curl`, `wget`.

`ip` - робота з маршрутами, пристроями, тунелями.

`netstat` – відображення поточного статусу підключень по TCP/IP або UDP, таблиць маршрутизації, кількості адаптерів та статистики протоколів.

`ss` - виведення статистики сокетів (заміна для команди `netstat`).

`netcat` - робота з TCP/UDP з'єднаннями, HTTP серверами .

`iptables` - підтримка мережевих сокетів.

`lsof` – підтримка списку всіх відкритих файлів і процесів, які їх відкрили.

`curl` – передача та отримання даних з сервера з використанням протоколів DICT, FILE, FTP, FTPS, GOPHER, HTTP, HTTPS, IMAP, LDAP, LDAPS, POP3, RTMP, SCP, SFTP, SMTP, TELNET, TFTP.

`wget` – завантаження файлів з web серверів.

2. Стратегії клієнт-сервер і рівний-до-рівного (лорд-лорд, peer-to-peer)

У стратегії клієнт-сервер одна програма назначається сервером, а інша клієнтом. Сервер прослуховує порти на запити, обробляє їх і відправляє відповіді. Клієнти надсилають запити і отримують відповіді. Веб-сервер і веб-переглядач є прикладом такої пари. Переглядач завжди ініціює обмін даними з веб-сервером. Стратегія клієнт-сервер є найбільш поширеною через її простоту.

Засобами Bash можна написати тільки клієнт, так як він не має можливостей для прослуховування вхідних з'єднань для заданого порту. Але у Bash сценарії можуть використовуватися команди Linux, які мають такі можливості.

У стратегії рівний-до-рівного (peer-to-peer) в одній програмі інтегровано клієнт і сервер, шляхом призначення функцій клієнта і сервера різним портам. Така програма може приймати і відправляти запити. Ця стратегія використовується для розподілення великих задач або даних між декількома комп'ютерами, працюючими паралельно.

3. IP-адреса і сокети

Кожний комп'ютер в мережі має унікальну IP-адресу і одне або декілька імен хосту `hostname`. `IP`-адресу за іменем хосту `hostname` можна отримати командою `host`

```
$ host pnu.edu.ua
pnu.edu.ua has address 194.44.152.134
```

Локальну адресу хосту можна отримати командою

```
> ip addr
etho: …
```

```
inet 192.168.1.101
```

Коли комп'ютер працює не в мережі використовується віртуальний інтерфейс `loopback`, призначений для внутрішнього зв'язку. Інтерфейс запускається на хості з іменем `localhost` і IP-адресою 127.0.0.1. Реальна фізична адреса інтерфейсу (`eth0`) 192.168.1.1.

Для мережевого зв'язку між комп'ютерами використовують протокол Інтернету і локальних мереж (LAN) TCP/IP (Transport Control Protocol/Internet Protocol). В цього протоколі важливим поняттям є сокет (socket).

Сокет – це мережева структура даних, що реалізує поняття "Кінцевої точки зв'язку". Перш ніж встановити зв'язок, мережеві застосування спочатку повинні створити сокети. Їх можна порівняти з телефонними розетками, без підключення до яких неможливо увійти до телефонної мережі. Сокети бувають двох різновидів: файлові і мережеві.

Можна відкрити мережеві TCP/IP сокети у Bash використовуючи спеціальний шлях `/dev/tcp`. Це не шлях до реального файлу пристрою, а Bash інтерпретує його як запит на відкриття сокету. Bash може також створити UDP (User Datagram Protocol) сокет використовуючи шлях `/dev/udp`. Однак, UDP використовуються для надсилання коротких повідомлень, частина яких може не прибувати до місця призначення.

На додаток до протоколу і сокету потрібний порт. Порт є числом, яке використовуються для ідентифікації різних служб на віддаленому комп'ютері. Наприклад, веб-сервер звичайно використовує порт з номером 80. Для Bash сценарію сокет є реальним файлом з незвичайним шляхом. Сокет діє як іменований канал, заблокований, поки в ньому є нова інформація для зчитування. Коли сокет закритий, команда `read` повертає признак досягнення кінця файлу. На відміну від каналів сокет завжди доступний для читання і запису.

4. Синтаксис мережевих сокетів

Для відкриття мережевих сокетів TCP або UDP використовується наступний синтаксис:

```
exec {file-descriptor}<>/dev/{protocol}/{host}/{port}
```

`file-descriptor` – файловий дескриптор який асоціюється із сокетом і має починатися з 3, так як дескриптори 0, 1, 2 зарезервовані за `stdin`, `stdout` і `stderr`.

`<`, `>`, `<>` - дескриптор відкритий для читання, записування, читання/записування;

`protocol` – `tcp` або `udp`.

Після завершення передачі даних відкритий сокет має бути закритий з використанням наступного синтаксису:

```
$ exec {file-descriptor}<&-
$ exec {file-descriptor}>&-
```

Перша команда закриває вхідне з'єднання, а друга – вихідне.

Створення сокету TCP для читання і записування:

```
exec 3<>/dev/tcp/pnu.edu.ua/80
```

5. Bash клієнт. Читання і записування у відкритий сокет

Запис повідомлення із змінної `$MESSSAGE` у сокет:

```
> echo -ne $MESSAGE >&3
> printf $MESSAGE >&3
```

Читання повідомлення із сокету у змінну:

```
> read -r -u -n $MESSAGE <&3
> MESSAGE=$(dd bs=$NUM_BYTES count=$COUNT <&3 2> /dev/null)
```

Приклади.

1. Завантаження віддаленої web-сторінки і її друк:

```
# 1.sh
# 2-й рядок відкрити файловий дескриптор 3 для читання/записування у сокет TCP/IP
# 3-й рядок відправити запит у сокет
# 3,4-й прочитати відповідь і вивести на екран
#!/bin/bash
exec 3<>/dev/tcp/pnu.edu.ua/80
echo -e "GET /HTTP/1.1\r\nhttps://pnu.edu.ua\r\nConnection: close\r\n\r\n">&3
msg=”$(cat <&3)”
echo $msg

./1.sh
HTTP/1.1 400 Bad Request
Date: Tue, 27 Nov 2018 11:00:19 GMT
Server: Apache
Content-Length: 226
Connection: close
Content-Type: text/html; charset=iso-8859-1
```

2. Висвітлення версії віддаленого SSH:

```
# 2.sh
#!/bin/bash
exec 3< /dev/tcp/192.168.0.10/22
timeout 1 cat <&3
```

В дійсності цей сценарій може бути скорочений до

```
#!/bin/bash
timeout 1 cat </dev/tcp/192.168.0.10/22
```

3. Друк поточного часу з сервера точного часу:

```
# 3.sh
#!/bin/bash
cat </dev/tcp/time.nist.gov/13

./3.sh
58449 18-11-27 10:34:41 00 0 0 925.7 UTC(NIST) *
```

4. Перевірка Інтернет з’єднання:

```
# 4.sh
#!/bin/bash
HOST=pnu.edu.ua
PORT=80
(echo >/dev/tcp/${HOST}/${PORT}) &>/dev/null
if [ $? -eq 0 ]; then
   echo "Connection successful"
```

```
else
   echo "Connection unsuccessful"
fi
$ ./4.sh
Connection successful
```

5. Сканування портів віддаленого хосту:

```
# 5.sh
#!/bin/bash
host=$1
port_first=1
port_last=2000
for ((port=$port_first; port<=$port_last; port++))
do
  (echo >/dev/tcp/$host/$port) >/dev/null 2>&1 && echo "$port open"
done

$ ./5.sh localhost
25 open
631 open
1716 open
```

6. Сервер і клієнт з використанням команди `netcat`

Команда `netcat (nc)` використовується для роботи з TCP і UDP сокетами, HTTP серверами. При роботі із сокетами використовуються наступні опції:

-4 – адресація IP4;

-6 – адресація IP6;

-i інтервал – затримка між відправленням і отриманням пакету;

-k – заставляє команду продовжувати прослуховувати інші з'єднання, після завершення поточного;

-l – прослуховування вхідних з'єднань;

-n – не використовувати DNS та інші сервіси;

-p – номер використовуваного порту;

-s – IP-адреса для відправлення пакетів;

-u – використання UDP пакетів замість TCP;

-w timeout – час закриття з'єднання, яке знаходиться в режимі очікування.

Приклад. Простий сервер і клієнт TCP для передачі повідомлення:

Консоль 1. Сервер:

```
> nc -lk localhost 3000
```

Консоль. Клієнт:

```
> nc localhost 3000
Привіт
світ!
<CTRL+C>
```

Приклад. Пересилання файлів.

Машина 1:

```
> nc -l 3000 > filename.out
```

Машина 2:

```
> nc host.example.com 3000 < filename.in
```

Приклад. Сервер:

```
# 6.sh
for i in 300{0..5}; { echo "Server" | ( nc -lknv "$i") & }


./6.sh
Connection from 127.0.0.1 port 3001 [tcp/*] accepted
Client 3001
Connection from 127.0.0.1 port 3004 [tcp/*] accepted
Client 3004
Connection from 127.0.0.1 port 3002 [tcp/*] accepted
Client 3002
Connection from 127.0.0.1 port 3003 [tcp/*] accepted
```

Приклад. Клієнт:

```
# 7.sh
for i in 300{0..5}; { echo "Client $i" | ( nc localhost "$i" ) }

./7.sh
Serber
Server
Server
```

Приклад. Отримання домашньої сторінки із Web сервера:

```
$ echo -n "GET / HTTP/1.0\r\n\r\n" | nc pnu.edu.ua 80
```

Приклад. Відправлення e-mail на SMTP сервер:

```
> nc [-C] localhost 25 << EOF
HELO pnu.edu.ua
MAIL FROM:<user@pnu.edu.ua>
RCPT TO:<user2@pnu.edu.ua>
DATA
Body of email.
...
QUIT
EOF
```

Приклад. Перевірка стану портів у заданому діапазону (без встановлення з'єднання):

```
> nc -z pnu.edu.ua 20-30
Connection to host.example.com 22 port [tcp/ssh] succeeded!
Connection to host.example.com 25 port [tcp/smtp] succeeded!
```

Приклад. Отримання інформації про програмне забезпечення сервера:

```
$ echo "QUIT" | nc pnu.edu.ua 20-30
SSH-1.99-OpenSSH_3.6.1p2
Protocol mismatch.
220 host.example.com IMS SMTP Receiver Version 0.84 Ready
```

Приклад. Відкрити TCP з'єднання до 42 порту хоста pnu.edu.ua, використовуючи порт 31337 як джерело, з часом простою timeout 5 сек:

```
> nc -p 31337 -w 5 pnu.edu.ua 42
```

Приклад. Відкрити UDP з'єднання до порту 53 хосту pnu.edu.ua:

```
> nc -u pnu.edu.ua 53
```

Приклад. Відкрити TCP з'єднання до порту 42 pnu.edu.ua використовуючи 10.1.2.3 як IP для локального кінця з'єднання:

```
> nc -s 10.1.2.3 pnu.edu.ua 42
```

Приклад. Створити і прослуховувати сокет Unix домену:

```
> nc -lU /var/tmp/dsocket
```

Приклад. З'єднатися з портом 42 хосту pnu.edu.ua через HTTP проксі з IP-адресою 10.2.3.4, порт 8080:

```
> nc -x10.2.3.4:8080 -Xconnect pnu.edu.ua 42
```

Цей самий приклад, але з ідентифікацією користувача з username "ruser" як проксі:

```
> nc -x10.2.3.4:8080 -Xconnect -Pruser host.example.com 42
```

7. Команда `netstat`

Команда `netstat` відображає статистику активних підключень TCP, прослуховуваних портів комп'ютером, статистику Ethernet, таблиці маршрутизації, статистики IPv4 і IPv6.

Формат командного рядка:

```
netstat [-a] [-b] [-e] [-f] [-n] [-o] [-p протокол] [-r] [-s] [-t] [інтервал]
```

-a – відображення всіх підключень і очікуючих портів.

-b – відображення виконуваного файлу, який бере участь у створенні кожного підключення або очікує відповіді.

-e – відображення статистики Ethernet.

-f – відображення повного імені домену для зовнішніх адрес.

-n – відображення адрес і номерів портів у числовому форматі.

-o – відображення коду (ID) кожного процесу.

-p протокол – TCP, UDP.

-r – відображення вмісту таблиць маршрутів.

-s – відображення статистики протоколу.

-t – відображення поточного стану підключення.

-v – детальне виведення інформації при можливості.

Приклади.

Відобразити всі з'єднання у посторінковому режимі виведення на екран:

```
netstat –a | more
```

Відобразити всі з'єднання із статусом LISTENING, тобто порти які прослуховуються:

```
netstat –a | find /I "LISTENING"
```

Відобразити всі з'єднання і зв'язані з ними програми:

```
netstat –ab
```

Приклад. Перевірка відкритих портів командою `netstat`:

```
> netstat -tulanp
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address Foreign Address  State PID/Program name
tcp 0  0 127.0.0.1:631  0.0.0.0:* LISTEN - tcp 0 0 127.0.0.1:25 0.0.0.0:* LISTEN -
tcp 0  0       ::1:631       :::* LISTEN -
tcp 0  0       ::1:25        :::* LISTEN -
tcp 0  0        :::1716      :::* LISTEN 2104/kdeconnectd

> netstat pnu.edu.ua
Proto RefCnt Flags Type State  I-Node  Path
unix  2      [ ]   DGRAM      26897    /run/user/1000/systemd/notify
unix  3      [ ]   DGRAM       9260    /run/systemd/notify
unix  2      [ ]   DGRAM       9262    /run/systemd/cgroups-agent
```

8. Команда ss

Команда ss виводить статистику сокетів TCP в більшому обсязі порівняно з іншими командами. Команда має наступні опції:

-n – не розкривати імена служб;
-r – не розкривати імена портів;
-a – показувати всі сокети;
-l – показувати слухаючі сокети;
-o – показувати інформацію таймера;
-e – показувати детальну інформацію про сокет;
-m – показати використання пам’яті сокетом;
-p – показати процес, який використовує сокет;
-i – показати внутрішню інформацію TCP;
-4 – показувати тільки сокети IP4;
-6 – показувати тільки сокети IP6;
-0 – показувати пакетні сокети;
-t – показувати тільки сокети TCP;
-u – показувати тільки сокети UDP;
-d – показувати тільки сокети DCCP;
-w – показувати тільки сокети RAW;
-x – показувати тільки сокети Unix.
Приклади.
`ss -t -a` – показати всі сокети TCP;
`ss -u -a` – показати всі сокети UDP.

9. Команда налаштування мережі ip

Синтаксис команди:

```
ip [опції] об’єкт команди [параметри]
```

Опції:

-`v` – виведення інформації про команду.
-`s` – виведення статистичної інформації.
-`f` – протокол `bridge | dnet | inet | inet6 | ipx | link`

-o – виведення кожного запису з нового рядка.
-4 – для –f inet.
-6 – для –f inet6.
-B – для –f bridge.
-O – для –f link.

Об’єкти:
address – мережева адреса на пристрої.
link – фізичний мережевий пристрій.
monitor – монітор стану пристрою.
neigh – ARP.
route – керування маршрутизацією.
rule – правила маршрутизації.
tunnel – налаштування тунелювання.

Команди налаштування мережі:
add, change, del, flush, get, list, show, monitor, replace, restore, save, set, update.

Параметри (залежать від об’єкта і команди):
dev ім’я пристрою – мережевий пристрій.
up – включити.
down – виключити.
llarrr – MAC адреса.
initcwnd – розмір вікна перевантаження TCP при ініціалізації.
window – розмір вікна TCP.
cwnd – розмір вікна перевантаження TCP.
type – тип.
via – підключитися до роутера.
default – маршрут за замовчуванням.
blackhole – маршрут “чорна дірка”.
prohibit – маршрут “заборони”
unreachable – недосяжний маршрут.

Приклади використання команди ip link:
ip link show – показати стан всіх мережевих інтерфейсів
ip link list up – показати стан всіх включених інтерфейсів
ip link set eth1 up/down – включити/виключити eth1.

Приклади використання команди ip neigbour:
ip neigh show – показати всі записи ARP.

Приклади використання команди ip address:
ip addres show – показати всі ip адреси і їх інтерфейси
ip addr add 1.1.1.13/24 dev eth0 – встановити ip адресу для інтерфейсу eth0

`ip addr del 1.1.1.13/24 dev eth0` – вилучити ip адресу для інтерфейсу `eth0`

Приклади використання команди `ip route`:

`ip r sh` - показати всі маршрути в таблиці маршрутизації.

`Ip route get 10.10.20.0/24 from 192.168.12.9` – відобразити маршрут до вказаної мережі від заданого інтерфейсу.

10. CGI-сценарій

CGI (Common Gateway Interface) сценарій це програма, яка виконується веб-сервером. CGI-сценарії розміщуються у спеціальному каталозі `/srv/www/cgi-bin`. CGI-сценарію дають розширення `.cgi`, щоб відрізнити його від сценаріїв загального призначення із розширеннями `.sh`.

CGI сценарій записує коротке HTTP повідомлення веб-серверу (заголовок) перед відправленням любої інформації. Найкоротший заголовок містить рядок з описанням різновиду даних, які буде повернено веб-переглядачу і кодування, а після нього новий порожній рядок.

```
printf ”Content-type: text/plain; charset=utf-8\n\n”
```

CGI-сценарій записується у каталог `/srv/www/cgi-bin` з правом на виконання і викликається із переглядача:

- у локальному режимі `http://localhost/cgi-bin/env.cgi;`
- у віддаленому режимі `http://194.44.152.134/cgi-bin/env.cgi`.

Взнати IP-адресу хоста можна командою `host`:

```
>  host pnu.edu.ua
194.44.152.134
```

Так як CGI-програма викликається на виконання переглядачем, то її не можна налагодити у консольному режимі. Найбільш загальні помилки, які виникають при виконанні CGI-програми

```
403 Forbidden — The script permissions are wrong or the script is not in a CGI directory
404 Not Found — The CGI script was missing or the wrong URL was used.
500 Internal Server Error—The script didn’t return a proper CGI header.
```

Всі помилки, які записуються у стандартний потік помилок, записуються також і у журнал веб-сервера (log файл). Так як незручно у журналі веб-сервера шукати помилки, рекомендується захоплювати їх у сценарії і направляти у потік стандартного виведення на веб-сторінку. Для того щоб поверталася веб-сторінка, а не “плоский” (plain) текст, необхідно у вмісті сторінки вказати `text/html`:

```
printf ”Content-type: text/html; charset=utf-8\n\n”
```

При виведенні помилок у веб-сторінку потрібно передбачити, щоб їх колір і шрифт відрізнявся.

10.1. Змінні середовища CGI-програми

CGI-програма має додаткові змінні середовища, які назначаються веб-сервером:

```
AUTH_TYPE — Authorization type if pages are password protected
CONTENT_LENGTH — Number of bytes being written to standard input (for POST forms)
CONTENT_TYPE — The form’s content type
```

```
DOCUMENT_ROOT — The root directory of the Web server's document tree
GATEWAY_INTERFACE — The version of the CGI standard being used by the Web server
HTTP_ACCEPT — Types of data acceptable to the browser (for example, text/html)
HTTP_ACCEPT_CHARSET — Character set requested by the Web browser
HTTP_ACCEPT_ENCODING — Compression methods allowed by the Web browser (for example, gzip)
HTTP_ACCEPT_LANGUAGE — Language requested by the Web browser (for example,en for English)
HTTP_USER_AGENT — The browser used by the user
HTTP_HOST — The URL's hostname
HTTP_REFERER — The Web page executing this CGI program
PATH_INFO — Extra information included in the URL
PATH_TRANSLATED — PATH_INFO, as a file/directory under the root of the document tree
QUERY_STRING — For GET forms, the variables on the form
REMOTE_ADDR — IP of the user's computer
REMOTE_HOST — Hostname of the user's computer
REMOTE_USER — Username used when accessing password-protected pages
REQUEST_METHOD — Usually GET or POST
SCRIPT_NAME — Pathname of the script being executed
SCRIPT_FILENAME — The absolute pathname of the script being executed
SERVER_ADDR — IP address of the Web server
SERVER_ADMIN — Email address to email messages to the person in charge of the Web server
SERVER_NAME — Domain name
SERVER_PORT — The TCP/IP port used to connect to the Web server
SERVER_PROTOCOL — Version of HTTP used by the server
SERVER_SOFTWARE — Description of the Web server
```


10.2. Оброблення форм

HTML еквівалентом змінних середовища є форма. Кожна форма містить набір змінних, наприклад тег `<input type="hidden" name="user" value="Ivanenko">` є HTML еквівалентом `declare user="Ivanenko"`. Форма може містити багато інших тегів, атрибути яких змінюються користувачем і відправляються CGI-програмі. Для відправлення даних використовується два методи GET і POST.

У методі GET змінні форми зберігаються у змінній середовища `QUERY_STRING` і передаються разом з URL адресою.

Для наступної html сторінки змінна середовища `QUERY_STRING` буде містити імена і значення двох змінних `user=Ivanenko&submit=Click+Me%21`

```
<html>
<head>
<title>Form Test</title>
</head>
<body>
<form action="http://localhost/cgi-bin/form.cgi>
<input type="hidden" name="user" value="Ivanenko">
<input type="submit" name="submit" value="Click Me!">
</form>
</body>
</html>
```

Одна з причин, чому важко обробляти форми, полягає у кодуванні інформації. Стандартне кодування форми `(x-www-form-urlencoded)` замінює знак "+" і не букво-цифрові символи їх шістнадцятковими ASCII кодами із знаком "%" попереду.

POST метод записує змінні у стандартний вхід, запобігаючи можливості переповнення буфера веб-сервера для довгого списку змінних. Змінні передаються окремо від URL адреси

веб-сторінки. Коли сценарій читає змінні у циклі `while`, останній рядок не обробляється так як він містить тільки признак нового рядка.

```
while read LINE ; do
  echo ”$LINE<br />”
done
  echo ”$LINE<br />”
```

Таким чином виводиться та сама закодована інформація, яка міститься у змінній середовища `QUERY_STRING` для метода `GET`.

Контрольні запитання.

1. Команди роботи з мережею.
2. Стратегії клієнт-сервер.
3. IP адреса і сокет.
4. Синтаксис мережевих сокетів.
5. Робота Bash-клієнта із сокетами.
6. Створення сервера і клієнта з використанням команди `netcat`.
7. Команди `netstat, ss`.
8. Команда налаштування мережі `ip`.
9. Правила написання CGI-сценарію.
10. Оброблення форм методами GET і POST.

Завдання.

1. Написати функцію `run_time()`, яка виводить час виконання довільного сценарію, з використанням функцій `start_time()` і `stop_time()`, які запам'ятовують у змінних поточний час із сервера поточного часу.

2. Написати сервер для роботи з TCP з використанням команди `netcat`, який приймає повідомлення від декількох клієнтів і відправляє їм відповідь.

3. Написати сервер для роботи з UDP з використанням команди `netcat`, який приймає повідомлення від декількох клієнтів і відправляє їм відповідь.

4. Написати сценарій, який використовує команду `netstat` для збору статистики про активні підключення TCP і записує її в log і файл.

5. Написати сценарій, який використовує команду `ss` для збору статистики про активні підключення TCP і записує її в log файл.

6. Написати сценарій, який включає/виключає мережевий інтерфейс `eth0`.

7. Написати сценарій, який закриває задані порти.

8. Написати на Bash CGI-програму, яка повертає стрічку “Вітання всім!”

9. Написати сценарій, який виводить HTML сторінку із змінними середовища `env`.

10. Написати сценарій, який декодує і виводить змінні простої GET і POST форми.

Приклади

1. localhost/chi-bin/get_form.sh (permission 755)

```
#!/bin/bash
echo "Content-type: text/html"
echo ""
```

```
echo "Hello World!"
```

Результат виклику

```
Hello Word
```

2. localhost/chi-bin/get_form.sh (permission 755)

```
#!/bin/bash
# env.sh - виведення змінних середовища доступних CGI програмі
echo "Content-type: text/html"
echo ""

echo '<html>'
echo '<head>'
echo '<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">'
echo '<title>Environment Variables</title>'
echo '</head>'

echo '<body>'
echo 'Змінні середовища'
echo '<pre>'
env | sort
echo '</pre>'

echo '</body>'
echo '</html>'

exit 0
```

Результат виклику

```
Змінні середовища
CONTEXT_DOCUMENT_ROOT=/srv/www/cgi-bin/
CONTEXT_PREFIX=/cgi-bin/
DOCUMENT_ROOT=/srv/www/htdocs
GATEWAY_INTERFACE=CGI/1.1
HTTP_ACCEPT=text/html,application/xhtml+xml,application/xml;q=0.9,image/
avif,image/webp,*/*;q=0.8
HTTP_ACCEPT_ENCODING=gzip, deflate, br
HTTP_ACCEPT_LANGUAGE=en-US,en;q=0.5
HTTP_CONNECTION=keep-alive
HTTP_HOST=127.1.1.1
HTTP_SEC_FETCH_DEST=document
HTTP_SEC_FETCH_MODE=navigate
HTTP_SEC_FETCH_SITE=none
HTTP_SEC_FETCH_USER=?1
HTTP_UPGRADE_INSECURE_REQUESTS=1
HTTP_USER_AGENT=Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/
115.0
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
PWD=/srv/www/cgi-bin
QUERY_STRING=
REMOTE_ADDR=127.0.0.1
REMOTE_PORT=35776
REQUEST_METHOD=GET
REQUEST_SCHEME=http
REQUEST_URI=/cgi-bin/env.sh
```

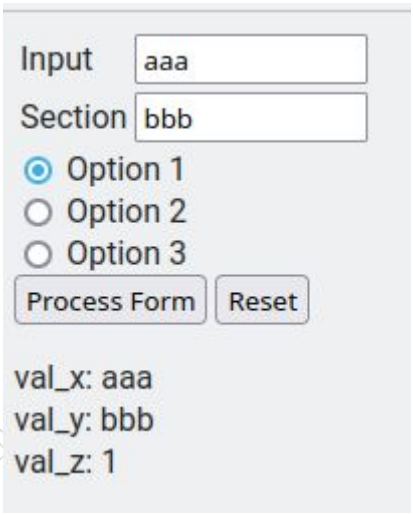
```
SCRIPT_FILENAME=/srv/www/cgi-bin/env.sh
SCRIPT_NAME=/cgi-bin/env.sh
SERVER_ADDR=127.1.1.1
SERVER_ADMIN=[no address given]
SERVER_NAME=127.1.1.1
SERVER_PORT=80
SERVER_PROTOCOL=HTTP/1.1
SERVER_SIGNATURE=
SERVER_SOFTWARE=Apache
SHLVL=1
_=/usr/bin/env
```

3. localhost/chi-bin/get_form1.sh (permission 755) метод GET

```
#!/bin/bash

echo "Content-type: text/html"
echo ""

echo '<html>'
echo '<head>'
echo '<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">'
echo '<title>Form Example</title>'
echo '</head>'
echo '<body>'

  echo "<form method=GET action=\"${SCRIPT}\">"\
       '<table nowrap>'\
          '<tr><td>Input</TD><TD><input type="text" name="val_x" size=12></td></
tr>'\
          '<tr><td>Section</td><td><input type="text" name="val_y" size=12
value=""></td>'\
          '</tr></table>'

  echo '<input type="radio" name="val_z" value="1" checked> Option 1<br>'\
       '<input type="radio" name="val_z" value="2"> Option 2<br>'\
       '<input type="radio" name="val_z" value="3"> Option 3'

  echo '<br><input type="submit" value="Process Form">'\
       '<input type="reset" value="Reset"></form>'

  # Make sure we have been invoked properly.

  if [ "$REQUEST_METHOD" != "GET" ]; then
        echo "<hr>Script Error:"\
             "<br>Usage error, cannot complete request, REQUEST_METHOD!=GET."\
             "<br>Check your FORM declaration and be sure to use
METHOD=\"GET\".<hr>"
        exit 1
  fi

  # If no search arguments, exit gracefully now.

  if [ -z "$QUERY_STRING" ]; then
        exit 0
```

```
  else
     # No looping this time, just extract the data you are looking for with sed:
     XX=`echo "$QUERY_STRING" | sed -n 's/^.*val_x=\([^&]*\).*$/\1/p' | sed "s/
%20/ /g"`
     YY=`echo "$QUERY_STRING" | sed -n 's/^.*val_y=\([^&]*\).*$/\1/p' | sed "s/
%20/ /g"`
     ZZ=`echo "$QUERY_STRING" | sed -n 's/^.*val_z=\([^&]*\).*$/\1/p' | sed "s/
%20/ /g"`
     echo "val_x: " $XX
     echo '<br>'
     echo "val_y: " $YY
     echo '<br>'
     echo "val_z: " $ZZ
  fi
echo '</body>'
echo '</html>'

exit 0
```

Відповідь сервера



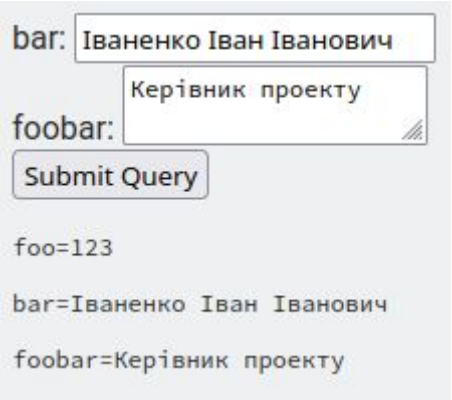
4. localhost/chi-bin/post-get.sh (permission 755) метод GET або POST з ulr-декодуванням символів

```
#!/bin/bash
echo -e "Content-type: text/html\n\n"

# (внутрішня) програма для зберігання POST даних
function cgi_get_POST_vars()
{
    # тільки обробка POST запитів
    [ "$REQUEST_METHOD" != "POST" ] && return

    # зберігання POST змінних (тільки при першому виклику)
    [ ! -z "$QUERY_STRING_POST" ] && return

    # пропустити порожній зміст
    [ -z "$CONTENT_LENGTH" ] && return
```

```
    # перевірити тип змісту
    # FIXME: не перевірено чи можна обробити  uploads...
    [ "${CONTENT_TYPE}" != "application/x-www-form-urlencoded" ] && \
        echo "bash.cgi warning: you should probably use MIME type "\
             "application/x-www-form-urlencoded!" 1>&2

    # перетворення multipart у urlencoded
    local handlemultipart=0 # дозвіл на обробку multipart/form-data (небезпечно?)
    if [ "$handlemultipart" = "1" -a "${CONTENT_TYPE:0:19}" = "multipart/form-
data" ]; then
        boundary=${CONTENT_TYPE:30}
        read -N $CONTENT_LENGTH RECEIVED_POST
        # FIXME: не використовувати awk, обробити бінарні дані (Content-Type:
application/octet-stream)
        QUERY_STRING_POST=$(echo "$RECEIVED_POST" | awk -v b=$boundary 'BEGIN
{ RS=b"\r\n"; FS="\r\n"; ORS="&" }
           $1 ~ /^Content-Disposition/ {gsub(/Content-Disposition: form-data;
name=/, "", $1); gsub("\"", "", $1); print $1"="$3 }')

    # прийняти входу стрічку як
    else
        read -N $CONTENT_LENGTH QUERY_STRING_POST
    fi

    return
}

# (внутрішня) програма для декодування urlencoded стрічок
function cgi_decodevar()
{
    [ $# -ne 1 ] && return
    local v t h
    # замінити всі  + пропуском і добавити  %%
    t="${1//+/ }%%"
    while [ ${#t} -gt 0 -a "${t}" != "%" ]; do
        v="${v}${t%%\%*}" # обробити до першого %
        t="${t#*%}"       # вилучити оброблену частину
        # декодувати, якщо є що, інакше кінець стрічки
        if [ ${#t} -gt 0 -a "${t}" != "%" ]; then
            h=${t:0:2} # зберегти перші два символи
            t="${t:2}" # вилучити ці
            v="${v}"`echo -e \\\\x${h}` # перетворити hex у спеціальний char
        fi
    done
    # повернути декодовану стрічку
    echo "${v}"
    return
}

# програма для отримання змінних із http запитів
# використання: cgi_getvars method varname1 [.. varnameN]
# метод: GET або POST або BOTH
# ім'я магічної змінної ALL, яка приймає (gets) все
function cgi_getvars()
```

```
{
    [ $# -lt 2 ] && return
    local q p k v s
    # отримання запиту (get query)
    case $1 in
        GET)
            [ ! -z "${QUERY_STRING}" ] && q="${QUERY_STRING}&"
            ;;
        POST)
            cgi_get_POST_vars
            [ ! -z "${QUERY_STRING_POST}" ] && q="${QUERY_STRING_POST}&"
            ;;
        BOTH)
            [ ! -z "${QUERY_STRING}" ] && q="${QUERY_STRING}&"
            cgi_get_POST_vars
            [ ! -z "${QUERY_STRING_POST}" ] && q="${q}${QUERY_STRING_POST}&"
            ;;
    esac
    shift
    s=" $* "
    # розбір даних запиту
    while [ ! -z "$q" ]; do
        p="${q%%&*}"  # отримання першої частини стрічки запиту
        k="${p%%=*}"  # отримання ключа (імені змінної) із стрічки запиту
        v="${p#*=}"   # отримання значення із стрічки запиту
        q="${q#$p&*}" # очищення першої частини від стрічки запиту
        # декодування і призначення змінної при необхідності
        [ "$1" = "ALL" -o "${s/ $k /}" != "$s" ] && \
            export "$k"="`cgi_decodevar \"$v\"`"
    done
    return
}

# реєстрація всіх  GET і POST змінних
cgi_getvars BOTH ALL

cat <<EOF
<html>
<body>
<form action="?foo=${foo:=123}" method="POST" enctype="application/x-www-form-
urlencoded">
bar: <input type="text" name="bar" value="$bar"><br/>
foobar: <textarea name="foobar">$foobar</textarea></br>
<input type="submit">
</form>

<pre>foo=$foo</pre>
<pre>bar=$bar</pre>
<pre>foobar=$foobar</pre>

</body>
</html>
EOF
```

Відповідь сервера



5. Використання команди curl для завантаження файлів із сайтів

```
#!/usr/bin/env bash

set -euo pipefail

declare -A rfc_urls=(
    [http-error]="https://www.rfc-editor.org/rfc/rfc7808.txt"
    [http-one]="https://www.rfc-editor.org/rfc/rfc7231.txt"
    [datetime-format]="https://www.rfc-editor.org/rfc/rfc3339.txt"
)

echo "========================="
echo "початок завантаження rfcs"
echo "========================="
echo ""

for key in "${!rfc_urls[@]}"; do
    value=${rfc_urls[$key]}
    echo "Downloading rfcs ${key}: ${value}"
    curl -OJLs "${value}"
done

echo ""
echo "============================"
echo "завантаження rfcs завершено"
echo "============================"
$ ls r*
 rfc3339.txt  rfc7231.txt  rfc7808.txt
```

Лабораторна робота 13

Автоматизація сценаріїв

Мета: вивчення засобів автоматизації сценаріїв

1. Теоретична частина

Для автоматизації інтерактивної роботи Bash використовується інструмент на основі мови Tcl – `expect`.

Встановлення `expect`:

в Ubuntu – `$ apt install expect`

в CentOs – `$ yum install expect`

в Suse Leap – `$ zypper instal expect`

Шлях розміщення expect: `$ which expect`

`expect` надає набір команд, які дозволяють взаємодіяти з утилітами командного рядка.

Основні команди `expect`:

- `spawn` – запуск процесу або програми. Наприклад, це може бути командна оболонка, ftp, Telnet, ssh, scp і т. д.
- `expect` – очікування даних, які виводить програма. При написанні сценарію можна вказати, якого саме виводу він чекає і як потрібно на нього реагувати.
- `send` – відправлення відповіді. `expect` сценарій за допомогою цієї команди може відправляти вхідні дані автоматизованій програмі. Вона подібна на команду `echo` в звичайних bash сценаріях.
- `interact` – дозволяє переключатися на «ручний» режим керування програмою.

1.1. Автоматизація bash-сценарію

Приклад сценарію, який взаємодіє з користувачем та автоматизує його за допомогою `expect`.

Код bash сценарію `questions`:

```
#!/bin/bash
echo "Привіт, Ви хто?"
read $REPLY
echo "Чи можу я задати декілька запитань?"
read $REPLY
echo "Яка Ваша улюблена тема?"
read $REPLY
```

Приклад `expect` сценарію `answerbot.sh`, який запустить сценарій `questions.sh` і буде відповідати на його запитання:

```
#!/usr/bin/expect -f
set timeout  -1
spawn ./questions
expect "Привіт, Ви хто?\r"
send – "Я Іван\r"
expect "Чи можу я задати декілька запитань?"
```

```
send -- "Так\r"
expect "Яка Ваша улюблена тема?\r"
send -- "Технології\r"
expect eof
```

На початку сценарію `answerbot.sh` знаходиться рядок ідентифікації, який містить шлях до expect, так як інтерпретувати сценарій буде саме `expect`. В другому рядку відключається дія команди завершення сценарію за часом, встановлюючи змінну expect `timeout` у значення -1. Решта рядків коду сценарію і є автоматизацією роботи з сценарієм `questions.sh`.

Спочатку, за допомогою команд `spawn`, запускається сценарій `questions.sh`. Звичайно, тут може бути викликана люба інша утиліта командного рядка. Далі задана послідовність запитань, що надходять від сценарію `questions.sh`, і відповідей, які дає на них `answerbot.sh`. Отримавши запитання від підпроцесу, `expect` видає йому задану відповідь і очікує наступного запитання.

В останній команді `expect` очікує признак кінця файлу, сценарій, дійшовши до цієї команди, завершується.

Для перевірки сценаріїв зробимо їх виконуваними файлами:

```
$ chmod u+x ./questions.sh
$ chmod u+x ./answerbot.sh
```

і виконаємо:

```
$ ./answerbot
Привіт, Ви хто?
Я Іван
Чи можу я задати декілька запитань?
Так
Яка Ваша улюблена тема?
Технології
```

Як видно, сценарій `answerbot.sh` вірно відповідає на запитання сценарію `questions.sh` в повністю автоматичному режимі. Теж саме можна зробити для будь-якої утиліти командного рядка. Тут потрібно відзначити, що сценарій `questions.sh` простий і відомо, які саме дані він виводить, тому написати expect сценарій `answerbot.sh` для взаємодії з ним нескладно. Задача ускладнюється при роботі з програмами, які написані іншими розробниками. Однак, тут допоможе засіб автоматизованого створення очікуваних сценаріїв.

1.2. Autoexpect – автоматизоване створення очікуваних сценаріїв

Команда `autoexpect` (`/usr/bin/autoexpect`) дозволяє запускати програми, які потрібно автоматизувати, після чого записує те, що вони виводять, і те, що користувач вводить, відповідаючи на їх запитання. Викличемо `autoexpect`, передавши цій команді ім'я сценарію `questions.sh`. У цьому режимі взаємодії з сценарієм `questions.sh` нічим не відрізняється від звичайного: користувач сам вводить відповіді на запитання:

```
$ autoexpect ./questions.sh
autoexpect started, file is script.exp
Привіт, Ви хто?
Я Іван
Чи можу я задати декілька запитань?
```

```
Так
Яка Ваша улюблена тема?
Технології
autoexpect done, file is script.exp
```

Після завершення роботи з сценарієм `questions.sh`, `autoexpect` повідомляє про те, що збережені дані записані у файл script.exp. Вміст цього файлу:

```
…
# -Don
set timeout  -1
spawn ./questions.sh
match_max 100000
expect "Привіт, Ви хто?\r"
”
send — "Я Іван\r"
expect "Чи можу я задати декілька запитань?"
”
send -- "Так\r"
expect "Яка Ваша улюблена тема?\r"
”
send -- "Технології\r"
expect eof
```

У цілому, за винятком деяких деталей, отримано такий же сценарій, який був написаний вручну. Якщо запустити на виконання цей сценарій, результат буде таким же:

```
$ ./questions.sh
spawn ./questions.sh
Привіт, Ви хто?
Я Іван
Чи можу я задати декілька запитань?
Так
Яка Ваша улюблена тема?
Технології
```

1.3. Робота із змінними та параметрами командного рядка

Для оголошення змінних в expect сценаріях використовується команда `set`. Наприклад, для того, щоб присвоїти значення 5 змінній VAR1, використовується наступна конструкція:

```
set VAR1 5
```

Для доступу до значення змінної перед її ім'ям потрібно додати знак долара `$`. У даному випадку це буде виглядати як `$VAR1`.

Для того, щоб отримати доступ до аргументів командного рядка, з якими викликаний `expect` сценарій, можна поступити так:

```
set VAR [lindex $argv 0]
```

Тут оголошується змінна VAR і записується в неї вказівник на перший аргумент командного рядка, `$argv 0`.

Так можна записати значення першого аргументу, що задає ім'я користувача, у змінну `my_name`. Значення другого аргументу, що задає уподобання користувача, можна записати у змінну `my_favorite`:

```
set my_name [lindex $argv 0]
set my_favorite [lindex $argv 1]
```

Ці змінні добавимо у сценарій `answerbot`:

```
 #!/usr/bin/expect -f
set my_name [lindex $argv 0]
set my_favorite [lindex $argv 1]
set timeout -1
spawn ./questions
expect "Привіт, Ви хто?\r"
send -- "Я $my_name\r"
expect "Чи можу я задати декілька запитань?\r"
send -- "Так\r"
expect "Яка Ваша улюблена тема?\r"
send -- "$my_favorite\r"
expect eof
```

Запустимо сценарій, передавши в нього перший параметр `Петро`, а другий – `Програмування`:

```
$ ./answerbot.sh SomeName Programming
spawn ./questions.sh
Привіт, Ви хто?
Я Петро
Чи можу я задати декілька запитань?
Так
Яка Ваша улюблена тема?
Програмування
```

Тепер `expect` сценарій відповідає на запитання bash сценарію, використовуючи передані йому параметри командного рядка.

1.4. Відповіді на різні запитання, які можуть з'явитися в одному і тому же місці

Якщо автоматизована програма може, в одній ситуації, видати один рядок, а в іншій, у тому самому місці – інший рядок, то можна використати блоки, взяті у фігурних дужки, які містять різні варіанти реакцій сценаріїв на різні дані, отримані від програми.

```
expect {
   "something" { send -- "send this\r" }
   "*another"  { send -- "send another\r" }
}
```

У цьому прикладі, якщо expect сценарій отримає «`something`», він відправить відповідь «`send this`», а якщо це буде якась стрічка, що закінчується на «`another`», він відправить відповідь «`send another`».

Модифікований сценарій `questions`, який випадковим чином задає в одному місці різні запитання:

```
#!/bin/bash
let number=$RANDOM
if [ $number -gt 25000 ]; then
echo "Яка Ваша улюблена тема?"; else
echo "Який Ваш улюблений фільм?"
fi
```

```
read $REPLY
```

Тут генерується випадкове число при кожному запуску сценарію, і, проаналізувавши його, виводиться одне із двох запитань.

Для автоматизації такого сценарію використовуються блоки у `answerbot.sh`:

```
#!/usr/bin/expect -f
set timeout  -1
spawn ./questions
expect {
  "*тема?"  { send -- "Програмування\r" }
  "*фільм?" { send -- "Зоряні війни\r" }
}
expect eof
```

```
$ ./answerbot.sh
spqwn ./questions.sh
Який Ваш улюблений фільм?
Зоряні війни

$ ./answerbot.sh
spqwn ./questions.sh
Яка Ваша улюблена тема?
Програмування
```

1.5. Умовний оператор

Expect підтримує умовний оператор if-else та інші керуючі конструкції. Приклад використання умовного оператора в `answerbot1.sh`:

```
#!/usr/bin/expect -f
set TOTAL 1
if { $TOTAL < 5 } {
puts "\nTOTAL is less than 5\n"
} elseif { $TOTAL > 5 } {
puts "\nTOTAL greater than 5\n"
} else {
puts "\nTOTAL is equal to 5\n"
}
expect eof
```

```
$ ./asnwerbot1.sh
TOTAL < 5
```

Зверніть увагу на конфігурацію фігурних дужок. Чергова відкриваюча дужка повинна бути розміщена в тому ж рядку, що і попередня конструкція.

1.6. Цикл while

Цикли `while` в `expect` дуже подібні до тих, що використовуються в звичайних bash сценаріях, але тут застосовуються фігурні дужки:

```
#!/usr/bin/expect -f
set COUNT 0
while { $COUNT <= 5 } {
puts "COUNT = $COUNT"
```

```
set COUNT [ expr $COUNT + 1 ]
}
puts ""
```

```
$ ./while.sh
COUNT = 0
COUNT = 1
COUNT = 2
COUNT = 3
COUNT = 4
COUNT = 5
```

1.7. Цикл for

Цикл `for` в `expect` влаштований по особливому. На початку циклу, в самостійних парах фігурних дужок вказується змінна лічильник, умови зупинки циклу, правила модифікації лічильника. Потім, знову у фігурних дужках, іде тіло циклу:

```
#!/usr/bin/expect -f
for {set COUNT 0} {$COUNT <= 5} {incr COUNT} {
puts "\nCOUNT is at $COUNT"
}
puts ""
```

```
$ ./for.sh
COUNT = 0
COUNT = 1
COUNT = 2
COUNT = 3
COUNT = 4
COUNT = 5
```

1.8. Об'єднання та використання функцій

`expect` дозволяє оголошувати функції, використовуючи ключове слово `proc`:

```
proc myfunc { MY_COUNT } {
set MY_COUNT [expr $MY_COUNT + 1]
return "$MY_COUNT"
}
```

Приклад використання у `expect` сценарії функцій:

```
#!/usr/bin/expect -f
proc myfunc { MY_COUNT } {
set MY_COUNT [expr $MY_COUNT + 1]
return "$MY_COUNT"
}

set COUNT 0
while {$COUNT <= 5} {
puts "\nCOUNT is currently at $COUNT"
set COUNT [myfunc $COUNT]
}
puts ""
```

```
$ ./proc.sh
COUNT = 0
COUNT = 1
COUNT = 2
COUNT = 3
COUNT = 4
COUNT = 5
```

1.9. Команда interact

Іноді автоматизовані за допомогою `expect` програми вимагають введення конфіденційних даних, паролів, які не можна зберігати у звичайному тексті в коді сценарію. У подібній ситуації можна скористатися командою `interact`, яка автоматизує оброблення таких даних, а потім передає керування в `expect`.

Коли виконується interact команда, `expect` сценарій переключається на читання відповіді на запит програм з клавіатури.

Приклад bash сценарію, який очікує введення пароля у відповідь на одне із своїх запитань:

```
#!/bin/bash
echo "Привіт, Ви хто?"
read $REPLY
echo "Який Ваш пароль?"
read $REPLY
echo "Яка Ваша улюблена тема?"
read $REPLY
```

Приклад `expect` сценарію з дією на введення паролю:

```
#!/usr/bin/expect -f
set timeout  -1
spawn ./questions2.sh
expect "Привіт, Ви хто?\r"
send -- "Я Іван\r"
expect "*пароль?\r"
interact ++ return
send "\r"
expect "*тема?\r"
send -- "Технології\r"
expect eof
```

Зустрівши команду `interact`, `expect` сценарій зупиняється і надає можливість ввести пароль. Після введення пароля потрібно ввести «++» і `expect` сценарій продовжить роботу.

```
$ ./answerbot2.sh
Привіт, Ви хто?
Я Іван
Який Ваш пароль
anc123
Яка Ваша улюблена тема?
Технології
```

Контрольні запитання.

1. Для чого призначений інструмент `expect` і які він має команди.

2. Мета автоматизації bash сценарію з використанням `expect` сценарію.
3. Автоматизоване створення expect сценаріїв за допомогою `autoexpect`.
4. Змінні і параметри в `expect` сценаріях.
5. Створення відповідей на різні запитання, які можуть появитися в одному і тому ж місці `expect` сценарію.
6. Оператор умови `if` в `expect`.
7. Цикл `while` в `expect`.
8. Цикл `for` в `expect`.
9. Оголошення і використання функцій в `expect`.
10. Команда `interact`.

2. Практична частина

Завдання.

Автоматизувати сценарій з використанням змінних і параметрів командного рядка, відповідей, які можуть появитися в одному і тому ж місці сценарію, умовних операторів, циклів, функцій і команди `interact`.

1. Реєстрація користувача в системі “Дистанційна освіта”.
2. Реєстрація користувача в системі “Інтернет провайдер”
3. Реєстрація користувача в системі “Правила дорожнього руху”.
4. Реєстрація користувача в системі “Авіаційні білети”
5. Реєстрація користувача в системі “Залізнодорожні білети”.
6. Реєстрація користувача в системі “Театр”.
7. Реєстрація користувача в системі “Купівля машини”.
9. Реєстрація користувача в системі “Бронювання місць в готелі”.
10. Реєстрація користувача в системі “Приват24”.
11. Реєстрація користувача в системі “Абітурієнт”.
12. Реєстрація користувача в системі “Поліклініка”
13. Реєстрація користувача в системі “Призовник”.
14. Реєстрація користувача в системі “Житло”.
15. Реєстрація користувача в системі “Придбання автомобіля”.

СПИСОК ЛІТЕРАТУРИ

Основний

1. Системне програмне забезпечення [Електронний ресурс] : конспект лекцій для здобувачів освітнього ступеня «бакалавр» зі спеціальності 123 Комп'ютерна інженерія освітньої програми Спеціалізовані комп'ютерні системи денної форми навчання / упоряд.: Є.Є. Федоров, Т.Ю. Уткіна ; М-во освіти і науки України, Черкас. держ. технол. ун-т. – Черкаси: ЧДТУ, 2023. – 106 с. – Частина 1. – режим доступу: https://elib.chdtu.edu.ua/e-books/5239
2. Методичні рекомендації до лабораторних робіт з дисципліни «Системне програмне забезпечення» для здобувачів освітнього ступеня «бакалавр» зі спеціальності 123 Комп'ютерна інженерія освітньої програми Спеціалізовані комп'ютерні системи денної форми навчання [Електронний ресурс] / упоряд.: Є.Є. Федоров, Т.Ю. Уткіна ; М-во освіти і науки України, Черкас. держ. технол. ун-т. – Черкаси: ЧДТУ, 2023. – 14 с. – Частина 1 – режим доступу: https://elib.chdtu.edu.ua/e-books/5238 .
3. Операційні системи [Електронний ресурс] : навчальний посібник / І. М. Федотова-Півень, І. В. Миронець, О. Б. Півень, С. В. Сисоєнко, Т. В. Миронюк ; за ред. В. М. Рудницького, Черкаський державний технологічний університет. – Харків : ТОВ «ДІСА ПЛЮС», 2019. – 216 с. – ISBN 978-617-7645-93-0.
4. Зайцев, В. Г. Операційні системи. / Навчальний посібник для студентів спеціальності 123 «Комп'ютерна інженерія» / В. Г. Зайцев, І. П. Дробязко. – Київ: КПІ ім. Ігоря Сікорського, 2019. – 240 с.
5. Навчально-методичний посібник до виконання курсової роботи з дисципліни "Системне програмне забезпечення" для студентів спеціальності 6. 050102 "Комп'ютерна інженерія" всіх форм навчання [Електронний ресурс] / уклад. : І. М. Федотова-Півень, О. Б. Півень ; М-во освіти і науки України, Черкас. держ. технол. ун-т. – Черкаси : ЧДТУ, 2015. – 60 с.
6. Системне програмне забезпечення: навчальний посібник з дисципліни «Системне програмне забезпечення» для студентів базового напряму 6.050102 «Комп'ютерна інженерія»/ Укл.: І. В. Мороз, Л. О. Березко, О. Ю. Бочкарьов. – Львів: Видавництво Національного університету «Львівська політехніка», 2014. – 162 с.
7. Лабораторний практикум з дисципліни "Системне програмне забезпечення" для студентів напряму підготовки 6.050903 "Телекомунікації" всіх форм навчання [Електронний ресурс] / уклад. А. В. Чепинога. – Черкаси: ЧДТУ, 2010. – 66 с.

Додатковий

8. Nick Aleks, Dolev Farhi. Black Hat Bash: Creative Scripting for Hackers and Pentesters. No Starch Press, 2024. – 344 p.
9. Michael Kofler. Scripting: Automation with Bash, PowerShell, and Python—Automate Everyday IT Tasks from Backups to Web Scraping in Just a Few Lines of Code. Rheinwerk Computing, 2024. – 500 p.
10. Richard Blum, Christine Bresnahan. Linux Command Line and Shell Scripting Bible 4th Edition. Wiley, 2021. – 832 p.
11. Paul Troncone, Carl Albing.Cybersecurity Ops with bash: Attack, Defend, and Analyze from the Command Line 1st Edition. O'Reilly Media, 2019. 303 p.

12. William Shotts. The Linux Command Line, 2nd Edition: A Complete Introduction. No Starch Press, 2019. – 504 p.
13. Carl Albing. bash Cookbook: Solutions and Examples for bash. 2nd Edition. O'Reilly Media, 2017. – 723 p.

Інформаційні ресурси

14. Linux Mint 22 [електронний ресурс]: – Режим доступу: https://linuxmint.com/ – назва з екрану
15. Canonical Ubuntu. [електронний ресурс]: – Режим доступу: https://ubuntu.com/ – назва з екрану
16. Open Suse [електронний ресурс]: – Режим доступу: https://www.opensuse.org/ – назва з екрану
17. GNU BASH: – Режим доступу: https://www.gnu.org/software/bash/ – назва з екрану.