

**ST. JOSEPH’S COLLEGE OF ENGINEERING AND TECHNOLOGY, PALAI**

**4TH SEMESTER**

**CSL204 OPERATING SYSTEMS LAB**

**2019 Regulations**

LIST OF EXPERIMENTS

| **EXP. NO** | **NAME OF EXPERIMENT** | **COs** | **PAGE NO** |
| --- | --- | --- | --- |
| 1 | Basic Linux commands | CO1 | 3 |
| 2 | Shell programming | CO1 | 11 |
| 3 | System calls of Linux operating system | CO1 | 16 |
| 4 | I/O system calls of Linux operating system | CO1 | 20 |
| 5 | Inter Process Communication using Shared Memory | CO2 | 22 |
| 6 | Semaphores | CO2 | 25 |
| 7 | CPU scheduling algorithms. a) Round Robin b) SJF c) FCFS d) Priority | CO3 | 28 |
| 8 | Memory Allocation Methods for fixed partition a) First Fit b) Worst Fit c) Best Fi | CO4 | 37 |
| 9 | Page replacement algorithms a) FIFO b) LRU c) LFU | CO4 | 45 |
| 10 | Banker’s algorithm for deadlock avoidance | CO5 | 49 |
| 11 | Deadlock detection algorithm | CO5 | 54 |
| 12 | File allocation strategies. b) Sequential b) Indexed c) Linked | CO6 | 57 |
| 13 | Disk scheduling algorithms. a) FCFS b) SCAN c) C-SCAN | CO6 | 62 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**EXPERIMENT NO:1**

**BASIC LINUX COMMANDS**

**AIM**

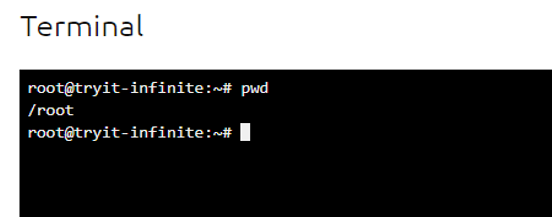
To implement basic Linux commands.

Command 1. pwd

Purpose: To know which is the current directory you are in.

Syntax-pwd

Output:

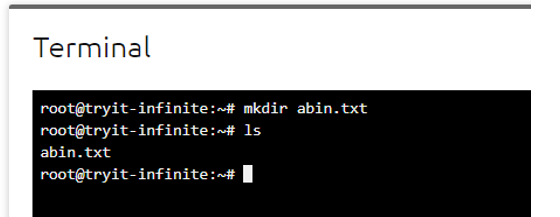


Command 2. ls

Purpose: To know the list of files and directories in the present directory.

Syntax: ls option

Output:



Options:

-r : to print the list in reverse order.

-R: to display the content of the sub-directories also.

-l: to display the details of the files and folders such as file permission, creation time etc.

~: to get the content of home directory

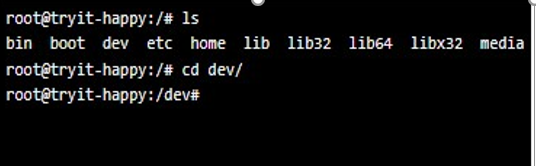
../: to get the content of parent directory.

Command 3. cd

Purpose: This command is used to go to a particular directory which is present in the current directory.

Syntax: cd [directory\_name]

Output:



Cd /: used to move to the root directory from current directory.

Cd ~: used to move to home directory

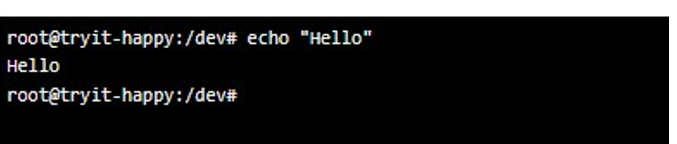
Cd ..: used to move to the parent directory of current directory.

Command 4. echo

Purpose:-Command in Linux is used to display line of text/string that are passed as an

argument.

Syntax: echo “[string]”

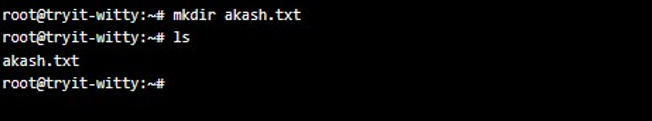
Output:

Command 5. mkdir

Purpose: Used to create a folder or a directory.

Syntax: mkdir [options...] [directories...]

Output:



Options

--version: It displays the version number, some information regarding the license and exits.

--help:  It displays the help related information and exits

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

Command 6. date

Purpose: Used to display the system date and time.

Syntax: date

Output:

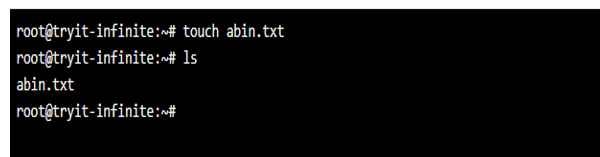


7. touch

Purpose: Is used to create, change and modify timestamps of a file.

Syntax: touch filename

Output:

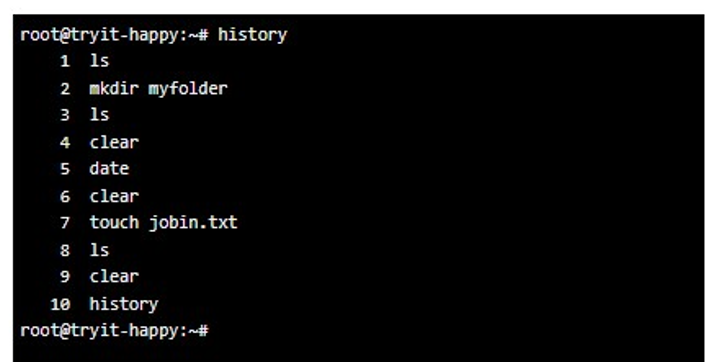


Command 8. history

Purpose: Used to view the previously executed command.

Syntax: history

Output:



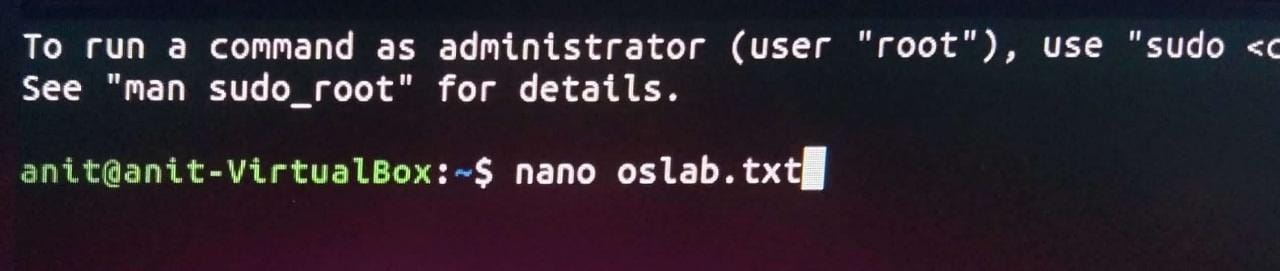
Command 9. nano

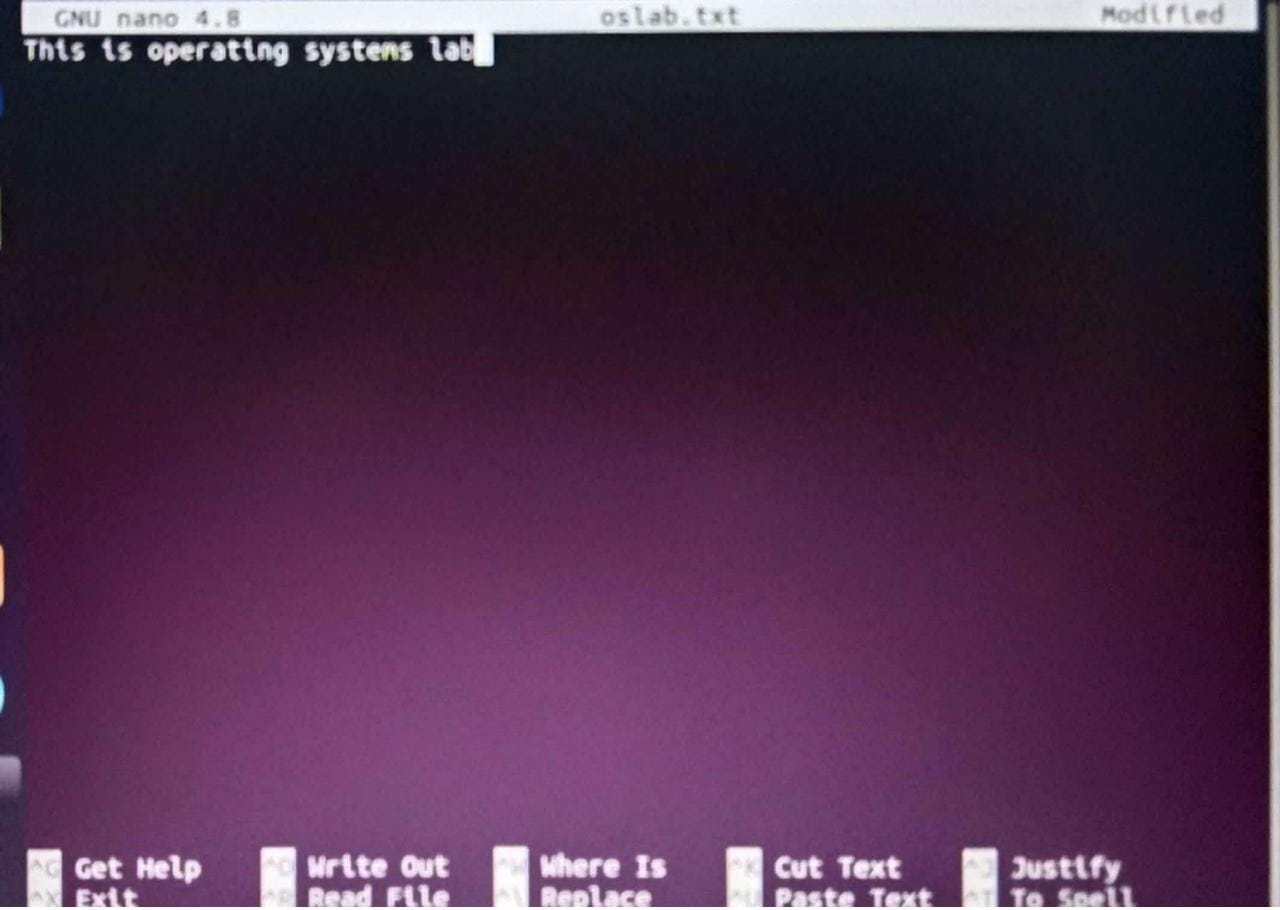
Purpose: The **nano** command is a good text editor that denotes keywords with colour

and can recognize most languages.

Syntax: nano new filename

Output:





Command 10. clear

Purpose: Is used to clear the screen

Syntax: clear

Command 11. sudo

Purpose: A widely used command in the Linux command line, **sudo** stands for

"SuperUser Do". So, if you want any command to be done with administrative

for root privileges, you can use the **sudo** command.

Syntax: sudo [command]

Command 12. cat

Purpose: Cat(concatenate) command is very frequently used in Linux. It reads data

from the file and gives their content as output. It helps us to create, view, and

concatenate files.

Syntax: cat filename

cat file1 file2

command 13. man

Purpose: man command in Linux is used to display the user manual of any command

that we can run on the terminal. It provides a detailed view of the command

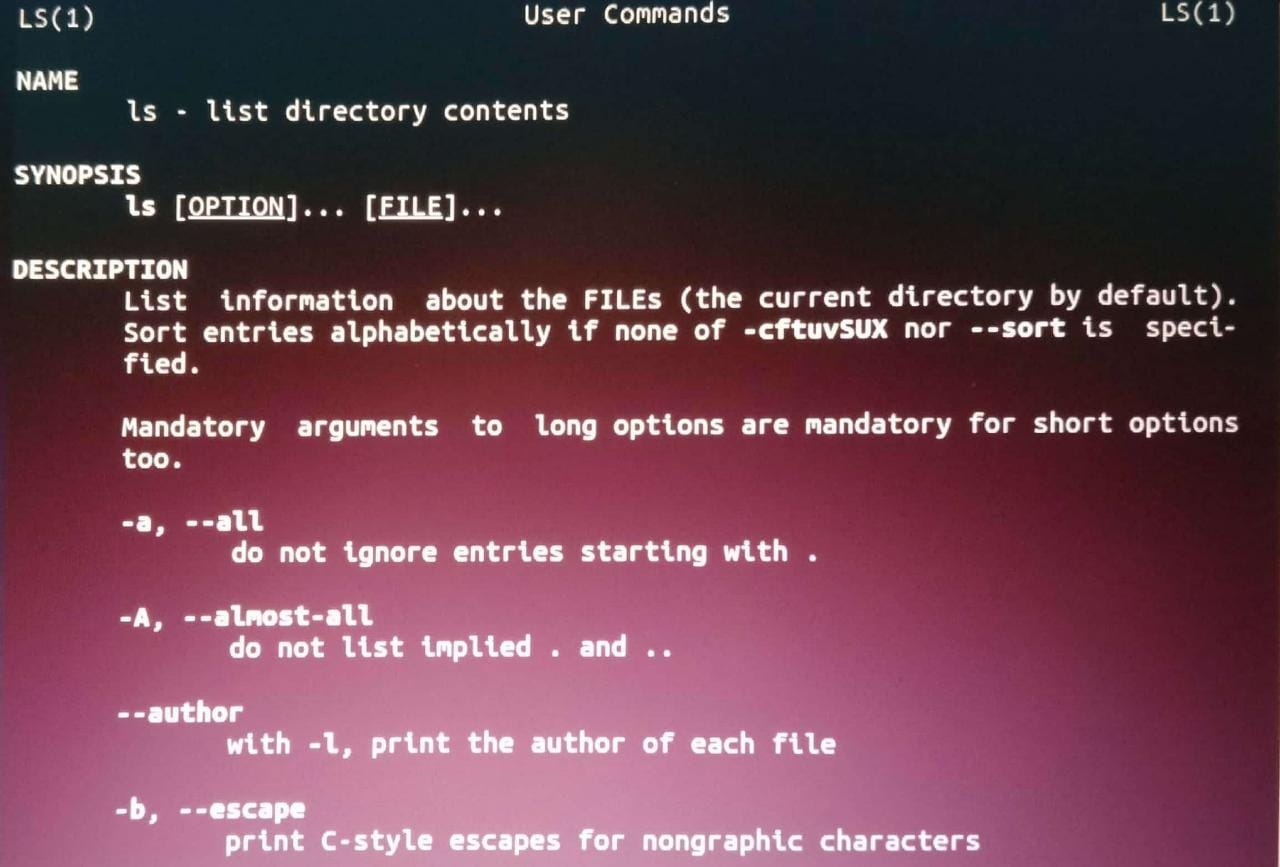
which includes NAME, SYNOPSIS, DESCRIPTION, OPTIONS, EXIT STATUS,

RETURN VALUES, ERRORS, FILES, VERSIONS, EXAMPLES, AUTHORS and

SEE ALSO.

Syntax: man [command name].

Output: man ls



Command 14. uname

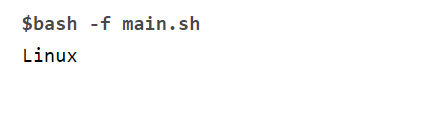
Purpose: Use **uname** to show the information about the system your Linux distro is

running. Using the command “**uname -a**” prints most of the information about

the system. This prints the kernel release date, version, processor type, etc.

Syntax-uname [option]

Output



Command 15. whoami

Purpose: whoami command is used both in Unix Operating System and as well as in

Windows Operating System. It is basically the concatenation of the strings

“who”,”am”,”i” as whoami. It displays the username of the current user when this

command is invoked. It is similar to running the id command with the options.

-un

Syntax-whoami

Output:



Command 16. rm

Purpose: rm command is used to remove objects such as files, directories, symbolic

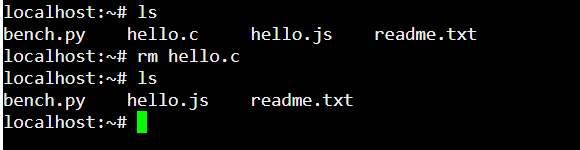
links and so on from the file system like UNIX. To be more precise, rm

removes references to objects from the filesystem, where those objects might

have had multiple references (for example, a file with two different names).

Syntax-rm [option].....[file]....

Output



Command 17. rmdir

Use: rmdir command is used remove empty directories from the filesystem in

Linux. The rmdir command removes each and every directory specified in the

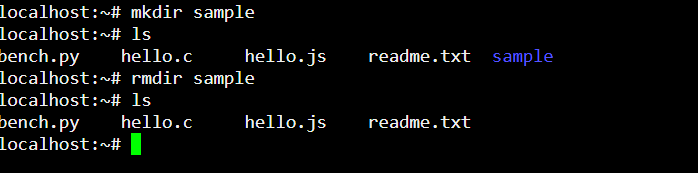
command line only if these directories are empty. So if the specified directory

has some directories or files in it then this cannot be removed by rmdir

command.

Syntax: rmdir [option].... Directory

Output



Command 18. cp

Use-The Linux cp command is used for copying files and directories to another

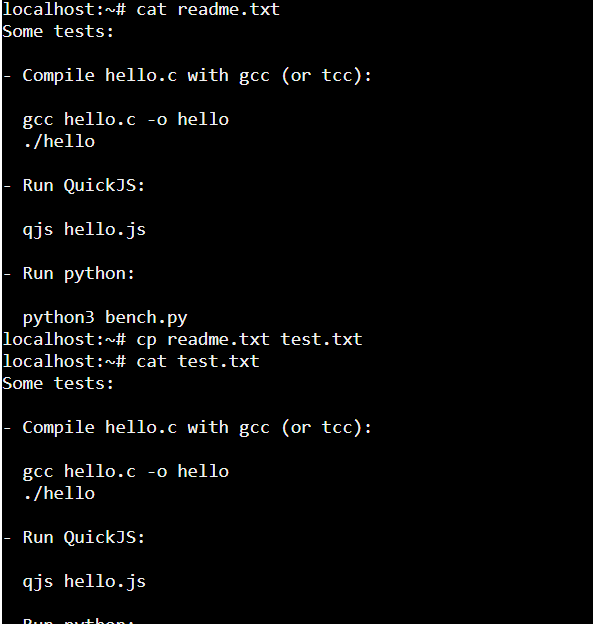
location. To copy a file, specify “cp” followed by the name of a file to copy.

Then, state the location at which the new file should appear. The new file does

not need to have the same name as the one you are copying

Syntax-cp [OPTION] Source Destination

cp [OPTION] Source Directory



**RESULT:**

Thus, the experiment to implement simple Linux commands has executed successfully and output verified.

**EXPERIMENT NO:2**

**SHELL PROGRAMMING**

**AIM**

To write simple functions with basic tests, loops, patterns using shell programs.

**PROGRAM**

1 .Write a program to display your name, semester,

batch, roll number and register number.

PROGRAM:

echo "Enter your name";

read name;

echo "Enter your semester";

read sem;

echo "Enter your batch";

read batch;

echo "Enter your roll number";

read rollno;

echo "Enter your register number";

read regno;

echo "Name: $name"

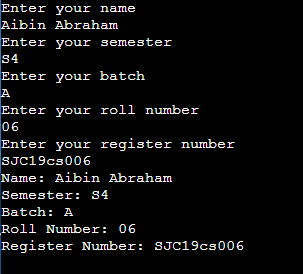
echo "Semester: $sem"

echo "Batch: $batch"

echo "Roll Number: $rollno"

echo "Register Number: $regno"

**OUTPUT:**



2. Write a program to display the sum of two numbers.

PROGRAM

echo "Enter the first number";

read a;

echo "Enter the second number";

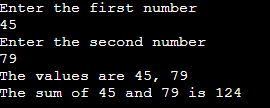
read b;

echo "The values are $a, $b";

sum=$(($a+$b));

echo "The sum of $a and $b is $sum"

OUTPUT



3. Write a program to display the biggest of two numbers.

PROGRAM

echo "Enter the first number:";

read x;

echo "Enter the second number:";

read y;

if(($x>$y))

then

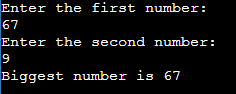
echo "Biggest number is $x";

else

echo "Biggest number is $y";

fi

OUTPUT:



4. Write a program to print the first 10 natural numbers

PROGRAM

echo "The first ten natural numbers are:"

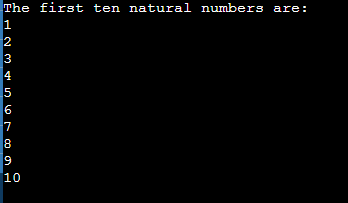
for((i=1;i<=10;i++))

do

echo $i

done

OUTPUT:



5. Write a program to find the factorial of a number.

PROGRAM

echo "Enter a number: "

read num

fact=1

for(( i=1; i<=num; i++ ))

do

fact=$[ $fact \* $i ]

done

echo "The factorial of $num is $fact"

OUTPUT:



6. Write a program to find the Fibonacci series up to n numbers.

PROGRAM

echo "Enter the number"

read n

x=0

y=1

echo "The Fibonacci series of $n is"

for((i=1;i<=n;i++))

do

echo "$x"

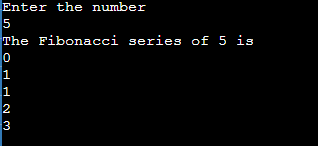
f=$(($x + $y))

x=$y

y=$f

done

OUTPUT:



7. Write a program to implement a simple calculator.

PROGRAM

echo "Enter the first number"

read n1

echo "Enter the second number"

read n2

echo "1.Addition"

echo "2.Subtraction"

echo "3.Multiplication"

echo "4.Division"

echo "Choose the operation(1-4)"

read op

case $op in

1)

rs=$(($n1 + $n2))

echo "The sum is $rs";;

2)

rs=$(($n1 - $n2))

echo "The difference is $rs";;

3)

rs=$(($n1 \* $n2))

echo "The product is $rs";;

4)

rs=$(($n1 / $n2))

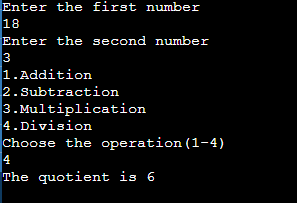
echo "The quotient is $rs";;

\*)

echo "Wrong choice entered";;

esac

OUTPUT:



**RESULT:**

The program to implement various shell programs has executed and output verified.

**EXPERIMENT NO: 3**

**SYSTEM CALLS OF LINUX OPERATING SYSTEM**

**AIM**

To implement system calls in Linux such as fork, exec, getpid, exit, wait, close, stat, opendir, readdir etc.

**ALGORITHM**

1. Start
2. Call fork()
3. If it returns zero print “child process start and ends”
4. Else call wait
5. When wait is called process executes the commands from the argument list and exit
6. Stop

**Program to implement fork(),exec(), getpid(), exit()**

#include <stdio.h>

#include<sys/types.h>

#include<sys/wait.h>

#include<unistd.h>

#include<stdlib.h>

int main()

{

int pid;

printf("getpid()\n The process id of current process: %d\n",getpid());

printf("fork() creates child process\n");

if(fork()!=0)

{

printf("Parent process starts and wait() executes\n");

wait(NULL);

printf("waiting and executes another funtion");

char \*args[]={"ls",NULL};

execvp(args[0],args);

exit(0);

printf("executes not ends and parent ends");

}

else

{

printf("Child process starts and ends\n");

}

printf("exit function\n");

exit(0);

printf("exit not executing");

return 0;

}

**OUTPUT**

getpid()

The process id of current process: 633

fork() creates child process

Parent process starts and wait() executes

Child process starts and ends

exit function

a.out main.c

**ALGORITHM**

1. Start
2. Create a stat pointer
3. Read the file name and assign to stat
4. Using the stat variable print the parameter values.
5. Stop

**Program to implement stat()**

#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 statistics 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("number of links:%d\n",nfile->st\_nlink);

}

**OUTPUT**  
enter name of file whose statistics has to main.c

user id 0

block size :4096

last access time 1632723792

time of last modification 1632723792

porduction mode 33279

size of file 1008

number of links:1

**ALGORITHM**

1. Start
2. Create a DIR pointer
3. Read the directory name
4. Open the directory
5. If it returns null print error
6. While there is content in the directory print the items
7. Close the directory
8. Stop

**Program to implement opendir(). readdir(), closedir()**

#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

**RESULT**

The program to implement different Linux system call has been implemented and output is verified.

**EXPERIMENT NO: 4**

**I/O SYSTEM CALLS OF LINUX**

**AIM**

Write programs using the I/O system calls of Linux operating system (open, read, write)

**ALGORITHM**

1. Start
2. Create a file in the read/write mode
3. If it returns -1 then error in opening the file.
4. Close the file and open it for writing
5. Write some content into the file
6. Close and open it for reding
7. Read the content and display on the monitor until EOF
8. Close the file
9. Stop

**PROGRAM**

#include<stdio.h>

#include<sys/types.h>

#include<sys/stat.h>

#include <fcntl.h>

int main()

{

int fd,sz;

fd=creat("file1.dat", S\_IREAD|S\_IWRITE);

if(fd==-1)

printf("Error in opening file1.dat\n");

else

{

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

printf("\nfile1.dat is currently empty");

}

close(fd);

fd = open("file1.txt", O\_WRONLY | O\_CREAT | O\_TRUNC, 0644);

if (fd < 0)

{

perror("r1");

exit(1);

}

sz = write(fd, "hello world\n", strlen("hello world\n"));

close(fd);

printf("Written something into file\n");

char \*c = (char \*) calloc(100, sizeof(char));

fd = open("file1.txt", O\_RDONLY);

if (fd < 0) { perror("r1"); exit(1); }

sz = read(fd, c, 12);

c[sz] = '\0';

printf("Those bytes are as follows: % s\n", c);

}

**OUTPUT**

file1.dat opened for read/write access

file1.dat is currently empty

Written something into file

Those bytes are as follows: hello world

**RESULT**

Thus, the program to implement Linux file system calls are implemented and output is verified.

**EXPERIMENT NO: 5**

**INTER PROCESS COMMUNICATION USING SHARED MEMEORY**

**AIM**

Implement programs for Inter Process Communication using Shared Memory

**ALGORITHM:**

Step 1: Start the process

Step 2: Declare the segment size

Step 3: Create the shared memory

Step 4: Read the data from the shared memory

Step 5: Write the data to the shared memory

Step 6: Edit the data

Step 7: Stop the process

**PROGRAM:**

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<string.h>

#include<sys/ipc.h>

#include<sys/shm.h>

#include<sys/types.h>

#define SEGSIZE 100

int main(int argc, char \*argv[ ])

{

int shmid,cntr; key\_t key;

char \*segptr;

char buff[]="poooda......";

key=ftok(".",'s');

if((shmid=shmget(key, SEGSIZE, IPC\_CREAT | IPC\_EXCL | 0666))== -1)

{

if((shmid=shmget(key,SEGSIZE,0))==-1)

{

perror("shmget");

exit(1);

}

}

else

{

printf("Creating a new shared memory seg \n");

printf("SHMID:%d",shmid);

}

system("ipcs –m");

if((segptr=(char\*)shmat(shmid,0,0))==(char\*)-1)

{

perror("shmat");

exit(1);

}

printf("Writing data to shared memory…\n");

strcpy(segptr,buff);

printf("DONE\n");

printf("Reading data from shared memory…\n");

printf("DATA:-%s\n",segptr);

printf("DONE\n");

printf("Removing shared memory Segment…\n"); if(shmctl(shmid,IPC\_RMID,0)== -1)

printf("Can‟t Remove Shared memory Segment…\n");

else

printf("Removed Successfully");

}

**OUTPUT**

Creating a new shared memory seg

------ Message Queues --------

key msqid owner perms used-bytes messages

------ Shared Memory Segments --------

key shmid owner perms bytes nattch status

0x7301e46b 1 runner6 666 100 0

------ Semaphore Arrays --------

key semid owner perms nsems

SHMID:1Writing data to shared memory…

DONE

Reading data from shared memory…

DATA:-Hai......

DONE

Removing shared memory Segment…

Removed Successfully

**RESULT**

Thus the program to perform inter-process communication has been executed and output is verified.

**EXPERIMENT NO:6**

**SEMAPHORES**

**AIM**

To implement producer consumer problem using semaphores.

**ALGORITHM:**

Step 1: Start the program.

Step 2: Declare the required variables.

Step 3: Initialize the buffer size and get maximum item you want to produce.

Step 4: Get the option, which you want to do either producer, consumer or exit from the operation.

Step 5: If you select the producer, check the buffer size if it is full the producer should not produce the item or otherwise produce the item and increase the value buffer size. Step 6: If you select the consumer, check the buffer size if it is empty the consumer should not consume the item or otherwise consume the item and decrease the value of buffer size.

Step 7: If you select exit come out of the program.

Step 8: Stop the program.

**PROGRAM:**

#include<stdio.h>

int mutex=1, full=0, empty=3, x=0;

main()

{

int n;

void producer();

void consumer();

int wait(int);

int signal(int);

printf("\n1.PRODUCER\n2.CONSUMER\n3.EXIT\n");

while(1)

{

printf("\nENTER YOUR CHOICE\n");

scanf("%d",&n); switch(n)

{

case 1:

if((mutex==1)&&(empty!=0))

producer();

else

printf("BUFFER IS FULL");

break;

case 2:

if((mutex==1)&&(full!=0))

consumer();

else

printf("BUFFER IS EMPTY");

break;

case 3:

exit(0);

break;

} } }

int wait(int s)

{

return(--s);

}

int signal(int s)

{ return(++s); }

void producer()

{ mutex=wait(mutex);

full=signal(full);

empty=wait(empty);

x++;

printf("\nproducer produces the item%d",x);

mutex=signal(mutex);

}

void consumer()

{

mutex=wait(mutex);

full=wait(full);

empty=signal(empty);

printf("\n consumer consumes item%d",x);

x--;

mutex=signal(mutex);

}

**OUTPUT**

1.PRODUCER

2.CONSUMER

3.EXIT

ENTER YOUR CHOICE

1

producer produces the item1

ENTER YOUR CHOICE

10

ENTER YOUR CHOICE

1

producer produces the item2

ENTER YOUR CHOICE

20

ENTER YOUR CHOICE

2

consumer consumes item2

ENTER YOUR CHOICE

2

consumer consumes item1

ENTER YOUR CHOICE

2

BUFFER IS EMPTY

ENTER YOUR CHOICE

**RESULT**

Thus, the program to implement the producer consumer problem using semaphore is implemented and output is verified.

**EXPERIMENT NO:7**

**SCHEDULING ALGORITHMS**

**AIM**

To implement different scheduling algorithm

1. **FIRST COME FIRST SERVED (FCFS)**

**ALGORITHM**

1. START the program

2. Get the number of processors

3. Get the Burst time of each processor

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**

#include<stdio.h>

void main()

{

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;

int i;

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");

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];

}

wt[0]=0;

for(i=1;i<n;i++)

{

wt[i]=wt[i-1]+bt[i-1];

tot\_wt = tot\_wt+wt[i];

}

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]);

printf("\n\nTotal Turn around Time:%d",tot\_tat);

printf("\nAverage Turn around Time :%d ", tot\_tat/n);

printf("\nTotal Waiting Time:%d",tot\_wt);

printf("\nTotal avg. Waiting Time:%d",tot\_wt/n);

}

**OUTPUT**

Enter the no.of processes

4

Enter burst time for each process

8

12

11

4

FCFS Algorithm

PROCESS BURST TIME TURN AROUND TIME WAITING TIME

process[0] 8 8 0

process[1] 12 20 8

process[2] 11 31 20

process[3] 4 35 31

Total Turn around Time:94

Average Turn around Time :23

Total Waiting Time:59

Total avg. Waiting Time:14

1. **SJF (SHORTEST JOB FIRST) CPU SCHEDULING**

**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**

#include<stdio.h>

void main()

{

int i,j;

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;

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;

}

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]);

}

}

}

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];

}

wt[0]=0;

for(i=1;i<n;i++)

{

wt[i]=wt[i-1]+bt[i-1];

tot\_wt = tot\_wt+wt[i];

}

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]);

printf("\n\nTotal Turn around Time:%d",tot\_tat);

printf("\nAverage Turn around Time :%d ", tot\_tat/n);

printf("\nTotal Waiting Time:%d",tot\_wt);

printf("\nTotal avg. Waiting Time:%d",tot\_wt/n);

}

int swap(int \*a, int \*b)

{

int t;

t = \*a;

\*a = \*b;

\*b = t;

return 0;

}

**OUTPUT**

Enter the no.of processes

4

Enter burst time for each process

8

5

4

7

PROCESS BURST TIME TURN AROUND TIME WAITING TIME

process[3] 4 4 0

process[2] 5 9 4

process[4] 7 16 9

process[1] 8 24 16

Total Turn around Time:53

Average Turn around Time :13

Total Waiting Time:29

Total avg. Waiting Time:7

1. **ROUND ROBIN 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**

#include<stdio.h>

int main()

{

int i, limit, total = 0, x, counter = 0, time\_quantum;

int wait\_time = 0, turnaround\_time = 0, burst\_time[10], temp[10];

float average\_wait\_time, average\_turnaround\_time;

printf("\nEnter Total Number of Processes:\t");

scanf("%d", &limit);

x = limit;

for(i = 0; i < limit; i++)

{

printf("\nEnter Details of Process[%d]\n", i + 1);

printf("Burst Time:\t");

scanf("%d", &burst\_time[i]);

temp[i] = burst\_time[i];

}

printf("\nEnter Time Quantum:\t");

scanf("%d", &time\_quantum);

printf("\nProcess ID\t\tBurst Time\t Turnaround Time\t Waiting Time\n");

for(total = 0, i = 0; x != 0;)

{

if(temp[i] <= time\_quantum && temp[i] > 0)

{

total = total + temp[i];

temp[i] = 0;

counter = 1;

}

else if(temp[i] > 0)

{

temp[i] = temp[i] - time\_quantum;

total = total + time\_quantum;

}

if(temp[i] == 0 && counter == 1)

{

x--;

printf("\nProcess[%d]\t\t%d\t\t %d\t\t\t %d", i + 1, burst\_time[i], total,

total - burst\_time[i]);

wait\_time = wait\_time + total - burst\_time[i];

turnaround\_time = turnaround\_time + total ;

counter = 0;

}

if(i == limit - 1)

{

i = 0;

}

else

{

i++;

}

}

average\_wait\_time = wait\_time \* 1.0 / limit;

average\_turnaround\_time = turnaround\_time \* 1.0 / limit;

printf("\n\nAverage Waiting Time:\t%f", average\_wait\_time);

printf("\nAvg Turnaround Time:\t%f\n", average\_turnaround\_time);

}

**OUTPUT**

Enter Total Number of Processes: 2

Enter Details of Process[1]

Burst Time: 4

Enter Details of Process[2]

Burst Time: 5

Enter Time Quantum: 3

Process ID Burst Time Turnaround Time Waiting Time

Process[1] 4 7 3

Process[2] 5 9 4

Average Waiting Time: 3.500000

Avg Turnaround Time: 8.000000

1. **PRIORITY SCHEDULING**

#include<stdio.h>

void main()

{

int i,j,k;

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;

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;

}

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]);

}

}

}

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];

}

wt[0]=0;

for(i=1;i<n;i++)

{

wt[i]=wt[i-1]+bt[i-1];

tot\_wt = tot\_wt+wt[i];

}

printf("\nPROCESS\t\tBURST TIME\tPRIORITY\tTURN AROUND TIME\tWAITINGTIME");

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]);

printf("\n\nTotal Turn around Time:%d",tot\_tat);

printf("\nAverage Turn around Time :%d ", tot\_tat/n);

printf("\nTotal Waiting Time:%d",tot\_wt);

printf("\nTotal avg. Waiting Time:%d",tot\_wt/n);

}

int swap(int \*a, int \*b)

{

int t;

t = \*a;

\*a = \*b;

\*b = t;

return 0;

}

**OUTPUT**

Enter the no.of processes

4

Enter burst time and priority of process[1]:5

2

Enter burst time and priority of process[2]:4

3

Enter burst time and priority of process[3]:6

1

Enter burst time and priority of process[4]:3

4

PROCESS BURST TIME PRIORITY TURN AROUND TIME WAITINGTIME

process[3] 6 1 6 0

process[1] 5 2 11 6

process[2] 4 3 15 11

process[4] 3 4 18 15

Total Turn around Time:50

Average Turn around Time :12

Total Waiting Time:32

Total avg. Waiting Time:8

**RESULT:**

The programs to implement different scheduling algorithm has been executed and output is verified.

**EXPERIMENT NO: 8**

**MEMORY ALLOCATION METHODS**

**AIM**

Implementation of the Memory Allocation Methods for fixed partition using a) First Fit b) Worst Fit c) Best Fit algorithms

**FIRST FIT:**

**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];

111

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 5

enter the size for 5 blocks

enter the size for 0 block 20

enter the size for 1 block 50

enter the size for 2 block 40

enter the size for 3 block 30

enter the size for 4 block 80

enter the number of process 3

enter the size of 3 processors !

enter the size of process 0 25

enter the size of process 1 30

enter the size of process 2 45

Allocation of blocks using first fit is as follows

process process size blocks

0 25 50

1 30 40

2 45 80

**BESTFIT**:

**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 previous 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++)

111

{

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 20

Enter the size of memory block 2 50

Enter the size of memory block 3 40

Enter the size of memory block 4 30

Enter the size of memory block 5 80

enter the number of files3

enetr the size of file 1 30

enetr the size of file 2 20

enetr the size of file 3 45

file no file.size block no block size

1 30 4 30

2 20 1 20

3 45 2 50

**WORST FIT**

**ALGORITHM**

1. Start
2. Read the number of blocks and block sizes
3. Read the number of process and process sizes.
4. Find the block which has maximum size if the process can be allocated to it, allocate, else show message memory not enough
5. Repeat this process for every process.
6. Stop

**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, max=0,index;

printf("\n Worst 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 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++)

{

max=0;

for(j=0;j<n;j++)

{

if(s[j].alloc!=1)

{

if (max<s[j].size)

{

max=s[j].size;

index=j;

}

}

}

if(p[i].flag!=1)

{

if(p[i].ps<=s[index].size)

{

p[i].flag=1;

s[index].alloc=1;

}

}

printf("\n\t\t %d\t\t\t%d\t%d\t",i,p[i].ps,s[index].size);

}

for(i=0;i<np;i++)

{

if(p[i].flag!=1)

printf("sorry !!!!!!!process %d must wait as there is no sufficient memory",i);

}

}

**OUTPUT**

Worst fit

enter the number of blocks 5.

enter the size for 5 blocks

enter the size for 0 block 20

enter the size for 1 block 50

enter the size for 2 block 40

enter the size for 3 block 30

enter the size for 4 block 80

enter the number of process 3

enter the size of 3 processors !

enter the size of process 0 35

enter the size of process 1 45

enter the size of process 2 25

Allocation of blocks using first fit is as follows

process process size blocks

0 35 80

1 45 50

2 25 40

**RESULT**

Thus, the program to implement different memory allocation scheme has implemented and output is verified.

**EXPERIMENT NO: 9**

**PAGE REPLACEMENT SCHEMES**

**AIM**

Implement page replacement algorithms a) FIFO b) LRU c) LFU

**(a)FIFO**:

**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

**(b)LRU:**

**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.

RESULT:

MENU

1. Input

2. LRU (Least Recently Used) Algorithm

0. Exit.

Enter your ch 0

**RESULT**

Thus, the program to implement different page replacement algorithm has been implemented and output verified.

**EXPERIMENTS NO: 10**

**BANKERS ALGORITHM**

**AIM**

Implement the banker’s algorithm for deadlock avoidance.

**ALGORITHM**

1. Start
2. Enter the number of instances of resources and the allocated and maximum matrix
3. Find the need matrix
4. Calculate the available matrix
5. If need is greater than available, the system is in unsafe state
6. Else if the process can proceed with the resources, complete and release the resources.
7. Repeat this for all process
8. Stop

**PROGRAM**

#include <stdio.h>

int main()

{

// P0, P1, P2, P3, P4 are the Process names here

int avail[10], alloc[10][10], max[10][10], need[10][10], maxres[10], m, n, i,j,k,sum;

printf("\nEnter the number of processes and the number of resources:\n");

scanf("%d%d", &n, &m);

printf("\nEnter maximum instances of resources\n");

for (j = 0; j < m; j++)

{

scanf("%d", &maxres[j]);

avail[j] = maxres[j];

}

printf("\nEnter the Allocated Matrix:\n");

for (i = 0; i < n; i++)

{

for (j = 0; j < m; j++)

scanf("%d", &alloc[i][j]);

}

printf("\nEnter the Max Matrix:\n");

for (i = 0; i < n; i++)

{

for (j = 0; j < m; j++)

{

scanf("%d", &max[i][j]);

need[i][j] = max[i][j] - alloc[i][j];

}

}

printf("\nThe Need Matrix is:\n");

for (i = 0; i < n; i++)

{

for (j = 0; j < m; j++)

printf("%d ", need[i][j]);

printf("\n");

}

for (j = 0; j < m; j++) //calculating available matrix after allocation

{

sum = 0;

for (i = 0; i < n; i++)

sum += alloc[i][j];

avail[j] -= sum;

}

int finish[10],safeseq[10], ind = 0;

for (k = 0; k < n; k++)

{

finish[k] = 0;

}

int y = 0;

for (k = 0; k < n; k++)

{

for (i = 0; i < n; i++)

{

if (finish[i] == 0)

{

int flag = 0;

for (j = 0; j < m; j++)

{

if (need[i][j] > avail[j])

{

flag = 1;

break;

}

}

if (flag == 0)

{

safeseq[ind++] = i;

for (y = 0; y < m; y++)

avail[y] += alloc[i][y];

finish[i] = 1;

}

}

}

}

for (i = 0; i < n; i++)

{

if (finish[i] == 0)

{

printf("system is in unsafe state.");

return(0);

}

printf("Following is the SAFE Sequence\n");

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

printf(" P%d ->", safeseq[i]);

printf(" P%d", safeseq[n - 1]);

return(0);

}

}

**OUTPUT**

Enter the number of processes and the number of resources:

2

3

Enter maximum instances of resources

5

7

6

Enter the Allocated Matrix:

2 0 3

1 2 5

Enter the Max Matrix:

3 1 3

4 3 5

The Need Matrix is:

1 1 0

3 1 0

system is in unsafe state.

**RESULT**

Thus the program to implement Bankers algorithm has been implemented and output is verified.

**EXPERIMENT NO: 11**

**IMPLEMENTATION OF DEADLOCK DETECTION ALGORITHM**

**AIM**

To implement deadlock detection algorithm

**ALGORITHM**

1. Mark each process that has a row in the Allocation matrix of all zeros.
2. Initialize a temporary vectorW to equal the Available vector.
3. Find an indexi such that processi is currently unmarked and thei th row ofQ  
   is less than or equal to W . That is,Q ik … Wk, for 1 … k … m . If no such row is  
   found, terminate the algorithm.
4. If such a row is found, mark processi and add the corresponding row of the  
   allocation matrix to W . That is, setWk = Wk + Aik, for 1 … k … m . Return  
   to step 3.

**PROGRAM**

#include<stdio.h>

{  
static int mark[20];  
int i,j,np,nr;  
int main()  
{  
int alloc[10][10],request[10][10],avail[10],r[10],w[10];  
printf("\nEnter the no of process: ");  
scanf("%d",&np);  
printf("\nEnter the no of resources: ");  
scanf("%d",&nr);  
for(i=0;i<nr;i++)  
{  
printf("\nTotal Amount of the Resource R%d: ",i+1);  
scanf("%d",&r[i]);  
}  
printf("\nEnter the request matrix:");for(i=0;i<np;i++)  
for(j=0;j<nr;j++)  
scanf("%d",&request[i][j]);  
printf("\nEnter the allocation matrix:");  
for(i=0;i<np;i++)  
for(j=0;j<nr;j++)  
scanf("%d",&alloc[i][j]);  
/\*Available Resource calculation\*/  
for(j=0;j<nr;j++)  
{  
avail[j]=r[j];  
for(i=0;i<np;i++)  
{  
avail[j]-=alloc[i][j];  
}}  
//marking processes with zero allocation  
  
for(i=0;i<np;i++)  
{  
int count=0;  
 for(j=0;j<nr;j++)  
   {  
      if(alloc[i][j]==0)  
        count++;  
      else  
        break;  
    }  
 if(count==nr)  
 mark[i]=1;  
}  
// initialize W with avail  
for(j=0;j<nr;j++)  
    w[j]=avail[j];  
//mark processes with request less than or equal to W  
for(i=0;i<np;i++)  
{  
int canbeprocessed=0;  
 if(mark[i]!=1)  
{  
   for(j=0;j<nr;j++)  
    {  
      if(request[i][j]<=w[j])  
        canbeprocessed=1;  
      else  
         {  
         canbeprocessed=0;  
         break;  
          }  
     }  
if(canbeprocessed)  
{  
mark[i]=1;  
  
for(j=0;j<nr;j++)  
w[j]+=alloc[i][j];  
}}}  
  
//checking for unmarked processes  
int deadlock=0;  
for(i=0;i<np;i++)  
if(mark[i]!=1)  
deadlock=1;  
if(deadlock)  
printf("\n Deadlock detected");  
else  
printf("\n No Deadlock possible");  
}

**OUTPUT**

Enter the no of process: 4  
Enter the no of resources: 5  
  
Total Amount of the Resource R1: 2  
Total Amount of the Resource R2: 1  
Total Amount of the Resource R3: 1  
Total Amount of the Resource R4: 2  
Total Amount of the Resource R5: 1  
  
Enter the request matrix:0 1 0 0 1  
0 0 1 0 1  
0 0 0 0 1  
1 0 1 0 1  
  
Enter the allocation matrix:1 0 1 1 0  
1 1 0 0 0  
0 0 0 1 0  
0 0 0 0 0  
  
 Deadlock detected

**RESULT**

Thus, the program to implement deadlock detection algorithm has been implemented and output is verified.

**EXPERIMENT NO:12**

**FILE ALLOCTAION STRATEGIES**

**AIM**

To simulate file allocation strategies. b) Sequential b) Indexed c) Linked

1. Sequential File Allocation Strategy

**ALGORITHM**

1. Start the program
2. Get the number of files.
3. Get the memory requirement of each file.
4. Allocate the required locations to each in sequential order

a). Randomly select a location from available location s1= random(100);

b). Check whether the required locations are free from the selected location.

c). Allocate and set flag=1 to the allocated locations.

5. Print the results fileno, length, Blocks allocated.

6. Stop the program.

**PROGRAM**

#include<stdio.h>

main()

{

int f[50],i,st,j,len,c,k;

clrscr();

for(i=0;i<50;i++)

f[i]=0;

X:

printf("\n Enter the starting block & length of file");

scanf("%d%d",&st,&len);

for(j=st;j<(st+len);j++)

if(f[j]==0)

{

f[j]=1;

printf("\n%d->%d",j,f[j]);

}

else

{

printf("Block already allocated");

break;

}

if(j==(st+len))

printf("\n the file is allocated to disk");

printf("\n if u want to enter more files?(y-1/n-0)");

scanf("%d",&c);

if(c==1)

goto X;

else

exit();

getch();

}

**OUTPUT**

Enter the starting block & length of file 4 10

4->1

5->1

6->1

7->1

8->1

9->1

10->1

23

11->1

12->1

13->1

The file is allocated to disk

If you want to enter more files? (Y-1/N-0)

1. **Linked File Allocation Strategy**

**ALGORITHM**

1. Start the Program
2. Get the number of files.
3. Allocate the required locations by selecting a location randomly
4. Check whether the selected location is free.
5. If the location is free allocate and set flag =1 to the allocated locations.
6. Print the results file no, length, blocks allocated.
7. Stop the execution

**PROGRAM**

#include<stdio.h>

main()

{

int f[50],p,i,j,k,a,st,len,n,c;

clrscr();

for(i=0;i<50;i++)

f[i]=0;

printf("Enter how many blocks that are already allocated");

scanf("%d",&p);

printf("\nEnter the blocks no.s that are already allocated");

for(i=0;i<p;i++)

{

scanf("%d",&a);

f[a]=1;

}

X:

printf("Enter the starting index block & length");

scanf("%d%d",&st,&len);

k=len;

for(j=st;j<(k+st);j++)

{

if(f[j]==0)

{

f[j]=1;

printf("\n%d->%d",j,f[j]);

}

else

{

printf("\n %d->file is already allocated",j);

k++;

}

}

printf("\n If u want to enter one more file? (yes-1/no-0)");

scanf("%d",&c);

if(c==1)

goto X;

else

exit();

getch( );}

**OUTPUT:**

Enter how many blocks are already allocated 3

Enter the blocks no’s that are already allocated 4 7 9

Enter the starting index block & length 3

7

3-> 1

4-> File is already allocated

5->1

6->1

7-> File is already allocated

8->1

9-> File is already allocated

10->1

11->1

12->1

If u want to enter one more file? (yes-1/no-0)

1. **Indexed File Allocation**

**ALGORITHM**

1. Start the Program
2. Get the number of files.
3. Get the memory requirement of each file.
4. Allocate the required locations by selecting a location randomly.
5. Print the results file no,length, blocks allocated.
6. Stop the execution

**PROGRAM**

#include<stdio.h>

int f[50],i,k,j,inde[50],n,c,count=0,p;

main()

{

clrscr();

for(i=0;i<50;i++)

f[i]=0;

x:

printf("enter index block\t");

scanf("%d",&p);

if(f[p]==0)

{

f[p]=1;

printf("enter no of files on index\t");

scanf("%d",&n);

}

else

{

printf("Block already allocated\n");

goto x;

}

for(i=0;i<n;i++)

scanf("%d",&inde[i]);

for(i=0;i<n;i++)

if(f[inde[i]]==1)

{

printf("Block already allocated");

goto x;

}

for(j=0;j<n;j++)

f[inde[j]]=1;

printf("\n allocated");

printf("\n file indexed");

for(k=0;k<n;k++)

printf("\n %d->%d:%d",p,inde[k],f[inde[k]]);

printf(" Enter 1 to enter more files and 0 to exit\t");

scanf("%d",&c);

if(c==1)

goto x;

else

exit();

getch();

}

**OUTPUT:**

Enter index block 9

Enter no of files on index 3

1 2 3

Allocated

File indexed

9-> 1:1

9-> 2:1

9->3:1

Enter 1 to enter more files and 0 to exit

**RESULT**

Thus the program to simulate different file allocation strategies has implemented and output is verified.

**EXPERIMENT NO: 13**

**SIMULATION OF DISK SCHEDULING ALGORITHMS**

**AIM**

Simulate disk scheduling algorithms. a) FCFS b) SCAN c) C-SCAN

**ALGORITHM**

Step 1: Start

Step 2: read head

Step 3: read n

Step 4: read disk request

Step 5: a[0]=head

Step 6: for(i=0;i<n;i++)

distance=(a[i]>a[i+1])?a[i]-a[i+1]:a[i+1]-a[i]

print a[i],a[i+1],distance

seektime=seektime+distance

Step 7: print seektime

Step 8: Stop

**PROGRAM**

#include<stdio.h>

int head,a[20],i,distance,n,seektime;

int main()

{

printf("\nEnter Head position:");

scanf("%d",&head);

printf("\nEnter number of disk requests:");

scanf("%d",&n);

printf("\nEnter the disk requests:");

for(i=1; i<=n; i++)

scanf("%d",&a[i]);

a[0]=head;

printf("\*\*\*\*\*FCFS DISK SCHEDULING\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

for(i=0;i<n;i++)

{

distance=(a[i]>a[i+1])?a[i]-a[i+1]:a[i+1]-a[i];

printf("Head movement from %d to %d : %d\n",a[i],a[i+1],distance);

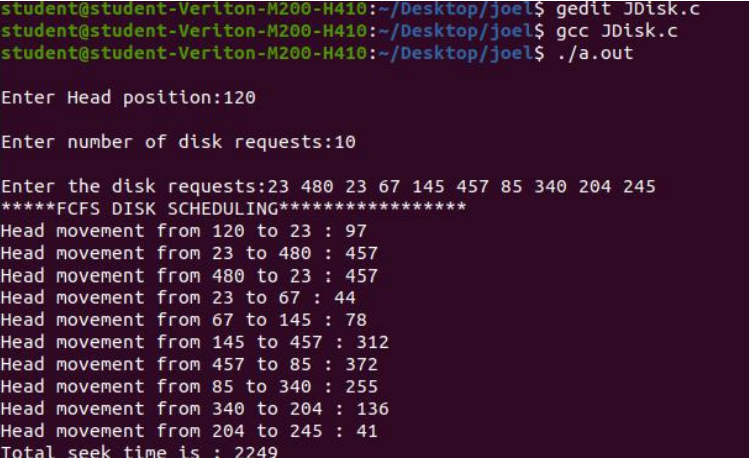
seektime=seektime+distance;

}

printf("Total seek time is : %d\n",seektime);

}

**OUTPUT**



**b) C-SCAN algorithm**

**ALGORITHM**

1. Start
2. Read the number of request, sequence of requests, initial head position, total disk size, and the ehad movement direction.
3. Sort the request sequence
4. Find out the index of the sequence which is larger than the current location.
5. Move the head to the extreme end of that direction
6. Move to the other end of extreme position.
7. Print the total head movement
8. Stop

**PROGRAM**

#include<stdio.h>

#include<stdlib.h>

int main()

{

int RQ[100],i,j,n,TotalHeadMoment=0,initial,size,move;

printf("Enter the number of Requests\n");

scanf("%d",&n);

printf("Enter the Requests sequence\n");

for(i=0;i<n;i++)

scanf("%d",&RQ[i]);

printf("Enter initial head position\n");

scanf("%d",&initial);

printf("Enter total disk size\n");

scanf("%d",&size);

printf("Enter the head movement direction for high 1 and for low 0\n");

scanf("%d",&move);

// logic for C-Scan disk scheduling

/\*logic for sort the request array \*/

for(i=0;i<n;i++)

{

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

{

if(RQ[j]>RQ[j+1])

{

int temp;

temp=RQ[j];

RQ[j]=RQ[j+1];

RQ[j+1]=temp;

}

}

}

int index;

for(i=0;i<n;i++)

{

if(initial<RQ[i])

{

index=i;

break;

}

}

// if movement is towards high value

if(move==1)

{

for(i=index;i<n;i++)

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);

initial=RQ[i];

}

// last movement for max size

TotalHeadMoment=TotalHeadMoment+abs(size-RQ[i-1]-1);

/\*movement max to min disk \*/

TotalHeadMoment=TotalHeadMoment+abs(size-1-0);

initial=0;

for( i=0;i<index;i++)

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);

initial=RQ[i];

}

}

// if movement is towards low value

else

{

for(i=index-1;i>=0;i--)

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);

initial=RQ[i];

}

// last movement for min size

TotalHeadMoment=TotalHeadMoment+abs(RQ[i+1]-0);

/\*movement min to max disk \*/

TotalHeadMoment=TotalHeadMoment+abs(size-1-0);

initial =size-1;

for(i=n-1;i>=index;i--)

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);

initial=RQ[i];

}

}

printf("Total head movement is %d",TotalHeadMoment);

return 0;

}

**OUTPUT**

Enter the number of Request

8

Enter the Requests Sequence

95 180 34 119 11 123 62 64

Enter initial head position

50

Enter total disk size

200

Enter the head movement direction for high 1 and for low 0

1

Total head movement is 382

**c) SCAN**

**ALGORITHM**

1. Start
2. Read maximum range of disk, number of queue requests, initial head position
3. Read the disk position to read.
4. Now if the requested position is greater than current head position, then push that to array queue1
5. Else if temp < current head position, then push to array queue2[]
6. Sort array queue1 and queue2
7. Copy queue1 to queue.
8. Setting queue[i] to maxrange because the head goes to end of disk and comes back in scan Algorithm
9. Copy queue2 to queue
10. Setting queue[i] to 0. Because that is the innermost cylinder.
11. At this point, we have the queue[] with the requests in the correct order of execution as per scan algorithm.
12. Now we have to set 0th index of queue[] to be the initial headposition.
13. Calculating SEEK TIME. seek is initially set to 0 in the declaration part.
14. start from the current headposition to last
15. Finding the difference between next position and current position.
16. Adding difference to the current seek time value
17. Displaying a message to show the movement of disk head
18. Calculating Average Seek time
19. Display Total and Average Seek Time(s)
20. Stop

**PROGRAM**

#include<stdio.h>

int absoluteValue(int); // Declaring function absoluteValue

void main()

{

int queue[25],n,headposition,i,j,k,seek=0, maxrange,

difference, temp, queue1[20], queue2[20], temp1=0, temp2=0;

float averageSeekTime;

// Reading the maximum Range of the Disk.

printf("Enter the maximum range of Disk: ");

scanf("%d", &maxrange);

// Reading the number of Queue Requests (Disk access requests)

printf("Enter the number of queue requests: ");

scanf("%d",&n);

// Reading the initial head position. (ie. the starting point of execution)

printf("Enter the initial head position: ");

scanf("%d", &headposition);

// Reading disk positions to be read in the order of arrival

printf("Enter the disk positions to be read(queue): ");

for(i=1;i<=n;i++) // Note that i varies from 1 to n instead of 0 to n-1

{

scanf("%d",&temp); //Reading position value to a temporary variable

//Now if the requested position is greater than current headposition,

//then pushing that to array queue1

if(temp>headposition)

{

queue1[temp1]=temp; //temp1 is the index variable of queue1[]

temp1++; //incrementing temp1

}

else //else if temp < current headposition,then push to array queue2[]

{

queue2[temp2]=temp; //temp2 is the index variable of queue2[]

temp2++;

}

}

//Now we have to sort the two arrays

//SORTING array queue1[] in ascending order

for(i=0;i<temp1-1;i++)

{

for(j=i+1;j<temp1;j++)

{

if(queue1[i]>queue1[j])

{

temp=queue1[i];

queue1[i]=queue1[j];

queue1[j]=temp;

}

}

}

//SORTING array queue2[] in descending order

for(i=0;i<temp2-1;i++)

{

for(j=i+1;j<temp2;j++)

{

if(queue2[i]<queue2[j])

{

temp=queue2[i];

queue2[i]=queue2[j];

queue2[j]=temp;

}

}

}

//Copying first array queue1[] into queue[]

for(i=1,j=0;j<temp1;i++,j++)

{

queue[i]=queue1[j];

}

//Setting queue[i] to maxrange because the head goes to

//end of disk and comes back in scan Algorithm

queue[i]=maxrange;

//Copying second array queue2[] after that first one is copied, into queue[]

for(i=temp1+2,j=0;j<temp2;i++,j++)

{

queue[i]=queue2[j];

}

//Setting queue[i] to 0. Because that is the innermost cylinder.

queue[i]=0;

//At this point, we have the queue[] with the requests in the

//correct order of execution as per scan algorithm.

//Now we have to set 0th index of queue[] to be the initial headposition.

queue[0]=headposition;

// Calculating SEEK TIME. seek is initially set to 0 in the declaration part.

for(j=0; j<=n; j++) //Loop starts from headposition. (ie. 0th index of queue)

{

// Finding the difference between next position and current position.

difference = absoluteValue(queue[j+1]-queue[j]);

// Adding difference to the current seek time value

seek = seek + difference;

// Displaying a message to show the movement of disk head

printf("Disk head moves from position %d to %d with Seek %d \n",

queue[j], queue[j+1], difference);

}

// Calculating Average Seek time

averageSeekTime = seek/(float)n;

//Display Total and Average Seek Time(s)

printf("Total Seek Time= %d\n", seek);

printf("Average Seek Time= %f\n", averageSeekTime);

}

// Defining function absoluteValue

int absoluteValue(int x)

{

if(x>0)

{

return x;

}

else

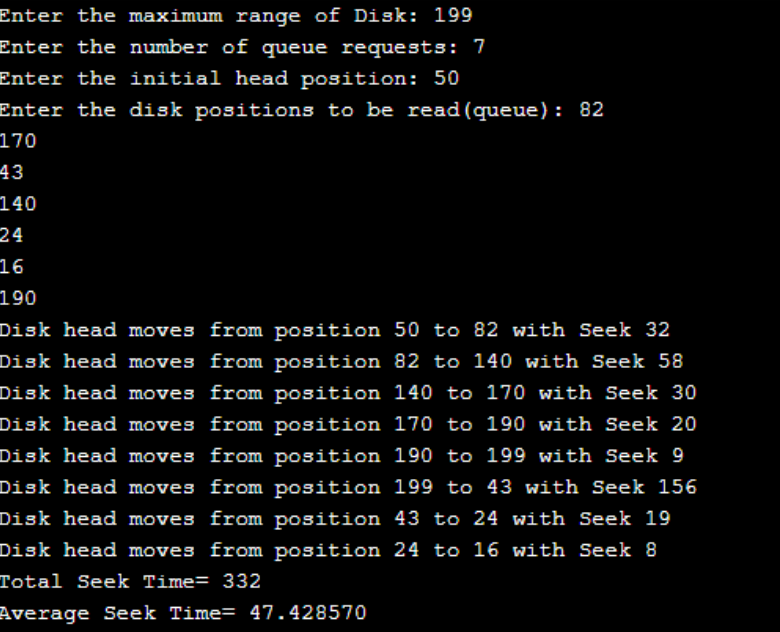
{

return x\*-1;

}

}

**OUTPUT**



**RESULT**

Thus the program to implement different disk scheduling algorithm has been implemented and output verified.