INTRODUCTION TO SHELL PROGRAMMING

A **Shell** provides you with an interface to the Unix system. It gathers input from you and executes programs based on that input. When a program finishes executing, it displays that program's output.

Shell is an environment in which we can run our commands, programs, and shell scripts. There are different flavors of a shell, just as there are different flavors of operating systems. Each flavor of shell has its own set of recognized commands and functions.

Shell Prompt

The prompt, **$**, which is called the **command prompt**, is issued by the shell. While the prompt is displayed, you can type a command.

Shell reads your input after you press **Enter**. It determines the command you want executed by looking at the first word of your input. A word is an unbroken set of characters. Spaces and tabs separate words.

Following is a simple example of the **date** command, which displays the current date and time −

```
$date
Thu Jun 25 08:30:19 MST 2009
```

You can customize your command prompt using the environment variable PS1 explained in the Environment tutorial.

Shell Types

In Unix, there are two major types of shells −

- **Bourne shell** − If you are using a Bourne-type shell, the **$** character is the default prompt.
- **C shell** − If you are using a C-type shell, the % character is the default prompt.

The Bourne Shell has the following subcategories −

- Bourne shell (sh)
- Korn shell (ksh)
- Bourne Again shell (bash)
- POSIX shell (sh)

The different C-type shells follow −

- C shell (csh)

- TENEX/TOPS C shell (tcsh)

The original Unix shell was written in the mid-1970s by Stephen R. Bourne while he was at the AT&T Bell Labs in New Jersey.

Bourne shell was the first shell to appear on Unix systems, thus it is referred to as "the shell".

Bourne shell is usually installed as **/bin/sh** on most versions of Unix. For this reason, it is the shell of choice for writing scripts that can be used on different versions of Unix.

In this chapter, we are going to cover most of the Shell concepts that are based on the Borne Shell.

Shell Scripts

The basic concept of a shell script is a list of commands, which are listed in the order of execution. A good shell script will have comments, preceded by # sign, describing the steps.

There are conditional tests, such as value A is greater than value B, loops allowing us to go through massive amounts of data, files to read and store data, and variables to read and store data, and the script may include functions

In Unix-like operating systems, the **chmod** command is used to change the access mode of a file. The name is an abbreviation of **change mode**.

Syntax :

chmod [reference][operator][mode] file...

The references are used to distinguish the users to whom the permissions apply i.e. they are list of letters that specifies whom to give permissions. The references are represented by one or more of the following letters:

Reference	Class	Description
u	owner	file's owner
g	group	users who are members ofthe file's group
o	others	users who are neither the file's owner nor members of the file's group
a	all	All three of the above, same as ugo

The operator is used to specify how the modes of a file should be adjusted. The following operators are accepted:

Operator	Description
+	Adds the specified modes to the specified classes
-	Removes the specified modes from the specified classes
=	The modes specified are to be made the exact modes for the specified classes

The modes indicate which permissions are to be granted or removed from the specified classes. There are three basic modes which correspond to the basic permissions:

r	Permission to read the file.
w	Permission to write (or delete) the file.
x	Permission to execute the file, or, in the case of a directory, search it.

Types of permissions which we will be changing using chmod command :
In linux terminal, to see all the permissions to different files, type ls -l command which lists the files in the working directory in long format

EX.NO:1 BASIC COMMANDS IN LINUX

AIM:

To Study the basic commands in Linux.

COMMANDS:

1. Task : To display system date and time.

   Syntax: Date

   Explanation: This command displays the current date and time on the screen.

2. Task: To display current month.

   Syntax: Date +%m

   Explanation: This command displays the current month on the screen.

3. Task : To display the name of the current month.

   Syntax: Date +%h

   Explanation: This command displays name of the current month on the screen.

4. Task: To display current system date.

   Syntax: Date +%d

   Explanation: This command displays the current system date on the screen.

5. Task: To display current year.

   Syntax: Date +%y

   Explanation: This command displays the current year on the screen.

6. Task: To display current system time.

   Syntax: Date +%H

   Explanation: This command displays the current system time on the screen.

7. Task: To display current system time in minutes.

   Syntax: Date +%m

   Explanation: This command displays the current system time in minutes on the screen.

8. Task: To display current system time in seconds.

   Syntax: Date +%s

   Explanation: This command displays the current system time in seconds on the screen.

9. Task: To display the calendar of current month.

   Syntax: cal

Explanation: This command displays the current month calender on the screen.

10. Task: To display user defined message.

    Syntax: echo ‘message’

    Explanation: This command displays the message.

SHELL PROGRAMMING

Ex.No.2a EVEN OR ODD

AIM:

To write a program to find whether a number is even or odd.

ALGORITHM:

STEP 1: Read the input number.

STEP 2: Perform modular division on input number by 2.

STEP 3: If remainder is 0 print the number is even.

STEP 4: Else print number is odd.

STEP 5: Stop the program.

PROGRAM

```
echo "enter the number"
read num
echo "enter the number"
read num
if [ `expr $num % 2` -eq 0 ]
then
echo "number is even"
else
echo "number is odd"
fi
```

OUTPUT:

enter the number: 5

the number is odd.

RESULT:

Thus the program has been executed successfully.

Ex.No.2b **BIGGEST OF TWO NUMBERS**

AIM :

To write a program to find biggest in two numbers.

ALGORITHM :

STEP 1: Read The Two Numbers.

STEP 2: If Value Of A Is Greater Than B Is Big.

STEP 3: Else Print B Is Big.

STEP 4: Stop The Program.

PROGRAM:

```
echo "enter the number"
read a b
if [ $a -gt $b ]
then
echo "A is big"
else
echo "B is big"
fi
```

OUTPUT:

Enter The Two Number:

23 67

B is Big.

RESULT:

Thus the program has been executed successfully.

Ex.No.2c **BIGGEST OF THREE NUMBERS**

AIM:

To Write a Program to Find Biggest In Three Numbers.

ALGORITHM:

STEP 1: Read The Three Numbers.

STEP 2: If A Is Greater Than B And A Is Greater Than C Then Print A Is Big.

STEP 3: Else If B is greater Than C Then C Is Big.

STEP 4: Else Print C Is Big.

STEP 5: Stop The Program.

PROGRAM:

```
echo "enter three numbers"
read  a b c
if [ $a -gt $b ] && [ $a -gt $c ]
then
echo "A is big"
else if [ $b -gt $c ]
then
echo "B is big"
else
echo "C is big"
fi
fi
```

OUTPUT:

ENTER THREE NUMBERS:

23 54 78

C IS BIG.

RESULT:

Thus the program has been executed successfully.

Ex.No.2d FACTORIAL OF NUMBER

AIM:

To find a factorial of a number using shell script.

ALGORITHM:

Step 1: read a number.

Step 2: Initialize fact as 1.

Step 3: Initialize I as 1.

Step 4: While I is lesser than or equal to no.

Step 5: Multiply the value of I and fact and assign to fact increment the value of I by 1.

Step 6: print the result.

Step 7: Stop the program.

PROGRAM:

```
echo "enter the number"
read n
fact=1
i=1
while [ $i -le $n ]
do
fact=`expr $i * $fact`
i=`expr $i + 1`
done
echo "the fcatorial number of $n  is $fact”
```

OUTPUT:

Enter the number :

4

The factorial of 4 is 24.

RESULT:

Thus the program has been executed successfully.

FIBONACCI SERIES

Ex.No.2e

AIM :

To write a program to display the Fibonacci series.

ALGORITHM:

Step 1: Initialize n1& n2 as 0 & 1.

Step 2: enter the limit for Fibonacci.

Step 3: initialize variable as 0

Step 4: Print the Fibonacci series n1 and n2.

Step 5: While the var number is lesser than lim-2

Step 6: Calculate n3=n1+n2.

Step 7: Set n1=n2 and n2=n3

Step 8: Increment var by 1 and print n2

Step 9: stop the program.

PROGRAM:

```
echo " ENTER THE LIMIT FOR FIBONNACI SERIES"
read lim
n1=0
n2=1
var=0
echo "FIBONACCI SERIES IS "
echo "$n1"
echo "$n2"
while [ $var -lt `expr $lim - 2` ]
do
n3=`expr $n1 + $n2 `
```

```
n1=`expr $n2 `
n2=`expr $n3 `
var=`expr $var + 1 `
echo "$n2"
done
```

OUTPUT :

enter the limit for Fibonacci:

5

The Fibonacci series is:

0

1

1

2

3

RESULT:

Thus the program has been executed successfully.

INTRODUCTION TO OPERATING SYSTEMS

An Operating System is a program that manages the Computer hardware. It controls and coordinates the use of the hardware among the various application programs for the various users.

A Process is a program in execution. As a process executes, it changes state

- New: The process is being created
- Running: Instructions are being executed
- Waiting: The process is waiting for some event to occur
- Ready: The process is waiting to be assigned to a process
- Terminated : The process has finished execution

Apart from the program code, it includes the current activity represented by

- Program Counter,
- Contents of Processor registers,
- Process Stack which contains temporary data like function parameters, return addresses and local variables
- Data section which contains global variables
- Heap for dynamic memory allocation

A Multi-programmed system can have many processes running simultaneously with the CPU multiplexed among them. By switching the CPU between the processes, the OS can make the computer more productive. There is Process Scheduler which selects the process among many processes that are ready, for program execution on the CPU. Switching the CPU to another process requires performing a state save of the current process and a state restore of new process, this is Context Switch.

Scheduling Algorithms

CPU Scheduler can select processes from ready queue based on various scheduling algorithms. Different scheduling algorithms have different properties, and the choice of a particular algorithm may favour one class of processes over another. The scheduling criteria include

- CPU utilization:

- Throughput: The number of processes that are completed per unit time.
- Waiting time: The sum of periods spent waiting in ready queue.
- Turnaround time: The interval between the time of submission of process to the time of completion.
- Response time: The time from submission of a request until the first response is produced.

The different scheduling algorithms are

1. **FCFS: First Come First Serve Scheduling**

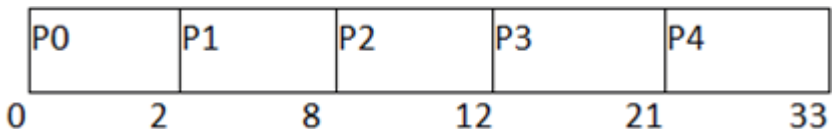
- It is the simplest algorithm to implement.

- The process with the minimal arrival time will get the CPU first.
- The lesser the arrival time, the sooner will the process gets the CPU.
- It is the non-pre-emptive type of scheduling.
- The Turnaround time and the waiting time are calculated by using the following formula.

Turn Around Time = Completion Time - Arrival Time
Waiting Time = Turnaround time - Burst Time

Process ID	Arrival Time	Burst Time	Completion Time	Turn Around Time	Waiting Time
0	0	2	2	2	0
1	1	6	8	7	1
2	2	4	12	8	4
3	3	9	21	18	9
4	4	12	33	29	17

Avg Waiting Time=31/5



2. SJF: Shortest Job First Scheduling

- The job with the shortest burst time will get the CPU first.
- The lesser the burst time, the sooner will the process get the CPU.
- It is the non-pre-emptive type of scheduling.
- However, it is very difficult to predict the burst time needed for a process hence this algorithm is very difficult to implement in the system.
- In the following example, there are five jobs named as P1, P2, P3, P4 and P5. Their arrival time and burst time are given in the table below.

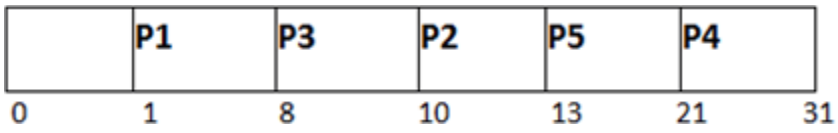
Process ID	Arrival Time	Burst Time	Completion Time	Turn Around Time	Waiting Time
1	1	7	8	7	0
2	3	3	13	10	7
3	6	2	10	4	2

4	7	10	31	24	14
5	9	8	21	12	4

Since, No Process arrives at time 0 hence; there will be an empty slot in the **Gantt chart** from time 0 to 1 (the time at which the first process arrives)

.

- According to the algorithm, the OS schedules the process which is having the lowest burst time among the available processes in the ready queue.
- Till now, we have only one process in the ready queue hence the scheduler will schedule this to the processor no matter what is its burst time.
- This will be executed till 8 units of time.
- Till then we have three more processes arrived in the ready queue hence the scheduler will choose the process with the lowest burst time.
- Among the processes given in the table, P3 will be executed next since it is having the lowest burst time among all the available processes.



Avg Waiting Time = 27/5

3. SRTF: Shortest Remaining Time First Scheduling

- It is the pre-emptive form of SJF. In this algorithm, the OS schedules the Job according to the remaining time of the execution

4. Priority Scheduling

- In this algorithm, the priority will be assigned to each of the processes.
- The higher the priority, the sooner will the process get the CPU.
- If the priority of the two processes is same then they will be scheduled according to their

arrival time.

5. Round Robin Scheduling

- In the Round Robin scheduling algorithm, the OS defines a time quantum (slice).
- All the processes will get executed in the cyclic way.
- Each of the process will get the CPU for a small amount of time (called time quantum) and then get back to the ready queue to wait for its next turn. It is a pre-emptive type of scheduling.

6. Multilevel Queue Scheduling

- A multi-level queue scheduling algorithm partitions the ready queue into several separate queues.
- The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type.
- Each queue has its own scheduling algorithm.

7. Multilevel Feedback Queue Scheduling

- Multilevel feedback queue scheduling, however, allows a process to move between queues.
- The idea is to separate processes with different CPU-burst characteristics.
- If a process uses too much CPU time, it will be moved to a lower-priority queue.
- Similarly, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue.
- This form of aging prevents starvation.

EX.NO.3 SYSTEM CALLS OF UNIX OPERATING SYSTEM

(a) Stat:

AIM :

To Execute a Unix Command in a 'C' program using stat() system call.

ALGORITHM:

1. Start the program

2. Declare the variables for the structure stat
3. Allocate the size for the file by using malloc function
4. Get the input of the file whose statistics want to be founded
5. Repeat the above step until statistics of the files are listed
6. Stop the program.
7.

PROGRAM:

```
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<stdlib.h>
int main(void)
{
char *path,path1[10];
struct stat *nfile;
nfile=(struct stat *) malloc (sizeof(struct stat));
printf("enter name of file whose stsistics has to");
scanf("%s",path1);
stat(path1,nfile);
printf("user id %d\n",nfile->st_uid);
printf("block size :%d\n",nfile->st_blksize);
printf("last access time %d\n",nfile->st_atime);
printf("time of last modification %d\n",nfile->st_atime);
printf("porduction mode %d \n",nfile->st_mode);
printf("size of file %d\n",nfile->st_size);
printf("nu,mber of links:%d\n",nfile->st_nlink);
}
```

OUTPUT:

enter name of file whose stsistics has to stat.c

user id 621

block size :4096

last access time 1145148485

time of last modification 1145148485

porduction mode 33204

size of file 654

nu,mber of links:1

Result:

Thus the program for stat system call has been executed successfully.

(b) Wait:

AIM :

To Execute a Unix Command in a ‘C’ program using wait() system call.

AlLGORITHM :

1. Start the program
2. Initialize the necessary variables
3. Use wait() to return the parent id of the child else return -1 for an error
4. Stop the program.

PROGRAM:

```
#include<stdio.h>
#include<unistd.h>
int main(void)
{
int pid,status,exitch;
if((pid=fork())==-1)
{
perror("error");
exit (0);
}
if(pid==0)
{
sleep(1);
printf("child process");
exit (0);
}
else
{
printf("parent process\n");
```

```
if((exitch=wait(&status))==-1)
{
perror("during wait()");
exit (0);
}
printf("parent existing\n");
exit (0);
}
}
```

OUTPUT :

```
parent process
child processparent existing
```

(c) GETPID:

AIM:

To Execute a Unix Command in a ‘C’ program using getpid() system call.

ALGORITHM:

1. Start the program
2. Declare the necessary variables
3. The getpid() system call returns the process ID of the parent of the
4. calling process
5. Stop the program.

PROGRAM:

```
#include<stdio.h>
int main()
{
int pid;
pid=getpid();
printf("process ID is %d\n",pid);
pid=getppid();
printf("parent process ID id %d\n",pid);
}
```

OUTPUT:

```
process ID is 2848
parent process ID id 2770
```

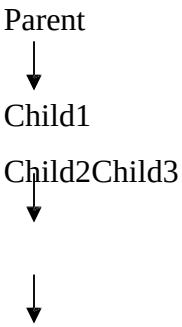
RESULT:

Thus the program for getpid system call has been executed successfully.

(d) Fork:

AIM:

To create a process in the following hierarchy



ALGORITHM:

1. Declare the necessary variables.
2. Parent process is the process of the program which is running.
3. Create the child1 process using fork() When parent is active.
4. Create the child2 process using fork() when child1 is active.
5. Create the child3 process using fork() when child2 is active.

PROGRAM:

```
#include<stdio.h>
int main(void)
{
int fork(void),value;
value=fork();
printf("main:value =%d\n",value);
return 0;
}
```

Output:

```
main:value =0
main:value =2860
```

Result:

Thus the program for fork system call has been executed successfully.

(e) Exec:

To Execute a Unix Command in a 'C' program using exec() system call.

AIM:

ALGORITHM:

1. Start the program
2. Declare the necessary variables
3. Use the prototype execv (filename,argv) to transform an executable binary file into process
4. Repeat this until all executed files are displayed
5. Stop the program.

PROGRAM:

```
#include<stdio.h>
main()
{
    int pid;
    char *args[]={"/bin/ls","-l",0};
    printf("\nParent Process");
    pid=fork();
    if(pid==0)
    {
        execv("/bin/ls",args); printf("\nChild
        process");
    }
    else
    {
        wait();
        printf("\nParent process");exit(0);
    }
}
```

Output:

total 440

-rwxrwxr-x 1 skec25 skec25 5210 Apr 16 06:25 a.out

-rw-rw-r-- 1 skec25 skec25 775 Apr 9 08:36 bestfit.c

-rw-rw-r-- 1 skec25 skec25 1669 Apr 10 09:19 correctpipe.c

-rw-rw-r-- 1 skec25 skec25 977 Apr 16 06:15 correctprio.c

-rw------- 1 skec25 skec25 13 Apr 10 08:14 datafile.dat

-rw------- 1 skec25 skec25 13 Apr 10 08:15 example.dat

-rw-rw-r-- 1 skec25 skec25 166 Apr 16 06:25 exec.c

-rw-rw-r-- 1 skec25 skec25 490 Apr 10 09:43 exit.c

Parent Process

Result:

Thus the program for exec system call has been executed successfully.

(f) Opendir,readdir:

AIM:

To write a C program to display the files in the given directory

ALGORITHM:

1. Start the program
2. Declare the variable to the structure dirent (defines the file system-independent directory) and also for DIR
3. Specify the directory path to be displayed using the opendir system call
4. Check for the existence of the directory and read the contents of the directory using readdir system call (returns a pointer to the next active directory entry)
5. Repeat the above step until all the files in the directory are listed
6. Stop the program

PROGRAM:

```
#include<stdio.h>
#include<dirent.h>
struct dirent *dptr;
int main(int argc,char *argv[])
{
    char buff[256];
    DIR *dirp;
    printf("\n\nEnter directory name");
    scanf("%s",buff);
    if((dirp=opendir(buff))==NULL)
    {
        printf("Error");
        exit(1);
    }
    while(dptr=readdir(dirp))
    {
```

```
            printf("%s\n",dptr->d_name);
        }
        closedir(dirp);
    }
```

Output:

```
Enter directory name
oslab
openreaddir.c
a.out
..vidhya.c
vidhya.
```

(h)Open:

AIM:

To Execute a Unix Command in a ‘C’ program using open() system call.

ALGORITHM:

1. Start the program
2. Declare the necessary variables
3. Open file1.dat to read or write access
4. Create file1.dat if it doesn’t exist
5. Return error if file already exist
6. Permit read or write access to the file
7. Stop the program.

PROGRAM:

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/stat.h>
int main()
{
    int fd;
    fd=creat("file1.dat",S_IREAD|S_IWRITE);
    if(fd==-1)
        printf("Error in opening file1.dat\n");
    else
    {                    }
```

111

\nfile1.dat opened for read/write access\n");printf("\nfile1.dat is currently empty");

printf("

close(fd);

}

OUTPUT:

file1.dat opened for read/write access.

RESULT:

Thus the program for open system call has been executed successfully.

Ex.No:4 I/O SYSTEM CALLS OF UNIX OPERATING SYSTEM

(OPEN, READ, WRITE, ETC)

OPEN,READ,WRITE:

AIM:

To implement UNIX I/O system calls open, read , write etc.

ALGORITHM:

1. Create a new file using creat command (Not using FILE pointer).
2. Open the source file and copy its content to new file using read and write command.
3. Find size of the new file before and after closing the file using stat command.

PROGRAM:

```
#include<fcntl.h>            //file control
#include<sys/types.h>
#include<sys/stat.h>
static char message[]="hai Hello world";

int main()
{
    int fd;
    char buffer[80];
    fd=open("new2file.txt",O_RDWR|O_CREAT|O_EXCL,S_IREAD|S_IWRITE);
    if(fd!=-1)
    {
        printf("new2file.txt opened for read/write access\n");
        write(fd,message,sizeof(message)); lseek(fd,0l,0);
        if(read(fd,buffer,sizeof(message))==sizeof(message))
            printf("\"%s\" was written to new2file.txt\n",buffer);
```

```
        else
            printf("***Error readind new2file.txt***\n");
        close(fd);
    }
    else
        printf("***new2file.txt  already  exists***\n");
    exit(0);
}
```

OUTPUT:

```
new2file.txt opened for read/write access
"hai Hello world" was written to new2file.txt
```

Ex.No:5 C PROGRAMS TO SIMULATE UNIX COMMANDS LIKE LS, GREP.

(a) LS:

AIM:

To implement ls command in c.

ALGORITHM:

1. Include a dirent.h header file.
2. Create a variable DIR as pointer
3. Create a structure pointer of dirent
4. Using opendir function,open the current directory
5. Read the directory for files using readdir function
6. Display it till the end of the file.

PROGRAM:

```
#include<stdio.h>
#include<dirent.h>
#include<errno.h>
#include<sys/stat.h>
int main(int argc,char ** argv)
{
    DIR *dir;
    struct dirent *dirent; char * where=NULL;
    if(argc==1)where=get_current_dir_name();
    else
        where=argv[1];
    if(NULL==(dir=opendir(where))){
    fprintf(stderr,"%d(%s)opendir %s failed\n",errno,strerror(errno),where);
    return 2;
```

```
}
while(NULL!=(dirent=readdir(dir)))
{
        printf("%s\n",dirent->d_name);
}
closedir(dir);
return 0;
}
```

OUTPUT:

openreaddir.c
file.txt
openclose.c
staffrr.c
fifo.c
example.dat
newgetpid.c

RESULT :

Thus the system call program has been executed successfully.

(b) GREP:

AIM:

To implement the grep command in c

ALGORITHM:

1. Obtain the required pattern to be searched and file name from the user
2. Open the file and read the constants used by word till the end of the file.
3. Match the given pattern with the read word and if it matches,display the line of occurance
4. Do this till the end of the file is reached.

PROGRAM:

```
#include<stdio.h>
#include<string.h>
int main(int argv,char * args[])
{
```

```
FILE * f;
char str[100];
char c;
int i,flag,j,m,k;
char arg[]="HI";
char temp[30];
if(argv<3)
{
	printf("usage grep<s> <val.txt>\n");
	return;
}
f=fopen(args[2],"r");
while(!feof(f))
{
```

```
    i=0;
    while(1)
    {
        fscanf(f,"%c",&c);
  if(feof(f))
        {
            str[i++]='\0';
            break;
        }
        if(c=='\n')
        {
            str[i++]='\0';
            break;
        }
        str[i++]=c;
    }
    if(strlen(str)>=strlen(args[1]))
    for(k=0;k<=strlen(str)-strlen(args[1]);k++)
    {
     for(m=0;m<strlen(args[1]);m++)
        temp[m]=str[k+m];
        temp[m]='\0';
        if(strcmp(temp,args[1])==0)
        {
            printf("%s\n",str);
          break;
        }
    }
    }
    return 0;
}
```

OUTPUT:

```
[skec25@localhost ~]$ ./a.out print stat.c
printf("enter name of file whose stsistics has to");
printf("user id %d\n",nfile->st_uid);
printf("block size :%d\n",nfile->st_blksize); printf("last
access time %d\n",nfile->st_atime); printf("time of last
modification %d\n",nfile->st_atime); printf("porduction
mode %d \n",nfile->st_mode); printf("size of file
%d\n",nfile->st_size);
printf("nu,mber of links:%d\n",nfile->st_nlink);
```

RESULT:

Thus the program has been executed successfully.

Ex.No:6a FCFS (FIRST COME FIRST SERVE) CPU SCHEDULING

AIM:

To write a program to implement the FCFS (First Come First Serve) CPU scheduling Algorithm

ALGORITHM:

1. START the program
2. Get the number of processors
3. Get the Burst time of each processors
4. Calculation of Turn Around Time and Waiting Time
    a) tot_TAT = tot_TAT + pre_TAT
    b) avg_TAT = tot_TAT/num_of_proc
    c) tot_WT = tot_WT + pre_WT + PRE_BT
    d) avg_WT = tot_WT/num_of_proc
5. Display the result
6. STOP the program

PROGRAM: (FCFS Scheduling)

```
#include<stdio.h>
#include<conio.h>
int p[30],bt[30],tot_tat=0,wt[30],n,tot_wt=0,tat[30],FCFS_wt=0,FCFS_tat=0;
float awt,avg_tat,avg_wt;
void main()
{
  int i;
  clrscr();
  printf("\nEnter the no.of processes \n");
  scanf("%d",&n);
  printf("Enter burst time for each process\n");
```

```
  for(i=0;i<n;i++)
  {
    scanf("%d",&bt[i]);
    p[i] = i;
  }
  printf("\n FCFS Algorithm \n");
  WT_TAT(&FCFS_tat,&FCFS_wt);
  printf("\n\nTotal Turn around Time:%d",FCFS_tat);
  printf("\nAverage Turn around Time :%d ", FCFS_tat/n);
  printf("\nTotal Waiting Time:%d",FCFS_wt);
  printf("\nTotal avg. Waiting Time:%d",FCFS_wt/n);
  getch();
}
int WT_TAT(int *a, int *b)
```

```
{
  int i;
  for(i=0;i<n;i++)
  {
    if(i==0)
      tat[i] = bt[i];
    else
      tat[i] = tat[i-1] + bt[i];
    tot_tat=tot_tat+tat[i];
  }
  *a = tot_tat;
  wt[0]=0;
  for(i=1;i<n;i++)
  {
    wt[i]=wt[i-1]+bt[i-1];
    tot_wt = tot_wt+wt[i];
  }
  *b = tot_wt;
  printf("\nPROCESS\t\tBURST  TIME\tTURN  AROUND  TIME\tWAITING TIME");
  for(i=0; i<n; i++)
  printf("\nprocess[%d]\t\t%d\t\t%d\t\t%d",p[i],bt[i],tat[i],wt[i]);
  return 0;
}
```

OUTPUT: (FCFS Scheduling Algorithm)

RESULT: Thus the program to implement the FCFS (First Come First Serve) CPU scheduling Algorithm was written, executed and the output was verified successfully.

Ex.No:6b SJF (SHORTEST JOB FIRST) CPU SCHEDULING ALGORITHM

AIM:

To write a program to implement the SJF (Shortest Job First) CPU scheduling Algorithm

ALGORITHM:

1. START the program
2. Get the number of processors
3. Get the Burst time of each processors
4. Sort the processors based on the burst time
5. Calculation of Turn Around Time and Waiting Time
    e) tot_TAT = tot_TAT + pre_TAT
    f) avg_TAT = tot_TAT/num_of_proc
    g) tot_WT = tot_WT + pre_WT + PRE_BT
    h) avg_WT = tot_WT/num_of_proc
6. Display the result
7. STOP the program

PROGRAM: (SJF Scheduling)

```
#include<stdio.h>
#include<conio.h>
int p[30],bt[30],tot_tat=0,wt[30],n,tot_wt=0,tat[30],SJF_wt=0,SJF_tat=0;
float awt,avg_tat,avg_wt;
void main()
{
  int i;
  clrscr();
```

```
printf("\nEnter the no.of processes \n");
scanf("%d",&n);
printf("Enter burst time for each process\n");
for(i=0;i<n;i++)
{
  scanf("%d",&bt[i]);
  p[i] = i;
}
sort();
WT_TAT(&SJF_tat,&SJF_wt);
printf("\n\nTotal Turn around Time:%d",SJF_tat);
printf("\nAverage Turn around Time :%d ", SJF_tat/n);
printf("\nTotal Waiting Time:%d",SJF_wt);
printf("\nTotal avg. Waiting Time:%d",SJF_wt/n);
```

```
  getch();
}
int sort()
{
  int t,i,j;
  for(i=0;i<n;i++)
  {
    for(j=i+1;j<n;j++)
    {
      if(bt[i]>bt[j])
      {
            swap(&bt[j],&bt[i]);
            swap(&p[j],&p[i]);
      }
    }
  }
  return 0;
}
int swap(int *a, int *b)
{
  int t;
  t =  *a;
  *a = *b;
  *b = t;
  return 0;
}
int WT_TAT(int *a, int *b)
{
  int i;
  for(i=0;i<n;i++)
  {
    if(i==0)
```

```
    tat[i] = bt[i];
  else
    tat[i] = tat[i-1] + bt[i];
  tot_tat=tot_tat+tat[i];
}
*a = tot_tat;
wt[0]=0;
for(i=1;i<n;i++)
{
  wt[i]=wt[i-1]+bt[i-1];
  tot_wt = tot_wt+wt[i];
}
*b = tot_wt;
printf("\nPROCESS\t\tBURST  TIME\tTURN  AROUND  TIME\tWAITING TIME");
```

```
 for(i=0; i<n; i++)
 printf("\nprocess[%d]\t\t%d\t\t%d\t\t%d",p[i]+1,bt[i],tat[i],wt[i]);
 return 0;
}
```

OUTPUT: (SJF Scheduling Algorithm)

```
Enter the no.of processes
5
Enter burst time for each process
45
32
21
67
89

PROCESS         BURST TIME      TURN AROUND TIME        WAITING TIME
process[3]              21              21              0
process[2]              32              53              21
process[1]              45              98              53
process[4]              67              165             98
process[5]              89              254             165

Total Turn around Time:591
Average Turn around Time :118
Total Waiting Time:337
Total avg. Waiting Time:67
```

RESULT:

Thus the program to implement the SJF (Shortest Job First) CPU scheduling Algorithm was written, executed and the output was verified successfully.

Ex.No:7a PRIORITY CPU SCHEDULING ALGORITHM

AIM:

To write a program to implement the Priority CPU scheduling Algorithm

ALGORITHM:

1. START the program
2. Get the number of processors
3. Get the Burst time of each processors
4. Get the priority of all the processors
5. Sort the processors based on the priority
6. Calculation of Turn Around Time and Waiting Time
    i) tot_TAT = tot_TAT + pre_TAT
    j) avg_TAT = tot_TAT/num_of_proc
    k) tot_WT = tot_WT + pre_WT + PRE_BT
    l) avg_WT = tot_WT/num_of_proc
7. Display the result
8. STOP the program

PROGRAM: (Priority Scheduling)

```
#include<stdio.h>
#include<conio.h>
int p[30],bt[30],tot_tat=0,pr[30],wt[30],n,tot_wt=0,tat[30],PR_wt=0,PR_tat=0;
float awt,avg_tat,avg_wt;
void main()
{
  int i;
  clrscr();
```

```
printf("\nEnter the no.of processes \n");
scanf("%d",&n);
for(i=0;i<n;i++)
{
  printf("Enter burst time and priority of process[%d]:",i+1);
  scanf("%d%d",&bt[i],&pr[i]);
  p[i] = i;
}
sort();
WT_TAT(&PR_tat,&PR_wt);
printf("\n\nTotal Turn around Time:%d",PR_tat);
printf("\nAverage Turn around Time :%d ", PR_tat/n);
printf("\nTotal Waiting Time:%d",PR_wt);
printf("\nTotal avg. Waiting Time:%d",PR_wt/n);
```

```
  getch();
}
int sort()
{
  int t,i,j,t2,t1;
  for(i=0;i<n;i++)
  {
    for(j=i+1;j<n;j++)
    {
      if(pr[i]>pr[j])
      {
            swap(&bt[j],&bt[i]);
            swap(&p[j],&p[i]);
            swap(&pr[j],&pr[i]);
      }
    }
  }
  return 0;
}
int swap(int *a, int *b)
{
  int t;
  t =  *a;
  *a = *b;
  *b = t;
  return 0;
}
int WT_TAT(int *a, int *b)
{
  int i;
  for(i=0;i<n;i++)
  {
```

```
  if(i==0)
    tat[i] = bt[i];
  else
    tat[i] = tat[i-1] + bt[i];
  tot_tat=tot_tat+tat[i];
}
*a = tot_tat;
wt[0]=0;
for(i=1;i<n;i++)
{
  wt[i]=wt[i-1]+bt[i-1];
  tot_wt = tot_wt+wt[i];
}
*b = tot_wt;
```

```
 printf("\nPROCESS\t\tBURST TIME\tPRIORITY\tTURN AROUND TIME\tWAITING
TIME");
 for(i=0; i<n; i++)
   printf("\nprocess[%d]\t\t%d\t\t%d\t\t%d\t\t%d"
         ,p[i]+1,bt[i],pr[i],tat[i],wt[i]);
 return 0; }
```

OUTPUT: (Priority Scheduling Algorithm)

```
Enter the no.of processes
5
Enter burst time and priority of process[1]:45 2
Enter burst time and priority of process[2]:43 5
Enter burst time and priority of process[3]:21 1
Enter burst time and priority of process[4]:78 3
Enter burst time and priority of process[5]:30 4

PROCESS         BURST TIME      PRIORITY        TURN AROUND TIME        WAITING TIME
process[3]              21              1               21              0
process[1]              45              2               66              21
process[4]              78              3               144             66
process[5]              30              4               174             144
process[2]              43              5               217             174

Total Turn around Time:622
Average Turn around Time :124
Total Waiting Time:405
Total avg. Waiting Time:81
```

RESULT:

Thus the program to implement the Priority CPU scheduling Algorithm was written, executed and the output was verified successfully.

Ex.No:7b ROUND ROBIN CPU SCHEDULING ALGORITHM

AIM:

To write a program to implement the Round Robin CPU scheduling Algorithm

ALGORITHM:

1. START the program
2. Get the number of processors
3. Get the Burst time(BT) of each processors
4. Get the Quantum time(QT)
5. Execute each processor until reach the QT or BT
6. Time of reaching processor's BT is it's Turn Around Time(TAT)
7. Time waits to start the execution, is the waiting time(WT) of each processor
8. Calculation of Turn Around Time and Waiting Time
    m) tot_TAT = tot_TAT + cur_TAT
    n) avg_TAT = tot_TAT/num_of_proc
    o) tot_WT = tot_WT + cur_WT
    p) avg_WT = tot_WT/num_of_proc
9. Display the result
10. STOP the program

PROGRAM: (Round Robin Algorithm)

```
#include<stdio.h>
#include<conio.h>
int TRUE = 0;
int FALSE = -1;
int tbt[30],bt[30],tat[30],n=0,wt[30],qt=0,tqt=0,time=0,lmore,t_tat=0,t_wt=0;
void main()
{
  int i,j;
```

```
clrscr();
printf("\nEnter no. of processors:");
scanf("%d",&n);
printf("\nEnter Quantum Time:");
scanf("%d",&qt);
for(i=0;i<n;i++)
{
  printf("\nEnter Burst Time of Processor[%d]:",i+1);
  scanf("%d",&bt[i]);
  tbt[i] = bt[i];
  wt[i] = tat[i] = 0;
}
```

```
 lmore = TRUE;
 while(lmore == TRUE)
 {
  lmore = FALSE;
  for(i=0;i<n;i++)
  {
   if(bt[i] != 0)
        wt[i] = wt[i] + (time - tat[i]);
   tqt = 1;
   while(tqt <= qt && bt[i] !=0)
   {
        lmore = TRUE;
        bt[i] = bt[i] -1;
        tqt++;
        time++;
        tat[i] = time;
   }
  }
 }
 printf("\nProcessor ID\tBurstTime\tTurnAroundTime\tWaitingTime\n");
 for(i=0;i<n;i++)
 {
  printf("Processor%d\t\t%d\t\t%d\t\t%d\n",i+1,tbt[i],tat[i],wt[i]);
  t_tat = t_tat + tat[i];
  t_wt = t_wt + wt[i];
 }
 printf("\nTotal Turn Around Time:%d",t_tat);
 printf("\nAverage Turn Around Time:%d",t_tat/n);
 printf("\nTotal Waiting Time:%d",t_wt);
 printf("\nAverage Waiting Time:%d",t_wt/n);
 getch();
}
```

OUTPUT: (Round Robin Scheduling Algorithm)

```
Enter no. of processors:5

Enter Quantum Time:6

Enter Burst Time of Processor[1]:21

Enter Burst Time of Processor[2]:18

Enter Burst Time of Processor[3]:12

Enter Burst Time of Processor[4]:30

Enter Burst Time of Processor[5]:15

Processor ID    BurstTime       TurnAroundTime  WaitingTime
Processor1              21              84              63
Processor2              18              72              54
Processor3              12              48              36
Processor4              30              96              66
Processor5              15              81              66

Total Turn Around Time:381
Average Turn Around Time:76
Total Waiting Time:285
Average Waiting Time:57
```

RESULT:

Thus the program to implement the Round Robin CPU scheduling Algorithm was written, executed and the output was verified successfully.

Ex.No:8 DEVELOPING APPLICATION USING INTER PROCESS COMMUNICATION (USING SHARED MEMORY, PIPES OR MESSAGE QUEUES)

(a) Shared memory:

AIM:

To implement the interprocess communication using shared memory.

ALGORITHM:

1. Start the program
2. Declare the necessary variables
3. shmat() and shmdt() are used to attach and detach shared memory segments. They are prototypes as follows:

   void *shmat(int shmid, const void *shmaddr, int shmflg);

   int shmdt(const void *shmaddr);
4. shmat() returns a pointer, shmaddr, to the head of the shared segment associated with a valid shmid. shmdt() detaches the shared memory segment located at the address indicated by shmaddr
5. Shared1.c simply creates the string and shared memory portion.
6. Shared2.c attaches itself to the created shared memory portion and uses the string (printf)
7. Stop the program.

PROGRAM:

Shared1.c:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#define SHMSZ     27
main()
{
   char c;
```

```
int shmid;
key_t key;
char *shm, *s;
key = 5678;
if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666)) < 0) {
    perror("shmget");
    exit(1);
}

if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
    perror("shmat");
    exit(1);
```

```
    }
```

Shared2.c

```
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#define SHMSZ     27
main()
{
    int shmid;
   key_t  key;
   char *shm, *s;
   key = 5678;
    if ((shmid = shmget(key, SHMSZ, 0666)) < 0) {
        perror("shmget");
        exit(1);
    }
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        exit(1);
    }
    for (s = shm; *s != NULL; s++)
        putchar(*s);
    putchar('\n');
    *shm = '*';
    exit(0);
}
```

OUTPUT:

Abcdefghijklmnopqrstuvwxyz

RESULT:

Thus the program has been executed successfully and the output is verified.

(b) PIPES:

AIM:

To implement the interprocess communication using pipes.

ALGORITHM:

1. Start the program
2. Create the child process using fork()
3. Create the pipe structure using pipe()
4. Now close the read end of the parent process using close()
5. Write the data in the pip[e using write()
6. Now close the write end of child process using close()
7. Read the data in the pipe using read()
8. Display the string

9. Stop the program.

PROGRAM:

```
#include<stdio.h>
int main()
{
int fd[2],child;
char a[10];
printf("\n enter the string to enter into the pipe:");
scanf("%s",a);
pipe(fd);
child=fork();
if(!child)
{ close(fd[10]);
write(fd[1],a,5);
wait(0);
}
else
{
close(fd[1]);
read(fd[0],a,5);
printf("\n the string retrived from pipe is %s\n",a);
}
return 0;
}
```

OUTPUT:

```
enter the string to enter into the pipe:computer
 the string retrived from pipe is computer
```

RESULT:

Thus the program has been executed and the output is verified successfully.

(c) message queue:

AIM:

To implement the interprocess communication using message passing.

ALGORITHM:

1. Start the program
2. Create two files msgsend and msgrecv.c
3. Msgsend creates a message queue with a basic key and message flag msgflg = IPC_CREAT | 0666 -- create queue and make it read and appendable by all, and sends one message to the queue.
4. Msgrecv reads the message from the queue

5. A message of type (sbuf.mtype) 1 is sent to the queue with the message ``Did you get this?"
6. The Message queue is opened with msgget (message flag 0666) and the ***same*** key as message_send.c
7. A message of the ***same*** type 1 is received from the queue with the message ``Did you get this?" stored in rbuf.mtext.
8. Stop the program.

PROGRAM:

Msgsend:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <string.h>

#define MSGSZ     128
typedef struct msgbuf {
        long    mtype;
        char    mtext[MSGSZ];
        } message_buf;

main()
{
   int msqid;
   int msgflg = IPC_CREAT | 0666;
   key_t key;
   message_buf sbuf;
   size_t buf_length;

  key = 1234;
```

```
(void) fprintf(stderr, "\nmsgget: Calling msgget(%#lx,\
%#o)\n",
key, msgflg);

    if ((msqid = msgget(key, msgflg )) < 0) {
        perror("msgget");
        exit(1);
    }
    else
     (void) fprintf(stderr,"msgget: msgget succeeded: msqid = %d\n", msqid);
    sbuf.mtype = 1;
```

```
    (void) fprintf(stderr,"msgget: msgget succeeded: msqid = %d\n", msqid);
    (void) strcpy(sbuf.mtext, "Did you get this?");
    (void) fprintf(stderr,"msgget: msgget succeeded: msqid = %d\n", msqid);

    buf_length = strlen(sbuf.mtext) + 1 ;
 if (msgsnd(msqid, &sbuf, buf_length, IPC_NOWAIT) < 0) {
       printf ("%d, %d, %s, %d\n", msqid, sbuf.mtype, sbuf.mtext, buf_length);
        perror("msgsnd");
        exit(1);
    }

   else
     printf("Message: \"%s\" Sent\n", sbuf.mtext);

    exit(0);
}
```

Msgrecv:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>

#define MSGSZ     128
typedef struct msgbuf {
    long    mtype;
    char    mtext[MSGSZ];
} message_buf;
```

```
main()
{
   int msqid; key_t
   key; message_buf
   rbuf;
key = 1234;

   if ((msqid = msgget(key, 0666)) < 0) {
      perror("msgget");
      exit(1);
   }
if (msgrcv(msqid, &rbuf, MSGSZ, 1, 0) < 0) {
```

```
        perror("msgrcv");
        exit(1);
    }
printf("%s\n", rbuf.mtext);
    eixit(0);
}
```

OUTPUT:

"Did you get this?"

RESULT:

Thus the program has been executed successfully and the output is verified.

Ex.No:9 INTERPROCESS COMMUNICATION USING MESSAGE PASSING.

AIM:

To write a program to solve the Producer Consumer Problem using Semaphores

ALGORITHM:

1. START the program
2. If the request is for Producer
    a. Get the count of number of items to be produced
    b. Add the count with current Buffer size
    c. If the new Buffer size is full

        Don't produce anything; display an error message

        Producer has to wait till any consumer consume the existing item
    d. Else

        Produce the item

        Increment the Buffer by the number of item produced
3. If the request is for Consumer
    a. Get the count of number of items to be consumed
    b. Subtract the count with current Buffer size
    c. If the new Buffer size is lesser than 0

        Don't allow the consumer to consume anything; display an error message

        Consumer has to wait till the producer produce new items
    d. Else

        Consume the item

        Decrement the Buffer by the number of item consumed
4. STOP the program

PROGRAM:

```
#include<stdio.h>
int n=0,buffersize=0,currentsize=0;
void producer()
{
  printf("\nEnter number of elements to be produced: ");
  scanf("%d",&n);
  if(0<=(buffersize-(currentsize+n)))
  {
    currentsize+=n;
    printf("%d Elements produced by producer where buffersize is %d\n", currentsize,
buffersize);
```

```
  }
  else
    printf("\nBuffer is not sufficient\n");
}

void consumer()
{
  int x;
  printf("\nEnter no. of elements to be consumed: ");
  scanf("%d",&x);
  if(currentsize>=x)
  {
    currentsize-=x;
    printf("\nNumber of elements consumed: %d, Number of Elements left: %d", x, currentsize);
  }
  else
  {
    printf("\nNumber of Elements consumed should not be greater than Number of Elements
produced\n");
  }
}
void main()
{
  int c;
  printf("\nEnter maximum size of buffer:");
  scanf("%d",&buffersize);
  do
  {
    printf("\n1.Producer 2.Consumer 3.Exit");
    printf("\nEnter Choice:");
    scanf("%d",&c);
    switch(c)
```

```
{
  case 1:
  if(currentsize >= buffersize)
    printf("\nBuffer is full. Cannot produce");
  else
    producer();
  break; case 2:
  if(currentsize <= 0)
    printf("\nBuffer is Empty. Cannot consume");
  else
    consumer();
  break;
  default:
```

```
            exit();
        break;
  }
}
while(c!=3);
}
```

OUTPUT: (Producer Consumer problem)

```
Enter maximum size of buffer:20

1.Producer 2.Consumer 3.Exit
Enter Choice:2

Buffer is Empty. Cannot consume
1.Producer 2.Consumer 3.Exit
Enter Choice:1

Enter number of elements to be produced: 23

Buffer is not sufficient

1.Producer 2.Consumer 3.Exit
Enter Choice:1

Enter number of elements to be produced: 5
5 Elements produced by producer where buffersize is 20

1.Producer 2.Consumer 3.Exit
Enter Choice:2

Enter no. of elements to be consumed: 4

Number of elements consumed: 4, Number of Elements left: 1
1.Producer 2.Consumer 3.Exit
Enter Choice:2

Enter no. of elements to be consumed: 4

Number of Elements consumed should not be greater than Number of Elements produced

1.Producer 2.Consumer 3.Exit
Enter Choice:1

Enter number of elements to be produced: 5
6 Elements produced by producer where buffersize is 20

1.Producer 2.Consumer 3.Exit
Enter Choice:1

Enter number of elements to be produced: 15

Buffer is not sufficient
```

RESULT: Thus the program has been executed successfully and the output is verified.

Ex.No:9 MEMORY MANAGEMENT SCHEME-I

(a) Firstfit:

AIM:

To write a program to implement first fit algorithm for memory management.

ALGORITHM:

1. Start the process
2. Declare the size
3. Get the number of processes to be inserted
4. Allocate the first hole that is big enough searching
5. Start at the beginning of the set of holes
6. If not start at the hole that is sharing the pervious first fit search end
7. Compare the hole
8. if large enough then stop searching in the procedure
9. Display the values
10. Stop the process

PROGRAM:

```
#include<stdio.h>
struct process
{
int ps;
int flag;
} p[50];
```

```
struct sizes
{
int size;
int alloc;
}
s[5];
int main()
{
int i=0,np=0,n=0,j=0;
printf("\n first fit");
printf("\n");
```

```
printf("enter the number of blocks \t");
scanf("%d",&n);
printf("\t\t\n enter the size for %d blocks\n",n);
for(i=0;i<n;i++)
{
printf("enter the size for %d block \t",i);
scanf("%d",&s[i].size);
}
printf("\n\t\t enter the number of process\t",i);
scanf("%d",&np);
printf("enter the size of %d processors !\t",np);
printf("/n");
for(i=0;i<np;i++)
{
printf("enter the size of process %d\t",i);
scanf("\n%d",&p[i].ps);
}
printf("\n\t\t Allocation of blocks using first fit is as follows\n");
printf("\n\t\t process \t process size\t blocks\n");
for(i=0;i<np;i++)
{
for(j=0;j<n;j++)
{
if(p[i].flag!=1)
{
if(p[i].flag!=1)
{
if(p[i].ps<=s[j].size)
{
if(!s[j].alloc)
{
p[i].flag=1;
```

```
s[j].alloc=1;
printf("\n\t\t %d\t\t\t%d\t%d\t",i,p[i].ps,s[j].size);
}
}
}
}
}
}
for(i=0;i<np;i++)
{
if(p[i].flag!=1)
printf("sorry !!!!!!!process %d must wait as there is no sufficient memory");
}
}
```

\

OUTPUT:

first fit

enter the number of blocks 4

enter the size for 4 blocks

enter the size for 0 block 3

enter the size for 1 block 2

enter the size for 2 block 5

enter the size for 3 block 10

enter the number of process 4

enter the size of 4 processors !

enter the size of process 0 2

enter the size of process 1 6

enter the size of process 2 7

enter the size of process 3 9

Allocation of blocks using first fit is as follows

process process size blocks

0 2 3

sorry !!!!!!!process 1 must wait as there is no sufficient memory

1 6 10

RESULT:

The program for first fit was implemented and hence verified

(b) BESTFIT:

AIM:

To write a program to implement best fit algorithm for memory management.

ALGORITHM:

1. Start the process
2. Declare the size
3. Get the number of processes to be inserted
4. Allocate the best hole that is small enough searching

5. Start at the best of the set of holes
6. If not start at the hole that is sharing the pervious best fit search end
7. Compare the hole
8. If small enough then stop searching in the procedure
9. Display the values
10. Stop the process

PROGRAM:

```
#include<stdio.h>
#define MAX 20
int main()
{
int bsize[MAX],fsize[MAX],nb,nf;
int temp,low=10000;
static int bflag[MAX],fflag[MAX];
int i,j;
printf("\n enter the number of blocks");
scanf("%d",&nb);
for(i=1;i<=nb;i++)
{
printf("Enter the size of memory block % d",i);
scanf("%d", &bsize[i]);
}
printf("\n enter the number of files");
scanf("%d",&nf);
for(i=1;i<=nf;i++)
{
printf("\n enetr the size of  file %d",i);
scanf("%d",&fsize[i]);
}
for(i=1;i<=nf;i++)
```

```
{
for(j=1;j<=nb;j++)
{
if(bflag[j]!=1)
{
temp=bsize[j]-fsize[i];
if(temp>=0)
{
if(low>temp)
{
fflag[i]=j;
low=temp;
}
}
```

```
}}
bflag[fflag[i]]=1;
low=10000;
}
printf("\n  file no \t file.size\t block no \t block size");
for(i=1;i<=nf;i++)
printf("\n \n %d \t\t%d\t\t%d\t\t%d",i,fsize[i],fflag[i],bsize[fflag[i]]);
}
```

OUTPUT:

```
enter the number of blocks5
Enter the size of memory block  1 10
Enter the size of memory block  2 12
Enter the size of memory block  3 10
Enter the size of memory block 4 5
Enter the size of memory block  5 7

 enter the number of files 5
 enetr the size of file 1 5
 enetr the size of  file 2 6
```

file no	file.size	block no	block size
1	5	4	5
2	6	5	7
3	7	1	10
4	10	3	10
5	12	2	12

RESULT:

The program for best fit was implemented and hence verified

Ex.No 11 **MEMORY MANAGEMENT SCHEME II**

(a) FIFO:

AIM:

To write a program to implement FIFO page replacement algorithm

ALGORITHM:

1. Start the process
2. Declare the size with respect to page length
3. Check the need of replacement from the page to memory
4. Check the need of replacement from old page to new page in memory
5. Forma queue to hold all pages
6. Insert the page require memory into the queue
7. Check for bad replacement and page fault
8. Get the number of processes to be inserted
9. Display the values
10. Stop the process

PROGRAM:

```
#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 number:\n");
for(i=1;i<=n;i++)
scanf("%d",&a[i]);
printf("\n enter the number of frames:\n");
scanf("%d",&no);
for(i=0;i<no;i++)
frame[i]=-1;
j=0;
printf("\tref string\t page frmaes\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("page fault is %d",count);
getch();
return 0;
}
```

OUTPUT:

```
enter the number of pages:
4

 enter the reference string:
7
2
1
0

 enter the number of frames:
3
      ref string      page frmaes
7           7     -1     -1
2           7     2      -1
1           7     2      1
0           0     2      1
```

page fault is 4

RESULT:

The program for FIFO page replacement was implemented and hence verified.

(b) LRU:

AIM:

To write a program a program to implement LRU page replacement algorithm

ALGORITHM :

1. Start the process
2. Declare the size
3. Get the number of pages to be inserted
4. Get the value
5. Declare counter and stack
6. Select the least recently used page by counter value
7. Stack them according the selection.

8. Display the values
9. Stop the process

PROGRAM:

```
#include<stdlib.h>
#include<stdio.h>
#define max 100
#define min 10
int ref[max],count,frame[min],n;
void input()
{
int i,temp;
count=0;
printf("\n\n\tEnter the number of page frames : ");
scanf("%d",&n);
printf("\n\n\tEnter the reference string (-1 for end) : ");
scanf("%d",&temp);
while(temp != -1)
{
ref[count++]=temp;
scanf("%d",&temp);
}
}
void LRU()
{
```

```
int i,j,k,stack[min],top=0,fault=0;
system("CLS");
for(i=0;i<count;i++)
{
if(top<n)
stack[top++]=ref[i],fault++;
else
{
for(j=0;j<n;j++)
if(stack[j]==ref[i])
break;
if(j<n)
{
for(k=j;k<n-1;k++)
```

```
stack[k]=stack[k+1];
stack[k]=ref[i];
}
else
{
for(k=0;k<n-1;k++)
stack[k]=stack[k+1];
stack[k]=ref[i];
fault++;
}
}
printf("\n\nAfter inserting %d the stack status is : ",ref[i]);
for(j=0;j<top;j++)
printf("%d ",stack[j]);
}
printf("\n\n\tEnd to inserting the reference string.");
printf("\n\n\tTotal page fault is %d.",fault);
printf("\n\n\tPress any key to continue.");
}
void main()
{
int x;
//freopen("in.cpp","r",stdin);
while(1)
{
printf("\n\n\t-----MENU---- ");
printf("\n\t1. Input ");
printf("\n\t2. LRU (Least Recently Used) Algorithm");
printf("\n\t0. Exit.");
printf("\n\n\tEnter your choice.");
scanf("%d",&x);
switch(x)
{
case 1:
input();
```

```
break;
case 2:
LRU();
break;
case  0:
exit(0);
}
}
}
```

OUTPUT:

```
-----MENU-------
    1. Input
```

2. LRU (Least Recently Used) Algorithm

0. Exit.

Enter your choice.1

Enter the number of page frames : 3

Enter the reference string (-1 for end) : 2

0

1

1

-1

.....MENU.....

1. Input

2. LRU (Least Recently Used) Algorithm

0. Exit.

Enter your choice.2

After inserting 2 the stack status is : 2

After inserting 0 the stack status is : 2 0

After inserting 1 the stack status is : 2 0 1

After inserting 1 the stack status is : 2 0 1

End to inserting the reference string.

Total page fault is 3.

Press any key to continue.

MENU

RESULT:

1. Input
2. LRU (Least Recently Used) Algorithm
0. Exit.

Enter your choice.0

The program for LRU page replacement was implanted and hence verified.

Ex.No:12 FILE ALLOCATION TECHNIQUE-CONTIGUOUS

AIM:

To Write a C Program to implement file allocation technique.

ALGORITHM:

1. Start the process
2. Declare the necessary variables
3. Get the number of files
4. Get the total no. of blocks that fit in to the file
5. Display the file name, start address and size of the file.
6. Stop the program.

PROGRAM:

```
#include<stdio.h>
void main()
{
 int i,j,n,block[20],start:
 printf("Enter the no. of file:\n");
 scanf("%d",&n);
printf("Enter the number of blocks needed for each file:\n");
for(i=0,i<n;i++)
scanf("%d",&block[i]);
start=0;
printf("\t\tFile name\tStart\tSize of file\t\t\n");
printf("\n\t\tFile1\t\t%d\t\t\t%d\n",start,block[0]);
for(i=2;i<=n;i++)
{
Start=start+block[i-2];
Printf("\t\tFile%d\t\t%d\t\tD\n",i,start,block[i-1]);
```

```
}
}
```

Output

```
Enter the number of  file:4
Enter the number of blocks needed for each file:
3
5
6
```

1

Filename	start	size of file
File 1	0	3
File 2	3	5
File 3	8	6
File 4	14	1

Result:

Thus the program for file allocation technique has been done successfully.