MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
A constituent institution of Manipal University

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

CERTIFICATE

This is to certify that Ms./Mr. …………………...…………………………………

Reg. No.: …..…………… Section: ……..…………… Roll No.: ………………...

has satisfactorily completed the lab exercises prescribed for OPERATING SYSTEMS LAB [CSE 3142] of Third Year B. Tech. Degree at MIT, Manipal, in the academic year 2024-2025.

Date: ……...................................

Signature
Faculty in Charge

CONTENTS

Course Objectives

- Illustrate and explore system calls related to Linux operating system.
- Learn process management and thread programming concepts which include scheduling algorithms and inter process communication.
- Understand the working of memory management schemes, disk scheduling algorithms, and page replacement algorithms through simulation.

Course Outcomes

At the end of this course, students will be able to:

- Execute Linux commands, shell scripting using appropriate Linux system calls.
- Design thread programming, simulate process management and inter process communication techniques.
- Implement the memory management, disk scheduling and page replacement algorithms.

Evaluation Plan

- Internal Assessment Marks : 60%
    - ✓ Continuous Observation component (for each experiment): 2 x 12 = 24 marks
    - ✓ Continuous Execution component (for each experiment) : (1+2) x 12 = 36 marks
    - ✓ Total marks of the 12 experiments will sum up to 60
- End semester assessment of 2-hours duration: 40 Marks

INSTRUCTIONS TO THE STUDENTS

Pre-Lab Session Instructions

1. Students should carry the Class notes, Lab Manual and the required stationery to every lab session.
2. Be in time and follow the Instructions from Lab Instructors.
3. Must Sign in the log register provided.
4. Make sure to occupy the allotted seat and answer the attendance.
5. Adhere to the rules and maintain the decorum.

In-Lab Session Instructions

- Follow the instructions on the allotted exercises given in Lab Manual.
- Show the program and results to the instructors on completion of experiments.
- On receiving approval from the instructor, copy the program and results in the Lab record.
- Prescribed textbooks and class notes can be kept ready for reference if required.

General Instructions for the exercises in Lab

- The programs should meet the following criteria:
  - Programs should be interactive with appropriate prompt messages, error messages if any, and descriptive messages for outputs.
  - Programs are properly indented and comments should be given whenever it is required.
  - Use meaningful names for variables and procedures.
- Plagiarism (copying from others) is strictly prohibited and would invite severe penalty during evaluation.
- The exercises for each week are divided under three sets:
  - Solved exercise
  - Lab exercises - to be completed during lab hours
  - Additional Exercises - to be completed outside the lab or in the lab to enhance the skill.

- In case a student misses a lab class, he/she must ensure that the experiment is completed at students end or in a repetition class (if available) with the permission of the faculty concerned but credit will be given only to one day's experiment(s).
- Questions for lab tests and examination are not necessarily limited to the questions in the manual, but may involve some variations and/or combinations of the questions.
- A sample note preparation is given later in the manual as a model for observation.

THE STUDENTS SHOULD NOT

- Carry mobile phones while working with computer.
- Go out of the lab without permission.

LAB NO.: 1 **Date:**

LINUX BASIC COMMANDS, SHELL CONCEPTS AND FILE FILTERS

Objectives:

In this lab, the student will be able to:

- Learn to Linux basic commands
- Understand the working of commands and important shell concepts and file filters
- Write and execute basic commands in a Shell

shell a utility program that enables the user to interact with the Linux operating system. Commands entered by the user are passed by the shell to the operating system for execution. The results are then passed back by the shell and displayed on the user's display. There are several shells available like Bourne shell, C shell, Korn shell, etc. Each shell differs from the other in Command interpretation. The most popular shell is bash.

shell prompt a character at the start of the command line which indicates that the shell is ready to receive the commands. The character is usually a '%' (percentage sign) or a '$' (dollar sign).

For. e.g.

Last login : Thu April 11 06:45:23

$ **_** (This is the shell prompt, the cursor shown by the **_** character).

Linux commands are executable binary files located in directories with the name bin (for binary). Many of the commands that are generally used are located in the directory /usr/bin.

echo is a command for displaying any string in the command prompt.

For e.g. $ echo "Welcome to MIT Manipal"

Environment variables: Shell has built in variables which are called environment variables. For e.g. the user who has logged in can be known by typing

$**echo** $USER

The above will display the current user's name.

When the command name is entered, the shell checks for the location of the command in each directory in the PATH environment variable. If the command is found in any of the directories mentioned in PATH, then it will execute. If not found, will give a message "Command not found".

COMMONLY USED LINUX COMMANDS

who: Unix is a system that can be concurrently used by multiple users and to know the users who are using the system can be known by a **who** command. For e.g. Current users are kumar, vipul and raghav. These are the user ids of the current users.

$ **who** [Enter] kumar pts/10 May 1
09.32 vipul pts/4 May 1 09.32
raghav pts/5 May 1 09.32

The first columns indicates the user name of the user, second column indicates the terminal name and the third column indicates the login time. To know the user who has invoked the command can be known by the following command. For e.g. if kumar is the user who has typed the who command above then, $ **who am i** [Enter] kumar pts/10 May 1 09.32 **ls**: UNIX system has a large number of files that control its functioning and users also create files on their own. These files are stored in separate folders called directories. We can list the names of the files available in this directory with **ls** command. The list is displayed in the order of creation of files.

$ **ls** [Enter]

README
chap01 chap02
chap03 helpdir
progs

In the above output, **ls** displays a list of six files. We can also list specific files or directories by specifying the file name or directory names. In this we can use regular expressions.

For e.g. to list all files beginning with chap we can use the following command.

$ ls chap* [Enter] chap01

chap02 chap03

To list further detailed information we can use **ls -l** command, where **-l** is an option between the command and filenames. The details include, file type, file or directory access permissions, number of links, owner name, group name, file or directory size, modification time and name of file or directory.

$ ls -l chap* [Enter]

```
-rw-r- - r- -l kumar users        5670 Apr 3 09.30 chap01
-rw-r- - r- -l kumar users        5670 Feb 23  09.30 chap02
-rw-r- - r- -l kumar users        5670 Apr 30  09.34 chap03
```

The argument beginning with hyphen is known as option. The main feature of option is it starts with hyphen. The command **ls** prints the columnar list of files and directories. With the **–l** option it displays all the information as shown above.

General syntax of **ls** command:

ls –[options][file list][directory list]

In Linux, file names beginning with period are hidden files, are not normally displayed in **ls** command. To display all files, including the hidden ones, use option **–a** in **ls** command as shown below:

$ ls -a

$ ls / will display the name of the files and sub-directories under the root directory.

pwd: This command gives the present working directory where the user is currently located.

$ **pwd**

/home/kumar/pis

cd: To move around in the file system use cd (change directory) command. When used with argument, it changes the current directory to the directory specified as argument, for instance:

$ **pwd**
/home/kumar

$ **cd** progs

$ **pwd**
$ /home/kumar/progs

cd .. : To change the working directory to the parent of the current directory we need to use **$ cd** ..
.. (double dot) indicates parent directory. A single dot indicates current directory.

cat: **cat** is a multipurpose command. Using this we can display a file, create a file as well as concatenate files vertically.

$ **cat >** filename[Enter] cat >
os.txt

Welcome to Manipal. (This the content which will be placed in file with filename)

[Ctrl D] End of input

$_ (comes to the shell prompt)

The above command will create a file named os.txt in the current directory. To see the contents of the file.

$ **cat** os.txt[Enter]

Welcome to Manipal.

To display a file we can use **cat** command as shown above.

We can use **cat** for displaying more than one file, one after the other by listing the files after **cat**. For e.g.

$ cat os.txt lab.txt

will display os.txt followed with lab.txt **cp**: To copy the
contents of one file to another.

Syntax: **cp** sourcefilename targetfilename [Enter]

This command is also used to copy one or more files to a directory. The syntax of this form of **cp** command is

Syntax : **cp** filename(s) directoryname

If the file **os.txt** in current directory i.e. /home/kumar/pis needs to be copied into /home directory then it will be done as follows.

$ **cp** os.txt /home/ **OR $ cp** os.txt ../../ **mv**: This command renames or moves files. It has two distinct function: It renames a file or a directory and it moves a group of files to a different directory.

Syntax: **mv** oldfilename newfilename Syntax of another form of this command is

mv file(s) directory **mv** doesn't create a copy of the file, it merely renames it. No additional space is consumed on disk for the file after renaming. To rename the file chap01 to man01,

$ mv chap01 man01.

If the destination file doesn't exist, it will be created. For the above example, **mv** simply replaces the filename in the existing directory with the new name. By default **mv** doesn't prompt for overwriting the destination file if it exist.

The following command moves three files to the progs directory: $ **mv** chap01 chap02 chap03 progs

mv can also be used to rename a directory for instance pis to pos:

$ **mv** pis pos **rm**: This command deletes one or more files. Syntax: **rm** filename

The following command deletes three files

$ **rm** chap01 chap02 chap03[Enter]

A file once deleted can be recovered subject to conditions by using additional software. **rm** won't normally remove a directory but it can

remove files from one or more directories. It can remove two chapters from the progrs directory by using:

$ **rm** progrs/chap01 progrs/chap02

mkdir: Directories are created by **mkdir** command. The command is followed by the name of the directories to be created.

Syntax: **mkdir** directoryname

$ **mkdir** data [Enter]

This creates a directory named data under the current directory.

$ **mkdir** data dbs doc

The above command creates three directories with names data, dbs and doc.

rmdir : Directories are removed by **rmdir** command. The command is followed by the name of the directory to be removed. If a directory is not empty, then the directory will not be removed.

Syntax: **rmdir** directoryname

$ **rmdir** patch [Enter]

The command removes the directory by the name patch.

In Linux every file and directory has access permissions. Access permissions define which users have permission to access a file or directory. Permissions are three types, read, write and execute. Access permissions are defined for user, group and others.

For e.g. If access permission is only read for user, group and others, then it will be r- -r--r- -

Access permissions can also be represented as a number. This number is in octal system. An access permission represented in numerical octal format is called absolute permission. The absolute permission for the above is

444

If the access permission is read, write for user, read, execute for group and only execute for others then it will be, rw-r-x- -x

The absolute permission for the above is

651

chmod: changes the permission specified in the argument and leaves the other permissions unaltered. In this mode the following is the syntax.

Sytax: **chmod** category operation permission filename(s) **chmod** takes as its argument an expression comprising some letters and symbols that completely describe the user category and the type of permission being assigned or removed. The expression contains three components:

User category (user, group, others)

The operation to be performed (assign or remove a permission). The type of permission (read, write and execute)

The abbreviations used for these three components are shown in Table 1.1.

E.g. to assign execute permission to the user of the file xstart;

$ **chmod u+x xstart**

$ ls –l xstart

- rwxr- - r- -   l kumar   metal   1980   May 01 20:30 xstart.

The command assigns (+) execute (x) permission to the user (u), but other permissions remain unchanged. Now the owner of the file can execute the file but the other categories i.e. group and others still can’t. To enable all of them to execute this file:

$ **chmod ugo+x xstart**

$ **ls –l** xstart

- rwxr-x r- x l kumar metal 1980 May 01 20:30 xstart.

The string **ugo** combines all the three categories user, group and others. This command accepts multiple filenames in the command line:

$ **chmod u+x note note1 note3**

$ **chmod a-x, go+r xstart; ls –l xstart** (Two commands can be run simultaneously with ;)

- rw-r--rwx l kumar metal 1980 May 01 20:30 xstart.

Table 1.1: Abbreviations Used by chmod

Category	Operation	Permission
u- User	+ Assigns permission	r- Read permission
g- Group	- Removes permission	w- Write permission
o- Others	= Assigns absolute permission	x- Execute permission
a- All(ugo)		

Absolute Permissions:

Sometimes without needing to know what a file's current permissions the need to set all nine permission bits explicitly using **chmod** is done.

Read permission – 4 (Octal 100)

Write permission – 2 (Ocal 010)

Execute permission – 1 (Octal 001)

For instance, 6 represents read and write permissions, and 7 represents all permissions as can easily be understood from Table 1.2.

Table 1.2: Absolute Permissions

Binary	Octal	Permissions	Significance
000	0	---	No permissions
001	1	--x	Executable only
010	2	-w-	Writable only
011	3	-wx	Writable and executable
100	4	r--	Readable only
101	5	r-x	Readable and executable
110	6	rw-	Readable and writable
111	7	rwx	Readable, writable and executable

$ **chmod 666 xstart; ls –l xstart**

- rw-rw- rw -   l kumar   metal   1980   May 01 20:30 xstart.

The 6 indicates read and write permissions (4 + 2).

date: This displays the current date as maintained in the internal clock run perpetually. $ **date** [Enter]

clear: The screen clears and the prompt and cursor are positioned at the top-left corner. $ **clear [Enter] man**: is used to display help file related to a command or system call. Syntax: **man {command name/system call name} e.g. man** date **man** open **wc**: displays a count of lines, words and characters in a file. **e.g. wc os.txt** 1 3 19 os.txt

Syntax: **wc [-c | -m | -C] [-l] [-w] [file….]** Options: The following options are supported:

-c Count bytes.

-m Count characters.

-C Same as – m,

-l Count lines

-w Count words delimited by white space characters or new line characters.

If no option is specified the default is –lwc (count lines, words, and bytes).

Redirection Operators

For any program whether it is developed using C, C++ or Java, by default three streams are available known as input stream, output stream and error stream. In programming languages, to refer to them some symbolic names are used (i.e. they are system defined variables).

The following operators are the redirection operators

1. > standard output operator

> is the standard output operator which sends the output of any command into a file.

Syntax: command > file1 e.g. **ls > file1**

Output of the **ls** command is sent to a file1. First, file file1 is created if not exists otherwise, its content is erased and then output of the command is written.

E.g.: **cat file1 > file2**

Here, file2 get the content of file1.

E.g.: **cat file1 file2 file3 > file4**

This creates the file file4 which gets the content of all the files file1, file2 and file3 in order.

2. < standard input operator

< operator (standard input operator) allows a command to take necessary input from a file.

Syntax: **$ command < file**

E.g.: **cat<file1**

This displays output of file file1 on the screen.

E.g.: **cat <file1 >file2**

This makes cat command to take input from the file file1 and write its output to the file file2. That is, it works like a **cp** command.

3. >> appending operator

Similarly, >> operator can be used to append standard output of a command to a file.

E.g.: **command>>file1**

This makes, output of the given command to be appended to the file1. If the file1 doesn't exist, it will be created and then standard output is written. **4. << document operator**

There are occasions when the data of your program reads is fixed and fairly limited. The shell uses the << symbols to read data from the same file containing the script. This is referred to as **here document**, signifying that the data is here rather than in a separate file. Any command using standard input can also take input from a here document.

Example.:

#!/bin/bash cat
<<DELIMITER

hello this is a
here document
DELIMITER

This gives the output:

hello this is a
here document

Shell Concepts

This section will describe some of the features that are common in all of the shells.

1. Wild-card: The metacharacters that are used to construct the generalized pattern for matching filenames belong to a category called wild-cards.

List of shell's wild-cards:

Wild-card	**Matches**
*	Any number of characters including none
?	A single or zero character
[ijk]	A single character - either an i, j or k
[x –z]	A single character between x and z
[!ijk]	A single character that is not an i, j or k. [!x–
z]	A single character not between x and z.

{pat1, pat2, ….} pat1, pat2, etc.

Example: Consider a directory structure /home/kumar which have the following files:

README

chap01 chap02

chap03 helpdir

progs

Then with the below command the following output would be displayed.

$ **ls chap***

chap chap01 chap02 chap03

$ **ls .***

.bash_profile .exrc .netscape .profile

2. **Pipes:** Standard input and standard output constitute two separate streams that can be individually manipulated by the shell. If so then one command can take input from the other. This is possible with the help of pipes.

   Assume if the **ls** command which produces the list of files, one file per line, use redirection to save this output to a file:

   $ **ls > user.txt**

   $ **cat user.txt**

   The file shows the list of files.

   Now to count the number of files:

   $ **ls | wc – l**

   The above command gives the number of files. This is how | (pipe) is used.

   There's no restriction on the number of commands to be used in pipe.

3. **Command substitution:** The shell enables the connecting of two commands in yet another way. While a pipe enables a command to obtain its standard input from the standard output of another command, the shell enables one or more command arguments to be obtained from the standard output of another command. This feature is called command substitution. $ **echo The date today is `date`**

   The date today is Sat May 6 19:01:56 IST 2019

   $ **echo "There are total `ls | wc –l ` files and sub-directory in the current directory**
   There are 15 files in the current directory.

4. **Sequences:** Two separate commands can be written in one line using ";" in sequences.
   $ **chmod 666 xstart; ls –l xstart**

5. **Conditional Sequences:** The shell provides two operators that allow conditional execution - the && and ||, which typically have this syntax:
   cmd1 && cmd2 cmd1 ||
   cmd2

The && delimits two commands; the command cmd2 is executed only when cmd1 succeeds.

The || operator plays inverse role; the second command cmd2 is executed only when the first command cmd1 fails.

Note: All built-in shell commands returns non-zero if they fail. They return zero on success.

e.g: if there is a program hello.c which displays 'Hello World' on compilation and execution. Then the following command in conditional sequences could be used to display the same:

$ **cc hello.c && ./a.out**

This command displays the output 'Hello World' if the compilation of the program succeeds. Similarly in case the compilation fails for the program the following output 'Error' could be displayed with the following command:

$ **cc hello.c || echo 'Error'**

File Filters commands in Linux:

1. **head:** To see the top 10 lines of a file - $ **head <file name>** To see the top 5 lines of a file - $ **head -5 <file name>**
2. **tail:** To see last 10 lines of a file - $ **tail < file name>**
   To see last 20 lines of a file - $ **tail -20 <file name>**

3. **more:** To see the contents of a file in the form of page views - $ more <file name>
   $ **more f1.txt**

4. **grep:** To search a pattern of word in a file, **grep** command is used.

   Syntax: **$ grep < word name> < file name>**
   $ **grep hi file_1**
   To search multiple words in a file

   $ **grep -E 'word1|word2|word3' <file name>**
   $ **grep -E 'hi|beyond|good' file_1**
5. **sort:** This command is used to sort the file.
   $ **sort <file name>**
   $ **sort file_1**
   To sort the files in reverse order

   $ **sort -r <file name>**

   To display only files

   $ ls –l | grep "^-"

   To display only directories

   $ ls –l | grep "^d"

Lab Exercises:

1. Write shell commands for the following.
   i) To create a directory in your home directory having 2 subdirectories.
   ii) In the first subdirectory, create 3 different files with different
   content in each of them.
   iii) Copy the first file from the first subdirectory to the second subdirectory.
   iv) Create one more file in the second subdirectory, which has the output of the number of users and number of files.
   v) To list all the files which starts with either a or A. vi)To count the number of files in the current directory
   vi) To count the number of files in the current directory.
   vii) Display the output if the compilation of a program succeeds.
   viii) Count the number of lines in an input file.
2. Execute the following commands in sequence: i) date ii) ls iii) pwd

Additional Exercises:

1. Write shell commands for the following.
   i) To display an error message if the compilation of a program fails.
   ii) To write a text block into a new file. iii.List all the files.
   iii) To count the number of users logged on to the system.

LAB NO.: 2 **Date:**

SHELL SCRIPTING 1

Objectives:

In this lab, the student will be able to:

- Understand the importance of scripts
- Write and execute shell scripts

The Linux shell is a program that handles interaction between the user and the system. Many of the commands that are typically thought of as making up the Linux system are provided by the shell. Commands can be saved as files called scripts, which can be executed like a program.

SHELL PROGRAMS: SCRIPTS

SYNTAX: **scriptname**

NOTE: A file that contains shell commands is called a script. Before a script can be run, it must be given execute permission by using **chmod** utility (chmod +x script). To run the script, only type its name. They are useful for storing commonly used sequences of commands to full-blown programs.

VARIABLES

Table 2.1 :Parameter Variables

$@	an individually quoted list of all the positional parameters
$#	the number of positional parameters
$!	the process ID of the last background command
$0	The name of the shell script.
$$	The process ID of the shell script, often used inside a script for generating unique temporary filenames; for example /tmp/tmpfile_$$.
$1, $2, …	The parameters given to the script.
$*	A list of all the parameters, in a single variable, separated by the first character in the environment variable IFS.

Lab Exercises:

1. Try the following shell commands
   $ echo $HOME, $PATH

   $ echo $MAIL

   $ echo $USER, $SHELL, $TERM

2. Try the following snippet, which illustrates the difference between local and environment variable:

   $ firstname=Rakesh ……local variables

   $ lastname=Sharma

   $ echo $firstname $lastname

   $ export lastname …..make “lastname” an envi var

   $ sh …..start a child shell

   $ echo $firstname $lastname

   $ ^D …..terminate child shell

   $ echo $firstname $lastname

3. Try the following snippet, which illustrates the meaning of special local variables:
   $ cat >script.sh echo the name of this script is $0 echo the first argument is $1 echo a list of all the arguments is $* echo this script places the date into a temporary file echo called $1.$$ date > $1.$$ # redirect the output of date
   ls $1.$$ # list the file rm $1.$$ # remove the file

   ^D

   $ chmod +x script.sh

   $ ./script.sh Rahul Sachin Kumble

NOTE: A shell supports two kinds of variables: local and environment variables. Both hold data in a string format. The main difference between them is that when a shell invokes a subshell, the child shell gets a copy of its parent shell’s environment variables, but not its local variables. Environment variables are therefore used for transmitting useful information between parent shells and their children.**Few predefined environment variables:**

$HOME pathname of our home directory

$PATH list of directories to search for commands

$MAIL pathname of our mailbox

$USER our username

$SHELL pathname of our login shell

$TERM type of the terminal **Creating a local variable:** variableName=value

Operations:

- Simple assignment and access
- Testing of a variable for existence
- Reading a variable from standard input
- Making a variable read only
- Exporting a local variable to the environment

Creating / Assigning a variable

Syntax: {name=value}

Example: $ firstName=Anand lastname=Sharma age=35 $ echo $firstname $lastname $age

$ name = Anand Sharma

$ echo $name

$ name = “Anand Sharma”

$ echo $name Accessing variable:

Syntax: $name / ${name}

Example: $ verb=sing

$ echo I like $verbing

Reading a variable from standard input:

Syntax: read {variable}+

Example: $ cat > script.sh

echo “Please enter your name:” read name echo your name is $name

^D

Read-only variables:

Syntax: readonly {variable}+

Example: $ password=manipal

$ echo $password

$ readonly password

$ readonly …..list

$ password=mangalore

Running jobs in Background

A multitasking system lets a user do more than one job at a time. Since there can be only one job in foreground, the rest of the jobs have to run in the background. There are two ways of doing this: with the shell's **& operator** and **nohup** command. The latter permits to log out while the jobs are running, but the former doesn't allow that.

$ **sort –o emp.lst &**

550

The shell immediately returns a number the PID of the invoked command (550). The prompt is returned and the shell is ready to accept another command even though the previous command has not been terminated yet. The shell however remains the parent of the background process. Using an & many jobs can be run in background as the system load permits.

In the above case, if the shell which has started the background job is terminated, the background job will also be terminated. **nohup** is a command for running a job in background in which case the background job will not be terminated if the shell is close. nohup stands for no hang up. e.g.

$ **nohup sort-o emp.lst &**

586

The shell returns the PID too. When the **nohup** command is run it sends the standard output of the command to the file **nohup.out**. Now the user can log out of the system without aborting the command.

JOB CONTROL

1. ps: ps is a command for listing processes. Every process in a system will have unique id called process id or PID. This command when used displays the process attributes.

$ **ps**

PID TTY TIME CMD

291 console 0:00 bash

This command shows the PID, the terminal TTY with which the process is associated, the cumulative processor time that has been consumed since the process has started and the process name (CMD).

2. **kill:** This command sends a signal usually with the intention of killing one or more process. This command is an internal command in most shells. The command uses one or more PIDs as its arguments and by default sends the SIGTERM(15) signal. Thus:

$ **kill 105** terminates the job having PID 105. The command can take many PIDs at a time to be terminated.

3. **sleep:** This command makes the calling process sleep until the specified number of seconds or a signal arrives which is not ignored. $ **sleep** 2

Lab Exercises:

1. Try the following, which illustrates the usage of **ps**:

$ (sleep 10; echo done) &
$ **ps**

2. Try the following, which illustrates the usage of **kill**:

$ (sleep 10; echo done) &
$ **kill pid** …..pid is the process id of background process

3. Try the following, which illustrates the usage of **wait**:

$ **(sleep 10; echo done 1) &** $ **(sleep 10; echo done 2) &**
$ **echo done 3; wait ; echo done 4** ….wait for children

NOTE: The following two utilities and one built-in command allow the listing controlling the current processes.

ps: generates a list of processes and their attributes, including their names, process ID numbers, controlling terminals, and owners **kill:** allows to terminate a process on the basis of its ID number **wait:** allows a shell to wait for one or all of its child processes to terminate

Sample Program: $ cat>script.sh echo there are $# command line arguments: $@

^D

$ script.sh arg1 arg2

Example: #!/bin/sh salutation="Hello" echo $salutation echo "The program $0 is now running" echo "The second parameter was $2" echo "The first parameter was $1" echo "The parameter list was $*" echo "The user's home directory is $HOME" echo "Please enter a new greeting" read salutation echo $salutation
echo "The script is now complete"

exit 0

If we save the above shell script as try.sh, we get the following output: $ ./try.sh foo bar baz

Hello

The program ./try.sh is now running

The second parameter was bar

The first parameter was foo

The parameter list was foo bar baz

The user’s home directory is /home/rick

Please enter a new greeting

Sire

Sire

The script is now complete

$

Lab Exercises:

Write Shell Scripts to do the following:

1. List all the files under the given input directory, whose extension has only one character
2. Write a shell script that accepts two command line parameters. First parameter indicates the directory and the second parameter indicates a regular expression. The script should display all the files and directories in the directory specified in the first argument matching the format specified in the second argument.
3. Count the number of users logged on to the system. Display the output as Number of users logged into the system.
4. Count only the number of files in the current directory.
5. Write a shell script that takes two sorted numeric files as input and produces a single sorted numeric file without any duplicate contents.
6. Write a shell script that accepts two command line arguments. First argument indicates format of file and the second argument indicates the destination directory. The script should copy all the files as specified in the first argument to the location indicated by the second argument. Also, try the script where the destination directory name has space in it.

Additional Exercises:

1. Write Shell Scripts to do the following

i) To list all the .c files in any given input subdirectory. ii)Write a script to include n different commands.