Міністерство освіти і науки України
ДВНЗ “Прикарпатський національний університет імені Василя Стефаника”
Кафедра комп’ютерної інженерії та електроніки

Методичний посібник
до виконання лабораторних робіт з дисципліни
“СИСТЕМНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ”

Затверджено
на засіданні кафедри радіофізики і електроніки
протокол № 4 від 8 листопада 2017 р

Розроблено:
доц. Голота В.І.

Івано-Франківськ 2017

Зміст

Лабораторна робота № 1

1. Команди Linux і командного інтерпретатора Bash

Мета роботи: вивчення команд Linux і командного інтерпретатора bash.
Теоретичний матеріал лекції, література [1-7].

1.1. Короткі теоретичні відомості про команди Linux

Команди Linux використовуються у наступних випадках:
- у всіх випадках коли не спрацьовують функції графічного інтерфейсу;
- при віддаленому адмініструванні серверів;
- для реалізації функцій, які не забезпечує графічне середовище;
- якщо не стартує або пошкоджене графічне середовище X window.

1.2. Команди Linux впорядковані за алфавітним порядком

(аналогічні команди вбудовані в оболонку `Bash-4.2` підкреслені)

alias - створення синонімів (alias) команд
apropos - пошук заданого слова в описаннях команд
awk (gawk) - потоковий редактор з вбудованою мовою програмування
bg - виконання завдань (jobs) у фоновому режимі
break - вихід з циклу
bind - керування розкладкою клавіатури
builtin - виконання з консолі вбудованих команд оболонки (shall)
bzip2 блоко-орієнтований файловий стискач/розтискач файлів
cal - виведення на екран календаря
caller – повертає контекст любого активного виклику підпрограми
case - виконання команд за умовою (застаріла !)
cat - об'єднати файли або стандартні входи і направити на стандартний вихід
cd - зміна робочого каталогу
cfdisk - керування таблицею розбивки жорсткого диска
chgrp - зміна власника групи
chmod - зміна бітів режиму доступу до файлу
chown - зміна власника і/або групи файлу
chroot - виконання команд і інтерактивної оболонки із із спеціального нового root каталогу
cksum - виведення контрольних CRC сум і кількості байтів
clear - очистка екрану
cmp - побайтове порівняння двох файлів
comm - порядкове порівняння двох файлів
command - виконання Linux команд, ігноруючи вбудовані команди і функції оболонки
compgen – генерація можливих співпадінь для слова відповідно до опцій
complete – описання завершення аргументів для кожного імені
compopt – модифікація опцій завершення
continue - пропуск поточної ітерації
cp - копіювання файлів і каталогів
cron - демон виконання програм за розкладом
crontab - підтримка crontab файлів для індивідуальних користувачів
csplit - розбивка файлу на секції за заданим шаблоном
cut – виведення заданих частин рядків кожного рядка файлу на стандартний вивід
date - виведення або зміна дати і часу
dc - потоковий калькулятор для чисел з фіксованою і плаваючою крапкою
dd - конвертування і копіювання файлів
declare - оголошення змінних і/або визначення їх атрибутів
df - використання файлового дискового простору
diff - порядкове порівняння двох файлів
diff3 - порядкове порівняння трьох файлів

dir - виведення вмісту каталогу
dircolors - символьні імена кольорів які розпізнає Tk
dirname - виділення із повного шляху тільки шляху каталогів
dirs - виведення поточного стеку каталогів
disown – робота з таблицею активних завдань (jobs)
du - довідка про використання файлового простору
echo – виведення на екран аргументів розділених пропуском з добавлення newline
egrep (grep, frep) - друк рядків файлу, які співпадають із заданим шаблоном
eject - вийняти змінний пристрій (CD-ROOM, гнучкий диск)
enable - заборона або дозвіл виконання вбудованих команд оболонки
env - виконання програми у модифікованому середовищі
ethtool - виведення і зміна встановлень ethernet карт
eval - оцінка декількох команд/скриптів. Виконання команд після виконання підставлень/розширень в оболонці
exec - виконання команд. Вихід з оболонки і перемикання на нову програму
exit - вихід з оболонки і перемикання на нову програму
expand - заміна табуляції на пропуски
export – встановлення змінних середовища. Підтримку експорту змінних
expr - оцінювання виразів
false – фальш. Повернення ненульового коду завершення
fc - оброблення списку історії команд
fdformat - низькорівневе форматування гнучких дисків
fdisk - робота з розділами жорсткого диску в Linux
fg - виконання задач у фоновому режимі
file – інформація про тип файлу
find - пошук файлів у ієрархії каталогів, які відповідають заданим критеріям
fmt - простий форматувач тексту
fold - перенесення рядків при досягненні заданої довжини
for – команда циклу
format - форматування символьних рядків у стилі sprintf
free - інформація про використання оперативної пам'яті
fsck - тестування і відновлення файлової системи
ftp - протокол передачі файлів
function - оголошення функції у сценаріях
gawk - потоковий редактор і мова програмування awk
getopts - програма видобування аргументів із списку параметрів командного рядка
grep - пошук у файлах символьних рядків , які співпадають із заданим шаблоном
groups - виведення імен груп в які входить користувач
gzip, gunzip – програми стискання і розтискання файлів
hash - запам’ятовування у змінних середовища повного шляху викликуваних команд
head - виведення на друк початкової частини файла (файлів)
help – довідкова система
history – історія списку команд введених з консолі
hostname – виведення або встановлення імені комп’ютера
id – реальні і ефективні ідентифікатори користувача і групи
if - задання умови виконання команд в сценаріях
import - захоплення вікна X сервера і збереження зображення у файлі
install - копіювання файлів і встановлення атрибутів
jobs - статус завдань у поточній сесії
join - об'єднання рядків двох файлів
kill - завершення процесу
less - поекранне/порядкове виведення вмісту файла з можливістю переміщення вперед/назад
let - виконання операцій над арифметичними виразами
ln - встановлення зв'язків між файлами
local – створення змінних. Оголошення локальних змінних у функціях
logname - виведення імені зареєстрованого користувача
logout - вихід із оболонки. Вихід із оболонки в інтерактивному режимі
look – виведення рядка з файлу за заданим символьним рядком
lpc - програма керування порядковою друкаркою
lpr - друк файлів
lprm - відміна завдань на друк

ls - виведення вмісту каталога
lsof - виведення списку відкритих файлів
make - утиліти перекомпіляції груп програм
man - довідкова система man
mapfile -> readarray
mkdir - створення нового каталогу
mkfifo - створення каналів (іменованих) типу FIFO
mkisofs - створення гібридної файлової системи ISO9660/Joliet/HFS
mknod - створення блоку або файлів спеціальних символів
more - поекранне виведення вмісту файла з можливістю порядкового переміщення вперед
mount - монтування файлової системи
mtools - утиліта роботи з MS-DOS файлами
mv - пересилання і перейменування файлів або каталогів
nc – (netcat) утиліта для встановлення мережевих з'єднань TCP, UDP
netstat - інформація про мережу, мережеві з'єднання, інтерфейсну статистику
nice - встановлення пріоритету команд або завдань
nl - нумерування рядків файлу
nohup - виконання команди нечутливої до сигналів hangups
passwd - зміна пароля користувача
paste - об'єднання рядків файлів
pathchk - перевірка допустимості імені файлу
ping - перевірка мережевого з'єднання
popd – вилучення із стеку каталогів назви поточного каталогу
pr - підготовка файлів до друку
printenv - виведення змінних середовища
printf - форматування і виведення даних. Форматоване виведення даних
ps - інформація поточні процеси
pushd – добавити ім'я каталогу у стек каталогів
pv – pipeview, команда для робот из каналами (pipes)
pwd - виведення абсолютного шляху і назви поточного каталогу
quotactl – встановлення дискових квот
ram - дисковий пристрій типу RAM
read - читання одного рядка із стандартного вводу. Читання із стандартного вводу або файлу
readonly - призначення іменам змінних/функціях атрибуту "тільки для читання".
return – повернення з функцій
rm – вилучення файлів або каталогів
rmdir – вилучення каталогів
rsync - віддалене копіювання файлів
screen - менеджер термінального вікна
scp - віддалене безпечне копіювання файлів
sdiff – об'єднання порядкової різниці двох файлів
sed - потоковий редактор
seq - виведення числових послідовностей
set - встановлення опцій для імен і значень змінних
sftp - програма безпечного пересилання файлів
shift - зсув позиційних параметрів
shopt – перемикання значень змінних які встановлюють властивості оболонки Bash
shutdown - завершення або перевантаження Linux
sleep - затримка на заданий проміжок часу
sort - сортування текстових файлів
source - виконання команд файлу з '.'
split - розбиття файлу на частини фіксованих розмірів
ssh - клієнт безпечної оболонки (програма віддаленої реєстрації в системі)
stat – детальна інформація про файли і файлову систему
strace - трасування системних викликів і сигналів
su - виконання програм з правами інших користувачів
sum - друк контрольних сум файлів і підрахунок кількості блоків у файлах
suspend – затримка виконання оболонки до отримання сигналу SIGCONT
symlink - створення нового символьного імені файлу (нового шляху, який містить старий шлях)

sync - скидання буфера файлової системи (суперблоку) на диск
tail - виведення останньої частини файлу
tar - утиліта створення архіву типу tar
tee – перенаправлення виводу на стандартний вихід і файл (або декілька файлів)
test - оцінка умовних виразів і повернення результати 0 або 1
time - визначення часу виконання команд
times - час системи і користувача. Виведення статистики часу виконання оболонки
touch - - зміна часових атрибутів файлу, створення порожнього файлу
top - виведення діючих процесів Linux
traceroute - трасування маршрутів пакетів до мережевого хоста
trap – виконання команд при обробленні і захоплення сигналів
tr - трансляція або вилучення символів
true - логічне значення істина
tsort - топологічне сортування
tty - виведення імені терміналу на stdin
type – описання команди. Визначення типу імен, при використанні їх як команд
typeset - оголошення змінних
ulimit - встановлення і отримання обмежень для користувача. Встановлення контролю над ресурсами оболонки
umask - встановлення і отримання обмежень для користувача. Задання режиму створення файлів користувачем
umount - розмонтування пристрою
unalias – вилучення аласів. Вилучення імені із списку аліасів
uname - виведення інформації про систему
unexpand - перетворення пропусків у табуляцію
uniq - виведення або пропущення рядків які повторюються у файлах
units – перетворення одиниць вимірювань з однієї шкали в іншу
unset - вилучення змінних або функцій за заданим іменем
unshar - розпакування архівів shell сценаріїв
until – виконання команд (до помилки)
useradd - створення нового облікового запису користувача
usermod - корегування облікового запису користувача
users - виведення імен користувачів, зареєстрованих у системі
uuencode - кодування бінарного файлу
uudecode – декодування файлу створеного uuencode
vdir - виведення вмісту каталогу (’ls -l –b’)
vi – текстовий редактор
wait – очікувати на завершення заданого процесу і повернути його код завершення
watch – виконання або виведення результатів роботи програми періодично
wc - виведення кількості байт, слів і рядків
whereis - пошук всіх появлень команди у файлах
which - пошук програми у шляху користувача
while - виконання сценарію за умовою циклу
who - виведення інформації про зареєстрованих користувачів
whoami - виведення id і імені поточного користувача
wget - неінтерактивне завантаження веб-сторінок або файлів через HTTP, HTTPS, FTP
xargs - побудова і виконання командного рядку із стандартного вводу
yes - виведення у нескінченому циклі символьного рядка (до зняття CTRL+C)
.(крапка) - виконання команд з файлу
- коментар

1.3. 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

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

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

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

Виконання команд групами:

command-1;command-2;command-3

{command-1;command-2} > test.txt

1.4. Команди адміністрування ОС Linux

uname -a - версія ядра Linux

cat /etc/os-release – інформація про версію ОС SuSE

clear - очищення екрану термінала;

date – поточна дати і час

cal -3 – календар попереднього, поточного і наступного місяця

uptime - поточний час та робота системи без перевантаження і виключення

whois linux.org – інформація про домен linux.org;

history – історія команд

history | tail -10 – останні 10 введені команди

!! - виконати останню команду

exit - завершити сеанс поточного користувача

passwd - змінити пароль поточного користувача

shutdown -h now - вихід з Linux

poweroff - вихід з Linux

reboot - перевантаження системи

last reboot - статистика перевантажень

winecfg - налаштування Wine (не емулятор WinAPI)

finger, users, who, w – інформація про користувачів системи

1.5. Команди для роботи з файлами, каталогами і каналами

pwd – вивести поточний шлях

ls – виведення впорядкованого списку каталогів і файлів

ls –las - виведення відсортованого за іменами списку усіх каталогів і файлів

ls –laX - виведення відсортованого за розширеннями списку усіх каталогів і файлів

cd - перейти у домашній каталог

cd /home – перейти в каталог /home

cd $HOME - перейти в home каталог

cd ~ - перейти в home каталог

cd ~user1 - перейти в каталог user1 в каталозі home

cd – - перейти в попередній робочий каталог

cd $OLDPWD перейти в попередній робочий каталог

cd ~/public_html - перейти у каталог public_html в home каталозі

cd .. - перейти в батьківський каталог

cd /usr/bin - перейти в каталог usr/bin з root каталогу

cd /dev; ls –al sda* ttyS* - виведення списку файлів пристроїв

head /var/log/messages - вивести початок файлу

tail /var/log/messages - вивести кінець файлу (для великих файлів)

more file – поекранне виведення файлу з рухом вперед

less file - поекранне виведення файлу з рухом назад

echo "Останній рядок" | sudo tee -a /home/text - добавлення текстового рядка "Останній рядок" у кінець файлу /home/text

cp /home/user1/my.txt /home/new.txt - копіювати файл /home/user1/my.txt у файл home/new.txt
ln /user/file1 /user/file2 - створити жорсткий зв'язок між файлами (hard link) (аналог до cp –l /user/file1 /user/file2)
ln –s /user/file1 /user/file2 - створити м'який зв'язок між файлами (soft link) (аналог до cp –s /user/file1 /user/file2)
mkdir /home/user1/newdir – створення нового каталогу /home/user1/newdir
rmdir /home/dir – вилучити каталог dir
rm -rf /home/user1/dir - вилучити каталог dir з вкладеними каталогами
cp -la /dir1 /dir2 - копіювати каталог dir1 в каталог dir2
mv /dir1 /dir2 - перейменувати каталог dir1 в каталог dir2
du -sh /home/Documents – визначення розміру каталогу
du -sh /home/file – визначення розміру файла
touch /home/newfile – створення порожнього файлу /home/newfile
> /home/newfile – створення порожнього файлу /home/newfile
mv /home/file1 /home/file2 – перейменування файлів
rm /home/file – вилучення файлу
cp /home/file1 /home/file2 – копіювання файлу
dd if=/dev/zero of=/tmp/mynullfile count=1 - створення null-файлу (/dev/zero є спеціальний файл,який генерує null символи, count – лічильник блоків, за замовчуванням блок має 512 байт)
od -vt x1 /tmp/mynullfile - перегляд 8-го дампу файла
dd if=/dev/hda of=mymbrfile bs=512 count=1 – копіювання MBR із завантажувального сектора IDE жорсткого диска
dd if=/dev/cdrom of=whatever.iso - копіювання ISO образа із CD або DVD
vim, kwrite – редагування файлів з використання редакторів
sort – сортування рядків файлу
nl - нумерація рядків файлу
grep – друк рядків файлу, які співпадають з шаблоном
find file – пошук файлів в ієрархії каталогів
cat file – виведення вмісту файлу
stat – виведення параметрів файлу, прав і часу доступу до файлу
file – виведення типу файлу
mkfifo – створення іменованого каналу (черга типу FIFO)
wget http://itshaman.ru/images/logo_white.png - скачати файл http://itshaman.ru/images/logo_white.png у поточну папку
wget --convert-links -r http://www.linux.org/ - копіювати сайт повністю (на 5 рівнів в глибину) і конвертувати посилання для автономної роботи
md5sum openSUSE-10.3-RC1-KDE-i386.iso - перевірка контрольної суми файлу
sha1sum openSUSE-10.3-RC1-KDE-i386.iso - перевірка контрольної суми файлу
zip, bzip2 – створення архівів файлів
find file1 знайти файл у поточному каталозі
find / -name e100 -print 2> /dev/null пошук файлу e100, починаючи з root каталогу і направлення повідомлень про помилки в нульовий пристрій
echo "Команда для роботи з каналом" | pv –qL 5 посимвольне виведення стрічки із швидкістю 5 байт/сек
cat file.txt | pv –s `du –sb file.txt | cut –f1` виведення файлу на екран з візуалізацією прогресу виконання в %.

Приклади використання команд для роботи з файлами:
>**file *** отримати інформацію про типи файлів
>**file /usr/bin/card** отримати інформацію про типи файлі card
>**touch myfile1** створити порожній файл
>**> myfile2** створити порожній файл
>**echo "text" > myfile1** записати дані у файл
>**cat myfile1** вивести вміст файлу на екран
>**rm myfile1** вилучити файл
>**ls –l myfile1** інформація про параметри файлу
>**ls -l** інформація про файли і каталоги в поточному каталозі
>**ls -la** інформація про файли і каталоги включно зі скритими(.)
>**ls -lt** інформація про файли, впорядковані за часом зміни

>**ls -lu** інформація про файли, впорядковані за часом доступу
>**ls -lS** інформація про файли, впорядковані за розміром
>**ls -li** інформація про файли і їх inodes
>**ls -ln** список ID для user/group
>**ls -lh** список розмірів файлів у Кбайт, Мбайт
>**ls -lR** рекурсивний список файлів для поточного каталогу і підкаталогів
>**ln myfile1 myfile1-hardlink** створення м'якого (символьного) посилання на файл або каталог. Символічне посилання може бути на різних розділах диску і має інший inode номер.
>**ln –s myfile2 myfile2-softlink** створення жорсткого посилання на файл (створення різних імен для одного фізичного файлу). Жорстке посилання моє розміщуватися на тому самому розділі, що і фізичний файл.
>**ls –li myfile*** отримання інформації про створені посилання.

Файли пристроїв:

>**ls -l /dev/tty0 /dev/sda1** список символьних і блокових пристроїв
>**mknode /dev/ttyS12 c 4 68** створення пристрою для 4-го послідовного порта
>**mkfifo mypipe** створення іменованих каналів для міжпроцесної взаємодії

Права доступу до файлів і каталогів

Після створення файлу або каталогу *перший символ* у виведенні команди `ls –l` вказує тип файлу:

`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 власника (`rwx`), групи (`rwx`), інших (`rwx`). Для зміни прав доступу служить команда `chmod`. Біти доступу кодуються вісімковими тетрадами:

```
rwx          7   = 421
rw-          6   = 420
r--          4   = 400
rwxrwxrwx – 777   (вісімкові числа)
rw-rw-rw- - 666
r—r--r--  - 444
```

>**chmod 644 myfile** змінити права доступу до файлу
>**chmode –R 644 /tmp/test** рекурсивно змінити права доступу каталогу і всіх його файлів

1.6. Команди робота із стеком

Команди `pushd, popd, dirs` використовуються для маніпуляції порядком каталогів у стеку.

>**pushd /home/user1** записати каталог у стек
>**pushd /home/user2**
>**push -0** помістити останній каталог в стеку на верхівку стеку
>**dirs –v** вивести список каталогів у стеку
>**popd** прочитати каталог із стеку

1.7. Команди для роботи із завданнями, процесами і сигналами

at - запускає програми у визначений час

atq - виводить список завдань, поставлених в чергу командою at
atrm - вилучає завдання з черги команд at
/etc/crontab – файл, що містить таблицю розкладу запуску завдань
kill – посилання сигналів процесу по PID процесу
killall - перериває виконання процесу по імені процесу
nice - задає пріоритет процесу перед його запуском
renice - змінює пріоритет працюючого процесу
ps, (ps –aux) - виводить інформацію про працюючі процеси користувачів
pstree – інформація про дерево процесів
pgrep - інформація про ID процесів
fuser – пошук процесів, які мають відкриті файли або сокети
top - виводить динамічну інформацію про процеси
fg - виводить процес з фонового режиму
bg - продовження виконання фонового процесу, якщо він зупинений натисканням <Ctrl+Z>
hash – інформація про hash таблиці
ipcs - взаємодія процесів (розділювана пам'ять, семафори, повідомлення)

Для отримання більш детальної інформації, можна використати help (наприклад: **ps** --help), або документацію (наприклад: man ps, для виходу натисніть q).

Запуск фонового процесу здійснюється так:

ps -x &

При завантаженні системи, необхідні процеси, завантажуються у фоновий режим, їх називають "демонами". Вони знаходяться у каталозі /etc/rc.d/init.d/.

Деякі комбінації клавіш:

<Ctrl+Z> - призупинити виконання завдань
<Ctrl+C> - завершити виконання завдань

Зв'язування процесів за допомогою каналів. Запуск декількох команд з передачею вихідного потоку наступній програмі, "|" означає передачу вихідного потоку від першої програми до другої.

ps -ax | more

запускається команда ps -ax, і передає вихідний потік програмі more яка запускається на виконання.

Перенаправлення вводу/вивода. Запуск команди із записом вихідного потоку у файл

ps -ax > test.txt
ps -ax >> test.txt - добавить у кінець файлу
jobs – інформація про задачі
-l вивести PID задач додатково до звичайної інформації
-p вивести тільки PID задачі
-n вивести тільки процеси, які змінили свій статус після останнього Повідомлення
-r показати задачі, які виконуються
-i показати задачі, які зупинені

kill – завершити задачу
-l n список імен сигналів
-s вказати ім'я сигналу, який надсилається
-n вказати номер сигналу, який надсилається

renice – змінити пріоритет фонової задачі, якою володіє користувач
-p pids змінити пріоритет використовуючи номери списку PID
-g gids змінити пріоритет використовуючи список процесів групи
-u users змінити пріоритет використовуючи список користувачів

ps – інформація про процеси
Ключі команди ps:
-A – вибрати всі процеси
-a – вибрати всі процеси з tty крім лідерів сеансу
A – вибрати всі процеси терміналу, включно з іншими користувачами
-C – вибрати процеси за іменем команди
-c – інформація про різні планувальники для -l опції
C – використовувати сирий (raw) час ЦП для %CPU замість середнього

C – використовувати дійсні імена команд
--columns (or –cols or --width) – встановити ширину екрану
--cumulative (or S) – включити дані деякий мертвих процесів нащадків (чк суму з батьком)
-d – вибрати всі процеси, але опустити лідерів сесії
--deselect – відмінити вибір
-e – вибрати всі процеси
e – показати середовище після команди
-f – висвітити повний список
--forest (or -H or f) – показати ієрархію процесів (ліс)
--format (or u) – висвітити орієнтований на користувача формат
--Group (or -G) – вибрати за іменем реальної групи або ID
--group (or -g) – вибрати за ефективним іменем групи або ID
--no-headers (or h) – не друкувати заголовки рядків
--html—Shows HTML - escaped вивід
--headers – повторити заголовки рядків
-j (or j) – використати формат jobs
-l (or l) – використати довгий формат
L – вивести списком всі специфікатори формати
--lines (or --rows) – встановити висоту екрану
-m – показувати потоки (threads)
m – показувати всі потоки
-N – аналогічно як --deselect
-n (or N) – встановити namelist файл
—Numeric вихід для WCHAN і USER
--no-headers – не друкувати заголовок рядка
--nul (or –null or --zero) – показати не вирівняний вихід з NULLs
-O (or O) – попередник завантаження -o
-o (or o) – формат визначений користувачем
--pid (or -p) – вибрати за ID процесом
P — вибрати за ID процесом
r – обмежити вивід процесу, який виконується
--sid (or –s or –n or n) – вибрати процеси, які належать до заданої сесії, де *n* є SID
s – висвітити формат сигналу
--sort – задати порядок сортування
T – вибрати всі процеси терміналу
--tty (or –t) – вибрати за терміналом
--User (or -U) – вибрати за реальним username або ID
--user (or -u)- вибрати за ефективним username або ID
U – вибрати процеси заданих користувачів
--version (or –V or V) – друкувати версію
-w (or w) – показати широкий вивід
X – вибрати процеси без контролю ttys
X – старий Linux i386 регістровий формат
-y – не показувати прапори; показувати rss на місці addr

Аргументи команди ps:

c (or cmd) – просте ім’я виконуваного файлу
C (or cmdline) – повний командний рядок
f (or flags) – Прапори у довгому форматі F поля
g (or pgrp) – ID групи процесів
G (or pgid) – контроль ID групи tty процесів
j (or cutime) – накопичений час користувача
J (or cstime) – накопичений час системи
k (or utime) – час користувача
K (or stime) – час системи
m (or min_flt)- номер мінімальної сторінки faults
M (or maj_flt) – номер максимальної сторінки faults
n (or cmin_flt) – накопичена мінімальна сторінка faults
N (or cmaj_flt) – накопичена максимальна сторінка faults
o (or session) – ID сесії

p (or pid) – ID процесу
P (or ppid) – ID батьківського процесу
r (or rss) – розмір резидентного набору
R (or resident) – резидентні сторінки
s (or size) – розмір пам'яті в kilobytes
S (or share) – кількість розподілених сторінок
t (or tty) – мінімальний номер tty пристрою
T (or start_time) – час старту процесу
U (or uid)— ID номер користувача
u (or user)- користувач
v (or vsize) – кількість використовуваної віртуальної память в bytes
y (or priority) – пріоритет керування ядром

1.8. Інформація про використання ресурсів і пристроїв

free – інформація про використання оперативної пам'яті
top – динамічна інформація про використання оперативної пам'яті
vmstat 1 – інформація про використання пам'яті за заданий період часу
vmstat -d – статистика про введення-виведення жорсткого диску
slabtop – використання кеш пам'яті ядром
cat /proc/cpuinfo – інформація про процесор
lsof | less – інформація про поточні відкриті файли і каталоги
dmesg | less – виведення інформації про кільцевий буфер ядра
lsmod – інформація про завантажені модулі
modinfo module – розширена інформація про конкретний модуль
ipcs –m – розподіл сторінок пам'яті
df – обсяги пам'яті на дисках
du – кількість дискових блоків в каталогах
hdparm /dev/sda – інформація про параметри жорсткого диску
lspci - інформація про PCI пристрої
dmidecode – інформація про апаратні пристрої
netstat –s | less – статистика про пересилання мережевих пакетів
nmap 192.168.2.100 – сканування портів

1.9. Інформація про мережеві з'єднання

arp –v – інформація з кеша ARP
ethtool eth0 – інформація про ethernet карти
export http_proxy=http://your.proxy:port - змінити значення змінної оточення http_proxy, для використання Інтернету через proxy-сервер
ifconfig eth0 – отримання MAC адреси і IP-адреси TCP/IP з'єднання
ip addr show – отримання інформації про мережеві з'єднання
hostname – отримання мережевого імені хоста локально] машини
host itshaman.ru - виведення IP-адреси заданого сайту
ip route – тестування шлюзу з таблиці маршрутизації
ip route show – виведення інформація про маршрутизацію
netstat –i – статистика про мережеві інтерфейси
ping 192.168.2.1 – перевірка доступності IP-адреси
nc – підтримка мережевих з'єднань TCP, UPD. Приклад пересилання файлу з комп'ютера А (IP=192.168.32.1) на комп'ютер Б:

```
А >cat backip.iso | pv –b | nc –l 3333
Б >nc 192.168.32.1 3333 | pv –b > backup.iso
```

pppconfig - створення і налаштування Dial-Up з'єднання для виходу в Інтернет з використанням модему
pppoeconf - створення і налаштування виходу в Інтернет через ADSL-модем
service network status – інформація про мережу
wall Привіт – посилання повідомлення "Привіт" на термінали інших користувачів

Запитання.

1. Коли використовуються команди Linux?
2. Які команди використовуються для отримання загальної довідкової інформації?

3. Які команди використовуються для пошуку окремої команди?
4. Як виконати групу команд?
5. Які команди використовуються для адміністрування ОС Linux?
6. Які команди використовуються для роботи з файлами і каталогами?
7. Як створюються м’які і жорсткі посилання на файли і в чому між ними різниця?
8. Як є команди для роботи із стеком каталогів?
9. Які є команди для роботи із завданнями, процесами і сигналами?
10. Які є команди для отримання інформації про використання ресурсів і пристроїв?
11. Які є команди для отримання інформації про мережеві з’єднання?
12. Які є команди для копіювання файлів і перевірки контрольних сум?

Завдання.

1. Запустити на виконання команди з різними опціями та пояснити отримані результати: **cat, find, file, df, fsck.**

2. Запустити на виконання команди з різними опціями та пояснити отримані результати: **dd, grep, touch, top, mkfs.**

3. Запустити на виконання команди з різними опціями та пояснити отримані результати: **cal, mv, mknode, vmstat, dump.**

4. Запустити на виконання команди з різними опціями та пояснити отримані результати: **shutdown, cp, mkfifo, lsof, rsync.**

5. Запустити на виконання команди з різними опціями та пояснити отримані результати: **diff, locate, chmod, lsmod, dd.**

6. Запустити на виконання команди з різними опціями та пояснити отримані результати: **mkfifo, mkdir, at, chown, lsdev.**

7. Запустити на виконання команди з різними опціями та пояснити отримані результати: **finger, ln, sort, kill, du, free.**

8. Запустити на виконання команди з різними опціями та пояснити отримані результати: **wget, cp, hdparm /dev/sda, route.**

9. Запустити на виконання команди з різними опціями та пояснити отримані результати: **who, mount, ps, lsattr, export.**

10. Запустити на виконання команди з різними опціями та пояснити отримані результати: **passwd, cd, fg, ip, host.**

11. Запустити на виконання команди з різними опціями та пояснити отримані результати: **history, ls, tar, nmap, slabtop.**

12. Запустити на виконання команди з різними опціями та пояснити отримані результати: **uptime, pwd, zip, modinfo, ifconfig.**

Лабораторна робота № 2

2. Основи створення сценаріїв на мові Bash

Мета роботи: навчитися створювати прості сценарії із змінними і запускати їх на виконання.

Теоретичний матеріал лекції, література [1-7].

2.1. Короткі теоретичні відомості

2.1.1. Редактор `Vim`

Для роботи із сценаріями (скриптами, scripts) на мові Bash можуть використовуватися редактори `vim, kwrite, kate`. Найпростішим із них є редактор `vim`. Редактор `Vim` має специфічний набір команд, порівнюючи з іншими редакторами, тому з ним не бажають працювати деякі користувачі. `Але` редактор `Vim` працює із даними, які розміщуються у буфері пам'яті, тому він є платформо незалежним. `Vim` може працювати у двох режимах: нормальному (normal) і вставленні (insert).

Нормальний або командний режим. При відкритті нового або існуючого файлу редактор працює у командному режимі. У командному режимі редактор інтерпретує натискання клавіш як команди. У командному режимі можна переміщатися по тексту використовуючи клавіші `hjkl` або ←↓↑→ 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.

У командному режимі є команди для *вилучення, додавання і заміщення даних*:

`x` – вилучити символ у поточній позиції;

`dd` – вилучити рядок у поточній позиції курсора;

`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 char` – замінити символ у поточній позиції курсора на char;

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

З командного режиму можна перейти у *командний рядок* натиснувши клавішу

`Shift` + ":".

В командному рядку, який розміщується внизу екрана, можна вводити додаткові команди, які керують робою редактора `Vim`:

`q` – вийти з редактор, якщо не було зроблено змін в даних;

`q!` – вийти з редактор без збереження внесених змін;

`w filename` – записати вміст буфера даних у заданий файл;

`wq` – записати вміст буфера даних у поточний файл і вийти.

Режим вставлення (insert) або *текстовий режим*. З командного режиму можна перейти у текстовий режим натисканням клавіші **"i"**. З текстового режиму у командний режим

повертаються натисканням клавіші **Esc**. У текстовому режимі можна вставляти і вилучати *як окремі символи, так і їх блоки*:

Для виділення тексту при його редагуванні використовується *візуальний* (visual) режим.

Перехід у всі режими здійснюється тільки через командний режим.

Щоб перейти у візуальний режим з командного, потрібно перемістити курсор у потрібне місце і натиснути клавішу **"v"**. Щоб вийти з візуального режиму потрібно повторно натиснути клавішу **"v".** При подальшому переміщенні курсора виділений текст буде підсвічуватися. Команди командного режиму будуть працювати із виділеним блоком.

Пошук. 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 Insert.

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`

`: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.

2.1.2. Інтерпретатор сценаріїв `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.

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 крапкою з комою:

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

Для виведення дати і часу служить команда

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

Змінні Bash

Існує два типи змінних, які можна використовувати у сценаріях:
- змінні середовища;
- змінні користувача;

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

Використання змінної середовища 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"
```

Змінній може бути присвоєний результат виконання команди двома способами:
- за допомогою символа зворотнього апострофа «`»;
- за допомогою конструкції `$()`;

```
$ DATE=`date`  # або
$ DATE=;(date)
$ printf "%s\n" "$DATE"
```

Якщо команди розділені крапкою з комою то вони виконуються послідовно

```
$ printf "%s\n" "Повідомлення 1" ; printf "%s\n" "Повідомлення 2"
Повідомлення 1
Повідомлення 2
```

Якщо команди розділені подвійним амперсандом && то виконуються до першої помилки

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

Якщо команди розділені подвійними вертикальними прямими || то виконуються після помилки інші команди

```
$ 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
```

Bash зберігає історію команд у файлі `.bash_history`

```
> ls –al | grep .bash*
```

Bash має вбудовану команду history, яка виводить список введених команд

```
$ history           виводить історію всіх команд
$ history  10       виводить 10 останніх команд
$ !10               виконує  10-у команду
```

Отримання інформації про поточний каталог.

```
$ pwd
/home/user1
```

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
$ push /home/user1/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
```

Нульова команда «:»

```
$ :
$ : `date`   перенаправляє вивід у пристрій /dev/null аналогічно як
$ date > /dev/null
```

Використання лапок.

В сценаріях використовуються два типи лапок: подвійні ”…”, одинарні ‘…’ і зворотні апострофи `…`.

Подвійні лапки використовуються в основному для об’єднання декількох слів в стрічку із збереженням в ній пропусків. Приклад створення двох каталогів:

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

Приклад створення імені одного каталога з двох слів:

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

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`. 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
```

Зворотні апострофи використовуються для присвоєння змінним результатів виконання команди, сценарію або арифметичного виразу

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

Налагодження сценаріїв

`bash –n` виявлення помилок сценарію без його виконання

```
$ bash -n bad.sh
```

Опції Bash записуються на початку сценарію

```
#!/bin/bash
# A simple script to list files
shopt -o errexit # завершити сценарій якщо команда завершилася з помилкою
shopt -o nounset # завершити сценарій якщо зустрілася невизначена змінна
shopt –o xtrace  # вивести на консоль кожну команду перед її виконанням
```

Запитання.

1. Як виконати сценарій (скрипт)?
2. Як виконати сценарій із трасуванням команд?
3. Які є режими роботи редактора Vim і як здійснюється перехід між ними?
4. Як отримати інформацію про користувачів.
5. Які є типи змінних. Як їм присвоюються і виводяться значення?
6. Яка різниця у використанні подвійних і одинарних лапок, зворотних апострофів?
7. Розділення команд символами , ||, &&, \.
8. Запуск команд на виконання із історії команд.
9. Як присвоїти змінній результат виконання команди або сценарію?
10. Як перенаправити виведення команди у файл і ввести дані з файлу у команду?
11. Як перенаправити потік даних між командами?
12. Як виконати арифметичні команди у сценарії над цілими числами?

13. Як виконати арифметичні команди у сценарії над числами з плаваючою крапкою з використанням програмного калькулятора bc?
14. Як перевірити статус закінчення сценарію?

Завдання.

1. Запустити доступний редактор коду (Vim, Kwrite, Kate, Pluma), налаштувати розмір шрифту, темний фон, режим нумерації рядків. Записати і зберегти сценарій.
2. Перенаправити у файл інформацію отриману командою "`ls -l`".
3. Написати сценарій для створеня файлу (команда touch).
4. Написати сценарій для створеня каталогу (команда touch).
5. Написати сценарій для вилучення файлу (команда rm).
6. Написати сценарій для пошуку заданого слова у файлі (команда grep).
7. Написати сценарій для виведення текстого файлу з перенумерованими рядками (команда nl).
8. Написати сценарій для виконання виразу (1+5)*(8-3)/(2*3).
9. Написати сценарій для виконання виразу (1.1+5.5)*(8.8-3.3)/(2.2*3) з точністю три цифри після крапки.
10. Написати сценарій в якому командою `exit` повертається кількість файлів в каталозі.
11. Написати сценарій в якому виводяться на екран файли з розширенням *.sh.
12. Написати сценарій в якому виводяться на екран файли поточного каталогу у відсортованому порядку (команда sort).

Приклади для самостійної роботи

1. Створення сценарію файлу:

```
# test1
#!/bin/bash
# Сценарій висвічує дату і користувачів
echo Час і дата:
date
echo Користувачі:
who
```

2. Запуск сценарію на виконання:

```
bash test1

chmod u+x test1
>./test1
```

3. Виконання сценарію з трасуванням команд

```
#!/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

```
# test1a
#!/bin/bash
# Сценарій висвічує дату і користувачів
```

```
echo -n Час і дата:
date
echo Користувачі:
who
```

5.Виведення змінних середовища

```
$printenv
```

6. Виведення інформації про користувачів

```
# test2
#!/bin/bash
# Інформація про користувача із системи
echo "Інформація про користувача із системи: $USER"
echo UID: $UID
echo HOME: $HOME
```

7. Використання змінних користувача

```
# test3
#!/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"

# test4
#!/bin/bash
# Присвоєння значень іншій змінній
var1=10
printf ”%s\n” $var1
var2=$var1
echo var2=$var2
```

8. Використання зворотного апострофа (backtick – ‘)

```
# test5
#!/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
```

9. Перенаправлення вводу і виводу

```
# test6
#!/bin/bash
# Перенаправлення виводу команда->файл
echo "******************"
date > test6out
cat test6out
echo "******************"
# створення нового файлу
who > test6out
cat test6out
echo "******************"
# дописування в існуючий
date > test6out
who >> test6out
cat test6out
```

```
# test6a
#!/bin/bash
# Перенаправлення даних команда<-файл
#  wc - число ліній, число слів, число байтів у вхідному файлі
echo "******************************"
f1=test6a
echo "Число рядків, слів, байтів у вхідному файлі $f1"
wc < $f1
# введення з консолі
echo "******************************"
echo "Число рядків, слів, байтів при вводі з консолі"
wc << EOF
1
12
123
1234
12345
EOF

# test6b
#!/bin/bash
# перенаправлення потоку даних (piping) між командами
# rpm -qa > rpm.list
# sort < rpm.list
#rpm -qa | sort > rpm.list
 rpm -qa | sort | more
```

10. Операції цілочисельної математики

```
# test7
#!/bin/bash
# Цілочисельна математика
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
```

11. Операції арифметики з плаваючою крапкою

```
# test8
# Набрати в консолі
>bc -q
3.44/5
0
scale=4
3.44/5
.6880
quit
>
```

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

```
# test9
#!/bin/bash
var1=`echo " scale=4; 3.44 / 4" | bc`
echo Результат=$var1
```

13. Арифметика з плаваючою крапкою

```
# tes10
#!/bin/bash
var1=100
var2=45
var3=`echo "scale=4; $var1 / $var2" | bc`
echo Результат: $var3
```

14. Арифметика з плаваючою крапкою

```
# test11
#!/bin/bash
var1=20
var2=3.14159
var3=`echo "scale=4; $var1 * $var2" | bc`
var4=`echo "scale=4; $var3 * $var2" | bc`
echo Результат:$var4
```

15. Арифметика з плаваючою крапкою

```
# test12
#!/bin/bash
var1=10.46
var2=43.67
var3=33.2
var4=71
var5=`bc << EOF
scale = 4
a1 = ($var1 * $var2)
b1 = ($var3 * $var4)
a1 + b1
EOF
`
echo Результат: $var5
```

16. Вихід із сценаріїв командою exit

```
# test13
#!/bin/bash
#Перевірка статусу  exit
var1=10
var2=30
var3=$[ $var1 + $var2 ]
echo var3 = $var3
exit 5
>echo $?
```

17. Перевірка статусу exit

```
# test14
#!/bin/bash
var1=10
var2=30
var3=$[ $var1 + $var2 ]
echo var3 = $var3
exit $var3
>echo $?
```

18. Перевірка статусу exit

```
# test14b
#!/bin/bash
var1=10
var2=30
var3=$[ $var1 * $var2 ]
echo var3 = $var3
exit $var3
```

```
>echo $?
```

19. Перевірка статусу в інтерактивному режимі

```
>cd mytmp
>if (( $? ));then rm * ;fi
```

Лабораторна робота № 3

3. Команди if-then-else

Мета роботи: навчитися застосовувати вкладені команди if-then-else.

Теоретичний матеріал лекції, література [1-7].

3.1. Короткі теоретичні відомості

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 `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.

Команда перевірки умов: `test умова`. Команда може об'єднуватися з `if-then` оператором.

```
if test умова
then команди
fi
```

Команди можна *згрупувати використовуючи фігурні д*ужки. Так перевіряється 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”
```

Команда `ls` стає входом для команд у фігурних дужках:

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

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 test ! -f "$TMP -o -f "$TMP" ; then
if test \( ! -f "$TMP" \) -o -f "$TMP" ; then
if [ -f "$TMP" -a ! -w "$TMP" ] ; then
```

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 ]`:

`-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 символічним 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 пуста
`-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 Ви 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 (Вашим сценарієм)
`-s` *file* — True якщо файл існує і не пустий
`-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 (Вашим сценарієм)
`-x` *file* — True якщо файл можна виконати (Вашим сценарієм)
`-z` *s* — (*zero length*) True якщо стрічка є пуста

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

Для створення складних умов перевірки використовується булева логіка:

```
[ умова1 ] && [ умова 2 ]
[ умова 1 ] || [ умова 2 ]
```

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
$ 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
+, - — додавання, віднімання
<<, >> — лівий, правий бітовий зсув
<=, >=, <, > - порівняння
==, != — рівність, нерівність
& — побітове І (AND)
^ — побітове виключне АБО ( XOR )
| — побітове АБО ( OR )
&& — Logical AND
|| — Logical OR
expr ? expr2 : 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
```

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
```

Обчислення і порівняння складних арифметичних виразів.

Подвійні круглі дужки `(( вираз ))` дозволяють *обчислювати і використовувати в умовах складні арифметичні вирази*, наприклад

```
$ var1=((2**8))
$ echo $var1
256
if  (( $var1 ** 2 > 32 )) ; then
echo ”більше 32”
fi
```

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).

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

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

Команда `case` підтримує багатозначний вибір для однієї змінної:

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

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. Який синтаксис інструкції `case-esac`?
12. Як здійснити групування команд?

Завдання.

1. Написати сценарій, який виводить логічне значення для символів `0, 1, -1`.
2. Написати сценарій, який виводить найменше із трьох чисел `a=1, b=2, c=3`.
3. Написати сценарій, який виводить суму найменшого і найбільшого із трьох чисел `a=1, b=2, c=3`.
4. Написати сценарій, яки генерує три випадкові числа і записує їх у масив.

5. Написати сценарій, який порівнює символьні стрічки `str1=”aa”, str2=”b”, str=”ccc”` і виводить найдовшу стрічку.

6. Написати сценарій обчислення `n!`, де значення `n` вводиться з консолі, а результат записується у файл.

7. Написати сценарій який виводить кількість файлів у поточному і батьківському каталогах.

8. Написати сценарій який виводить на екран файли поточного каталогу, створені у поточному місяці.

9. Написати сценарій, який створює в поточному каталозі файл `tmp` і записує в нього історію виконаних `bash` команд.

10. Написати сценарій, який створює в поточному каталозі файл `err` і записує в нього помилки, які виникають при запуску сценаріїв.

11. Написати сценарій, який виводить на екран відсортовані файли каталогу /`tmp`, вибрані за шаблоном `[a-xA-X]`.

12. Написати сценарій, який записує у масив 10 простих чисел, а масив записує у файл.

Приклади для самостійної роботи

1. Перевірка користувача у системі

```
#!/bin/bash
user=student
if grep $user /tec/passwd
then
echo ”Користувач $user відсутній”
fi
```

2. Тестування каталогів, опція -d

```
#!/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

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

4. Перевірка наявності файлу, опція -f

```
#!/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

```
# test14
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 $file не існує"
fi
```

6. Перевірка чи файл порожній

```
# test15
#!/bin/bash
file=t15test
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 пустий"
fi
```

7. Перевірка? чи в файл можна записувати (is writeable)

```
# test16
#!/bin/bash
logfile=$HOME/t16test
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?

```
# test17
#!/bin/bash
if [ -x test16 ]
then
echo "Ви можете виконати сценарій:"
./test16
else
echo "Вибачте, ви не можете виконати сценарій"
fi
```

9. Перевірка власника файлу

```
# test18
#!/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 групи файлу

```
#test19
#!/bin/bash
if [ -G $HOME/testing ]
then
echo "Ви в такій групі як і файл"
else
echo "Ваша група не є власником файлу"
fi
```

11. Перевірка дати файлу

```
# test20
#!/bin/bash
if [ ./test19 -nt ./test18 ]
then
echo "Файл test19 новіший ніж test18"
else
echo "Файл test18 новіший ніж test19"
fi
if [ ./test17 -ot ./test19 ]
then
echo "Файл test17 є старіший ніж файл test19"
fi
```

12. Перевірка дати файлів

```
#test21
#!/bin/bash
if [ ./badfile1 -nt ./badfile2 ]
then
echo "Файл badfile1 новіший ніж файл badfile2"
else
echo "Файл badfile2 новіший ніж badfile1"
fi
```

13. Складені порівняння

```
#test22
#!/bin/bash
if [ -d $HOME ] && [ -w $HOME/testing ]
then
echo "Файл існує і в нього можна записувати"
else
echo "У файл не можна записувати"
fi
```

14. Використання команди declare

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

15. Використання подвійних дужок

```
# test23
#!/bin/bash
val1=10
if (( $val1 ** 2 > 90 ))
then
(( val2 = $val1 ** 2 ))
echo "$val1 в степені 2 буде $val2"
fi
```

16. Використання шаблонів для пошуку

```
# test24
#!/bin/bash
if [[ $USER == r* ]]
then
echo "Вітаю $USER"
else
echo "На жаль, я не знаю Вас"
fi
```

17. Пошук необхідного значення

```
# tesr25
#!/bin/bash
if [ $USER = "rich" ]
then
echo "Запрошую $USER"
echo "Насолоджуйтесь візитом"
elif [ $USER = barbara ]
```

```
then
echo "Запрошую  $USER"
echo "Насолоджуйтесь візитом"
elif [ $USER = testing ]
then
echo "Перевірка заданого користувача"
elif [ $USER = jessica ]
then
echo "Не забудьте вийти з системи (logout) по закінченню роботи"
else
echo "На жаль, Ви не маєте прав доступу"
fi
```

 # 18. Використання команди case

```
# test26
#!/bin/bash
case $USER in
rich | barbara)
echo "Запрошуємо, $USER"
echo "Насолоджуйтесь своїм візитом";;
testing)
echo "Перевірка конкретного користувача";;
jessica)
echo "Не забудьте вийти із системи (log off) по закінченню роботи";;
*)
echo "На жаль Ви не маєте прав доступу";;
esac
```

Лабораторна робота № 4

4. Команди циклів

Мета роботи: вивчення команд циклів.

Теоретичний матеріал лекції, література [1-7].

4.1. Короткі теоретичні відомості

Цикл `for … in … do … 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
```

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
```

Вкладені цикли `for`:

```
#!/bin/bash
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 C:

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

Приклад організації циклу з 10 ітерацій з використанням змінних:

```
#!/bin/bash
x=0;
while [ "$x" -le 10 ]
do
  echo "Поточне значення x: $x"
  # Збільшуємо значення x:
  # x=$(expr $x + 1)
  x=$[ $x + 1]
  sleep 1
done
```

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`

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. Мультикоманди `while` і `until`.
11. Інструкції для виходу і продовження циклу.
12. Як перенаправляються дані із циклу?

Завдання.

1. Написати сценарій, який виводить значення полів файлу `/etc/passwod`, які розділені символом ‘:’.

2. Написати сценарій, який виводить список файлів каталогу `/home/user` і список каталогів з `/home`.

3. Написати сценарій, який у циклі виводить різницю значень двох масивів `mas1=(6,5,4,3,2), mas2=(1,2,3,4,5).`

4. Написати сценарій, який в циклі `for` перенаправляє у файл `tmp` окремі слова із стрічкової змінної `str=”Як ваше здоров’я? Якщо погано, то перестаньте палити цигарки, які труять димом nicotini-ana tabacum легені, і пити алкогольні напої, які руйнують печінку продуктами розкладу розчину` C_2H_5OH в H_20.”

5. Написати сценарій який поелементно об’єднує значення двох стрічкових змінних `x=”1 2 3 4”, y=”a b c d”` і результат перенаправляє у файл.

6. Написати сценарій, який читає із заданого каталогу файли із розширенням `*.sh`.

7. Написати сценарій, який використовує цикли у стилі С для додавання і множення елементів двох масивів `mas1=(1 2 3 4 5), mas2=(5 6 7 8 9).`

8. Написати сценарій, який використовує цикли у стилі С для додавання і множення елементів трьох масивів `mas1=(1 2 3), mas2=(4 5 6), mas3=(7 8 9)` і виводить результати `mas1[i]+mas2[j]*mas3[k]`, якщо вони є простими числами.

9. Написати сценарій, який зчитує файли і каталоги у окремі масиви масив і підраховує кількість елементів масивів.

10. Написати сценарій, який використовує два цикли `for` для зчеплення елементів стрічок `str1=”1 2 3 4 5”`, `str2=”a b c d e”`, а результат перенаправляє у файл.

11. Написати сценарій, який виводить з файлу `/etc/passwd` стрічку, яка відповідає поточному користувачу, працюючому в системі.

12. Написати сценарій, який порівнює файли у двох заданих каталогах та підкаталогах і виводить імена співпавших файлів.

Приклади для самостійної роботи

1. Команда циклу

```
# tesr1
#!/bin/bash
# basic for command
for test in Alabama Alaska Arizona Arkansas California Colorado
do
   echo The next state is $test
done
```

2. Значення змінної після циклу

```
# tes1b
#!/bin/bash
for test in Alabama Alaska Arizona Arkansas California Colorado
do
  echo "Наступний штат є $test"
done
echo "Останній штат був $test"
test=Connecticut
echo "Тепер ми очікуємо наступний штат $test"
```

3. Слова з апострофами

```
# test 2
#!/bin/bash
for test in I don ’t know if this’ll work
do
  echo "слово:$test"
done
```

4. Слова з екранованими апострофами

```
# test 2
#!/bin/bash
for test in I don\’t know if "this’ll" work
do
   echo "word:$test" >> tmp
done
cat tmp | sort -r
```

5. Складені слова

```
# tesr2b
#!/bin/bash
for test in Nevada New Hampshire New Mexico New York North Carolina
do
   echo "Штат $test"
done
```

6. Об'єднання слів

```
# test3
#!/bin/bash
for test in Nevada "New Hampshire" "New Mexico" "New York"
do
   echo "Штат $test"
done
```

7. Використання змінної для зберігання списку

```
# test4
#!/bin/bash
list="Alabama Alaska Arizona Arkansas Colorado"
list=$list" Connecticut"
for state in $list
do
   echo "Ви були в штаті $state?"
done
```

Створити файл states:

```
Alabama
Alaska
Arizona
Arkanzas
Colorado
Connecticut
Delaware
Florida
Georgia
New York
New Hampshire
North Carolina
```

8. Читання з файлу

```
# test5
#!/bin/bash
file="states"
for state in `cat $file`
do
   echo "Штат $state"
done
```

9. Читання з файлу

```
# test5b
#!/bin/bash
```

```
# зберегти значення розділювача полів
IFS_OLD=$IFS
file="states"
# нове значення розділювача полів, одне або декілька,
# наприклад: \n, :, ;, ”. (IFS=$’\n’:;”)
IFS=$’\n’
for state in `cat $file`
do
  echo "Штати $state"
done
# відновлення значення
IFS=$IFS_OLD
```

10. Ітерація по файлах каталогу

```
# ім’я каталогу і файлу взято в лапки бо в імені може бути символ пропуску
# test6
#!/bin/bash
for file in /home/user1/*
do
  if [ -d "$file" ]
  then
     echo "$file : каталог"
  elif [ -f "$file" ]
  then
     echo "$file : файл"
  fi
done
```

11. Ітерація в списку каталогів

```
# test7
#!/bin/bash
for file in /home/user1/.b* /home/user2/*
do
  if [ -d "$file" ]
  then
    echo "$file : каталог"
  elif [ -f "$file" ]
  then
    echo "$file : файл"
  else
    echo "$file : не існує"
  fi
done
```

12. Цикл у стилі C

```
# tes8
#!/bin/bash
for (( i=1; i<=10; i++ ))
do
   echo "Наступна цифра $i"
done
```

13. Цикл по декількох змінних в стилі C

```
# test9
#!/bin/bash
for (( a=1, b=10; a<=10; a++, b-- ))
do
   echo "$a - $b"
done
```

14. Команда while

```
# test10
```

```
#!/bin/bash
var1=10
while [ $var1 -gt 0 ]
do
  echo $var1
  var1=$[ $var1 - 1 ]
done
```

15. Мультикоманда while

```
# test11
#!/bin/bash
var1=10
while echo $var1
[ $var1 -gt 0 ]
do
  echo "В середині циклу"
  var1=$[ $var1 - 1 ]
done
```

16. Команда until

```
# test12
#!/bin/bash
var1=100
until [ $var1 -eq 0 ]
do
  echo $var1
  var1=$[ $var1 - 25 ]
done
```

17. Мультикоманда until

```
# test13
#!/bin/bash
var1=100
until echo $var1
[ $var1 -eq 0 ]
do
  echo В середині циклу: $var1
  var1=$[ $var1 - 25 ]
done
```

18. Вбудовані цикли

```
# test14
#/!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
```

19. Розміщення циклу for всередині циклу while

```
# test15
#!/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
```

20. Цикли until і while

```
# test16
#!/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
```

21. Зміна IFS значення

```
# test16a
#!/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
```

22. Вихід (breaking) з циклу for

```
# test17
#!/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"
```

23. Вихід (breaking) з while циклу

```
# test18
#!/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"
```

24. Вихід (breaking) з внутрішнього циклу

```
# test19
#!/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
```

25. Вихід (breaking) із зовнішнього циклу

```
# test20
#!/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
```

26. Використання команди continue

```
# test21
#!/bin/bash
for (( var1 = 1; var1 < 15; var1++ ))
do
  if [ $var1 -gt 5 ] && [ $var1 -lt 10 ]
  then
    continue
  fi
  echo "Номер ітерації: $var1"
done
```

27. Невірне використання команди continue в циклі while

```
# test21a
#!/bin/bash
# запуск на виконання: test21a | more
var1=0
while echo "Ітерація while: $var1"
   [ $var1 -lt 15 ]
do
  if [ $var1 -gt 5 ] && [ $var1 -lt 10 ]
  then
    continue
  fi
  echo "Номер ітерації в середині циклу while: $var1"
```

```
  var1=$[ $var1 + 1 ]
done
```

28. Продовження зовнішнього циклу

```
# test22
#!/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
```

29. Перенаправлення виходу в файл

```
# test23
#!/bin/bash
for (( a = 1; a < 10; a++ ))
do
  echo "Число $a"
done > test23.txt
echo "Команда for закінчена."
```

30. Направлення (piping) даних циклу в іншу команду

```
# test24
#!/bin/bash
for state in "North Dakota" Connecticut Illinois Alabama Tennessee

  echo "$state буде відвідано наступним"
done | sort
echo "Кінець мандрівки"
```

Лабораторна робота № 5

5 Параметри і ключі командного рядка

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

Теоретичний матеріал лекції, література [1-7].

5.1 Короткі теоретичні відомості

5.1.1 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
tesh.sh 10 20
```

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

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

`$@, $*` повертають всі параметри. Команди відрізняються, якщо параметри взяті у подвійні дужки. У цьому випадку:

`$@` – завжди розділяє параметри символом пропуску і розглядує їх як *окремі слова*, навіть якщо вони взяті у подвійні лапки. `$@` часто використовується для передачі набору ключів іншій команді, наприклад `ls $@`.

`$*` – розділяє параметри першим символом із змінної середовища `$IFS` і розглядує їх як *одне слово*. Якщо перший символ `$IFS` є символом пропуску – то розділювач пропуск, якщо символ невстановлений (`unset`) – то нічим.

Розглянемо різницю між цими 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 пройдемося по вмісту цих змінних в 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

./ test.sh 1 2 3
Параметр $@ = 1
Параметр $@ = 2
Параметр $@ = 3
Параметр $* = 1
```

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

`$_` - містить шлях розміщення Bash або сценарію, коли він стартує вперше. Після виконання кожної команди містить команду або її останній аргумент.

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

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

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
```

5.1.2 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. 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.

Формат команди при виклику у сценарії:

```
getopts optstring variable
```

`optstring` – список букв ключів (після букви ставиться двокрапка, якщо ключ з параметром). 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
```

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
```

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 -- arg2 arg3
```

Для заміни опцій і параметрів командного рядка відформатованою версію команди `getopt` використовується команда `set`.

```
set -- `getopt -q ab:c "$@"`
```

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
```

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`.

Завдання.

1. Написати сценарій, який виводить суми парних і добутки непарних параметрів командного рядка для їх довільного числа.
2. Написати сценарій, який використовує команду `getopts` для контролю ключів і параметрів командного рядка `-a par_a -b -c par_c -d par_d par4`.
3. 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`.
4. Написати сценарій, який використовує команду `getopts` для контролю ключів і параметрів командного рядка `-a -b par_b -d par3 par4 -h help`.
5. Написати сценарій, який використовує команду `getopts` для контролю ключів і параметрів командного рядка `-a -b -c -d pard par1 par2 -e par_e`.
6. Написати сценарій, який використовує команду `getopts` для контролю ключів і параметрів командного рядка `-a par_a -b -d par_d par1`.
7. Написати сценарій, який використовує команду `getopt` для контролю ключів і параметрів командного `-a -b par1 -c -d par2 -- par3 par4`.
8. Написати сценарій, який використовує команду `getopt` для контролю ключів і параметрів командного `-a -b par_b -c -d par_d -- par1 par2`.

9. Написати сценарій, який використовує команду `getopt` для контролю ключів і параметрів командного `-a -b -c -e par_e -- par1`.

10. Написати сценарій, який використовує команду `getopt` для контролю ключів і параметрів командного `-- par1 par1`.

11. Написати сценарій, який використовує команду `getopt` для контролю ключів і параметрів командного `-a par_1 -c -d -e -- par1`.

12. Написати сценарій, який використовує команду `getopt` для контролю ключів і параметрів командного `-a -b -c par_c -- par1 par2 par3`.

Приклади для самостійної роботи

```
# 1. Параметри командного рядка: $1,...,$9
# test0
#/bin/bash
echo "Перший параметр" $1
echo "Другий параметр" $2
total=$[ $1 * $2 ]
echo "Добуток" $total

# 2. Використання параметрів командного рядка
# test1
#!/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
# test3
#!/bin/bash
echo Привіт $1, радий тебе бачити.

# 4. Обробка параметрів при їх кількості більше 9
# test4
#!/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
# test5
#!/bin/bash
echo "Введена команда:" $0

# 6. Використання імені сценарію
# test6
#!/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

```
# test7
#!/bin/bash
if [ -n "$1" ]
then
  echo Параметер 1 наявний
else
  echo "Параметри відсутні"
fi
```

 # 8. Визначення кількості параметрів в командному рядку

```
# test8
#!/bin/bash
echo Є $# параметрів
```

 # 9. Порівняння кількості параметрів

```
# test9
#!/bin/bash
if [ $# -ne 2 ]
then
  echo Виконання програми: test9 a b
else
  total=$[ $1 + $2 ]
  echo Результат: $total
fi
```

 # 10. захоплення останнього параметра

```
# test10
#!/bin/bash
params=$#
echo Число параметрів $params
echo Останній параметр ${!#}
```

 # 11. Тестування параметрів командного рядка $* and $@

```
# test11
#!/bin/bash
echo "Використання \$* методу: $*"
echo "Використання \$@ методу: $@"
```

 # 12. Перевірка параметрів командного рядка $* і $@

```
# test12
#!/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
# test13
#!/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
# test14
#!/bin/bash
echo "Початкові параметри: $*"
shift 2
echo "Зсунуті параметри: $1"

# 15. Видобування опцій з параметрів командного рядка
# test15
#!/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

# 16. Видобування опцій і параметрів з командного рядка
# test16
#!/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

# 17. Видобування опцій і значень аргументів командного рядка
# test17
#!/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

# 18. Використання команди getopts
# test18
#!/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
./test18 -a -b 1111 -c
Знайдено опцію -a
Знайдено опцію -b з параметром 1111
Знайдено опцію -c

# 19. Обробка опцій командою getopts
# test19
#!/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
./test19 -a -b 1111 -c -d
Знайдено опцію -a
Знайдено опцію -b з параметром 1111
Знайдено опцію -c
Знайдено опцію -d
```

 # 20. Обробка опцій командою getopts

```
#!/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 аргумента"
  echo "    -f    Ще ключ без аргумента"
}

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
```

 # 21. Видобування опцій і параметрів командного рядка командою getopt

```
# test20
#!/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
./test20 -a -b 111 -c
Знайдено опцію -a
Знайдено опцію -b з параметром 111
Знайдено опцію -c

# 22. Тестування команди  read
# test21
#!/bin/bash
echo -n "Введіть ім'я: "
read name
echo "Привіт  $name, запрошуємо в нашу програму."

# 23. Тестування команди read з опцією  -p
# test21
#!/bin/bash
read -p "Скільки годин:" god
sec=$[ $god * 3600 ]
echo Введені години мають $sec секунд

# 24. Введення декількох змінних
# test23
#!/bin/bash
read -p "Введіть ім'я, по батькові: " first last
echo "Вітаю Вас $first $last,..."

# 25. Тестування REPLAY змінної середовища
# test24
#!/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"

# 26. Час на введення даних
# test25
#!/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

# 27. Обмеження вводу одним символом
# test26
```

```
#!/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"
```

 # 28. Читання даних з файлу test28.txt

```
# test27
#!/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

6. Перенаправлення введення-виведення

Мета роботи: навчитися перенаправляти дескриптори файлів введення-виведення.

Теоретичний матеріал лекції, література [1-7].

1. Короткі теоретичні відомості

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

В Linux стрічка `&0` використовується для посилання на потік стандартного введення, `&1` - на потік стандартного виведення, а `&2` – на потік виведення помилок.

```
printf ”Sales are up” > results.txt # направити у файл на диску
printf ”Sales are up” > /dev/tty # направити на екран (безпосередньо)
printf ”Sales are up” # направити на екран через стандартний вивід
printf ”Sales are up >&1 #  -”- через стандартний вивід
printf ”Sales are up >/dev/stdout # -”- через стандартний вивід

$ bash listorders.sh > listing.txt
$ ls -l incoming/orders # записано у listing.txt
$ ls -l incoming/orders 1>&1 # записано у listing.txt
$ ls -l incoming/orders > /dev/tty # виведено на екран

$ printf ”$SCRIPT:$LINENO: Відсутні файли для оброблення” >&2
```

В Linux можна перенаправити результати виконання команд і сценаріїв у файли, а файли перенаправити як ввід для команд.

Перенаправлення результатів виконання команд у файл:

```
command > file  # перезаписування даних у файлі результатом виконання команди
command >> file # добавлення даних у файл результатом виконання команди
```

Перенаправлення даних з файлу на вхід команди:

```
command < file
$ wc < test6
2 3 6  # число рядків число слів число байтів
```

Синтаксис перенаправлення даних з консолі на вхід команди

```
command << унікальні_символи_початку_даних
>дані
…
>дані
>Унікальні_символи кінця даних (мають співпадати із символами початку)
```

Приклад:

```
$ wc << abc
>1
>2
>3
>abc
3 3 6
```

Для передачі даних з виходу однієї команди на вхід іншої використовуються канали (pipes):

```
command1 | command2
```

Так, виконання двох команд можна замінити одною командою

```
>rpm –qa > rpm.list
>sort < rpm.list
>rpm –qa | sort > rpm.list
```

Перенаправлення стандартних дескрипторів

`1> file1` – перенаправлення стандартного вихідного потоку `stdout` у файл
`1>> file1` – добавлення стандартного вихідного потоку `stdout` у файл
`2> file2` – перенаправлення потоку помилок `stderr` у файл
`2>> file2` – добавлення потоку помилок `stderr` у файл
`&> file3` - перенаправлення потоків `stdout` і `stderr` у файл
`&>> file3` - добавлення потоків `stdout` і `stderr` у файл

Файлові дескриптори можуть бути переназначені:
- тимчасово для одноразового виконання команди;
- постійно для всіх команд сценарію.

При тимчасовому переназначенні дескриптора перед ним ставиться знак &:

```
echo ”Привіт”  >&1
echo ”Помилка” >&2
```

Для постійного переназначення файлового дескриптора використовується команда `exec`, яка запускає нову оболонку сценаріїв (`shell`) і переназначає файловий дескриптор на час дії сценарію з командою `exec`.

```
exec 1> testout  # перенаправлення stdout, stderr у файл
exec 1> err      # перенаправлення stderr у файл
```

Можна перенаправити вхідний потік STDIN з клавіатури на файл

```
exec 0< file
```

Оболонка сценаріїв може мати до дев’яти відкритих файлових дескрипторів. Дескриптори з третього по дев’ятий можуть використовуватися для перенаправлення як введення так і виведення.

Можна назначати стандартні файлові дескриптори альтернативним і навпаки.

```
file=”test.txy”
exec 3>&1
echo ”Переназначити дескриптор 3 в 1” >&3 $file
echo ”3=>1” > $file
exec 1>&3
echo ”Відновити дескриптор 1 з 3" >&1
cat $file
```

Аналогічно можна переназначити стандартний файловий дескриптор входу. Наприклад файловий дескриптор STDIN зберігається в альтернативному дескрипторі перед

перенаправленням у нього файлу. Після зчитування файлу можна відновити STDIN у його попереднє значення.

```
exec 6<&0
exec 0< testfile
…
read line
…
exec 0<&6
```

Для призначення файловим дескрипторам можливостей введення/виведення даних використовується команда `exec` з відповідним номером файлового дескриптора і символом <>:

```
exec  3<> file
read line <&3
echo ”Test” >&3
```

При створенні нових вхідних або вихідних файлових дескрипторів Bash оболонка закриває їх при завершенні сценарію. Для ручного закриття файлових дескрипторів їх необхідно переназначити у спеціальний символ `&-`:

```
exec 3>&-
```

Приклад читання і запису в задану позицію файлу:

```
file=”test”
: < file                # обнулення файлу
echo ”12345”            # запис у файл стрічки
exec 5<> file           # зв’язування з файлом дескриптора 5
read –n 2 <&5           # читання 2-х символів з файлу
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
```

Для подавлення виведення команд використовується нульовий пристрій Linux:

```
ls –l > /dev/null
```

В Linux існує спеціальний каталог `/tmp` для розміщення тимчасових файлів. Систему можна конфігурувати так, що тимчасові файли будуть знищуватися при завантаженні системи. Для створення тимчасових файлів і каталогів служить команда `mktemp` з відповідними ключами.

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

```
tee filename
```

Команда `tee` також використовується з командою створення каналу для розгалуження перенаправлення виходу любої команди

```
who | tee testfile
```

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 назначити стандартні файлові дескриптори альтернативним і навпаки?

6. Як назначити файловим дескрипторам можливість введення/виведення даних?

7. Як закриваються файлові дескриптори?

8. Якою командою можна отримати інформацію про відкриті файлові дескриптори системи Linux?

9. Як подавити виведення команд у Linux?

10. Де і як створюються тимчасові файли і каталоги?

Завдання.

1. Ввести стрічку “Створення файлу” з консолі у порожній файл.

2. За допомогою команд `ls` і `wc` визначити кількість файлів у поточному каталозі і результат перенаправити у файл `file`.

3. Прочитати два файли і перенаправити їх вміст у третій файл.

4. Перенаправити повідомлення про помилки при виконанні команди `ls xxx` у файд `err`.

5. Переназначити дескриптори 5, 6 в 1, вивести повідомлення “Hello” у файли зв’язані з дескрипторами 5, 6.

6. Переназначити дескриптор 0 в 5, 6 і зчитати дані з файлів, зв’язаних з дескрипторами 5, 6 і вивести у stdout.

7. Створити дескриптор 5 з можливістю читання і запису файлу. Записати у файл декілька рядків даних, прочитати їх і перезаписати непарні рядки символом “*”.

8. Створити дескриптор 5 з можливістю читання і запису файлу. Записати у файл декілька рядків даних, прочитати їх і перезаписати парні рядки символом “+”.

9. Створити дескриптор 5 з можливістю читання і запису файлу. Записати у файл декілька рядків даних, прочитати їх і перезаписати рядки з першим символом “+”.

10. Створити тимчасові файли у поточному каталозі і каталозі `/tmp`, записати туди дані і висвітити їх на консоль. Закрити файлові дескриптори і знищити тимчасові файли.

11. Створити тимчасові каталоги і в них тимчасові файли у поточному каталозі та каталозі `/tmp`, записати туди дані і висвітити їх на консоль. Закрити файлові дескриптори і знищити тимчасові файли та каталоги.

12. Створити за допомогою команди `tee` реєстраційний файл виведення сценарію.

Приклади для самостійної роботи

1. Перенаправлення стандартних дескрипторів

```
# test2
# дескриптор файлу: 0 - STDIN, 1 – STDOUT, 2 - STDERR
# направлення виходу в STDOUT
ls -l > test2.txt
# добавлення виходу до файлу
ls -l >> test2.txt
who >> test2.txt
# направлення виходу помилок в STDERR
ls -al xyz 2> err
# направлення STDERR і STDOUT в один файл
ls -al test2 xyz &> test2.txt
# направлення STDERR в test2, SRDOUT в test1
ls – al test0 2> test2.txt test1 1> test1.txt
```

2. Тимчасове перенаправлення повідомлення у файловий дескриптор STDERR

```
# test8
#!/bin/bash
echo "Це повідомлення про помилку для STDERR" >&2
echo "Це звичайне повідомлення для STDOUT"
```

3. Постійне перенаправлення всіх виходів у файл

```
# test10
#!/bin/bash
exec 1> test10.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

```
# test11
#!/bin/bash
# перенаправлення виводу у файл
exec 2>test11.err
echo "Старт сценарію"
echo "перенаправлення всіх виходів"
exec 1>test11.txt
echo "Це   повідомлення буде виведене у файл test11.txt"
echo "а це повідомлення буде виведене у файл test11.err" >&2
```

5. Перенаправлення введення з файлу за допомогою команди exec

```
# створити файл test12.txt і ввести в нього 3-4 рядки даних
# test12
#!/bin/bash
exec 0< test12.txt
count=1
while read line
do
echo "Рядок #$count: $line"
count=$[ $count + 1 ]
done
```

6. Призначення альтернативних файлових дескрипторів (скрипт може мати до дев'яти відкритих файлових дескрипторів)

```
# test13
#!/bin/bash
exec 3>test13.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

```
# test14
#!/bin/bash
exec 3>&1
echo " Призначення дескриптора 3 для 1" >&3
exec 1>test14.out
echo "Перенаправлення STDOUT в файл"
exec 1>&3
echo " Призначення дескриптора 1 для 3" >&1
```

8. Перенаправлення дескрипторів вхідного файлу

```
# test15
#!/bin/bash
exec 6<&0
exec 0< test15.txt
count=1
  while read line
  do
    echo "Рядок #$count: $line"
    count=$[ $count + 1 ]
  done
```

```
  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
```

9. Створення дескрипторів файлу з можливістю введення/виведення даних

```
# test 16
#!/bin/bash
exec 3<> test16.txt
read line <&3
echo "Читання: $line"
echo "Це тестовий рядок" >&3
```

10. Закриття дескриптора файлу

```
# test17
#!/bin/bash
exec 3> test17.txt
echo "Це тестовий рядок даних" >&3
exec 3>&-
cat test17.txt
exec 3> test17.txt
echo "Це буде невірно" >&3
```

11. Використання команди lsof для отримання інформації про відкриті файлові дескриптори всієї системи Linux

```
# test18
#!/bin/bash
exec 3> test18.1
exec 6> test18.2
exec 7< test18.txt
#/usr/sbin/lsof -a -p $$ -d 0,1,2,3,6,7
lsof -a -p $$ -d 0,1,2,3,6,7
```

12. Подавлення виведення команд

```
# test19a
#/bin/bash
echo Подавлено вивід на STDOUT
ls -al xyz  1> /dev/null
echo Подавлено вивід на SDTERR
ls -al xyz  2> /dev/null
```

13. Створення і використання тимчасового локального файлу

```
# test19
#!/bin/bash
tempfile=`mktemp test19.XXXXXX`
exec 3>$tempfile
echo "Скрипт записує дані в тимчасовий файл $tempfile"
echo "Перший рядок" >&3
echo "Другий рядок" >&3
echo "Третій рядок" >&3
# обнулення дескриптора
3>&-

echo "Створено тимчасовий файл. Його вміст:"
cat $tempfile
# вилучення файлу
rm -f $tempfile 2> /dev/null
```

14. Створення тимчасового файлу в /tmp

```
# test20
```

```
#!/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
```

15. Використання тимчасових каталогів

```
# test21
#!/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
```

16. Команда tee

```
# test22
#!/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
```

Лабораторна робота № 7
7. Керування задачами і оброблення сигналів ОС

Мета роботи: вивчення команд керування задачами і оброблення сигналів ОС.

Теоретичний матеріал лекції, література [1-7].

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 &
```

Перевести задачу із фонового режиму в основний можна командою `fg`.

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 %?sle
$
[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 команди 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 С, показані в таблиці 7.1.

Таблиця 7.1 – **Сигнали**

Числове значення	Символьне ім’я (мова С)	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	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	Можливий ввід/вивід
30	SIGPWR	Помилка живлення
31	SIGSYS	Помилковий системний виклик

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 добавляється у файл `nohup.out`:

```
$ nohup ./test1 &
[1] 30797
$ nohup: ignoring input and appending output to `nohup.out’
```

1.3. Команда `suspend`

Команда `suspend` зупиняє виконання сценарію в якому вона записана. Вона є аналогом до команди `kill -SIGSTOP $$` (зупинка сценарію самого себе).

```
#!/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. Коли виконуються сценарії потрібні спеціальні дії для обробки 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 під сценаріям. 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 ]
```

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
$ trap 28
```

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 вводу/виводу
```

```
    R – виконання або очікування виконання
    S – очікування (sleeping)
    T – трасується або зупинений
    Z – не функціонуючий процес, очікуючий вилучення ( zombie )
UID – ідентифікатор користувача
PID – ідентифікаційний номер процесу
PPID - ідентифікаційний номер процесу, який виконує програму
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

Команда at дозволяє запускати сценарії у заданий час. Команда надсилає завдання у чергу в спеціальний каталог (`/var/spool/at`), з якого bash буде їх запускати. Формат команди

`at [-f filename] time`

`filename` – команда або скрипт, який потрібно запустити на виконання;
`time` – час запуску завдання (ГГ.ХХ, наприклад 20.45).

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.

Список задач, які стоять в черзі на виконання можна отримати командою `atq`. Вилучити задачу із списку за її номером `N`, можна командою `atrm N`.

Запускати задачі на виконання при зменшенні навантаження на систему можна командою `batch`. Команда `batch` перевіряє середнє завантаження системи і якщо воно менше 0.8, вона запускає на виконання любу задачу із черги задач. Формат команди

`batch [-f filename] [time]`

`time` – час, починаючи з якого команда може запукати завдання на виконання.

Для регулярного запуску завдань (за календарем) призначена програма `cron`. Програма `cron` виконується у фоновому режимі і перевіряє спеціальну `cron` таблицю, яка містить завдання на виконання. Таблиця містить записи наступного формату

`хвилини години день_місяця місяць день_тижня команда`

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
```

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
```

Редагування `cron` таблиці:

```
> crontab [-u user] file
```

Програма `cron` не контролює виконання завдань у випадку відмови системи. Цей недолік усуває програма `anacron`, яка ставить часові позначки для визначення виконання задачі. Якщо задача у заданий час не було виконана, то програма `anacron` запустить її на виконання при

першій можливості. Програма 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.

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. Яке функціональне призначення команд cron, anacron?

Завдання.

1. Написати сценарій, який у фоновому режимі, без консолі, запише у файл назву сценарію і поточну дату

2. 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 запускає на виконання інший сценарій в якому кожні 2 сек видається на екран повідомлення “Сценарій 2 працює”, потім сигналом KILL завершує його виконання.

3. Написати сценарій, який запускає на виконання інший сценарій в якому кожні 2 сек видається на екран повідомлення “Сценарій 2 працює”, потім сигналом SIGSTOP призупиняє його виконання на 10 сек і сигналом SIGCONT продовжує його виконання.

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 виводить свій поточний пріоритет командою nice, а потім зміняю його командою renice до значення 15 використовуючи PID свого процесу.

6. Написати сценарій, який щохвилини виводить поточний час. Написати команду at для запуску цього сценарію через 5 хвилин від поточного часу (формат Now + 5 minutes).

7. Написати сценарій, який обробляє направлений йому сигнал SIGINT і виводить про це повідомлення.

8. Написати сценарій, який обробляє направлений йому сигнал SIGQUIT і виводить дамп пам’яті та повідомлення про це.

9. Написати сценарій, який обробляє направлений йому сигнал SIGTERM і виводить про це повідомлення.

10. Написати сценарій, який зупиняє SIGSTOP, а потім SIGCONT продовжує інший процесс (сценарій).

Приклади для самостійної роботи

1. Перехоплення сигналу CTRL+C і виконання команди echo

```
# test1
#!/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"
Цикл #1
Цикл #2
Ха-Ха!
Цикл #3
Цикл #4
Цикл #5
Кінець програми
```

2. Перехоплення команди exit і виконання команди echo

```
# test2
#!/bin/bash
trap "echo Перехоплення команди EXIT" EXIT
count=1
while [ $count -le 5 ]
do
echo "Цикл #$count"
sleep 3
count=$[ $count + 1 ]
done
Цикл #1
Цикл #2
Цикл #3
Цикл #4
Цикл #5
Перехоплення команди EXIT
```

3. Вилучення встановленої пастки опцією trap ”-”

```
# test3
#!/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"
Цикл #1
Цикл #2
Цикл #3
Цикл #4
Цикл #5
Пастка вилучена
```

3а. Запуск задачі в фоновому режимі

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

```
./test1 &
Цикл #1
Цикл #2
<ENTER>
ps au
Цикл #3
Цикл #4
Цикл #5
Кінець програми
```

3b. Запуск декількох фонових задач

```
# test3b
echo Запуск першої фонової задачі
./test1 &
echo Запуск другої фонової задачі
./test1 &
Цикл #1
Цикл #2
<ENTER>
ps au
…
Цикл #5
Цикл #5
Кінець програми
Кінець програми
```

4. Тестування команди jobs

```
# test4
#!/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"
Це тестова програма 4155
Цикл #1
Цикл #2
Цикл #3
Цикл #4
Цикл #5
Кінець тестової програми
```

4a. Зміна пріоритетів

```
# test4a
#!/bin/bash
# пріоритети -10(самий високий),  0(сценарій),  10 (самий низький)
# зміна пріоритету задачі
nice -n 10 ./test4 > test4.txt &
# список задач які виконуються
ps al
# зміна пріоритету задачі під час її виконання
# renice 10 -p  PID
```

5. Виконання сценаріїв без консолі (у фоновому режимі)

```
nohup ./test1 &
cat nohup.out
Цикл #1
Цикл #2
Цикл #3
Цикл #4
Цикл #5
Кінець програми
```

5. Тестування команди at

```
#Для виконання команди at має бути запущений демон atd із правами root
# test5
#!/bin/bash
time=`date +%T`
echo "Сценарій виконується в $time"
echo "Кінець сценарію" >&2

# ввести відповідний час
>at -f test5 ГГ:ХХ
```

Лабораторна робота № 8

8. Функції Bash

Мета роботи: навчитися створювати і використовувати функції.

Теоретичний матеріал лекції, література [1-7].

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`;
- любі обробники сигналів встановлені програмою `trap`;
- експортовані змінні середовища;
- експортовані функції інтерпретатора 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`.

Підсценарії успадковують своє середовище від батьківського сценарію. Середовище 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`).

2. Створення функцій

Існує два формати описання функцій. У першому форматі функція оголошується командою `function`, а у другому задається тільки ім'я функції `name()`:

```
function name {
  commands
}
```

```
name() {
   commands
}
```

Імена функцій мають бути унікальними, а визначення функцій – розміщуватися на початку сценарію. Для запобігання перевизначення функцій використовується команда

```
>readonly -f fun_name
```

Змінні `$SCRIPT` і `$FUNCNAME` містять імена сценарію і функції. Вони використовуються при визначення місця виникнення помилок `echo $SCRIPT $FUNCNAME`.

Так як функція не є сценарієм вона повертає статус код і декларовані змінні по іншому. У функції команда `exit` завершує сценарій. Для повернення статус коду без завершення функції

використовується команда `return`. Команда `return` може повертати як статус код фіксоване значення `return 1` або обчислюване значення `return $[ $value * 2 ]`.

При завершенні, без команд `exit` і `return`, функція повертає значення останньої виконаної команди. Це значення містить спеціальна змінна середовища основного сценарію `$?`.

Стандартний вивід функції можна присвоїти змінній `result=`myfun`` або `result=$(myfun)`.

Функція може використовувати стандартні параметри змінних середовища. Ім'я функції визначено у змінній `$0`, параметри функції – у змінних `$1,…$9`, число параметрів – у змінній `$#`. Так як функція використовує змінні середовища для своїх параметрів, то вона не має доступу до параметрів сценарію, які передаються через командний рядок.

Функції використовують два типи змінних:

- глобальні;
- локальні.

Для явного оголошення змінних рекомендується використовувати команду `declare`. Змінні оголошені в основному сценарії і функціях є глобальними. Для оголошення 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`. Якщо змінна явно не оголошена (тобто є глобальною), то вона не знищується при завершенні функції, що може спричиняти неочікувані результати. Локальні змінні знищуються при завершенні роботи функції.

```
$ function f { COMPANY="Rogy i kopyta Inc." ; }
$ f
$ printf "%s\n" "$COMPANY"
Rogy I kopyta Inc.

$ function f { declare COMPANY="Rogy i kopyta Inc." ; }
$ f
$ printf "%s\n" "$COMPANY"
```

Всі змінні Bash зберігаються як символьні рядки. Кожна змінна має атрибути, які можна встановити командою `declare`, приклад оголошення цілочисельної змінної

```
>declare –i nomer=15
>printf "%d\n" $i
>nomer="Hello" # помилка: присвоєння цілочисельній змінній значення стрічки
```

Змінну можна оголосити константою `declare -r` (`read-only`).

Атрибути змінної або функції можна вивести командою

```
>declare –p my_variable
>declare –p my_function
```

Якщо змінні 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 імя цього сценарію
declare -rx who="/usr/bin/who" # команда who
```

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="Rogy & Kopyta"
bash inner.sh
```

```
printf ”%s\n” ”$COMPANY_NAME”
exit 0
#inner.sh
declare -p COMPANY_NAME
COMPANY_NAME=”Xvosty & vucha”
```

Для того щоб функція була доступною за межами її визначення іншим сценаріям, потрібно використати команду `export`

```
$export -f my_fun
```

Наявність локальних змінних дозволяє створювати рекурсивні функції. Рекурсивні функції використовують локальні змінні і тільки ті зовнішні змінні, які передаються як параметри 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
```

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
```

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, бібліотечний файл можна виконати командою

```
>. ./mylib.
#!/bin/bash
# source1.sh: включення файлу source2.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
source $source2
printf “The variable YEAR is %s\n” “$YEAR”
exit 0

# source2.sh: цей файл є фрагментом, а не окремим сценарієм і  буде
#             вставлений в файл source1.sh
declare -r YEAR=’date ’+%Y’’

>./source2.sh  # запуск на виконання
```

Для того щоб функції бібліотечного файлу були доступні при завантаженні bash їх потрібно помістити у файл `.bashrc`.

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`.

Запитання.

1. Що містить середовище виконання програми і підсценарію?
2. Формати описання функцій. Що місять змінні середовища `$SCRIPT i $FUNCNAME?`

3. Які значення може повертати функція і як їх отримати?

4. Які змінні середовища використовує функція для передачі параметрів?

5. Область видимості змінних у Bash сценарії. Як оголошуються локальні і глобальні змінні.

6. Як задати і перевірити атрибути змінних і функцій?

7. Особливості виконання зовнішнього сценарію командами `source` і `exec`?

8. Створення бібліотечного файлу функцій і підключення його до сценарію?

9. Сценарій демон, як він запускається, функціонує і завершується.

Завдання.

1. Передати в основний сценарій чотири цілі числа з командного рядка. Суму двох перших чисел обчислити у функції, яка розміщена всередині основного сценарію, а різницю двох наступних чисел обчислити у функції, яка записана в окремому файлі. В основному сценарії порахувати добуток сум і вивести на екран.

2. Написати рекурсивну функцію для визначення факторіалу числа.

3. Написати функцію, яка виводить на екран результат додавання, віднімання, множення і ділення двох цілих чисел використовуючи бібліотечний файл з відповідними функціями.

4. Написати сценарій, якому створюються два масиви і передаються у функцію. Функція обчислює суму елементів цих масивів і повертає значення в основний сценарій.

5. Написати сценарій, якому створюються два глобальні масиви. Функція обчислює суму елементів цих масивів і повертає значення в основний сценарій.

6. Написати сценарій, в якому створюється масив і передається у функцію. Функція знаходить мінімальний елемент масиву і повертає значення в основний сценарій.

7. Написати сценарій, в якому створюється масив і передається у функцію. Функція знаходить парні елементи масиву і повертає значення нового масиву в основний сценарій.

8. Написати сценарій, в якому створюється масив і передається у функцію. Функція знаходить перші три парні елементи масиву і повертає значення нового масиву в основний сценарій.

9. Написати сценарій, в якому створюється глобальний масив. Функція знаходить максимальний елемент масиву і повертає значення в основний сценарій.

10. Написати сценарій, в якому створюється глобальний масив. Функція знаходить непарні елемент масиву і повертає новий масив в основний сценарій.

11. Замінити у сценарії `polling.sh` команду `sleep` командою `suspend`, що спричинить його зупинку. Запустити сценарій на виконання `./16.sh`. Показати його стан командою `ps au`. Записати у каталог `/home/user/ftp/ftp_incoming ftp_incoming` нові файли. Командою `kill -SIGCONT` продовжити роботу сценарію. Показати, як реагує сценарій на появу нових файлів.

12. Написати програму демон, яка перед вилученням файлів поточного каталогу записує їх в каталог `trash`.

Приклади для самостійної роботи

1. використання функцій в сценаріях

```
# test1
#!/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"
```

2. Розміщення функції в середині сценарію

```
# test2
#!/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"
}
```

3. Дублювання імені функції

```
# test3
#!/bin/bash
function func1 {
echo "Перше визначення функції func1"
}
func1
function func1 {
echo "Повторне визначення функції func1"
}
func1
echo "Кінець сценарію"
```

4. Тестування exit статусу функції

```
# test4
#!/bin/bash
func1() {
echo "спроба звернення до неіснуючого файлу"
ls -l badfile
}
echo "тестування функції:"
func1
echo "exit статус: $?":
```

4b. Тестування exit статусу функції

```
# test4b
#!/bin/bash
func1() {
ls -l badfile
echo "тестування поганої команди"
}
echo "тестування функції:"
func1
echo "exit статус: $?"
```

5. Використання команди return у функції

```
# test5
#!/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 $?"
```

6. Використання echo для повернення значення

```
# test5a
#!/bin/bash
function func1 {
read -p "Введіть значення: " value
echo $[ $value ]
}
result=`func1`
echo "Повернене значення $result"
```

7. Доступ до параметрів сценарію всередині функції

```
# test7
#!/bin/bash
# $# - кількість параметрів, які передаються у функцію
function func7 {
 echo $[ $1 * $2 ]
}
 if [ $# -eq 2 ]
 then
   value=`func7 $1 $2`
   echo "Результат $value"
 else
   echo "Виконання: test7 a b"
 fi
```

8. Використання глобальних змінних для передачі значень

```
# test8
#!/bin/bash
function f1 {
value=$[ $value * 2 ]
}
read -p "Введіть число: " value
f1
echo "Результат: $value"
```

9. Використання локальних змінних

```
# test9
#!/bin/bash
function func1 {
  local temp=$[ $value + 5 ]
  echo "значення у функції temp=$temp"
  result=$[ $temp * 2 ]
}

 temp=4
 echo "значення в сценарію temp=$temp"
 value=6
 func1
 echo "Результат $result"

if [ $temp -gt $value ]
```

```
then
 echo "temp є більше"
else
 echo "temp є менше"
fi
```

10. Передача вектора змінних у функцію

```
# test10
#!/bin/bash
# передача вектора змінних у функцію
function testit {
 local newarray
 newarray=(`echo "$@"`)
 echo "Новий вектор змінних: ${newarray[*]}"
}
myarray=(1 2 3 4 5)
echo "Початковий вектор змінних ${myarray[*]}"
testit ${myarray[*]}
```

11. Присвоєння значень вектору

```
# test11
#!/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. Повернення вектора значень

```
# test12
#!/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. Використання рекурсії

```
# test13
#!/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
result=`factorial $value`
echo "Факторіал від $value : $result"
```

14. Використання функцій визначених у бібліотечному файлі поточного каталогу

```
# файл myfuncs
function addem {
 echo $[ $1 + $2 ]
}
function multem {
 echo $[ $1 * $2 ]
}
function divem {
 echo $[ $1 / $2 ]
}

# test14
#!/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"
```

15. Використання функцій визначених в файлі .bashrc

```
# .bashrc
# Source global definitions
if [ -r /etc/bashrc ]; then
. /etc/bashrc
fi
# підключенні бібліотечного файлу користувача командою source (.).
. /home/user/libraries/myfuncs

# test15
#!/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"
```

16. Програма демон з опитуванням

```
#!/bin/bash
# polling.sh: демон використовує опитування для виявлення нових файлів
shopt -s -o nounset
declare -rx SCRIPT=${0##*/}
declare -rx INCOMING_FTP_DIR="/home/user/ftp/ftp_incoming"
declare -rx PROCESSING_DIR="/home/user/ftp/processing"
declare -rx stat="/usr/bin/statftime"
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 ftp_incoming - aborted" >&1
exit 1
fi
if test ! -r "$PROCESSING_DIR" ; then
   printf "%s\n" "$SCRIPT:$LINENO: помилка каталогу processing - aborted" >&1
exit 1
fi
if test ! -r "$statftime" ; then
   printf "%s\n" "$SCRIPT:$LINENO: неможливо знайти або виконати $stat -\
   aborted" >&1
   exit 1
fi

# опитування нових FTP файлів
cd $INCOMING_FTP_DIR
while true; do
# Перевірка на незмінність нових файлів кожні 30 сек
FILES='find . -type f -mmin +30 -print'
# Якщо нові файли існують перемістити їх у каталог processing
if [ ! -z "$FILES" ] ; then
   printf "$SCRIPT: нові файли поступили в %s\n" "'date'"
   printf "%s\n" "$FILES" | {
   while read FILE ; do
   # Remove leading "./"
     FILE="${FILE##*/}"
     # Перейменувати файли з поточним розміром і датою
     NEW_FILE=`$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
```

#При звичайному запуску демона всі повідомлення направляються на екран
```
./sh.16
16.sh стартував у Sun Apr 21 10:32.56 EEST 2013
16.sh: нові файли поступили у Sun Apr 21 10:32.56 EEST 2013
16.sh: переслано hello.sh у /home/user/ftp/processing/16.sh 41 2013-04-21 22:15:05
```

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

9. Створення текстових і графічних списків вибору

Мета роботи: навчитися створювати тестові і графічні списки вибору, керувати кольорами тексту.

Теоретичний матеріал лекції, література [1-7].

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 подавляє виведення символу 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`.

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 забілом. 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 list
do
  commands
done
```

де `list` список елементів меню відокремлених пропусками, `PS3` – змінна середовища, значення якої виводиться у кінці перенумерованого списку елементів меню.

Приклад команди `select`:

```
#!/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
2) Display logged on users 4) Exit program
Enter option:
```

3 Керування кольорами із сценаріїв

Більшість програм емуляторів терміналів розпізнають керуючі ANSI `Esc`-символи. ANSI `Esc`-символи починаються з керуючої послідовності CSI (control sequence indicator), яка вказує, що за нею розміщуються параметри, які встановлюють режим відображення дисплею та задають колір тексту і фону.

Режимом відображення дисплею задають параметри SGR (Select Graphic Rendition). Синтаксис параметрів SGR:

`CSI`*`n`*`[;`*`k`*`]m`

`CSI` – керуючі `Esc`-символи;

`n`, `k` параметри, які визначають режим відображення дисплею;

`m` – признак SGR параметрів.

Можна задати один або два параметри одночасно, розділивши їх символом `';'`

Значення параметрів для керування режимом відображення дисплею показані в табл. 1.

Таблиця 1

Параметри SGR для керування режимом 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

Приклади керування режимами відображення дисплею:

`CSI3m` – відображення шрифтів `italic`;

`CSI3;5m` – відображення шрифтів `italic` з повільним блиманням.

Для керування кольором тексту і фону використовуються коди ANSI кольорів показані у табл. 2.

Таблиця 2

Коди ANSI кольорів

Код	Описання
0	Чорний
1	Червоний
2	Зелений
3	Жовтий
4	Голубий
5	Бордовий
6	Бірюзовий
7	Білий

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:

`CSI37m` – білий текст

`CSI47m` – білий фон

Можна об’єднати колір тексту і фону, задавши їх через символ `";"`:

`CSI31;40m` – текст червоний, фон – чорний.

ANSI `Esc`-символи можна послати у сесії терміналу командою `echo`. 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 редакторах використовується послідовність натискання клавіш `Ctrl-v`, а потім `Esc`. В результаті появиться символ **`^[`**. Таким чином отриманий код CSI можна перевірити у командному рядку:

```
>echo ^[[41mThis is a test^[[40m
```

У команді `printf` escape символ (`Esc - ^[`) можна задати послідовністю символів `'\e'`, наприклад:

```
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 Підекрес \e[0m \n"
printf "\e[31;1m Червоний жирний \e[0m \n"
printf "\e[37;4;2m Білий, підкреслений, слабо інтенсивний \e[0m \n"
```

Зміна кольору стрічки повідомлення командного рядка:

```
$ PS1='\e[1;34m\u@h:\w\e[0m\$'
```

де `\e[1;34m` – встановлення синього кольору тексту;
`\u` – user;
`\h` – host;
`\w` – повний шлях (`\W` – остання частина повного шляху).

Для зміни кольорів тексту і фону можна використати команду `tput`:
`tput setaf color` – задати колір тексту;
`tput setaf 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;

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 вивід 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 (одиночний 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>
```

Автоматичний вибір розмірів при `height` і `width = 0`. Автоматичний вибір максимальних розмірів при `height` і `width = -1`. Автоматичний вибір глобальних розмірів, 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.

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] ділог 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
повертаж 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 свій вивід у стандартний потік STDOUT.

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 сценарії?
5. Формат і SGR параметри для керування режимом відображення дисплею і кольором тексту та фону?
8. Як ввести керуючі послідовності символів CSI в команді `printf`?
9. Як створити меню вибору з використанням команди `dialog`?
10. Які стандартні віджети можна створити командою `dialog`?
11. Як створити меню вибору з використанням команди `kdialog` середовища KDE X WINDOW?
12. Які стандартні віджети можна створити командою `kdialog`

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 для виведення 5 основних віджетів команди `dialog`.

11. Написати сценарій для виведення 5 основних віджетів команди `kdialog`.

Приклади для самостійної роботи

1. menu1.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
```

2. menu2.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 "^[[1m\t0. Вихід\n\n^[[0m^[[44;33m"
  echo -en "\t\tВведіть вибір: "
  read -n 1 option
}

echo "^[[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 "^[[0m"
clear
```

4. 4.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”
```

5. Зміна кольору стрічки повідомлення командного рядка:

```
$ PS1=’\e[1;34m\u@h:\w\e[0m\$’
```

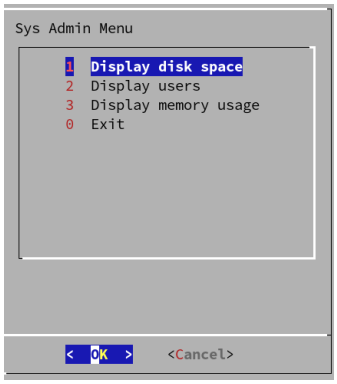
6. Використання команди tput:

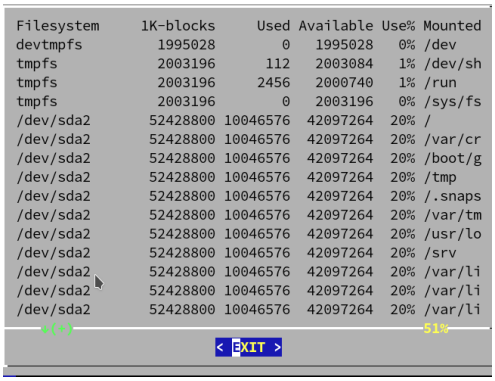
```
#!/bin/bash
tput setaf 1
echo ”Червоний”
tput setaf 2
echo ”Зелений”
tput setaf 4
```

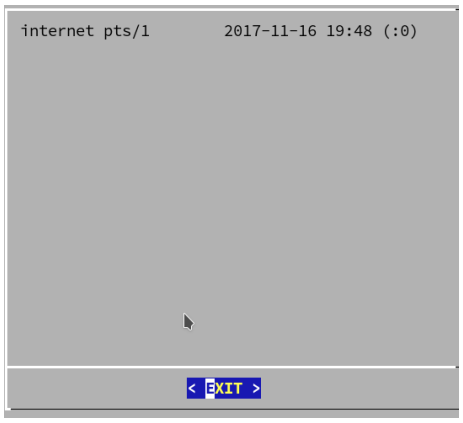
```
echo ”Синій”
tput sgr0
```

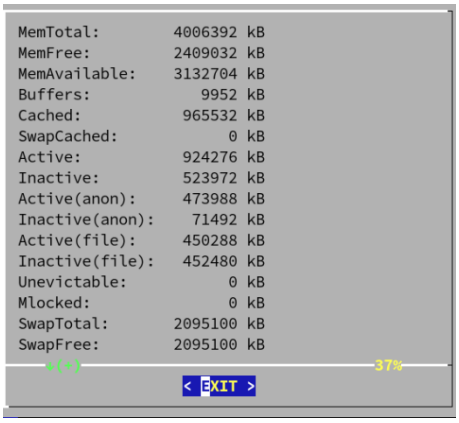
6. menu6.sh - використання команди dialog для створення графічних віджетів

```
#!/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
```





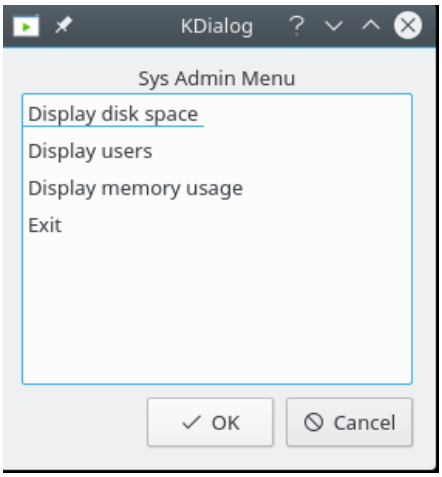


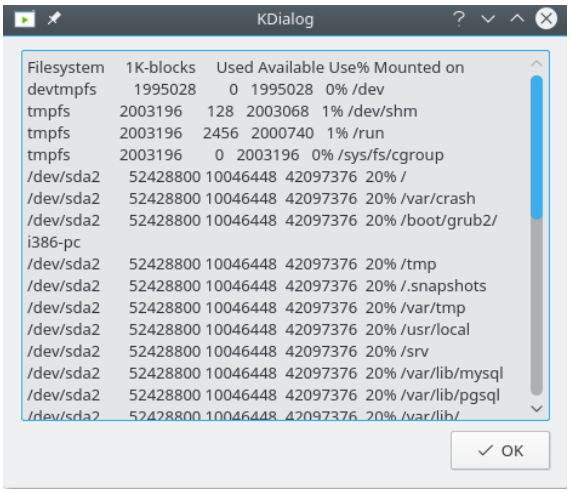


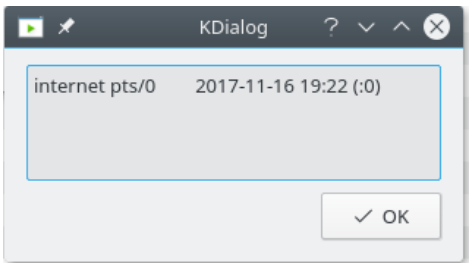
7. menu7.sh - використання команди kdialog для створення віджетів у графічному середовищі KDE)

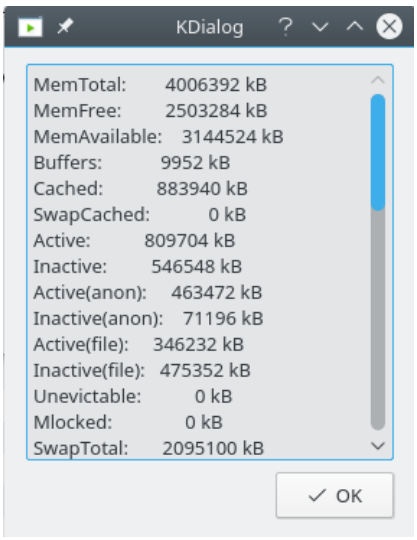
```
#!/bin/bash
# створення меню командою kdialog

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
```









Лабораторна робота № 10

10. Потокові редактори sed, gawk

Мета роботи: вивчити основи роботи з потоковими редакторами `sed` і `gawk`.

Теоретичний матеріал лекції, література [1-2].

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` і `gawk`.

2 Потоковий редактор `sed`

Потоковий редактор `sed` маніпулює даними 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 редактора, 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] {command1 command2…} file
sed [-n][-e] 'command' file(s)
sed [-n] –f gawk_script file(s)
```

`options` – опції команди `sed`:

`-e 'command'` – додає команди задані в командному рядку (якщо більше одної), до команд які обробляють вхід;

`-f gawk_script` – додає команди задані в файлі, до команд які обробляють вхід;

`-n` – не виводить вихід кожної команди, але чекає на команду `print`.

`{command1 command2 …}` – задаються окремі команди, які застосовуються до одної адреси і вводяться з нового рядка у командному рядку.

`file -` 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`.

Для запуску `sed` з читанням команд із файлу потрібно команди записати у файл `script1`, а дані – у файл `data1`

```
$ cat script1
s/test/big test/
$ cat data1
This is a test
$ sed -f script1 data1
$ cat data1
This is a big test
```

Потоковий редактор `sed` має велику кількість команд і форматів (`$ sed --help`) для редагування даних.

Команда підставлення 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;
`p` – друк вмісту файлу шаблона;
`w file` – вивести у файл результат підставлень.

Замість розділювача стрічок ”/” можна використовувати “!” (у випадку наявності в тексті символів /):

```
$ sed 's!/bin/bash!/bin/csh!' /etc/passwd
```

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
```

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
```

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
```

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
```

Рядки із заданими номерами `n,m` вилучаються командою `n,m`**`d`** `(delete)`. Вилучення 2-го рядку, 2- і 3-го рядка, 2-го рядка і до кінця тексту:

```
sed '3d'   data2
sed '2,3d' data2
sed '3,$d' data2
```

Рядки із заданим шаблоном `/pattern/` вилучаються командою `/pattern/`**`d`**. Вилучення порожніх рядків:

```
sed /^$/d file > new_file
```

Вилучення рядків, які починаються або закінчуються на xxx:

```
sed '/^xxx/d; /xxx$/d' file > new_file
```

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' data2
$ sed '3a\Цей рядок всталяється після 3-го рядка' data2
```

Команда заміни всього рядка за номером або за шаблоном **c** (change):

```
sed '[address]c\new line'  file
$ sed '3c\Цей рядок замінить третій рядок' data2
$ sed '/abcd/c\Цей рядок замінить рядок за шаблоном' data2
```

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/'
```

Для друкування 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.

Команда **p** друкує задані рядки (-n подавити вивід решти рядків):

```
$ echo "Тестовий рядок" | sed 'p'
$ sed -n '/number 3/p' data2
$ sed -n '2,3p' data2
$ sed -n '/3/{p s/line/test/p }' data2
```

Друк усіх рядків з номерами і друк рядків за шаблоном

```
$ sed '=' data1
$ sed -n '/number 4/{ = p }' data2
```

Друк рядків з недрукованими ASCII символами

```
$ sed -n 'l' data2
```

Команди запису **w** (write) і читання **r** (read) файлів

```
[address]w filename
[address]r filename
```

Записати у файл 1-й і 2-й рядок із вхідного потоку:

```
$ sed '1,2w test' data6
```

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 файла 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
```

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 правил, які звичайно записуються в окремих рядках:

```
шаблон  { дія }
шаблон  { дія }
…
```

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
```

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;

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 декілька команд:

```
$ 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
```

Виконання команд програми перед і після 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
```

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
$
```

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
```

3.3 Введення і виведення 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, …
```

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
```

Всередині конструкцій циклів можуть використовуватися інструкції `break`, `continue`, `next`, `nextfile`, `exit`.

3.5 Масиви

Масиви `gawk` є асоціативними, тобто елемент масиву складається з індекса і значення. Індексом може бути як число, так і стрічка.

```
Index 3 Value 30          Index "dog" Value "chien"
Index 1 Value "foo"       Index "cat" Value "chat"
Index 0 Value 8           Index "one" Value "un"
Index 2 Value ""          Index 1 Value "un"
```

Доступ до елементів масиву:

```
{                                 {
  if ($1 > max)                     for (i = 1; i <= NF; i++)
    max = $1                        used[$i] = 1
    arr[$1] = $0                  }
}
                                  END {
END {                               for (x in used) {
for (x = 1; x <= max; x++)             if (length(x) > 10) {
  if (x in arr)                           ++num_long_words
    print arr[x]                          print x
}                                      }
                                    }
                                    print num_long_words, " > 10 chars"
                                  }
```

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])`.

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:

```
function name([parameter-list])
{
   body-of-function
}
```

```
function myprint(num)
{
   printf "%6.3g\n", num
}
if ($3 > 0) { myprint($3) }
```

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 команд заміни рядків і окремих символів у редакторі `sed`?
7. Які можливості і формат команд друку у редакторі `sed`?
8. Який формат команд читання і запису даних у файл у редакторі `sed`?
9. Яка різниця між потоковими редакторами `sed` і `gawk`?
10. Який формат програми `gawk`?
11. Як прочитати окремі поля рядків тексту командами програми `gawk`?
12. Які правила запису команд програми `gawk`?
13. Як виконати команди програми `gawk` до і після обробки даних?

Завдання.

1. Написати команди редактора `sed` для заміни англійських символів кириличними.

2. Використовуючи команди редактора `sed` надрукувати у табличному вигляді поля записів файлу `/etc/passwd`.

3. Написати команди редактора `sed`, які будуть вставляти оцінки за екзамен з файлу “Оцінки” у файлі “Відомість” (формат запису - прізвище : оцінка, порядок оцінок відповідає порядку призвіщ). Кількість оцінок і призвіщ взяти рівною 10.

4. Виконати пункт 3 з використання програми gawk. Підрахувати і вивести середній бал групи після списку прізвищ з оцінками.

5. Написати програму `gawk`, які розміщують записи файлу ”Відомість” в порядку зменшення оцінок в іншому файлі ”Відомість_sorted”.

6. Написати програму `gawk`, яка читає файл Сі програми і записує її в інший файл як Java програму. Файл Сі програми:

```
int main(argv, **argc)
{
   i=5;
   while (i--) printf(”i=%d\n”,i);
   return 0;
}
```

7. Написати програму `gawk`, яка читає файл Сі програми (із пр. 6) і записує її в інший файл як Python програму.

8. Написати програму `gawk`, яка читає файл Сі програми (із пр. 6) і записує її в інший файл як Ada програму.

9. Написати програму `gawk`, яка зчитує довільне речення з консолі і виводить його на екран в рамці із символів ‘*’. Наприклад:

```
                                    ************************************
Студенти проходять виробничу практику => *Студенти проходять виробничу практику*
```

****

10. Написати програму gawk, яка читає текстовий файл і підраховує кількість повторів кожної букви абетки, загальну кількість букв і процент використання букв.

11. Написати програму gawk, яка шифрує і розшифровує текстовий файл за алгоритмом Цезаря.

12. . Написати програму gawk, яка шифрує і розшифровує текстовий файл за алгоритмом абетки в'язниці.

Приклади для самостійної роботи

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 cat.
The quick brown fox jumps over the lazy cat.
The quick brown fox jumps over the lazy cat.
The quick brown fox jumps over the lazy cat.
The quick brown fox jumps over the lazy cat.
```

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)
$ sed -n 's/test/trial/p' data4
# зберігання підстановок у новий файл
$  sed 's/test/trial/w data5a' data4a
This is a test line.
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 повторються.

```
# 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.

```
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}'
```

Лабораторна робота № 11

11. Регулярні вирази

Мета роботи: навчитися писати і застосовувати регулярні вирази.

Теоретичний матеріал лекції, література [1-7].

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 двигуна регулярних виразів. У світі Linux набули поширення два двигуни регулярних виразів:

- базовий двигун регулярних виразів POSIX (the POSIX Basic Regular Expression (BRE) engine);
- розширений двигун регулярних виразів POSIX (the POSIX Extended Regular Expression (ERE) engine).

2. Базові регулярні вирази

Найбільш поширеним шаблоном базових регулярних виразів (РВ) є співпадіння символів шаблону і тексту, наприклад пошук послідовності символів `test` у вхідному потоці і їх друк у всіх випадках співпадінь

```
$ 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 РЕ 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 $2

$ 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 елемент ”.” використовується для позначення любого символу в заданій позиції, крім символу нового рядка “\n”.

```
$ echo ”Книжка на столі”  | sed –n ‘/.a/p’
Книжка
на
```

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 користувача є класи спеціальних символів, табл.1.

Таблиця 1 – **Класи спеціальних символів**

Клас	Описання
[[:alpha:]]	співпадіння любих алфавітних символів [a-zA-z]
[[:alnum:]]	співпадіння любих число-алфавітних символів [a-zA-z0-9]
[[:blank:]]	співпадіння забілу або символу табуляції
[[:digit:]]	співпадіння цифр від 0 до 9 [0-9]
[[:lower:]]	співпадіння малих алфавітних символів [a-z]
[[:print:]]	співпадіння любих друкованих символів
[[:punct:]]	співпадіння розділових символів
[::space:]]	співпадіння символів забіл, Tab, NL, FF, VT, CR
[[:upper:]]	співпадіння 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 "abbb" | sed -n '/ab*c/p'
abbb
```

Для задання любого числа любих символів використовується шаблон ”.*” .

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 співпадіння 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 розглядується 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}'
```

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 шаблон регулярного виразу для пошуку рядків, що починаються зі слова then.
2. Написати шаблон регулярного виразу для пошуку рядків, що починаються словом do і закінчуються словом done.
3. Написати шаблон регулярного виразу для фільтрування електронної пошти всіх користувачів mail.pu.if.ua.
4. Написати шаблон регулярного виразу для фільтрування номерів мобільних телефонів користувачів Київ Стар.
5. Написати шаблон регулярного виразу для підрахунку кількості файлів в усіх підкаталогах каталогу /home.

Приклади для самостійної роботи

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. Сивол ”*” 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
```

Список літератури

1. Гордеев В.В., Молчанов А.Ю. Системное программное обеспечение. – СПб.: Питер, 2001. – 736.

2. Шеховцов В.А. Операційні системи. – К.: Видавнича група BHV, 2005. – 576 с.

3. Блум Ричард, Бреснахэн Кристина. Командная строчка Linux и сценарии оболочки. Библия пользователя, 2-е узд.: Пер. с англ. – М.: ООО "И.Д. Вильямс", 2012. – 784 с.

4. Джонсон, Майкл К., Троан, Єрик В. Разработка приложений в среде Linux, 2-е изд.: Пер. с англ. –М.: "ООО И.Д. Вильямс", 2007. – 544 с.

5. С.Л. Скловская. Команды Linux. – СПб.: Питер, 2004. – 848 с.

6. Митчелл М., Оулдем Д., Самьюєл А. Программирование для Linux. Профессиональный поход. – М.: Вильямс, 2002. – 288 с.

7. C. Newham, B. Rosenblat. Learning the bash shell. Third Edition. O'Reilly, 2005. – 333 p.

8. Richurd Blum. Command line and shell scripting bible. Indianapolis, Indiana: Wiley Publishing. – 2008. – 809 p.

9. Arnold D. Robbins. GAWK: Effective gawk programming. A User's Guide for GNU Awk, Edition 4.1. Free Software Faundation, Inc. 2013. – 490 p.

10. Dale Dougherty, Arnold Robbins. Sed & awk. O'Really, Second Edition, 1997. – 432 p.

11. Тейнсли Д. Язык Shell Linux и Unix. – Пер. с англ. – СПб.: БХВ-Петербург, 2001. – 512с.

12. Феникс Т., Шварц Р. Изучаем Perl. – Пер. с англ. – СПб.: БХВ-Петербург, 2002. – 288с.

13. Брент Б. Уєлш, Кен Джонс, Джеффри Хоббс. Практическое программирование на Tcl и Tk. – Пер. с англ. – СПб.: Изд. дом "Вильямс", 2004. – 1138 с.