PRINCIPALS OF OPERATING SYSTEMS

LAB MANUAL

INDEX

S.no	Practical	Date	Signature
1.	Study of LINUX Operating System (Linux kernel, shell, basic commands pipe& filter)		
2.	Writing of Shell Scripts (Shell programming).		
3.	To write a C program for implementation of System Calls.		
4.	To write a C program for File Permissions.		
5.	To write a C program for File Operations.		
6.	To write a C program for File Copy and Move.		
7.	To write a C program for Dining Philosophers Program.		
8.	To write a C program for Producer-Consumer Problem concept.		
9.	To write a C program for the following Job Scheduling Algorithms: 1. First Come First Serve Algorithm 2. Shortest Job First Scheduling Algorithm 3. Round Robin Scheduling Algorithm 4. Priority Scheduling Algorithm		
10.	To implement the following memory management schemes: 1. First Fit Algorithm 2. Best Fit Algorithm		
11.	To implement Page replacement algorithms: 1. First In First Out (FIFO) 2. Least Recently Used (LRU)		
12	Write a shell program to perform operations using CASE statement such as addition, subtraction, multiplication and division.		
13	Write a shell program to find largest of three numbers.		
14	Write a shell program to find average of two numbers		

PRACTICAL-1

AIM: Study of Linux operating system (Kernel, shell, Basic commands, Pipe and Filter commands)

What isLinux?

The primary author of Linux is Linus Torvalds. Since his original versions, it has been improved by countless numbers of people. Linux is a freely distributed, multitasking, multiuser operating system that behaves like UNIX. The term "Linux" is actually somewhat vague. "Linux" is used in two ways: specifically to refer to the kernel itself –the heart of any version of Linux –and more generally to refer to any collection of applications that run on the kernel, usually referred to a distribution. The kernel's job is to provide the basic environment in which applications can run, including the basic interfaces with hardware.
Today, thousands of software developers are busy upgrading it, all the while, and extending support online.
Today, Linux is used for a variety of applications. This includes:

- File and Print Server
- E-mail Server
- Fax Server
- Internet gateway
- Firewall
- Database Server
- ISP Server
- Application Server
- Desktop OS

Basic Features of Linux Operating System

Linux systems excel in many areas, ranging from end user concerns such as stability, speed, and ease of use, to serious concerns such as development and networking.

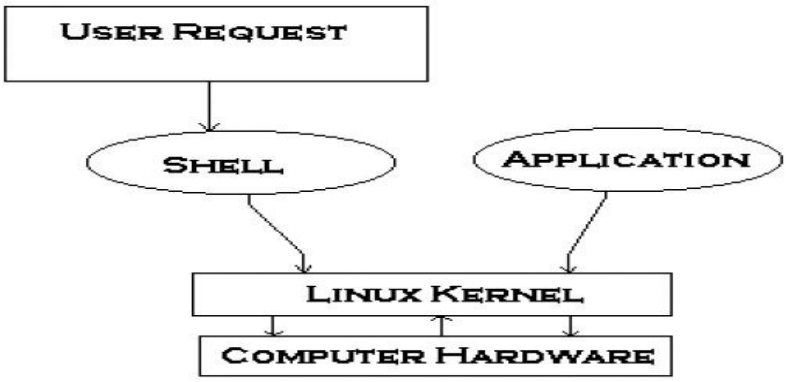
The important features are listed here.

- Multi-programming
- Time Sharing
- Multi-tasking
- High Speed
- Virtual memory
- Shared libraries
- POSIX compliance
- X concept
- Samba
- Dos emulator
- CRON scheduler
- Office suits
- Data archiving utilities
- Licensing
- Web server
- Network information service(NIS)
- Support for programming languages
- Text processing and Word processing

Linux architecture

1. **Linux kernel**

The central nervous system of Linux is the kernel, the operating system code that runs the whole computer. The Linux kernel is the heart of Linux operating system and was originally developed for the Intel 80386 CPU's. Mem01y management is especially strong with the 80386 (compared to earlier CPU's). Linux kernel has the ability to have full access to the entire hardware capabilities of the machine. Actually there is no restriction on what a kernel module is allowed to do.



Typically a kernel might implement a device driver, a file system or a networking protocol. To support 1t large memory requirements when only small amounts of physical RAM are available, Linux suppo1ts swap space. Swap space allows pages of memory to be written to a reserved area of a disk and treated as an extension of physical memory. By moving pages back and forward between the swap space and RAM, Linux can effectively behave as if it had much more physical RAM than it does. Besides this, Linux kernel supports the following features:

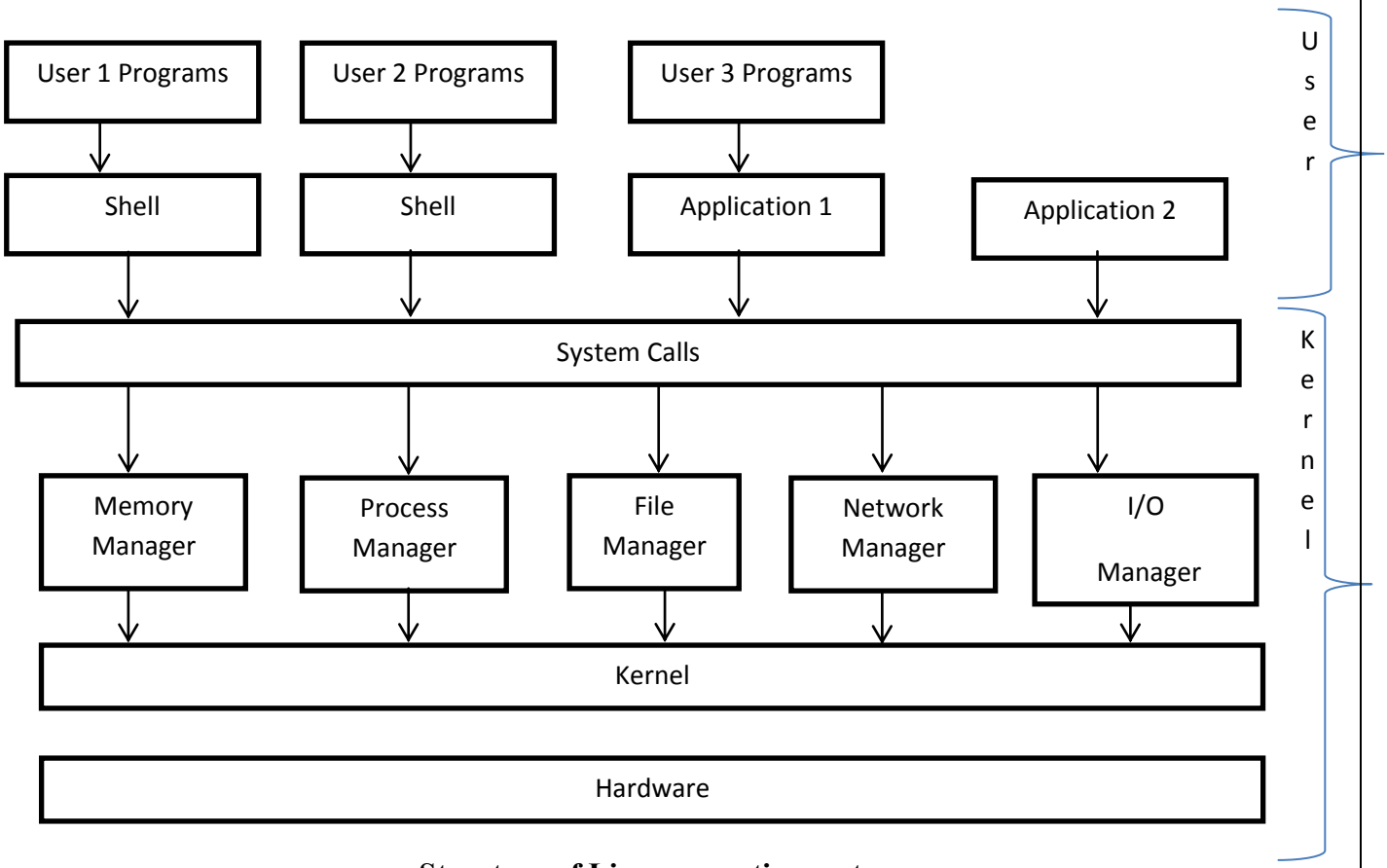
- Has memory protection between processes, so that one program can't bring the whole system down.
- Demand loads executable: Linux only. reads from disk those parts of a program that are actually use.

2. Linux Shell

A shell is a program that run every time you log in. It is a program that lets you interact with the machine. Most users don't even notice the fact that they're manning a shell. They just know that a prompt appears and that from there, they can read their mail and perform other tasks. This prompt is the shell that waits for you to tell what to do. Thus, Linux shell is a user program that the kernel runs when you log in. The shell acts as an interface between the user and the Linux operating system and helps in command inte1pretation for the kernel.

The figure shows that each user gets a copy of the shell where he is allowed to work. The shell interacts with the kernel through system calls. System calls provide a set of routines that allow an application to access kernel services.

The structure of the Linux operating system is as follows:



Structure of Linux operating system

Shells available in Linux

We have several different shells available here:

sh

The first shell, historically, was sh also known as the Bourne shell. It is good for Writing shell scripts, but not so popular for interactive use.

csh

Also known as C-Shell, csti features a syntax somewhat like the C language. It allows (among other things) adding your commands (aliasing), hist01y substitution (re-execution of previously typed commands), and filename-completion.

tcsh

This shell allows you to edit your command line while you're typing it, using emacs like commands. It has a number of other nifty features, but is otherwise compatible with csh.

zsh

The z shell has the best of the features of the Tcsh shell. It also has the capability to emulate all the features of the. Kim shell and supports a large number of utilities and a detailed documentation.

bash

Bash is an acronym for 'BoUine again shell'. It is an enhancement to the BoUine shell

and is the default shell for most of the Linux systems. Compatible with sh for programming purposes, it has many of the good features of csh and tcsh: file name completion, job control, history substitution, emacslike command-line editing, and many more.

Pdksh

Pdksh stands for public domain Kom shell and is an enhancement of the Kom shell. It has been Witten by several voluntary programmers. On Linux systems, ksh is the symbolic link to the pdksh shell.

Basic Linux commands

- **The date command**

Linux maintains a system clock. As for now you can simply display the current date with the date command, which shows the date and time for the nearest second as shown:

```
$ date
Thur Nov 4 11:23:52 IST 1999
```

- **The who command**

Linux maintains an account of all the current users of the system. A list of all the users is displayed by the who command. The who command produces a three column output .This indicates the number of users of the system with their login names in the first column. The second column shows the device names of their respective terminals. The third column indicates the date and time of logging in.

The –H option prints the column headers:

```
$ who –H
```

- **The tty command**

Linux treats even terminals as files. It is therefore reasonable to expect a command which tells you the device name of the terminal you are using with the help of the tty command.

```
$ tty
/dev/ttyp1
$
```

- **The cal command**

This command is used for printing the calendar of any particular month or the entire year. Any calendar from the year 1 to 9999 can be displayed with this command:

```
$ cal 2000
```

- **The man command**

Linux systems maintain an online documentation about each command so that you can get to know the complete description about a particular command. The command:

```
$ man
```

will give a description of the man command in general.

The man command is used to view the description of other commands. For this purpose the

man command is followed by the command name. It has the following format:
$ man command_name

- **The finger utility**

This utility is used to display the status of all the users currently logged on to the Linux system. The finger utility without any parameter, displays a single line output for each user currently.

logged on . It displays the information like the user's login name, full name, terminal name, write status, idle time, login time, the machine's address where the user is currently logged in and the office number. The write permission is displayed along with the terminal name as an asterisk(*). If the asterisk appears after the terminal name, it means that the write permission is denied. The syntax of the finger command is as follows:
finger [options] [user name]

- **The chfn utility**

The chfn (change your finger information) utility is used to change the user's finger information. The chfn utility checks for the user's information from the /etc/passwd file and allows the user to chane information. The syntax for this utility is as follows:
chfn [options] [username]
The chfn asks for the password of the user to authenticate the user for changing the finger information. If you do not want to change any particular information, you can press Enter at the corresponding prompt and proceed with the remaining parameters.

- **The head command**

The head command is used to display the top few records of the file. The syntax of the command is as follows:
$ head [option] file
The option is as follows:
count Display the first count lines of file
The easiest way to use this command is to specify a filename without specifying the number of lines to be displayed. If this is the case, the first ten records of the file are displayed.
$ head tmp.lst
will display the first ten records of the file tmp.lst.

- **The tail command**

The tail command displays the end few records of the file. If no line count is given, the tail command displays the last ten lines of the file. We also have an option to specify a count and select that many lines from the end of file.
Consider an example: $ tail –3 emp.lst

- **The mesg command**

The user has the option to allow or disallow other users to write on his terminal. The two options available with the mesg command are as follows:

mesg y
This option allows other users to write to your terminal
mesg n
This option disallows other users to write to your terminal
If you type the mesg command without any option, it displays the status of your terminal, say for example,
$ mesg
is y

- **The wall command**

The wall command is used to write to all the users. The wall command sends the message to all the users who are currently logged on to the Linux system and have their mesg permission set to 'y'. The syntax of the command is as follows:

wall type in the above command at the command prompt. Press <Enter>
Now write the message you want to broadcast. Press ctrl <d>. The message would be broadcasted to all the users currently logged on.

- **pwd (Print working directory) command**

Use the pwd command to print the working directory (the current directory you are in).

```
> pwd
/home
> cd /home/rich/www
> pwd
/home/rich/www
>
```

Line 1 of this example shows the command pwd has been entered
Line 2 displays, or 'prints' the output of the pwd command (ie: the directory you are in - /home in this case)
Line 3 uses the cd command (change directory) to move to the /home/rich/www directory
Line 4 enters the pwd command again
Line 5 shows we are now in the /home/rich/www directory
Line 6 is the prompt again

- **cd (Change directory) command**

Use the **cd** command to change to another directory.

The syntax is cd followed by the name of the directory you want to go to.
Example: cd /home/user/www will change the directory you are in to /home/user/www.

```
> cd /home/rich/www
> pwd
/home/rich/www
>
```

Line 1 shows the command cd /home/rich/www, which should put me in the folder /home/rich/www
Line 2 is the pwd command (print working directory) to see if we are in the right directory
Line 3 is the output of the pwd command - which shows that we are indeed in /home/rich/www
Line 4 is the prompt again

Pipe Command

If you have a series of commands in which the output of one command is the input of the next, you can pipe the output without saving it in temporary files:

first_command | next_command

For example, if you wanted to point out a s01ted version of a file that contained a list of names
and phone numbers, you could use a pipe (as well as input redirection):

sort < my_phone_list|lpr

Similarly, in order to display the contents of the current directory, a screen-full at a time, you can give the following commands:

$ls > myfile

$ more myfile

Here, the listing of the direct01y is stored in the file, myfile, by the first co and this file is then used as input by the more command.
The above two steps can be combined and executed as a single command without creating a temporary file,

$ ls| more

The ve1tical bar (|) is the pipe character which indicates to the shell that the output of the command before the '|' is the input to the command after the '|'.

Filter Commands

A filter is a program that takes its input from the standard input file, processes (or filters)it and sends its output to file standard output file. Linux has a rich set of filters that can be used to work on data in an efficient way. Some examples of filters are:

- grep
- tr
- Sort
- cut
- wc

The grep command

The grep command searches a file for a specified pattern ad displays it on screen. The syntax of this command is as follows:

grep[options] regular expression filename[s]

The tr command

The tr command is used to squeeze the space between columns or it transliterates characters i.e. it copies the standard input to standard output with substitution or deletion of specific

characters.

The sort command

The sort command sorts lines of all the specified files together and writes the result on the standard output.

Example

$ sort myfile

will sort the file considering each record of a file as one single field.

The wc command

This command is used to count the number of lines, words and characters in specified file or standard input.

Three options are possible:

-w for words

-l for lines

-c for characters

The cut command

This command is used to delete some columns in multi columnar output.

Some more basic linux commands

Command	Description
Ls	lists the content of a directory
Cd	change directory
cd ..	parent directory
Mkdir	creates a new directory
Rmdir	eliminates a directory
Cp	copy a file
Mv	moves a file
Rm	removes a file
Passwd	changes the user's password

Cat	displays the file's content and creates file
More	displays the file's content with pauses
Lpr	prints the requested file

PRACTICAL-2

AIM: Study of shell programming.

Normally shells are interactive. It means shell accept command from you (via keyboard) and execute them. But if you use command one by one (sequence of 'n' number of commands) , the you can store this sequence of command to text file and tell the shell to execute this text file instead of entering the commands. This is known as **shell script**.

Shell script defined as:

"Shell Script is **series of command** written **in plain text file**. Shell script is just like batch file is MS-DOS but have more power than the MS-DOS batch file."

Benefits of shell script

- Shell script can take input from user, file and output them on screen.
- Useful to create our own commands.
- Save lots of time.
- To automate some task of day today life.
- System Administration part can be also automated.

How to write shell script

Following steps are required to write shell script:

(1) Use any editor like vi or mcedit to write shell script.

(2) After writing shell script set execute permission for your script as follows

syntax:

chmod permission your-script-name

Examples:

$ chmod +x your-script-name
$ chmod 755 your-script-name

Note: This will set read write execute(7) permission for owner, for group and other permission is read and execute only(5).

(3) Execute your script as

syntax:

bash your-script-name
sh your-script-name
./your-script-name

Examples:
$ bash bar

$ sh bar
$ ./bar

NOTE In the last syntax ./ means current directory, But only . (dot) means execute given command file in current shell without starting the new copy of shell, The syntax for . (dot) command is as follows

Syntax:
. command-name

Example:
$. foo

FIRST SHELL SCRIPT THAT WILL PRINT "KNOWLEDGE IS POWER" ON SCREEN

$ vi first
#
 # My first shell script
#
clear
echo "Knowledge is Power"

After saving the above script, you can run the script as follows:
$ ./first

This will not run script since we have not set execute permission for our script *first*; to do this type command
$ chmod 755 first
$ ./first

First screen will be clear, then Knowledge is Power is printed on screen.

Script Command(s)	Meaning
$ vi first	Start vi editor
# # My first shell script #	# followed by any text is considered as comment. Comment gives more information about script, logical explanation about shell script. *Syntax:* # comment-text
Clear	clear the screen
echo "Knowledge is Power"	To print message or value of variables on screen, we use echo command, general form of echo command is as follows *syntax:* echo "Message"

Variables in shell

To process our data/information, data must be kept in computers RAM memory. RAM memory is divided into small locations, and each location had unique number called memory location/address, which is used to hold our data. Programmer can give a unique name to this memory location/address called memory variable or variable (Its a named storage location that may take different values, but only one at a time).

In Linux (Shell), there are two types of variable:

(1) **System variables** - Created and maintained by Linux itself. This type of variable defined in CAPITAL LETTERS.

(2) **User defined variables (UDV)** - Created and maintained by user. This type of variable defined in lower letters. You can see system variables by giving command like **$ set**, some of the important System variables are:

System Variable	Meaning
BASH=/bin/bash	Our shell name
BASH_VERSION=1.14.7(1)	Our shell version name
COLUMNS=80	No. of columns for our screen
HOME=/home/vivek	Our home directory
LINES=25	No. of columns for our screen
LOGNAME=students	students Our logging name
OSTYPE=Linux	Our Os type
PATH=/usr/bin:/sbin:/bin:/usr/sbin	Our path settings
PS1=[\u@\h \W]\$	Our prompt settings
PWD=/home/students/Common	Our current working directory
SHELL=/bin/bash	Our shell name
USERNAME=vivek	User name who is currently login to this PC

AIM: Write a shell program to perform operations using CASE statement such as addition, subtraction, multiplication and division.

PROGRAM:

```
echo n1 n2
read n1 n2
echo a=add
echo b=sub
echo c=mul
echo d=div
echo ch
read ch
case $ch in
    a) let z=$n1+$n2
       echo add=$z
       ;;
    b) let z=$n1-$n2
       echo sub=$z
       ;;
    c) let z=$n1*$n2
       echo mul=$z
       ;;
    d) let z=$n1/$n2
       echo div=$z
       ;;
     *)
     echo invalid option
     ;;
     Esac
```

OUTPUT

```
n1 n2
3   4
a=add
b=sub
c=mul
d=div
ch
a
add=7
```

AIM: Write a shell program to find largest of three numbers.

PROGRAM:

```
echo enter1
read n1
echo enter2
read n2
echo enter3
read n3
if[$n1-gt $n2]&&[$n1-gt $n3]
then echo $n1 is larger
elif[$n2-gt $n1]&&[$n2-gt $n3]
then echo $n2 is larger
else echo $n3 is larger
fi
```

OUTPUT

n1 n2 n3
12 36 48
48 is larger

AIM: Write a shell program to find average of two numbers.

PROGRAM:

```
echo Enter the 1st number
read a
echo Enter the 2nd number
read b
c='expr $a+$b'
d=$c/2
echo The average of two numbers is $d
```

OUTPUT

Enter the 1st number
6
Enter the 2nd number
4
The average of two numbers is 5

AIM: Write a shell program to add two numbers

PROGRAM:

```
Echo enter the first number
Read a
Echo enter the second number
Read b
C=’expr $a+$b’
Echo addition of two numbers is $c.
```

OUTPUT

Enter the 1st number
6
Enter the 2nd number
4
The sum of two numbers is 10

PRACTICAL-3

AIM: To write a C program for implementation of System Calls.

A system call is a request for service that a program makes of the kernel. The service is generally something that only the kernel has the privilege to do, such as doing I/O. Programmers don't normally need to be concerned with system calls because there are functions in the Linux Library to do virtually everything that system calls do. These functions work by making system calls themselves. For example, there is a system call that changes the permissions of a file, but you don't need to know about it because you can just use the Library's chmod function.

System calls are sometimes called kernel calls.

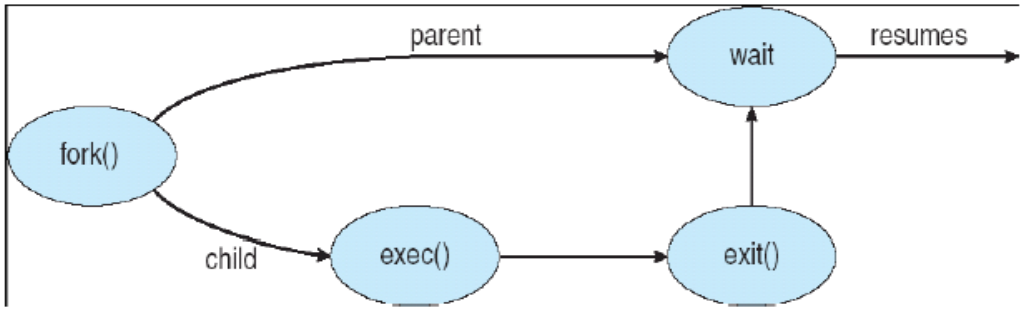
NAME
fork – create a new process system call.

SYNOPSIS
#include<unistd.h>
pid_t fork(void);

DESCRIPTION
The *fork()* function shall create a new process. The new process (child process) shall be an exact copy of the calling process (parent process). The child process shall have a unique process ID.
System call **fork()** is used to create processes. It takes no arguments and returns a process ID. The purpose of **fork()** is to create a ***new*** process, which becomes the *child* process of the caller. After a new child process is created, ***both*** processes will execute the next instruction following the ***fork()***system call. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of **fork()**:

- If **fork()** returns a negative value, the creation of a child process was unsuccessful.
- **fork()** returns a zero to the newly created child process.
- **fork()** returns a positive value, the ***process ID*** of the child process, to the parent.

EXECUTION
Parent and children execute concurrently
Parent waits until children terminate



NAME
execl, -execute a file

SYNOPSIS
#include<unistd.h>
extern char **environ;
int execl(const char *path, const char *arg(), …/*, (char *)0 */);

DESCRIPTION
The *exec* family of functions shall replace the current process image with a new process image. The new image shall be constructed from regular, executable file called the new *process image file*. There shall be no return from a successful exec, because the calling process image is overlaid by the new process image.

Fork-exec is a commonly used technique in Unix whereby an executing process spawns a new program. fork() is the name of the system call that the parent process uses to "divide" itself ("fork") into two identical processes. After calling fork(), the created child process is actually an exact copy of the parent - which would probably be of limited use - so it replaces itself with another process using the system call exec(). The parent process can either continue execution or wait for the child process to complete. The child, after discovering that it is the child, replaces itself completely with another program, so that the code and address space of the original program are lost. If the parent chooses to wait for the child to die, then the parent will receive the exit code of the program that the child executed. Otherwise, the parent can ignore the child process and continue executing as it normally would; to prevent the child becoming a zombie it should wait on children at intervals or on SIGCHLD. When the child process calls exec(), all data in the original program is lost, and replaced with a running copy of the new program. This is known as overlaying. Although all data is replaced, the file descriptors that were open in the parent are closed only if the program has explicitly marked them *close-on-exec*. This allows for the common practice of the parent creating a pipe prior to calling fork() and using it to communicate with the executed program.

Using execl()

The following example executes the *ls* command, specifying the pathname of the executable **(/bin/ls)**
and using arguments supplied directly to the command to produce single-column output.

#include<unistd.h>
int ret;
…
ret=execl(“/bin/ls”, “ls”, “-l”, (char *)0);

NAME
exit, _exit – terminate a process

SYNOPIS
#include<stdlib.h>
void exit(int status);

DESCRIPTION
The exit status will be n, if specified. Otherwise, the value will be the exit value of the last command executed, or zero if no command was executed. When *exit* is executed in a trap action, the last command is considered to be the command that executed immediately preceding the trap action. This system call is used to terminate (normal/abnormal) the current running program.

NAME
wait – wait for a child process to stop or terminate

SYNOPIS
#include<sys/types.h>
#include<sys/wait.h>
pid_t wait(int *stat_loc);

DESRIPTION
The *wait()* and *waitpid()* functions allow the calling process to obtain status information pertaining to one of its child processes. Various options permit status information to be obtained for child processes that have terminated or stopped. If status information is available for two or more child processes, the order in which their status is reported is unspecified.

There are a number of system calls that a process can use to obtain file information. The most useful one is "stat" system call. The stat() system call is used to obtain file information.

- A parent process usually needs to synchronize its actions by waiting until the child process has either stopped or terminated its actions.
- The wait() system call allows the parent process to suspend its activities until one of these actions has occurred.
- The wait() system call accepts a single argument, which is a pointer to an integer and returns a value defined as type pid_t.
- If the calling process does not have any child associated with it, wait will return immediately with a value of -1.
- If any child processes are still active, the calling process will suspend its activity until a child process terminates.

NAME
Readdir() –To read a directory

SYNOPSIS
#include <sys/types.h>
#include <dirent.h>
struct dirent *readdir(DIR *dir);

DESCRIPTION
The readdir() function returns a pointer to a dirent structure representing the next directory entry in the directory stream pointed to by dir. It returns NULL on reaching the end-of-file or if an error occurred. On Linux, the dirent structure is defined as follows:

```
struct dirent {
ino_t d_ino; /* inode number */
```

```
off_t d_off; /* offset to the next dirent */
unsigned short d_reclen; /* length of this record */
unsigned char d_type; /* type of file */
char d_name[256]; /* filename */
};
```

The data returned by readdir() may be overwritten by subsequent calls to readdir() for the same directory stream.

RETURN VALUE

The readdir() function returns a pointer to a dirent structure, or NULL if an error occurs or end-of-file is reached. On error, errno is set appropriately.

NAME

Opendir() – To open a directory

SYNOPSIS

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *name);
```

DESCRIPTION

The opendir() function opens a directory stream corresponding to the directory name, and returns a pointer to the directory stream. The stream is positioned at the first entry in the directory.

RETURN VALUE

The opendir() function returns a pointer to the directory stream. On error, NULL is returned, and errno is set appropriately.

Program: C Program for implementation of the system calls

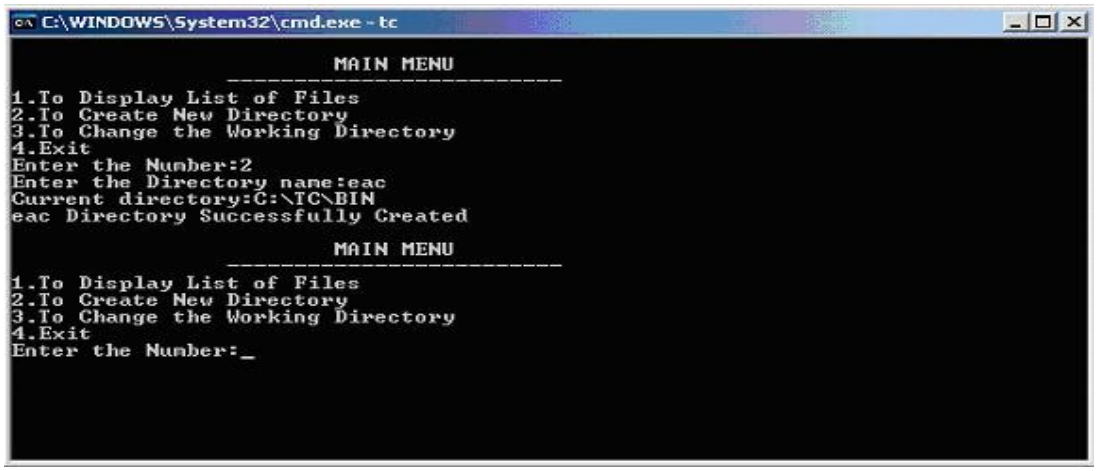
```
#include<stdio.h>
#include<conio.h>
#include<dir.h>
#include<dos.h>
void main()
{
 int ch;
 clrscr();
 do
 {
 printf("\n\t\t\tMAIN MENU\n\t\t------------------------\n");
 printf("1.To Display List of Files\n");
 printf("2.To Create New Directory\n");
 printf("3.To Change the Working Directory\n");
 printf("4.Exit\n");
 printf("Enter the Number:");
 scanf("%d",&ch);
 switch(ch)
 {
  case 1:
    list_file();
    break;

    case 2:
    directory();
    break;

  case 3:
    change_dir();
    break;

    case 4:
    exit(0);
 } }while(ch<=4);
}
int list_file()
{
 int l;
 char e[]="*.*";
 clrscr();
 printf("\t\tLIST FILE DETAIL\n\t-------------------------------------\n");
 printf("1.List All Files\n2.List of Extention Files\n3.List of Name Wise\n");
 scanf("%d",&l);
```

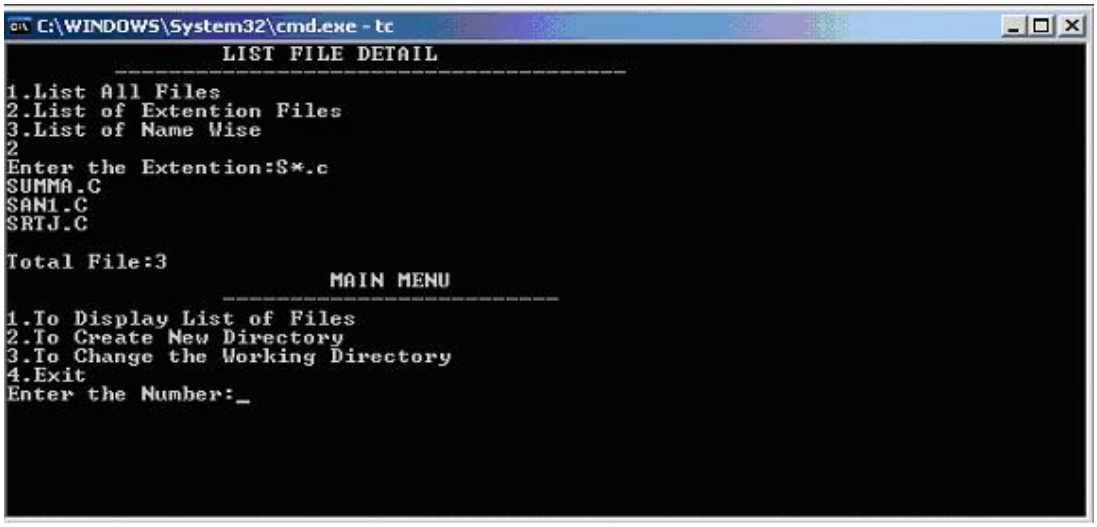
```
 switch(l)
 {
  case 1:
   printf("List of All(*.*) Files\n");
   subfun(e);
   break;

  case 2:
   printf("Enter the Extention:");
   scanf("%s",&e);
   subfun(e);
   break;

  case 3:
   printf("Enter the Name wise(eg:moha*.*):");
   scanf("%s",&e);
   subfun(e);
   break;
 }
 return 0;
}

int directory()
{
 struct ffblk ffblk;
 unsigned attrib;
 int d;
 char name[10],buffer[MAXPATH];
 printf("Enter the Directory name:");
 scanf("%s",&name);
 getcwd(buffer,MAXPATH);
 printf("Current directory:%s\n",buffer);

 if(_dos_getfileattr(name,&attrib)==0)
  {
     printf("%s has already available",name);
     return 0;
  }
  else
   {
    mkdir(name);
    printf("%s Directory Successfully Created\n",name);
   }
 return 0;
}
```

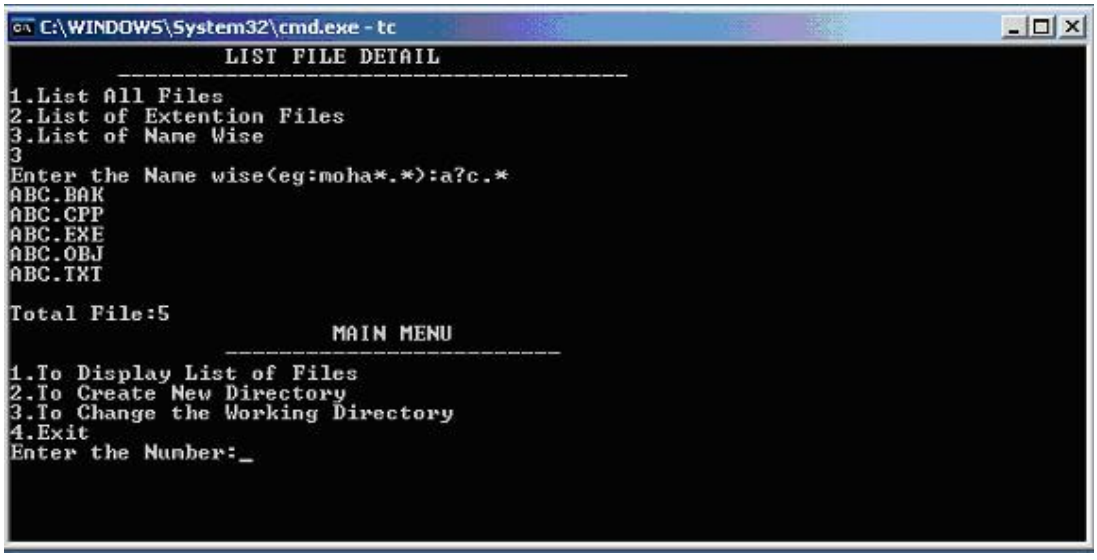
```
int change_dir()
{
 char buffer[MAXPATH];
 int d,d1;
 printf("\nCurrent Directory:%s\n",getcwd(buffer,MAXPATH));
 printf("\t\tChange Directory\n\t\t----------------\n");
 printf("\n1.Step by Step Backward\n2.Goto Root Directory\n3.Forward Directory \nEnter the
number:");
 scanf("%d",&d);
 switch(d)
 {
  case 1:
      chdir("..");
      break;

  case 2:
     chdir("\\");
     break;

  case 3:
     printf("Please enter the Filename:");
     scanf("%s",buffer);
     chdir(buffer);
     break;
 }

 printf("\nCurrent Directory:%s",getcwd(buffer,MAXPATH));
 return 0;
}



int subfun(s)
char s[10];
{
   struct ffblk ffblk;
   int d,p=0,i=0;
   d=findfirst(s,&ffblk,0);
   while(!d)
   {
    printf("%s\n",ffblk.ff_name);
    d=findnext(&ffblk);
    i++;
    p=p+1;
```

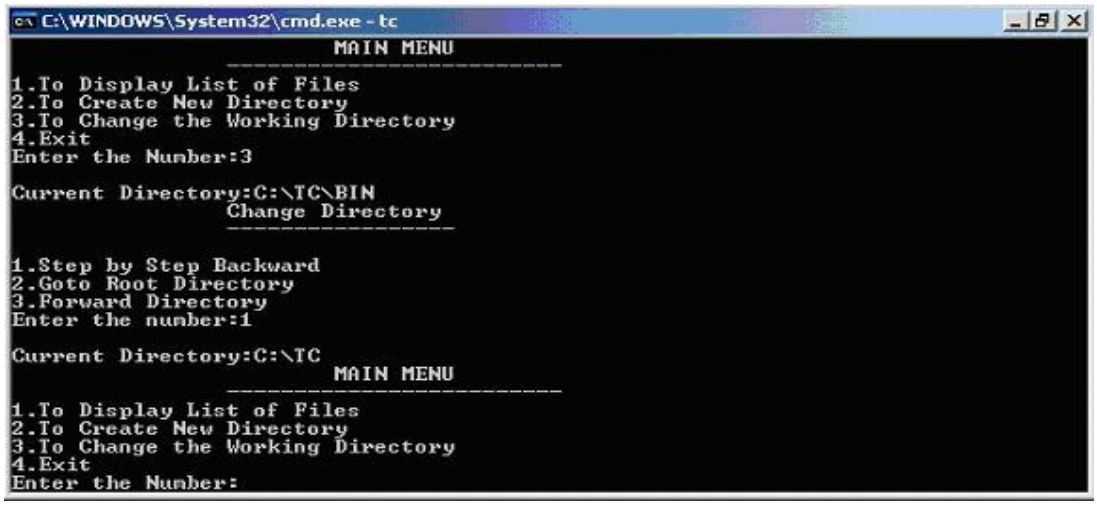
```
    if(p>=22)
    {
     printf("Press any key to continue:\n");
     getch();
     p=0;
    }
   }
   printf("\nTotal File:%d",i);
return 0;
}
```

SAMPLE INPUT AND OUTPUT:

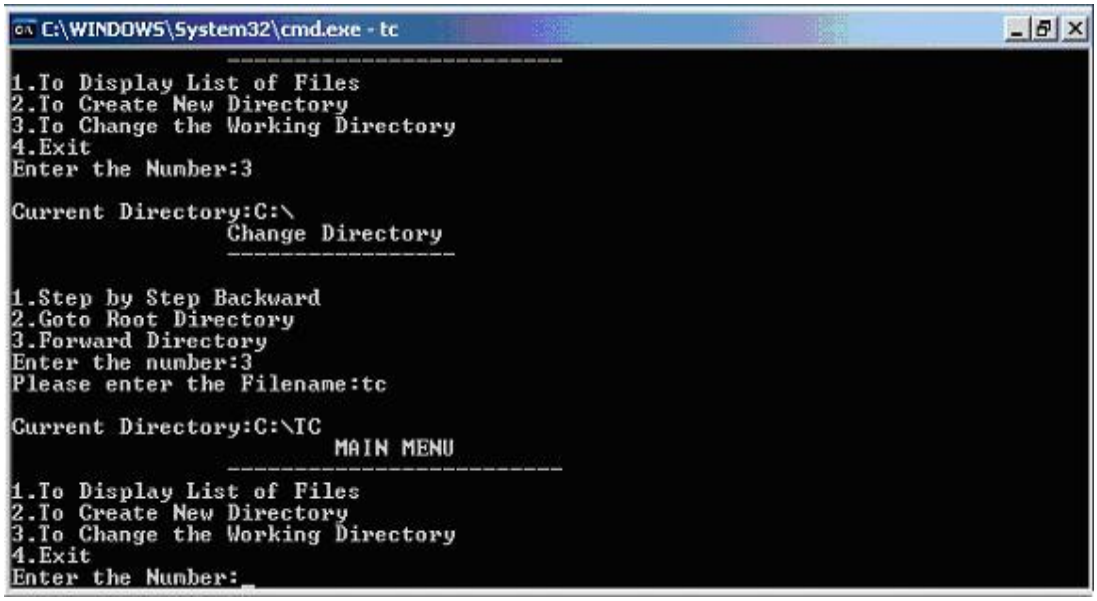







C:\WINDOWS\System32\cmd.exe - tc
MAIN MENU
1.To Display List of Files
2.To Create New Directory
3.To Change the Working Directory
4.Exit
Enter the Number:3
Current Directory:C:\TC\BIN
Change Directory
1.Step by Step Backward
2.Goto Root Directory
3.Forward Directory
Enter the number:1
Current Directory:C:\TC
MAIN MENU
1.To Display List of Files
2.To Create New Directory
3.To Change the Working Directory
4.Exit
Enter the Number:


C:\WINDOWS\System32\cmd.exe - tc
MAIN MENU
1.To Display List of Files
2.To Create New Directory
3.To Change the Working Directory
4.Exit
Enter the Number:3
Current Directory:C:\TC
Change Directory
1.Step by Step Backward
2.Goto Root Directory
3.Forward Directory
Enter the number:2
Current Directory:C:\
MAIN MENU
1.To Display List of Files
2.To Create New Directory
3.To Change the Working Directory
4.Exit
Enter the Number:


C:\WINDOWS\System32\cmd.exe - tc
1.To Display List of Files
2.To Create New Directory
3.To Change the Working Directory
4.Exit
Enter the Number:3
Current Directory:C:\
Change Directory
1.Step by Step Backward
2.Goto Root Directory
3.Forward Directory
Enter the number:3
Please enter the Filename:tc
Current Directory:C:\TC
MAIN MENU
1.To Display List of Files
2.To Create New Directory
3.To Change the Working Directory
4.Exit
Enter the Number:

PRACTICAL – 4

AIM: To write a C Program for File Permissions.

Every user on a Unix system has a unique username, and is a member of at least one group (the primary group for that user). This group information is held in the password file (/etc/passwd). A user can also be a member of one or more other groups. The auxiliary group information is held in the file /etc/group. Only the administrator can create new groups or add/delete group members (one of the shortcomings of the system).

Every directory and file on the system has an owner, and also an associated group. It also has a set of permission flags which specify separate read, write and execute permissions for the 'user' (owner), 'group', and 'other' (everyone else with an account on the computer) The 'ls' command shows the permissions and group associated with files when used with the -l option. On some systems (e.g. Coos), the '-g' option is also needed to see the group information.

An example of the output produced by 'ls -l' is shown below.

drwx------ 2 richard staff 2048 Jan 2 1997 private

drwxrws--- 2 richard staff 2048 Jan 2 1997 admin

-rw-rw---- 2 richard staff 12040 Aug 20 1996 admin/userinfo

drwxr-xr-x 3 richard user 2048 May 13 09:27 public

Understanding how to read this output is useful to all unix users, but especially people using group access permissions.

Field 1: a set of ten permission flags.
Field 2: link count (don't worry about this)
Field 3: owner of the file
Field 4: associated group for the file
Field 5: size in bytes
Field 6-8: date of last modification (format varies, but always 3 fields)
Field 9: name of file (possibly with path, depending on how ls was called)

The permission flags are read as follows (left to right)

position	**Meaning**
1	directory flag, 'd' if a directory, '-' if a normal file, something else occasionally may appear here for special devices.
2,3,4	read, write, execute permission for User (Owner) of file
5,6,7	read, write, execute permission for Group
8,9,10	read, write, execute permission for Other

value	**Meaning**
-	in any position means that flag is not set

r	file is readable by owner, group or other
w	file is writeable. On a directory, write access means you can add or delete files
x	file is executable (only for programs and shell scripts - not useful for data files). Execute permission on a directory means you can list the files in that directory
s	in the place where 'x' would normally go is called the set-UID or set-groupID flag.

On an executable program with set-UID or set-groupID, that program runs with the effective permissions of its owner or group.

For a directory, the set-groupID flag means that all files created inside that directory will inherit the group of the directory. Without this flag, a file takes on the primary group of the user creating the file. This property is important to people trying to maintain a directory as group accessible. The subdirectories also inherit the set-groupID property.

The default file permissions (umask):

Each user has a default set of permissions which apply to all files created by that user, unless the software explicitly sets something else. This is often called the 'umask', after the command used to change it. It is either inherited from the login process, or set in the .cshrc or .login file which configures an individual account, or it can be run manually.

Typically the default configuration is equivalent to typing 'umask 22' which produces permissions of:

-rw-r--r-- for regular files, or

drwxr-xr-x for directories.

In other words, user has full access, everyone else (group and other) has read access to files, lookup access to directories.

When working with group-access files and directories, it is common to use 'umask 2' which produces permissions of:

-rw-rw-r-- for regular files, or

drwxrwxr-x for directories.

For private work, use 'umask 77' which produces permissions:

-rw------- for regular files, or

drwx------ for directories.

The logic behind the number given to umask is not intuitive.

The command to change the permission flags is "chmod". Only the owner of a file can change its permissions.

The command to change the group of a file is "chgrp". Only the owner of a file can change its group, and can only change it to a group of which he is a member.

See the online manual pages for details of these commands on any particular system (e.g. "man chmod").

Examples of typical useage are given below:

chmod g+w myfile

give group write permission to "myfile", leaving all other permission flags alone

chmod g-rw myfile

remove read and write access to "myfile", leaving all other permission flags alone

chmod g+rwxs mydir

give full group read/write access to directory "mydir", also setting the set-groupID flag so that directories created inside it inherit the group

chmod u=rw,go= privatefile

explicitly give user read/write access, and revoke all group and other access, to file 'privatefile'

chmod -R g+rw .

give group read write access to this directory, and *everything* inside of it (-R = recursive)

chgrp -R medi .

change the ownership of this directory to group 'medi' and *everything* inside of it (-R = recursive). The person issuing this command must own all the files or it will fail.

WARNINGS:

Putting 'umask 2' into a startup file (.login or .cshrc) will make these settings apply to everything you do unless manually changed. This can lead to giving group access to files such as saved email in your home directory, which is generally not desireable.

Making a file group read/write without checking what its group is can lead to accidentally giving access to almost everyone on the system. Normally all users are members of some default group such as "users", as well as being members of specific project-oriented groups. Don't give group access to "users" when you intended some other group.

Remember that to read a file, you need execute access to the directory it is in AND read access to the file itself. To write a file, your need execute access to the directory AND write access to the file. To create new files or delete files, you need write access to the directory. You also need execute access to all parent directories back to the root. Group access will break if a parent directory is made completely private.

Program: A C Program to implement File Operations and System Calls

```
#include<stdio.h>
#include<conio.h>
#include<dir.h>
#include<dos.h>
void main()
{
 int ch;
 clrscr();
 do
 {
 printf("\n\t\t\tMAIN MENU\n\t\t------------------------\n");
 printf("1.To Display List of Files\n");
 printf("2.To Create New Directory\n");
 printf("3.To Change the Working Directory\n");
 printf("4.Exit\n");
 printf("Enter the Number:");
 scanf("%d",&ch);
 switch(ch)
 {
  case 1:
    list_file();
    break;

    case 2:
    directory();
    break;

  case 3:
    change_dir();
    break;

    case 4:
    exit(0);
 } }while(ch<=4);
}
int list_file()
{
 int l;
 char e[]="*.*";
 clrscr();
 printf("\t\tLIST FILE DETAIL\n\t-------------------------------------\n");
 printf("1.List All Files\n2.List of Extention Files\n3.List of Name Wise\n");
 scanf("%d",&l);
```

```
 switch(l)
 {
  case 1:
   printf("List of All(*.*) Files\n");
   subfun(e);
   break;

  case 2:
   printf("Enter the Extention:");
   scanf("%s",&e);
   subfun(e);
   break;

  case 3:
   printf("Enter the Name wise(eg:moha*.*):");
   scanf("%s",&e);
   subfun(e);
   break;
 }
 return 0;
}

int directory()
{
 struct ffblk ffblk;
 unsigned attrib;
 int d;
 char name[10],buffer[MAXPATH];
 printf("Enter the Directory name:");
 scanf("%s",&name);
 getcwd(buffer,MAXPATH);
 printf("Current directory:%s\n",buffer);

 if(_dos_getfileattr(name,&attrib)==0)
  {
     printf("%s has already available",name);
     return 0;
  }
  else
   {
    mkdir(name);
    printf("%s Directory Successfully Created\n",name);
   }
 return 0;
}
```

```
int change_dir()
{
 char buffer[MAXPATH];
 int d,d1;
 printf("\nCurrent Directory:%s\n",getcwd(buffer,MAXPATH));
 printf("\t\tChange Directory\n\t\t----------------\n");
 printf("\n1.Step by Step Backward\n2.Goto Root Directory\n3.Forward Directory \nEnter the
number:");
 scanf("%d",&d);
 switch(d)
 {
  case 1:
      chdir("..");
      break;

  case 2:
     chdir("\\");
     break;

  case 3:
     printf("Please enter the Filename:");
     scanf("%s",buffer);
     chdir(buffer);
     break;
 }

 printf("\nCurrent Directory:%s",getcwd(buffer,MAXPATH));
 return 0;
}



int subfun(s)
char s[10];
{
   struct ffblk ffblk;
   int d,p=0,i=0;
   d=findfirst(s,&ffblk,0);
   while(!d)
   {
    printf("%s\n",ffblk.ff_name);
    d=findnext(&ffblk);
    i++;
    p=p+1;
```

```
    if(p>=22)
    {
     printf("Press any key to continue:\n");
     getch();
     p=0;
    }
   }
   printf("\nTotal File:%d",i);
return 0;
}
```

PRACTICAL-5

AIM: To write a C program for File Operations

DESCRIPTION

File-I/O through system calls is simpler and operates at a lower level than making calls to the C file-I/O library. There are seven fundamental file-I/O system calls:

- creat() Create a file for reading or writing.
- open() Open a file for reading or writing.
- close() Close a file after reading or writing..
- write() Write bytes to file.
- read() Read bytes from file.

These calls were devised for the UNIX operating system and are not part of the ANSI C spec. Use of these system calls requires a header file named "fcntl.h":

The creat() System Call

The "creat()" system call, of course, creates a file.

It has the syntax:

int fp; /* fp is the file descriptor variable */

fp = creat(<filename>, <protection bits>);

Ex: fp=creat("students.dat",RD_WR);

This system call returns an integer, called a "file descriptor", which is a number that identifies the file generated by "creat()". This number is used by other system calls in the program to access the file.

Should the "creat()" call encounter an error, it will return a file descriptor value of -1.

The "filename" parameter gives the desired filename for the new file. The "permission bits" give the "access rights" to the file.

A file has three "permissions" associated with it:

1. **Write permission** - Allows data to be written to the file.
2. **Read permission** - Allows data to be read from the file.
3. **Execute permission** - Designates that the file is a program that can be run.

These permissions can be set for three different levels:

User level: Permissions apply to individual user.

Group level: Permissions apply to members of user's defined "group".

System level: Permissions apply to everyone on the system.

The open() Sytem Call

The "open()" system call opens an existing file for reading or writing.

It has the syntax:

<file descriptor variable> = open(<filename>, <access mode>);

The "open()" call is similar to the "creat()" call in that it returns a file descriptor for the given file, and returns a file descriptor of -1 if it encounters an error. However, the second parameter is an "access mode", not a permission code.

There are three modes (defined in the "fcntl.h" header file):

O_RDONLY Open for reading only.

O_WRONLY Open for writing only.

O_RDWR Open for reading and writing.

For example, to open "data" for writing, assuming that the file had been created by another program, the following statements would be used:

```
int fd;
fd = open( "students.dat", O_WRONLY );
```

A few additional comments before proceeding:

A "creat()" call implies an "open()". There is no need to "creat()" a file and then "open()" it.

The close() Sytem Call

The "close()" system call is very simple. All it does is "close()" an open file when there is no further need to access it.

The "close()" system call has the syntax:

close(<file descriptor>);

The "close()" call returns a value of 0 if it succeeds, and returns -1 if it encounters an error.

The write() Sytem Call

The "write()" system call writes data to an open file.

It has the syntax:

write(<file descriptor>, <buffer>, <buffer length>);

The file descriptor is returned by a "creat()" or "open()" system call. The "buffer" is a pointer to a variable or an array that contains the data; and the "buffer length" gives the number of bytes to be written into the file. While different data types may have different byte lengths on different systems, the "sizeof()" statement can be used to provide the proper buffer length in bytes. A "write()" call could be specified as follows:

float array[10];

write(fd, array, sizeof(array));

The "write()" function returns the number of bytes it actually writes. It will return -1 on an error.

The read() Sytem Call

The "read()" system call reads data from a open file. Its syntax is exactly the same as that of the "write()" call:

read(<file descriptor>, <buffer>, <buffer length>);

The "read()" function returns the number of bytes it actually returns.

At the end of file it returns 0, or returns -1 on error.

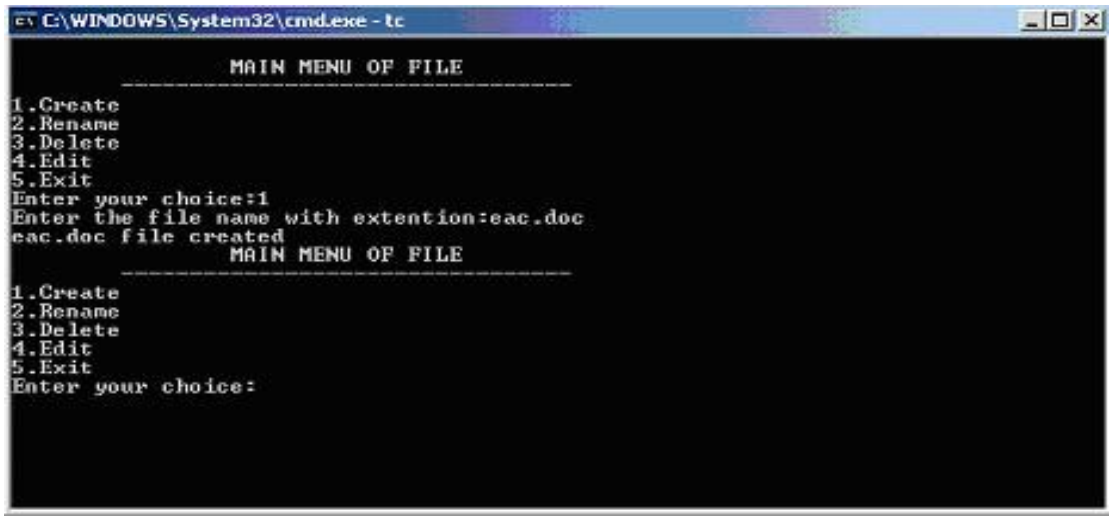
Algorithm:

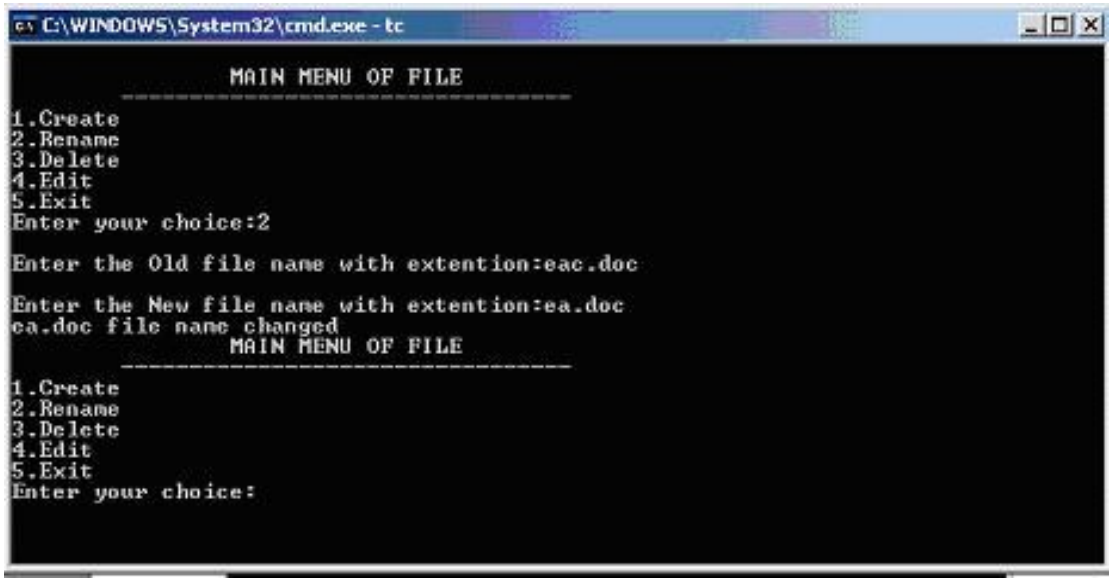
1. Start the Program.
2. Open a file for O_RDWR for read and write, O_CREATE for creating a file, O_TRUNC for truncates a file.
3. Using getchar function, read the character and stored in the string [] array.
4. The string [] array is write into a file and close it.
5. Then the file is opened for read only mode and read the characters and displayed it and close the file.
6. Stop the program.

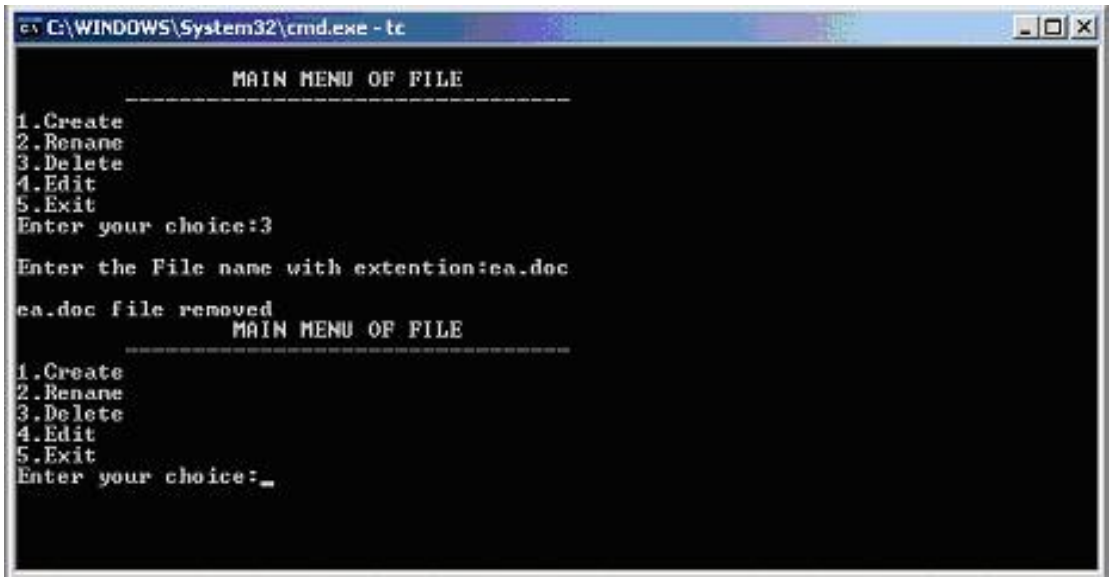
Program: To write a C program for File Operations

```
#include<process.h>
#include<dos.h>
#include<stdio.h>
#include<conio.h>
#include<dir.h>
#include<io.h>
void main()
{
 char name[10],c,ch;
 unsigned attrib;
 int d,n,f,p;
 clrscr();
 printf("\t\tMAIN MENU OF PERMISSION\n\t--------------------------------\n");
 printf("1.Only Read\n2.Only Write\n3.Exit\nEnter your choice:\n");
 scanf("%d",&n);
 switch(n)
 {

  case 1:
   printf("\nEnter the File Name:\n");
   scanf("%s",name);
   attrib |= _A_RDONLY;
   printf("%s file  read permission Accepted",name);
   break;

  case 2:
   printf("\nEnter the File Name:\n");
   scanf("%s",name);
   attrib = _A_ARCH;
   printf("%s excute permission accepted",name);
   break;

  case 3:
    exit(0); }



 if(_dos_setfileattr(name,attrib)!=0)
 {
 perror("\nUnable to set");
 getch();
 return 0;
 }
 getch();
}
```

SAMPLE INPUT AND OUTPUT:


C:\WINDOWS\System32\cmd.exe - tc
MAIN MENU OF FILE
1.Create
2.Rename
3.Delete
4.Edit
5.Exit
Enter your choice:1
Enter the file name with extention:eac.doc
eac.doc file created
MAIN MENU OF FILE
1.Create
2.Rename
3.Delete
4.Edit
5.Exit
Enter your choice:


C:\WINDOWS\System32\cmd.exe - tc
MAIN MENU OF FILE
1.Create
2.Rename
3.Delete
4.Edit
5.Exit
Enter your choice:2
Enter the Old file name with extention:eac.doc
Enter the New file name with extention:ea.doc
ea.doc file name changed
MAIN MENU OF FILE
1.Create
2.Rename
3.Delete
4.Edit
5.Exit
Enter your choice:


C:\WINDOWS\System32\cmd.exe - tc
MAIN MENU OF FILE
1.Create
2.Rename
3.Delete
4.Edit
5.Exit
Enter your choice:3
Enter the File name with extention:ea.doc
ea.doc file removed
MAIN MENU OF FILE
1.Create
2.Rename
3.Delete
4.Edit
5.Exit
Enter your choice:

PRACTICAL – 6

AIM: To write a C program for File Copy and Move

Copying and moving files

When copying files from one directory to another you need to know for certain which directory is the current working directory. If you are not sure, use the pwd command.

Make a copy in the current directory

cp **oldfilename newfilename**

eg

cp **file1.html file2.html**

Make a copy in a sub-directory of the current directory

cp **filename dir-name**

eg

cp **file1.html public_html**

This will make a copy of file1.html within the public_html directory (assuming the directory exists).

Move a file into a sub-directory

mv **filename dir-name**

The cp command makes a copy of the file with the new name or in the new location, and leaves the original file in place. If you use the mv command the file is moved, so the original is deleted and the only copy is the one in the new location.

Re-naming a file

The mv command can also be used to rename a file while leaving it in the same directory.

mv **oldfilename newfilename**

Copying (or moving) to the parent directory

cp **filename ..**

or

mv **filename ..**

Note the space before the two dots. The two dots represent the parent directory.

Copying (or moving) from the parent directory into the current directory

You would use the following commands to copy or move a file from the parent directory into the current directory

cp **../filename .**

or

mv **../filename .**

The dot at the end of these commands stands for the current directory. Note that there is a space in front of this final dot.

Copying a file into a sub-directory of a sub-directory

Suppose you have a sub-directory within your home directory called public_html, and within public_html you have a sub-directory called personal.

If you are currently in your home directory you can copy (or move) a file from your home directory into this sub-sub-directory with the following command:

cp **filename public_html/personal**

or

mv **filename public_html/personal**

or if you wanted not only to move the file into the different directory, but also to give it a different name you could use this command:

mv **file1.html public_html/personal/file2.html**

Program: To write a C Program to implement File Copy and Move Operations

```
#include<stdio.h>
#include<conio.h>
#include<dos.h>
#include<dir.h>

char fn2[20];

main()
{
 int c;
 clrscr();
 do{
 printf("\n\t\tMain Menu\n-------------------------------\n");
 printf("1.Copy a File\n2.Move a File\n3.Exit\n");
 scanf("%d",&c);
 switch(c)
 {
  case 1:
   copy_file();
   break;

  case 2:
   move_file();
   break;

  case 3:
   exit(0);
 }
 }while(c<=3);
 getch();
 return 0;
}



copy_file()
{
 FILE *f1,*f2;
 char ch,s[10],fn1[20];
 int a;
 printf("\nAre u see the privious files(1/0)?");
 scanf("%d",&a);
 if(a==1)
```

```
print_file();
printf("Enter the source file name:");
scanf("%s",&fn1);
printf("Enter the Destination file name:");
scanf("%s",&fn2);
f1=fopen(fn1,"r");
if(f1==NULL)
 printf("Can't open the file");
else {
 f2=fopen(fn2,"w");
 while((ch=getc(f1))!=EOF)
  putc(ch,f2);
 printf("One File Copied");
 fclose(f2);
                                }
 fclose(f1);
 return 0;
}


move_file()
{
FILE *f1,*f2;
char ch,s[10],fn1[20];
int a;
printf("\nAre u see the privious files(1/0)?");
scanf("%d",&a);
if(a==1)
print_file();
printf("Enter the source file name:");
scanf("%s",&fn1);
printf("Enter the Destination file name:");
scanf("%s",&fn2);

f1=fopen(fn1,"r");
if(f1==NULL)
 printf("Can't open the file");
else {
 f2=fopen(fn2,"w");
 while((ch=getc(f1))!=EOF)
  putc(ch,f2);
 printf("One File moved");
 fclose(f2);
 remove(fn1);
                                }
```

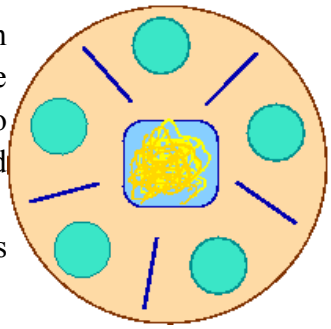
```
 fclose(f1);
 return 0;
}


print_file()
{
 struct ffblk ffblk;
 int d,p=0;
 char ch;
 d=findfirst("*.*",&ffblk,0);
 while(!d)
 {
 printf("%s\n",ffblk.ff_name);
 d=findnext(&ffblk);
 p=p+1;
 if(p>=20)
 {
  printf("Press any key to continue");
  getchar();
  p=0;
 }
}
return 0;
}
```

PRACTICAL-7

Aim: To write a C Program to implement Dining Philosophers Program

The *Dining Philosophers* problem is a classic synchronization problem introducing *semaphores* as a conceptual synchronization mechanism.

Dining Philosophers



There is a dining room containing a circular table with five chairs. At each chair is a plate, and between each plate is a single chopstick. In the middle of the table is a bowl of spaghetti. Near the room are five philosophers who spend most of their time thinking, but who occasionally get hungry and need to eat so they can think some more.
In order to eat, a philosopher must sit at the table, pick up the two chopsticks to the left and right of a plate, then serve and eat the spaghetti on the plate.
Thus, each philosopher is represented by the following pseudocode:

```
process P[i]
while true do
  { THINK;
   PICKUP(CHOPSTICK[i], CHOPSTICK[i+1 mod 5]);
   EAT;
   PUTDOWN(CHOPSTICK[i], CHOPSTICK[i+1 mod 5])
   }
```

A philosopher may THINK indefinately. Every philosopher who EATs will eventually finish. Philosophers may PICKUP and PUTDOWN their chopsticks in either order, or nondeterministically, but these are atomic actions, and, of course, two philosophers cannot use a single CHOPSTICK at the same time.

The problem is to design a protocol to satisfy the liveness condition: *any philosopher who tries to EAT, eventually does.*

Solution:
Other authors, including Dijkstra, have posed simpler solutions to the dining philosopher problem than that proposed by Tannenbaum (depending on one's notion of "simplicity," of course). One such solution is to restrict the number of philosophers allowed access to the table. If there are *N* chopsticks but only *N-1* philosophers allowed to compete for them, at least one will succeed, even if they follow a rigid sequential protocol to acquire their chopsticks.
This solution is implemented with an *integer* semaphore, initialized to *N-1*. Both this and Tannenbaum's solutions avoid *deadlock* a situation in which all of the philosophers have grabbed one chopstick and are deterministically waiting for the other, so that there is no hope of recovery. However, they may still permit *starvation*, a scenario in which at least one hungry philosopher never gets to eat.

Starvation occurs when the asynchronous semantics may allow an individual to eat repeatedly, thus keeping another from getting a chopstick. The starving philosopher runs, perhaps, but doesn't make progress. The observation of this fact leads to some further refinement of what *fairness* means. Under some notions of fairness the solutions given above can be said to be correct.

Program: To write a C program to implement Dining Philosophers Problem

```
#include<stdio.h>
#include<conio.h>
#include<dos.h>
#include<dir.h>

char fn2[20];

main()
{
 int c;
 clrscr();
 do{
 printf("\n\t\tMain Menu\n-------------------------------\n");
 printf("1.Copy a File\n2.Move a File\n3.Exit\n");
 scanf("%d",&c);
 switch(c)
 {
  case 1:
   copy_file();
   break;

  case 2:
   move_file();
   break;

  case 3:
   exit(0);
 }
 }while(c<=3);
 getch();
 return 0;
}



copy_file()
{
 FILE *f1,*f2;
 char ch,s[10],fn1[20];
 int a;
 printf("\nAre u see the privious files(1/0)?");
 scanf("%d",&a);
 if(a==1)
```

```
print_file();
printf("Enter the source file name:");
scanf("%s",&fn1);
printf("Enter the Destination file name:");
scanf("%s",&fn2);
f1=fopen(fn1,"r");
if(f1==NULL)
 printf("Can't open the file");
else {
 f2=fopen(fn2,"w");
 while((ch=getc(f1))!=EOF)
  putc(ch,f2);
 printf("One File Copied");
 fclose(f2);
                                }
 fclose(f1);
 return 0;
}


move_file()
{
FILE *f1,*f2;
char ch,s[10],fn1[20];
int a;
printf("\nAre u see the privious files(1/0)?");
scanf("%d",&a);
if(a==1)
print_file();
printf("Enter the source file name:");
scanf("%s",&fn1);
printf("Enter the Destination file name:");
scanf("%s",&fn2);

f1=fopen(fn1,"r");
if(f1==NULL)
 printf("Can't open the file");
else {
 f2=fopen(fn2,"w");
 while((ch=getc(f1))!=EOF)
  putc(ch,f2);
 printf("One File moved");
 fclose(f2);
 remove(fn1);
                                }
```

```
 fclose(f1);
 return 0;
}


print_file()
{
 struct ffblk ffblk;
 int d,p=0;
 char ch;
 d=findfirst("*.*",&ffblk,0);
 while(!d)
 {
 printf("%s\n",ffblk.ff_name);
 d=findnext(&ffblk);
 p=p+1;
 if(p>=20)
 {
  printf("Press any key to continue");
  getchar();
  p=0;
 }
}
return 0;
}
```

SAMPLE INPUT AND OUTPUT:

```
Philos name     Right fork     Left fork
-----------------------------------------------------------------
      0          0          1
      1          1          2
      2          2          3
      3          3          4
      4          4          0
-----------------------------------------------------------------
Enter the Two Eating Philosophers number:1 3
Round 1
-------------------------
Philosopherre 1 is eating with rhf=1 and  lhf=2.
Philosopherre 3 is eating with rhf=3 and lhf=4.
Round  2
------------------------
Philosopherre 0 is eating with rhf=0 and  lhf=1.
Philosopherre 2 is eating with rhf=2 and   lhf=3.
Round  4
------------------------
Philosopherre 2 is eating with rhf=2 and  lhf=3.
Philosopherre 4 is eating with rhf=4 and   lhf=0.
```

```
      Philos name      Right fork      Left fork
-----------------------------------------------------------------
      0          0          1
      1          1          2
      2          2          3
      3          3          4
      4          4          0
-----------------------------------------------------------------
Enter the Two Eating Philosophers number:1 4
Round 1
-------------------------
Philosopherre 1 is eating with rhf=1 and  lhf=2.
Philosopherre 4 is eating with rhf=4 and lhf=0.
Round  2
------------------------
Philosopherre 0 is eating with rhf=0 and  lhf=1.
Philosopherre 2 is eating with rhf=2 and   lhf=3.
Round  4
------------------------
Philosopherre 2 is eating with rhf=2 and  lhf=3.
Philosopherre 4 is eating with rhf=4 and   lhf=0.
```

PRACTICAL – 8

Aim: To write a C Program to implement Producer Consumer Problem

DESCRIPTION

This is an illustration of a solution to the classic producer-consumer (bounded-buffer) problem using semaphores.

CONCEPT: Producers produce items to be stored in the buffer. Consumers remove and consume items which have been stored. Mutual exclusion must be enforced on the buffer itself. Moreover, producers can store only when there is an empty slot, and consumers can remove only when there is a full slot. Three semaphores are used. The binary semaphore mutex controls access to the buffer itself.

The counting semaphore empty keeps track of empty slots, and the counting semaphore full keeps track of full slots. In this example, the buffer is implemented as an array of size MAX treated as a circular (ring) buffer. Variables in and out give the index of the next position for putting in and taking out (if any). Variable count gives the number of items in the buffer.

Algorithm:

1. Start the program.
2. Declare the variables in the type of pthread_t as tid_produce tid_consume.
3. Declare a structure for semaphore variables.
4. During run time read the number of items to be produced and consumed.
5. Declare and define semaphore function for creation and destroy.
6. Define producer function.
7. Define consumer function.
8. Call producer and consumer function.
9. Stop the execution.

Program: To write a C Program to implement Producer Consumer Problem

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

void main()
{
 int ch,n,c1=0,c2=0,produce[23],consume[23];
 clrscr();
 printf("\n\n\n\n\n\t\n\n\t\t\tEnter Stack Size :  ",n);
 scanf("%d",&n);
 while(1)
 {
  clrscr();
  printf("\t\tProducer Stack (Stack Size : %d
                    )\n\t\t~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~",n);
  display(c1,produce);
  printf("\n\n\t\tConsumer Stack (Stack Size : %d
                    )\n\t\t~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~",n);
  display(c2,consume);
  printf("\n\t\tCHOICES\n\t\t~~~~~~~\n\t1.Producer\n\t2.Consumer\n\t3.
                                        Exit\nEnter your choice :  ");
  scanf("%d",&ch);
  switch(ch)
  {
   case 1:
          if(c1==n)
           printf("Produer stack is FULL.So Producer goes to SLEEP\n");
          else
          {
          c1++;
          printf("\t\tEnter PRODUCE item is :");
          scanf("%d",&produce[c1]);
          }
          break;

   case 2:
          if(c2==n)
          printf("Consumer Stack is FULL.So it goes to SLEEP!..........\n\tReset the Cosumer
          Stack\n",c2=0);
          else if(c1==0)
            printf("\tProducer stack is EMPTY\n");
          else
           {
           c2++;
```

```
            consume[c2]=produce[c1];
            printf("\t\tCONSUME one item");
            c1--;
            }
           break;

   case 3:
           exit(0);

   default:
           printf("\tIt is Wrong choice,Please enter correct choice!.....\n");
   }
  getch();
 }
}

display(int c,int stack[])
{
 int i;
 printf("\n---------------------------------------------------------------------\n");
 if(c==0)
 printf("\tStack is EMPTY\n\t\t(Now It is sleeping)");
 else
 for(i=1;i<=c;i++)
 printf("\t%d",stack[i]);
 printf("\n---------------------------------------------------------------------------\n");
 }
```

SAMPLE INPUT AND OUTPUT:

```
Enter Stack Size :  4

  Producer Stack (Stack Size : 4)
            ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
---------------------------------------------------------------------------------
      Stack is EMPTY
            (Now It is sleeping)
--------------------------------------------------------------------------- -----

            Consumer Stack (Stack Size : 4)
            ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
---------------------------------------------------------------------------------
      Stack is EMPTY
            (Now It is sleeping)
--------------------------------------------------------------------------- -----
            CHOICES
            ~~~~~~~
      1.Producer
      2.Consumer
      3.Exit
      Enter your choice :  2
          Producer stack is EMPTY
            Producer Stack (Stack Size : 4)
            ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
---------------------------------------------------------------------------------
      10
--------------------------------------------------------------------------- -----

            Consumer Stack (Stack Size : 4)
            ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
---------------------------------------------------------------------------------
      Stack is EMPTY
            (Now It is sleeping)
--------------------------------------------------------------------------- -----
            CHOICES
            ~~~~~~~
      1.Producer
      2.Consumer
      3.Exit
Enter your choice : 1
                  Enter PRODUCE item is :30
            Producer Stack (Stack Size : 4)
            ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
-----------------------------------------------------------------------------
        10      30
------------------------------------------------------------------------ -----

                Consumer Stack (Stack Size : 4)
                ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
-----------------------------------------------------------------------------
        Stack is EMPTY
                (Now It is sleeping)
------------------------------------------------------------------------ -----
                CHOICES
                ~~~~~~~
        1.Producer
        2.Consumer
        3.Exit
Enter your choice :  2
                Producer Stack (Stack Size : 4)
                ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
-----------------------------------------------------------------------------
        10
------------------------------------------------------------------------ -----

                Consumer Stack (Stack Size : 4)
                ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
-----------------------------------------------------------------------------
        30
------------------------------------------------------------------------ -----
                CHOICES
                ~~~~~~~
        1.Producer
        2.Consumer
        3.Exit
Enter your choice :3
```

PRACTICAL-9

Aim: To write a C program for the following Job Scheduling Algorithms:

1. **First Come First Serve Algorithm**
2. **Shortest Job First Scheduling Algorithm**
3. **Round Robin Scheduling Algorithm**
4. **Priority Scheduling Algorithm**

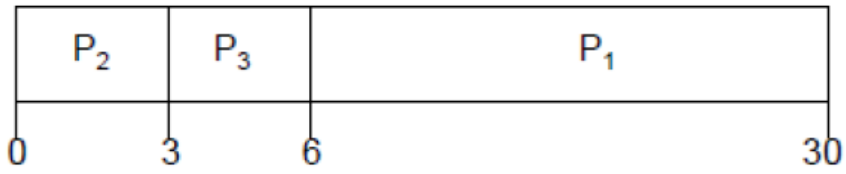
DESCRIPTION

First-Come, First-Served (FCFS) Scheduling

Suppose that the processes arrive in the order

P_2 , P_3 , P_1

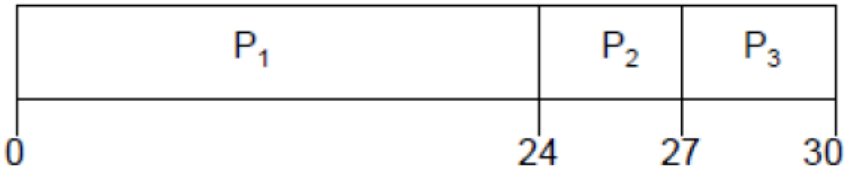
- The Gantt chart for the schedule is:



- Waiting time for P_1 = 6; P_2 = 0; P_3 =.3
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case
- *Convoy effect* short process behind long process

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:



- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

Algorithm:

1. Start the program.
2. Get the number of processes and their burst time.
3. Initialize the waiting time for process 1 is 0.
4. The waiting time and turn-around time for other processes are calculated as
p[i].wt=p[i-1].bt+p[i-1].wt;
p[i].tt=p[i].bt+p[i].wt;
5. The waiting time and turn-around time for all the processes are summed and then the average waiting time and turn-around time are calculated.
6. The average waiting time and turn-around time are displayed.
7. Stop the program.

Program: To implement FCFS job Scheduling Algorithm

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

void main()
{
 int n,b[10],w[10],i,j,h;
 int stime[10];
 float avg=0;
 clrscr();
 printf("\n\tJOB SCHEDULING ALGORITHM[FCFS]");
 printf("\n\t**********************************************
                                                    ****\n");
 printf("\nEnter howmany jobs:");
 scanf("%d",&n);

 printf("\nEnter burst time for corresponding job....\n");
 for(i=0;i<n;i++)
 {
  printf("\nProcess %d:",i+1);
  scanf("%d",&b[i]);
 }

 w[0]=0;

 printf("\nprocess 1 waiting time is 0");
 for(i=1;i<n;i++)
 {
  w[i]=b[i-1]+w[i-1];
  printf("\nProcess %d waiting time: %d",i+1,w[i]);
  avg+=w[i];
 }

 printf("\ntotal waiting time:%f",avg);

 printf("\n\nthe average waiting time is:%f\n",avg/n);
 printf("\nGant
          Chart\n**************************************\n\n\t");
 h=22;

 for(i=0;i<n;i++)
 {
  printf("%d",b[i]);
```

```
  for(j=1;j<=b[i];j++)
   printf("%c",h);
 }
 printf("\n\t");
 for(i=0;i<n;i++)
 {
  printf("%d",w[i]);
  for(j=1;j<=w[i];j++)
   printf("%c",h);
  delay(1000);
 }
getch();
}
```

SAMPLE INPUT AND OUTPUT:

Enter howmany jobs:5

Enter burst time for corresponding job....

Process 1:4

Process 2:2

Process 3:8

Process 4:2

Process 5:9

process 1 waiting time is 0
Process 2 waiting time: 4
Process 3 waiting time: 6
Process 4 waiting time: 14
Process 5 waiting time: 16
total waiting time:40.000000

the average waiting time is:8.000000

Gaunt Chart
****

4——2——8——2——9

04——6——14——16——

IMPLEMENTATION OF SJF SCHEDULING

DESCRIPTION

Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time

Two schemes:

Non Preemptive: once CPU given to the process it cannot be preempted until completes its CPU burst

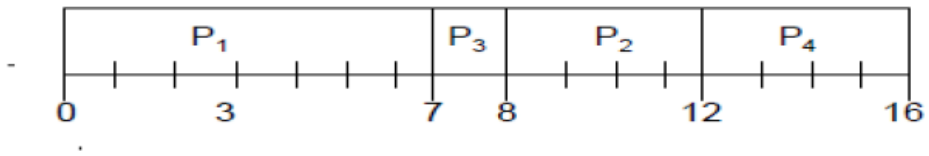
Preemptive: if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is known as the Shortest-Remaining-Time-First (SRTF)

SJF is optimal: gives minimum average waiting time for a given set of processes

Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

SJF (non-preemptive)

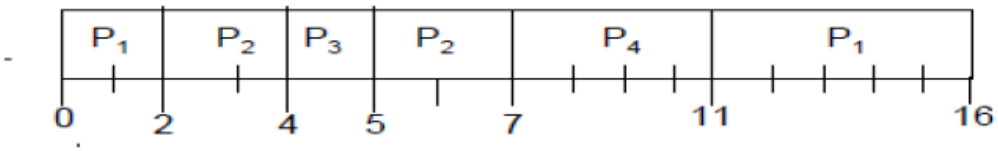


Average waiting time = (0 + 6 + 3 + 7)/4 = 4

Example of Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (preemptive)



- Average waiting time = (9 + 1 + 0 +2)/4 = 3

Algorithm:

1. Start the program.
2. Get the number of processes and their burst time.
3. Initialize the waiting time for process 1 is 0.
4. The processes are stored according to their burst time.
5. The waiting time and turn-around time for other processes are calculated as
p[i].wt=p[i-1].bt+p[i-1].wt;
p[i].tt=p[i].bt+p[i].wt;
6. The waiting time and turn-around time for all the processes are summed and then the average waiting time and turn-around time are calculated.
7. The average waiting time and turn-around time are displayed.
8. Stop the program.

Program: To implement Shortest Job First Scheduling Algorithm

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

void main()
{
 int n,b[10],w[10],i,j,h,t,tt;
 int stime[10],a[10];
 float avg=0;
 clrscr();
 printf("\n\tJOB SCHEDULING ALGORITHM[SJF]");
 printf("\n\t*************************************************\n");
 printf("\nEnter howmany jobs:");
 scanf("%d",&n);
 printf("\nEnter burst time for corresponding job....\n");
          for(i=1;i<=n;i++)
          {
           printf("\nProcess %d:",i);
           scanf("%d",&b[i]);  a[i]=i;
          }
                          for(i=1;i<=n;i++)
                                  for(j=i;j<=n;j++)
                                    if(b[i]>b[j])
                                    {
                                    t=b[i]; tt=a[i];
                                    b[i]=b[j];  a[i]=a[j];
                                    b[j]=t;     a[j]=tt;
                                    }

w[1]=0;


printf("\nprocess %d waiting time is 0",a[1]);

for(i=2;i<=n;i++)
{
 w[i]=b[i-1]+w[i-1];
 printf("\nProcess %d waiting time: %d",a[i],w[i]);
 avg+=w[i];
}

printf("\ntotal waiting time:%f",avg);
printf("\n\nthe average waiting time is:%f\n",avg/n);
```

```
 printf("\nGaunt Chart\n**************************************\n\n\t");

 h=22;

 for(i=1;i<=n;i++)
 {
  printf("%d",b[i]);
  for(j=1;j<=b[i];j++)
   printf("%c",h);
 }
 printf("\n\t");

 for(i=1;i<=n;i++)
 {
  printf("%d",w[i]);
  for(j=1;j<=w[i];j++)
   printf("%c",h);
  delay(1000);
 }

getch();
}
```

SAMPLE INPUT AND OUTPUT:

Enter howmany jobs:3

Enter burst time for corresponding job....

Process 1:5

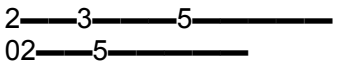
Process 2:2

Process 3:3

process 2 waiting time is 0
Process 3 waiting time: 2
Process 1 waiting time: 5
total waiting time:7.000000

the average waiting time is:2.333333

Gaunt Chart
****



IMPLEMENTATION OF PRIORITY SCHEDULING

Aim:

To write the program to perform priority scheduling.

DESCRIPTION

A priority number (integer) is associated with each process
The CPU is allocated to the process with the highest priority (smallest integer => highest priority)
- Preemptive
- non preemptive

SJF is a priority scheduling where priority is the predicted next CPU burst time
Problem Starvation – low priority processes may never execute
Solution Aging – as time progresses increase the priority of the process

Algorithm:

1. Start the program.
2. Get the number of processes, their burst time and priority.
3. Initialize the waiting time for process 1 is 0.
4. Based upon the priority processes are arranged.
5. The waiting time and turn around time for other processes are calculated as
p[i].wt=p[i-1].bt+p[i-1].wt;
p[i].tt=p[i].bt+p[i].wt;
6. The waiting time and turn around time for all the processes are summed and then the average
waiting time and turn around time are calculated.
7. The average waiting time and turn around time are displayed.
8. Stop the program.

Program: To implement Priority Scheduling Algorithm

```
//priority scheduling
#include<stdio.h>
#include<conio.h>
#include<dos.h>

void main()
{
 int n,b[10],w[10],i,j,h,t,tt;
 int stime[10],a[10],p[10];
 float avg=0;
 clrscr();
 printf("\n\tPRIORITY SCHEDULING ALGORITHM");
 printf("\n\t**************************************************\n");
 printf("\nEnter howmany jobs:");
 scanf("%d",&n);
 printf("\nEnter burst time & priority for corresponding job....\n");
 for(i=1;i<=n;i++)
 {
  printf("\nProcess %d:",i);
  scanf("%d %d",&b[i],&p[i]);  a[i]=i;
 }
                                for(i=1;i<=n;i++)
                                        for(j=i;j<=n;j++)
                                          if(p[i]<p[j])
                                          {
                                          t=b[i]; tt=a[i];
                                          b[i]=b[j]; a[i]=a[j];
                                          b[j]=t;     a[j]=tt;
                                          }

 w[1]=0;


 printf("\nprocess %d waiting time is 0",a[1]);
 for(i=2;i<=n;i++)
 {
  w[i]=b[i-1]+w[i-1];
  printf("\nProcess %d waiting time: %d",a[i],w[i]);
  avg+=w[i];
 }
 printf("\ntotal waiting time:%f",avg);
 printf("\n\nthe average waiting time is:%f\n",avg/n);
 printf("\nGant Chart\n**************************************\n\n\t");
```

```
 h=22;

 for(i=1;i<=n;i++)
 {
  printf("%d",b[i]);
  for(j=1;j<=b[i];j++)
   printf("%c",h);
 }
 printf("\n\t");
 for(i=1;i<=n;i++)
 {
  printf("%d",w[i]);
  for(j=1;j<=w[i];j++)
   printf("%c",h);
  delay(1000);
 }

getch();
}
```

SAMPLE INPUT AND OUTPUT:

Enter howmany jobs:3

Enter burst time & priority for corresponding job....

Process 1:5 2

Process 2:7 1

Process 3:6 3

PRIORITY SCHEDULING ALGORITHM
****

Enter howmany jobs:3

Enter burst time & priority for corresponding job....

Process 1:5 2

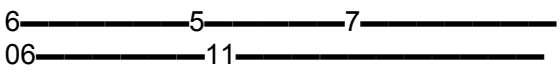
Process 2:7 1

Process 3:6 3

process 3 waiting time is 0
Process 1 waiting time: 6
Process 2 waiting time: 11
total waiting time:17.000000

the average waiting time is:5.666667

Gant Chart
****



IMPLEMENTATION OF ROUND ROBIN SCHEDULING

Aim: To write a program for Round Robin Scheduling.

DESCRIPTION

Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue. If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once. No process waits more than (*n*-1)*q* time units.

Performance

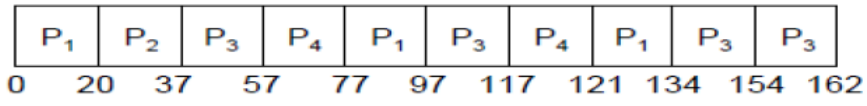
q large: FIFO

q small: *q* must be large with respect to context switch, otherwise overhead is too high

Example of RR with Time Quantum = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

The Gantt chart is:



Typically, higher average turnaround than SJF, but better *response*

Algorithm:

1. Start the program.
2. Get the number of processes, their burst time and time slice for each process.
3. Calculate the total burst time.
4. To check whether the burst time of process is less then the time slice, the required time only allotted to the process.
5. If the burst time of process is greater than the time slice one time slice is allotted for the process and burst time is subtracted by one time slice.
6. Step 4 and 5 is repeated up to process burst time is 0.
7. The waiting time and turn around time for all the processes are summed and then the average
waiting time and turn around time are calculated.
8. The average waiting time and turn around time are displayed.
9. Stop the program.

Program: To implement Round Robin Algorithm

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

int z[10],b[10],n,m[50],r,q,e=0,avg=0,i,j;
float f;

main()
{
 clrscr();
 printf("\n\tJOB SCHEDULING ALGORITHM[RR]");
 printf("\n\t*****************************************************\n");
 printf("\nEnter how many jobs:");
 scanf("%d",&n);
 printf("\nEnter burst time for corresponding job...\n");
 for(i=1;i<=n;i++)
 {
  printf("\nProcess %d: ",i);
  scanf("%d",&b[i]); z[i]=b[i];
 }

 printf("\nENTER THE TIME SLICE VALUE:");
 scanf("%d",&q);

 rr();

 average();

 getch();
 return 0;
}

rr()
{
 int max=0;
 max=b[1];
                                for(j=1;j<=n;j++)
                                  if(max<=b[j])
                                max=b[j];

 if((max%q)==0)
   r=(max/q);
 else
  r=(max/q)+1;
```

```
  for(i=1;i<=r;i++)
  {
    printf("\nround %d",i);
    for(j=1;j<=n;j++)
                                {
                                 if(b[j]>0)
                                   {
                                    b[j]=b[j]-q;

                                    if(b[j]<=0)
                                      {
                                      b[j]=0;
                                    printf("\nprocess %d is completed",j);
                                      }
                                    else
                                    printf("\nprocess %d remaining time is %d",j,b[j]);
                                   }
                                 }
                                 delay(1000);
  }
  return 0;
}


average()
{
 for(i=1;i<=n;i++)
   {
    e=0;
    for(j=1;j<=r;j++)
     {
                               if(z[i]!=0)
                               {
                                if(z[i]>=q)
                                {
                                m[i+e]=q;  z[i]-=q;
                                }
                                else
                                {
                                m[i+e]=z[i];  z[i]=0;
                                }
                               }
                               else
                                m[i+e]=0;
      e=e+n;
```

```
    }
   }
  for(i=2;i<=n;i++)
     for(j=1;j<=i-1;j++)
      avg=avg+m[j];
  for(i=n+1;i<=r*n;i++)
  {
    if(m[i]!=0)
    {
      for(j=i-(n-1);j<=i-1;j++)
      avg=m[j]+avg;
    }
  }
  f=avg/n;
  printf("\nTOTAL WATING:%d",avg);
  printf("\n\nAVERAGE WAITING TIME:%f\n",f);
                                  for(i=1;i<=r*n;i++)
                                   { if(m[i]!=0)
                                     if(i%n==0){
                                     printf("P%d",(i%n)+(n));      }

                                     else
                                     printf("P%d",(i%n));
                                     for(j=1;j<=m[i];j++)
                                        printf("%c",22);
                                    }
printf("\n");
 getch();
 return 0;
}
```

SAMPLE INPUT AND OUTPUT:

```
Enter how many jobs:4
Enter burst time for corresponding job...
Process 1: 7
Process 2: 5
Process 3: 4
Process 4: 2
ENTER THE TIME SLICE VALUE:2
round 1
process 1 remaining time is 5
process 2 remaining time is 3
process 3 remaining time is 2
process 4 is completed
round 2
process 1 remaining time is 3
process 2 remaining time is 1
process 3 is completed
round 3
process 1 remaining time is 1
process 2 is completed
round 4
process 1 is completed
TOTAL WATING:39
AVERAGE WAITING TIME:9.000000
P1——P2——P3——P4——P1——P2——P3——P1——P2—P1—
```

PROGRAM – 10

Aim: To implement the following memory management schemes:

1. **First Fit Algorithm**
2. **Best Fit Algorithm**

DESCRIPTION

Memory Management Algorithms

In an environment that supports dynamic memory allocation, the memory manager must keep a record of the usage of each allocable block of memory. This record could be kept by using almost any data structure that implements linked lists. An obvious implementation is to define a free list of block descriptors, with each descriptor containing a pointer to the next descriptor, a pointer to the block, and the length of the block. The memory manager keeps a free list pointer and inserts entries into the list in some order conducive to its allocation strategy. A number of strategies are used to allocate space to the processes that are competing for memory.

First Fit

Another strategy is first fit, which simply scans the free list until a large enough hole is found.

Despite the name, first-fit is generally better than best-fit because it leads to less fragmentation. Small holes tend to accumulate near the beginning of the free list, making the memory allocator search farther and farther each time. Solution: Next Fit

Best Fit

The allocator places a process in the smallest block of unallocated memory in which it will fit. It requires an expensive search of the entire free list to find the best hole. More importantly, it leads to the creation of lots of little holes that are not big enough to satisfy any requests. This situation is called *fragmentation*, and is a problem for all memory-management strategies, although it is particularly bad for best-fit. One way to avoid making little holes is to give the client a bigger block than it asked for. For example, we might round all requests up to the next larger multiple of 64 bytes.

That doesn't make the fragmentation go away, it just hides it.

· Unusable space in the form of holes is called *external fragmentation*

· Unusable space in the form of holes is called *external fragmentation*

Algorithm:

1. Start the program.
2. Get the number of segments and size.
3. Get the memory requirement and select the option.
4. If the option is ‘2’ call first fit function.
5. If the option is ‘1’ call best fit function.
6. Otherwise exit.
7. For first fit, allocate the process to first possible segment which is free.

8. For best fit, do the following steps.
a. Sorts the segments according to their sizes.
b. Allocate the process to the segment which is equal to or slightly greater than the process size.
9. Stop the program.

Program: To write a C program to implement First Fit and Best Fit Memory Management Algorithms

```
#include<stdio.h>
#include<conio.h>
#define MAXSIZE 25

void printlayout(int[],int );
int firstfit(int[],int, int);
int bestfit(int[],int, int);

void main()
{
int i,a[25],n,req,choice,pos,ch;
clrscr();
printf("How MAny Segments");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("Segent Size");
scanf("%d",&a[i]);
}
loop:
printf("How Much Is Your Memory Requirment");
scanf("%d",&req);
printf("\n1.Bestfit\n2.Firstfit\n3.Exit\n");
printf("Enter Your Choice\n");
scanf("%d",&choice);
switch(choice)
{
case 1:
pos=bestfit(a,n,req);
break;
case 2:
pos=firstfit(a,n,req);
break;
}
printf("\tBestfit and Firstfit Algorithm\n");
printf("\t_____\n\n");
printlayout(a,n);
printf("Your Memory Requirment is :%d\n\n",req);
printf("Alloted Memory Region is:%d\n\n",a[pos]);
a[pos]=0;
printf("Do You Want To Continue 1/0");
scanf("%d",&ch);
```

```
if(ch==1)
goto loop;
getch();
}
void printlayout(int a[],int n)
{
int i,j;
printf("\t\tMemory Free List");
printf("\n\t\t_____\n\n");
printf("\t\t|~~~|\n");
for(i=0;i<n;i++)
{
if(a[i]!=0)
{
for(j=1;j<=(a[i]/100);j++)
printf("\t\t| |\n");
printf("\t\t| %d|\n",a[i]);
printf("\t\t|---|\n");
}}
printf("\n\n");
}

int firstfit(int a[],int n,int r)
{
int i;
for(i=0;i<n;i++)
if(a[i]>=r)
break;
return i;
}

int bestfit(int a[],int n,int r)
{
int b[25],i,j,temp,val;
for(i=0;i<n;i++)
b[i]=a[i];
for(i=0;i<n-1;i++)
for(j=i;j<n-1;j++)
if(b[i]>b[j])
{
temp=b[i];
b[i]=b[j];
b[j]=temp;
}
for(i=0;i<n;i++)
```

```
if(b[i]>=r)
break;
val=b[i];
for(i=0;i<n;i++)
if(a[i]==val)
break;
return i;
}
```

SAMPLE INPUT AND OUTPUT:

How Many Segments4
Segent Size500
Segent Size200
Segent Size300
Segent Size400
How Much Is Your Memory Requirment200
1.Bestfit
2.Firstfit
3.Exit
Enter Your Choice 1
Bestfit and Firstfit Algorithm

Memory Free List

|~~~|
| |
| |
| |
| |
500
200
300
72
| --- |
| |
| |
| |
| |
400
Your Memory Requirment is :200
Alloted Memory Region is:200
Do You Want To Continue 1/0
1
How Much Is Your Memory Requirment250
1.Bestfit

2.Firstfit
3.Exit
Enter Your Choice
1
Bestfit and Firstfit Algorithm

Memory Free List

|~~~|
| |
| |
| |
|500|
73
| --- |
| |
| |
300
400
Your Memory Requirment is :250
Alloted Memory Region is:300
Do You Want To Continue 1/0
0

PROGRAM-11

Aim: To implement Page replacement algorithms:

1. **First In First Out (FIFO)**
2. **Least Recently Used (LRU)**

DESCRIPTION

Page replacement algorithms are used to decide what pages to page out when a page needs to be allocated. This happens when a page fault occurs and free page cannot be used to satisfy allocation

First In First Out (FIFO):

The frames are empty in the beginning and initially no page fault occurs so it is set to minus one. When a page fault occurs the page reference string is brought into memory. The operating system keeps track of all the pages in memory, herby keeping track of the most recently arrived and the oldest one. If the page in the page reference string is not in memory, the page fault is incremented and the oldest page is replaced. If the page in the page reference string is in the memory, take the next page without calculating the page fault. Take the next page in the page reference string and check if the page is already present in the memory or not. Repeat the process until all the pages are referred and calculate the page fault for all those pages in the page reference string for the number of available frames.

Least Recently Used (LRU) :

A good approximation to the optimal algorithm is based on the observation that pages that have been heavily used in the last few instructions will probably be heavily used again in the next few.

Conversely, pages that have not been used for ages will probably remain unused for a long time. This idea suggests a realizable algorithm: when a page fault occurs, throw out the page that has been unused for the longest time. This strategy is called LRU (Least Recently Used) paging.

Algorithm:

1. Start the program
2. Obtain the number of sequences, number of frames and sequence string from the user
3. Now when a page is not in the frame comes, increment the number of page fault and remove
the page that come in the first in FIFO algorithm
4. In LRU algorithm, when a page fault occurs, the page which most recently used is removed
5. Display the number of faults
6. Stop the program

Program: To write a C Program to Implement FIFO and LRU Page Replacement Algorithms

```
#include<stdio.h>

int m,n,i,j,k,flag,count=0,refer[100],page_frame[100][2],fault=0,min,no_frames;

void replace(int z)
{
for(i=0;i<n;i++)
{
flag=1;
for(j=0;j<no_frames;j++)
if(refer[i]==page_frame[j][0])
{
m=j;
flag=0;
}
if(flag)
{
fault++;
min=32000;
for(j=0;j<no_frames;j++)
if(page_frame[j][1]<min)
{
min=page_frame[j][1];
k=j;
}
page_frame[k][0]=refer[i];
page_frame[k][1]=++count;
for(j=0;j<no_frames;j++)
printf("%d",page_frame[j][0]);
printf("\n");
}
else
{
printf("no page fault\n");
if(z==2)
page_frame[m][1]=++count;
}
}
printf("number of page fault is:%d\n",fault);
}

int main()
{
printf("\nEnter the number of reference:");
scanf("%d",&n);
printf("\nEnter the number of frames:");
scanf("%d",&no_frames);
printf("\nEnter the reference string:");
```

```
for(i=0;i<n;i++)
scanf("%d",&refer[i]);
printf("\t\t\tFIFO ALGORITHM \n");
for(i=0;i<no_frames;i++)
{
page_frame[i][0]=-1;
page_frame[i][1]=count;
}
replace(1);
fault=0;
count=0;
printf("\t\t\tLRU ALGORITHM \n");
for(i=0;i<no_frames;i++)
{
page_frame[i][0]=-1;
page_frame[i][1]=count;
}

replace(2);
return 0;
getch();
}
```

SAMPLE INPUT AND OUTPUT:

```
Enter the number of reference:10
Enter the number of frames:3
Enter the reference string:7 0 1 2 0 3 0 4 2 6
FIFO ALGORITHM
7 -1 -1
7 0 -1
7 0 1
2 0 1
no page fault
2 3 1
2 3 0
4 3 0
4 2 0
4 2 6
number of page fault is:9
LRU ALGORITHM
7 -1 -1
7 0 -1
7 0 1
2 0 1
no page fault
2 0 3
no page fault
4 0 3
4 0 2
4 6 2
number of page fault is:8
```

PRACTICAL 12

AIM: Write a shell program to perform operations using CASE statement such as addition, subtraction, multiplication and division.

PROGRAM:

```
echo n1 n2
read n1 n2
echo a=add
echo b=sub
echo c=mul
echo d=div
echo ch
read ch
case $ch in
    a) let z=$n1+$n2
       echo add=$z
       ;;
    b) let z=$n1-$n2
       echo sub=$z
       ;;
    c) let z=$n1*$n2
       echo mul=$z
       ;;
    d) let z=$n1/$n2
       echo div=$z
       ;;
      *)
      echo invalid option
      ;;
      Esac
```

OUTPUT

```
n1 n2
3   4
a=add
b=sub
c=mul
d=div
ch
a
add=7
```

PRACTICAL 13

AIM: Write a shell program to find largest of three numbers.

PROGRAM:

```
echo enter1
read n1
echo enter2
read n2
echo enter3
read n3
if[$n1-gt $n2]&&[$n1-gt $n3]
then echo $n1 is larger
elif[$n2-gt $n1]&&[$n2-gt $n3]
then echo $n2 is larger
else echo $n3 is larger
fi
```

OUTPUT

```
n1   n2   n3
12   36   48
48 is larger
```

PRACTICAL 14

AIM: Write a shell program to find average of two numbers.

PROGRAM:

```
echo Enter the 1st number
read a
echo Enter the 2nd number
read b
c=’expr $a+$b’
d=$c/2
echo The average of two numbers is $d
```

OUTPUT

Enter the 1st number
6
Enter the 2nd number
4
The average of two numbers is 5