

SSBT's College of Engineering & Technology, Bambhori, Jalgaon
Department of Computer Applications

Practical: 02

DOP:

DOC:

Title: Linux Commands and Shell Programming

Objective : To Demonstrate and study Linux Commands .

1. Theory and Lab Task:

Shell Scripting with Bash

Shell scripts are plain text files that contain a sequence of commands that are run by a shell, one after another. Bash is the default shell in most modern Linux distributions and we will leverage its programming capabilities to create simple scripts. As we gain experience, we can use what we have learned to develop more robust programs.

System administrators often use shell scripts to automate routine tasks. As a rule of thumb, if a task has to be performed periodically (even when it's only once a month), it needs to be automated.

Structure of a Shell Script

The first line in a shell script must indicate the shell (also known as *interpreter*) that will be used to execute it. For Bash, this means

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

bash

The above line must be followed by the commands that should be run by the shell, one per line.

Although preferences between system administrators may vary, effective and well-maintained shell scripts often include the following sections:

Header

The script header is a commented-out section where the developer can include items such as:

- Description/purpose of the script
- Revision history
- License terms and/or copyright notice

The shell ignores blank and commented-out lines. The former are only meant as information for the author, other reviewers, and people using the program. To comment out a line, simply place a # sign at the beginning.

A common header looks as follows:

```
1#
```

```
=====
```

```

2# SCRIPT NAME: systeminfo.sh
3
4# PURPOSE: Demonstrate simple Bash programming
concepts
5
6# REVISION HISTORY:
7
8# AUTHOR          DATE          DETAILS
9# -----
-----
10# Gabriel A. Cnepa    2017-10-21    Initial version
11
12# LICENSE: CC Attribution-ShareAlike 4.0
International
13#
=====
=====

```

bash

Body

The body of the script is where the sequence of commands is placed, one per line. A command can be executed directly by the shell or have its output be saved into a container known as *variable*. You can think of variables as boxes where we can store a fixed value (such as text or a number) or the output of a command, for later reuse.

As a best practice, system administrators often use comments in the body to indicate what a given line of code is supposed to do. This also serves as a reminder for himself/herself and others who will later work on the same file.

To store the output of a command in a variable, enclose the command between parentheses and preface them with the dollar sign. Thus, in `MYVAR=$(command)`, the variable MYVAR contains the output of `command`, where `command` can be any command executed by the shell. To use the contents of MYVAR in the script, add `$MYVAR` wherever it is needed.

For example:

```

1echo "Starting to run the script..."
2# VARIABLE ASSIGNMENT
3# Show hostname:
4HOST=$(hostname)
5# User executing the script:

```

```

6CURRENTUSER=$(whoami)
7# Current date:
8CURRENTDATE=$(date +%F)
9# Host IP address:
10IPADDRESS=$(hostname -I | cut -d ' ' -f1)
11
12# SHOW MESSAGES
13echo "Today is $CURRENTDATE"
14echo "Hostname: $HOST ($IPADDRESS)"
15echo "User info for $CURRENTUSER:"
16grep $CURRENTUSER /etc/passwd

```

bash

Let's put everything together (`#!/bin/bash`, header, and body) and save as `systeminfo.sh`. Next, make the file executable and run it:

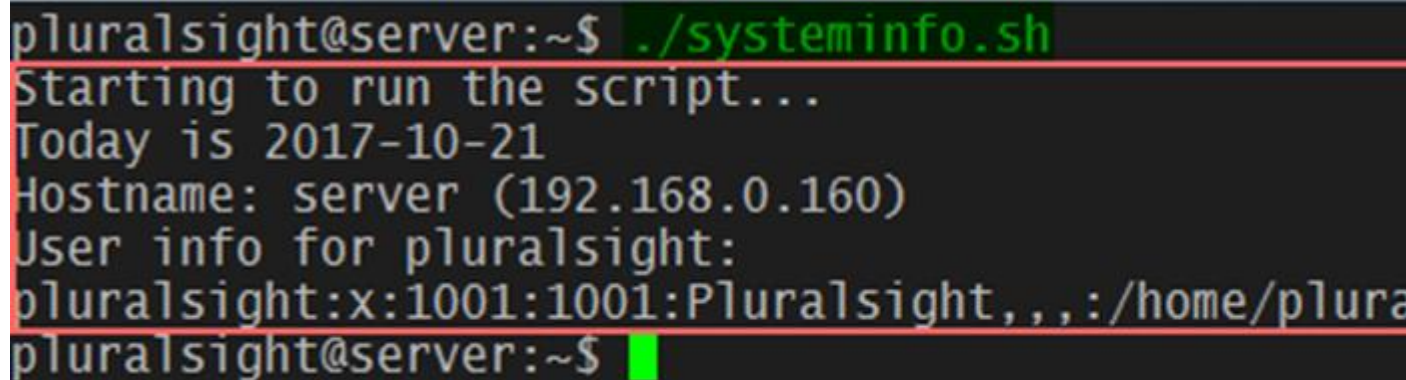
```

1chmod a+x systeminfo.sh
2./systeminfo.sh

```

bash

The following image shows the output of the script:



```

pluralisight@server:~$ ./systeminfo.sh
Starting to run the script...
Today is 2017-10-21
Hostname: server (192.168.0.160)
User info for pluralisight:
pluralisight:x:1001:1001:Pluralisight,,,:/home/plura
pluralisight@server:~$

```

As you can see in the above image, the `echo` Bash built-in allows us to embed variables and fixed text inside double quotes. Thus, the content of the given variable(s) will be part of the output, along with the text.

Conditional Statements for Shell Scripts

When you need to execute different series of commands depending on a condition, the shell provides a standard `if/elif/else` construct, similar to most programming languages.

The syntax of a basic conditional statement is shown in pseudo-code as follows. Note that there must be a space after the opening bracket and before the closing one:

```

1if [ condition 1 ]
2then

```

```
3  commands
4elif [ condition 2 ]
5  commands
6else
7  commands
8fi
```

bash

The elif section is optional and used only when needed to test for a specific alternative to the condition specified through the if.

Let's use the following example to illustrate:

```
1#!/bin/bash
2CURRENTDAYOFTHEMONTH=$(date +%d)
3
4if [ $CURRENTDAYOFTHEMONTH -le 10 ]
5then
6    echo "We are within the first 10 days of the
month";
7elif [ $CURRENTDAYOFTHEMONTH -le 20 ]
8then
9    echo "We are within the first 20 days of the
month";
10else
11    echo "We are within the last 10 days of the
month";
12fi
```

bash

The above script returns different messages depending on the current day of the month. The first if tests whether the current day is less or equal to 10. If this condition evaluates to true, the message *We are within the first 10 days of the month* is displayed and the other conditions are not tested. Otherwise, elif checks if `$CURRENTDAYOFTHEMONTH` is less than or equal to 20. If that is the case, *We are within the first 20 days of the month* will be returned instead. If none of the previous conditions are met, the default else block will apply, returning *We are within the last 10 days of the month*.

Loops

Once in a while, some tasks will need to be executed repeatedly either a fixed number of times or until a specified condition is satisfied. That is where the for and while looping constructs come in handy. Additionally, at times you will need a script to choose between different courses of action based on the value of a given variable - we'll use the case statement for that.

The For Loop

This looping construct is used to operate on each element of a list of known items. Such a list can be specified explicitly (by listing each element one by one) or as the result of a command. The basic syntax of the for loop is illustrated in the following pseudo-code:

```
1for variable-name in list
2do
3    Run command on variable-name as $variable-name
4done
```

bash

variable-name is an arbitrary name that represents an item in list during each iteration. For example, we can change the permissions on files file1.txt, file2.txt, and file3.txt to 640 using a for loop very easily:

```
1for FILE in file1.txt file2.txt file3.txt
2do
3    chmod 640 $FILE
4done
```

bash

In the above example, the variable named FILE is used inside the loop (between the do and done lines) as \$FILE. During the first, second, and third iteration, \$FILE represents file1.txt, file2.txt, and file3.txt, respectively.

An alternative approach is to provide the list of files using the `ls` command and `grep` to only return the files where the names begin with the word *file*.

```
1for FILE in $(ls -1 | grep file)
2do
3    chmod 640 $FILE
4done
```

bash

This produces the same result as the previous example.

The While Loop

As opposed to the for loop, while is typically used when the number of iterations is not known beforehand or when using for is impractical. Examples include, but are not limited to, reading a file line by line, increasing or decreasing the value of a variable until it reaches a given value, or responding to user input.

The basic syntax of the while loop is:

```
1while condition is true
2do
```

```
3   Run commands here
```

```
4done
```

```
bash
```

To illustrate, let's consider two examples. First, let's read the `/etc/passwd` file line by line and return a message showing each username with its corresponding UID. This is an example of the case when we need to repeat an iteration an undetermined number of times.

The UID, or User ID, is an integer number that identifies each user in the third field in `/etc/passwd`. It can be returned for each individual account using the `id` command followed by the username.

```
1while read LINE
2do
3   USERNAME=$(echo $LINE | cut -d':' -f 1)
4   USERID=$(echo $LINE | cut -d':' -f 3)
5   echo "The UID of $USERNAME is $USERID"
6done < /etc/passwd
```

```
bash
```

In this example, the condition that is checked at the beginning of each iteration is whether we have reached the end of the file. The variable named `LINE` represents each line in `/etc/passwd`. This file is set as the input to the loop by using the `<` redirection operator. When each line is read, the contents of the first and third fields are stored in `USERNAME` and `USERID`, respectively.

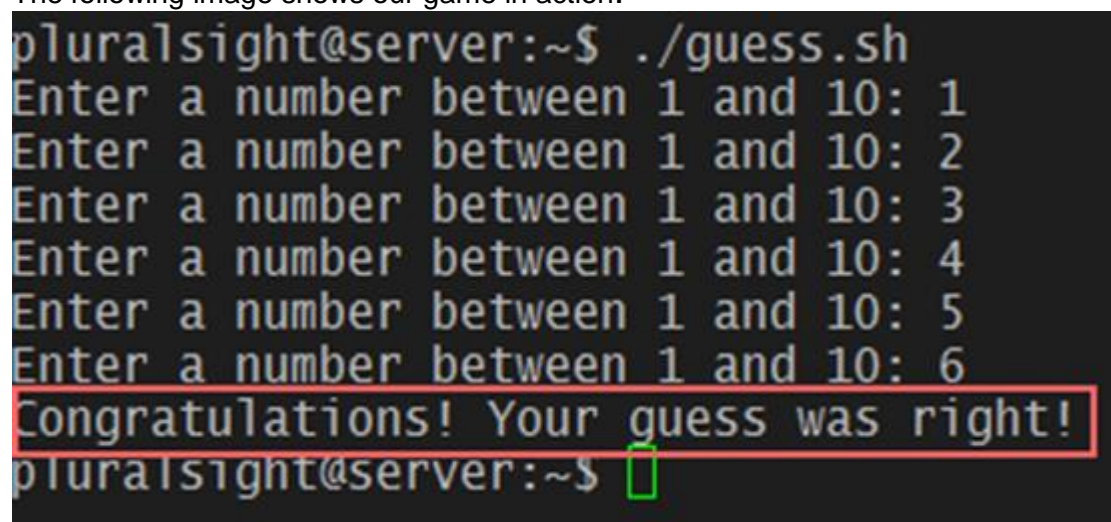
Let's take a look at the output of the while loop, which we have saved in a script named `users.sh` in the current working directory. The content of the output may vary from system to system depending on the number and names of user accounts.

```
pluralsight@server:~$ ./users.sh
The UID of root is 0
The UID of daemon is 1
The UID of bin is 2
The UID of sys is 3
The UID of sync is 4
The UID of games is 5
The UID of man is 6
The UID of lp is 7
The UID of mail is 8
The UID of news is 9
The UID of uucp is 10
The UID of proxy is 13
The UID of www-data is 33
The UID of backup is 34
The UID of list is 38
The UID of irc is 39
The UID of gnats is 41
The UID of nobody is 65534
The UID of systemd-timesync is 100
The UID of systemd-network is 101
The UID of systemd-resolve is 102
The UID of systemd-bus-proxy is 103
The UID of syslog is 104
The UID of _apt is 105
The UID of lxd is 106
The UID of messagebus is 107
The UID of uidd is 108
The UID of dnsmasq is 109
The UID of sshd is 110
The UID of gacanepe is 1000
The UID of pluralsight is 1001
The UID of student is 1002
pluralsight@server:~$
```


Finally, let's write a simple number-guessing game in a script called guess.sh. When the script is launched, a random number between 1 and 10 is generated and stored in the variable RANDOMNUM. The script then will expect input from the user and indicate if the guess is correct, less than, or greater than the right number. This will continue until the user guesses the number correctly. In this example, `$NUMBER != $RANDOMNUM` is enclosed within square brackets to indicate accurately what is the condition to test.

```
1#!/bin/bash
2RANDOMNUM=$(shuf -i1-10 -n1)
3
4NUMBER=0
5
6while [ $NUMBER != $RANDOMNUM ]
7do
8    read -p "Enter a number between 1 and 10: " NUMBER
9done
10echo "Congratulations! Your guess was right!"
bash
```

The following image shows our game in action:



```
pluralsight@server:~$ ./guess.sh
Enter a number between 1 and 10: 1
Enter a number between 1 and 10: 2
Enter a number between 1 and 10: 3
Enter a number between 1 and 10: 4
Enter a number between 1 and 10: 5
Enter a number between 1 and 10: 6
Congratulations! Your guess was right!
pluralsight@server:~$
```

A:- Linux Commands

Getting Information About the Command

Memorizing command options is usually not necessary and maybe a waste of time. Usually, if you are not using the command frequently, you can easily forget its options.

Most commands have a `--help` option, which prints a short message about how to use the command and exits:

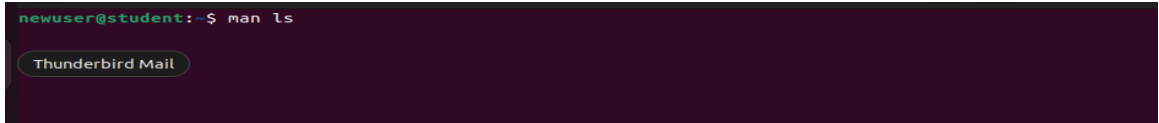
command_name --help

The man command

Almost all Linux commands are distributed together with man pages. A man or manual page is a form of documentation that explains what the command does, examples of how you run the command, and what arguments it accepts.

The man command is used to display the manual page of a given command.

man command_name



For example, to open the man page of the, ls command, you would type:

man ls

```
LS(1)                                User Commands                                LS(1)
NAME
  ls - list directory contents
SYNOPSIS
  ls [OPTION]... [FILE]...
DESCRIPTION
  List information about the FILES (the current directory by default).  Sort entries alphabetically if none of -cftuvSUX nor --sort is specified.
  Mandatory arguments to long options are mandatory for short options too.
  -a, --all
      do not ignore entries starting with .
  -A, --almost-all
      do not list implied . and ..
  --author
      with -l, print the author of each file
  -b, --escape
      print C-style escapes for nongraphic characters
  --block-size=SIZE
      with -l, scale sizes by SIZE when printing them; e.g., '--block-size=M'; see SIZE format below
  -B, --ignore-backups
      do not list implied entries ending with ~
  -c
      with -lt: sort by, and show, ctime (time of last modification of file status information); with -l: show ctime and sort by name; otherwise: sort by ctime, newest first
  -C
      list entries by columns
Manual page ls(1) line 1 (press h for help or q to quit)
```

To navigate the man pages, use the Arrow, Page Up, and Page Down keys. You can also press the Enter key to move one line at a time, the Space bar to move to the next screen, and the b key to go one screen back. To exit the man page, press the q key.

Navigating the File System

In Linux, every file and directory is under the root directory, the first or top-most directory in the directory tree. The root directory is referred to by a single leading slash /.

When navigating the file system on operating on files, you can use either the absolute or the relative path to the resource.

The absolute or full path starts from the system root /, and the relative path starts from your current directory.

Current Working Directory (pwd command)

The current working directory is the directory in which the user is currently working. Each time you interact with your command prompt, you are working within a directory.

Use the pwd command to find out what directory you are currently in:

pwd

The command displays the path of your current working directory:

/home/mohit

A terminal window titled 'Firefox Web Browser' showing the command 'pwd' being executed. The output is '/home/mohit'. The prompt is 'newuser@student: ~\$'.

Changing directory (cd command)

The cd (“change directory”) command is used to change the current working directory in Linux and other Unix-like operating systems.

When used without any argument, cd will take you to your home directory:

cd

To change to a directory, you can use its absolute or relative pathname.

Assuming that the directory Downloads exists in the directory from which you run the command, you can navigate to it by using the relative path to the directory:

cd Downloads

You can also navigate to a directory by using its absolute path:

cd ospractical

A terminal window showing the command 'mkdir ospractical2' being executed, followed by 'cd ospractical2'. The output is './ospractical2\$'.

Two dots (..), one after the other, represent the parent directory or, in other words, the directory immediately above the current one.

Suppose you are currently in the /usr/local/share directory. To switch to the /usr/local directory (one level up from the current directory), you would type:

cd ../

```
newuser@student:~$ cd ../
```

To move two levels up:

cd ../../

```
newuser@student:~$ cd ../../
```

To change back to the previous working directory, use the dash (-) character as an argument:

cd -

```
newuser@student:~$ cd ospract
newuser@student:~/ospract$ cd ~
newuser@student:~$
```

If the directory you want to change to has spaces in its name, you should either surround the path with quotes or use the backslash (\) character to escape the space:

cd ospract \ abc

```
newuser@student:~$ cd ospract \ abc
```

Working with Files and Directories

Listing directory contents (ls command)

The ls command list sinformation about files and directories within a directory.

When used with no options and arguments, ls displays a list in alphabetical order of the names of all files in the current working directory:

ls

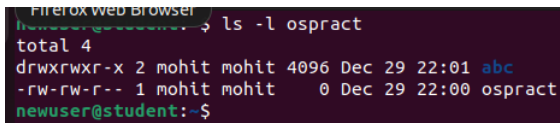
To list files in a specific directory, pass the path to the directory as an argument:

ls ospract

```
newuser@student:~$ ls ospract
abc  ospract
newuser@student:~$
```

The default output of the ls command shows only the names of the files and directories. Use the -l to print files in a long listing format:

ls -l ospract



```
newuser@student:~$ ls -l ospract
total 4
drwxrwxr-x 2 mohit mohit 4096 Dec 29 22:01 abc
-rw-rw-r-- 1 mohit mohit   0 Dec 29 22:00 ospract
newuser@student:~$
```

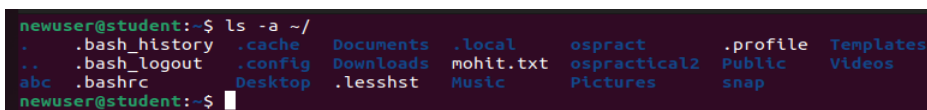
The output includes the file type, permissions, number of hard links, owner, group, size, date, and filename:

-rw-r--r-- 1 root root 337 Oct 4 11:31 /etc/hosts

The ls command doesn't list the hidden files by default. A hidden file is any file that begins with a period (.).

To display all files, including the hidden files, use the -a option:

ls -a ~/



```
newuser@student:~$ ls -a ~/
.      .bash_history  .cache  Documents  .local  ospract  .profile  Templates
..     .bash_logout  .config Downloads  mohit.txt ospractical2 Public    Videos
abc    .bashrc       Desktop  .lessht    Music    Pictures  snap
newuser@student:~$
```

Displaying file contents (cat command)

The cat command prints the contents of one or more files and merge (concatenate) files by appending one file's contents to the end of another file

To display the contents of a file on the screen, pass the file name to cat as an argument:

cat ospract

Creating files (touch command)

The touch command is used to update the timestamps on existing files and directories as well as to create new, empty files.

To create a file , specify the file name as an argument:

touch ospract.txt

If the file already exists, touch will change the file's last access and modification times to the current time.

Creating directories (mkdir command)

In Linux, you can create new directories (also known as folders) using the mkdir command.

To create a directory, pass the name of the directory as the argument to the command:

mkdir ospract2

```
newuser@student:~$ mkdir ospract2
newuser@student:~$
```

mkdir can take one or more directory names as its arguments.

If the argument is a directory name, without the full path, the new directory is created in the current working directory.

To create parent directories, use the -p option:

mkdir -p abc/xyz

```
newuser@student:~$ mkdir -p pqr\xyz
newuser@student:~$
```

The command above creates the whole directory structure.

When mkdir is invoked with the -p option, it creates the directory only if it doesn't exist.

Creating symbolic links (ln command)

A symbolic link (or symlink) is a special type of file that points to another file or directory.

To create a symbolic link to a given file, use the ln command with the -s option, the name of the file as the first argument, and the name of the symbolic link as the second argument:

ln -s source_filesymbolic_link

```
newuser@student:~/ospract$ ln -s /home/mohit/ospract.txt /home/mohit/ospract/
```

If only one file is given as an argument, ln creates a link to that file in the current working directory with the same name as the file it points to.

Removing files and directories (rm command)

To remove files and directories, use the rm command.

By default, when executed without any option, rm doesn't remove directories. It also doesn't prompt the user for whether to proceed with the removal of the given files.

To delete a file or a symlink, use the rm command followed by the file name as an argument:

rm ospract.txt

rm accepts one or more file or directory names as its arguments.

The -i option tells rm to prompt the user for each given file before removing it:

rm -i ospract.txt

```
newuser@student:~$ rm -i ospract.txt
rm: remove regular empty file 'ospract.txt'?
```

rm: remove regular empty file 'ospract.txt'?

Use the -d option to remove one or more empty directories:

rm -d dirname

```
newuser@student:~$ mkdir xyz
newuser@student:~$ rm -d xyz
newuser@student:~$
```

To remove non-empty directories and all the files within them recursively, use the -r (recursive) option:

rm -rf dirname

```
newuser@student:~$ rm -rf ospract
newuser@student:~$
```

The -f option tells rm never to prompt the user and to ignore nonexistent files and arguments.

```
newuser@student:~$ rm -f  
newuser@student:~$
```

ing files and directories (cp command)

The cp command allows you to files and directories.

To a file in the current working directory, use the source file as a first argument and the new file as the second:

cp file file_backup

```
newuser@student:~/ospractical2$ cp abc.txt xyz.txt  
newuser@student:~/ospractical2$
```

To a file to another directory, specify the absolute or the relative path to the destination directory. When only the directory name is specified as a destination, the copied file will have the same name as the original file.

cp file.txt /backup

```
newuser@student:~/ospractical2$ cp abc.txt xyz.txt  
newuser@student:~/ospractical2$
```

By default, if the destination file exists, it will be overwritten.

To a directory, including all its files and subdirectories, use the -R or -r option:

cp -R Pictures /opt/backup

```
Firefox Web Browser  
newuser@student:~$ cp -r abc.txt ospracticat2\abc
```

Moving and renaming files and directories (mv command)

The mv command (short from move) is used to rename and move and files and directories from one location to another.

For example, to move a file to a directory, you would run:

mv xyz.txt abc.txt

```
newuser@student:~$ cd ospractical2
newuser@student:~/ospractical2$ mv xyz.txt abc.txt
newuser@student:~/ospractical2$
```

To rename a file you need to specify the destination file name:

mv xyz.txt abc.txt

```
newuser@student:~$ cd ospractical2
newuser@student:~/ospractical2$ mv xyz.txt abc.txt
newuser@student:~/ospractical2$
```

The syntax for moving directories is the same as when moving files.

To move multiple files and directories at once, specify the destination directory as the last argument:

mv abc.txt xyz.txt /tmp

```
newuser@student:~/ospractical2$ mv abc.txt xyz.txt /tmp
```

Installing and Removing Packages

A package manager is a tool that allows you to install, update, remove, and otherwise manage distro-specific software packages.

Different Linux distributions have different package managers and package formats.

Only root or user with sudo privileges can install and remove packages.

Ubuntu and Debian (apt command)

Advanced Package Tool or APT is a package management system used by Debian-based distributions.

There are several command-line package management tools in Debian distributions, with apt and apt-get being the most used ones.

Before installing a new package first, you need to update the APT package index:

apt update


```
[sudo] password for mohit:
Hit:1 http://in.archive.ubuntu.com/ubuntu jammy InRelease
Get:2 http://security.ubuntu.com/ubuntu jammy-security InRelease [110 kB]
Get:3 http://in.archive.ubuntu.com/ubuntu jammy-updates InRelease [114 kB]
Get:4 http://in.archive.ubuntu.com/ubuntu jammy-backports InRelease [99.8 kB]
Get:5 http://security.ubuntu.com/ubuntu jammy-security/main amd64 DEP-11 Metadata [20.0 kB]
Get:6 http://in.archive.ubuntu.com/ubuntu jammy-updates/main amd64 DEP-11 Metadata [95.0 kB]
Get:7 http://in.archive.ubuntu.com/ubuntu jammy-updates/universe amd64 DEP-11 Metadata [257 kB]
Get:8 http://security.ubuntu.com/ubuntu jammy-security/universe amd64 DEP-11 Metadata [13.2 kB]
Get:9 http://in.archive.ubuntu.com/ubuntu jammy-updates/multiverse amd64 DEP-11 Metadata [944 B]
Get:10 http://in.archive.ubuntu.com/ubuntu jammy-backports/universe amd64 Packages [6,752 B]
Get:11 http://in.archive.ubuntu.com/ubuntu jammy-backports/universe i386 Packages [5,200 B]
Get:12 http://in.archive.ubuntu.com/ubuntu jammy-backports/universe amd64 DEP-11 Metadata [11.5 kB]
Get:13 http://in.archive.ubuntu.com/ubuntu jammy-backports/universe amd64 c-n-f Metadata [348 B]
Fetched 734 kB in 4s (196 kB/s)
```

The APT index is a database that holds records of available packages from the repositories enabled in your system.

To upgrade the installed packages to their latest versions, run:

apt upgrade

[illegible]

Installing packages is as simple as running:

apt install package_name

[illegible]

To remove an installed package , enter:

apt remove package_name

[illegible]

B:-Shell Scripting

1)write a Shell Scripting on simple calculator program.

Algorithm:-

1. Read Two Numbers
2. Input Choice (1-Addition, 2-Subtraction, 3-Multiplication, 4-Division)
3. if Choice equals 1
 Calculate $res = a + b$
 else If Choice equals 2
 Calculate $res = a - b$
 else if Choice equals 3
 Calculate $res = a * b$
 else if Choice equals 4
 Calculate $res = a / b$
4. Output Result, res

Program:-

```
#!/bin/bash
j=1
while [ $j -eq 1 ]
do
echo "Enter the First Operand;"
read f1
echo "Enter the second operand:"
read f2
echo "1-> Addition"
echo "2-> Subtraction"
echo "3-> Multiplication"
echo "4-> Division"
echo "Enter your choice"
read n
case "$n" in
1)
echo "Addition"
f3=$((f1+f2))
echo "The result is:$f3"
;;
2)
echo "Subtraction"
```

```

let "f4=$f1 - $f2"
echo "The result is:$f4"
;;
3)
echo "Multiplication"
let "f5=$f1 * $f2"
echo "The result is:$f5"
;;
4)
echo "Division"
let "f6=$f1 / $f2"
echo "The result is:$f6"
;;
esac
echo "Do you want to continue(press:1 otherwise press any key to
quit)"
read j
done
OUTPUT:-

```

```

student@student-MS-7C85:~$ bash cal.sh
Enter the First Operand;
4
Enter the second operand:
6
1-> Addition
2-> Subtraction
3-> Multiplication
4-> Division
Enter your choice
1
Addition
The result is:10
Do you want to continue(press:1 otherwise press any key to
quit)
1
Enter the First Operand;
5
Enter the second operand:
3
1-> Addition
2-> Subtraction
3-> Multiplication
4-> Division
Enter your choice
2
Subtraction
The result is:2
Do you want to continue(press:1 otherwise press any key to
quit)

```

2)write a Shell Scripting on factorial of given number.

Algorithm:-

1. Get a number
2. Use for loop or while loop to compute the factorial by using the below formula

3. $\text{fact}(n) = n * n-1 * n-2 * \dots 1$

4. Display the result.

Program:-

```
#!/bin/bash
# Recursive factorial function

factorial()
{
    product=$1

    # Defining a function to calculate factorial using recursion
    if((product <= 2)); then
        echo $product
    else
        f=$((product - 1))

# Recursive call

f=$(factorial $f)
f=$((f*product))
echo $f
fi
}

# main program
# reading the input from user
echo "Enter the number:"
read num

# defining a special case for 0! = 1
if((num == 0)); then
    echo 1
else
    #calling the function
    factorial $num
fi
```

OUTPUT:-

```
student@student-MS-7C85:~$ bash factorial.sh
Enter the number:
5
120
student@student-MS-7C85:~$ bash factorial.sh
Enter the number:
9
362880
```

3)write a Shell Scripting on greater number.

Algorithm:-

1. Get three numbers. Say num1, num2, num2
2. If (num1 > num2) and (num1 > num3)
echo value of num1
3. elif(num2 > num1) and (num2 > num3)
echo value of num2
4. Otherwise,
echo value of num3

Program:-

```
#!/bin/bash
echo "Enter Num1"
read num1
echo "Enter Num2"
read num2
echo "Enter Num3"
read num3

if [ $num1 -gt $num2 ] && [ $num1 -gt $num3 ]
then
    echo $num1
elif [ $num2 -gt $num1 ] && [ $num2 -gt $num3 ]
then
    echo "grater num is:"
    echo $num2
else
    echo "grater num is:"
    echo $num3
```

fi

OUTPUT:-

```
student@student-MS-7C85:~$ bash greater.sh
Enter Num1
5
Enter Num2
8
Enter Num3
7
grater num is:
8
student@student-MS-7C85:~$
```

04)write a Shell Scripting on to check number is odd or even.

Algorithm:-

1. 'clear' command to clear the screen.
2. Read a number which will be given by user.
3. Use `expr \$n % 2`. If it equal to 0 then it is even otherwise it is odd.
4. Use if-else statements to make this program more easier.
5. Must include 'fi' after written 'if-else' statements.

Program:-

```
#!/bin/bash
echo "Enter a number : "
read n
rem=$(( $n % 2 ))
if [ $rem -eq 0 ]
then
echo "$n is even number"
else
echo "$n is odd number"
fi
```

```
student@student-MS-7C85:~$ bash oddno.sh
Enter a number :
3
3 is odd number
student@student-MS-7C85:~$ bash oddno.sh
Enter a number :
6
6 is even number
student@student-MS-7C85:~$
```

05)write a Shell Scripting on to check String is pelindrom.

Algorithm:-

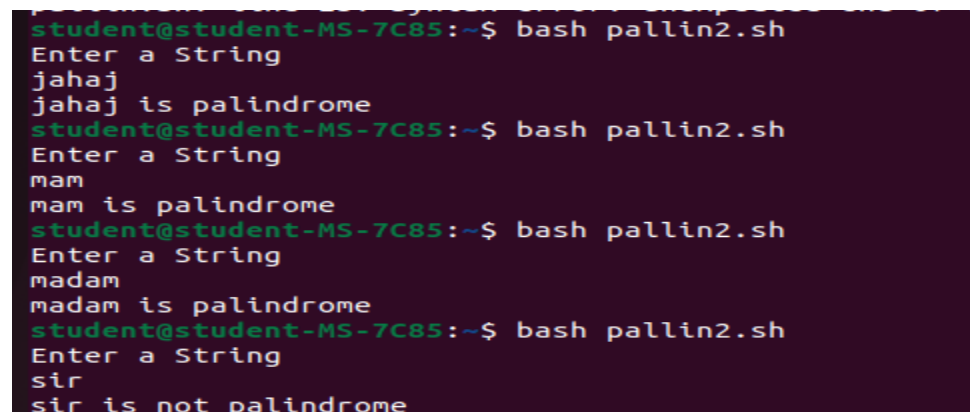
1. Get the given string.
2. Reverse the string.
3. Compare the given string with reversed string if they are equal print "palindrome" otherwise "not palindrome".

Program:-

```
#!/bin/bash
echo "Enter a String"
read input
reverse=""

len=${#input}
for (( i=$len-1; i>=0; i-- ))
do
    reverse="$reverse${input:$i:1}"
done
if [ $input == $reverse ]
then
    echo "$input is palindrome"
else
    echo "$input is not palindrome"
fi
```

OUTPUT:-



```
student@student-MS-7C85:~$ bash pallin2.sh
Enter a String
jahaj
jahaj is palindrome
student@student-MS-7C85:~$ bash pallin2.sh
Enter a String
mam
mam is palindrome
student@student-MS-7C85:~$ bash pallin2.sh
Enter a String
madam
madam is palindrome
student@student-MS-7C85:~$ bash pallin2.sh
Enter a String
sir
sir is not palindrome
```

06) write a Shell Scripting on to check number is prime.

Algorithm:-

- 1; Loop from 2 to $n/2$, i as loop variable
- 2 :- if number is divisible, print "The number is not prime" and flag = 1;

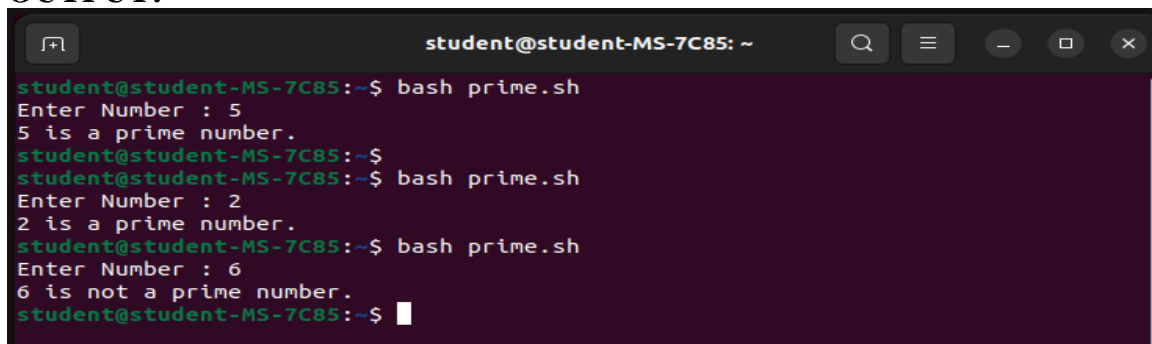
3 :- if flag != 1, then print "The number is prime".

4 :- Exit.

Program:-

```
#!/bin/bash
echo -e "Enter Number : \c"
read n
for((i=2; i<=$n/2; i++))
do
    ans=$(( n%i ))
    if [ $ans -eq 0 ]
    then
        echo "$n is not a prime number."
        exit 0
    fi
done
echo "$n is a prime number."
```

OUTPUT:-

A terminal window titled 'student@student-MS-7C85: ~' showing the execution of a shell script named 'prime.sh'. The script prompts the user to 'Enter Number :'. Three test cases are shown: 1. Input 5: '5 is a prime number.' 2. Input 2: '2 is a prime number.' 3. Input 6: '6 is not a prime number.' The prompt returns to the shell after each execution.

```
student@student-MS-7C85:~$ bash prime.sh
Enter Number : 5
5 is a prime number.
student@student-MS-7C85:~$ bash prime.sh
Enter Number : 2
2 is a prime number.
student@student-MS-7C85:~$ bash prime.sh
Enter Number : 6
6 is not a prime number.
student@student-MS-7C85:~$
```

07)write a Shell Scripting on reverse number.

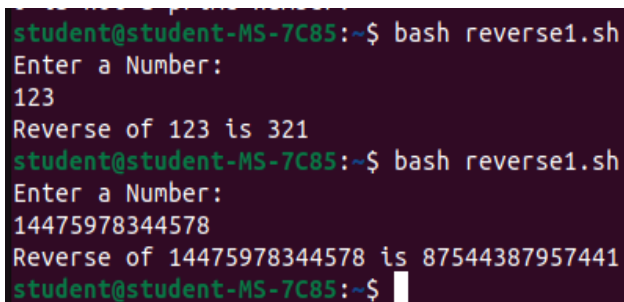
Algorithm:-

1. Initialize rev_num = 0
2. Loop while num > 0
 - (a) Multiply rev_num by 10 and add remainder of num
divide by 10 to rev_num
$$\text{rev_num} = \text{rev_num} * 10 + \text{num} \% 10;$$
 - (b) Divide num by 10
- 3.Return rev_num

Program:-

```
#!/bin/bash
echo "Enter a Number:"
read a
rev=0
sd=0
or=$a
while [ $a -gt 0 ]
do
sd=`expr $a % 10`
temp=`expr $rev \* 10`
rev=`expr $temp + $sd`
a=`expr $a / 10`
done
echo "Reverse of $or is $rev"
```

OUTPUT:-



```
student@student-MS-7C85:~$ bash reverse1.sh
Enter a Number:
123
Reverse of 123 is 321
student@student-MS-7C85:~$ bash reverse1.sh
Enter a Number:
14475978344578
Reverse of 14475978344578 is 87544387957441
student@student-MS-7C85:~$
```

08)write a Shell Scripting on sum of number.

Algorithm:-

1. Get N (Total Numbers).
2. Get N numbers using loop.
3. Calculate the sum.
4. Print the result.

Program:-

```
#!/bin/bash
# Take input from user and calculate sum.

read -p "Enter first number: " num1
read -p "Enter second number: " num2
```

```
sum=$(( $num1 + $num2 ))
```

```
echo "Sum is: $sum"
```

OUTPUT:-

```
student@student-MS-7C85:~$ bash sumofno.sh
Enter first number: 5
Enter second number: 6
Sum is: 11
student@student-MS-7C85:~$ bash sumofno.sh
Enter first number: 6
Enter second number: 7
Sum is: 13
```

09)write a Shell Scripting on greeting.

Program:-

```
#!/bin/bash
h=$(date +"%H")
if [ $h -gt 6 -a $h -le 12 ]
then
echo good morning
elif [ $h -gt 12 -a $h -le 16 ]
then
echo good afternoon
elif [ $h -gt 16 -a $h -le 20 ]
then
echo good evening
else
echo good night
fi
```

OUTPUT:-

```
student@student-MS-7C85:~$ bash greeting.sh
good afternoon
student@student-MS-7C85:~$
```

10)write a Shell Scripting on given number pyramid.

Program:-

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

```
read -p "How many levels? : " n
```

```
for((i = 0; i < n; i++))
```

```
do
```

```
k=0
```

```
while((k < $((i+1))))
```

```
do
```

```
echo -e "$((i+1))\c"
```

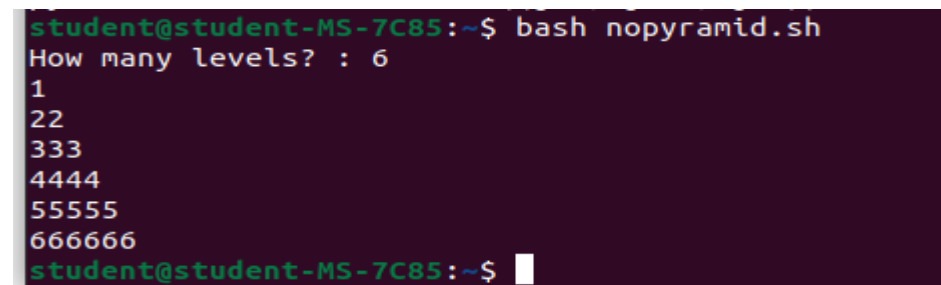
```
k=$((k+1))
```

```
done
```

```
echo " "
```

```
done
```

OUTPUT:-



```
student@student-MS-7C85:~$ bash nopyramid.sh
How many levels? : 6
1
22
333
4444
55555
666666
student@student-MS-7C85:~$
```

5. Conclusion:

In this practical we have learned the basic linux command ,shell scripting structure, Control statements in shell script and basic shell scripting programme.

Submitted By

:

Sign :

Name :

Roll No :

Checked By :