

Practical - 1

Aim:

A. LINUX Architecture

B. Types of OS- Linux, Flavors of LINUX UNIX, MAC, Window etc.

C. Difference Between Lollipop and Marshmallow Operating System Version.

Answer:

LINUX ARCHITECTURE

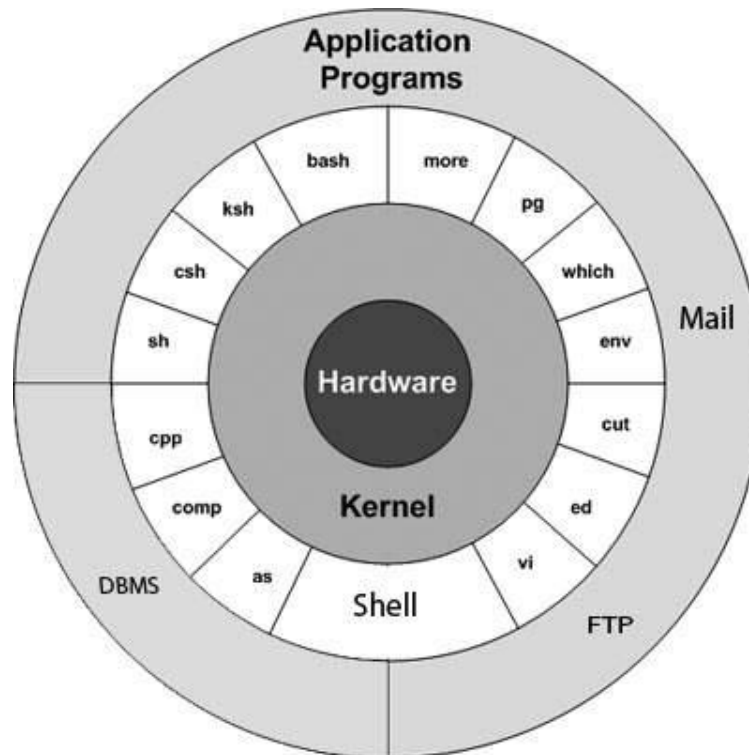
- **UNIX** is an operating system which was first developed in the 1960s, and has been under constant development ever since. By operating system, we mean the suite of programs which make the computer work. It is a stable, multi-user, multi-tasking system for servers, desktops and laptops.
- UNIX systems also have a graphical user interface (GUI) similar to Microsoft Windows which provides an easy to use environment. However, knowledge of UNIX is required for operations which aren't covered by a graphical program, or for when there are no windows interface available, for example, in a telnet session.

TYPES OF UNIX

- There are many different versions of UNIX, although they share common similarities. The most popular varieties of UNIX are:
- **Sun Solaris, GNU/Linux, and MacOS X.**



UNIX ARCHITECTURE



- **Hardware** – It consists of all hardware related information.
- **Kernel** – The kernel is the heart of the operating system. It interacts with the hardware and most of the tasks like memory management, task scheduling and file management. It allocates time and memory to programs and handles the file store and communications in response to system calls.
- **Shell** – The shell is the utility that processes your requests. When you type in a command at your terminal, the shell interprets the command and calls the program that you want. The shell uses standard syntax for all commands.
- **Commands and Utilities** – There are various commands and utilities which you can make use of in your day to day activities. cp, mv, cat and grep, etc. are few examples of commands and utilities.
- **Application Layer** – It is the outermost layer that executes the given external applications
- **Files and Directories** – All the data of UNIX is organized into files. All files are then organized into directories. These directories are further organized into a tree-like structure called the file system.

OPERATING SYSTEMS

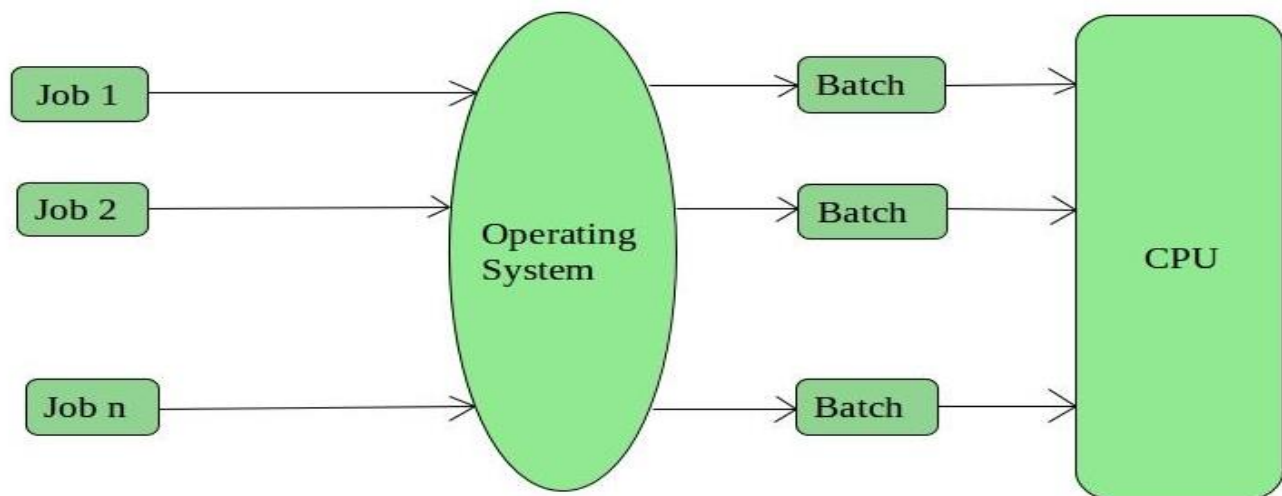
- An Operating System performs all the basic tasks like managing file, process, and memory. Thus operating system acts as manager of all the resources, i.e. resource manager. Thus operating system becomes an interface between user and machine.

TYPES OF OPERATING SYSTEMS

- Some of the widely used operating systems are as follows...

BATCH OPERATING SYSTEM

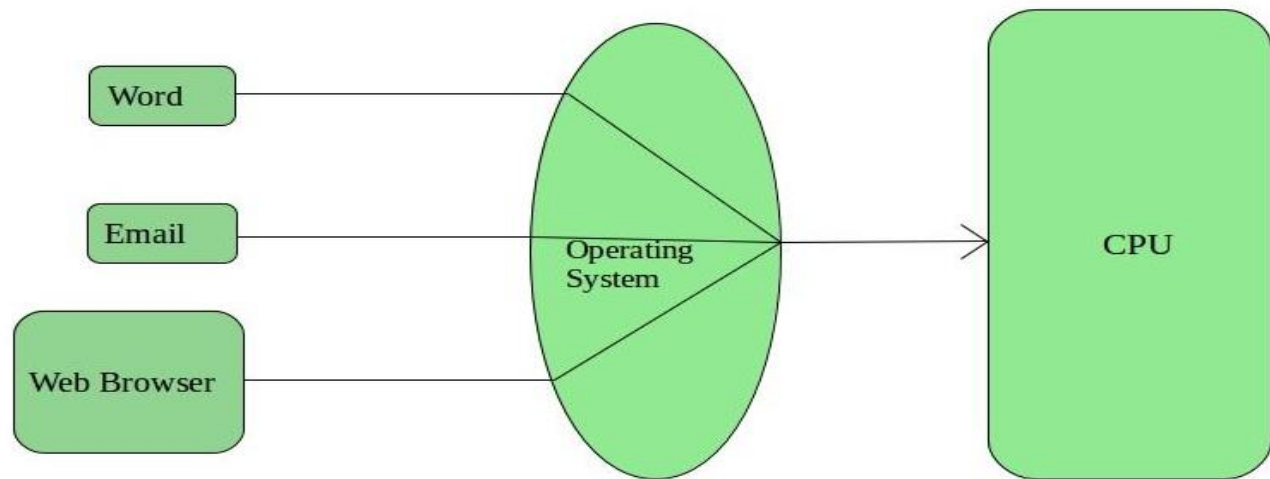
- This type of operating system does not interact with the computer directly. There is an operator which takes similar jobs having same requirement and group them into batches. It is the responsibility of operator to sort the jobs with similar needs.



- Examples of Batch based Operating System: Payroll System, Bank Statements etc.

TIME SHARING OPERATING SYSTEMS

- Each task is given some time to execute, so that all the tasks work smoothly. Each user gets time of CPU as they use single system. These systems are also known as Multitasking Systems. The task can be from single user or from different users also. The time that each task gets to execute is called quantum. After this time interval is over OS switches over to next task.



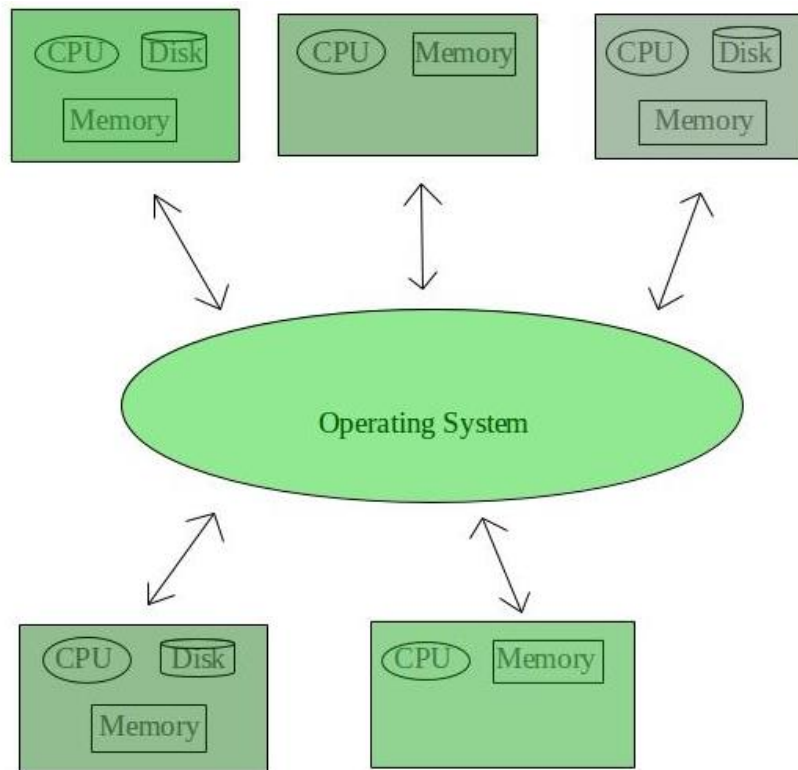
- Examples of Time-Sharing OSs are: Multics, Unix etc.

DISTRIBUTED OPERATING SYSTEM

- These types of operating system have various autonomous interconnected computers communicate each other using a shared communication network. Independent systems possess their own memory unit and CPU. These are referred as loosely coupled systems or distributed systems. These system's processors differ in size and function. The major benefit of working with these types of operating system is that it is always possible that one user can access the files or software which are not actually present on his system but on some other system connected within this network i.e., remote access is enabled within the devices connected in that network.
- Examples of Distributed Operating System are- LOCUS etc.

Types of Distributed Operating Systems

1. Client-Server Systems
2. Peer-to-Peer System

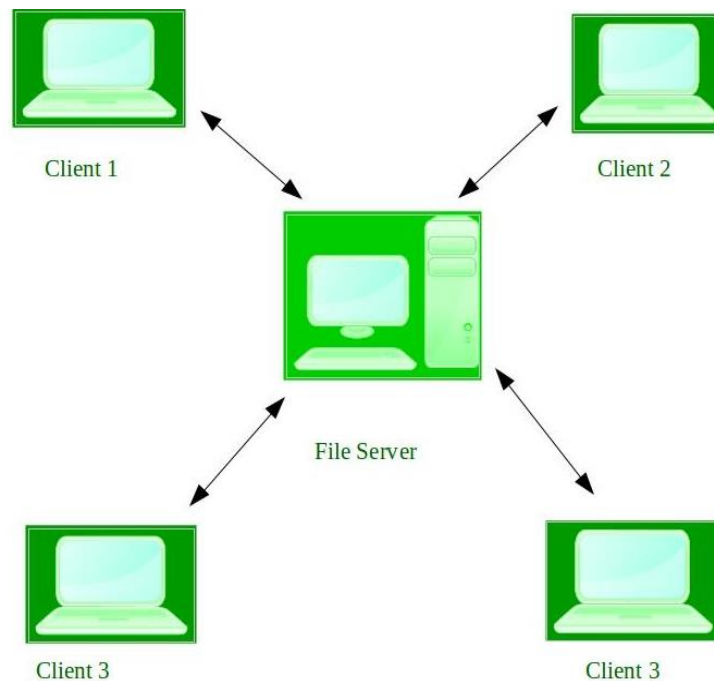


NETWORK OPERATING SYSTEM

- These systems run on a server and provide the capability to manage data, users, groups, security, applications, and other networking functions. These type of operating systems allow shared access of files, printers, security, applications, and other networking functions over a small private network. One more important aspect of Network Operating Systems is that all

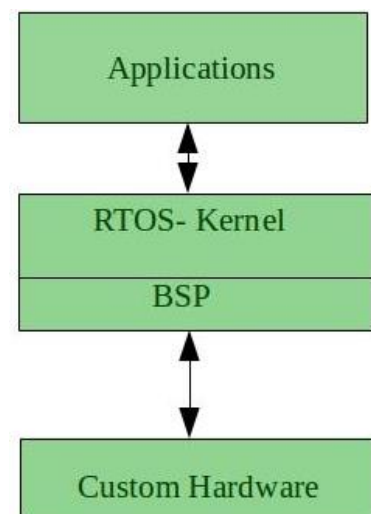
the users are well aware of the underlying configuration, of all other users within the network, their individual connections etc. and that's why these computers are popularly known as tightly coupled systems.

- Examples of Network Operating System are: Microsoft Windows Server 2003, Microsoft Windows Server 2008, UNIX, Linux, Mac OS X, Novell NetWare, and BSD etc.



REAL TIME OPERATING SYSTEM

- It is defined as an operating system known to give max for each of the critical operations that it performs, like OS calls and interrupt handling.
- Real time systems are used when there are time req. are very strict like missile systems, air traffic control systems, robots etc.



Two types of Real-Time Operating System are as follows:

- **Hard Real-Time Systems:**

These OSs are meant for the applications where time constraints are very strict and even the shortest possible delay is not acceptable. These systems are built for saving life like automatic parachutes or air bags which are required to be readily available in case of any accident. Virtual memory is almost never found in these systems.

- **Soft Real Time Systems:**

These OSs are for applications where for time-constraint is less strict.

SIMPLE BATCH SYSTEMS

- In this type of system, there is no direct interaction between user and the computer.
- The user has to submit a job (written on cards or tape) to a computer operator.
- Then computer operator places a batch of several jobs on an input device.
- Jobs are batched together by type of languages and requirement.
- Then a special program, the monitor, manages the execution of each program in the batch.
- The monitor is always in the main memory and available for execution.

MULTIPROGRAMMING BATCH SYSTEMS

- In this the operating system picks up and begins to execute one of the jobs from memory.
- Once this job needs an I/O operation operating system switches to another job (CPU and OS always busy).
- Jobs in the memory are always less than the number of jobs on disk (Job Pool).
- If several jobs are ready to run at the same time, then the system chooses which one to run through the process of CPU Scheduling.
- In Non-multi programmed system, there are moments when CPU sits idle and does not do any work.

- In Multiprogramming system, CPU will never be idle and keeps on processing.

CLUSTERED SYSTEMS

- Like parallel systems, clustered systems gather together multiple CPUs to accomplish computational work.
- Clustering is usually performed to provide high availability.
- A layer of cluster software runs on the cluster nodes. Each node can monitor one or more of the others.
- If the monitored machine fails, the monitoring machine can take ownership of its storage, and restart the application(s) that were running on the failed machine. The failed machine can remain down, but the users and clients of the application would only see a brief interruption of service.

MULTIPROCESSOR SYSTEMS

DESKTOP SYSTEMS

- Earlier, CPUs and PCs lacked the features needed to protect an operating system from user programs. PC operating systems therefore were neither multiuser nor multitasking. However, the goals of these operating systems have changed with time; instead of maximizing CPU and peripheral utilization, the systems opt for maximizing user convenience and responsiveness.
- These systems are called Desktop Systems and include PCs running Microsoft Windows and Apple Macintosh. Operating systems for these computers have benefited in several ways from the development of operating systems for mainframes.

HANDHELD SYSTEMS

- Handheld systems include Personal Digital Assistants (PDAs), such as Palm-Pilots or Cellular Telephones with connectivity to a network such as the Internet. They are usually of limited size due to which most handheld devices have a small amount of memory, include slow processors, and feature small display screens.

DIFFERENT FLAVOURS OF LINUX

- Linux comes in many different guises. The basic system is the same, but the look and feel and the subsystems around it are different. Each version is produced by a different organisation with its own ethos and aims.

UBUNTU

- <http://www.ubuntulinux.org/>
- Currently the most popular distro, GUI driven Linux, based on the Debian core. The most user-friendly version for Linux newbies Unusually, Canonical provides a free server version of Ubuntu for non- commercial use

FEDORA

- <http://fedoraproject.org/>
- Generally, Fedora requires a little more tinkering than Ubuntu or Mint, with the user having to resort to the command line more frequently, but is more reliable and ideally suited to the slightly more adventurous user

LINUX MINT

- <http://www.linuxmint.com/>
- Currently in third place is another user friendly version of Linux. Mint adds various and comes with more applications pre-installed.

PUPPY LINUX

- [http://puppylinux.org/main/Overview and Getting Started.htm](http://puppylinux.org/main/Overview%20and%20Getting%20Started.htm)
- Small footprint (100Mb) Linux, suitable for old hardware or low specification machines. Can run easily from a USB memory stick or Live CD/DVD. Includes a full desktop GUI, browser. Great for old / low specification hardware

TINYCORE

- <http://www.tinycorelinux.com/>

- Very small footprint (10Mb) Linux, suitable for old hardware or low specification machines / embedded devices. It ships with a minimal desktop GUI but no applications. Ideal for ancient hardware or occasional use

MEPIS LINUX

- <http://www.mepis.org/>
- There are two versions: the full version is known as Simply MEPIS but there is also a version called AntiX, which is suitable for old hardware or low specification machines. Both can run from a hard drive or direct from a Live CD/DVD.

ZORIN OS

- <http://zorin-os.com/index.html>
- It's easing the move from Windows to Linux. It comes with a full-set of applications pre-installed. Once again, well suited for those coming from a Windows background.

Practical – 2

Aim: Study of Unix Architecture and the following Unix commands with option.

User Access:	login, logout, passwd, exit
Help:	man, help
Directory:	mkdir, rmdir, cd, pwd, ls, mv
Editor:	vi, gedit, ed, sed
File Handling / Text Processing:	cp, mv, rm, sort, cat, pg, lp, pr, file, find, more, cmp, diff, comm, head, tail, cut, grep, touch, tr, uniq
Security and Protection:	chmod, chown, chgrp, newgrp
Information:	learn, man, who, date, cal, tty, calendar, time, bc, whoami, which, hostname, history, wc
System Administrator:	su or root, date, fsck, init 2, wall, shut down, mkfs, mount, unmount, dump, restor, tar, adduser, rmuser
Terminal:	echo, printf, clear
Process:	ps, kill, exec
I/O Redirection (<, >, >>), Pipe (), *, gcc	

Answer:

User Access

Command – login

Description: The login program is used to establish a new session with the system. It is normally invoked automatically by responding to the "login:" prompt on the user's terminal.

Options:

- f: Do not perform authentication; user is pre-authenticated. In that case, username is mandatory.
- h: Name of the remote host for this login.
- p: Preserve environment.
- r: Perform autologin protocol for rlogin.

Output:

```
parth@parth:~$ sudo login parth
Password:
Welcome to Ubuntu 22.04.1 LTS (GNU/Linux 5.15.0-47-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

0 updates can be applied immediately.

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.
```

Command – logout

Description: logout command allows you to programmatically logout from your session. causes the session manager to take the requested action immediately.

Output:

```
parth@parth:~$ logout
```

Command – passwd

Description: passwd command in Linux is used to change the user account passwords. The root user reserves the privilege to change the password for any user on the system, while a normal user can only change the account password for his or her own account.

Options:

- d, --delete: This option deletes the user password and makes the account password-less.
- e, --expire: This option immediately expires the account password and forces the user to change password on their next login.
- h, --help: Display help related to the passwd command.

-i, -inactive INACTIVE_DAYS: This option is followed by an integer, INACTIVE_DAYS, which is the number of days after the password expires that the account will be deactivated.

Output:

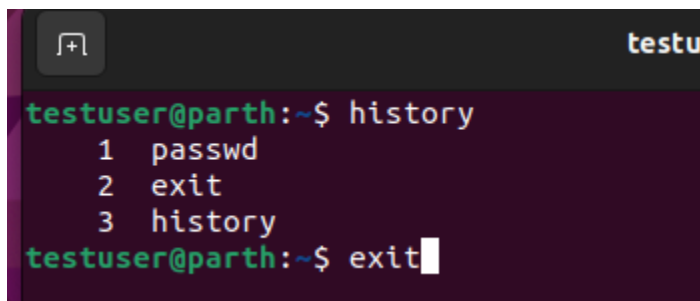
```
testuser@parth:~$ passwd
Changing password for testuser.
Current password:
New password:
Retype new password:
passwd: password updated successfully
testuser@parth:~$
```

Command – exit

Description: exit command in linux is used to exit the shell where it is currently running. It takes one more parameter as [N] and exits the shell with a return of status N. If n is not provided, then it simply returns the status of last command that is executed.

Output:

Before Exit

A terminal window titled 'testu' with a dark background. It shows the command history for 'testuser@parth:~'. The history list contains three entries: '1 passwd', '2 exit', and '3 history'. Below the history, the command 'exit' is entered at the prompt, followed by a cursor.

```
testuser@parth:~$ history
1  passwd
2  exit
3  history
testuser@parth:~$ exit
```

After entering the command it closes the terminal

Help**Command – Man**

Description: The man command is a built-in manual for using Linux commands. It allows users to view the reference manuals of a command or utility run in the terminal.

It provides a detailed view of the command which includes NAME, SYNOPSIS, DESCRIPTION, OPTIONS, EXIT STATUS, RETURN VALUES, ERRORS, FILES, and VERSIONS.

Options:

1. **man [Command Name]** - It displays the whole manual of the command.
2. **-f option:** One may not be able to remember the sections in which a command is present. So this option gives the section in which the given command is present.
3. **-a option:** This option helps us to display all the available intro manual pages in succession.
4. **-k option:** This option searches the given command as a regular expression in all the manuals and it returns the manual pages with the section number in which it is found.
5. **-w option:** This option returns the location in which the manual page of a given command is present.

Output:

```
[root@localhost ~]# man man
MAN(1)                                Manual pager utils                                MAN(1)

NAME
    man - an interface to the system reference manuals

SYNOPSIS
    man [man options] [[section] page ...] ...
    man -k [apropos options] regexp ...
    man -K [man options] [section] term ...
    man -f [whatis options] page ...
    man -l [man options] file ...
    man -w|-W [man options] page ...

DESCRIPTION
    man is the system's manual pager. Each page argument given to man is
    normally the name of a program, utility or function. The manual page
    associated with each of these arguments is then found and displayed. A
    section, if provided, will direct man to look only in that section of
    the manual. The default action is to search in all of the available
    sections following a pre-defined order (see DEFAULTS), and to show only
    the first page found, even if page exists in several sections.

    The table below shows the section numbers of the manual followed by the
    types of pages they contain.

    1 Executable programs or shell commands
    2 System calls (functions provided by the kernel)
    3 Library calls (functions within program libraries)
    4 Special files (usually found in /dev)
```

Command – help

Description: help command displays the information about the built-in commands present in the Linux shell.

Options:

1. **-m Option :** This option displays the output in pseudo-manpage format.

2. **-d Option** : This option is used in cases when we just need to get an overview of any command built in the shell which means this option just gives a brief description of a command like what it does without providing any details of the options.
3. **-s Option** : *This option displays only the syntax of a particular command.*

Output:

```
[root@localhost ~]# help help
help: help [-dms] [pattern ...]
    Display information about builtin commands.

    Displays brief summaries of builtin commands.  If PATTERN is
    specified, gives detailed help on all commands matching PATTERN,
    otherwise the list of help topics is printed.

    Options:
      -d      output short description for each topic
      -m      display usage in pseudo-manpage format
      -s      output only a short usage synopsis for each topic matching
              PATTERN

    Arguments:
      PATTERN  Pattern specifying a help topic

    Exit Status:
      Returns success unless PATTERN is not found or an invalid option is given.
[root@localhost ~]#
```

Directory

Command – mkdir

Description : mkdir command in Linux allows the user to create directories.

This command can create multiple directories at once as well as set the permissions for the directories.

Options:

-v : It displays a message for every directory created.

-p: A flag which enables the command to create parent directories as necessary. If the directories exist, no error is specified.

Output:

```
[root@localhost ~]# mkdir -v Hi Hello
mkdir: created directory 'Hi'
mkdir: created directory 'Hello'
[root@localhost ~]#
```

Command – rmdir

Description : rmdir command is used remove empty directories from the filesystem in Linux. The rmdir command removes each and every directory specified in the command line only if these directories are empty.

Options:

- v: This option displays verbose information for every directory being processed.
- p: In this option each of the directory argument is treated as a pathname of which all components will be removed, if they are already empty, starting from the last component.

Output:

```
[root@localhost ~]# ls
bench.py  Hello  hello.c  Hi
[root@localhost ~]# rmdir Hello Hi
[root@localhost ~]# ls
bench.py  hello.c
[root@localhost ~]#
```

Command – cd

Description: cd command in linux known as change directory command. It is used to change current working directory.

Options:

- .. : this command is used to move to the parent directory of current directory, or the directory one level up from the current directory. “..” represents parent directory.

“dir name”: This command is used to navigate to a directory with white spaces. Instead of using double quotes we can use single quotes then also this command will work.

Output:

```
[root@localhost /]# mkdir Directory
[root@localhost /]# cd Directory
[root@localhost Directory]# cd ..
[root@localhost /]#
```

Command – pwd

Description: **pwd** stands for **Print Working Directory**. It prints the path of the working directory, starting from the root.

Output:

```
[root@localhost /]# cd Directory
[root@localhost Directory]# pwd
/Directory
[root@localhost Directory]#
```

Command – ls

Description: **ls** is a Linux shell command that lists directory contents of files and directories. Some practical examples of **ls** command are shown below.

Options:

ls -l : To show long listing information about the file/directory.

ls -lt : To sort the file names displayed in the order of last modification time. You will be finding it handy to use it in combination with **-l** option

Output:

```
[root@localhost /]# ls
bin    Directory  etc      lib64    mnt      root    srv      usr
boot  Directory1 home    lost+found  opt      run      sys      var
dev    Directory2 lib      media     proc     sbin    tmp
[root@localhost /]#
```

Command – mv

Description: **mv** stands for **move**. **mv** is used to move one or more files or directories from one place to another in a file system like UNIX. It has two distinct functions:

- (i) It renames a file or folder.
- (ii) It moves a group of files to a different directory.

Options:

-n (no-clobber): With **-n** option, **mv** prevent an existing file from being overwritten.

–version: This option is used to display the version of **mv** which is currently running on your system.

Output:

```
[root@localhost Directory]# cat > source.txt
This is Source
[root@localhost Directory]# cat > destination.txt
This is Destination
[root@localhost Directory]# mv source.txt destination.txt
[root@localhost Directory]# cat destination.txt
This is Source
[root@localhost Directory]#
```

Editor**Command: vi**

Description: Editing files using the screen-oriented text editor **vi** is one of the best ways.

An improved version of the **vi** editor which is called the **VIM** has also been made available now. Here, **VIM** stands for **Vi IMproved**.

Options:

-R filename - Opens an existing file in the read-only mode.

Output:

```
[root@localhost practical2]# cat hello.c
/* This C source can be compiled with:
   gcc -o hello hello.c
 */
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char **argv)
{
    printf("Hello World\n");
    return 0;
}
```

Command: gedit

Description: gedit is the official text editor of the GNOME desktop environment.

gedit features a flexible plugin system which can be used to dynamically add new advanced features to gedit itself.

Options:

--encoding : Set the character encoding to be used for opening the files listed on the command line.

--new-document : Create a new document in an existing instance of gedit.

-s, --standalone :Run gedit in standalone mode.

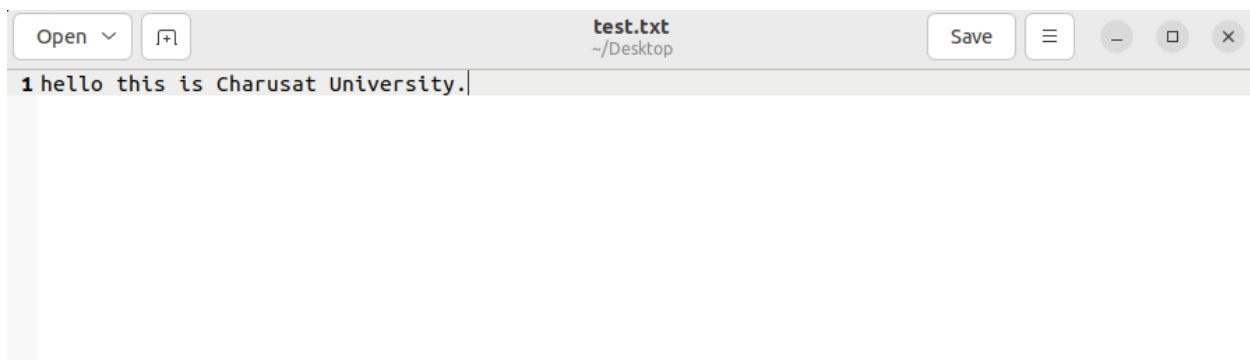
-w, --wait : Open files and block the gedit process.

Output:

Creating File

```
parth@parth:~/Desktop$ cat > test.txt
hello this is Charusat University.
^C
parth@parth:~/Desktop$ cat test.txt
hello this is Charusat University.
parth@parth:~/Desktop$ gedit test.txt
```

Opening it with gedit

**Command: ed**

Description: ed command in Linux is used for launching the ed text editor which is a line-based text editor with a minimal interface which makes it less complex for working on text files i.e creating, editing, displaying and manipulating files.

Options:

-G :- Forces backwards compatibility. Affects the commands 'G', 'V', 'F', 'I', 'm', 't', and '!!'.

-s :- Suppresses diagnostics. This should be used if ed's standard input is from a script.

p string :- Specifies a command prompt. This may be toggled on and off with the 'P' command.

Output:

```
[root@localhost ~]# cd ..  
[root@localhost /]# cd Directory  
[root@localhost Directory]# cat > Hello.txt  
Hello World!  
[root@localhost Directory]# ed Hello.txt  
13
```

Command: sed

Description: The SED command in Linux stands for Stream EDiTOr and is helpful for a myriad of frequently needed operations in text files and streams. Sed helps in operations like selecting the text, substituting text, modifying an original file, adding lines to text, or deleting lines from the text.

Options:

-n :- Use option along with the /p print flag to display only the replaced lines

Output:

```
[root@localhost practical2]# sed 's/Hello/Hii/' test.txt  
Hii
```

File Handling / Text Processing:**Command:** cp

Description: cp stands for copy. This command is used to copy files or group of files or directory. It creates an exact image of a file on a disk with different file name. cp command require at least two filenames in its arguments.

Options:

-i(interactive): With this option system first warns the user before overwriting the destination file. cp prompts for a response, if you press y then it overwrites the file and with any other option leave it uncopied.

-b(backup): With this option cp command creates the backup of the destination file in the same folder with the different name and in different format.

-f(force): If the system is unable to open destination file for writing operation because the user doesn't have writing permission for this file then by using -f option with cp command, destination file is deleted first and then copying of content is done from source to destination file.

-r or -R: Copying directory structure. With this option cp command shows its recursive behavior by copying the entire directory structure recursively.

Output:

```
[root@localhost Directory]# cp Hello.txt Copy.txt
[root@localhost Directory]# cat Copy.txt
Hello World!
[root@localhost Directory]# cat Hello.txt
Hello World!
[root@localhost Directory]#
```

Command – mv

Description: **mv** stands for **move**. mv is used to move one or more files or directories from one place to another in a file system like UNIX. It has two distinct functions:

- (i) It renames a file or folder.
- (ii) It moves a group of files to a different directory.

Options:

-n (no-clobber): With -n option, **mv** prevent an existing file from being overwritten.

-version: This option is used to display the version of **mv** which is currently running on your system.

Output:

```
[root@localhost Directory]# cat > source.txt
This is Source
[root@localhost Directory]# cat > destination.txt
This is Destination
[root@localhost Directory]# mv source.txt destination.txt
[root@localhost Directory]# cat destination.txt
This is Source
[root@localhost Directory]#
```

Command: rm

Description: rm stands for remove here. rm command is used to remove objects such as files, directories, symbolic links and so on from the file system like UNIX. To be more precise, rm removes references to objects from the filesystem, where those objects might have had multiple references (for example, a file with two different names). By default, it does not remove directories.

Options:

-i (Interactive Deletion): Like in cp, the -i option makes the command ask the user for confirmation before removing each file, you have to press y for confirm deletion, any other key leaves the file un-deleted.

-f (Force Deletion): rm prompts for confirmation removal if a file is write protected

-r (Recursive Deletion): With -r(or -R) option rm command performs a tree-walk and will delete all the files and sub-directories recursively of the parent directory. At each stage it deletes everything it finds.

-version: This option is used to display the version of rm which is currently running on your system.

Output:

```
[root@localhost Directory]# ls
2.txt Copy.txt destination.txt Hello.txt
[root@localhost Directory]# rm 2.txt
[root@localhost Directory]# ls
Copy.txt destination.txt Hello.txt
[root@localhost Directory]#
```

Command: sort

Description: SORT command is used to sort a file, arranging the records in a particular order. By default, the sort command sorts file assuming the contents are ASCII.

Options:

-o Option: Unix also provides us with special facilities like if you want to write the output to a new file, output.txt, redirects the output like this or you can also use the built-in sort option -o, which allows you to specify an output file.

-r Option: Sorting In Reverse Order: You can perform a reverse-order sort using the -r flag. the -r flag is an option of the sort command which sorts the input file in reverse order i.e. descending order by default.

Output:

```
[root@localhost Directory]# cat > Sort.txt
Zebra
Monkey
Horse
Dog
[root@localhost Directory]# sort Sort.txt
Dog
Horse
Monkey
Zebra
```

Command: Cat**Description:**

Cat(concatenate) command is very frequently used in Linux. It reads data from the file and gives their content as output. It helps us to create, view, concatenate files. So let us see some frequently used cat commands.

Options:

- E : cat command can highlight the end of line.
- s : Cat command can suppress repeated empty lines in output.
- n : To view contents of a file preceding with line numbers.

Output:

```
[root@localhost Directory]# cat Sort.txt
Zebra
Monkey
Horse
Dog
[root@localhost Directory]#
```

Command: pg

Description: The pg command displays the contents of text files, one page at a time.

If input comes from a pipe, pg stores the data in a buffer file while reading to make navigation possible.

Options:

- number: The number of lines per page. Usually, this is the number of CRT lines minus one.

-c: Clear the screen before a page is displayed, if the terminfo entry for the terminal provides this capability.

-e: Do not pause and display "(EOF)" at the end of a file.

-f: Do not split long lines.

-n: Without this option, commands must be terminated by a newline character. With this option, pg advances once a command letter is entered.

Command: lp

Description: On Unix-like operating systems, the lp command prints files.

lp submits files for printing, or alters a pending print job. Use a file name of "-" to specify printing from the standard input.

Options:

-E: Forces encryption when connecting to the server.

-U: username Specifies the username to use when connecting to the server.

-c: This option is provided for backward compatibility only. On systems that support it, this option forces the print file to be copied to the spool directory before printing. In CUPS, print files are always sent to the scheduler via IPP which has the same effect.

-d (destination): Prints files to the destination printer.

-h hostname[:port] Chooses an alternate server.

Command: pr

Description: pr command is used to prepare a file for printing by adding suitable footers, headers, and the formatted text. pr command actually adds 5 lines of margin both at the top and bottom of the page. The header part shows the date and time of the last modification of the file with the file name and the page number.

Options:

-n: provide number lines which helps in debugging the code -n option is used.

-d: reduces clutter -d option is used.

Output:

```
[root@localhost Directory]# pr -3 Sort.txt

2022-08-03 18:29                               Sort.txt                               Page 1

Zebra                                           Horse                                           Dog
Monkey
```

Command: File

Description: file command is used to determine the type of a file. .file type may be of human-readable(e.g. 'ASCII text') or MIME type(e.g. 'text/plain; charset=us-ascii'). This command tests each argument in an attempt to categorize it.

Options:

-b, --brief : This is used to display just file type in brief mode

* option : Command displays the all files's file type.

directoryname/* option : This is used to display all files filetypes in particular directory.

[range]* option: To display the file type of files in specific range.

Output:

```
parth@parth:~/Desktop$ file test.txt
test.txt: ASCII text
```

Command: find

Description: The find command in UNIX is a command line utility for walking a file hierarchy. It can be used to find files and directories and perform subsequent operations on them. It supports searching by file, folder, name, creation date, modification date, owner and permissions.

Options:

-exec CMD: The file being searched which meets the above criteria and returns 0 for as its exit status for successful command execution.

-ok CMD : It works same as -exec except the user is prompted first.

-inum N : Search for files with inode number 'N'.

-links N : Search for files with 'N' links.

-name demo : Search for files that are specified by 'demo'.

- newer file : Search for files that were modified/created after 'file'.
- perm octal : Search for the file if permission is 'octal'.
- print : Display the path name of the files found by using the rest of the criteria.
- empty : Search for empty files and directories.

Output:

```
[root@localhost practical-2]# find ./sample-find-dir found.text
./sample-find-dir
./sample-find-dir/found.text
```

Command – more

Description: more command is used to view the text files in the command prompt, displaying one screen at a time in case the file is large (For example log files). The more command also allows the user to scroll up and down through the page.

Options:

- f : This option does not wrap the long lines and displays them as such
- p : This option clears the screen and then displays the text

Output:

```
Hello this is a sample file.
[root@localhost /]#
```

Command – cmp

Description: cmp command in Linux/UNIX is used to compare the two files byte by byte and helps you to find out whether the two files are identical or not.

Options:

- b(print-bytes) : If you want cmp displays the differing bytes in the output when used with -b option
- i [bytes-to-be-skipped] : Now, this option when used with cmp command helps to skip a particular number of initial bytes from both the files and then after skipping it compares the files.
- l option : This option makes the cmp command print byte position and byte value for all differing bytes.

Output:

```
Hello this is a sample file.  
[root@localhost /]# cat > Sample2.txt  
This is another sample file.  
[root@localhost /]# cmp Sample.txt Sample2.txt  
Sample.txt Sample2.txt differ: byte 1, line 1  
[root@localhost /]#
```

Command – diff

Description: diff stands for difference. This command is used to display the differences in the files by comparing the files line by line.

Output:

```
[root@localhost /]# diff Sample.txt Sample2.txt  
1c1  
< Hello this is a sample file.  
---  
> This is another sample file.  
[root@localhost /]#
```

Command – comm

Description: comm compare two sorted files line by line and write to standard output; the lines that are common and the lines that are unique.

Options:

- 1 :suppress first column(lines unique to first file)
- 2 :suppress second column(lines unique to second file).
- 3 :suppress third column(lines common to both files).

Output:

```
[root@localhost /]# comm Sample.txt Sample2.txt  
Hello this is a sample file.  
      This is another sample file.  
[root@localhost /]#
```

Command – head

Description: The head command, as the name implies, print the top N number of data of the given input. By default, it prints the first 10 lines of the specified files.

Output:

```
[root@localhost ~]# head -n 5 Long.txt
a
b
c
d
e
[root@localhost ~]#
```

Command – tail

Description: The tail command, as the name implies, print the last N number of data of the given input. By default, it prints the last 10 lines of the specified files.

Output:

```
[root@localhost ~]# tail -n 5 Long.txt
l
m
n
o
p
[root@localhost ~]#
```

Command – cut

Description: The cut command in UNIX is a command for cutting out the sections from each line of files and writing the result to standard output.

Output:

```
[root@localhost /]# cat > Fruits.txt
Mango
Apple
Banana
Watermelon
Strawberry
[root@localhost /]# cut -b 1,2,3 Fruits.txt
Man
App
Ban
Wat
Str
[root@localhost /]#
```

Command – grep

Description: Grep is an acronym that stands for Global Regular Expression Print.

Grep is a Linux / Unix command-line tool used to search for a string of characters in a specified file. The text search pattern is called a regular expression. When it finds a match, it prints the line with the result.

Options:

- w: To search for the word phoenix in all files in the current directory.
- i: Ignore case
- r : search for subdirectory

Output:

```
[root@localhost practical-2]# grep abc states.txt
abc
```

Command – touch

Description: The **touch** command is a standard command used in UNIX/Linux operating system which is used to create, change and modify timestamps of a file.

Output:

```
boot  etc      lib    lost+found  opt  run      sbin    tmp
dev   Fruits.txt lib64  media      proc Sample2.txt srv      usr
[root@localhost ~]# ll
total 92
lrwxrwxrwx  1 root root    7 Aug 12  2020 bin -> usr/bin
drwxrwxr-x  2 root root   37 Dec  8  2020 boot
drwxr-xr-x  3 root root 12180 Jan  1  1970 dev
drwxr-xr-x  2 root root   37 Aug  3  21:53 Directory
drwxr-xr-x 125 root root  6673 Dec 26  2020 etc
-rw-r--r--  1 root root   41 Aug  3  22:38 Fruits.txt
drwxr-xr-x  2 root root   37 Aug 12  2020 home
lrwxrwxrwx  1 root root    7 Aug 12  2020 lib -> usr/lib
lrwxrwxrwx  1 root root    9 Aug 12  2020 lib64 -> usr/lib64
-rw-r--r--  1 root root   38 Aug  3  22:33 Long.txt
drwx----- 2 root root   37 Dec  8  2020 lost+found
drwxr-xr-x  2 root root   37 Aug 12  2020 media
drwxr-xr-x  2 root root   37 Aug 12  2020 mnt
drwxr-xr-x  2 root root   37 Aug 12  2020 opt
dr-xr-xr-x 29 root root    0 Jan  1  1970 proc
dr-xr-xr-x  2 root root  243 Dec 26  2020 root
drwxr-xr-x 26 root root  716 Dec 26  2020 run
-rw-r--r--  1 root root   29 Aug  3  22:26 Sample2.txt
-rw-r--r--  1 root root   29 Aug  3  22:42 Sample.txt
lrwxrwxrwx  1 root root    8 Aug 12  2020 sbin -> usr/sbin
drwxr-xr-x  2 root root   37 Aug 12  2020 srv
dr-xr-xr-x 11 root root    0 Jan  1  1970 sys
drwxrwxrwt  3 root root   188 Dec 26  2020 tmp
drwxr-xr-x 12 root root   274 Dec  8  2020 usr
drwxr-xr-x 19 root root   457 Dec  8  2020 var
```

Command – tr

Description: The **tr** command in UNIX is a command line utility for translating or deleting characters. It supports a range of transformations including uppercase to lowercase, squeezing repeating characters, deleting specific characters and basic find and replace.

Options:

-d: To delete specific characters.

-c: To complement the sets

Output:

```
[root@localhost practical-2]# cat > file1.txt
hello
^C
[root@localhost practical-2]# cat file1.txt | tr "[a-z]" "[A-Z]"
HELLO
```

Command- uniq

Description: The uniq command in Linux is a command-line utility that reports or filters out the repeated lines in a file.

In simple words, uniq is the tool that helps to detect the adjacent duplicate lines and also deletes the duplicate lines.

Options:

-c – -count : It tells how many times a line was repeated by displaying a number as a prefix with the line.

-d – -repeated : It only prints the repeated lines and not the lines which aren't repeated.

-u – -unique : It allows you to print only unique lines.

Output:

```
parth@parth:~/Desktop$ cat test.txt
hello this is Charusat University.
It is a wonderful day.
Hi hi.
hello
hel
hello
hello
hello
parth@parth:~/Desktop$ uniq test.txt
hello this is Charusat University.
It is a wonderful day.
Hi hi.
hello
hel
hello
parth@parth:~/Desktop$
```


Security and Protection:

Command – chmod

Description: The “chmod” command is used to change the access mode of a file. The name is an abbreviation of change mode.

Output:

```
[root@localhost Directory1]# ls -l hello.txt
-rw-r--r-- 1 root root 0 Aug 24 20:02 hello.txt
[root@localhost Directory1]# chmod u=x hello.txt
[root@localhost Directory1]# chmod -x hello.txt
[root@localhost Directory1]# ls -l hello.txt
----r--r-- 1 root root 0 Aug 24 20:02 hello.txt
[root@localhost Directory1]#
```

Command – chown

Description: It is used to change the ownership and group of files, directories and links. By default, the owner of a file system object is the user that created it.

Output:

```
[root@localhost Directory1]# sudo chown :root hello.txt
[root@localhost Directory1]# ls -l hello.txt
----r--r-- 1 root root 0 Aug 24 20:02 hello.txt
[root@localhost Directory1]#
```

Command – chgrp

Description: It is used to change the Group of File or Directory. All Files in Linux basically belongs to an owner and a group.

Output:

```
[root@localhost Directory1]# sudo groupadd ListDirectory
[root@localhost Directory1]# sudo chgrp hello.txt
chgrp: missing operand after 'hello.txt'
Try 'chgrp --help' for more information.
[root@localhost Directory1]# sudo chgrp -c hello.txt
chgrp: missing operand after 'hello.txt'
Try 'chgrp --help' for more information.
[root@localhost Directory1]# sudo chgrp -c ListDirectory hello.txt
changed group of 'hello.txt' from root to ListDirectory
[root@localhost Directory1]# ls -l hello.txt
----r--r-- 1 root ListDirectory 0 Aug 24 20:02 hello.txt
[root@localhost Directory1]#
```

Command - newgrp

Description: The newgrp command changes a user's real group identification. When you run the command, the system places you in a new shell and changes the name of your real group to the group specified with the Group parameter. By default, the newgrp command changes your real group to the group specified in the /etc/passwd file.

Options:

- : Changes the environment to the login environment of the new group.
- l: Indicates the same value as the - flag.

Output:

```
parth@parth:~/Desktop$ newgrp parth
```

Information**Command – Man**

Description: The man command is a built-in manual for using Linux commands. It allows users to view the reference manuals of a command or utility run in the terminal.

It provides a detailed view of the command which includes NAME, SYNOPSIS, DESCRIPTION, OPTIONS, EXIT STATUS, RETURN VALUES, ERRORS, FILES, and VERSIONS.

Options:

1. **man [Command Name]** - It displays the whole manual of the command.
2. **-f option:** One may not be able to remember the sections in which a command is present. So this option gives the section in which the given command is present.
3. **-a option:** This option helps us to display all the available intro manual pages in succession.
4. **-k option:** This option searches the given command as a regular expression in all the manuals and it returns the manual pages with the section number in which it is found.
5. **-w option:** This option returns the location in which the manual page of a given command is present.

Output:

```
[root@localhost ~]# man man
MAN(1)                                Manual pager utils                                MAN(1)

NAME
    man - an interface to the system reference manuals

SYNOPSIS
    man [man options] [[section] page ...] ...
    man -k [apropos options] regexp ...
    man -K [man options] [section] term ...
    man -f [whatis options] page ...
    man -l [man options] file ...
    man -w|-W [man options] page ...

DESCRIPTION
    man is the system's manual pager. Each page argument given to man is
    normally the name of a program, utility or function. The manual page
    associated with each of these arguments is then found and displayed. A
    section, if provided, will direct man to look only in that section of
    the manual. The default action is to search in all of the available
    sections following a pre-defined order (see DEFAULTS), and to show only
    the first page found, even if page exists in several sections.

    The table below shows the section numbers of the manual followed by the
    types of pages they contain.

    1 Executable programs or shell commands
    2 System calls (functions provided by the kernel)
    3 Library calls (functions within program libraries)
    4 Special files (usually found in /dev)
```

Command – who

Description: The “who” command shows the information of all the users who all are logged into the system.

Output:

```
parth@parth:~$ who
parth    tty2          2022-09-03 15:21 (tty2)
parth@parth:~$
```

Command – date

Description: **date** command is used to display the system date and time. date command is also used to set date and time of the system. By default the date command displays the date in the time zone on which unix/linux operating system is configured.

Options:

date (no option) : With no options, the date command displays the current date and time, including the abbreviated day name, abbreviated month name, day of the month, the time separated by colons, the time zone name, and the year.

-u Option: Displays time in GMT(Greenwich Mean Time)/UTC(Coordinated Universal Time)time zone.

–date option : For displaying future date

Output:

```
[root@localhost ~]# date
Thu Jul 28 01:16:44 PM UTC 2022
[root@localhost ~]# date -u
Thu Jul 28 07:46:49 AM UTC 2022
[root@localhost ~]# date --date="next tue"
Tue Aug  2 12:00:00 AM UTC 2022
[root@localhost ~]#
```

Command – cal

Description: Shows current month calendar on the terminal.

Output:

```
[root@localhost Directory1]# cal
      August 2022
Su Mo Tu We Th Fr Sa
      1  2  3  4  5  6
 7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30 31
```

Command – tty

Description: The “tty” command displays the system information on the terminal.

Output:

```
[root@localhost ~]# tty
/dev/hvc0
[root@localhost ~]#
```

Command – calender

Description: The “calendar” command gives the occasions of the current and the following days.

Output:

```
└─$ calendar
Aug 07 Jack the Ripper makes his first kill, 1888
Aug 07 Battle of Boyaca in Colombia
Aug 07* Parshat Re'eh
Aug 07 Jonathan Mini <mini@FreeBSD.org> born in San Mateo
, California, United States, 1979
Aug 07 Aujourd'hui, c'est la St(e) Gaëtan.
Aug 07 Gründung der Sozialdemokratischen Arbeiterpartei i
n Eisenach
      unter der Führung von August Bebel und Wilhelm Lie
bknecht, 1869
Aug 07 Ibolya
Aug 07* Көлік және байланыс қызметкерлері күні
Aug 07* Grandparents Day
Aug 08 Dustin Hoffman born in Los Angeles, 1937
Aug 08 Atomic bomb dropped on Nagasaki, 1945
Aug 08 Montenegro declares war on Germany, 1914
Aug 08 Richard Nixon resigns the US presidency, 1974
Aug 08 The Great Train Robbery -- $7,368,000, 1963
Aug 08 Fire in the mine 'Le bois du Casier', 263 deaths,
1956
Aug 08 Belgian parliament in exile comes back to Bruxelle
s, 1944
Aug 08 Mikolaj Golub <trociny@FreeBSD.org> born in Kharko
v, USSR, 1977
Aug 08 Juergen Lock <nox@FreeBSD.org> died in Bremen, Ger
many, 2015
Aug 08 N'oubliez pas les Dominique !
Aug 08 Bonne fête aux Cyriaque !
Aug 08 Friedensfest (Augsburg)
Aug 08 Atombombenabwurf auf Nagasaki, 1945
Aug 08 László
```

Command – time

Description: The “Time” command gives the execution time for each command whenever entered to the terminal input.

Output:

```
[root@localhost ~]# time  
user      0m0.53s  
sys       0m1.26s  
[root@localhost ~]#
```

Command – bc

Description: bc command is used for command line calculator. It is similar to basic calculator by using which we can do basic mathematical calculations.

Options:

- h, { - -help } : Print the usage and exit
- i, { - -interactive } : Force interactive mode
- l, { - -mathlib } : Define the standard math library
- w, { - -warn } : Give warnings for extensions to POSIX bc
- s, { - -standard } : Process exactly the POSIX bc language
- q, { - -quiet } : Do not print the normal GNU bc welcome
- v, { - -version } : Print the version number and copyright and quit

Output:

```
[root@localhost ~]# echo "var=10;var" | bc  
10  
[root@localhost ~]# echo "var=10;var^=2;var" | bc  
100  
[root@localhost ~]# echo "var=10;++var" | bc  
11  
[root@localhost ~]#
```

Command – whoami

Description: The “whoami” command returns the username of the current user.

Output:

```
parth@parth:~$ whoami  
parth
```

Command – which

Description: “Which” command in Linux is a command which is used to locate the executable file associated with the given command by searching it in the path environment variable.

Output:

```
[root@localhost ~]# which man
/bin/man
[root@localhost ~]#
```

Command – hostname

Description: “hostname” command returns the host name on which the operating system is working.

Output:

```
[root@localhost ~]# hostname
localhost
[root@localhost ~]#
```

Command - History**Description:**

history command is used to view the previously executed command. This feature was not available in the Bourne shell. Bash and Korn support this feature in which every command executed is treated as the event and is associated with an event number using which they can be recalled and changed if required.

Options:

!! – most recent ones

-c – to clear history

| tail – to show last 10 commands

Output:

```
parth@parth:~$ history
 1 sudo apt update
 2 sudo apt install build-essential
 3 ls
 4 ls -a
 5 pwd
 6 cd ..
 7 ls
 8 cd parth/
 9 ls
10 exit
11 ls
12 cd Desktop/
13 ls
14 vim test.txt
15 vi test.txt
16 ls
17 sudo apt install vim
18 sudo apt get update
19 sudo apt get-update
20 sudo apt update
21 sudo apt upgrade
22 sudo apt install vim
23 cat > test.txt
24 cat test.txt
25 gedit test.txt
26 exit
27 cd Desktop/
28 vi test.txt
```

Command – wc

Description: wc stands for word count. As the name implies, it is mainly used for counting purpose.

It is used to find out number of lines, word count, byte and characters count in the files specified in the file arguments.

By default it displays four-columnar output.

First column shows number of lines present in a file specified, second column shows number of words present in the file, third column shows number of characters present in file and fourth column itself is the file name which are given as argument.

Options:

1. -l: This option prints the **number of lines** present in a file.

2. 2. -w: This option prints the **number of words** present in a file. With this option the command displays two-columnar output, 1st column shows number of words present in a file and 2nd is the file name.
3. -c: This option displays **count of bytes** present in a file.

Output:

```
[root@localhost ~]# wc 1.txt
5  5 33 1.txt
[root@localhost ~]# wc 2.txt
3  3 26 2.txt
[root@localhost ~]#
```

System Administrator:

Command – su or root

Description: The su (short for substitute or switch user) utility allows you to run commands with another user's privileges, by default the root user.

Options:

-c command_name – to switch user and run command in interactive mode

Su - - allows to switch in an environment very similar to login

Output:

```
parth@parth:~$ su --help

Usage:
su [options] [-] [<user> [<argument>...]]

Change the effective user ID and group ID to that of <user>.
A mere - implies -l. If <user> is not given, root is assumed.

Options:
-m, -p, --preserve-environment    do not reset environment variables
-w, --whitelist-environment <list> don't reset specified variables

-g, --group <group>               specify the primary group
-G, --supp-group <group>          specify a supplemental group

-, -l, --login                    make the shell a login shell
-c, --command <command>          pass a single command to the shell with -c
--session-command <command>     pass a single command to the shell with -c
                                and do not create a new session
-f, --fast                       pass -f to the shell (for csh or tcsh)
-s, --shell <shell>              run <shell> if /etc/shells allows it
-P, --pty                        create a new pseudo-terminal

-h, --help                       display this help
-V, --version                     display version
```

Command – date

Description: Displays Date

Output:

```
$ date  
Sun Aug 7 01:25:10 PM EDT 2022
```

Command – fsck

Description: “Fsck” stands for “File System Consistency check”. The use of “fsck” command is that you can use it to check and repair your filesystem.

Output:

```
[root@localhost ~]# fsck  
fsck from util-linux 2.36
```

Command – Init 2

Description: In simple words the role of init is to create processes from script stored in the file /etc/inittab which is a configuration file which is to be used by initialization system. It is the last step of the kernel boot sequence. /etc/inittab Specifies the init command control file.

Output:

```
Loading...  
Welcome to Fedora 33 (riscv64)  
[root@localhost ~]# init 2  
mount: /proc: none already mounted on /proc.  
mount: /sys: none already mounted on /proc.  
mkdir: cannot create directory '/dev/pts': File exists  
RTNETLINK answers: File exists  
Welcome to Fedora 33 (riscv64)  
[root@localhost ~]#
```

Command – wall

Description: wall command in Linux system is used to write a message to all users. This command displays a message, or the contents of a file, or otherwise its standard input, on the terminals of all currently logged in users. The lines which will be longer than 79 characters, wrapped by this command. Short lines are whitespace padded to have 79 characters.

Options:

1. wall -n: This option will suppress the banner.
2. wall -t: This option will abandon the write attempt to the terminals after timeout seconds. This timeout needs to be a positive integer. The by default value is 300 seconds, which is a legacy from the time when peoples ran terminals over modem lines. -c: This option displays **count of bytes** present in a file.
3. wall -V : This option display version information and exit.

Output:

```
[root@localhost ~]# wall -V
wall from util-linux 2.36
[root@localhost ~]# wall -h

Usage:
  wall [options] [<file> | <message>]

Write a message to all users.

Options:
  -g, --group <group>      only send message to group
  -n, --nobanner            do not print banner, works only for root
  -t, --timeout <timeout>  write timeout in seconds

  -h, --help                display this help
  -V, --version             display version

For more details see wall(1).
[root@localhost ~]#
```

Command – shutdown

Description: The “shutdown” command is used to shutdown the device directly from the terminal.

Output:

```
$ shutdown
Shutdown scheduled for Sun 2022-08-07 13:29:26 EDT, use 's
hutdown -c' to cancel.
```

Command – mkfs

Description: The “mkfs” is used to build a Linux file system on a device, usually a hard disk partition.

Output:

```
$ mkfs -t ext2 /dev/bin
mke2fs 1.46.5 (30-Dec-2021)
```

Command – mount

Description: The “mount” command serves to attach or mount the file system found on some device to the main file system of the current device.

Output:

```
[root@localhost ~]# mount
root on / type 9p (rw,relatime,dirsync,mmap,access=client,trans=virtio)
devtmpfs on /dev type devtmpfs (rw,relatime,size=93120k,nr_inodes=23280,mode=755)
none on /proc type proc (rw,relatime)
none on /sys type sysfs (rw,relatime)
devpts on /dev/pts type devpts (rw,relatime,mode=600,ptmxmode=000)
```

Command – unmount

Description: The “unmount” command serves to detach any file system found on some device from the main file system of the current device.

Output:

```
$ unmount
Command 'unmount' not found, did you mean:
  command 'umount' from deb mount
Try: sudo apt install <deb name>
```

Command – dump

Description :

dump command in Linux is used for backup the filesystem to some storage device. It backs up the complete file system and not the individual files. In other words, it backups the required files to tape, disk or any other storage device for safe storage.

Options:

1. **-level #** : The dump level which is an integer ranging from 0-9. If the user has asked to perform a full backup or a backup of only those new files added after the last dump of a lower level.
2. **-f file** : This specifies the file where the backup will be written to. The file can be a tape drive, a floppy disk, ordinary file or standard output.
3. **-u** : This records and updates the backup in /etc/dumpdates file.

Command – restore

Description: “Restore” command in Linux system is used for restoring files from a backup created using dump. The restore command performs the exact inverse function of dump.

Output:

```
(kali㉿kali)-[~/Documents/OS_practicals]
$ restore
restore 0.4b47 (using libext2fs 1.46.5 of 30-Dec-2021)
usage:  restore -C [-cdeHlMuvVy] [-b blocksizes] [-D filesystem] [-E mls]
        [-f file] [-F script] [-L limit] [-s fileno]
        restore -i [-acdehHlMouvVy] [-A file] [-b blocksizes] [-E mls]
        [-f file] [-F script] [-Q file] [-s fileno]
        restore -P file [-acdHlMuvVy] [-b blocksizes]
        [-f file] [-F script] [-s fileno] [-X filelist] [file ...]
        restore -r [-cdeHlMuvVy] [-b blocksizes] [-E mls]
        [-f file] [-F script] [-s fileno] [-T directory]
        restore -R [-cdeHlMuvVy] [-b blocksizes] [-E mls]
        [-f file] [-F script] [-s fileno] [-T directory]
        restore -t [-cdhHlMuvVy0] [-A file] [-b blocksizes]
        [-f file] [-F script] [-Q file] [-s fileno] [-X filelist] [file ...]
        restore -x [-acdehHlMouvVy] [-A file] [-b blocksizes] [-E mls]
        [-f file] [-F script] [-Q file] [-s fileno] [-X filelist] [file ...]
```

Terminal

Command – echo

Description : echo command in linux is used to display line of text/string that are passed as an argument . This is a built in command that is mostly used in shell scripts and batch files to output status text to the screen or a file.

Options:

\b : it removes all the spaces in between the text

\c : suppress trailing new line with backspace interpreter ‘-e’ to continue without emitting new line.

\n : this option creates new line from where it is used

Output:

```
[root@localhost ~]# echo -e "This \nis \na \nnew\nline"
This
is
a
new
line
[root@localhost ~]#
```

Command - printf

Description: “printf” command in Linux is used to display the given string, number or any other format specifier on the terminal window. It works the same way as “printf” works in programming languages like C.

Output:

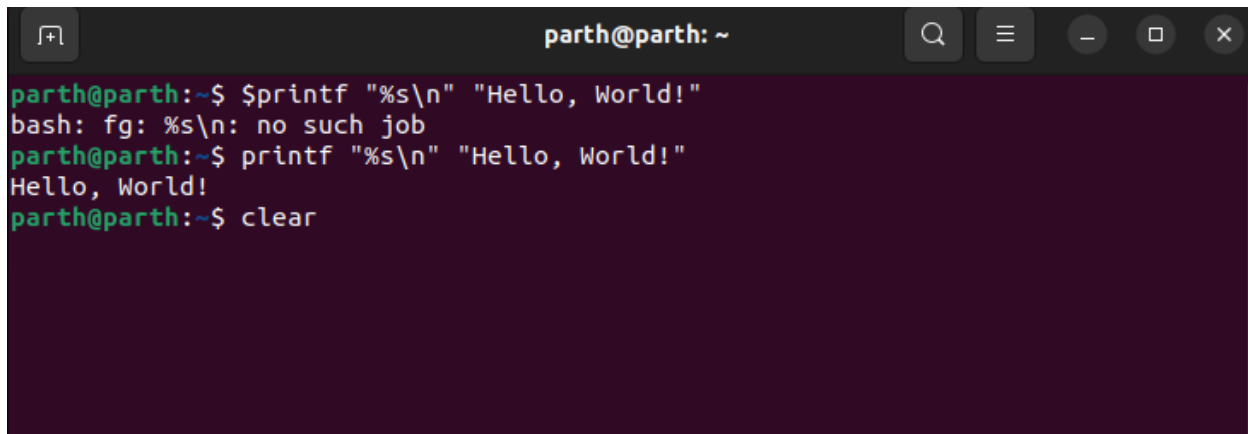
```
parth@parth:~$ printf "%s\n" "Hello, World!"
Hello, World!
parth@parth:~$
```

Command - clear

Description: clear is a standard Unix computer operating system command that is used to clear the terminal screen. This command first looks for a terminal type in the environment and after that, it figures out the terminfo database for how to clear the screen. And this command will ignore any command-line parameters that may be present.

Output:

Before clear



```
parth@parth: ~  
parth@parth:~$ $printf "%s\n" "Hello, World!"  
bash: fg: %s\n: no such job  
parth@parth:~$ printf "%s\n" "Hello, World!"  
Hello, World!  
parth@parth:~$ clear
```

After Clear



```
parth@parth: ~  
parth@parth:~$
```

Process**Command - ps**

Description: The “ps” command in Linux is an abbreviation of “process status”. It is used to get information about the processes running within your system. The output of this command can vary depending upon the parameters used with it.

Options:

- A : To display all running processes.
- T : All processes associated with current terminal.
- u UserName : All processes associated with a particular user.

Screenshot:

```
parth@parth:~$ ps
  PID TTY          TIME CMD
  5179 pts/0    00:00:00 bash
  5188 pts/0    00:00:00 ps
parth@parth:~$
```

Command - kill

Description: kill command in Linux (located in /bin/kill), is a built-in command which is used to terminate processes manually. kill command sends a signal to a process which terminates the process. If the user doesn't specify any signal which is to be sent along with kill command then default TERM signal is sent that terminates the process.

Options:

- l: To display all the available signals
- s: To show how to send signal to processes.

Screenshot:

```
parth@parth:~$ kill -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
parth@parth:~$
```


Command – exec

Description: exec command in Linux is used to execute a command from the bash itself. This command does not create a new process it just replaces the bash with the command to be executed. If the exec command is successful, it does not return to the calling process.

Options:

c: It is used to execute the command with empty environment.

a name: Used to pass a name as the zeroth argument of the command.

l: Used to pass dash as the zeroth argument of the command.

Output:

```
[root@localhost ~]# exec ls
1.txt 2.txt bench.py hello.c
[root@localhost ~]#
```

Conclusion: In this practical I learnt implementing various Unix commands.

I/O Redirection

Command: > (Overwrite)

Description: Commands with a single bracket '>' overwrite existing file content.

Output:

```
[root@localhost ~]# cat > hello.txt
hello this is Charusat University
^C
[root@localhost ~]# cat hello.txt
hello this is Charusat University
[root@localhost ~]#
```

Command: < (Input)

Description: We can overwrite the standard input using the '<' symbol.

Output:

```
root@educative:/# touch file.txt
root@educative:/# echo "hello world"
hello world
root@educative:/# echo "hello world" > file.txt
root@educative:/# cat file.txt
hello world
root@educative:/# wc -l file.txt
1 file.txt
root@educative:/# wc -l < file.txt
1
root@educative:/#
```

Command: >> (Append)

Description: Commands with a double bracket '>>' do not overwrite the existing file content.

Output:

```
[root@localhost ~]# cat >> hello.txt
This line is appended.
^C
[root@localhost ~]# cat hello.txt
hello this is Charusat University
This line is appended.
[root@localhost ~]#
```

Command: | (pipe)

Description: Pipe redirects a stream from one program to another. When pipe is used to send standard output of one program to another program, first program's data will not be displayed on the terminal, only the second program's data will be displayed.

Output:

```
[root@localhost ~]# ls *.txt | cat > txtfileAll
[root@localhost ~]# cat txtfileAll
hello2.txt
hello3.txt
hello.txt
[root@localhost ~]#
```

Practical 3

Aim: 1. Write a script called hello which outputs the following:

- your username • the time and date • who is logged on
- Also output a line of asterisks (*****) after each section.

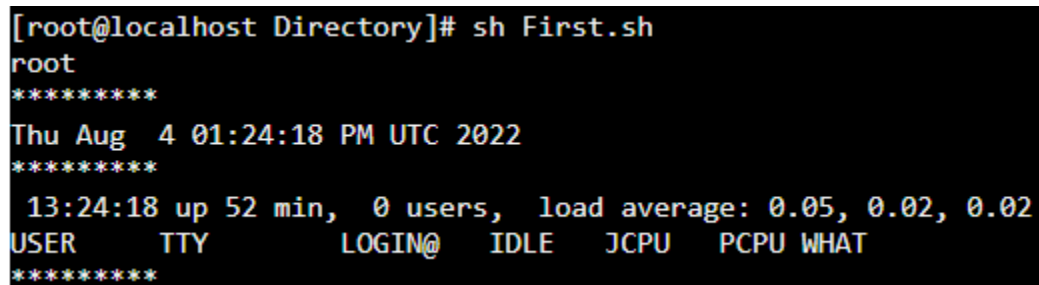
2. Write a shell script which calculates nth Fibonacci number where n will be provided as input when prompted.

3. Write a shell script which takes one number from user and finds factorial of a given number.

Code 1:

```
whoami
echo "*****"
date
echo "*****"
w
echo "*****"
```

Output:

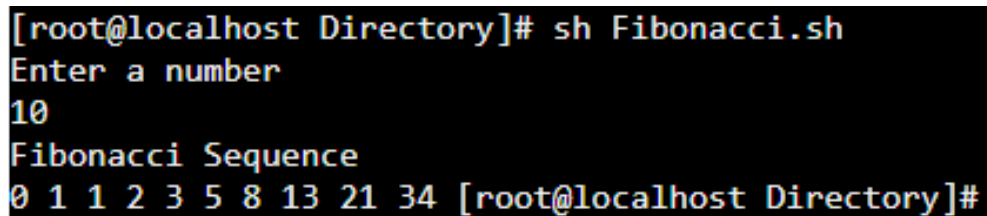


```
[root@localhost Directory]# sh First.sh
root
*****
Thu Aug 4 01:24:18 PM UTC 2022
*****
 13:24:18 up 52 min,  0 users,  load average: 0.05, 0.02, 0.02
USER      TTY      LOGIN@  IDLE   JCPU   PCPU   WHAT
*****
```

Code 2:

```
echo "Enter a number"
read x
a=0
b=1
echo "Fibonacci Sequence"
```

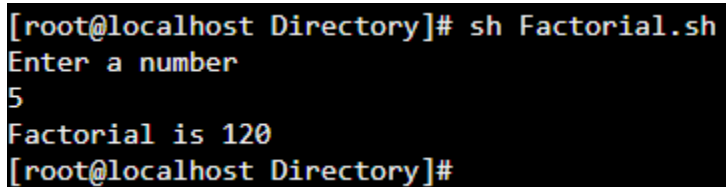
```
for (( i=0; i<x; i++ ))
do
    echo -n "$a "
    fn=$((a + b))
    a=$b
    b=$fn
done
```

Output:

```
[root@localhost Directory]# sh Fibonacci.sh
Enter a number
10
Fibonacci Sequence
0 1 1 2 3 5 8 13 21 34 [root@localhost Directory]#
```

Code 3:

```
echo "Enter a number"
read x
fact=1
for (( i=x; i>0; i-- ))
do
    fact=$((fact * i))
done
```

Output:

```
[root@localhost Directory]# sh Factorial.sh
Enter a number
5
Factorial is 120
[root@localhost Directory]#
```

Conclusion: In this practical I learnt about for loops, if conditions and how to provide logical arguments to them to perform programs like factorial and Fibonacci.

Practical 4

Aim: Program maintenance using make utility

A. Write a program that is spread over two files.

B. Use following Makefile for program maintenance. To use make utility, use make Command.

1. Code for calculator.hh

```
int sum(int a , int b){
    return a+b;
}
int subtraction (int a , int b){
    return a-b;
}
int multiply (int a , int b) {
    return a*b;
}
float divide (int a , int b){
    float ans = float(a)/float(b);
    return ans;
}
```

2. Code for add.cpp

```
#include <iostream>
#include "calculator.hh"
using namespace std;
int main(){
    int a =5;
    int b =6;
    int ans = sum(a,b);
```

```
    cout << ans << " ";  
    return 0;  
}
```

3. Code for makefile.mk

a.out : hello1

hello1 : calculator.hh add.cpp

g++ -o hello1 add.cpp

Output:

```
g20dcs103@cloudshell:~$ make -f makefile2.mk  
g++ -o hello1 add.cpp  
g20dcs103@cloudshell:~$ ./hello1  
11 g20dcs103@cloudshell:~$
```

Conclusion: In this practical we learnt how to create a makefile that takes in dependencies and executes a given command using those dependencies to build an executable file.

Practical 5

Aim: Write programs using the following system calls of UNIX operating system: fork, exec, getpid, exit, wait, stat, readdir, opendir.

- A. Write a program to execute fork () and find out the process id by getpid() system call.**
- B. Write a program to execute following system call fork (), execl(), getpid(), exit(), wait() for a process.**
- C. Write a program to find out status of named file (program of working stat () system cal**

A) Code for fork() and getpid() system calls

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    // make two process which run same
    // program after this instruction
    printf("Calling fork() to create child process\n");
    printf("Made by : 20DCS103");

    int pid;
    pid = fork();

    if (pid == 0)
    {
        printf ( "\nParent Process id : %d \n",getpid() );
        printf ( "\nChild Process with parent id : %d\n", getppid() );
    }
}
```

```
    return 0;
}
```

B)

System call execl()

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/wait.h>
#include<unistd.h>
int main()
{
    printf("step before execl call\n");
    execl("/bin/ps", "ps", NULL);
    printf("step after execl call");
    return 0;
}
```

system call wait() and fork()

```
#include<stdio.h>
#include<sys/wait.h>
#include<unistd.h>
int main()
{
    if (fork() == 0)
        printf("HC: hello from child\n");
    else
    {
        printf("HP: hello from parent\n");
    }
}
```



```
        wait(NULL);
        printf("CT: child has terminated\n");
    }
    printf("Bye\n");
    return 0;
}
```

System call exit(), fork(), getpid()

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/wait.h>
#include<unistd.h>
int main()
{
    pid_t cpid;
    if (fork() == 0)
        exit(0);      /* terminate child */
    else
        cpid = wait(NULL); /* reaping parent */
    printf("Parent pid = %d\n", getpid());
    printf("Child pid = %d\n", cpid);

    return 0;
}
```

C) System call stat() for a file

```
#include<stdio.h>
#include<sys/stat.h>
```

```
#include<fcntl.h>
#include<stdlib.h>

void sfile(char const filename[]);

int main(){
    ssize_t read;
    char* buffer = 0;
    size_t buf_size = 0;

    printf("Enter the name of a file to check: \n");
    read = getline(&buffer, &buf_size, stdin);

    if (read <=0 ){
        printf("getline failed\n");
        exit(1);
    }

    if (buffer[read-1] == '\n'){
        buffer[read-1] = 0;
    }
    int s=open(buffer,O_RDONLY);
    if(s==-1){
        printf("File doesn't exist\n");
        exit(1);
    }
    else{
        sfile(buffer);
    }
    free(buffer);
}
```

```
    return 0;
}

void sfile(char const filename[]){
    struct stat sfile;

    if(stat(filename,&sfile)==-1){
        printf("Error Occurred\n");
    }

    printf("\nFile st_uid %d \n",sfile.st_uid);
    printf("\nFile st_blksize %ld \n",sfile.st_blksize);
    printf("\nFile st_gid %d \n",sfile.st_gid);
    printf("\nFile st_blocks %ld \n",sfile.st_blocks);
    printf("\nFile st_size %ld \n",sfile.st_size);
    printf("\nFile st_nlink %u \n",(unsigned int)sfile.st_nlink);
    printf("\nFile Permissions User\n");
    printf((sfile.st_mode & S_IRUSR)? "r":"-");
    printf((sfile.st_mode & S_IWUSR)? "w":"-");
    printf((sfile.st_mode & S_IXUSR)? "x":"-");
    printf("\n");
    printf("\nFile Permissions Group\n");
    printf((sfile.st_mode & S_IRGRP)? "r":"-");
    printf((sfile.st_mode & S_IWGRP)? "w":"-");
    printf((sfile.st_mode & S_IXGRP)? "x":"-");
    printf("\n");
    printf("\nFile Permissions Other\n");
    printf((sfile.st_mode & S_IROTH)? "r":"-");
    printf((sfile.st_mode & S_IWOTH)? "w":"-");
    printf((sfile.st_mode & S_IXOTH)? "x":"-");
    printf("\n");
```

```
}
```

Output:

A) fork() and getpid() system calls

```
g20dcs103@cloudshell:~/practical5 (os-lab-361206)$ ./fork
Calling fork() to create child process
Made by : 20DCS103

Parent Process id : 703

Child Process with parent id : 702
```

B) execl()

```
g20dcs103@cloudshell:~/practical5 (os-lab-361206)$ ./5_B
step before execl call
  PID TTY          TIME CMD
  440 pts/2    00:00:00 bash
  715 pts/2    00:00:00 ps
```

Wait() system call

```
g20dcs103@cloudshell:~/practical5 (os-lab-361206)$ ./5_B_wait
HC: hello from child
HP: hello from parent
Bye
CT: child has terminated
Bye
```

Exit() system call

```
g20dcs103@cloudshell:~/practical5 (os-lab-361206)$ ./5_B_exit
Parent pid = 724
Child pid = 725
```

C) stat system call on file named test.txt

```
g20dcs103@cloudshell:~/practical5 (os-lab-361206)$ ./5_c
Enter the name of a file to check:
test.txt

File st_uid 1000

File st_blksize 4096

File st_gid 1000

File st_blocks 8

File st_size 49

File st_nlink 1

File Permissions User
rw-

File Permissions Group
r--

File Permissions Other
r--
```

Conclusion: In this practical I learnt about different system calls that can be executed such as fork(), getpid(), execl(), wait() and exit() and also get status of a file by executing stat() function.

Practical 6

Aim: Write a C program in LINUX to implement Process scheduling algorithms and compare.

- A. First Come First Serve (FCFS) Scheduling**
- B. Shortest-Job-First (SJF) Scheduling**
- C. Priority Scheduling (Non-preemption) after completion extend on Preemption.**
- D. Round Robin (RR) Scheduling**

A) code for First Come First Serve (FCFS) scheduling

```
#include<stdio.h>

// Function to find the waiting time for all
// processes
void findWaitingTime(int processes[], int n,
                    int bt[], int wt[])
{
    // waiting time for first process is 0
    wt[0] = 0;

    // calculating waiting time
    for (int i = 1; i < n ; i++ )
        wt[i] = bt[i-1] + wt[i-1] ;
}

// Function to calculate turn around time
void findTurnAroundTime( int processes[], int n,
                    int bt[], int wt[], int tat[])
{
    // calculating turnaround time by adding
```

```
// bt[i] + wt[i]
for (int i = 0; i < n ; i++)
    tat[i] = bt[i] + wt[i];
}

//Function to calculate average time
void findavgTime( int processes[], int n, int bt[])
{
    int wt[n], tat[n], total_wt = 0, total_tat = 0;

    //Function to find waiting time of all processes
    findWaitingTime(processes, n, bt, wt);

    //Function to find turn around time for all processes
    findTurnAroundTime(processes, n, bt, wt, tat);

    //Display processes along with all details
    printf("Processes Burst time Waiting time Turn around time\n");

    // Calculate total waiting time and total turn
    // around time
    for (int i=0; i<n; i++)
    {
        total_wt = total_wt + wt[i];
        total_tat = total_tat + tat[i];
        printf(" %d ",(i+1));
        printf("      %d ", bt[i] );
        printf("      %d",wt[i] );
    }
```

```

        printf("        %d\n",tat[i] );
    }
    float s=(float)total_wt / (float)n;
    int t=(float)total_tat / (float)n;
    printf("Average waiting time = %f",s);
    printf("\n");
    printf("Average turn around time = %d ",t);
}

```

```

int main()
{
    //process id's
    int processes[] = { 1, 2, 3 };
    int n = sizeof processes / sizeof processes[0];

    //Burst time of all processes
    int burst_time[] = { 10, 5, 8 };

    findavgTime(processes, n, burst_time);
    return 0;
}

```

B) code for Shortest-Job-First (SJF) Scheduling

```

#include <stdio.h>

int main()
{
    int A[100][4]; // Matrix for storing Process Id, Burst
                  // Time, Average Waiting Time & Average

```



```
                                // Turn Around Time.

int i, j, n, total = 0, index, temp;

float avg_wt, avg_tat;

printf("Enter number of process: ");
scanf("%d", &n);
printf("Enter Burst Time:\n");

// User Input Burst Time and allotting Process Id.
for (i = 0; i < n; i++) {
    printf("P%d: ", i + 1);
    scanf("%d", &A[i][1]);
    A[i][0] = i + 1;
}

// Sorting process according to their Burst Time.
for (i = 0; i < n; i++) {
    index = i;
    for (j = i + 1; j < n; j++)
        if (A[j][1] < A[index][1])
            index = j;

    temp = A[i][1];
    A[i][1] = A[index][1];
    A[index][1] = temp;

    temp = A[i][0];
    A[i][0] = A[index][0];
    A[index][0] = temp;
}

A[0][2] = 0;

// Calculation of Waiting Times
```

```

    for (i = 1; i < n; i++) {
        A[i][2] = 0;
        for (j = 0; j < i; j++)
            A[i][2] += A[j][1];
        total += A[i][2];
    }
    avg_wt = (float)total / n;
    total = 0;
    printf("P    BT    WT    TAT\n");
    // Calculation of Turn Around Time and printing the
    // data.
    for (i = 0; i < n; i++) {
        A[i][3] = A[i][1] + A[i][2];
        total += A[i][3];
        printf("P%d    %d    %d    %d\n", A[i][0],
            A[i][1], A[i][2], A[i][3]);
    }
    avg_tat = (float)total / n;
    printf("Average Waiting Time= %f", avg_wt);
    printf("\nAverage Turnaround Time= %f", avg_tat);
    printf("\n");
}

```

C) code for Priority Scheduling (Non-preemption) after completion extend on Preemption.

i) Non-preemptive

```
#include <stdio.h>
```

```
//Function to swap two variables
```

```
void swap(int *a,int *b)
{
    int temp=*a;
    *a=*b;
    *b=temp;
}

int main()
{
    int n;
    printf("Enter Number of Processes: ");
    scanf("%d",&n);

    // b is array for burst time, p for priority and index for process id
    int b[n],p[n],index[n];
    for(int i=0;i<n;i++)
    {
        printf("Enter Burst Time and Priority Value for Process %d: ",i+1);
        scanf("%d %d",&b[i],&p[i]);
        index[i]=i+1;
    }
    for(int i=0;i<n;i++)
    {
        int a=p[i],m=i;

        //Finding out highest priority element and placing it at its desired position
        for(int j=i;j<n;j++)
        {
            if(p[j] > a)
```

```
        {
            a=p[j];
            m=j;
        }
    }

    //Swapping processes
    swap(&p[i], &p[m]);
    swap(&b[i], &b[m]);
    swap(&index[i],&index[m]);
}

// T stores the starting time of process
int t=0;

//Printing scheduled process
printf("Order of process Execution is\n");
for(int i=0;i<n;i++)
{
    printf("P%d is executed from %d to %d\n",index[i],t,t+b[i]);
    t+=b[i];
}
printf("\n");
printf("Process Id   Burst Time   Wait Time   TurnAround Time\n");
int wait_time=0;
for(int i=0;i<n;i++)
{
    printf("P%d       %d       %d       %d\n",index[i],b[i],wait_time,wait_time + b[i]);
```

```
        wait_time += b[i];
    }
    return 0;
}
```

ii) Preemptive Solution

```
#include<stdio.h>
```

```
struct process
```

```
{
    int WT,AT,BT,TAT,PT;
};
```

```
struct process a[10];
```

```
int main()
```

```
{
    int n,temp[10],t,count=0,short_p;
    float total_WT=0,total_TAT=0,Avg_WT,Avg_TAT;
    printf("Enter the number of the process\n");
    scanf("%d",&n);
    printf("Enter the arrival time , burst time and priority of the process\n");
    printf("AT BT PT\n");
    for(int i=0;i<n;i++)
    {
        scanf("%d%d%d",&a[i].AT,&a[i].BT,&a[i].PT);

        // copying the burst time in
        // a temp array for further use
    }
}
```

```
temp[i]=a[i].BT;
}

// we initialize the burst time
// of a process with maximum
a[9].PT=10000;

for(t=0;count!=n;t++)
{
    short_p=9;
    for(int i=0;i<n;i++)
    {
        if(a[short_p].PT>a[i].PT && a[i].AT<=t && a[i].BT>0)
        {
            short_p=i;
        }
    }

    a[short_p].BT=a[short_p].BT-1;

    // if any process is completed
    if(a[short_p].BT==0)
    {
        // one process is completed
        // so count increases by 1
        count++;
        a[short_p].WT=t+1-a[short_p].AT-temp[short_p];
        a[short_p].TAT=t+1-a[short_p].AT;
```

```
// total calculation
total_WT=total_WT+a[short_p].WT;
total_TAT=total_TAT+a[short_p].TAT;

}

}

Avg_WT=total_WT/n;
Avg_TAT=total_TAT/n;

// printing of the answer
printf("ID WT TAT\n");
for(int i=0;i<n;i++)
{
    printf("%d %d\t%d\n",i+1,a[i].WT,a[i].TAT);
}

printf("Avg waiting time of the process is %f\n",Avg_WT);
printf("Avg turn around time of the process is %f\n",Avg_TAT);

return 0;
}
```

D) code for Round Robin (RR) Scheduling

```
#include<stdio.h>
```

```
void main()
```

```
{
    // initialize the variable name
    int i, NOP, sum=0, count=0, y, quant, wt=0, tat=0, at[10], bt[10], temp[10];
    float avg_wt, avg_tat;
    printf(" Total number of process in the system: ");
    scanf("%d", &NOP);
    y = NOP; // Assign the number of process to variable y

    // Use for loop to enter the details of the process like Arrival time and the Burst Time
    for(i=0; i<NOP; i++)
    {
        printf("\n Enter the Arrival and Burst time of the Process[%d]\n", i+1);
        printf(" Arrival time is: \t"); // Accept arrival time
        scanf("%d", &at[i]);
        printf(" \nBurst time is: \t"); // Accept the Burst time
        scanf("%d", &bt[i]);
        temp[i] = bt[i]; // store the burst time in temp array
    }
    // Accept the Time qunat
    printf("Enter the Time Quantum for the process: \t");
    scanf("%d", &quant);
    // Display the process No, burst time, Turn Around Time and the waiting time
    printf("\n Process No \t\t Burst Time \t\t TAT \t\t Waiting Time ");
    for(sum=0, i = 0; y!=0; )
    {
        if(temp[i] <= quant && temp[i] > 0) // define the conditions
        {
            sum = sum + temp[i];
```



```
temp[i] = 0;
count=1;
}
else if(temp[i] > 0)
{
    temp[i] = temp[i] - quant;
    sum = sum + quant;
}
if(temp[i]==0 && count==1)
{
    y--; //decrement the process no.
    printf("\nProcess No[%d] \t\t %d\t\t\t %d\t\t\t %d", i+1, bt[i], sum-at[i], sum-at[i]-bt[i]);
    wt = wt+sum-at[i]-bt[i];
    tat = tat+sum-at[i];
    count =0;
}
if(i==NOP-1)
{
    i=0;
}
else if(at[i+1]<=sum)
{
    i++;
}
else
{
    i=0;
}
```

```

}

// represents the average waiting time and Turn Around time

avg_wt = wt * 1.0/NOP;

avg_tat = tat * 1.0/NOP;

printf("\n Average Turn Around Time: \t%f", avg_wt);

printf("\n Average Waiting Time: \t%f", avg_tat);

getchar();

}

```

Output:**A) FCFS**

```

g20dcs103@cloudshell:~/practical6 (os-lab-361206)$ ./FCFS
Processes Burst time Waiting time Turn around time
1          10          0          10
2           5         10          15
3           8         15          23
Average waiting time = 8.333333
Average turn around time = 16

```

B) SJF

```

Enter number of process: 5
Enter Burst Time:
P1: 4
P2: 10
P3: 1
P4: 6
P5: 5
P      BT      WT      TAT
P3     1       0       1
P1     4       1       5
P5     5       5      10
P4     6      10      16
P2    10      16      26
Average Waiting Time= 6.400000
Average Turnaround Time= 11.600000

```

C)

i) Priority Scheduling (non-preemptive)

```

Enter Number of Processes: 3
Enter Burst Time and Priority Value for Process 1: 10 2
Enter Burst Time and Priority Value for Process 2: 5 0
Enter Burst Time and Priority Value for Process 3: 8 1
Order of process Execution is
P1 is executed from 0 to 10
P3 is executed from 10 to 18
P2 is executed from 18 to 23

```

Process Id	Burst Time	Wait Time	TurnAround Time
P1	10	0	10
P3	8	10	18
P2	5	18	23

ii) Priority Scheduling (preemptive)

```

Enter the number of the process
3
Enter the arrival time , burst time and priority of the process
AT BT PT
0 3 3
1 5 1
2 2 2
ID WT TAT
1 7 10
2 0 5
3 4 6
Avg waiting time of the process is 3.666667
Avg turn around time of the process is 7.000000

```

D) Round Robin Scheduling

```
Total number of process in the system: 4

Enter the Arrival and Burst time of the Process[1]
Arrival time is: 0

Burst time is: 8

Enter the Arrival and Burst time of the Process[2]
Arrival time is: 1

Burst time is: 5

Enter the Arrival and Burst time of the Process[3]
Arrival time is: 2

Burst time is: 10

Enter the Arrival and Burst time of the Process[4]
Arrival time is: 3

Burst time is: 11
Enter the Time Quantum for the process: 5
```

Process No	Burst Time	TAT	Waiting Time
Process No[2]	5	9	4
Process No[1]	8	23	15
Process No[3]	10	26	16
Process No[4]	11	31	20
Average Turn Around Time:		13.750000	
Average Waiting Time:		22.250000	

Conclusion: In this practical I learnt about different types of scheduling algorithms. On comparing all the algorithms, we can see that Round Robin comes out to be the most efficient. All the scheduling algorithms have their own pros and cons and are used in different situations as per our needs.

Practical 7

Aim: Process control system calls:

A) The demonstration of fork ()

B) execve() and wait() system calls along with zombie and orphan states.

Code:

A) Demonstration of fork()

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{

    // make two process which run same
    // program after this instruction
    printf("Calling fork() to create child process\n");
    printf("Made by : 20DCS103");
    int pid;
    pid = fork();
    if (pid == 0)
    {
        printf ( "\nParent Process id : %d \n",getpid() );
        printf ( "\nChild Process with parent id : %d\n", getppid() );
    }
    return 0;
}
```

B) execve() and wait() system calls along with zombie and orphan states.**Parent Process c code**

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{
    int val[10],ele,status;
    pid_t pid;
    char* cval[10];
    char *newenviron[] = { NULL };
    int i,j,n,temp;
    printf("\nEnter the size for an array: ");
    scanf("%d",&n);
    printf("\nEnter %d elements : ", n);
    for(i=0;i<n;i++)
        scanf("%d",&val[i]);

    printf("\nEnter elements are: ");
    for(i=0;i<n;i++)
        printf("\t%d",val[i]);

    for(i=1;i<n;i++)
    {
        for(j=0;j<n-1;j++)
```

```
{
    if(val[j]>val[j+1])
    {
        temp=val[j];
        val[j]=val[j+1];
        val[j+1]=temp;
    }
}

printf("\nSorted elements are: ");
for(i=0;i<n;i++)
    printf("\t%d",val[i]);
wait(&status);
printf("\nparent process complete");

printf("\nEnter element to search: ");
scanf("%d",&ele);
val[i] = ele;
for (i=0; i < n+1; i++)
{
    char a[sizeof(int)];
    snprintf(a, sizeof(int), "%d", val[i]);

    cval[i] = malloc(sizeof(a));
    strcpy(cval[i], a);
}
cval[i]=NULL;
pid=fork();
if(pid==0)
{
```

```
    execve(argv[1], cval, newenviron);
    perror("Error in execve call...");
}
}
```

Child process C code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[],char *en[])
{
    int i,j,c,ele;
    int arr[argc];

    for (j = 0; j < argc-1; j++)
    {
        int n=atoi(argv[j]);
        arr[j]=n;
    }
    ele=atoi(argv[j]);
    i=0;
    j=argc-1;
    c=(i+j)/2;
    while(arr[c]!=ele && i<=j)
    {
        if(ele > arr[c])
            i = c+1;
        else
            j = c-1;
    }
```



```
        c = (i+j)/2;
    }
    if(i<=j)
        printf("\nElement Found in the given Array...!!!\n");
    else
        printf("\nElement Not Found in the given Array...!!!\n");
}
```

Output:**A)**

```
g20dcs103@cloudshell:~/practical5 (os-lab-361206)$ ./fork
Calling fork() to create child process
Made by : 20DCS102

Parent Process id : 703

Child Process with parent id : 702
```

B)

```
Enter the size for an array: 7

Enter 7 elements : 3 10 1 16 12 8 6

Entered elements are:   3      10      1      16      12      8      6
Sorted elements are:   1      3      6      8      10     12     16
parent process complete
Enter element to search: 10

Element Found in the given Array...!!!
```

Conclusion: In this practical I learnt about zombie and orphan processes. Zombie processes are created when the process dies but it isn't removed from process table. Orphan processes are created when the parent for some reason is terminated and it dies, leaving the child process as an orphan process.

Practical 8

Aim: Thread management using pthread library. Write a simple program to understand it.

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *print_message_function( void *ptr );

main()
{
    pthread_t thread1, thread2;
    char *message1 = "Thread 1";
    char *message2 = "Thread 2";
    int  iret1, iret2;

    /* Create independent threads each of which will execute function */

    iret1 = pthread_create( &thread1, NULL, print_message_function, (void*) message1);
    iret2 = pthread_create( &thread2, NULL, print_message_function, (void*) message2);
    /* Wait till threads are complete before main continues. Unless we */
    /* wait we run the risk of executing an exit which will terminate */
    /* the process and all threads before the threads have completed. */

    pthread_join( thread1, NULL);
    pthread_join( thread2, NULL);

    printf("Thread 1 returns: %d\n",iret1);
    printf("Thread 2 returns: %d\n",iret2);
    exit(0);
}
```

```
void *print_message_function( void *ptr )
{
    char *message;
    message = (char *) ptr;
    printf("%s \n", message);
}
```

Output:

```
g20dcs103@cloudshell:~/practical8$ gcc -pthread p8.c -o p8
g20dcs103@cloudshell:~/practical8$ ./p8
Thread 1
Thread 2
Thread 1 returns: 0
Thread 2 returns: 0
```

Conclusion: In this practical I learnt about pthread library. Here we create two new threads along with main thread and main thread waits for the completion of these newly created threads namely thread 1 and thread 2. After completion we join threads and main resumes its execution.

Practical 9

Aim: Write a C program in LINUX to implement inter process communication (IPC) Using Semaphore

Code:

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<sys/sem.h>
#include<string.h>
#include<errno.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>

#define SHM_KEY 0x12345
#define SEM_KEY 0x54321
#define MAX_TRIES 20

struct shmseg {
    int cntr;
    int write_complete;
    int read_complete;
};

void shared_memory_cntr_increment(int, struct shmseg*, int);
void remove_semaphore();

int main(int argc, char *argv[]) {
    int shmid;
    struct shmseg *shmp;
    char *bufptr;
    int total_count;
    int sleep_time;
```

```
pid_t pid;
if (argc != 2)
total_count = 10000;
else {
    total_count = atoi(argv[1]);
    if (total_count < 10000)
        total_count = 10000;
}
printf("Total Count is %d\n", total_count);
shmid = shmget(SHM_KEY, sizeof(struct shmseg), 0644|IPC_CREAT);

if (shmid == -1) {
    perror("Shared memory");
    return 1;
}
// Attach to the segment to get a pointer to it.
shmp = shmat(shmid, NULL, 0);

if (shmp == (void *) -1) {
    perror("Shared memory attach: ");
    return 1;
}
shmp->cntr = 0;
pid = fork();

/* Parent Process - Writing Once */
if (pid > 0) {
    shared_memory_cntr_increment(pid, shmp, total_count);
} else if (pid == 0) {
    shared_memory_cntr_increment(pid, shmp, total_count);
    return 0;
} else {
    perror("Fork Failure\n");
    return 1;
}
```

```
    }
    while (shmp->read_complete != 1)
        sleep(1);

    if (shmdt(shmp) == -1) {
        perror("shmdt");
        return 1;
    }

    if (shmctl(shmid, IPC_RMID, 0) == -1) {
        perror("shmctl");
        return 1;
    }
    printf("Writing Process: Complete\n");
    remove_semaphore();
    return 0;
}

/* Increment the counter of shared memory by total_count in steps of 1 */
void shared_memory_cntr_increment(int pid, struct shmseg *shmp, int total_count) {
    int cntr;
    int numtimes;
    int sleep_time;
    int semid;
    struct sembuf sem_buf;
    struct semid_ds buf;
    int tries;
    int retval;
    semid = semget(SEM_KEY, 1, IPC_CREAT | IPC_EXCL | 0666);
    //printf("errno is %d and semid is %d\n", errno, semid);

    /* Got the semaphore */
    if (semid >= 0) {
        printf("First Process\n");
```

```
sem_buf.sem_op = 1;
sem_buf.sem_flg = 0;
sem_buf.sem_num = 0;
retval = semop(semid, &sem_buf, 1);
if (retval == -1) {
    perror("Semaphore Operation: ");
    return;
}
} else if (errno == EEXIST) { // Already other process got it
    int ready = 0;
    printf("Second Process\n");
    semid = semget(SEM_KEY, 1, 0);
    if (semid < 0) {
        perror("Semaphore GET: ");
        return;
    }

    /* Waiting for the resource */
    sem_buf.sem_num = 0;
    sem_buf.sem_op = 0;
    sem_buf.sem_flg = SEM_UNDO;
    retval = semop(semid, &sem_buf, 1);
    if (retval == -1) {
        perror("Semaphore Locked: ");
        return;
    }
}
sem_buf.sem_num = 0;
sem_buf.sem_op = -1; /* Allocating the resources */
sem_buf.sem_flg = SEM_UNDO;
retval = semop(semid, &sem_buf, 1);

if (retval == -1) {
    perror("Semaphore Locked: ");
```

```
    return;
}
cntr = shmp->cntr;
shmp->write_complete = 0;
if (pid == 0)
    printf("SHM_WRITE: CHILD: Now writing\n");
else if (pid > 0)
    printf("SHM_WRITE: PARENT: Now writing\n");
/* Increment the counter in shared memory by total_count in steps of 1 */
for (numtimes = 0; numtimes < total_count; numtimes++) {
    cntr += 1;
    shmp->cntr = cntr;
    /* Sleeping for a second for every thousand */
    sleep_time = cntr % 1000;
    if (sleep_time == 0)
        sleep(1);
}
shmp->write_complete = 1;
sem_buf.sem_op = 1; /* Releasing the resource */
retval = semop(semid, &sem_buf, 1);

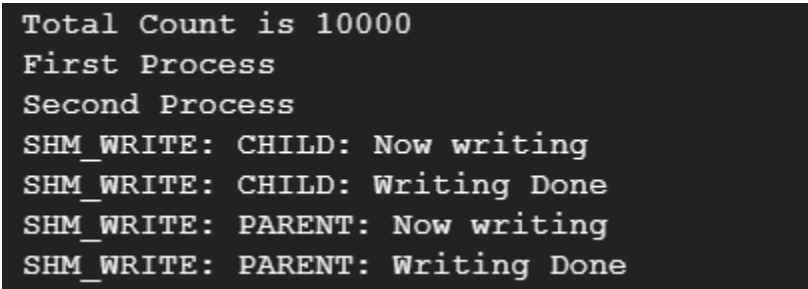
if (retval == -1) {
    perror("Semaphore Locked\n");
    return;
}

if (pid == 0)
    printf("SHM_WRITE: CHILD: Writing Done\n");
else if (pid > 0)
    printf("SHM_WRITE: PARENT: Writing Done\n");
    return;
}

void remove_semaphore() {
    int semid;
```



```
int retval;
semid = semget(SEM_KEY, 1, 0);
if (semid < 0) {
    perror("Remove Semaphore: Semaphore GET: ");
    return;
}
retval = semctl(semid, 0, IPC_RMID);
if (retval == -1) {
    perror("Remove Semaphore: Semaphore CTL: ");
    return;
}
return;
}
```

Output:

```
Total Count is 10000
First Process
Second Process
SHM_WRITE: CHILD: Now writing
SHM_WRITE: CHILD: Writing Done
SHM_WRITE: PARENT: Now writing
SHM_WRITE: PARENT: Writing Done
```

Conclusion: In this practical I learnt about creating inter-process communication using semaphore. We provide a shared lock mechanism in which child and parent process share it.

Practical 10

Aim: Simulate Following Page Replacement Algorithms.

A) First In First Out Algorithm

B) Least Recently Used Algorithm

C) Optimal Algorithm

Code for FIFO Algorithm:

```
#include <stdio.h>
int main()
{
    int referenceString[10], pageFaults = 0, m, n, s, pages, frames;
    printf("\nEnter the number of Pages:\t");
    scanf("%d", &pages);
    printf("\nEnter reference string values:\n");
    for( m = 0; m < pages; m++)
    {
        printf("Value No. [%d]:\t", m + 1);
        scanf("%d", &referenceString[m]);
    }
    printf("\n What are the total number of frames:\t");
    {
        scanf("%d", &frames);
    }
    int temp[frames];
    for(m = 0; m < frames; m++)
    {
        temp[m] = -1;
    }
    for(m = 0; m < pages; m++)
    {
        s = 0;
        for(n = 0; n < frames; n++)
        {
```

```

    if(referenceString[m] == temp[n])
    {
        s++;
        pageFaults--;
    }
}
pageFaults++;
if((pageFaults <= frames) && (s == 0))
{
    temp[m] = referenceString[m];
}
else if(s == 0)
{
    temp[(pageFaults - 1) % frames] = referenceString[m];
}
printf("\n");
for(n = 0; n < frames; n++)
{
    printf("%d\t", temp[n]);
}
}
printf("\nTotal Page Faults:\t%d\n", pageFaults);
return 0;
}

```

Code for LRU Algorithm:

```

#include<stdio.h>
int findLRU(int time[], int n){
    int i, minimum = time[0], pos = 0;

    for(i = 1; i < n; ++i){
        if(time[i] < minimum){
            minimum = time[i];
            pos = i;
        }
    }
}

```

```
}  
}  
return pos;  
}  
  
int main()  
{  
    int no_of_frames, no_of_pages, frames[10], pages[30], counter = 0, time[10], flag1, flag2, i, j,  
    pos, faults = 0;  
    printf("Enter number of frames: ");  
    scanf("%d", &no_of_frames);  
    printf("Enter number of pages: ");  
    scanf("%d", &no_of_pages);  
    printf("Enter reference string: ");  
    for(i = 0; i < no_of_pages; ++i){  
        scanf("%d", &pages[i]);  
    }  
  
    for(i = 0; i < no_of_frames; ++i){  
        frames[i] = -1;  
    }  
  
    for(i = 0; i < no_of_pages; ++i){  
        flag1 = flag2 = 0;  
  
        for(j = 0; j < no_of_frames; ++j){  
            if(frames[j] == pages[i]){  
                counter++;  
                time[j] = counter;  
                flag1 = flag2 = 1;  
                break;  
            }  
        }  
  
        if(flag1 == 0){
```

```
for(j = 0; j < no_of_frames; ++j){
    if(frames[j] == -1){
        counter++;
        faults++;
        frames[j] = pages[i];
        time[j] = counter;
        flag2 = 1;
        break;
    }
}

if(flag2 == 0){
    pos = findLRU(time, no_of_frames);
    counter++;
    faults++;
    frames[pos] = pages[i];
    time[pos] = counter;
}

printf("\n");

for(j = 0; j < no_of_frames; ++j){
    printf("%d\t", frames[j]);
}

printf("\n\nTotal Page Faults = %d", faults);

return 0;
}
```

Code for Optimal Algorithm:

```
#include<stdio.h>
int main()
```

```
{
    int no_of_frames, no_of_pages, frames[10], pages[30], temp[10], flag1, flag2, flag3, i, j, k,
    pos, max, faults = 0;

    printf("Enter number of frames: ");
    scanf("%d", &no_of_frames);

    printf("Enter number of pages: ");
    scanf("%d", &no_of_pages);

    printf("Enter page reference string: ");

    for(i = 0; i < no_of_pages; ++i){
        scanf("%d", &pages[i]);
    }

    for(i = 0; i < no_of_frames; ++i){
        frames[i] = -1;
    }

    for(i = 0; i < no_of_pages; ++i){
        flag1 = flag2 = 0;

        for(j = 0; j < no_of_frames; ++j){
            if(frames[j] == pages[i]){
                flag1 = flag2 = 1;
                break;
            }
        }

        if(flag1 == 0){
            for(j = 0; j < no_of_frames; ++j){
                if(frames[j] == -1){
                    faults++;
                    frames[j] = pages[i];
                    flag2 = 1;
                }
            }
        }
    }
}
```

```
        break;
    }
}
}

if(flag2 == 0){
    flag3 = 0;

    for(j = 0; j < no_of_frames; ++j){
        temp[j] = -1;

        for(k = i + 1; k < no_of_pages; ++k){
            if(frames[j] == pages[k]){
                temp[j] = k;
                break;
            }
        }
    }

    for(j = 0; j < no_of_frames; ++j){
        if(temp[j] == -1){
            pos = j;
            flag3 = 1;
            break;
        }
    }
    if(flag3 == 0){
        max = temp[0];
        pos = 0;
        for(j = 1; j < no_of_frames; ++j){
            if(temp[j] > max){
                max = temp[j];
                pos = j;
            }
        }
    }
}
```

```

    }
}
frames[pos] = pages[i];
faults++;
}
printf("\n");
for(j = 0; j < no_of_frames; ++j){
    printf("%d\t", frames[j]);
}
}
printf("\n\nTotal Page Faults = %d", faults);
return 0;
}

```

Output:**A) FIFO**

```

Enter the number of Pages:      5

Enter reference string values:
Value No. [1]:  5
Value No. [2]:  4
Value No. [3]:  3
Value No. [4]:  2
Value No. [5]:  1

What are the total number of frames:  4

5      -1      -1      -1
5       4      -1      -1
5       4       3      -1
5       4       3       2
1       4       3       2
Total Page Faults:      5

```


B) LRU

```
Enter number of frames: 3
Enter number of pages: 6
Enter reference string: 5 7 5 6 7 3
```

5	-1	-1
5	7	-1
5	7	-1
5	7	6
5	7	6
3	7	6

C) OA

```
Enter number of frames: 3
Enter number of pages: 10
Enter page reference string: 2 3 4 2 1 3 7 5 4 3
```

2	-1	-1
2	3	-1
2	3	4
2	3	4
1	3	4
1	3	4
7	3	4
5	3	4
5	3	4
5	3	4

Conclusion: In this practical I learnt about the page replacement algorithms and implemented it in a c program.

Practical 11

Aim: Thread synchronization using counting semaphores and mutual exclusion using mutex.

Code:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

pthread_t tid[2];
int counter;
pthread_mutex_t lock;

void* trythis(void* arg)
{
    pthread_mutex_lock(&lock);
    unsigned long i = 0;
    counter += 1;
    printf("\n Job %d has started\n", counter);
    for (i = 0; i < (0xFFFFFFFF); i++)
        printf("\n Job %d has finished\n", counter);
    pthread_mutex_unlock(&lock);

    return NULL;
}

int main(void)
{
    int i = 0;
    int error;

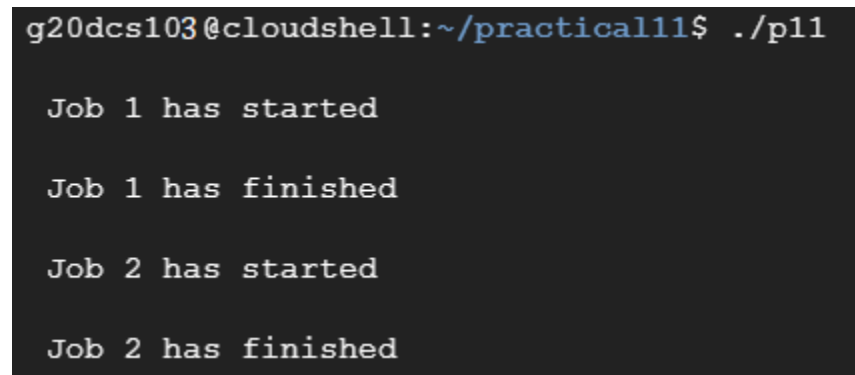
    if (pthread_mutex_init(&lock, NULL) != 0) {
```

```
        printf("\n mutex init has failed\n");
        return 1;
    }
    while (i < 2) {
        error = pthread_create(&(tid[i]),
                               NULL,
                               &trythis, NULL);

        if (error != 0)
            printf("\nThread can't be created :[%s]",
                  strerror(error));

        i++;
    }
    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    pthread_mutex_destroy(&lock);

    return 0;
}
```

Output:

```
g20dcs103@cloudshell:~/practical11$ ./p11

Job 1 has started

Job 1 has finished

Job 2 has started

Job 2 has finished
```

Conclusion: In this practical I learnt about mutexes using mutex and thread synchronization using counting semaphore. I created a simple job scheduling program with thread sync so that first job is started and finished before the second job begins.

Practical 12

Aim: Write a C program in LINUX to implement Bankers algorithm for Deadlock Avoidance.

Code:

```
#include <stdio.h>

int main()
{
    // P0, P1, P2, P3, P4 are the Process names here

    int n, m, i, j, k;
    n = 5; // Number of processes
    m = 3; // Number of resources
    int alloc[5][3] = { { 0, 1, 0 }, // P0 // Allocation Matrix
                        { 2, 0, 0 }, // P1
                        { 3, 0, 2 }, // P2
                        { 2, 1, 1 }, // P3
                        { 0, 0, 2 } }; // P4

    int max[5][3] = { { 7, 5, 3 }, // P0 // MAX Matrix
                     { 3, 2, 2 }, // P1
                     { 9, 0, 2 }, // P2
                     { 2, 2, 2 }, // P3
                     { 4, 3, 3 } }; // P4

    int avail[3] = { 3, 3, 2 }; // Available Resources

    int f[n], ans[n], ind = 0;
    for (k = 0; k < n; k++) {
        f[k] = 0;
    }
    int need[n][m];
    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++)
```

```

        need[i][j] = max[i][j] - alloc[i][j];
    }
    int y = 0;
    for (k = 0; k < 5; k++) {
        for (i = 0; i < n; i++) {
            if (f[i] == 0) {

                int flag = 0;
                for (j = 0; j < m; j++) {
                    if (need[i][j] > avail[j]){
                        flag = 1;
                        break;
                    }
                }

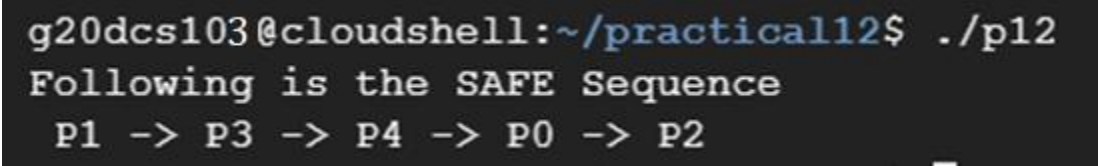
                if (flag == 0) {
                    ans[ind++] = i;
                    for (y = 0; y < m; y++)
                        avail[y] += alloc[i][y];
                    f[i] = 1;
                }
            }
        }
    }

    int flag = 1;

    for(int i=0;i<n;i++)
    {
        if(f[i]==0)
        {
            flag=0;
            printf("The following system is not safe");
            break;

```

```
    }  
    }  
  
    if(flag==1)  
    {  
        printf("Following is the SAFE Sequence\n");  
        for (i = 0; i < n - 1; i++)  
            printf(" P%d ->", ans[i]);  
        printf(" P%d", ans[n - 1]);  
    }  
  
    return (0);  
  
}
```

Output:

```
g20dcs103@cloudshell:~/practical12$ ./p12  
Following is the SAFE Sequence  
P1 -> P3 -> P4 -> P0 -> P2
```

Conclusion: In this practical I learnt about creating a c program for banker's algorithm which is an important deadlock prevention algorithm, with the help of this algorithm deadlock can be prevented and we can safely execute all the processes in the specified sequence.

Practical 13

Aim: Write a C program in LINUX to perform Memory allocation algorithms and Calculate Internal and External Fragmentation. (First Fit, Best Fit, Worst Fit).

Code for First Fit:

```
#include <stdio.h>

void implimentFirstFit(int blockSize[], int blocks, int processSize[], int processes)
{
    // This will store the block id of the allocated block to a process
    int allocate[processes];
    int occupied[blocks];

    // initially assigning -1 to all allocation indexes
    // means nothing is allocated currently
    for(int i = 0; i < processes; i++)
    {
        allocate[i] = -1;
    }

    for(int i = 0; i < blocks; i++){
        occupied[i] = 0;
    }

    // take each process one by one and find
    // first block that can accomodate it
    for (int i = 0; i < processes; i++)
    {
        for (int j = 0; j < blocks; j++)
        {
            if (!occupied[j] && blockSize[j] >= processSize[i])
            {
                // allocate block j to p[i] process
                allocate[i] = j;
                occupied[j] = 1;

                break;
            }
        }
    }
}
```

```
    }
    }
}
printf("\nProcess No.\tProcess Size\tBlock no.\n");
for (int i = 0; i < processes; i++)
{
    printf("%d \t\t %d \t\t", i+1, processSize[i]);
    if (allocate[i] != -1)
        printf("%d\n", allocate[i] + 1);
    else
        printf("Not Allocated\n");
}
//calculate fragmentation
int total_mem = 0;
int allocated_mem = 0;

for(int i=0 ; i<blocks ; i++) {
    total_mem += blockSize[i];
}

for (int i=0; i<processes ; i++){
    if (allocate[i] != -1){
        allocated_mem += processSize[i];
    }
}

printf("\n Total mem = %d and allocated memory = %d hence internal fragmentation = %d \n", total_mem, allocated_mem, (total_mem-allocated_mem));
}

void main()
{
    int blockSize[] = {30, 5, 10};
    int processSize[] = {10, 6, 9};
    int m = sizeof(blockSize)/sizeof(blockSize[0]);
    int n = sizeof(processSize)/sizeof(processSize[0]);

    implimentFirstFit(blockSize, m, processSize, n);
}
```


Code for Best Fit:

```
#include <stdio.h>

void implimentBestFit(int blockSize[], int blocks, int processSize[], int proccesses)
{
    // This will store the block id of the allocated block to a process
    int allocation[proccesses];
    int occupied[blocks];

    //calculate fragmentation
    int total_mem = 0;
    for(int i=0 ; i<blocks ; i++) {
        total_mem += blockSize[i];
    }
    // initially assigning -1 to all allocation indexes
    // means nothing is allocated currently
    for(int i = 0; i < proccesses; i++){
        allocation[i] = -1;
    }
    for(int i = 0; i < blocks; i++){
        occupied[i] = 0;
    }
    // pick each process and find suitable blocks
    // according to its size ad assign to it
    for (int i=0; i<proccesses; i++)
    {
        int indexPlaced = -1;
        for (int j=0; j<blocks; j++)
        {
            if (blockSize[j] >= processSize[i] && !occupied[j])
            {
                // place it at the first block fit to accomodate process
                if (indexPlaced == -1)
                    indexPlaced = j;

                // if any future block is larger than the current block where
                // process is placed, change the block and thus indexPlaced
                else if (blockSize[indexPlaced] < blockSize[j])
            }
        }
    }
}
```

```
        indexPlaced = j;
    }
}

// If we were successfully able to find block for the process
if (indexPlaced != -1)
{
    // allocate this block j to process p[i]
    allocation[i] = indexPlaced;

    // make the status of the block as occupied
    occupied[indexPlaced] = 1;

    // Reduce available memory for the block
    blockSize[indexPlaced] -= processSize[i];
}
}

printf("\nProcess No.\tProcess Size\tBlock no.\n");
for (int i = 0; i < processes; i++)
{
    printf("%d \t\t %d \t\t", i+1, processSize[i]);
    if (allocation[i] != -1)
        printf("%d\n", allocation[i] + 1);
    else
        printf("Not Allocated\n");
}

//calculate fragmentation
int allocated_mem = 0;

for (int i=0; i<processes ; i++){
    if (allocation[i] != -1){
        allocated_mem += processSize[i];
    }
}

printf("\n Total mem = %d and allocated memory = %d hence internal fragmentation = %d \n", total_mem, allocated_mem, (total_mem-allocated_mem));
}
```

```
int main()
{
    int blockSize[] = {5, 4, 3, 6, 7};
    int processSize[] = {1, 3, 5, 3};
    int blocks = sizeof(blockSize)/sizeof(blockSize[0]);
    int processes = sizeof(processSize)/sizeof(processSize[0]);

    implimentBestFit(blockSize, blocks, processSize, processes);

    return 0 ;
}
```

Code for Worst Fit:

```
#include<stdio.h>
#define max 25
void main()
{
    int frag[max],b[max],f[max],i,j,nb,nf,temp;
    static int bf[max],ff[max];
    printf("\n\tMemory Management Scheme - First Fit");
    printf("\nEnter the number of blocks:");
    scanf("%d",&nb);
    printf("Enter the number of files:");
    scanf("%d",&nf);
    printf("\nEnter the size of the blocks:-\n");
    for(i=1;i<=nb;i++)
    {
        printf("Block %d:",i);
        scanf("%d",&b[i]);
    }
    printf("Enter the size of the files :-\n");
    for(i=1;i<=nf;i++)
    {
        printf("File %d:",i);
        scanf("%d",&f[i]);
    }
    for(i=1;i<=nf;i++)
```

```

{
for(j=1;j<=nb;j++)
{
if(bf[j]!=1)
{
temp=b[j]-f[i];
if(temp>=0)
{
ff[i]=j;
break;
}
}
}
frag[i]=temp;
bf[ff[i]]=1;
}
printf("\nFile_no:\tFile_size :\tBlock_no:\tBlock_size:\tFragement");
for(i=1;i<=nf;i++)
printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,f[i],ff[i],b[ff[i]],frag[i]);
int fragmentation = 0;
for (i=1; i<=nf ; i++)
    fragmentation += frag[i];
printf("\n Total fragmentation that occurred is %d \n", fragmentation);
}

```

Output:**FF**

Process No.	Process Size	Block no.
1	10	1
2	6	3
3	9	Not Allocated

Total mem = 45 and allocated memory = 16 hence internal fragmentation = 29

BF

Process No.	Process Size	Block no.
1	1	5
2	3	4
3	5	1
4	3	2

Total mem = 25 and allocated memory = 12 hence internal fragmentation = 13

WF

```

Memory Management Scheme - First Fit
Enter the number of blocks:3
Enter the number of files:2

Enter the size of the blocks:-
Block 1:5
Block 2:2
Block 3:7
Enter the size of the files :-
File 1:1
File 2:4

File_no:      File_size :      Block_no:      Block_size:      Fragement
1             1             1             5             4
2             4             3             7             3

Total fragmentation that occurred is 7

```

Conclusion: In this practical I learnt to write a program for memory allocation algorithms namely First Fit, Best Fit and Worst Fit, also calculated the internal fragmentation that occurred in each of the algorithms.