EXP-1 STUDY OF BASIC LINUX AND VI EDITOR COMMANDS

AIM: To study and execute basic linux and vi editor commands.

PROGRAM:

File Handling commands

• **mkdir -** make directories
Usage: mkdir [OPTION] DIRECTORY...
eg. mkdirprabhat
• **ls - list directory contents**
Usage: ls [OPTION]... [FILE]...
eg. ls, ls l,
ls prabhat
• **cd** - changes directories
Usage: cd [DIRECTORY]
eg. cd prabhat
• **pwd -** printname of current working directory
Usage: pwd
• vim – vi improved, A programmers text editor
Usage: vim [OPTION] [file]…
Eg. vim file1.txt
• cp – copy files and directories
Uasge: cp[OPTION]…SOURCE DEST
Eg. cp sample.txt sample_copy.txt
cp smaple_copy.txt target_dir
• mv – move (rename) files
Uasge: mv [OPTION]…SOURCE DEST
Eg.mv source.txt target_dir
Mv old.txt new.txt
• rm – remove
Files or directories
Usage: rm[OPTION]…[file]
Eg. rm file1.txt, rm rf some_dir
• find – search for files in a directory hierarchy
Usage: find [OPTION] [path] [pattern]
Eg. find file1.txt, find name file1.txt
• history – prints recently used commands
Usage: history

Text Processing

• **cat**– concatenate files and print on the standard output
Usage: cat [OPTION] [FILE]...
eg. cat file1.txt file2.txt
cat nfile1.txt
• **echo**– display a line of text
Usage: echo [OPTION] [string] ...
eg. echo I love India
echo $HOME
• **wc – print the number of newlines, words, and bytes in files**
Usage: wc [OPTION]…[FILE]…
Eg. wc file1.txt
wc L file1.txt
• **sort – sort lines of text files**
Usage: sort [OPTION]…[FILE]…
Eg. sort file1.txt
sort r file1.txt

System Administration

• **chmod – change file access permissions**
Usage: chmod [OPTION] [MODE] [FILE]
Eg. chmod 744 calculate.sh
• **chown – change file owner and group**
Usage: chown [OPTION]… OWNER[:[GROUP]] FILE…
Eg. chown remo myfile.txt
• **su – change user ID**
Usage: su [OPTION] [LOGIN]
Eg. su remo, su
• **passwd – update a user's authentication token(s)**
Usage: passwd [OPTION]
Eg. passwd
• **who – show who is logged on**
Usage: who [OPTION]
Eg. who , who b, who q

Process management

- **ps – report a snapshot of the current processes**

Usage: who [OPTION]
Eg. ps, ps el

- **kill – to kill a process(using signal mechanism)**

Usage: kill [OPTION] pid
Eg. kill 9

Archival

- **tar – to achieve a file**

Usage: tar [OPTION] DEST SOURCE
Eg. tar cvf
/home/archive.tar /home/original tar
xvf/ home/archive.tar

- **zip – package and compress (archive) files**

Usage: zip [OPTION] DEST SOURCE
Eg. zip original.zip.original

- **unzip – list, test, and extract compressed files in a ZIP archive**

Usage: unzip filename
Eg. unzip original.zip

- **du – estimate file space usage**

Usage: du [OPTION] … [FILE]…
Eg. du

- **df – report filesystem disk space usage**

Usage: df [OPTION]… [FILE]…
Eg. df

- **quota – display disk usage and limits**

Usage: quota [OPTION]
Eg. quota v

Advanced Commands

- **reboot**– reboot the system

Usage: reboot [OPTION]
eg. reboot

- **poweroff**– power off the system

Usage: poweroff [OPTION]
eg. Poweroff

EXP-2 MENU BASED MATH CALCULATOR

AIM:-

To write a program for menu based math calculator using shell scripting commands.

ALGORITHM:-

1) Read the operator

2) Read the operands

3) Using the operator as choice for switch case write case for operators

4) End switch case

5) Stop

PROGRAM:-

```
echo "Menu Based Calculator"
echo "Enter the Operands"
read a
read b
echo "Enter the Operator"
read o
case $o in
"+" ) echo "$a + $b" = `expr $a + $b`;;
"-" ) echo "$a + $b" =  `expr $a - $b`;;
"*" ) echo "$a + $b" =   `expr $a * $b`;;
"/" ) echo "$a + $b" =  `expr $a / $b`;;
* ) echo "Inavlid Operation"
esac
```

OUTPUT:-

Menu Based Calculator
Enter the Operands
4
6
Enter the Operator
+
4+6=10

EXP-3 PRINTING PATTERN USING LOOP STATEMENT

AIM:-

To print a pattern using loop statement by using shell scripting commands.

ALGORITHM:-

1) Read the number given.

2) Initialize the for loop where i<=$n.

3) Initiallize one more loop inside the above loop with j<=$i.

4) Print “*” and close the two loops.

5) Continue until the required root loops(rows) reached.

PROGRAM:-

```
echo "Enter the limit"
read n
echo "Pattern"
for ((i=1;i<=$n;i++))
do
for ((j=1;j<=i;j++))
do
echo -n " $ "
done
echo " "
done
```

OUTPUT:-

```
Enter the limit
3
Pattern
$
$ $
$ $ $
```

EXP-4 SEARCHING A SUBSTRING IN A GIVEN TEXT

AIM:-

To write a program for searching a substring in a given text by using shell programming.

ALGORITHM:-

1) To select a substring from string using ${string: starting position :root position}

2) Comparing two strings is done by {$s1=$s2}

3) To check for zero length string use [-z String]

4) To check for empty string use [String]

5) To check for non zero length string use –n as [-n $string]

6) The length of string is obtained by ${#string}

PROGRAM:-

```
read str
read substr

prefix=${str%%$substr*}
index=${#prefix}

if [[ index -eq ${#str} ]];
then
   echo "Substring is not present in string."
else
   echo "Index of substring in string : $index"
fi
```

OUTPUT:-

```
string
r
Index of substring in string : 2
```

EXP -5 CONVERTING FILES NAMES FROM UPPERCASE TO LOWERCASE

AIM:-

To write a program for converting files from uppercase case to lowercase.

ALGORITHM:-

1) Get the file name.
2) store the name in a variable.
3) Apply conversion to that variable.
4) Store it in other variable.
5) Finally display the converted file name.

PROGRAM:-

```
echo -n "Enter the filename:"
read filename
if [ ! -f $filename ]; then
   echo "filename $filename does not exists"
   exit 1
else
   tr '[A-Z]' '[a-z]' < $filename
fi
```

OUTPUT:-

```
Enter the filename:test
hello
world
```

EXP-6 SHOWING VARIOUS SYSTEM INFORMATION

AIM:-

To write a program in vi editor to show various information using shell command

ALGORITHM:-

1. Get the system information such as network name and node name, kernel name, kernel version e.t.c .

2 Network $node name=$(uname –n).

Kernel name=$(uname –s)

Kernel ru=$(uname –a).

Operating system =$(uname –m).

All information $(uname –A).

PROGRAM:

```
echo "SYSTEM INFORMATION"
echo "Hello ,$LOGNAME"
echo "Current Date is = $(date)"
echo "User is 'who I am'"
echo "Current Directory = $(pwd)"
echo "Network Name and Node Name = $(uname -n)"
echo "Kernal Name =$(uname -s)"
echo "Kernal Version=$(uname -v)"
echo "Kernal Release =$(uname -r)"
echo "Kernal OS =$(uname -o)"
echo "Proessor Type = $(uname -p)"
echo "Kernel Machine Information = $(uname –m)"
echo "All Information =$(uname -a)"
```

OUTPUT:

```
SYSTEM INFORMATION
Hello ,
Current Date is = Thu Sep 22 10:03:03 UTC 2022
User is 'who I am'
Current Directory = /root
Network Name and Node Name = localhost
Kernal Name =Linux
Kernal Version=#21 Fri Aug 4 21:02:28 CEST 2017
Kernal Release =4.12.0-rc6-g48ec1f0-dirty
Kernal OS =Linux
Proessor Type = unknown
Kernel Machine Information = i586
All Information =Linux localhost 4.12.0-rc6-g48ec1f0-dirty #21 Fri
Aug 4 21:02:2 CEST 2017 i586 Linux
```

EXP-7 IMPLEMENTATION OF PROCESS SCHEDULING MECHANISM – FCFS, SJF,PRIORITY QUEUE.

AIM:

Write a C program to implement the various process scheduling mechanisms such as FCFS, SJF, Priority .

7.(A) FCFS SCHEDULING:

ALGORITHM FOR FCFS SCHEDULING:

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue

Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time

Step 4: Set the waiting of the first process as '0' and its burst time as its turn around time

Step 5: for each process in the Ready Q calculate

(a) Waiting time for process(n)= waiting time of process (n-1) + Burst time of process(n-1)

(b) Turn around time for Process(n)= waiting time of Process(n)+ Burst time for process(n)

Step 6: Calculate

(a) Average waiting time = Total waiting Time / Number of process

(b) Average Turnaround

PROGRAM:

```
#include<stdio.h>

 int main()

{
   int n,bt[20],wt[20],tat[20],avwt=0,avtat=0,i,j;
   printf("Enter total number of processes(maximum 20):");
   scanf("%d",&n);

   printf("\nEnter Process Burst Time\n");
   for(i=0;i<n;i++)
   {
      printf("P[%d]:",i+1);
```

```
        scanf("%d",&bt[i]);
    }

    wt[0]=0;

    for(i=1;i<n;i++)
    {
        wt[i]=0;
        for(j=0;j<i;j++)
            wt[i]+=bt[j];
    }

    printf("\nProcess\t\tBurst Time\tWaiting Time\tTurnaround
Time");

    for(i=0;i<n;i++)
    {
        tat[i]=bt[i]+wt[i];
        avwt+=wt[i];
        avtat+=tat[i];
        printf("\nP[%d]\t\t%d\t\t%d\t\t%d",i+1,bt[i],wt[i],tat[i]);
    }

    avwt/=i;
    avtat/=i;
    printf("\n\nAverage Waiting Time:%d",avwt);
    printf("\nAverage Turnaround Time:%d",avtat);

    return 0;
}
```

OUTPUT:

```
Enter total number of processes(maximum 20):3

Enter Process Burst Time
P[1]:33
P[2]:2
P[3]:1

Process         Burst Time      Waiting Time    Turnaround Time
P[1]        33          0           33
P[2]        2           33          35
P[3]        1           35          36
```

Average Waiting Time:22
Average Turnaround Time:34

7. (B) SJF

ALGORITHM FOR SJF:

Step 1: Start the process
Step 2: Accept the number of processes in the ready Queue
Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time
Step 4: Start the Ready Q according the shortest Burst time by sorting according to lowest to highest burst time.
Step 5: Set the waiting time of the first process as '0' and its turnaround time as its burst time.
Step 6: For each process in the ready queue, calculate
(a) Waiting time for process(n)= waiting time of process (n-1) + Burst time of process(n-1)
(b) Turnaround time for Process(n)= waiting time of Process(n)+ Burst time for process(n)

Step 7: Calculate
(a) Average waiting time = Total waiting Time / Number of process
(b) Average Turnaround time = Total Turnaround Time / Number of process

Step 8: Stop the process

PROGRAM:

```
#include <stdio.h>

int main()
{
    int arrival_time[10], burst_time[10], temp[10];
    int i, smallest, count = 0, time, limit;
    double wait_time = 0, turnaround_time = 0, end;
    float average_waiting_time, average_turnaround_time;
    printf("\nEnter the Total Number of Processes:\t");
    scanf("%d", &limit);
    printf("\nEnter Details of %d Processes\n", limit);
    for(i = 0; i < limit; i++)
```

```
    {
        printf("\nEnter Arrival Time:\t");
        scanf("%d", &arrival_time[i]);
        printf("Enter Burst Time:\t");
        scanf("%d", &burst_time[i]);
        temp[i] = burst_time[i];
    }
    burst_time[9] = 9999;
    for(time = 0; count != limit; time++)
    {
        smallest = 9;
        for(i = 0; i < limit; i++)
        {
            if(arrival_time[i] <= time && burst_time[i] < burst_time[smallest] &&
burst_time[i] > 0)
            {
                smallest = i;
            }
        }
        burst_time[smallest]--;
        if(burst_time[smallest] == 0)
        {
            count++;
            end = time + 1;
            wait_time = wait_time + end - arrival_time[smallest] -
temp[smallest];
            turnaround_time = turnaround_time + end - arrival_time[smallest];
        }
    }
    average_waiting_time = wait_time / limit;
    average_turnaround_time = turnaround_time / limit;
    printf("\n\nAverage Waiting Time:\t%lf\n", average_waiting_time);
    printf("Average Turnaround Time:\t%lf\n", average_turnaround_time);
    return 0;
}
```

OUTPUT:

```
Enter the Total Number of Processes: 4
```

Enter Details of 4 Processes

Enter Arrival Time: 1
Enter Burst Time: 4

Enter Arrival Time: 2
Enter Burst Time: 4

Enter Arrival Time: 3
Enter Burst Time: 5

Enter Arrival Time: 4
Enter Burst Time: 8

Average Waiting Time: 4.750000
Average Turnaround Time: 10.000000

7. (C).PRIORITY SCHEDULING.

ALGORITHM FOR PRIORITY SCHEDULING.

Step 1: Start the process
Step 2: Accept the number of processes in the ready Queue
Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time
Step 4: Sort the ready queue according to the priority number.
Step 5: Set the waiting of the first process as '0' and its burst time as its turn around time
Step 6: For each process in the Ready Q calculate
(a) Waiting time for process(n)= waiting time of process (n-1) + Burst time of process(n-1)

(b) Turn around time for Process(n)= waiting time of Process(n)+ Burst time for process(n)

Step 7: Calculate
(a) Average waiting time = Total waiting Time / Number of process

(b) Average Turnaround time = Total Turnaround Time / Number of process

Step 8: Stop the process

PROGRAM:

```
#include<stdio.h>

int main()
{
   int
bt[20],p[20],wt[20],tat[20],pr[20],i,j,n,total=0,pos,temp,avg_wt,avg_t
at;
   printf("Enter Total Number of Process:");
   scanf("%d",&n);

   printf("\nEnter Burst Time and Priority\n");
   for(i=0;i<n;i++)
   {
      printf("\nP[%d]\n",i+1);
      printf("Burst Time:");
      scanf("%d",&bt[i]);
      printf("Priority:");
      scanf("%d",&pr[i]);
      p[i]=i+1;          //contains process number
   }

   //sorting burst time, priority and process number in ascending
order using selection sort
   for(i=0;i<n;i++)
   {
      pos=i;
      for(j=i+1;j<n;j++)
      {
         if(pr[j]<pr[pos])
            pos=j;
      }

      temp=pr[i];
      pr[i]=pr[pos];
      pr[pos]=temp;

      temp=bt[i];
      bt[i]=bt[pos];
      bt[pos]=temp;

      temp=p[i];
```

```
        p[i]=p[pos];
        p[pos]=temp;
    }

    wt[0]=0; //waiting time for first process is zero

    //calculate waiting time
    for(i=1;i<n;i++)
    {
        wt[i]=0;
        for(j=0;j<i;j++)
            wt[i]+=bt[j];

        total+=wt[i];
    }

    avg_wt=total/n;      //average waiting time
    total=0;

    printf("\nProcess\t    Burst Time    \tWaiting Time\tTurnaround
Time");
    for(i=0;i<n;i++)
    {
        tat[i]=bt[i]+wt[i];     //calculate turnaround time
        total+=tat[i];
        printf("\nP[%d]\t\t  %d\t\t    %d\t\t\t%d",p[i],bt[i],wt[i],tat[i]);
    }

    avg_tat=total/n;     //average turnaround time
    printf("\n\nAverage Waiting Time=%d",avg_wt);
    printf("\nAverage Turnaround Time=%d\n",avg_tat);

return 0;
}
```

OUTPUT:

```
Enter Total Number of Process:4

Enter Burst Time and Priority

P[1]
Burst Time:6
```

Priority:3

P[2]
Burst Time:2
Priority:2

P[3]
Burst Time:14
Priority:1

P[4]
Burst Time:4
Priority:6

Process	Burst Time	Waiting Time	Turnaround Time
P[3]	14	0	14
P[2]	2	14	16
P[1]	6	16	22
P[4]	4	22	26

Average Waiting Time=13
Average Turnaround Time=19

EXP-8 READER-WRITERS PROBLEM

AIM:
To write a program to implement readers and writers problem.

ALGORITHM:
Start ;
/* Initialize semaphore variables*/
integer mutex=1; // Controls access to RC
integer DB=1; // controls access to data base
integer RC=0; // Number of process reading the database currently
1.Reader( ) // The algorithm for readers process
Repeat continuously
DOWN(mutex); // Lock the counter RC
RC=RC+1; // one more reader
If(RC=1)DOWN(DB); // This is the first reader.Lock the database for reading
UP(mutex); // Release exclusive access to RC
Read database(); // Read the database

DOWN(mutex); // Lock the counter RC
RC=RC-1; // Reader count less by one now
If(RC=0)UP(DB); // This is the last reader .Unlock the database.
UP(mutex); // Release exclusive access to RC
End
2.Writer( ) // The algorithm for Writers process
Reepeat continuously
DOWN(DB); // Lock the database
Write_Database(); // Read the database
UP(DB); // Release exclusive access to the database
End
Step a: initialize two semaphore mutex=1 and db=1 and rc,(Mutex controls the access to read count rc)
Step b: create two threads one as Reader() another as Writer()
Reader Process:
Step 1: Get exclusive access to rc(lock Mutex)
Step 2: Increment rc by 1
Step 3: Get the exclusive access bd(lock bd)
Step 4: Release exclusive access to rc(unlock Mutex)
Step 5: Release exclusive access to rc(unlock Mutex)
Step 6: Read the data from database **Step 7:** Get the exclusive access to rc(lock mutex)
Step 8: Decrement rc by 1, if rc =0 this is the last reader.
Step 9: Release exclusive access to database(unlock mutex)
Step 10: Release exclusive access to rc(unlock mutex)

Program:

```
#include<stdio.h>

int x=1,rc=0,readcount=1;

void p(int*a)

{

  while(*a==0)

  {

    printf("busy wait");

  }

  *a=*a-1;
```

```
}
void v(int*b)
{
   *b=*b+1;
}
void p1(int*c)
{
   while(*c==0)
   {
      printf("busy wait");
   }
   *c=*c-1;
}
void v1(int*d)
{
   *d=*d+1;
}
void reader()
{
   int flag=1;
   while(flag==1)
   {
      p(&readcount);
      rc=rc+1;
      if(rc==1)
      {
         p1(&x);
         v(&readcount);
         printf("\n reader is reading");
```

```
        }
        p(&readcount);
        rc=rc-1;
        if(rc==0)
        {
           v1(&x);
           v(&readcount);
        }
        flag=0;
    }
}
void writer()
{
    p1(&x);
    printf("\n writer is writing");
    v1(&x);
}
void main()
{
    reader();
    writer();
    reader();
    reader();
    writer();
}
```

Output:

reader is reading
writer is writing
reader is reading
reader is reading

writing is writing

EXP-9 DINING PHILOSOPHERS PROBLEM

AIM:
Write a program to solve the Dining Philosophers problem.

ALGORITHM:
1. Initialize the state array S as 0, Si =0 if the philosopher i is thinking or 1 if hungry.
2. Associate two functions getfork(i) and putfork(i) for each philosopher i.
3. For each philosopher I call getfork(i) , test(i) and putfork(i) if i is 0
4. Stop
Algorithm for getfork(i):
Step 1: set S[i]= 1 i.e. the philosopher i is hungry
Step 2: call test(i)
Algorithm for putfork(i)
Step 1: set S[i]=0 I.e. the philosopher i is thinking
Step 2: test(LEFT) and test(RIGHT)
Algorithm for test(i)
Step 1: check if (state[i]==HUNGRY && state[LEFT]!=EATING && state[RIGHT]!=EATING)
Step 2: give the i philosopher a chance to eat.

PROGRAM:

```
#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

#define LEFT (i+4)%5

#define RIGHT (i+1)%5

#define THINKING 0

#define HUNGRY 1

#define EATING 2

int state[5];
```

```
void put_forks(int);
void test(int);
void take_forks(int);
void philosopher(int i)
{
   if(state[i]==0)
   {
      take_forks(i);
      if(state[i]==EATING)
      {
         printf("\n Eating in progress..");
         put_forks(i);
      }
   }
}
void take_forks(int i)
{
   state[i]=HUNGRY;
   test(i);
}
void put_forks(int i)
{
   state[i]=THINKING;
   printf("\nphilosopher %d has completed its work",i);
   test(LEFT);
   test(RIGHT);
```

```
}
void test(int i)
{
   if(state[i]==HUNGRY&&state[LEFT]!=EATING&&state[RIGHT]!=EATING)
   {
      printf("\nphilosopher %d can eat",i);
      state[i]=EATING;
   }
}
void main()
{
   int i;
   for(i=1;i<=5;i++)
   state[i]=0;
   printf("\n\n\t\t\t DINNING PHILOSOPHERS PROBLEM");
   printf("\n\t\t\t~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
   printf("\n ALL THE PHILOSOPHERS ARE THINKING !!....\n",i);
   for(i=1;i<=5;i++)
   {
      printf("\n\n the philosopher %d falls hungry\n",i);
      philosopher(i);
      getch();
   }
}
```

OUTPUT:

DINNING PHILOSOPHERS PROBLEM

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

ALL THE PHILOSOPHERS ARE THINKING !!....

the philosopher 1 falls hungry

philosopher 1 can eat

Eating in progress..

philosopher 1 has completed its work

the philosopher 2 falls hungry

philosopher 2 can eat

Eating in progress..

philosopher 2 has completed its work

the philosopher 3 falls hungry

philosopher 3 can eat

Eating in progress..

philosopher 3 has completed its work

the philosopher 4 falls hungry

philosopher 4 can eat

Eating in progress..
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

philosopher 4 has completed its work

the philosopher 5 falls hungry

philosopher 5 can eat

Eating in progress..

philosopher 5 has completed its work

EXP-10 FIRST FIT , WORST FIT, BEST FIT ALLOCATION STRATEGY

AIM:
To implement
a) First fit
b) Best fit
c) Worst fit &
d) To make comparative study

ALGORITHM:
Step 1: Start the program.
Step 2: Get the number of memory partition and their sizes.
Step 3: Get the number of processes and values of block size for each process.
Step 4: First fit algorithm searches all the entire memory block until a hole which is big enough is encountered. It allocates that memory block for the requesting process.
Step 5: Best-fit algorithm searches the memory blocks for the smallest hole which can be allocated to requesting process and allocates if.
Step 6: Worst fit algorithm searches the memory blocks for the largest hole and allocates it to the process.
Step 7: Analyses all the three memory management techniques and display the best algorithm which utilizes the memory resources effectively and efficiently.
Step 8: Stop the program.

PROGRAM:

```
// C implementation of First - Fit algorithm
#include<stdio.h>

// Function to allocate memory to
// blocks as per First fit algorithm
void firstFit(int blockSize[], int m, int processSize[], int n)
{
    int i, j;
    // Stores block id of the
    // block allocated to a process
    int allocation[n];

    // Initially no block is assigned to any process
    for(i = 0; i < n; i++)
    {
        allocation[i] = -1;
    }

    // pick each process and find suitable blocks
    // according to its size ad assign to it
    for (i = 0; i < n; i++)       //here, n -> number of processes
    {
        for (j = 0; j < m; j++)       //here, m -> number of blocks
        {
            if (blockSize[j] >= processSize[i])
            {
```

```
            // allocating block j to the ith process
            allocation[i] = j;

            // Reduce available memory in this block.
            blockSize[j] -= processSize[i];

            break;    //go to the next process in the queue
          }
        }
    }

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

// Driver code
int main()
```

```
{
    int pno; //number of blocks in the memory
    int bno; //number of processes in the input queue
    int x;
    int y;

    int blockSize[100]; //blocks
    int processSize[100];//processes

    printf("Enter no. of blocks: ");
        scanf("%d", &bno);
        printf("\nEnter size of each block: ");
        for(x = 0; x < bno; x++)
                scanf("%d", &blockSize[x]);

        printf("\nEnter no. of processes: ");
        scanf("%d", &pno);
        printf("\nEnter size of each process: ");
        for(y = 0; y < pno; y++)
                scanf("%d", &processSize[y]);

    int m = bno;
    int n = pno;

    firstFit(blockSize, m, processSize, n);
```

```
    return 0 ;
}
```

OUTPUT:

Enter no. of blocks: 5

Enter size of each block: 100

500

200

300

600

Enter no. of processes: 4

Enter size of each process: 212

417

112

426

Process No.	**Process Size**	**Block no.**
1	**212**	**2**
2	**417**	**5**
3	**112**	**2**
4	**426**	**Not Allocated**

PROGRAM:

```
// C implementation of Best - Fit algorithm
```

```
#include<stdio.h>

void main()
{
int fragment[20],b[20],p[20],i,j,nb,np,temp,lowest=9999;
static int barray[20],parray[20];
printf("\n\t\t\tMemory Management Scheme - Best Fit");
printf("\nEnter the number of blocks:");
scanf("%d",&nb);
printf("Enter the number of processes:");
scanf("%d",&np);
printf("\nEnter the size of the blocks:-\n");
for(i=1;i<=nb;i++)
   {
printf("Block no.%d:",i);
      scanf("%d",&b[i]);
   }
printf("\nEnter the size of the processes :-\n");
for(i=1;i<=np;i++)
   {
      printf("Process no.%d:",i);
      scanf("%d",&p[i]);
   }
for(i=1;i<=np;i++)
{
```

```
for(j=1;j<=nb;j++)
{
if(barray[j]!=1)
{
temp=b[j]-p[i];
if(temp>=0)
if(lowest>temp)
{
parray[i]=j;
lowest=temp;
}
}
}
fragment[i]=lowest;
barray[parray[i]]=1;
lowest=10000;
}
printf("\nProcess_no\tProcess_size\tBlock_no");
for(i=1;i<=np && parray[i]!=0;i++)
printf("\n%d\t\t%d\t\t%d",i,p[i],parray[i]);
}
```

OUTPUT:

Memory Management Scheme - Best Fit

Enter the number of blocks:5

Enter the number of processes:4

Enter the size of the blocks:-

Block no.1:100

5Block no.2:500

Block no.3:200

Block no.4:300

Block no.5:400

Enter the size of the processes :-

Process no.1:212

Process no.2:417

Process no.3:112

Process no.4:426

Process_no	Process_size	Block_no
1	212	4
2	417	2
3	112	3

PROGRAM:

```
// C++ implementation of worst - Fit algorithm
#include<stdio.h>

// Function to allocate memory to blocks as per worst fit
// algorithm
void worstFit(int blockSize[], int m, int processSize, int n)
```

```
{
        // Stores block id of the block allocated to a
        // process
        int allocation[n];

        // Initially no block is assigned to any process
        for(int i = 0; i < n; i++)
   {
      allocation[i] = -1;
   }

        // pick each process and find suitable blocks
        // according to its size ad assign to it
        for (int i=0; i<n; i++)
        {
                // Find the best fit block for current process
                int wstIdx = -1;
                for (int j=0; j<m; j++)
                {
                        if (blockSize[j] >= processSize[i])
                        {
                                if (wstIdx == -1)
                                        wstIdx = j;
                                else if (blockSize[wstIdx] < blockSize[j])
                                        wstIdx = j;
                        }
```

```
            }

            // If we could find a block for current process
            if (wstIdx != -1)
            {
                // allocate block j to p[i] process
                allocation[i] = wstIdx;

                // Reduce available memory in this block.
                blockSize[wstIdx] -= processSize[i];
            }
        }

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

```
// Driver code
int main()
{
        int pno; //number of blocks in the memory
   int bno; //number of processes in the input queue
   int x;
   int y;

   int blockSize[100]; //blocks
   int processSize[100];//processes

   printf("Enter no. of blocks: ");
        scanf("%d", &bno);
        printf("\nEnter size of each block: ");
        for(x = 0; x < bno; x++)
                scanf("%d", &blockSize[x]);

        printf("\nEnter no. of processes: ");
        scanf("%d", &pno);
        printf("\nEnter size of each process: ");
        for(y = 0; y < pno; y++)
                scanf("%d", &processSize[y]);

   int m = bno;
   int n = pno;
```

```
    worstFit(blockSize, m, processSize, n);

    return 0 ;
}
```

OUTPUT:

Enter no. of blocks: 5

Enter size of each block: 100

500

200

300

600

Enter no. of processes: 4

Enter size of each process: 212

417

112

426

Process No.	Process Size	Block no.
1	212	5
2	417	2
3	112	5
4	426	Not Allocated

EXP-11 BANKERS ALGORITHM

AIM:
Write a program to implement Banker's Algorithm

ALGORITHM:
This algorithm was suggested by Dijkstar, the name banker is used here to indicate that it uses a banker's activity for providing loans and receiving payment against the given loan. This algorithm places very few restrictions on the processes competing for resources. Every request for the resource made by a process is thoroughly analyzed to check, whether it may lead to a deadlock situation. If the result is yes then the process is blocked on this request. At some future time, its request is considered once again for resource allocation. So this indicated that, the processes are free to request for the allocation, as well as de-allocation of resources without any constraints. So this generally reduces the idling of resources.
Suppose there are (P) number of Processes and (r) number of resources then its time complexity is proportional to P x r2
At any given stage the OS imposes certain constraints on any process trying to use the resource. At a given moment during the operation of the system, processes P, would have been allocated some resources. Let these allocations total up to S.
Let (K=r-1) be the number of remaining resources available with the system.
Then k>=0 is true, when allocation is considered.
Let maxk be the maximum resource requirement of a given process Pi.
Actk be the actual resource allocation to Pi at any given moment.
Then we have the following condition.
Maxk<=p for all k and to

PROGRAM:

```
#include <stdio.h>

int main()

{

 int n, m, i, j, k, y,alloc[20][20],max[20][20],avail[50],ind=0;

 printf("Enter the no of Proceses:");

 scanf("%d",&n);

 printf("Enter the no of Resources:");
```

```
 scanf("%d",&m);
 printf("Enter the Allocation Matrix:");
 for (i = 0; i < n; i++) {
 for (j = 0; j < m; j++)
 scanf("%d",&alloc[i][j]);
 }
 printf("Enter the Max Matrix:");
 for (i = 0; i < n; i++) {
 for (j = 0; j < m; j++)
 scanf("%d",&max[i][j]);
 }
printf("Enter the Available Matrix");
 for(i=0;i<m;i++)
 scanf("%d",&avail[i]);
 int finish[n], safesequence[n],work[m],need[n][m];
//calculating NEED matrix
 for (i = 0; i < n; i++) {
 for (j = 0; j < m; j++)
 need[i][j] = max[i][j] - alloc[i][j];
 }
 printf("NEED matrix is");
 for (i = 0; i < n; i++)
 {
 printf("\n");
 for (j = 0; j < m; j++)
 printf(" %d ",need[i][j]);
```

```
}
for(i=0;i<m;i++)
{
work[i]=avail[i];
}
for (i = 0; i < n; i++) {
finish[i] = 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] > work[j])
{
flag = 1;
break;
}
}
if (flag == 0) {
safesequence[ind++] = i;
for (y = 0; y < m; y++)
work[y] += alloc[i][y];
```

```
 finish[i] = 1;
 }
 }
 }
 }
 printf("\nFollowing is the SAFE Sequence\n");
 for (i = 0; i <= n - 1; i++)
 printf(" P%d ", safesequence[i]);
}
```

OUTPUT:

Enter the no of processess: 5

Enter the no of resources: 4

Enter the allocation matrix:

0 0 1 2

1 0 0 0

1 3 5 4

0 6 3 2

0 0 1 4

Enter the max matrix:

0 0 1 2

1 7 5 0

2 3 5 6

0 6 5 2

0 6 5 6

Enter the available matrix:

1 5 2 0

NEED Matrix is

0 0 0 0

0 7 5 0

1 0 0 2

0 0 2 0

0 6 4 2

Following is the SAFE Sequence

P0 P2 P3 P4 P1

EXP-12 IMPLEMENT THE PRODUCER CONSUMER PROBLEM USING SEMAPHORE

AIM:
To write a program to implement producer consumer problem using semaphore.

ALGORITHM:
Step 1: Start.
Step 2: Let n be the size of the buffer.
Step 3: check if there are any producer.
Step 4: if yes check whether the buffer is full.
Step 5: If no the producer item is stored in the buffer.
Step 6: If the buffer is full the producer has to wait.
Step 7: Check there is any consumer. If yes check whether the buffer is empty
Step 8: If no the consumer consumes them from the buffer.
Step 9: If the buffer is empty, the consumer has to wait.
Step 10: Repeat checking for the producer and consumer till required.
Step 11: Terminate the process.

PROGRAM:

#include<stdio.h>

```
#include<conio.h>
#include<stdlib.h>
int mutex=1,full=0,empty=3,x=0;
void main()
{
int n;
void producer();
void consumer();
int wait(int);
int signal(int);
printf("\n 1.producer \n 2.consumer \n 3.exit");
while(1)
{
printf("\n enter your choice:");
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("\n producer 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 item 1

Enter your choice : 1

Producer produces the item 2

Enter your choice : 1

Producer produces the item 3

Enter your choice : 1

Buffer is full

Enter your choice : 2

Consumer consumes item 3

Enter your choice : 2

Consumer consumes item 2

Enter your choice : 2

Consumer consumes item 1

Enter your choice : 2

Buffer is empty

Enter your choice : 3

EXP-13 TO IMPLEMENT THE MEMORY MANAGEMENT POLICY-PAGING

AIM:
To implement the memory management policy-paging

ALGORITHM:
Step 1: Read all the necessary input from the keyboard.
Step 2: Pages - Logical memory is broken into fixed - sized blocks.
Step 3: Frames – Physical memory is broken into fixed – sized blocks.
Step 4: Calculate the physical address using the following
Physical address = (Frame number * Frame size) + offset
Step 5: Display the physical address.
Step 6: Stop the process.

PROGRAM:

```
#include<stdio.h>

#include<conio.h>

#include<math.h>

void main()
```

```
{
int size,m,n,pgno,pagetable[3]={5,6,7},i,j,frameno;
double m1;
int ra=0,ofs;
printf("Enter process size (in KB of max 12KB):");
/*reading memeory size*/
scanf("%d",&size);
m1=size/4;
n=ceil(m1);
printf("Total No. of pages: %d",n);
printf("\nEnter relative address (in hexadecimal notation eg.0XRA) \n");
//printf("The length of relative Address is : 16 bits \n\n The size of offset is :12 bits\n");
scanf("%d",&ra);
pgno=ra/1000;
/*calculating physical address*/
ofs=ra%1000;
printf("page no=%d\n",pgno);
printf("page table");
for(i=0;i<n;i++)
printf("\n %d [%d]",i,pagetable[i]);
frameno=pagetable[pgno];
printf("\n Equivalent physical address : %d%d",frameno,ofs);
getch();
}
```

Output:

Enter process size (in KB of max 12KB):12

Total No. of pages: 3

Enter relative address (in hexadecimal notation eg.0XRA)

2643

page no=2

page table

0 [5]

1 [6]

2 [7]

Equivalent physical address : 7643