



**Тема: «Основи використання скриптової мови
інтерпретатору оболонки командного рядку *Unix*-подібних ОС»**

Викладач: Олександр А. Блажко,
доцент кафедри ІС Одеської політехніки, blazhko@ieee.org

Мета роботи: придбання навичок автоматизації керування ОС з використанням скриптової мови інтерпретатору оболонки командного рядку.

План роботи

1 Теоретичні відомості	1
2 Завдання до виконання	21

1 Теоретичні відомості

1.1 Основи використання скриптової мови інтерпретатору оболонки командного рядку

1.1.1 Переваги використання скриптової мови

Shell - це командна оболонка, яка є:

- проміжною ланкою між користувачем і ОС;
- мова програмування інтерпретуючого типу.

Програми на мові *shell* називають файлами-сценаріями, script-file (скриптами або скрипт-файлами).

Команди мови *shell* можна виконувати:

- безпосередньо в командному рядку;
- через попереднє зберігання команд у скрипт-файлі.

Основні причини знання мови програмування інтерпретатора командної оболонки:

1) скрипти підходять для швидкого створення прототипів складних програм, навіть не дивлячись на обмежений набір мовних конструкцій і певну "повільність", коли можна детально опрацювати структуру майбутньої програми, виявити можливі "пастки" і лише потім приступити до кодування на *C*, *C++*, *Java*.

2) синтаксис мови досить простий і нагадує команди, які вводяться в командному рядку;

3) скрипти повертають нас до класичної філософії ОС *Unix* - "розділяй і володій" (а може до класичної ідеї Адама Сміта) тобто розподіл складної задачі на декілька простих задач;

4) скрипт – це простий список команд системи, записаний в файл, тому створення скриптів допоможе зберегти час і сили, які витрачаються на введення послідовності команд щоразу, коли необхідно їх виконати повторно;

5) під час завантаження *Unix*-подібної ОС виконується великий список сценаріїв з каталогу */etc/rc.d*, які налаштовують конфігурацію ОС і запускають різні сервіси, тому дуже важливо чітко розуміти ці скрипти і мати достатньо знань, щоб вносити в них якісь зміни.

1.1.2 Ініціалізація скриптового файлу

Скрипт-файл можна виконати, передавши його в командному рядку відповідній програмі-інтерпретатору.

Але зручним способом є вказування у самому скрипт-файлі інформації про програму-інтерпретатор. Для цього перший рядок скрипт-файлу необхідно вказати послідовність *#!* – так звану *sha-bang*-послідовність, яка вказує ОС про шлях до файлу з програмою-інтерпретатором.

Приклади опису *sha-bang*-послідовні для різних програм-інтерпретаторів:

```
#!/bin/sh
#!/bin/bash
#!/usr/bin/python
#!/usr/bin/sed -f
#!/usr/bin/awk -f
```

Для коментарів у тексті скрипт-файлу також використовується символ *#* (решітка).

Для впорядкування процесів адміністрування в *Unix*-подібних ОС традиційно рекомендується визначати розширення файлу у відповідності з його типом. Для скрипт-файлів оболонки *Bash* рекомендується розширення *sh*

Скрипт-файл оболонки *Bash* можна виконати двома шляхами:

- 1) через команду *sh файл.sh*
- 2) безпосередньо виконавши сам скрипт через команду *./файл.sh*, якщо файл має права на виконання, які можна визначити командою *chmod +x файл.sh*

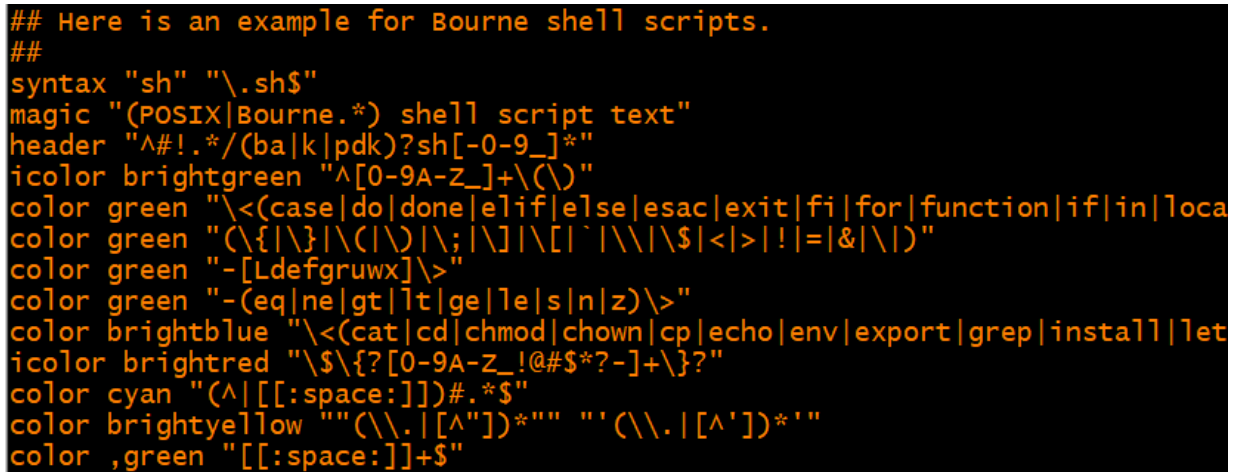
Для редагування файлів використовують різні текстові редактори, наприклад, *nano*, який присутній в *Git Bash* оболонці і має підсвітку синтаксису для різних мов програмування.

В ОС *Linux* часто за замовчуванням для редактору *nano* відсутня підсвітка синтаксису. Але її можна налаштувати:

- всі файли опису підсвітки синтаксичних конструкцій багатьох мов програмування зберігаються в каталозі */usr/share/nano/*
- опис підсвітки синтаксису скриптів з розширенням *sh* зберігається у файлі *sh.nanorc*

– за необхідністю можна змінити колір підсвітки через редагування вказаного файлу.

На рисунку 1 наведено приклад такого опису для оболонки *Bash*. Як видно з рисунку, при описі синтаксису використовуються регулярні вирази, які шукають підрядки тексту мови програмування та визначають для них відповідний колір.



```
## Here is an example for Bourne shell scripts.
##
syntax "sh" "\.sh$"
magic "(POSIX|Bourne.*) shell script text"
header "^#!.*/(ba|k|pdk)?sh[-0-9_]*"
icolor brightgreen "[0-9A-Z_]+\"
color green "\<(case|do|done|elif|else|esac|exit|fi|for|function|if|in|loca
color green "\{|\}|\(|\)|\;|\;|\[|`|\\\\|\\$|<|>|!|=|&|\\)"
color green "-[Ldefgruwx]\"
color green "-(eq|ne|gt|lt|ge|le|s|n|z)\"
color brightblue "\<(cat|cd|chmod|chown|cp|echo|env|export|grep|install|let
icolor brightred "\$\\{?[0-9A-Z_!@#\$*?~]+\\}?"
color cyan "(^|[:space:]))#.*$"
color brightyellow "\"(\\.|[^\"])*\" \"'([^\']*).*'"
color ,green "[[:space:]]+$"
```

Рис. 1 – Приклад опису підсвітки синтаксису скриптів з розширенням *sh*

Для активізації роботи підсвітки необхідно в домашньому каталозі користувача створити прихований файл *.nanorc* та вказати в ньому, наприклад, такі команди:

```
## HTML
include "/usr/share/nano/html.nanorc"
## AWK
include "/usr/share/nano/awk.nanorc"
```

1.1.3 Складна обробка текстових даних з використанням скриптів *AWK*

Утиліта *AWK* (перші літери прізвищ авторів *Alfred Aho*, *Peter Weinberger*, *Brian Kernighan*) була створена у 1977 як *C*-подібна скриптова мова рядкового аналізу та обробки вхідного потоку (наприклад, текстового файлу), яка має більше можливостей ніж утиліта *SED*.

Загальний синтаксис використання утиліти *AWK*:

```
awk -f скрипт-файл вхідний-файл
```

Скрипт-файл має наступну структуру:

```
BEGIN {початкові-команди змінні};
умова1 { команди змінні};
умова2 { команди змінні};
...
END {завершальні- змінні}
```

де:

– *умова1, умова2, ...* – умови обробки вхідних рядків файлу, які найчастіше є регулярним виразом та можуть бути об'єднані стандартними логічними операторами `&&` (оператор *AND*) та `||` (оператор *OR*);

– *команди* – це послідовності команд, аналогічні командам оболонки *Bash*, але які мають C-подібні конструкції, наприклад, для друку рядків використовується оператор *print*;

– *BEGIN* та *END* ... є необов'язковими та виконуються перед початком основних циклічних дій з обробки кожного рядка файлу або після завершення основних дій з обробки кожного рядка файлу.

AWK дозволяє описувати команди безпосередньо у командному рядку. Для цього треба їх виділяти у лапки з символом-роздільником ; між командами.

AWK оперує такими спеціальними змінними:

– *FS (Field Separator)* – роздільник стовпчиків у рядку, наприклад, *FS=" "* – прогалина між рядками, яка використовується за замовчуванням, *FS=""* (без символу-роздільнику) – кожен окремий символ стає окремим стовпчиком, *FS="re"* – роздільник-регулярний вираз *re*;

– *\$1, \$2, ...*: значення 1-го стовпчику, 2-го стовпчику поточного рядку файлу;

– *NF (Number of Field)* – кількість стовпців у рядку;

– *NR (Number of Row)* – поточний номер рядку;

– *\$0* – поточний рядок файлу;

– *OFS (Output Field Separator)* – роздільник для стовпчиків вихідних рядків;

– *ORS (Output Row Separator)* – роздільник між вихідними рядками, за замовчуванням – символ переходу на новий рядок `\n`.

Для операцій утиліта *AWK* надає C-подібний синтаксис:

– зміни значень: `=, +=, -=, *=, /=, %=, ++, --`.

– порівняння значень: `<, <=, ==, !=, >=, >`,

– порівняння із шаблоном: `~, !~`

– логічні: `!, ||, &&`

Структурні оператори:

```
if (умова) {оператори} [else {оператори}];
```

```
while (умова) {оператори};
```

```
for (вираз; умова; вираз) {оператори}
```

Наведемо декілька прикладів простого використання операторів.

Для того, щоб вивести на екран перший стовпчик файлу */etc/passwd*, де у рядках між стовпчиками роздільником є символ двокрапки, необхідно виконати наступну команду:

```
awk '{FS=":" ; print $1}' /etc/passwd
```

Наступний приклад виводить на екран перший та другий стовпчики, виставляючи між ними роздільником символ кома:

```
awk '{FS="," ; OFS="," ; print $1,$2}' /etc/passwd
```

Якщо вказані вище операції необхідно виконати над підмножиною рядків, тоді необхідно описати умови, використовуючи логічні вирази або регулярні вирази.

Значення регулярного виразу включається між двома слешами.

Наприклад, для отримання значень 6-го стовпчика з рядку, в якому значення 1-го стовпчика починається із літери *a*, необхідно:

```
awk 'BEGIN{FS=":"} $1~/^a/ {print $6}' /etc/passwd
```

Якщо попереднє завдання треба виконати лише для рядків з номерами до 100, необхідно перед регулярним виразом додати логічний вираз $NR \leq 100$ &&:

```
awk 'BEGIN{FS=":"} NR <= 100 && /^a/ {print $1}' /etc/passwd
```

На рисунку 2b наведено приклад виконання утиліти *AWK*, яка у рядках файлу *example.txt* зі змістом на рисунку 2(a) попередньо змінює роздільник між стовпчиками зі старого `\t` (символ табуляції) на новий `,` (символ коми).

На рисунку 2c наведено приклад скрипт-файлу, який додатково до дій з рисунку 2b виводить на екран рядок *0,0,0*. Приклад виконання скрипт-файлу представлено на рисунку 2d.

<pre>Об'єкт ПЗ 1 - 11 10 13 10 14 10 13 10 15 Об'єкт ПЗ ПЗ 2 ... 17 10 18 10 10 11 10 10 12 Об'єктт ПЗ 3 % 19 10 11 10 12 15 10 10 13</pre>											
(a) приклад текстового файлу <i>example.txt</i> із символом-роздільником <code>\t</code>											
<pre>[oracle@vpsj3IeQ Lab6]\$ awk '{FS="\t" ; OFS="," ; print \$1,\$2,\$3}' example.txt Об'єкт,ПЗ,1 Об'єкт ПЗ ПЗ 2,...,17 Об'єктт ПЗ 3,%,19</pre>											
(b) <code>awk 'BEGIN{FS="\t"; OFS=","} {print \$1,\$2,\$3}' example.txt</code>											
<pre>BEGIN{FS="\t"; OFS=","} {print \$1,\$2,\$3} END{print 0,0,0}</pre>				<pre>[blazhko@v~]\$ awk -f f.awk example.txt Об'єкт ПЗ 1,-,11 Об'єкт ПЗ ПЗ 2,...,17 Об'єктт ПЗ 3,%,19 ',', 0,0,0</pre>							
(c) Приклад скрипт-файлу				(d) <code>awk -f f.awk example.txt</code>							
<pre>#!/usr/bin/awk -f BEGIN{FS="\t"; OFS=","} {print \$1,\$2,\$3} END{print 0,0,0}</pre>				<pre>[blazhko@vps6iefe ~]\$ chmod +x f.awk [blazhko@vps6iefe ~]\$./f.awk example.txt Об'єкт ПЗ 1,-,11 Об'єкт ПЗ ПЗ 2,...,17</pre>							
(e) Приклад скрипт-файлу				(f) <code>./f.awk example.txt</code>							

Рис. 2 – Приклад виконання утиліти *AWK* з попередньою трансформацією значень у рядках

Для виконання скрипт-файлу без вказування назви самої утиліти *AWK* необхідно:

- 1) додати до скрипт-файлу перший рядок як *sha-bang*-послідовність *#!/usr/bin/awk -f*
- 2) надати скрипт-файлу права на виконання – *chmod +x*

1.1.4 Використання змінних оболонки *Bash*

Змінна з точки зору оболонки *Bash* – це параметр, що позначається ім'ям.

Змінні отримують значення за допомогою оператора такого вигляду:

```
name=value
```

де *name* - ім'я змінної, а *value* – значення змінної (може бути символом нового рядка).

Ім'я змінної може складатися тільки з цифр і букв і не може починатися з цифри. Значенням може бути будь-який текст. Якщо значення містить спеціальні символи, тоді його треба взяти у лапки. Якщо змінна вже задана, її можна видалити через вбудовану команду оболонки:

```
unset
```

Набір всіх встановлених змінних оболонки з присвоєними їм значеннями називається оточенням (*environment*) або середовищем оболонки. Ви можете переглянути його за допомогою команди *set* без параметрів.

Для подальшого використання змінної перед її іменем слід додавати символ *\$*

Для того щоб переглянути значення однієї конкретної змінної, можна замість команди *set* скористатися командою:

```
echo $name
```

Більш універсальною формою звертання до змінної є обертання імені у фігурні дужки {}, наприклад:

```
echo ${name}
```

Більшість мов програмування інтерпретаторів оболонки командного рядку не мають типів даних, тому тип даних змінної визначається під час першої ініціалізації значення змінної. На рисунку 3 наведено приклади ініціалізації змінних, з якого видно, що команда *echo* дозволяє комбінувати константи та змінні.

```
#!/bin/bash
y=11 #коректна ініціалізація змінної
v="" #також коректна ініціалізація
x = 3 #помилка - наявність прогалін між символом =
# ініціалізація змінної з арифметичними операціями
# зверніть увагу на необхідність дужок та пробілів
r=$(( $x + $y ))
# використання змінної в команді echo
# для отримання її значення на екран
echo "Результат=$r"
```

Рис. 3 – Приклади ініціалізації змінних

Існують зарезервовані змінні, наприклад:

\$HOME – домашній каталог користувача;

`$HOSTNAME` – DNS-назва серверу, наприклад, `vpsj3IeQ.s-host.com.ua`;

`$HOSTTYPE` – архітектура сервера, наприклад, `x86_32`, `x86_64`;

`$OSTYPE` – тип ОС;

`$PATH` – перелік каталогів, необхідних для швидкого пошуку файлів для виконання, які можуть виконуватися в командному рядку без вказування повного шляху до файлу;

`$SECONDS` – період часу роботи скрипт-файлу в секундах;

`$#` – Загальна кількість параметрів командного рядку, переданих скрипт-файлу;

`$*` або `$@` – перелік значень усіх параметрів командного рядку, переданих скрипту,

`$0`, `$1`, `$2`, ... – назва команди, перший, другий аргументи та подальші аргументи;

`$IFS` – роздільник стовпчиків у рядку (*IFS - Input Field Separator*), за замовчуванням, роздільником може бути символ прогалини, табуляція або перехід на новий рядок;

`$RANDOM` – генерація випадкового числа;

`$UID` –реальний ідентифікатор користувача, який встановлюється тільки при логіні;

`$GROUPS` – масив ідентифікаторів груп, до яких належить поточний користувач.

Змінні є нетипізованими, тому для визначення довжини у символах значення, яке зберігається у змінній, достатньо використати символ `#` перед змінною та фігурні дужки:

```
echo ${#VAR}
```

На рисунку 4 наведено приклади виклику зарезервованих змінних

```
#!/bin/bash
echo "$HOME – домашній каталог користувача"
echo "$HOSTNAME – DNS-назва серверу"
echo "$HOSTTYPE – архітектура сервера"
echo "$OSTYPE – тип ОС"
echo "$PATH – перелік каталогів"
echo "$SECONDS – період часу роботи скрипт-файлу в секундах"
echo "$# – загальна кількість параметрів командного рядку, переданих скрипт-файлу"
echo "$* – перелік значень параметрів командного рядку, переданих скрипт-файлу"
echo "$0,$1,$2 – назва команди, перший, другий аргументи, відповідно"
echo "$RANDOM – генерація випадкового числа"
echo "$UID – ідентифікатор користувача, який встановлюється тільки при логіні"
echo "$GROUPS – масив ідентифікаторів груп, до яких належить поточний користувач"
echo "${#GROUPS} – довжина у символах значення, яке зберігається у змінній $GROUPS"
```

(а) приклади виклику зарезервованих змінних

```
[oracle@vpsj3IeQ Lab6]$ ./e2.sh par1 par2
/home/oracle – домашній каталог користувача
vpsj3IeQ.s-host.com.ua – DNS-назва серверу
x86_64 – архітектура сервера
linux-gnu – тип ОС
/opt/oracle/product/18c/dbhomeXE/bin:/usr/local/bin:/usr/bin:/usr/local/sbin:/u
r/sbin:/home/oracle/.local/bin:/home/oracle/bin – перелік каталогів
0 – період часу роботи скрипт-файлу в секундах
2 – загальна кількість параметрів командного рядку, переданих скрипт-файлу
par1 par2 – перелік значень параметрів командного рядку, переданих скрипт-файлу
./e2.sh,par1,par2 – назва команди, перший, другий аргументи, відповідно
2650 – генерація випадкового числа
54321 – ідентифікатор користувача, який встановлюється тільки при логіні
54321 – масив ідентифікаторів груп, до яких належить поточний користувач
5 – довжина у символах значення, яке зберігається у змінній 54321
[oracle@vpsj3IeQ Lab6]$
```

(b) приклади результату виклику зарезервованих змінних

Рис. 4 – Приклади виклику зарезервованих змінних

Відомо, що команди оболонки *Bash* можуть передавати дані через конвейер, використовуючи символ `|` конвейєризації, наприклад:

```
команда1 | команда2
```

Але також є інший варіант особливої конвейєризації через включення однієї команди у іншу команду через такі синтаксичні конструкції:

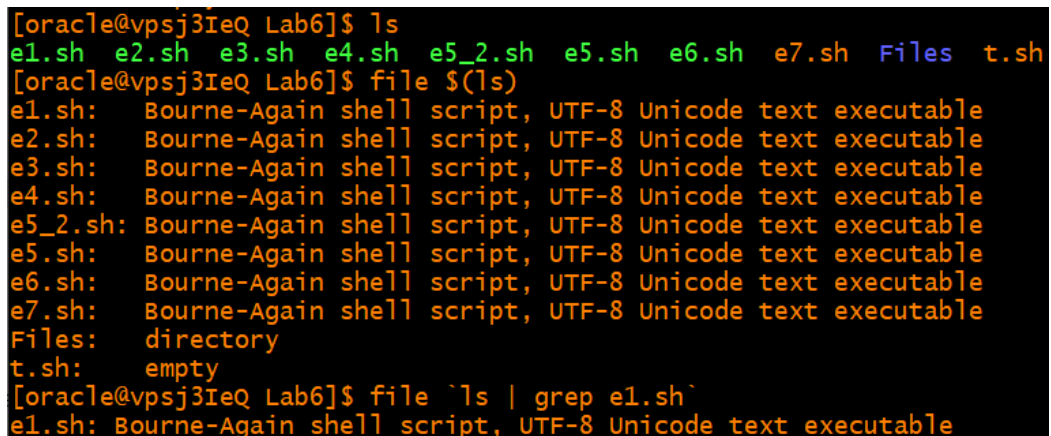
1) `команда1 $(команда2)`

2) `команда1 `команда2``

На рисунку 5 наведено приклади таких синтаксичних конструкцій, де результат команди *ls* передається на вхід команди *file*:

```
file $(ls)
```

```
file `ls | grep e1.sh`
```



```
[oracle@vpsj3IeQ Lab6]$ ls
e1.sh e2.sh e3.sh e4.sh e5_2.sh e5.sh e6.sh e7.sh Files t.sh
[oracle@vpsj3IeQ Lab6]$ file $(ls)
e1.sh: Bourne-Again shell script, UTF-8 Unicode text executable
e2.sh: Bourne-Again shell script, UTF-8 Unicode text executable
e3.sh: Bourne-Again shell script, UTF-8 Unicode text executable
e4.sh: Bourne-Again shell script, UTF-8 Unicode text executable
e5_2.sh: Bourne-Again shell script, UTF-8 Unicode text executable
e5.sh: Bourne-Again shell script, UTF-8 Unicode text executable
e6.sh: Bourne-Again shell script, UTF-8 Unicode text executable
e7.sh: Bourne-Again shell script, UTF-8 Unicode text executable
Files: directory
t.sh: empty
[oracle@vpsj3IeQ Lab6]$ file `ls | grep e1.sh`
e1.sh: Bourne-Again shell script, UTF-8 Unicode text executable
```

Рис. 5 – Приклади особливої конвейєризації команд

У скрипт-файлах також можна ініціалізувати змінні, використовуючи символ `$` та круглі дужки `()`.

У оболонці *Bash* є два види масивів: індексовані масиви (*indexed arrays*) та асоціативні масиви (*Associative Arrays* або *hash-масиви*). Для зберігання малої кількості значень використовуються класичні індексовані масиви.

Масиви можна ініціалізувати, використовуючи круглі дужки `()`.

Якщо для доступу до звичайних змінних можна вказувати лише `$змінна`, для доступу до елементів масиву вже необхідно вказувати фігурні дужки `{ }`.

На рисунку 6 наведено приклад роботи з масивом, який ініціалізується результатом виконання роботи команди *ls*


```
#!/bin/bash
# ініціалізація масиву значеннями результату команди ls
mas=$(ls)
# вивести значення 1-го та 2-го елементу масиву
echo "${mas[0]},${mas[1]}"
```

(a) приклад використання масиву

```
[oracle@vpsj3IeQ Lab6]$ ls
e1.sh e2.sh e3.sh
[oracle@vpsj3IeQ Lab6]$ ./e3.sh
e1.sh,e2.sh
[oracle@vpsj3IeQ Lab6]$ |
```

(b) результат роботи файл-скрипту

Рис. 6 - Приклади роботи з масивом

Для інтерактивного введення даних у скрипт-файл використовується команда:

read змінна

На рисунку 7 наведено приклад використання команди *read*:

- в команді *echo* використовується опція *-n*, щоб не переводити курсор на новий рядок;
- значення змінної *account_record* формується як результат виконання команди *grep*, яка шукає значення змінної *\$my_login* серед рядків файлу облікових записів */etc/passwd*

```
#!/bin/bash
echo -n 'Введіть обліковий запис користувача='
read my_login
# отримання рядка з описом облікового запису
account_record=$(grep $my_login /etc/passwd)
echo "$account_record"
```

Рис. 7 – Приклад використання команди *read*

1.1.5 Використання керуючих структур

Практично будь-яка програма вимагає виконання різних наборів команд в залежності від умов. Інтерпретатор *Bash* оснащений спеціальним оператором *if/then*, який приймає вираз і перетворювати його результат в логічне значення «істина» або «неправда». Якщо результатом перевірки є «істина», оператор *if* виконує команди, які в ньому містяться.

Хоча в багатьох високорівневих мовах програмування істинному твердженню відповідає число «1», а помилковому «0», інтерпретатор *Bash* поводить інакше. При перевірці вхідного значення або вираження істинності відповідає число «0», а неправді - «1». Це пов'язано з тим, що результат виразу насправді не перетворюється до логічного типу, він є кодом повернення програми. В *Unix* за успішний код завершення програми прийнято вважати число нуль, а за помилковий відповідно одиницю.

У загальному вигляді конструкцію застосування умовного оператора можна представити як показано на рисунку 11.

```
if перевіряємий_вираз
then набір_команд
fi
```

(а) варіант 1

```
if перевіряємий_вираз; then набір_команд
fi
```

(б) варіант 2

Рис. 8 – Приклади застосування умовного оператора

Як видно з рисунку 8, конструкція починається з ключового слова *if*, за яким розміщено вираз, що вимагає перевірки. Після перевірки повинно розташовуватися ключове слово *then*, що свідчить про початок блоку команд. Якщо перевірка вхідних параметрів розташовується на одній сходинці з *then*, після неї необхідно додати крапку з комою, щоб відокремитися від нової команди. Кінець блоку команд повинен бути завершений ключовим словом *fi*.

При використанні умовного оператора *if / then* можна задати блок команд, який буде виконаний в разі хибності висловлювання. *Bash* дозволяє зробити це за рахунок ключового слова *else*, розміщений між *then* та *fi*. Всі команди, що лежать між *else* та *fi*, будуть вважатися альтернативою істинності умови.

В *Unix*-подібних ОС існує команда *test* - утиліта для перевірки типу файлу і порівняння значень, яка повертає код 0 (істина) або 1 (неправда) в залежності від обчислення виразу-параметра. Приклад використання команди:

```
test [expr]
```

Вирази можуть бути як унарними, так і бінарними. Унарні вирази часто використовуються для перевірки статусу файлу. Також припустиме порівняння чисел і рядків.

Усередині виразу умовного оператора *if* можуть оброблятися одне або кілька значень. Наприклад, перевірка існування файлу вимагає вказівки одного значення - імені файлу. При цьому оператор перевірки записується спочатку. У разі порівняння двох параметрів оператори перевірок записуються між ними.

В таблиці 1 наведено список припустимих порівнянь.

Таблиця 1 – Приклади операторів для перевірки значень в умовах

Оператор	Опис
Порівняння чисел	
<i>-eq</i> або <i>=</i>	Дорівнює
<i>-ge</i> або <i>>=</i>	Більше або дорівнює
<i>-ne</i> або <i>!=</i>	Не дорівнює
<i>-gt</i> або <i>></i>	Більше
<i>-lt</i> або <i><</i>	Менше
<i>-le</i> або <i><=</i>	Менше або дорівнює
Порівняння рядків	
<i>=</i>	Рівність рядків
<i>!=</i>	Нерівність рядків
<i><</i>	Менше за ASCII-кодом
<i>></i>	Більше за ASCII-кодом
<i>-n</i>	У рядку більше нуля символів (рядок не порожній)
<i>-z</i>	У рядку нуль символів (рядок порожній)
Перевірка файлів	
<i>-e</i>	Файл існує
<i>-s</i>	Файл існує та має розмір більше 0
<i>-f</i>	Файл існує та є звичайним файлом (не каталогом)
<i>-d</i>	Файл існує та є каталогом
<i>-L</i>	Файл існує та є символічним зв'язком
<i>-r</i>	Файл доступний для читання для власника
<i>-w</i>	Файл доступний для модифікації для власника
<i>-x</i>	Файл є виконуваним для власника

В таблиці 2 наведено три варіанти використання виразів

Таблиця 2 – Засоби перевірки умов у виразах

Вираз	Опис
[вираз]	Одинарні квадратні дужки є псевдонімом вбудованої команди <i>test</i> для порівняння значення один з одним, а також перевірки файлів і каталогів на існування при вказуванні їх імен. Вираз повинен бути відокремлений від дужок прогалинами!!!
[[вираз]]	Використання подвійних квадратних дужок в умовному операторі <i>if</i> краще попереднього варіанту, бо тут не використовується псевдонім зовнішньої утиліти <i>test</i> , а використовується вбудована команда мови програмування інтерпретатора <i>Bash</i> , яка працює швидше. Також подвійні квадратні дужки дозволяють використовувати логічні оператори <i>&&</i> або <i> </i> для об'єднання декількох перевірок в одну, використовуючи логічне <i>AND</i> або <i>OR</i> . Вираз повинен бути відокремлений від дужок прогалинами!!!
((арифм_етична_операція))	Результат арифметичної операції також приймається за логічне значення. В середині подвійних круглих дужок можна використовувати оператори <i>"</i> , <i>"</i> , <i><=</i> , <i>></i> , <i>=</i> , <i>!=</i> для порівняння чисел. Операція повинна бути відокремлена від дужок прогалинами!!!

На рисунку 9 наведено приклад використання операторів в умовах.

```
#!/bin/bash
# використання test як утиліти командного рядку
if test -s e1.sh
then
    echo 'файл e1.sh існує'
else
    echo 'файл e1.sh не існує'
fi
# виклики утиліти test як псевдоніму
if [ -f e2.sh ]
then
    echo 'e2.sh - це звичайний файл'
    if test -x e2.sh
    then
        echo 'файл доступний для виконання'
    fi
fi
if test -d Files && test -w Files
then
    echo 'Files - це каталог, доступний для модифікації'
fi
```

(а) приклад використання утиліти *test*

Рис. 9 - Приклади використання утиліти *test* для перевірки значень в умовах

На рисунку 10 наведено приклад використання операторів для перевірки значень в умовах, еквівалентних команді *test*. Остання умова містить перевірку двох умов: файл є файлом-каталогом та файл доступний для модифікації.

```
#!/bin/bash
if [[ -s e1.sh ]]
then
    echo 'файл e1.sh існує'
else
    echo 'файл e1.sh не існує'
fi
if [[ -f e2.sh ]]
then
    echo 'e2.sh - це звичайний файл'
    if [[ -x e2.sh ]]
    then
        echo 'файл доступний для виконання'
    fi
fi
if [[ -d Files && -w Files ]]
then
    echo 'Files - це каталог, доступний для модифікації'
fi
```

Рис. 10 - Приклади використання операторів для перевірки значень в умовах

Також ще раз необхідно зазначити про наявність парних квадратних дужок при описі умов, інакше при виконанні інтерпретатор буде повідомляти про помилку, наприклад, *command not found*.

Для порівняння значення змінної з шаблоном регулярного виразу використовується символ `=~`

На рисунку 11 наведено приклад використання регулярного виразу в операторі *if*

```
#!/bin/bash
file_name='aaaa111'
echo "довжина назви=${#file_name}"
if [[ $file_name =~ [0-9]{3,} ]]; then
    echo "три та більше цифр підряд"
fi
if [[ $file_name =~ [0-9]{4} ]]; then
    echo "чотири цифр підряд"
fi
```

Рис. 11 - Приклад використання регулярного виразу в операторі *if*

При великій кількості альтернативних варіантів порівнянь, розташованих в невеликій частини скрипт-файлу, більш зручним рішенням може стати оператор множинного вибору *case*. Оператору *case* можна передати всього один параметр, який по черзі порівнюється з

підготовленими шаблонами. Якщо порівняння істинно, виконується блок коду, що лежить під шаблоном. Виконання оператора *case* завершиться при відпрацюванні блоку коду, або проходженні всіх шаблонів до кінця і відсутності збігів.

Правила оформлення оператора *case*:

- 1) конструкція множинного вибору починається з ключового слова *case*;
- 2) відразу за ключовим словом *case* визначається вхідний параметр, найчастіше це значення змінної;
- 3) початок визначення блоку порівнянь відкривається ключовим словом *in*, розташованим за вхідним параметром;
- 4) кожне порівняння задається шаблоном, який може бути рядком або числом, а також містити механізми підстановки, та в обов'язковому порядку закривається правою круглою дужкою;
- 5) відразу за правою круглою дужкою шаблону або на наступних рядках після нього розташовуються команди, що вимагають виконання при збігу вхідного параметра, при чому, остання команда всередині такого блоку повинна завершуватися двома точками з комою ;;
- 6) якщо потрібно призначити кілька шаблонів для одного і того ж блоку команд, використовується символ | для їх поділу, наприклад, «*.txt / *.bat)», як у регулярних виразах;
- 7) блок порівнянь повинен завершуватися ключовим словом *esac*.

На рисунку 12 представлено приклад використання оператора *case*.

```
#!/bin/bash
echo -n 'Введіть назву файлу: '
read file_name
case $file_name in
    *.sh)
        echo 'файл-скрипт Bash';;
    *.html)
        echo 'HTML-файл';;
    *.txt)
        echo 'текстовий файл';;
    *)
        echo 'Тип файлу не визначено';;
esac
```

Рис. 12 - Приклад використання оператора *case*

У *Bash* існує спеціальна конструкція, що дозволяє перебирати послідовності значень і обробляти кожні з них окремо - цикл *for*.

Правила складання звичайного циклу *for*:

- 1) конструкція починається з ключового слова *for*, за яким слід блок ініціалізації циклу;
- 2) блок ініціалізації відкривається визначенням змінної, в яку будуть записуватися значення поточних елементів на кожному етапі виконання;

3) після визначення змінної розміщується ключове слово *in*, яке є роздільником між змінної і списку елементів;

4) за ключовим словом *in* визначається список, призначений для перебору в циклі, який може бути масивом, простим рядом чисел або рядків, механізмом підстановки, що формують послідовності;

5) блок команд, який буде виконуватися для кожного елемента, задається між ключовими словами *do* і *done*, де *do* - початок, а *done* – кінець;

6) якщо ключове слово *do* розташовується на одному рядку з визначенням списку, перед ним повинна знаходитися крапка з комою.

На рисунку 13 наведено декілька прикладів використання циклу *for*

```
#!/bin/bash
#проста послідовність чисел і тексту
#все в один рядок, що ефективно для виклику у звичайному командному рядку
for value in 'one' 'two' 3 4; do echo $value; done

#в окремих рядках, що є ефективним для файлу-скрипту
for value in 'one' 'two' 3 4
do
    echo $value
done

#перегляд значень масиву в циклі
cities=('Одеса' 'Львів' 'Київ')
for city in ${cities[@]}
do
    echo $city
done

#використання підстановочного механізму
for letter in {a..z}
do
    echo $letter
done

#перегляд результатів виконання зовнішньої команди
for file in $(ls -l)
do
    echo $file
done
```

Рис. 13 - Приклади використання циклу *for*

У багатьох мовах програмування цикл *for* має іншу поведінку, яка дозволяє заздалегідь визначати кількість виконань блоку команд. *Bash* також підтримує цей альтернативний синтаксис. Як правило, такого роду циклом передається один параметр цілого типу. Він змінюється на кожному етапі виконання і порівнюється з граничним значенням. При досягненні порогу цикл завершує свою роботу.

На рисунку 14 наведено приклади використання циклу з лічильником.

```
#!/bin/bash
# цикл з лічильником
for ((i=1; i<=15 ; i++))
do
    echo $i
done
```

Рис. 14 - Приклади використання циклу з лічильником

Принцип роботи циклу *while* схожий з умовним оператором *if*: приймає на вході вираз, результат якого перетворюється в логічне значення. Якщо вираз є істинним, то цикл *while* продовжує виконувати блок команд. *Bash* дозволяє використовувати для циклу *while* ті ж самі перевірки та вирази, що і в умовному операторі *if*.

На рисунку 15 наведено приклади використання циклу *while*.

В останньому прикладі рисунку 15 показано процес зчитування змісту файлу, як потоку виводу з результату попередньої команди конвеєра.

```
#!/bin/bash
#виконуємо поки лічильник менше за порогове значення
counter=0
while [[ $counter -le 20 ]]
do
    echo $counter
    (( counter++ ))
done

#якщо файл існує виконуємо ще 10 перевірок
#потім видаляємо його
filename='/tmp/test.txt'
checks=0
touch $filename
while [[ -f $filename ]]
do
    if [[ checks -lt 10 ]]
    then
        echo "Файл існує. Перевірка #${checks} + 1"
        (( checks++ ))
    else
        rm $filename
    fi
done

#перебираємо результат команди, використовуючи конвеєр
ls -l | while read file
do
    echo $file
done
```

Рис. 15 - Приклади використання циклу *while*

На рисунку 16 наведено ще один приклад *Bash*-скрипту, який пропонує користувачу ввести назву каталогу і лише після успішного введення назви виводить список файлів каталогу

```
#!/bin/bash

while [[ "$directory" = "" ]]; do
    #виведення запрошення на введення (без переходу на новий рядок)
    echo -n "Введіть каталог: "
    # введення каталогу
    read directory
    # перевірка наявності значення змінної
    if [[ -n "$directory" ]]; then
        # перевірка наявності каталогу
        if [[ -d $directory ]]; then
            ls -l $directory
        else
            echo "Каталог $directory є відсутнім!"
        fi
    fi
done
```

Рис. 16 – Приклад файл-скрипту

На рисунку 17 наведено приклад обробки текстового файлу */etc/passwd* як еквівалент виконання утиліти *AWK* з виведення на екран окремих колонок рядків з символом-роздільником двокрапка.

```
# зчитування рядків CSV-файлу
while read line ;
do
    IFS=":" # визначення роздільника між колонками рядків файлу
    set -- $line # очищення та визначення змінних $1, $2, ... як колонок рядка
    echo $1
done < /etc/passwd
```

Рис. 17 – Приклад файл-скрипту з обробки текстового файлу */etc/passwd* як еквівалент виконання утиліти *AWK*

1.2 Особливості віртуальної файлової системи

Раніше вже визначалося, що файлові системи *Unix*-подібних ОС активно використовують віртуалізацію. Наприклад віртуальна файлова система ***Procfs*** дозволяє отримати доступ до інформації з ядра ОС про системні процеси. Зазвичай вона розташовується у каталозі */proc*. *Procfs* створює дворівневе уявлення просторів процесів. На верхньому рівні процеси представляють собою директорії, іменовані відповідно до їх ідентифікаторів.

У каталозі також розміщено віртуальні файли з описом різних апаратних компонент комп'ютера та ОС, наприклад:

– пам'ять - */proc/meminfo*

- процесор - */proc/cpuinfo*
- файл підкачки - */proc/swaps*
- версія ядра ОС - */proc/version*

1.2.1 Аналіз пам'яті у файлі */proc/meminfo*

Стан пам'яті представлено в файлі наступними важливими параметрами:

- *MemTotal* - доступний обсяг пам'яті, при цьому частина фізично доступної пам'яті резервується під час запуску ОС і не входить в зазначений тут обсяг;
- *MemFree* - невикористаний обсяг пам'яті і доступний для негайного виділення процесам;
- *MemAvailable* – доступний обсяг пам'яті без врахування розділів або файлів підкачки (*swap*);
- *Buffers* - область пам'яті, зайнята зберіганням даних, які очікують операції запису на диск, при цьому буфер дозволяє програмам продовжувати виконання свого завдання, не чекаючи моменту, коли дані будуть фізично записані на диск;
- *Cached* - обсяг зайнято під кеш читання сторінок з диску (файли, директорії, файли блокових пристроїв, дані, які стосуються механізму *IPC*, дані процесів рівня користувача, скинутих в область підкачки), але не включає в себе *SwapCached*;
- *SwapCached* - об'єм пам'яті, який одного разу був поміщений в область підкачки *swap*, але потім перенесений назад у пам'ять, однак дані все ще присутні в *swap*, і при необхідності цей обсяг пам'яті може бути знову звільнений без витрачання ресурсів на операції введення/виводу.
- *Active* - об'єм пам'яті, зайнятий найбільш часто використовуваними сторінками пам'яті, коли ці сторінки пам'яті активно використовуються процесами і звільнятимуться тільки в разі крайньої необхідності;
- *SwapTotal* - загальний обсяг області підкачки (як в розділі підкачки, так і в окремих *swap*-файлах, якщо вони використовуються);
- *SwapFree* – вільний об'єм в області підкачки (як в розділі підкачки, так і в окремих *swap*-файлах, якщо вони використовуються);
- *VmallocTotal* – загальний розмір віртуальної пам'яті.

На рисунку 18 представлено результат перегляду змісту файлу */proc/meminfo* зі всіма параметрами пам'яті.

MemTotal:	1006936 kB	KernelStack:	5824 kB
MemFree:	72744 kB	PageTables:	23804 kB
MemAvailable:	455820 kB	NFS_Unstable:	0 kB
Buffers:	181184 kB	Bounce:	0 kB
Cached:	302056 kB	WritebackTmp:	0 kB
SwapCached:	3184 kB	CommitLimit:	2599592 kB
Active:	438388 kB	Committed_AS:	1558116 kB
Inactive:	227748 kB	VmallocTotal:	34359738367 kB
Active(anon):	110988 kB	VmallocUsed:	0 kB
Inactive(anon):	101156 kB	VmallocChunk:	0 kB
Active(file):	327400 kB	HardwareCorrupted:	0 kB
Inactive(file):	126592 kB	AnonHugePages:	28672 kB
Unevictable:	3652 kB	CmaTotal:	0 kB
Mlocked:	3652 kB	CmaFree:	0 kB
SwapTotal:	2096124 kB	HugePages_Total:	0
SwapFree:	1956808 kB	HugePages_Free:	0
Dirty:	124 kB	HugePages_Rsvd:	0
Writeback:	0 kB	HugePages_Surp:	0
AnonPages:	184548 kB	Hugepagesize:	2048 kB
Mapped:	50732 kB	DirectMap4k:	118720 kB
Shmem:	26824 kB	DirectMap2M:	929792 kB
Slab:	163204 kB		
SReclaimable:	112996 kB		
SUnreclaim:	50208 kB		

Рис. 18 – Приклад змісту файлу */proc/meminfo*

1.2.2 Аналіз характеристик процесора у файлі */proc/cpuinfo*

Характеристики процесора представлено такими важливими параметрами:

- *vendor_id*
- *cpu family*
- *model*
- *model name*
- *cpu MHz*
- *cache size*
- *cpu cores*
- *cpuid level*
- *bogomips*
- *clflush size*

На рисунку 19 представлено результат перегляду змісту файлу */proc/cpuinfo* зі всіма характеристиками процесора.

```

processor      : 0
vendor_id     : GenuineIntel
cpu family    : 6
model         : 62
model name    : Intel(R) Xeon(R) CPU E5-2695 v2 @ 2.40GHz
stepping      : 4
microcode     : 0x1
cpu MHz       : 2399.998
cache size    : 30720 KB
physical id   : 0
siblings      : 1
core id       : 0
cpu cores     : 1
apicid        : 0
initial apicid : 0
fpu           : yes
fpu_exception : yes
cpuid level   : 13
wp            : yes
flags         : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca
m constant_tsc arch_perfmon rep_good nopl xtopology eagerfpu pni pclmulqdq
e avx f16c rdrand hypervisor lahf_lm ssbd ibrs ibpb stibp fsgsbase tsc_adju
bogomips      : 4799.99
clflush size  : 64
cache_alignment : 64
address sizes  : 46 bits physical, 48 bits virtual
power management:

```

Рис. 19 – Приклад змісту файлу */proc/cpuinfo*

1.2.3 Аналіз характеристик файлу підкачки в файлі */proc/swaps*

Стан файлу підкачки представлено наступними важливими параметрами:

- *Filename* – шлях до файлу підкачки;
- *Size* – загальний розмір у байтах;
- *Used* – загальний розмір використаної пам'яті.

На рисунку 20 представлено приклад змісту файлу */proc/swaps*

Filename	Type	Size	Used	Priority
/swapfile	file	4194300	1282048	-2

Рис. 20 – Приклад змісту файлу */proc/swaps*

1.2.4 Аналіз характеристик ядра ОС у файлі */proc/version*

Характеристики встановленої ОС *Linux* представлено параметрами:

- *Linux version* - версія ядра;
- *gcc version* - версія компілятора мови програмування C.

На рисунку 21 представлено приклад змісту файлу */proc/version*

```

Linux version 3.10.0-1127.19.1.el7.x86_64 (mockbuild@kbuilder.bsys.centos.org) (gcc version 4.8.5 20150623
(Red Hat 4.8.5-39) (GCC) ) #1 SMP Tue Aug 25 17:23:54 UTC 2020

```

Рис. 21 – Приклад змісту файлу */proc/version*

2 Завдання до виконання

Виконати наступні дії з підготовки до виконання завдань роботи:

- 1) встановити з'єднання з віддаленим *Linux*-сервером з IP-адресою = 46.175.148.116, логіном, наданим вам викладачем, та паролем, зміненим вами у попередній роботі;
- 2) перейти до каталогу *Git*-репозиторія;
- 3) створити нову *Git*-гілку з назвою «*Laboratory-work-6*»;
- 4) перейти до роботи зі створеною гілкою;
- 5) створити каталог з назвою «*Laboratory-work-6*»;
- 6) перейти до каталогу «*Laboratory-work-6*»;
- 7) в каталозі «*Laboratory-work-6*» створити файл *README.md* та додати до файлу рядок тексту із темою лабораторної роботи «*Основи використання скриптової мови інтерпретатору оболонки командного рядку Unix-подібних ОС*» як заголовок 2-го рівня *Markdown*-форматування;
- 8) виконати операції з оновлення *GitHub*-репозиторію змінами *Git*-репозиторія через послідовність *Git*-команд *add*, *commit* із коментарем «*Changed by Local Git*» та *push*;
- 9) на веб-сервісі *GitHub* перейти до створеної гілки «*Laboratory-work-6*»;
- 10) перейти до каталогу «*Laboratory-work-6*» та розпочати процес редагування файлу *README.md* ;
- 11) в подальшому за результатами рішень кожного наступного розділу завдань до файлу *README.md* додавати рядки як заголовки 3-го рівня *Markdown*-форматування з назвами розділу, а також знімки екрані та підписи до знімків екранів з описом пунктів завдань.

2.1 Складна обробка текстового файлу утилітою *AWK*

2.1.1 Використовуючи утиліту *AWK* та відповідні команди одного командного рядку, вивести на екран 5-тий стовпчик рядків файлу */etc/passwd*, враховуючи наступні умови:

- обробляються номери рядків не більше значення вашого варіанту + 100;
- значення 1-го стовпчику починається з символу, який співпадає з 1-ю латинською літерою вашого прізвища.

2.1.2 Повторити рішення попереднього завдання, оформивши команди *AWK* в окремому скрипт-файлі, який повинен мати права на виконання.

2.2 Робота з віртуальною файловою системою

2.2.1 Створити скрипт-файл на мові програмування *Bash* з назвою за шаблоном *Прізвище транслітерацією + OSParam*, наприклад, *BlazhkoOSParam.sh*, який виводить на екран лише окремі дані про параметри поточного стану ОС з віртуальної файлової системи *Procfs* у відповідності із варіантом, представленим у стовпчику «Параметр пам'яті або процесору» таблиці 3.

Під час виконання скрипт-файл повинен:

- 1) отримати назву параметру через параметр командному рядку, наприклад, *BlazhkoOSParam.sh parl*, де *parl* – назва параметру;
- 2) знайти значення параметру у відповідному за варіантом файлі;
- 3) вивести на екран значення параметру;
- 4) якщо параметр не знайдено, повідомити про помилку, наприклад: «параметр не знайдено»;
- 5) якщо в командному рядку додається ще один параметр з назвою *info*, наприклад, *BlazhkoOSParam.sh parl info*, тоді додатково вивести на екран опис призначення параметру українською мовою.

2.2.2 Для перевірки правильності роботи скрипт-файлу необхідно його протестувати:

- з одним параметром;
- із двома параметрами.

2.3 Інтерактивне керування файлами

2.3.1 Створити скрипт-файл на мові програмування *Bash*, який виконує дії у відповідності з варіантом, представленим у стовпчику «Назва скрипту-файлу» таблиці 3:

- *my_create_empty_file* – створити порожній файл (команда *touch файл*);
- *my_create_file* – створити файл через перенаправлення (команда *echo "" > файл*);
- *my_create_slink* – створити символічний зв'язок (команда *ln -s файл файл-зв'язок*);
- *my_create_hlink* – створити жорсткий зв'язок (команда *ln файл файл-зв'язок*);
- *my_create_directory* – створити каталог (команда *mkdir каталог*);
- *my_change_directory* – змінити назву каталогу (команда *mv каталог1 каталог2*);
- *my_change_file* – змінити назву файлу (команда *mv файл1 файл2*);
- *my_delete_file* – видалити файл (команда *rm файл*);
- *my_delete_link* – видалити файл-зв'язок (команда *rm файл*);
- *my_delete_directory* – видалити каталог (команда *rmdir каталог*)

Таблиця 3 – Варіанти завдань зі створення програм

№ варіанту	Параметр пам'яті/процесору	Назва скрипту-файлу	Обмеження на довжину	Обмеження на зміст назви
1	<i>MemTotal</i>	<i>my_create_empty_file</i>	≤ 5	≤ 2 цифри підряд
2	<i>MemFree</i>	<i>my_create_file</i>	≤ 6	≤ 3 цифри підряд
3	<i>MemAvailable</i>	<i>my_create_slink</i>	≤ 7	≤ 4 цифри підряд
4	<i>Buffers</i>	<i>my_create_hlink</i>	≤ 8	≤ 5 цифр підряд
5	<i>Cached</i>	<i>my_create_directory</i>	≤ 9	≤ 6 цифр підряд
6	<i>SwapCached</i>	<i>my_change_directory</i>	≤ 10	≤ 7 цифр підряд
7	<i>Active</i>	<i>my_change_file</i>	≤ 11	≤ 8 цифр підряд
8	<i>SwapTotal</i>	<i>my_delete_file</i>	≤ 12	≤ 9 цифр підряд
9	<i>SwapFree</i>	<i>my_delete_link</i>	≤ 13	≤ 10 цифр підряд
10	<i>VmallocTotal</i>	<i>my_delete_directory</i>	≤ 14	≤ 11 цифр підряд
11	<i>vendor_id</i>	<i>my_create_empty_file</i>	≤ 15	≤ 12 цифр підряд
12	<i>cpu family</i>	<i>my_create_file</i>	≤ 16	≤ 13 цифр підряд
13	<i>model</i>	<i>my_create_slink</i>	≤ 17	≤ 14 цифр підряд
14	<i>model name</i>	<i>my_create_hlink</i>	≤ 18	≤ 15 цифр підряд
15	<i>cpu MHz</i>	<i>my_create_directory</i>	> 4	> 1 цифри підряд
16	<i>cache size</i>	<i>my_change_directory</i>	> 5	> 2 цифри підряд
17	<i>cpu cores</i>	<i>my_change_file</i>	> 6	> 3 цифри підряд
18	<i>cpuid level</i>	<i>my_delete_file</i>	> 7	> 4 цифри підряд
19	<i>bogomips</i>	<i>my_delete_link</i>	> 8	> 5 цифр підряд
20	<i>clflush size</i>	<i>my_delete_directory</i>	> 9	> 6 цифр підряд
21	<i>MemTotal</i>	<i>my_create_empty_file</i>	> 10	> 7 цифр підряд
22	<i>MemFree</i>	<i>my_create_file</i>	> 11	> 8 цифр підряд
23	<i>MemAvailable</i>	<i>my_create_slink</i>	> 12	> 9 цифр підряд
24	<i>Buffers</i>	<i>my_create_hlink</i>	> 13	> 10 цифр підряд
25	<i>Cached</i>	<i>my_create_directory</i>	> 14	> 11 цифр підряд
26	<i>SwapCached</i>	<i>my_change_directory</i>	> 15	> 12 цифр підряд
27	<i>Active</i>	<i>my_change_file</i>	> 16	> 13 цифр підряд
28	<i>SwapTotal</i>	<i>my_delete_file</i>	> 17	> 14 цифр підряд
29	<i>SwapFree</i>	<i>my_delete_link</i>	> 18	> 15 цифр підряд

Для всіх варіантів передбачається наступний опис кроків алгоритму роботи програми:

1) запропонувати користувачу ввести назву об'єкту, використовуючи українське запрошення;

2) перевірити у файловій підсистемі присутність об'єкту з такою назвою (для команд видалення або зміни) або відсутність об'єкту з такою назвою (для команд створення), використовуючи відповідні команди та файли;

3) якщо об'єкт відсутній (для команд видалення або зміни) або присутній (для команд створення), тоді вивести на екран відповідне повідомлення про помилку та завершити роботу програми;

4) якщо назва об'єкту не відповідає вказаному обмеженню, тоді вивести на екран відповідне повідомлення про помилку та завершити роботу програми;

5) якщо назва об'єкту відповідає вказаному обмеженню (стовпчики «Обмеження на довжину» та «Обмеження на зміст назви»), виконати відповідну команду.

У кроках алгоритму об'єктом може виступати звичайний файл, файл-зв'язок або каталог, в залежності від варіанту завдання.

2.3.2 Для перевірки правильності роботи скрипт-файлу необхідно його протестувати:

– для команд видалення або зміни – попередньо створити об'єкт, виконати скрипт-файл, а потім ще раз його виконати, щоб створити помилку;

– для команд створення – попередньо видалити об'єкт, виконати скрипт-файл, а потім ще раз його виконати, щоб створити помилку.

2.4 Підготовка процесу *Code Review* для надання рішень завдань лабораторної роботи на перевірку викладачем

2.4.1 На веб-сервісі *GitHub* зафіксувати зміни у файлі *README.md*

2.4.2 Скопіювати файли, які було створено у попередніх розділах завдань, в каталог «*Laboratory-work-6*» *Git*-репозиторію.

2.4.3 Оновити *Git*-репозиторій змінами нової гілки «*Laboratory-work-6*» з *GitHub*-репозиторію.

2.4.4 Оновити *GitHub*-репозиторій змінами нової гілки «*Laboratory-work-6*» *Git*-репозиторію.

2.4.5 Виконати запит *Pull Request*.

Примітка: Увага! Не натискайте кнопку «Merge pull request»!

Це повинен зробити лише рецензент, який є вашим викладачем!

Коли рецензент-викладач перегляне ваше рішення він виконає злиття нової гілки та основної гілки, натиснувши кнопку «*Merge pull request*». Якщо рецензент знайде помилки, він повідомить про це у коментарях, які з'являться на сторінці *Pull request*.

2.5 Оцінка результатів виконання завдань лабораторної роботи

Оцінка	Умови
+3 бали	1) всі рішення відповідають завданням 2) <i>Pull Request</i> представлено не пізніше найближчої консультації після офіційного заняття із захисту лабораторної роботи
-0.5 балів за кожну помилку	в рішенні є помилка, про яку вказано в <i>Code Review</i>
-1 бал	<i>Pull Request</i> представлено пізніше дати найближчої консультації після офіційного заняття із захисту лабораторної роботи за кожний тиждень запізнення
+2 бали	1) ви увійшли до трійки перших з вашої групи, хто без помилок виконав усі завдання; 2) отримано правильну відповідь на два запитання, які стосуються призначення команд, представлених на знімках екранів