Exp No:

Regd.No: □□□□□□□□□□

Date :

1. a) Study of Unix/Linux general purpose utility command list - man,who,cat, cd, cp, ps, ls, mv, rm, mkdir, rmdir, echo, more, date, time, kill, history, chmod, chown, finger, pwd, cal, logout, shutdown.

<table>
<tr><td>Command</td><td>man</td></tr>
<tr><td>Syntax</td><td>$ man [command] [specific file]</td></tr>
<tr><td>Description</td><td>“man” stands for manual which is a reference book of Linux Operating System. It is similar to HELP in popular software.</td></tr>
<tr><td>Examples</td><td>$ man ls

This command displays the all the information about “ls” command including its syntax, description and usage.</td></tr>
<tr><td></td><td></td></tr>
<tr><td>Command</td><td>who</td></tr>
<tr><td>Syntax</td><td>$ who - [option]
<table>
<tr><td>Option</td><td>Description</td></tr>
<tr><td>-a</td><td>Same as -b -d –login -p -r -t -T -u</td></tr>
<tr><td>-b</td><td>Time of last system boot</td></tr>
<tr><td>-d</td><td>Print dead processes</td></tr>
<tr><td>-H</td><td>Print line of column headings</td></tr>
<tr><td>-l</td><td>Print system login processes</td></tr>
<tr><td>-m</td><td>Only hostname and user associated with stdin</td></tr>
<tr><td>-p</td><td>Print active processes spawned by init</td></tr>
<tr><td>-q</td><td>All login names and number of users logged on</td></tr>
<tr><td>-r</td><td>Print current runlevel</td></tr>
<tr><td>-t</td><td>Print last system clock change</td></tr>
<tr><td>-T</td><td>Add user’s message status as +, – or ?</td></tr>
<tr><td>-u</td><td>List users logged in</td></tr>
</table></td></tr>
<tr><td>Description</td><td>“who” command displays who all are logged onto the network.

“who am i” command displays information about the current user.</td></tr>
<tr><td>Examples</td><td>1) $ who

root console july 2 10:30

aaa   tty01   july 2 10:40

bbb   tty02   july 2 10:55

2) $ who - H
3) $ who - u
4) $ who – b</td></tr>
<tr><td></td><td></td></tr>
<tr><td>Command</td><td>cat</td></tr>
</table>

Syntax	$ cat [argument] [specific file]
Description	“cat" is short for concatenate. This command is used to create, view and concatenate files.
Examples	1) $ cat /etc/passwd This command displays the "/etc/passwd" file on your screen. 2) $ cat /etc/profile This command displays the "/etc/profile" file on your screen. Notice that some of the contents of this file may scroll off of your screen. 3) $ cat file1 file2 file3 > file4 This command combines the contents of the first three files into the fourth file.
Command	cd, chdir
Syntax	$ cd [name of directory you want to move to]
Description	"cd" stands for change directory. It is the primary command for moving around the filesystem from current directory to the specified directory.
Examples	1) $ cd /usr This command moves you to the "/usr" directory. "/usr" becomes your current working directory. 2) $ cd /usr/fred Moves you to the "/usr/fred" directory. 3) $ cd /u*/f* Moves you to the "/usr/fred" directory - if this is the only directory matching this wildcard pattern. 4) $ cd Issuing the "cd" command without any arguments moves you to your *home* directory. 5) $ cd - Using the Korn shell, this command moves you back to your previous working

	directory. This is very useful when you're in the middle of a project, and keep moving back-and-forth between two directories.
Command	cp
Syntax	$ cp [options] file1 file2 $ cp [options] files directory
Options	-b backup files that are about to be overwritten or removed -i interactive mode; if destination exists, you'll be asked whether to overwrite the file -p preserves the original file's ownership, group, permissions, and timestamp
Description	The "cp" command is used to copy files and directories. Note that when using the cp command, you must always specify both the source and destination of the file(s) to be copied. The cp command copies both text and binary files.
Examples	1) $ cp .profile .profile.bak This command copies your ".profile" to a file named ".profile.bak". 2) $ cp /usr/fred/Chapter1 . This command copies the file named "Chapter1" in the "/usr/fred" directory to the current directory. This example assumes that you have write permission in the current directory. 3) $ cp /usr/fred/Chapter1 /usr/mary This command copies the "Chapter1" file in "/usr/fred" to the directory named "/usr/mary". This example assumes that you have write permission in the "/usr/mary" directory. 4) $ cp file1 file2 This command simply copies file1 to file2.
Command	ps
Syntax	$ ps [options]
Description	The "ps" command (process statistics) lets you check the status of processes that are running on your Unix system. It gives a snapshot of the current process's attribute.
Examples	1) $ ps PID TTY TIME CMD 476 tty03 00:00:01 login

	659 tty03 00:00:01 sh 684 tty03 00:00:00 ps The ps command by itself shows minimal information about the processes *you* are running. Without any arguments, this command will not show information about other processes running on the system. 2) $ ps -f The -f argument tells ps to supply *full* information about the processes it displays. In this example, ps displays full information about the processes *you* are running. 3) $ ps -e The -e argument tells the ps command to show *every* process running on the system. 4) $ ps -ef The -e and -f arguments are normally combined like this to show full information about every process running on the system. *This is probably the most often-used form of the ps command.* 5) $ ps -ef | more Because the output normally scrolls off the screen, the output of the ps -ef command is often piped into the more command. The more command lets you view one screenful of information at a time. 6) $ ps -fu fred This command shows full information about the processes currently being run by the user named *fred* (the -u option lets you specify a username).
Command	ls
Syntax	$ ls [options] [names]
Description	"ls" stands for list. It is used to list information about files and directories.

Examples	1) $ ls This is the basic "ls" command, with no options. It provides a very basic listing of the files in your current working directory. Filenames beginning with a decimal are considered *hidden* files, and they are not shown. 2) $ ls -a The -a option tells the ls command to report information about all files, including hidden files. 3) $ ls -l The -l option tells the "ls" command to provide a *long* listing of information about the files and directories it reports. The long listing will provide important information about file permissions, user and group ownership, file size, and creation date. 4) $ ls -al This command provides a *long* listing of information about *all* files in the current directory. It combines the functionality of the -a and -l options. *This is probably the most used version of the ls command*. 5) $ ls -al /usr This command lists long information about all files in the "/usr" directory. 6) $ ls -alr /usr | more This command lists long information about all files in the "/usr" directory, and all sub-directories of /usr. The -r option tells the ls command to provide a *recursive* listing of all files and sub-directories. 7) $ ls -ld /usr Rather than list the files contained in the /usr directory, this command lists information about the /usr directory itself (without generating a listing of the contents of /usr). This is very useful when you want to check the permissions of the directory, and not the files the directory contains.
Command	mv
Syntax	$ mv [options] sources target
Options	-b backup files that are about to be overwritten or removed. -i interactive mode; if destination file exists, you'll be asked whether to overwrite the

	file or not.
Description	The "mv" command is used to move and rename files.
Examples	1) $ mv Chapter1 Chapter1.bad This command renames the file "Chapter1" to the new name "Chapter1.bad". 2) $ mv Chapter1 garbage This command renames the file "Chapter1" to the new name "garbage". (Notice that if "garbage" is a directory, "Chapter1" would be moved into that directory). 3) $ mv Chapter1 /tmp This command moves the file "Chapter1" into the directory named "/tmp". 4) $ mv tmp tmp.old Assuming in this case that tmp is a directory, this example renames the directory tmp to the new name tmp.old.
Command	rm
Syntax	$ rm [options] files
Options	-d, --directory unlink FILE, even if it is a non-empty directory (super-user only) -f, --force ignore nonexistent files, never prompt -i, --interactive prompt before any removal -r, -r, --recursive remove the contents of directories recursively -v, --verbose explain what is being done
Description	The "rm" command is used to remove files and directories. (Warning - be very careful when removing files and directories!)
Examples	1) $ rm Chapter1.bad This command deletes the file named "Chapter1.bad" (assuming you have permission to delete this file).

	2) $ rm Chapter1 Chapter2 Chapter3 This command deletes the files named "Chapter1", "Chapter2", and "Chapter3". 3) $ rm -i Chapter1 Chapter2 Chapter3 This command prompts you before deleting any of the three files specified. The -i option stands for *inquire*. You must answer y (for yes) for each file you really want to delete. This can be a safer way to delete files. 4) $ rm *.html This command deletes all files in the current directory whose filename ends with the characters ".html". 5) $ rm index* This command deletes all files in the current directory whose filename begins with the characters "index". 6) $ rm -r new-novel This command deletes the directory named "new-novel". This directory, and all of its' contents, are erased from the disk, including any sub-directories and files.
Command	grep
Syntax	$ grep [options] regular expression [files]
Options	-i case-insensitive search -n show the line# along with the matched line -v invert match, e.g. find all lines that do NOT match -w match entire words, rather than substrings
Description	Think of the "grep" command as a "search" command (most people wish it was named "search"). It is used to search for text strings within one or more files.
Examples	1) $ grep 'fred' /etc/passwd This command searches for all occurrences of the text string 'fred' within the "/etc/passwd" file. It will find and print (on the screen) all of the lines in this file that contain the text string 'fred', including lines that contain usernames like "fred" - and also "alfred". 2) $ grep '^fred' /etc/passwd This command searches for all occurrences of the text string 'fred' within the

	"/etc/passwd" file, but also requires that the "f" in the name "fred" be in the first column of each record (that's what the caret character tells grep). Using this more-advanced search, a user named "alfred" would not be matched, because the letter "a" will be in the first column. 3) $ grep 'joe' * This command searches for all occurrences of the text string 'joe' within all files of the current directory.
Command	mkdir
Syntax	$ mkdir [options] directory name
Description	The "mkdir" command is used to create new directories (sub-directories).
Examples	1) $ mkdir tmp This command creates a new directory named "tmp" in your current directory. (This example assumes that you have the proper permissions to create a new sub-directory in your current working directory.) 2) $ mkdir memos letters e-mail This command creates three new sub-directories (memos, letters, and e-mail) in the current directory. 3) $ mkdir /usr/fred/tmp This command creates a new directory named "tmp" in the directory "/usr/fred". "tmp" is now a sub-directory of "/usr/fred". (This example assumes that you have the proper permissions to create a new directory in /usr/fred.) 4) $ mkdir -p /home/joe/customer/acme This command creates a new directory named /home/joe/customer/acme, and creates any intermediate directories that are needed. If only /home/joe existed to begin with, then the directory "customer" is created, and the directory "acme" is created inside of customer.
Command	rmdir
Syntax	$ rmdir [options] directories
Description	The "rmdir" command is used to remove directories. (Warning - be very careful when removing files and directories!)
Examples	$rmdir customer This command deletes the directory named "customer" (assuming you have

	permission to delete this directory).
Command	echo
Syntax	$ echo [options] IDs
Description	The “echo” command is used to display a line of text, or messages. It takes zero, one r more arguments.
Examples	1) $ echo hi This displays a “hi” msg on the terminal 2) $ echo “Hello World” Two words can be displayed using echo by placing them in quotes.The output of the above command will be – Hello World
Command	more
Syntax	$ more [filename]
Description	This command is used to display the contents of a file. It allows us to set the output page size and pauses at the end of each page to allow us to read the file.
Examples	$ more chapter1
Command	date
Syntax	$ date [- options]
Description	The “date” command prints the date and time. Using this command, the user ca display the current date along with the time nearest to the second.
Examples	$ date This displays the system’s current date and time in IST.
Command	time
Syntax	$ time [options]
Description	This command is used to know the resource usage. It runs a program or command with given arguments, generates a timing statistics about the program run and directs this statistics report to the standard output.
Examples	$ time Real 0m0.000s User 0m0.000s Sys 0m0.000s
Command	kill
Syntax	$ kill [options] IDs

Description	The command “kill” ends one or more process IDs. In order to do this you must own the process or be designated a privileged user. To find the process ID of a certain job use “ps” command.
Examples	$ kill 456 Here 456 is the PID of the process.
Command	history
Syntax	$ history
Description	The command “history” is used to display the history of previous executed commands. When used without options, it displays the command history list with line numbers.
Examples	$ history 1 pwd 2 date 3 cal
Command	chmod
Syntax	$ chmod<permissions><filename>
Description	The “chmod” command is used to change permissions of a file after its creation. Only the Owner or Super User can change file permissions.
Examples	$ chmod u+x sample $ls - l sample -r w x r --r --sample
Command	chown
Syntax	$ chown <new_owner><filename>
Description	The command “chown” is used to change the file’s owner and group. Only the owner can change the major attributes of a file. Sometimes it is necessary to change the ownership of a file.
Examples	$ls -l sample -rwxr--r-x 1 dhoni July 19 11:55 sample $chown virat sample $ls -l sample -rwxr--r-x 1 virat July 19 11:55 sample.

Command	finger
Syntax	$ finger
Description	It is a user information lookup command. The “finger” command displays information about the system users.
Examples	$finger Login Name TTY Idle When Where root superuser tty01 1:24 Mon 14:00 501 ABD tty02 0:20 Mon 15:40 abd.cric.com 502 Federer tty03 2:12 Mon 15:40 fedrr.batm.com
Command	pwd
Syntax	$ pwd
Description	"pwd" stands for print working directory. It displays your current position in the UNIX filesystem. i.e., the location of the current directory in the directory sttucture. There are no options (or arguments) with the "pwd" command
Examples	$ pwd It is simply used to report your current working directory.
Command	cal
Syntax	$ cal <month><year>
Description	This command is used to print the calendar of a specific month or a specific year.
Examples	$ cal 05 2021
Command	logout
Syntax	$ logout
Description	This command is used to logout from the shell.
Examples	$ logout
Command	shutdown
Syntax	$shutdown
Description	This command is used to shutting down the system in a safe way.
Examples	$ shutdown

Exp No:

Regd.No: □□□□□□□□□□

Date :

b) Study of vi editor.

What is vi?

The fundamental UNIX editor is called **vi**. Vi stands for visual. It is a primitive editor, but it will be found on every UNIX system, and works well when logging in remotely from home. There are just a few commands that one needs to learn at first, and they will be sufficient to write simple files. The editor allows the users to see a portion of a file on the screen and to modify characters and lines by simply typing at the cursor positon.

The Basics of vi

There are essentially three Modes in vi, the **Insert Mode** where one is typing in new text, the **Edit Mode** where one is modifying text that is already there, and the **Command Mode** where one is interacting with the operating system with actions like reading and writing files.

Most of the commands require just a simple letter or two, but be careful to note that vi is ***case sensitive***, and capital letters do different things from lower case letters. It is not necessary to learn all of these commands, but eventually they will become second nature. The basic operations needed to create a file are related to each of those modes, so let's look at them separately.

INSERT Mode - For adding text to a file
The three most common ways to enter the Insert Mode are:

Letter	Action
i	Starts inserting in front of the current cursor position
I	Starts adding at the front of the line
a	Starts adding after the cursor
A	Starts adding at the end of the line
o	Starts opening a new line underneath the cursor
O	Starts opening a line above the cursor.

<Esc>	Gets out of Insert Mode

EDIT Mode

- Generally for moving the cursor and deleting stuff.
In the Edit Mode, the keys do not type letters, but do other actions such as cursor movement, deletions, copying lines, etc.

Letter	***Simple Cursor Movement***	
h	Moves cursor left one space	Note: the Arrow keys do work locally, but sometimes mess up over a network.
j	Moves cursor down one line	
k	Moves cursor up one line	
l	Moves cursor right one space	
Fast Cursor Movement		
w	Moves the cursor a full word at a time to the right	
b	Moves the cursor back to the left a word at a time	
^	Moves the cursor to the front of a line	
$	Moves the cursor to the end of a line	
<ctrl>f	Moves the cursor forward a full page of text at a time	
<ctrl>b	Moves the cursor backward a full page of text at a time	
Modifying Text		
x	Deletes the character under the cursor	
dd	Deletes the line where the cursor is located (type d twice!)	

n dd	Delete ***n*** consecutive lines (***n*** is an integer)
r	Replaces the character under the cursor with the next thing typed
J	Joins current line with the one below (Capital J!)
u	Undoes the last edit operation
<ctrl> r	Redo (Undoes the last undo operation)
Cut and Paste Operations	
yy	Copies or yanks a line (5yy yanks 5 lines)
p	Puts the yanked text on the line below the cursor (lower case p)
P	Puts the yanked text above the current line (capital P)
Note: If vi is already in the input mode, text from that or another window may be highlighted using the left mouse button, and copied into place by pressing the middle mouse button.	

COMMAND Mode - For interacting with the operating system.
To enter the Command Mode, a colon " **:** " must precede the actual command.

Letter	**Action**
: r <file>	reads a file from disk into the vi editor
: w <file>	writes current file to disk
: wq	writes the file and quits vi
: q!	quits without writing (useful if you've messed up!)

Lets start !!

<u>Starting with vi editor:</u>

Syntax:

$vi filename

Editing the file:

- Open the file using $ vi filename

- To add text at the end of the file, position the cursor at the last character of the file.

- Switch from command mode to text input mode by pressing ‘a’.

- Here ‘a’ stands for append.

- Inserting text in the middle of the file is possible by pressing ‘i’.

- The editor accepts and inserts the typed character until Esc key is pressed.

Saving text:

:w – save the file and remains in edit mode

:wq – save the file and quits from edit mode

:q – quit without changes from edit mode

Quitting vi:

Press zz or ‘:wq’ in command mode.

c) Study of Bash shell, Bourne shell and C shell in Unix/Linux operating system.

Types of Shells: The Shells are 4 types

a) The Bourne Shell (sh)

b) The C Shell (csh)

c) The Korn Shell (ksh)

d) The Bourne-Again Shell (bash)

Exp No:

Regd.No: □□□□□□□□□□

Date :

Study of Bash shell

Bourne again shell (Bash) is a free Unix shell that can be used in place of the Bourne shell. It is a complete implementation of the IEEE Portable Operating System Interface for Unix (POSIX) and Open Group shell specification.

Bash is basically a command processor that typically runs in a text window, allowing the user to type commands that cause actions. It can read commands from a file, called a script. Like all Unix shells it supports the following:

- File name wildcarding
- Piping
- Hear documents
- Command execution
- Variables and control structures for condition testing and iteration

Bash was written for the GNU Project by Brian Fox. It is called Bourne again shell for many reasons, the first being that it is the open-source version of the Bourne shell and the second as a pun on the concept of being born again. Its acronym is also a description of what the project did, which
was to bash together sh, csh, and ksh features.

A Unix shell is a command-line interpreter that provides users with a basic user interface. It allows users to communicate with the system through a series of commands that are typed in the command-line window. There are no buttons or pop-up windows in a shell, simply lots and lots of text.

Essentially, Bash allows users of Unix-like systems to control the innermost components of the operating system using text-based commands.

Bash has a number of extensions and runs on Unix-like operating systems like Linux and Mac OS X. It was ported to Windows through the Subsystem for UNIX-based Applications (SUA) and by POSIX emulation using Cygwin or MSYS. It can even be used in MS-DOS.

Study of Bourne shell

A Bourne shell (sh) is a UNIX shell or command processor that is used for scripting. It was developed in 1977 by Stephen Bourne of AT&T and introduced in UNIX Version 7, replacing the Mashey shell (sh).

The Bourne shell is also known by its executable program name, "sh" and the dollar symbol, "$," which is used with command prompts

A Bourne shell interprets and executes user defined commands and provides command based programming abilities. A Bourne shell enables the writing and executing of shell scripts, which provide basic program control flow, control over input/output (I/O) file descriptors and all key features required to create scripts or structured programs for shell.

A Bourne shell also executes commands and functions that are predefined or integrated; files that search or follow a command path, as well as text file commands.

The Bourne shell's architectural code was implemented in subsequent versions of UNIX shell, including Bourne Again shell (Bash), Korn shell and Zsh shell.

Study of C shell

The C shell was created by Bill Joy while he was a graduate student at UC Berkeley in the late 1970s. It was first released as part of the 2BSD Berkeley Software Distribution of Unix in 1978.

The C shell gets its name from its syntax, which is intended to resemble the C programming language.

The C shell (csh) is a command shell for Unix-like systems that was originally created as part of the Berkeley Software Distribution (BSD) in 1978. Csh can be used for entering commands interactively or in shell scripts. The shell introduced a number of improvements over the earlier Bourne shell designed for interactive use. These include history, editing operations, a directory stack, job control and tilde completion. Many of these features were adopted in the Bourne Again shell (bash), Korn shell (ksh) and in the Z shell (zsh). A modern variant, tcsh, is also very popular.

The C shell introduced features that were intended to make it easier to use interactively at the command line, though like other shells it is capable of being scripted. One of the most notable features was command history. Users can recall previous commands they have entered and either repeat them or edit these commands. Aliases allow users to define short names to be expanded into longer commands. A directory stack lets users push and pop directories on the stack to jump back and forth quickly. The C shell also introduced the standard tilde notation where "~" represents a user's home directory.

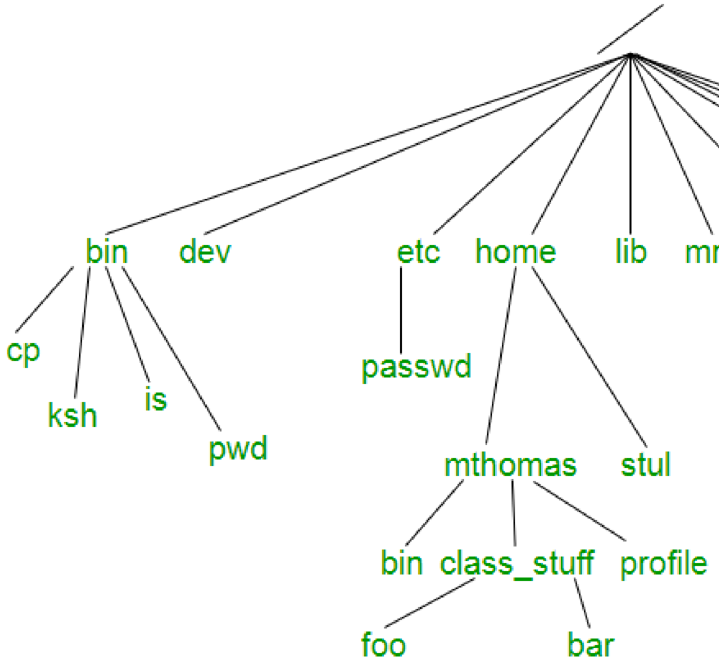
Most of these features have been incorporated into later shells, include the Bourne Again shell, the Korn shell and the Z shell. A popular variant is tsch, which is the current default shell on BSD systems, as well as on early versions of Mac OS X.

d) Study of Unix/Linux file system (tree structure).

Unix File System

Unix file system is a logical method of **organizing and storing** large amounts of information in a way that makes it easy to manage. A file is a smallest unit in which the information is stored. Unix file system has several important features. All data in Unix is organized into files. All files are organized into directories. These directories are organized into a tree-like structure called the file system.

Files in Unix System are organized into multi-level hierarchy structure known as a directory tree. At the very top of the file system is a directory called "root" which is represented by a "/". All other files are "descendants" of root.



Directories or Files and their description –

- **/ :**The slash / character alone denotes the root of the filesystem tree.
- **/bin :**Stands for "binaries" and contains certain fundamental utilities, such as ls or cp, which are generally needed by all users.
- **/boot :**Contains all the files that are required for successful booting process.

- **/dev :**Stands for “devices”. Contains file representations of peripheral devices and pseudo-devices.
- **/etc :**Contains system-wide configuration files and system databases. Originally also contained “dangerous maintenance utilities” such as init,but these have typically been moved to /sbin or elsewhere.
- **/home :**Contains the home directories for the users.
- **/lib :**Contains system libraries, and some critical files such as kernel modules or device drivers.
- **/media :**Default mount point for removable devices, such as USB sticks, media players, etc.
- **/mnt :**Stands for “mount”. Contains filesystem mount points. These are used, for example, if the system uses multiple hard disks or hard disk partitions. It is also often used for remote (network) filesystems, CD-ROM/DVD drives, and so on.
- **/proc :**procfs virtual filesystem showing information about processes as files.
- **/root :**The home directory for the superuser “root” – that is, the system administrator. This account’s home directory is usually on the initial filesystem, and hence not in /home (which may be a mount point for another filesystem) in case specific maintenance needs to be performed, during which other filesystems are not available. Such a case could occur, for example, if a hard disk drive suffers physical failures and cannot be properly mounted.
- **/tmp :**A place for temporary files. Many systems clear this directory upon startup; it might have tmpfs mounted atop it, in which case its contents do not survive a reboot, or it might be explicitly cleared by a startup script at boot time.
- **/usr :**Originally the directory holding user home directories,its use has changed. It now holds executables, libraries, and shared resources that are not system critical, like the X Window System, KDE, Perl, etc. However, on some Unix systems, some user accounts may still have a home directory that is a direct subdirectory of /usr, such as the default as in Minix. (on modern systems, these user accounts are often related to server or system use, and not directly used by a person).
- **/usr/bin :** This directory stores all binary programs distributed with the operating system not residing in /bin, /sbin or (rarely) /etc.
- **/usr/include :**Stores the development headers used throughout the system. Header files are mostly used by the **#include** directive in C/C++ programming language.
- **/usr/lib :**Stores the required libraries and data files for programs stored within /usr or elsewhere.
- **/var :**A short for “variable.” A place for files that may change often – especially in size, for example e-mail sent to users on the system, or process-ID lock files.
- **/var/log :**Contains system log files.
- **/var/mail :**The place where all the incoming mails are stored. Users (other than root) can access their own mail only. Often, this directory is a symbolic link to /var/spool/mail.
- **/var/spool :**Spool directory. Contains print jobs, mail spools and other queued tasks.
- **/var/tmp :**A place for temporary files which should be preserved between system reboots.

e) Study of .bashrc, /etc/bashrc and Environment variables.

An important Unix concept is the **environment**, which is defined by environment variables. Some are set by the system, others by you, yet others by the shell, or any program that loads another program.

A variable is a character string to which we assign a value. The value assigned could be a number, text, filename, device, or any other type of data.

For example, first we set a variable TEST and then we access its value using the **echo** command −

```
$TEST="Unix Programming"
$echo $TEST
```

It produces the following result.

```
Unix Programming
```

Note that the environment variables are set without using the **$** sign but while accessing them we use the $ sign as prefix. These variables retain their values until we come out of the shell.

When you log in to the system, the shell undergoes a phase called **initialization** to set up the environment. This is usually a two-step process that involves the shell reading the following files −

- /etc/profile
- profile

The process is as follows −

- The shell checks to see whether the file **/etc/profile** exists.
- If it exists, the shell reads it. Otherwise, this file is skipped. No error message is displayed.
- The shell checks to see whether the file **.profile** exists in your home directory. Your home directory is the directory that you start out in after you log in.
- If it exists, the shell reads it; otherwise, the shell skips it. No error message is displayed.

As soon as both of these files have been read, the shell displays a prompt −

```
$
```

This is the prompt where you can enter commands in order to have them executed.

Note − The shell initialization process detailed here applies to all **Bourne** type shells, but some additional files are used by **bash** and **ksh**.

The .profile File

The file **/etc/profile** is maintained by the system administrator of your Unix machine and contains shell initialization information required by all users on a system.

The file **.profile** is under your control. You can add as much shell customization information as you want to this file. The minimum set of information that you need to configure includes −

- The type of terminal you are using.
- A list of directories in which to locate the commands.
- A list of variables affecting the look and feel of your terminal.

You can check your **.profile** available in your home directory. Open it using the vi editor and check all the variables set for your environment.

Environment Variables

Following is the partial list of important environment variables. These variables are set and accessed as mentioned below −

Sr.No.	Variable & Description
1	**DISPLAY** Contains the identifier for the display that **X11** programs should use by default.
2	**HOME** Indicates the home directory of the current user: the default argument for the cd **built-in** command.
3	**IFS** Indicates the **Internal Field Separator** that is used by the parser for word splitting after expansion.
4	**LANG** LANG expands to the default system locale; LC_ALL can be used to override this. For example, if its value is **pt_BR**, then the language is set to (Brazilian) Portuguese and the locale to Brazil.
5	**LD_LIBRARY_PATH** A Unix system with a dynamic linker, contains a colonseparated list of directories that the dynamic linker should search for shared objects when building a process image after exec, before searching in any other directories.

6	**PATH** Indicates the search path for commands. It is a colon-separated list of directories in which the shell looks for commands.
7	**PWD** Indicates the current working directory as set by the cd command.
8	**RANDOM** Generates a random integer between 0 and 32,767 each time it is referenced.
9	**SHLVL** Increments by one each time an instance of bash is started. This variable is useful for determining whether the built-in exit command ends the current session.
10	**TERM** Refers to the display type.
11	**TZ** Refers to Time zone. It can take values like GMT, AST, etc.
12	**UID** Expands to the numeric user ID of the current user, initialized at the shell startup.
13	**MAIL** This variable holds the absolute pathname of the file where user’s mail is kept. Usually the name of this file is the user’s login name
14	**SHELL** This variable contains the name of the users shell program in the form of absolute pathname. System administrator sets the default shell If required, user can change it
15	**TERM** This variable holds the information regarding the type of the terminal being used. If TERM is not set properly, utilities like vi editor will not work

Following is the sample example showing few environment variables −

```
$ echo $HOME
/root
]$ echo $DISPLAY

$ echo $TERM
```

```
xterm
$ echo $PATH
/usr/local/bin:/bin:/usr/bin:/home/amrood/bin:/usr/local/bin
$
```

.bashrc

.bashrc file is automatically executed when new terminal(shell) is opened.
Purpose of bashrc file:

- You can export environment variables(So there is no need to export environment variable every time)
- You can define aliases
- You can provide the path for cross compiler
- You can add your own script which can start automatically whenever new shell is opened.
- You can change the history length

/etc/bashrc

- Like .bash_profile you will also commonly see a .bashrc file in your home directory. This file is meant for settingcommand aliases and functions used by bash shell users.
- Just like the /etc/profile is the system wide version of .bash_profile. The /etc/bashrc for Red Hat and /etc/bash.bashrc in Ubuntu is the system wide version of .bashrc.

Interestingly enough in the Red Hat implementation the /etc/bashrc also executes the shell scripts within /etc/profile.d but only if the users shell is a Interactive Shell (aka Login Shell)

2.Write a C program that makes a copy of a file using standard I/O, and system calls.

DESCRIPTION:-The *open()* function opens the file associated with file descriptor in the specified mode. The*read()* function attempts to read nbytes from the file associated with file descriptor and places the characters read into *buffer*. The *write()* function attempts to write nbytes from *buffer* to the file associated with file descriptor. The*close()* function closes the file associated with the file. descriptor.

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/types.h>
#include <unistd.h>
 #define BUF_SIZE 8192

int main(int argc, char* argv[])
{
int fd1, fd2;          /* Input and output file descriptors */
   ssize_t  ret_in;         /* Number of bytes returned by read()and write() */
char buffer[BUF_SIZE];       /* Character buffer */

if(argc != 3){
printf ("Usage: cp file1 file2");
return 1;
   }
    /* Create input file descriptor */
   fd1 = open (argv [1], O_RDONLY);
if (fd1 == -1) {
perror ("open");
return 2;
   }
    /* Create output file descriptor */
   fd2 = open(argv[2], O_WRONLY | O_CREAT, 0644);
if(fd2 == -1){
perror("open");
return 3;
   }
    /* Copy process */
while((ret_in = read (fd1, &buffer, BUF_SIZE)) > 0)
     {
write (fd2, &buffer, (ssize_t)ret_in);
```

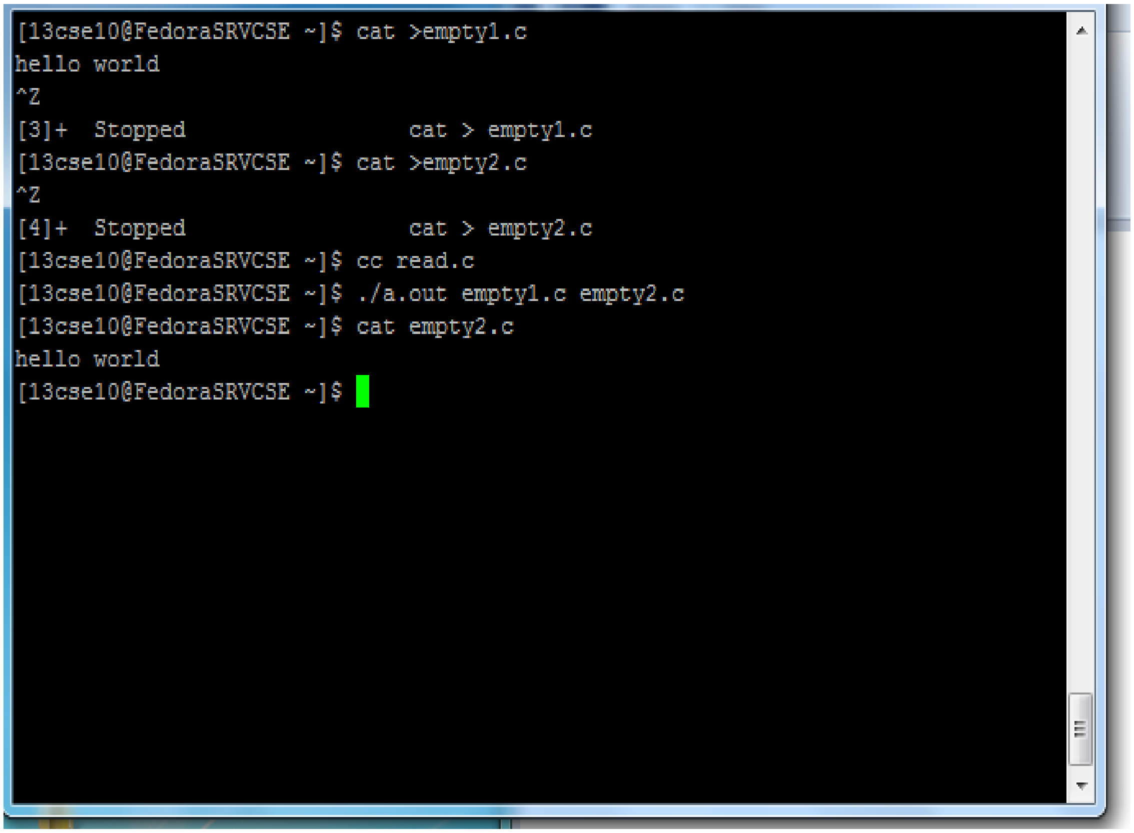
```
            }
close (fd1);
close (fd2);
 }
```

OUTPUT:

Exp No:

Regd.No: □□□□□□□□□□

Date :



```
[13cse10@FedoraSRVCSE ~]$ cat >empty1.c
hello world
^Z
[3]+  Stopped                 cat > empty1.c
[13cse10@FedoraSRVCSE ~]$ cat >empty2.c
^Z
[4]+  Stopped                 cat > empty2.c
[13cse10@FedoraSRVCSE ~]$ cc read.c
[13cse10@FedoraSRVCSE ~]$ ./a.out empty1.c empty2.c
[13cse10@FedoraSRVCSE ~]$ cat empty2.c
hello world
[13cse10@FedoraSRVCSE ~]$
```

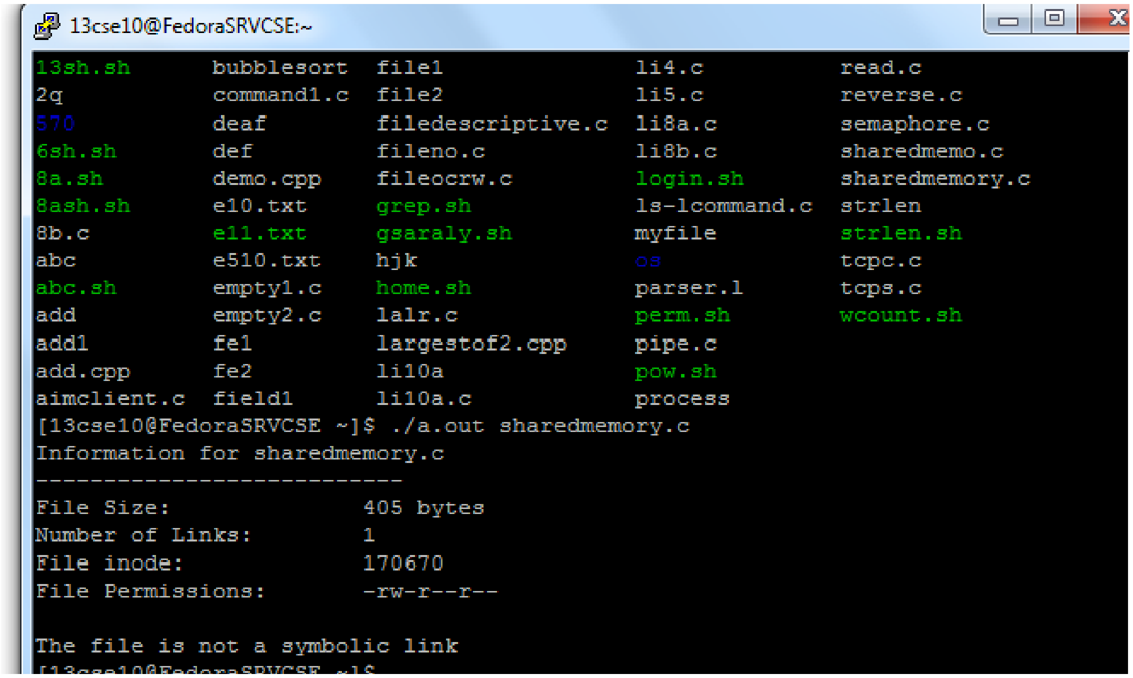
3. Write a C program to emulate the UNIX ls –l command

Description: The **ls** command lists the files and folders in the current directory. –l option is used for long listing format.For each file named, `ls` lists information for that file.

Program:

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

int main(int argc, char **argv)
{
if(argc != 2)
return 1;

struct stat fileStat;
if(stat(argv[1],&fileStat) < 0)
return 1;

printf("Information for %s\n",argv[1]);
printf("---------------------------\n");
printf("File Size: \t\t%ld bytes\n",fileStat.st_size);
printf("Number of Links: \t%ld\n",fileStat.st_nlink);
printf("File inode: \t\t%ld\n",fileStat.st_ino);

printf("File Permissions: \t");
printf( (S_ISDIR(fileStat.st_mode)) ? "d" : "-");
printf( (fileStat.st_mode & S_IRUSR) ? "r" : "-");
printf( (fileStat.st_mode & S_IWUSR) ? "w" : "-");
printf( (fileStat.st_mode & S_IXUSR) ? "x" : "-");
printf( (fileStat.st_mode & S_IRGRP) ? "r" : "-");
printf( (fileStat.st_mode & S_IWGRP) ? "w" : "-");
printf( (fileStat.st_mode & S_IXGRP) ? "x" : "-");
printf( (fileStat.st_mode & S_IROTH) ? "r" : "-");
printf( (fileStat.st_mode & S_IWOTH) ? "w" : "-");
printf( (fileStat.st_mode & S_IXOTH) ? "x" : "-");
printf("\n\n");

printf("The file %s a symbolic link\n", (S_ISLNK(fileStat.st_mode)) ? "is" : "is not");
return 0;
}
```

OUTPUT:



```
13sh.sh      bubblesort  file1              li4.c          read.c
2q           command1.c  file2              li5.c          reverse.c
570          deaf        filedescriptive.c  li8a.c         semaphore.c
6sh.sh       def         fileno.c           li8b.c         sharedmemo.c
8a.sh        demo.cpp    fileocrw.c         login.sh       sharedmemory.c
8ash.sh      e10.txt     grep.sh            ls-lcommand.c  strlen
8b.c         e11.txt     gsaraly.sh         myfile         strlen.sh
abc          e510.txt    hjk                os             tcpc.c
abc.sh       empty1.c    home.sh            parser.l       tcps.c
add          empty2.c    lalr.c             perm.sh        wcount.sh
add1         fe1         largestof2.cpp     pipe.c
add.cpp      fe2         li10a              pow.sh
aimclient.c  field1      li10a.c            process
[13cse10@FedoraSRVCSE ~]$ ./a.out sharedmemory.c
Information for sharedmemory.c
---------------------------
File Size:              405 bytes
Number of Links:        1
File inode:             170670
File Permissions:       -rw-r--r--

The file is not a symbolic link
[13cse10@FedoraSRVCSE ~]$
```

Exp No:

Regd.No: □□□□□□□□□□

Date :

4. Write a C program that illustrates how to execute two commands concurrently with a command pipe.

Ex:- ls –l | sort

Description:A pipe is a technique for passing information from one program process to another. Unlike other forms of inter process communication (IPC), a pipe is one-way communication only. Basically, a pipe passes a parameter such as the output of one process to another process which accepts it as input. The system temporarily holds the piped information until it is read by the receiving process.

Program:

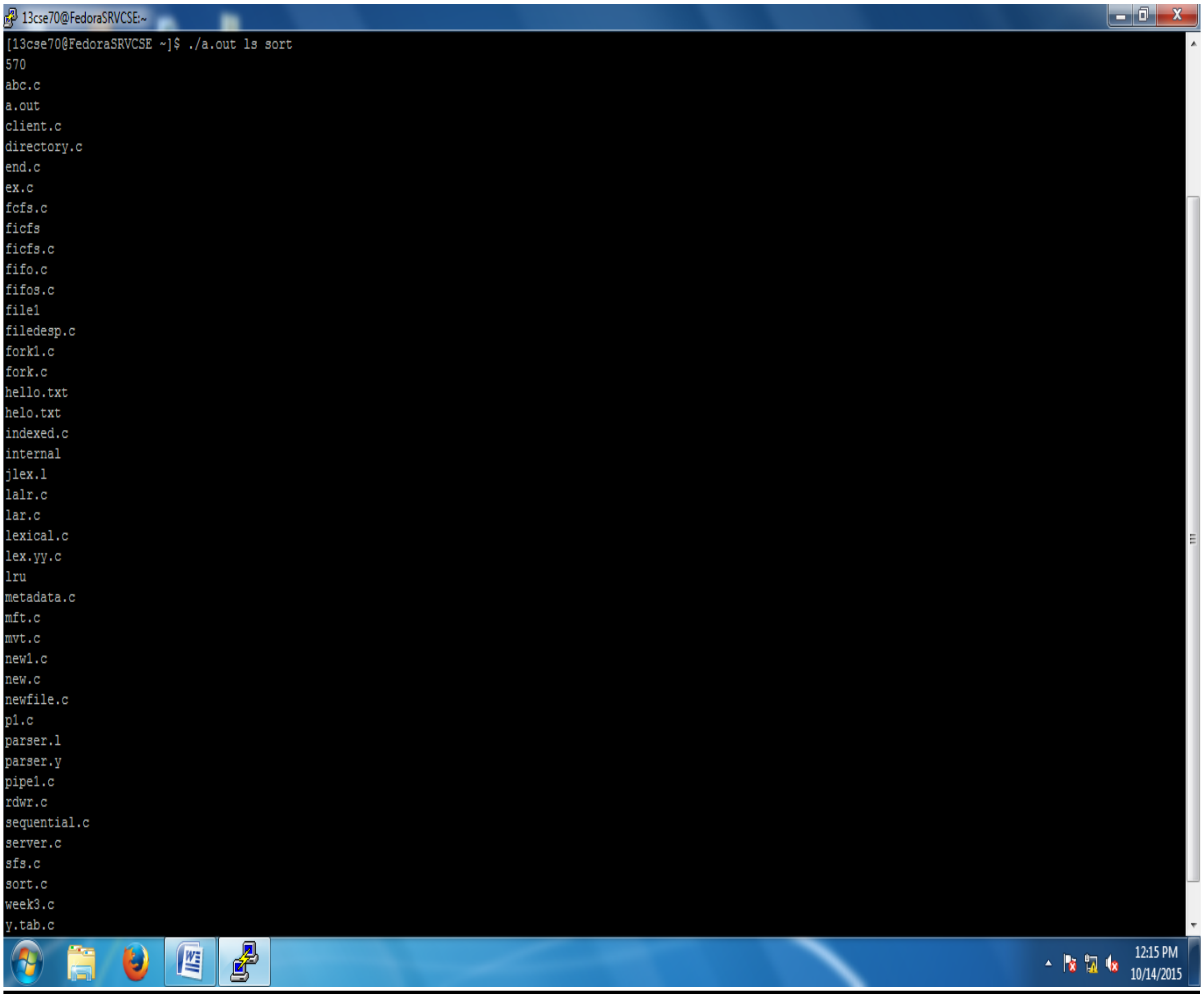
```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
//#include<sys/wait.h>
int main(int argc,char *argv[])
{
int fd[2],pid,k;
 k=pipe(fd);
if(k==-1)
 {
perror("pipe");
exit(1);
 }
pid=fork();
if(pid==0)
 {
close(fd[0]);
dup2(fd[1],1);
close(fd[1]);
execlp(argv[1],argv[1],NULL);
perror("execl");
 }
else
 {
wait(2);
close(fd[1]);
dup2(fd[0],0);
close(fd[0]);
execlp(argv[2],argv[2],NULL);
perror("execl");
 }
}
```

Exp No:

Regd.No: □□□□□□□□□□

Date :

OUTPUT:



```
[13cse70@FedoraSRVCSE ~]$ ./a.out ls sort
570
abc.c
a.out
client.c
directory.c
end.c
ex.c
fcfs.c
ficfs
ficfs.c
fifo.c
fifos.c
file1
filedesp.c
fork1.c
fork.c
hello.txt
helo.txt
indexed.c
internal
jlex.l
lalr.c
lar.c
lexical.c
lex.yy.c
lru
metadata.c
mft.c
mvt.c
new1.c
new.c
newfile.c
p1.c
parser.l
parser.y
pipe1.c
rdwr.c
sequential.c
server.c
sfs.c
sort.c
week3.c
y.tab.c
```

5) Simulate the following CPU scheduling algorithm

a)FCFS Scheduling

Description

Simplest CPU-scheduling algorithm is the first come, first-served (FCFS) scheduling algorithm. The process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue. When a process enters the ready queue, it is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue.

Program:

```
#include<stdio.h>
//#include<conio.h>
void main()
{
char  pn[10][10];
int  arr[10],bur[10],star[10],finish[10],tat[10],wt[10],i,n;
int  totwt=0,tottat=0;

printf("Enter the number of processes:");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("Enter the Process Name, Arrival Time & Burst Time:");
scanf("%s%d%d",&pn[i],&arr[i],&bur[i]);
}
for(i=0;i<n;i++)
{
if(i==0)
{
star[i]=arr[i];
wt[i]=star[i]-arr[i];
finish[i]=star[i]+bur[i];
tat[i]=finish[i]-arr[i];
}
else
{
star[i]=finish[i-1];
wt[i]=star[i]-arr[i];
finish[i]=star[i]+bur[i];
tat[i]=finish[i]-arr[i];
}
}
printf("\n PName Arrtime Burtime    Start     TAT   Finish");
for(i=0;i<n;i++)
{
printf("\n %s \t %6d \t\t %6d \t %6d \t %6d \t %6d",pn[i],arr[i],bur[i],star[i],tat[i],finish[i]);
totwt+=wt[i];
tottat+=tat[i];
}
printf("\n Average Waiting time:%f",(float)totwt/n);
printf("\n Average Turn Around Time:%f",(float)tottat/n);
}
```

OUTPUT:

```
Enter the number of processes:3
Enter the Process Number, Arrival Time & Burst Time:0
0
3
Enter the Process Name, Arrival Time & Burst Time:1
1
3
Enter the Process Name, Arrival Time & Burst Time:2
2
4

pnoArrtimeBurtime    Start  WT   TAT   Finish
0             0              3  0 0    3       3
11              3       3       2     5       6
2             2              4       6       4     8       10

Average Waiting time:2.000000
Average Turn Around Time:5.333333
```

Exp No:

Regd.No: □□□□□□□□□□

Date :

5) Simulate the following CPU scheduling algorithm
b)Shortest Job First Scheduling

Description:-

A different approach to CPU scheduling is the shortest-job first (SJF) scheduling algorithm. This algorithm associates with each process the length of the next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. if two processes have the same length next CPU burst, FCFS scheduling is used to break the tie.

Program

```
#include<stdio.h>
#include<string.h>
void main()
{
int  pn[10],et[20],at[10],n,i,j,temp,st[10],ft[10],wt[10],ta[10];
int  totwt=0,totta=0;
float  awt,ata;
printf("Enter the number of process:");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("Enter process number, arrival time & execution time:");
scanf("%d%d%d",&pn[i],&at[i],&et[i]);
}
for(i=0;i<n;i++)
for(j=0;j<n;j++)
{
if(et[i]<et[j])
{
temp=at[i];
at[i]=at[j];
at[j]=temp;
temp=et[i];
et[i]=et[j];
et[j]=temp;
temp=pn[i];
pn[i]=pn[j];
pn[j]=temp;
}
}
for(i=0;i<n;i++)
{
if(i==0)
{
st[i]=at[i];
wt[i]=st[i]-at[i];
ft[i]=st[i]+et[i];
ta[i]=ft[i]-at[i];
}
else
{
st[i]=ft[i-1];
wt[i]=st[i]-at[i];
ft[i]=st[i]+et[i];
ta[i]=ft[i]-at[i];
}
}
for(i=0;i<n;i++)
{
totwt+=wt[i];
totta+=ta[i];
}

awt=(float)totwt/n;
ata=(float)totta/n;
printf("\nPname\tarrivaltime\texecutiontime\twaitingtime\ttatime");
for(i=0;i<n;i++)
printf("\n%d\t%5d\t\t%5d\t\t%5d\t\t%5d",pn[i],at[i],et[i],wt[i],ta[i]);
printf("\n Average waiting time is:%f",awt);
printf("\n Average turn around time is:%f",ata);
}
```

OUTPUT:

```
Enter the number of process:3
Enter process number, arrival time & execution time:1 0 5
Enter process number, arrival time & execution time:2 0 4
Enter process number, arrival time & execution time:3 0 3

Pno Arrivaltime Starttime Executiontime Waitingtime Tatime
  3       0          0          3           0           3
  2       0          3          4           3           7
  1       0          7          5           7          12

Average waiting time is:3.333333
Average turnaroundtime is:7.333333
```

5) Simulate the following CPU scheduling algorithm

c) Priority Scheduling

Description:

The SJF algorithm is a special case of the general priority-scheduling algorithm. A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order. Priorities are generally some fixed range of numbers such as 0 to 7, 0 to 4095.

Program:

```
#include<stdio.h>
//#include<conio.h>
#include<string.h>
void main()
{
int pn[10],et[20],at[10],n,i,j,temp,p[10],st[10],ft[10],wt[10],ta[10];
int totwt=0,totta=0;
float awt,ata;
//clrscr();
printf("Enter the number of process:");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("Enter process number,arrivaltime,execution time & priority:");
//flushall();
scanf("%d%d%d%d",&pn[i],&at[i],&et[i],&p[i]);
}
for(i=0;i<n;i++)
for(j=0;j<n;j++)
{
if(p[i]<p[j])
{
temp=p[i];
p[i]=p[j];
p[j]=temp;
temp=at[i];
at[i]=at[j];
at[j]=temp;
temp=et[i];
et[i]=et[j];
et[j]=temp;
temp=pn[i];
pn[i]=pn[j];
pn[j]=temp;
}
}
for(i=0;i<n;i++)
{
if(i==0)
{
st[i]=at[i];
wt[i]=st[i]-at[i];
ft[i]=st[i]+et[i];
ta[i]=ft[i]-at[i];
}
else
{
st[i]=ft[i-1];
wt[i]=st[i]-at[i];
ft[i]=st[i]+et[i];
ta[i]=ft[i]-at[i];
}
totwt+=wt[i];
totta+=ta[i];
}
awt=(float)totwt/n;
ata=(float)totta/n;
printf("\nPno\tarrivaltime\texecutiontime\tpriority\twaitingtime\ttatime");
for(i=0;i<n;i++)
printf("\n%d\t%5d\t\t%5d\t\t%5d\t\t%5d\t\t%5d",pn[i],at[i],et[i],p[i],wt[i],ta[i]);
printf("\nAverage waiting time is:%f",awt);
printf("\nAverageturnaroundtime is:%f",ata);
}
```

OUTPUT:

```
Enter the number of process:3
Enter process number,arrivaltime,execution time & priority:1 2 3 2
Enter process number,arrivaltime,execution time & priority:2 1 3 1
Enter process number,arrivaltime,execution time & priority:3 0 2 0
Pno arrivaltime  executiontime   priority  waitingtime tatime
   3         0             2            0            0                2
  2         1            3            1            1                4
  1         2            3            2            3                6
Average waiting time is:1.333333
Average turnaroundtime is:4.000000
```

Exp No:

Regd.No: □□□□□□□□□□

Date :

5) Simulate the following CPU scheduling algorithm
d)Round Robin Scheduling

DESCRIPTION:-

The round-robin scheduling algorithm is designed especially for time-sharing systems. It is similar to FCFS scheduling, but preemption is added to switch between processes. A small unit of time called a time quantum is defined. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.

To implement RR scheduling we keep the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum and dispatches the process.

Program:

```
#include<stdio.h>
int main()
{
int i,j,n,tq,time=0,p[10],st[10],b[10],str[10],ft[10],tat[10],wt[10];
float avg1=0,avg2=0;
printf("enter the number of processes and time quantum");
scanf("%d%d",&n,&tq);
printf("enter the process number and service time");
for(i=0;i<n;i++)
{
scanf("%d%d",&p[i],&st[i]);
}
for(i=0;i<n;i++)
{
b[i]=st[i];
}
str[0]=0;
for(i=1;i<n;i++)
{
str[i]=str[i-1]+tq;
}
for(i=0;i<n;i++)
{
while(b[i]!=0)
{
for(j=0;j<n;j++)
{
if(b[j]==0)
{
```

```
continue;
}
if(b[j]>tq)
{
b[j]=b[j]-tq;
time=time+tq;
}
else
{
time=time+b[j];
b[j]=0;
ft[j]=time;
}
}
}
}
for(i=0;i<n;i++)
{
tat[i]=ft[i];
wt[i]=ft[i]-st[i];
avg1=avg1+tat[i];
avg2=avg2+wt[i];
}
printf("\n pnost\t    ft \t tat\t  wt");
for(i=0;i<n;i++)
{
printf("\n %d\t  %d\t   %d\t  %d\t  %d\t ",p[i],st[i],ft[i],tat[i],wt[i]);
}
printf("\n average tat is %f\n average wt is %f",(avg1/n),(avg2/n));
}
```

OUTPUT:

```
enter the number of processes and time quantum3 4
enter the process number and service time1 24
2  3
3  3

Pno     st      ft      tat     wt
 1      24      30      30      6
 2       3       7       7      4
 3       3      10      10      7

Averagetat is 15.666667
Averagewt is 5.666667
```

6.Implementation of Fork(), Wait(), Exec() and Exit() System calls.

Description: A fork system call creates a child process that is a clone of the parent. The child process may execute a different program in its context with a separate exec() system call.Parent may want to wait for children to finish by calling wait function.The wait() function suspends execution of its calling process untilstatus information is available for a terminated child process, or a signal is received.The exit() function is used to terminate the calling process immediately.

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h> /* for fork */
#include <sys/types.h> /* for pid_t */
#include <sys/wait.h> /* for wait */
int main(int argc,char** argv)
{
/*Spawn a child to run the program.*/
pid_t pid=fork();
if (pid==0)
{
/* child process */
execv("/bin/ls",argv);
exit(127); /* only if execv fails */
}
else
{
/*
pid!=0; parent process */
printf("\nWaiting for the child process to finish");
//waitpid(pid,0,0); /* wait for child to exit */
wait(NULL);
}
printf("\nExiting main process\n");
return 0;
}
```

Output:

```
a.out    file1.c  file2.c  commandpipe.c
fcfs.c   sjf.c    rr.c     priority.c
Waiting for the child process to finish
Exiting main process.
```

Exp No:

Regd.No: □□□□□□□□□□

Date :

7.a) Simulate MFT(Multiprogramming with Fixed number of Tasks)

Description: We can assume that the operating system occupies some fixed portion of main memory and that the rest of main memory is available for use by multiple processes. The simplest scheme for managing this available memory is to partition it into regions with fixed boundaries. Main memory is divided into a number of static partitions at system generation time. A process may be loaded into a partition of equal or greater size.

Main memory utilization is extremely inefficient. There is wasted space internal to a partition due to fact that the block of data loaded is smaller than the partition, is referred to as internal fragmentation.

Program:

```
#include<stdio.h>
intmain()
{
int pr[50],size[50],nba[50],ifm[50];
int m,b,bs,n,i,j,count=0;
printf("enter total memory size: ");
scanf("%d",&m);
printf("enter number of blocks: ");
scanf("%d",&b);
bs=m/b;
printf("block size is: %d\n",bs);
printf("enter number of processes: ");
scanf("%d",&n);
for(i=0;i<n;i++)
  {
    pr[i]=i+1;
    printf("enter process %d size: ",i+1);
    scanf("%d",&size[i]);
    if(size[i]>m)
    {
      count=1;
      break;
    }
nba[i]=size[i]/bs;
  if((size[i]%bs)!=0)
      nba[i]=nba[i]+1;
    ifm[i]=(nba[i]*bs)-size[i];
    m=m-size[i];
  }
  printf("\nProcess \t Size \t No.of-blocksalloted \t internal-fragementation\n");
  for(j=0;j<i;j++)
  {
    printf("p%d \t\t %d \t\t %d \t\t\t %d\n",pr[j],size[j],nba[j],ifm[j]);
  }
  if(count==1)
    printf("insufficient memory for remaining processes");
}
```

OUTPUT :

```
enter total memory size: 800
enter number of blocks: 8
block size is: 100
enter number of processes: 4
enter process 1 size: 100
enter process 2 size: 250
enter process 3 size: 325
enter process 4 size: 50

Process          Size     No.of-blocks alloted   internal-fragementation
p1               100           1                        0
p2               250           3                        50
p3               325           4                        75
p4               50            1                        50
```

7. b) Simulate MVT (Multiprogramming with Variable number of Tasks)

Description: An important operating system that used this technique was IBM’s mainframe operating system. With dynamic partitioning, the partitions are of variable length and number. When a process is brought into main memory, it is allocated exactly as much memory as it requires and no more. Partitions are created dynamically, so that each process is loaded into a partition of exactly the same size as that process. It leads to a situation in which there are a lot of small holes in memory. This phenomenon is referred to as external fragmentation, referring to the fact that the memory that is external to all partitions becomes increasingly fragmented. One technique for overcoming external fragmentation is compaction.

Program:

```
#include<stdio.h>
int main()
{
int  pr[50],size[50],rm[50];
int  m,b,bs,n,i,j,count=0;
printf("enter total memory size: ");
scanf("%d",&m);
printf("enter number of blocks: ");
scanf("%d",&b);
bs=m/b;
printf("block size is: %d\n",bs);
printf("enter number of processes: ");
scanf("%d",&n);
for(i=0;i<n;i++)
  {
    pr[i]=i+1;
    printf("enter process%d size: ",i+1);
    scanf("%d",&size[i]);
    if(size[i]>m)
    {
      count=1;
      break;
    }
    m=m-size[i];
    rm[i]=m;
  }
printf("\n pr \t size \t rm \n");
  for(j=0;j<i;j++)
    printf("p%d\t%d\t%d\n",pr[j],size[j],rm[j]);
  if(count==1)
    printf("insufficient memory for remaining processes\n");
printf("External fragmentation is : %d\n",m);
}
```

OUTPUT :-

```
enter total memory size: 800
enter number of blocks: 8
block size is: 100
enter number of processes: 3
enter process1 size: 50
enter process2 size: 150
enter process3 size: 200

pr      size    rm
p1      50      750
p2      150     600
p3      200     400
External fragmentation is : 400
```

8.Simulate Bankers Algorithm for Dead Lock Avoidance.

Description:-

Deadlock Definition

A set of processes is deadlocked if each process in the set is waiting for an eventthat only another process in the set can cause (including itself).Waiting for an event could be:

Waiting for access to a critical section

1. Waiting for a resource Note that it is usually a non-preemptable (resource).Preemptable resources can be yanked away and given to another.

Conditions for Deadlock

1. Mutual exclusion: resources cannot be shared.
2. Hold and wait:processes request resources incrementally, and hold on to whatthey've got.
3. No preemption: resources cannot be forcibly taken from processes.
4. Circular wait: circular chain of waiting, in which each process is waiting for aresource held by the next process in the chain.

Strategies for dealing with Deadlock

1. Ignore the problem altogether
2. Detection and recovery
3. Avoidance by careful resource allocation
4. Prevention by structurally negating one of the four necessary conditions.

Deadlock Avoidance

Avoid actions that may lead to a deadlock. Think of it as a state machine movingfrom one state to another as each instruction is executed.

Safe State:

Safe state is one where

1. It is not a deadlocked state
2. There is some sequence by which all requests can be satisfied.

To avoid deadlocks, we try to make only those transitions that will take you from onesafe state to another. We avoid transitions to unsafe state (a state that is notdeadlocked, and is not safe)

eg.Total # of instances of resource = 12(Max, Allocated, Still Needs)

P0 (10, 5, 5) P1 (4, 2, 2) P2 (9, 2, 7)

Free = 3 - Safe

The sequence is a reducible sequencethe first state is safe.

What if P2 requests 1 more and is allocated 1 more instance?

- results in Unsafe state

So do not allow P2's request to be satisfied.

Program:

```
#include<stdio.h>
//#include<conio.h>
 int C[4][3],A[4][3],RQ[4][3],V[3],R[3],K[4],sum=0,np,nr;
int main()
  {
void fun();
int i,j,count=0,pcount=0;
  //clrscr();
printf("\nEnter the total number of resources : ");
scanf("\n%d",&nr);
for(i=0;i<nr;i++)
   {
printf("\nEnter the no of resources int R%d : ",i+1);
scanf("%d",&R[i]);
   }
printf("\nEnter the no of processes to be executed : ");
scanf("%d",&np);
printf("\nEnter the claim matrix:\n");
for(i=0;i<np;i++)
for(j=0;j<nr;j++)
scanf("%d",&C[i][j]);
printf("\nEnter the allocation matrix:\n");
for(i=0;i<np;i++)
for(j=0;j<nr;j++)
scanf("%d",&A[i][j]);
for(i=0;i<np;i++)
for(j=0;j<nr;j++)
  RQ[i][j] = C[i][j] - A[i][j];
 /* FINDING THE REQUIRED  RESOURCES MATRIX(i.e., C-A) */
fun();
for(i=0;i<np;i++)
  {
count=0;
if(K[i] == i+1)
continue;      /* FOR SKIPPING THE PROCESSES(i.e., ROWS) WHICH
       WERE ALREADY EXECUTED */
for(j=0;j<nr;j++)
  {
if(V[j] >= RQ[i][j])
count++;
    }
if(count == nr)
  {
   K[i] = i+1;
for(j=0;j<nr;j++)
    C[i][j] = A[i][j] = RQ[i][j] = 0;
  pcount++;
  count = 0;
  i=-1;
  fun();
       }
     }

if(pcount == np)
printf("\nThere is no chance of deadlock.\nIt is a safe state.");
else
printf("\nThere is a chance of deadlock.\nIt isn't a safe state.");
   //getch();
   }

void fun()
   {
int i1,j1;
for(i1=0;i1<nr;i1++)
   {
for(j1=0;j1<np;j1++)
     {
sum = sum + A[j1][i1];
      }
V[i1] = R[i1] - sum;
sum = 0;
   }
  }
```

OUTPUT:

```
Enter the no of processes : 5
Enter the no of resources : 3
Enter the Max Matrix for each process :
For process 1 : 753
For process 2 : 322
For process 3 : 702
For process 4 : 222
For process 5 : 433
Enter the allocation for each process :
For process 1 : 010
For process 2 : 200
For process 3 : 302
For process 4 : 211
For process 5 : 002
Enter the Available Resources : 332
Max matrix:    Allocation matrix:
7 5 3          0 1 0
3 2 2          2 0 0
7 0 2          3 0 2
2 2 2          2 1 1
4 3 3          0 0 2
Process 2 runs to completion!
Max matrix:    Allocation matrix:
    7 5 3          0 1 0
    0 0 0          0 0 0
    7 0 2          3 0 2
    2 2 2          2 1 1
    4 3 3          0 0 2
Process 3 runs to completion!
Max matrix:    Allocation matrix:
    7 5 3          0 1 0
    0 0 0          0 0 0
    0 0 0          0 0 0
    2 2 2          2 1 1
    4 3 3          0 0 2
Process 4 runs to completion!
Max matrix:    Allocation matrix:
    7 5 3          0 1 0
    0 0 0          0 0 0
    0 0 0          0 0 0
    0 0 0          0 0 0
    4 3 3          0 0 2
Process 5 runs to completion!
The system is in a safe state!!Safe Sequence :< 2 3 4 1 5 >
```

Exp No:

Regd.No: □□□□□□□□□□

Date :

9. Simulate the following page replacement algorithm

a)FIFO Page Replacement

Description:A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue.Consider the following reference string - 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1.

```
#include<stdio.h>
int main()
{
int i,j,n,a[50],frame[10],no,k,avail,count=0;
printf("\n ENTER THE NUMBER OF PAGES:\n");
scanf("%d",&n);
printf("\n ENTER THE PAGE NUMBERS :\n");
for(i=1;i<=n;i++)
scanf("%d",&a[i]);
printf("\n ENTER THE NUMBER OF FRAMES :");
scanf("%d",&no);
for(i=0;i<no;i++)
frame[i]= -1;
  j=0;
  printf("ref string\t page frames\n");
for(i=1;i<=n;i++)
                        {
printf("%d\t\t",a[i]);
avail=0;
for(k=0;k<no;k++)
if(frame[k]==a[i])
avail=1;
if (avail==0)
{
frame[j]=a[i];
      j=(j+1)%no;
count++;
for(k=0;k<no;k++)
printf("%d\t",frame[k]);
}
printf("\n");
}
printf("No. of Page Faults %d", count);
}
```

```
#include<stdio.h>
int main()
{
int i,j,n,a[50],frame[10],no,k,avail,count=0;
printf("\n ENTER THE NUMBER OF PAGES:\n");
scanf("%d",&n);
printf("\n ENTER THE PAGE NUMBERS :\n");
for(i=1;i<=n;i++)
scanf("%d",&a[i]);
printf("\n ENTER THE NUMBER OF FRAMES :");
scanf("%d",&no);
for(i=0;i<no;i++)
frame[i]= -1;
  j=0;
  printf("ref string\t page frames\n");
for(i=1;i<=n;i++)
                        {
printf("%d\t\t",a[i]);
avail=0;
for(k=0;k<no;k++)
if(frame[k]==a[i])
avail=1;
if (avail==0)
{
frame[j]=a[i];
      j=(j+1)%no;
count++;
for(k=0;k<no;k++)
printf("%d\t",frame[k]);
}
printf("\n");
}
printf("No. of Page Faults %d", count);
}
```

OUTPUT:

```
ENTER THE NUMBER OF PAGES:  20
ENTER THE PAGE NUMBERS:
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
 ENTER THE NUMBER OF FRAMES: 3
ref string       page frames
7              7      -1      -1
0              7      0       -1
1              7      0       1
2              2      0       1
0
3              2      3       1
0              2      3       0
4              4      3       0
2              4      2       0
3              4      2       3
0              0      2       3
3
2
1              0      1       3
2              0      1       2
0
1
7              7      1       2
0              7      0       2
1              7      0       1

No.of Page Faults 15
```

9. Simulate the following page replacement algorithm

b) LRU Page Replacement

Description: If we will replace the page that has not been used for the longest period of time,the approach is the least recently used algorithm. By the principle of locality, this should be the page least likely to be referenced in the near future. Consider the following reference string - 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1.

Program:

```
#include<stdio.h>
//#include<conio.h>
int fr[3];
main()
{
void display();
int index,k,l,flag1=0,flag2=0,pf=0,frsize,i,j,fs[3],n,p[20];
printf("\nenter length of string\n");
scanf("%d",&n);
printf("\n  enter ref string\n");
for(i=0;i<n;i++)
scanf("%d",&p[i]);
printf("\n  enter frame size");
scanf("%d",&frsize);
for(i=0;i<frsize;i++)
{
fr[i]=-1;
}
for(j=0;j<n;j++)
{
flag1=0,flag2=0;
for(i=0;i<frsize;i++)
{
if(fr[i]==p[j])
{
flag1=1;
flag2=1;
break;
}
}
if(flag1==0)
{
for(i=0;i<frsize;i++)
{
if(fr[i]==-1)
{	pf++;
fr[i]=p[j];
flag2=1;
break;
}
}
}
if(flag2==0)
{
for(i=0;i<frsize;i++)
fs[i]=0;
for(k=j-1,l=1;l<=frsize-1;l++,k--)
{
for(i=0;i<frsize;i++)
{
if(fr[i]==p[k])
fs[i]=1;
}
}
for(i=0;i<frsize;i++)
{
if(fs[i]==0)
index=i;
}
fr[index]=p[j];
pf++;
}
display();
}
printf("\n no of page faults :%d",pf);

}
void display()
{
int i;
printf("\n");
for(i=0;i<3;i++)
printf("\t%d",fr[i]);
}
```

OUTPUT:

```
enter length of string 20
enter ref string
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
enter frame size 3
        7       -1      -1
        7       0       -1
        7       0       1
        2       0       1
        2       0       1
        2       0       3
        2       0       3
        4       0       3
        4       0       2
        4       3       2
        0       3       2
        0       3       2
        0       3       2
        1       3       2
        1       3       2
        1       0       2
        1       0       2
        1       0       7
        1       0       7
        1       0       7

no of page faults :12
```

9. Simulate the following page replacement algorithm

c) LFU Page Replacement

Description: The least frequently used page (LFU). Since LFU tracks the usage count of pages, it will take longer to replace pages with high count values, even if those pages are no longer in use. Consider the following reference string - 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1.

Program:

```
#include<string.h>
#include<stdio.h>
int i,j=1,s,k,l,re[30],p[10],ch,no,nr,c,a1=0,a,line=6,nk;
int display();
struct
{
 int st,l,ps,pos;
}opr;
int main()
{
 //clrscr();
 printf("Enter length of reference string:");
 scanf("%d",&nr);
 printf(" Enter reference string:");
 for(i=1;i<=nr;i++)
  scanf("%d",&re[i]);
 printf("\n Enter number of frames:");
 scanf("%d",&no);
 //clrscr();
 for(i=1;i<=no;i++)
  p[i]=-1;
 opr.st=100;
 for(i=1;i<=nr;i++)
 {
  a1=0;
  opr.st=100;
  opr.pos=100;
  for(c=1;c<=no;c++)
   if(re[i]==p[c])
    a1++;
   if(a1==0)
   {
    if(j<=no)
    {
     p[j]=re[i];
     j++;
    }
    else
    {
    for(k=1;k<=no;k++)
    {
     a=0;
     for(ch=i-1;ch>0;ch--)
     {
      if(p[k]==re[ch])
      {
        a++;
        nk=ch;
      }
     }
     if(a>1)
     {
      if(opr.st>a)
      {
       opr.st=a;
       opr.ps=k;
      }
      else
      if(opr.st==a)
      {
       if(opr.pos>ch)
        opr.ps=k;
      }
     }
     else
     if(a==1)
     {
      if(opr.pos>nk)
      {
       opr.pos=nk;
       opr.ps=k;
       opr.st=a;
      }
     }
    }
    p[(opr.ps)]=re[i];
   }
  }
  display(no,p,i);
 }
 printf("\n");
```

```
}
int display(int no,int p[],int i)
{
 int k;
 printf("\n%d",re[i]);
 //gotoxy(25,line++);
 for(k=1;k<=no;k++)
 {
  printf("  ");
  if(p[k]!=-1)
   printf("%d",p[k]);
  else
   printf(" ");
 }
}
```

OUTPUT:

```
Enter length of string  20
Enter ref string
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

        7      -1      -1
        7       0      -1
        7       0       1
        2       0       1
        2       0       1
        2       0       3
        2       0       3
        4       0       3
        4       0       2
        4       3       2
        0       3       2
        0       3       2
        0       3       2
        1       3       2
        1       3       2
        1       0       2
        1       0       2
        1       0       7
        1       0       7
        1       0       7

no of page faults :12
```

Exp No:

Regd.No: □□□□□□□□□□

Date :

10. Simulate the following file allocation strategy

a) Sequential file:

Description: The contiguous-allocation method requires each file to occupy a set of contiguous blocks on the disk. Disk addresses define a linear ordering on the disk. Accessing a file that has been allocated contiguously is easy. For sequential access, the file system remembers the disk address of the last block referenced and when necessary reads the next block.

Program:

```
#include<stdio.h>
#include<string.h>
int main()
{
int i,j,n,size[50],sblock[20],eblock[20];
char name[50];
printf("Enter no of files");
scanf("%d",&n);
printf("Enter file name and size and startblock");
for(i=0;i<n;i++)
{
scanf(" %c %d %d",&name[i],&size[i],&sblock[i]);
}
for(i=0;i<n;i++)
{eblock[i]=sblock[i]+size[i];
}
printf("file allocation table\n");
printf("name size startblockendblock\n");
for(i=0;i<n;i++)
{
printf("%c \t%d\t",name[i],size[i]);

printf("%d \t\t%d",sblock[i],eblock[i]);

printf("\n");
}
}
```

OUTPUT:

```
Enter no of files3
Enter file name and size and startblocka 3 0
b 5 10
c 4 18
file allocation table
name size startblock endblock
a      3      0            3
b      5      10           15
c      4      18           22
```

Exp No:

Regd.No: □□□□□□□□□□

Date :

10. Simulate the following file allocation strategy

b) Indexed file:

Description:

Each file has its own index block, which is an array of disk-block addresses. The ith entry in the index block points to the ith block of the file. The directory contains the address of the index block. To read the ith block, we use the pointer in the ith index-block entry to find and read the desired block. When the file is created, all pointers in the index block are set to nil. When the ith block is first written, a block is obtained from the free-space manager, and its address is put in the ith index-block entry.

Program:

```
#include<stdio.h>
#include<string.h>
int main()
{
int i,j,n,size[50],block[20][20],ib;
char name[50];
printf("Enter no of files");
scanf("%d",&n);
printf("Enter file name and size");
for(i=0;i<n;i++)
{
scanf(" %c %d",&name[i],&size[i]);
}
printf("Enter Index block");
scanf("%d",&ib);
printf("enter index block sequence for each file");
for(i=0;i<n;i++)
{
block[i][0]=ib;
for(j=1;j<size[i];j++)
{
scanf("%d",&block[i][j]);
}
}
printf("file allocation table\n");
printf("name size start block\n");
for(i=0;i<n;i++)
{
printf("%c \t%d\t",name[i],size[i]);
for(j=0;j<size[i];j++)
{
printf("%d-",block[i][j]);
}
printf("\n");
}
}
```

OUTPUT:

```
      Enter no of files3
      Enter file name and size a 3
      b 3
      c 3
      Enter Index block5
      enter index block sequence for each file1 4
      2 6
      3 9
      file allocation table
      name size start block
      a     3     5-1-4
      b     3     5-2-6
      c     3     5-3-9
```

Exp No:

Regd.No: □□□□□□□□□□

Date :

10. Simulate the following file allocation strategy

c) Linked file

Description: With linked allocation, each file is a linked list of disk block; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file. In linked allocation, each directory entry has a pointer to the first disk block of the file. This pointer is initialized to nil to signify an empty file. A write to the file causes a free block to be found via the free space management system, and this new block is then written to, and is linked to the end of the file. To read a file, we simply read blocks by following the pointers from block to block. There is no external fragmentation with linked allocation.

Program:

```
#include<stdio.h>
#include<string.h>
int main()
{
int i,j,n,size[50],block[20][20];
char name[50];
printf("Enter no of files");
scanf("%d",&n);
printf("Enter file name and size");
for(i=0;i<n;i++)
{
scanf(" %c %d",&name[i],&size[i]);
}
printf("enter block sequence for each file");
for(i=0;i<n;i++)
{
for(j=0;j<size[i];j++)
{
scanf("%d",&block[i][j]);
}
}
printf("file allocation table\n");
printf("name size blocks\n");
for(i=0;i<n;i++)
{
printf("%c \t%d\t",name[i],size[i]);
for(j=0;j<size[i];j++)
{
printf("%d-",block[i][j]);
}
printf("\n");
}
}
```

OUTPUT:

```
Enter no of files3
Enter file name and size a 2
b 2
c 2
enter block sequence for each file 1 4
2 6
3 9
file allocation table
name size blocks
a       2       1-4
b       2       2-6
c       2       3-9
```

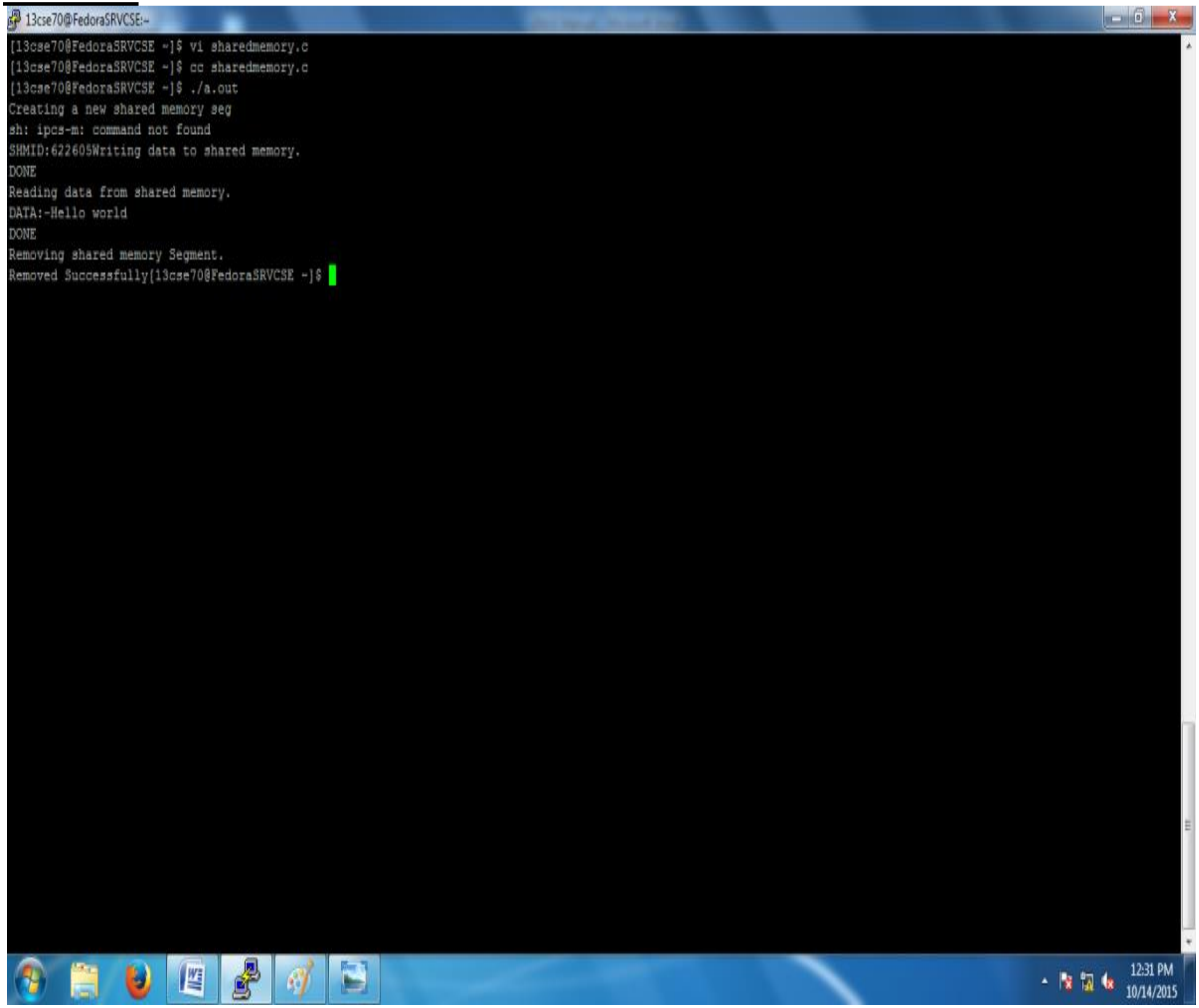
11. Write a C program that illustrates two processes communication using shared memory.

Description: shared memory is memory that may be simultaneously accessed by multiple programs with an intent to provide communication among them or avoid redundant copies. Shared memory is an efficient means of passing data between programs. Depending on context, programs may run on a single processor or on multiple separate processors.

Program:

```
#include <stdio.h>
#include<fcntl.h>
#include<string.h>
#include<stdlib.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<sys/types.h>
#define SEGSIZE 100
int main(int argc, char *argv[ ])
 {
int shmid,cntr;
key_t key;
char *segptr;
char buff[ ]= "Hello world";
key=ftok(".",'s');
if((shmid=shmget(key, SEGSIZE, IPC_CREAT | IPC_EXCL | 0666))== -1)
        {
if((shmid=shmget(key,SEGSIZE,0))== -1)
                {
perror("shmget");
exit(1);
                }
        }
else
        {
printf("Creating a new shared memory seg \n");
printf("SHMID:%d", shmid);
        }
system("ipcs –m");
if((segptr=shmat(shmid,0,0))==(char*)-1)
       {
perror("shmat");
exit(1);
       }
printf("Writing data to shared memory…\n");
strcpy(segptr,buff);
printf("DONE\n");
printf("Reading data from shared memory…\n");
printf("DATA:-%s\n",segptr);
printf("DONE\n");
printf("Removing shared memory Segment…\n");
if(shmctl(shmid,IPC_RMID,0)== -1)
printf("Can’t Remove Shared memory Segment…\n");
else
printf("Removed Successfully");
}
```

OUTPUT:



```
[13cse70@FedoraSRVCSE ~]$ vi sharedmemory.c
[13cse70@FedoraSRVCSE ~]$ cc sharedmemory.c
[13cse70@FedoraSRVCSE ~]$ ./a.out
Creating a new shared memory seg
sh: ipcs-m: command not found
SHMID:622605Writing data to shared memory.
DONE
Reading data from shared memory.
DATA:-Hello world
DONE
Removing shared memory Segment.
Removed Successfully[13cse70@FedoraSRVCSE ~]$
```

12. Write a C program to simulate producer and consumer problem using Semaphores.

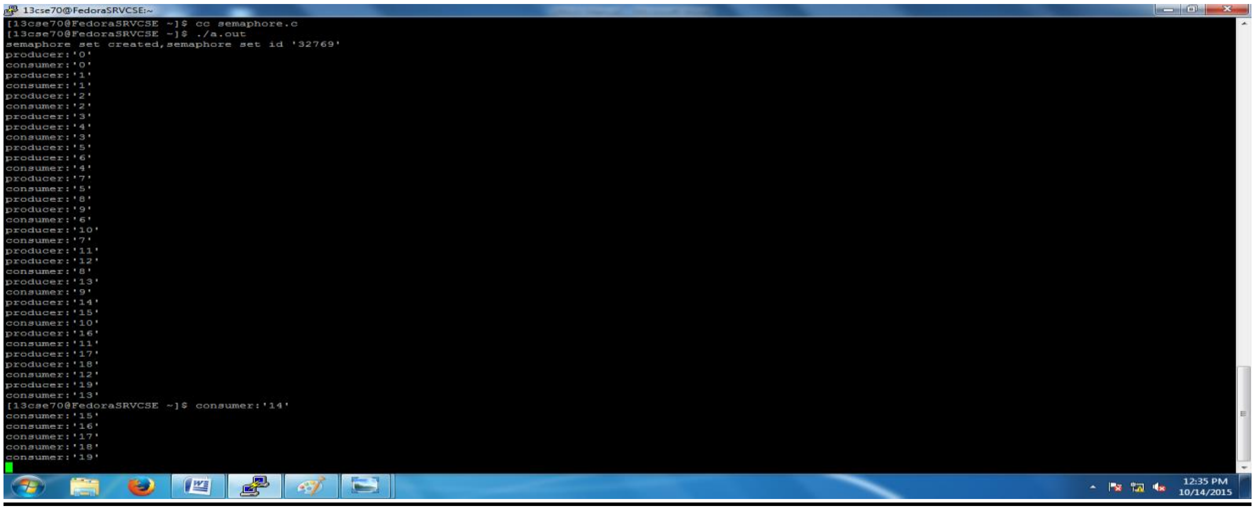
Description: a **semaphore** is a variable or abstract data type that is used for controlling access, by multiple processes, to a common resource in a concurrent system such as a multiprogramming operating system. Semaphores are a useful tool in the prevention of race conditions.Semaphores which allow an arbitrary resource count are called counting semaphores, while semaphores which are restricted to the values 0 and 1 (or locked/unlocked, unavailable/available) are called binary semaphores.

Program:

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<time.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/sem.h>
#define NUM_LOOPS 20
int main(int argc,char* argv[])
{
int sem_set_id;
int sem_val;
int child_pid;
int i;
struct sembuf sem_op;
int rc;
struct timespec delay;
sem_set_id=semget(IPC_PRIVATE,1,0600);
if(sem_set_id==-1)
{
perror("main:semget");
exit(1);
}
printf("semaphore set created,semaphore set id '%d'\n",sem_set_id);
sem_val=0;
rc=semctl(sem_set_id,0,SETVAL,sem_val);
child_pid=fork();
switch(child_pid)
{
case-1:
perror("fork");
exit(1);
case 0:
```

```
for(i=0;i<NUM_LOOPS;i++)
{
sem_op.sem_num=0;
sem_op.sem_op=-1;
sem_op.sem_flg=0;
semop(sem_set_id,& sem_op,1);
printf("consumer:'%d'\n",i);
fflush(stdout);
sleep(3);
}
break;
default:
for(i=0;i<NUM_LOOPS;i++)
{
printf("producer:'%d'\n",i);
fflush(stdout);
sem_op.sem_num=0;
sem_op.sem_op=1;
sem_op.sem_flg=0;
semop(sem_set_id,&sem_op,1);
sleep(2);
if(rand()>3*(RAND_MAX/4))
{
delay.tv_sec=0;
delay.tv_nsec=10;
nanosleep(&delay,NULL);
}
}
break;
}
return 0;
}
```

OUTPUT:



```
[13cse70@FedoraSRVCSE ~]$ cc semaphore.c
[13cse70@FedoraSRVCSE ~]$ ./a.out
semaphore set created,semaphore set id '32769'
producer:'0'
consumer:'0'
producer:'1'
consumer:'1'
producer:'2'
consumer:'2'
producer:'3'
producer:'4'
consumer:'3'
producer:'5'
producer:'6'
consumer:'4'
producer:'7'
consumer:'5'
producer:'8'
producer:'9'
consumer:'6'
producer:'10'
consumer:'7'
producer:'11'
producer:'12'
consumer:'8'
producer:'13'
consumer:'9'
producer:'14'
producer:'15'
consumer:'10'
producer:'16'
consumer:'11'
producer:'17'
producer:'18'
consumer:'12'
producer:'19'
consumer:'13'
[13cse70@FedoraSRVCSE ~]$ consumer:'14'
consumer:'15'
consumer:'16'
consumer:'17'
consumer:'18'
consumer:'19'
```

Exp No:

Regd.No: □□□□□□□□□□

Date :

13) Write a c program to create a thread using pthreads library and let it run its function.

Description: The POSIX thread (pthreads)librariesare a standards based thread API for C/C++. It allows one to spawn a new concurrent process flow. It is most effective on multi-processor or multi-core systems where the process flow can be scheduled to run on another processor thus gaining speed through parallel or distributed processing. Threads require less overhead than "forking" or spawning a new process because the system does not initialize a new system virtual memory space and environment for the process. While most effective on a multiprocessor system, gains are also found on uniprocessor systems which exploit latency in I/O and other system functions which may halt process execution. A thread is spawned by defining a function and it's arguments which will be processed in the thread. The purpose of using the POSIX thread library in your software is to execute software faster.

Program:

```
#include <pthread.h>
#include <stdio.h>
void *inc_x(void *x_void_ptr)
{
int *x_ptr = (int *)x_void_ptr;
while(++(*x_ptr) < 100);
printf("x increment finished\n");
return NULL;
}

int main()
{
int x = 0, y = 0;
printf("x: %d, y: %d\n", x, y);
pthread_t  inc_x_thread;
if(pthread_create(&inc_x_thread, NULL, inc_x, &x))
{
fprintf(stderr, "Error creating thread\n");
return 1;
}
while(++y < 100);
printf("y increment finished\n");
if(pthread_join(inc_x_thread, NULL))
{
fprintf(stderr, "Error joining thread\n");
return 2;
}
printf("x: %d, y: %d\n", x, y);
return 0;
}
```

To Compile :

```
  cc <program_name.c> -lpthread
```

OUTPUT:

```
x: 0, y:0
y increment finished
x increment finished
x: 100, y: 100
```

Exp No:

Regd.No: □□□□□□□□□□

Date :

14.Write a C program to illustrate concurrent execution of threads using pthreads library.

Description: Concurrent programming in a general sense to refer to environments in which the tasks we define can occur in any order. One task can occur before or after another, and some or all tasks can be performed at the same time. We'll use parallel programming to specifically refer to the simultaneous execution of concurrent tasks on different processors. Thus, all parallel programming is concurrent, but not all concurrent programming is parallel. Whether the threads actually run in parallel is a function of the operating system and hardware on which they run. Because Pthreads was designed in this way, a Pthreads program can run without modification on uniprocessor as well as multiprocessor systems.

Program:

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
void *mythread1(void *vargp)
{
int i;
printf("thread1\n");
for(i=1;i<=10;i++)
printf("i=%d\n",i);
printf("exit from thread1\n");
return NULL;
}
void *mythread2(void *vargp)
{
int j;
printf("thread2 \n");
for(j=1;j<=10;j++)
printf("j=%d\n",j);
printf("Exit from thread2\n");
return NULL;
}

int main()
{
pthread_t tid;
printf("before thread\n");
  pthread_create(&tid,NULL,mythread1,NULL);
  pthread_create(&tid,NULL,mythread2,NULL);
  pthread_join(tid,NULL);
  pthread_join(tid,NULL);
exit(0);
}
```

OUT PUT :

```
$ cc <program_name.c>-lpthread

$./a.out
thread1
i=1
i=2;
i=3
thread2
j=1
j=2
j=3
j=4
j=5
j=6
j=7
j=8
i=4
i=5
i=6
i=7
i=8
i=9
i=10
exit from thread1
j=9
j=10
exit from thread2
```