PALLAVI ENGINEERING COLLEGE

Accredited with 'A' grade by NAAC, Approved by AICTE, New Delhi & Affiliated to JNTU-Hyderabad
Nagole, Hyderabad, R.R Dist 501505, Telangana State, India
www.pallavienginerringcollege.ac.in

DEPARTMENT OF COMPUTER SCIENCE-CYBER SECURITY
OPERATING SYSTEMS LAB MANNUAL
B.TECH YEAR/SEM: II-II

Prepared by
TSKS JYOTHIRMAYI
HOD-Cyber Security

SYLLABUS

CS406PC: OPERATING SYSTEMS LAB (Using UNIX/LINUX)

B.TECH- II Year- II Sem L T P C

0 0 3 1.5

Prerequisites:

- A course on "Programming for Problem Solving".
- A course on "Computer Organization and Architecture".

Co-requisite:

- A course on "Operating Systems".

Course Objectives:

- To provide an understanding of the design aspects of operating system concepts through simulation
- Introduce basic Unix commands, system call interface for process management, interprocess communication and I/O in Unix

Course Outcomes:

- Simulate and implement operating system concepts such as scheduling, deadlock management, file management and memory management.
- Able to implement C programs using Unix system calls

LIST OF EXPERIMENTS:

1. Write C programs to simulate the following CPU Scheduling algorithms
   a) FCFS b) SJF c) Round Robin d) priority
2. Write programs using the I/O system calls of UNIX/LINUX operating system (open, read, write, close, fcntl, seek, stat, opendir, readdir)
3. Write a C program to simulate Bankers Algorithm for Deadlock Avoidance and Prevention.

4. Write a C program to implement the Producer – Consumer problem using semaphores usingUNIX/LINUX system calls.
5. Write C programs to illustrate the following IPC mechanisms
   a) Pipes b) FIFOs c) Message Queues d) Shared Memory
6. Write C programs to simulate the following memory managementtechniques
   a) Paging b) Segmentation

TEXT BOOKS:

1. Operating System Principles- Abraham Silberchatz, Peter B. Galvin, Greg Gagne 7th Edition,John Wiley
2. Advanced programming in the Unix environment, W.R.Stevens, *Pearson* education

WEEK-1

Write C programs to simulate the following CPU Scheduling algorithms.

a) FCFS
b) SJF
c) Round Robin
d) Priority

a)FCFS (First Come First Serve)

Aim: Write a C program to implement the various process scheduling mechanisms such as FCFS scheduling.

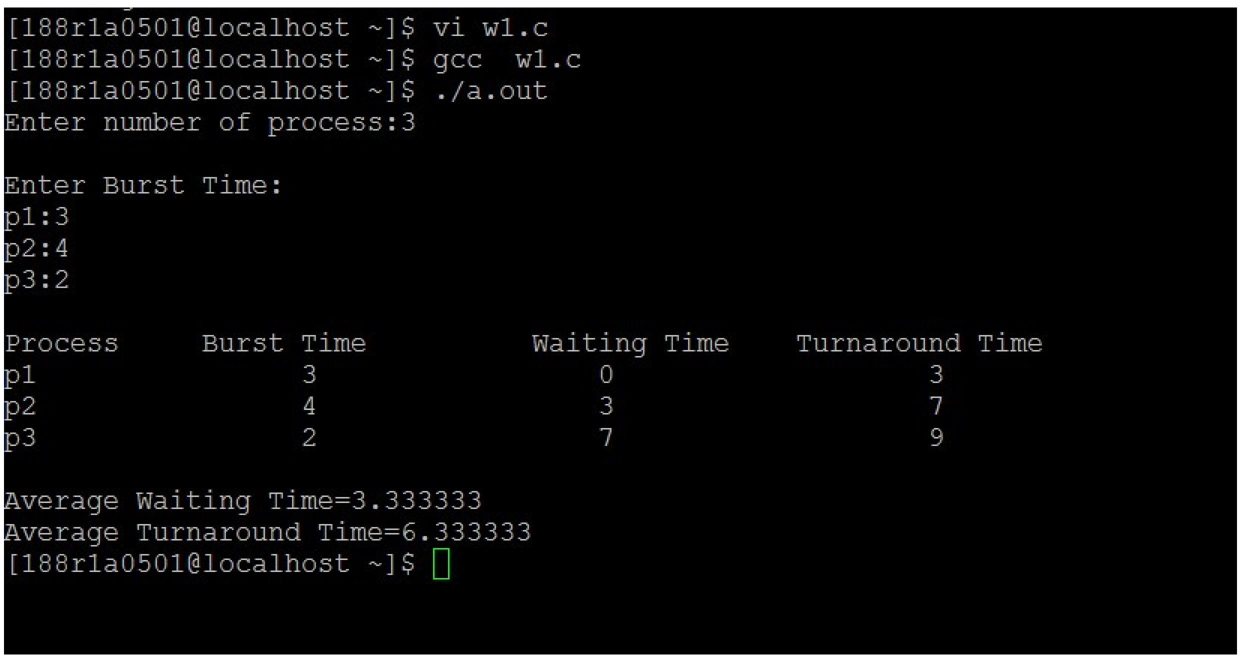
Algorithm:

```
1: Start the process
2: Accept the number of processes in the ready Queue
3: For each process in the ready Q, assign the process id and accept the CPU burst time
4: Set the waiting of the first process as '0' and its burst time as its turn around time
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.  Turnaround time for Process(n)= waiting time of Process(n)+ Burst time for process(n)
6: Calculate
a.  Average waiting time = Total waiting Time / Number of process
b.  Average Turnaround time = Total Turnaround Time / Number of process
7: Stop the process
```

Program:

```
#include<stdio.h>
int  main()
{
   int bt[20],p[20],wt[20],tat[20],i,j,n,total=0,pos,temp;
   float avg_wt,avg_tat;
   printf("Enter number of process:");
   scanf("%d",&n);
   printf("\nEnter Burst Time:\n");
   for(i=0;i<n;i++)
   {
      printf("p % d:",i+1);
      scanf("%d",&bt[i]);
      p[i]=i+1;           //contains process number
   }
wt[0]=0;        //waiting time for first process will be 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=(float)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=(float)total/n;     //average turnaround time
   printf("\n\nAverage Waiting Time=%f",avg_wt);
   printf("\nAverage Turnaround Time=%f\n",avg_tat);
}
```

Output:



```
[188r1a0501@localhost ~]$ vi w1.c
[188r1a0501@localhost ~]$ gcc  w1.c
[188r1a0501@localhost ~]$ ./a.out
Enter number of process:3

Enter Burst Time:
p1:3
p2:4
p3:2

Process      Burst Time          Waiting Time    Turnaround Time
p1                 3                  0                   3
p2                 4                  3                   7
p3                 2                  7                   9

Average Waiting Time=3.333333
Average Turnaround Time=6.333333
[188r1a0501@localhost ~]$ 
```

b) SJF (Shortest Job First)

Aim: Write a C program to implement the various process scheduling mechanisms such as SJF Scheduling.

Algorithm:

1: Start the process

2: Accept the number of processes in the ready Queue

3: For each process in the ready Q, assign the process id and accept the CPU burst time 4: Start the Ready Q according the shortest Burst time by sorting according to lowest to highest burst time.

5: Set the waiting time of the first process as '0' and its turnaround time as its burst time.

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) Turn around time for Process(n)= waiting time of Process(n)+ Burst time for process(n)
7: Calculate

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

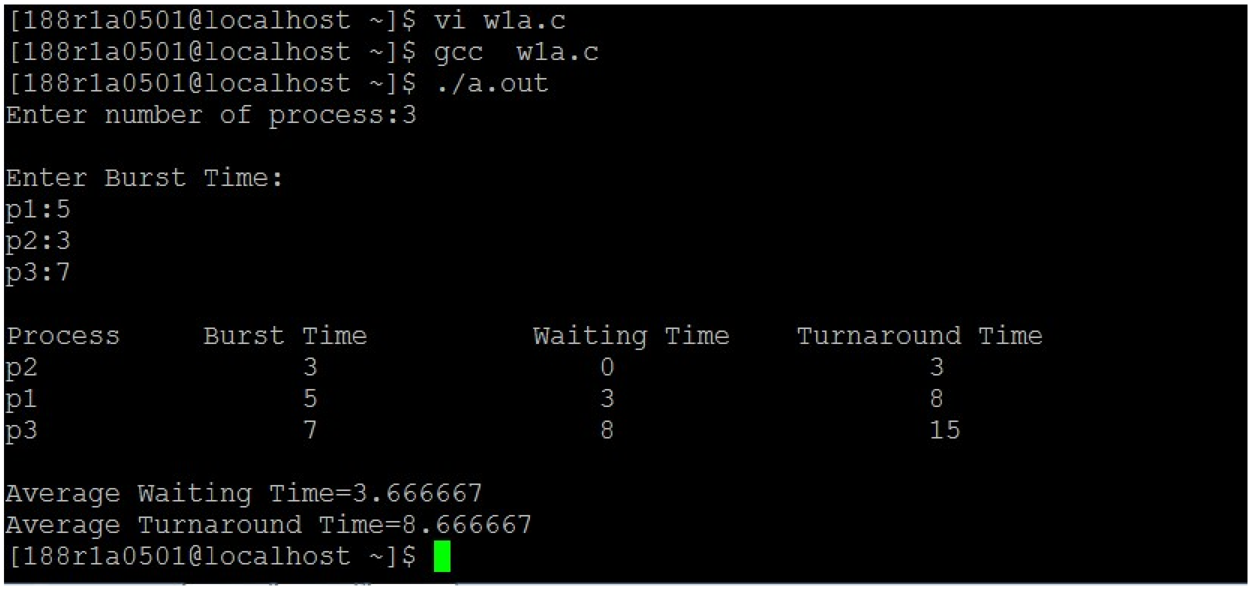
• Average Turnaround time = Total Turnaround Time / Number of process
8: Stop the process

Program:

```
#include<stdio.h>
int  main()
{
   int bt[20],p[20],wt[20],tat[20],i,j,n,total=0,pos,temp;
   float avg_wt,avg_tat;
   printf("Enter number of process:");
   scanf("%d",&n);

   printf("\nEnter Burst Time:\n");
   for(i=0;i<n;i++)
   {
      printf("p%d:",i+1);
      scanf("%d",&bt[i]);
      p[i]=i+1;           //contains process number
   }
//sorting burst time in ascending order using selection sort
   for(i=0;i<n;i++)
   {
      pos=i;
      for(j=i+1;j<n;j++)
      {
         if(bt[j]<bt[pos])
            pos=j;
      }
      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 will be 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=(float)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=(float)total/n;     //average turnaround time
    printf("\n\nAverage Waiting Time=%f",avg_wt);
    printf("\nAverage Turnaround Time=%f\n",avg_tat);
}
```

Output:



```
[188r1a0501@localhost ~]$ vi w1a.c
[188r1a0501@localhost ~]$ gcc  w1a.c
[188r1a0501@localhost ~]$ ./a.out
Enter number of process:3

Enter Burst Time:
p1:5
p2:3
p3:7

Process     Burst Time          Waiting Time    Turnaround Time
p2                3                   0                   3
p1                5                   3                   8
p3                7                   8                   15

Average Waiting Time=3.666667
Average Turnaround Time=8.666667
[188r1a0501@localhost ~]$
```

c) Round Robin

Aim: Write a C program to implement the various process scheduling mechanisms such as Round Robin Scheduling.

Algorithm

1: Start the process

2: Accept the number of processes in the ready Queue and time quantum (or) time slice

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

4: Calculate the no. of time slices for each process where

No. of time slice for process(n) = burst time process(n)/time slice

5: If the burst time is less than the time slice then the no. of time slices =1.

6: Consider the ready queue is a circular Q, calculate

(a) Waiting time for process(n) = waiting time of process(n-1)+ burst time of process(n-1) + the time difference in getting the CPU from process(n-1)

(b) Turn around time for process(n) = waiting time of process(n) + burst time of process(n)+ the time difference in getting CPU from process(n).

7: Calculate

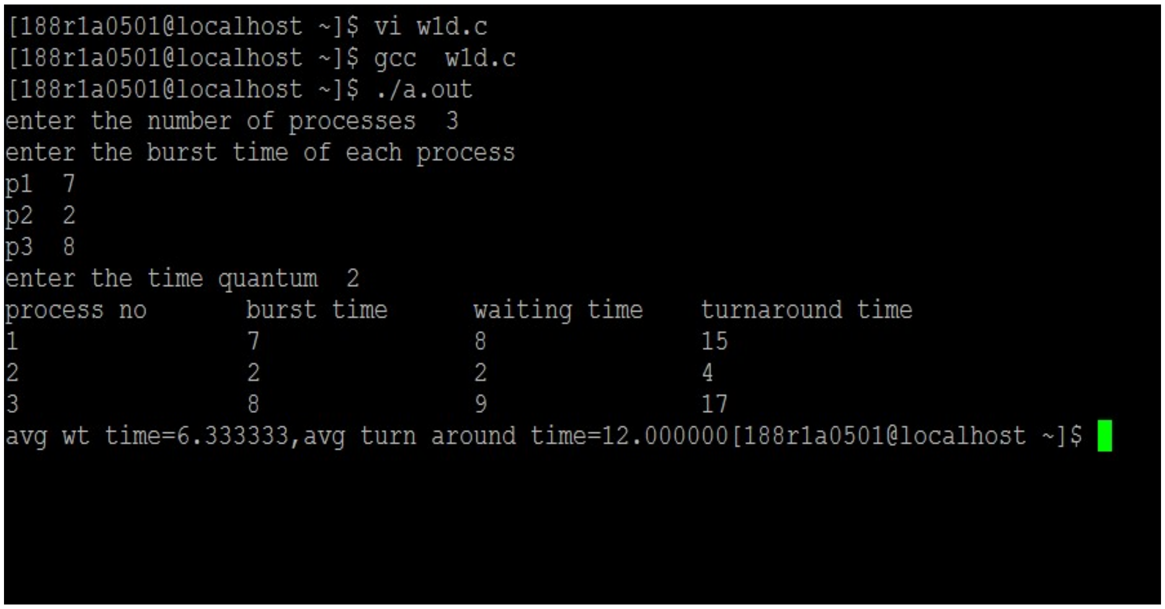
(a) Average waiting time = Total waiting Time / Number of process

(b) Average Turnaround time = Total Turnaround Time / Number of process Step 8: Stop the process

Program:

```
#include<stdio.h>
main()
{
        int st[10],bt[10],wt[10],tat[10],n,tq;
        int i,count=0,swt=0,stat=0,temp,sq=0;
        float awt,atat;
        printf("enter the number of processes");
        scanf("%d",&n);
        printf("enter the burst time of each process /n");
        for(i=0;i<n;i++)
        {
printf(("p%d",i+1);
scanf("%d",&bt[i]);
                st[i]=bt[i];
        }
        printf("enter the time quantum");
        scanf("%d",&tq);
        while(1)
        {
                for(i=0,count=0;i<n;i++)
                {
                        temp=tq;
                        if(st[i]==0)
                        {
                                count++;
                                continue;
                        }
                        if(st[i]>tq)
                        st[i]=st[i]-tq;
                        else
                        if(st[i]>=0)
                        {
                                temp=st[i];
                                st[i]=0;
                        }
                        sq=sq+temp;
                        tat[i]=sq;
                }
                if(n==count)
                break;
        }
        for(i=0;i<n;i++)
        {
                wt[i]=tat[i]-bt[i];
                swt=swt+wt[i];
                stat=stat+tat[i];
        }
        awt=(float)swt/n;
        atat=(float)stat/n;
        printf("process no\t burst time\t waiting time\t turnaround time\n");
        for(i=0;i<n;i++)
        printf("%d\t\t %d\t\t %d\t\t %d\n",i+1,bt[i],wt[i],tat[i]);
        printf("avg wt time=%f,avg turn around time=%f",awt,atat);
}
```

Output:



```
[188r1a0501@localhost ~]$ vi w1d.c
[188r1a0501@localhost ~]$ gcc  w1d.c
[188r1a0501@localhost ~]$ ./a.out
enter the number of processes  3
enter the burst time of each process
p1  7
p2  2
p3  8
enter the time quantum  2
process no       burst time       waiting time     turnaround time
1                7                8                15
2                2                2                4
3                8                9                17
avg wt time=6.333333,avg turn around time=12.000000[188r1a0501@localhost ~]$
```

d) Priority

Aim: Write a C program to implement the various process scheduling mechanisms such as Priority Scheduling.

Algorithm:

1: Start the process

2: Accept the number of processes in the ready Queue

3: For each process in the ready Q, assign the process id and accept the CPU burst time 4: Sort the ready queue according to the priority number.
5: Set the waiting of the first process as '0' and its burst time as its turn around time

6: For each process in the Ready Q calculate

(e) Waiting time for process(n)= waiting time of process (n-1) + Burst time of process(n-1)
(f) Turn around time for Process(n)= waiting time of Process(n)+ Burst time for process(n)
7: Calculate

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

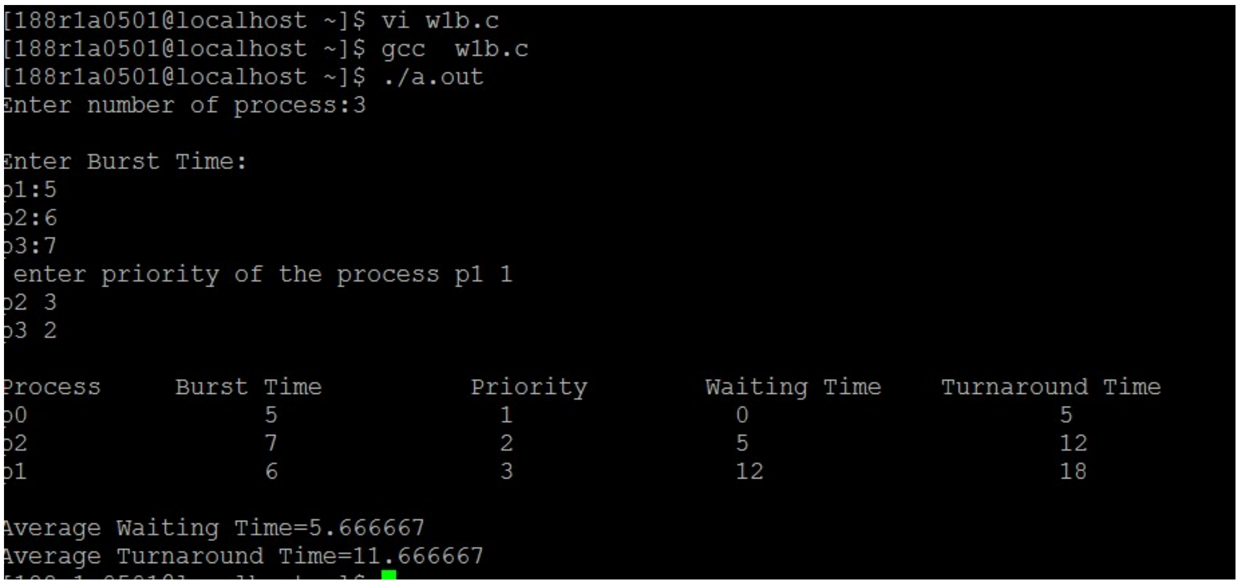
(h) Average Turnaround time = Total Turnaround Time / Number of process Step 8: Stop the process

Program:

```
#include<stdio.h>
int  main()
{
    int bt[20],p[20],wt[20],tat[20],pri[20],i,j,k,n,total=0,pos,temp;
    float avg_wt,avg_tat;
    printf("Enter number of process:");
    scanf("%d",&n);

    printf("\nEnter Burst Time:\n");
    for(i=0;i<n;i++)
    {
        printf("p%d:",i+1);
        scanf("%d",&bt[i]);
        p[i]=i+1;           //contains process number
    }
printf(" enter priority of the process ");
for(i=0;i<n;i++)
{
p[i] = i;
//printf("Priority of Process");
printf("p%d ",i+1);
scanf("%d",&pri[i]);
}
for(i=0;i<n;i++)
for(k=i+1;k<n;k++)
if(pri[i] > pri[k])
 {
  temp=p[i];
   p[i]=p[k];
   p[k]=temp;

  temp=bt[i];
bt[i]=bt[k];
bt[k]=temp;
temp=pri[i];
pri[i]=pri[k];
pri[k]=temp;
 }
```

```
    wt[0]=0;            //waiting time for first process will be 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=(float)total/n;      //average waiting time
    total=0;

    printf("\nProcess\t    Burst Time  \tPriority  \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  %d\t\t\t%d",p[i],bt[i],pri[i],wt[i],tat[i]);
    }

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

Output:



```
[188r1a0501@localhost ~]$ vi w1b.c
[188r1a0501@localhost ~]$ gcc  w1b.c
[188r1a0501@localhost ~]$ ./a.out
Enter number of process:3

Enter Burst Time:
p1:5
p2:6
p3:7
 enter priority of the process p1 1
p2 3
p3 2

Process     Burst Time           Priority        Waiting Time    Turnaround Time
p0              5                  1               0                     5
p2              7                  2               5                     12
p1              6                  3               12                    18

Average Waiting Time=5.666667
Average Turnaround Time=11.666667
```

WEEK-2

Write programs using the I/O system calls of UNIX/LINUX operating system (open, read, write, close, fcntl, seek, stat, opendir, readdir)

Aim: C program using open, read, write, close system calls

Theory:
There are 5 basic system calls that Unix provides for file I/O.
1. Create: Used to Create a new empty file
Syntax :int creat(char *filename, mode_t mode)
filename : name of the file which you want to create
mode : indicates permissions of new file.
2. **open**: Used to Open the file for reading, writing or both.
Syntax: int open(char *path, int flags [, int mode]);
Path : path to file which you want to use
flags : How you like to use
O_RDONLY: read only, O_WRONLY: write only, O_RDWR: read and write, O_CREAT: create file if it doesn't exist, O_EXCL: prevent creation if it already exists
3. **close**: Tells the operating system you are done with a file descriptor and Close the file which pointed by fd.
Syntax: int close(int fd);
fd :file descriptor
4. **read:** From the file indicated by the file descriptor fd, the read() function reads cnt bytes of input into the memory area indicated by buf. A successful read() updates the access time for the file.
Syntax: int read(int fd, char *buf, int size);
fd: file descripter
buf: buffer to read data from
cnt: length of buffer
5. **write**: Writes cnt bytes from buf to the file or socket associated with fd. cnt should not be greater than INT_MAX (defined in the limits.h header file). If cnt is zero, write() simply returns 0 without attempting any other action.
Syntax: int write(int fd, char *buf, int size);
fd: file descripter
buf: buffer to write data to
cnt: length of buffer

***File descriptor** is integer that uniquely identifies an open file of the process.

Algorithm

1. Star the program.
2. Open a file for O_RDWR for R/W,O_CREATE for creating a file ,O_TRUNC for truncate a file.
3. Using getchar(), read the character and stored in the string[] array.
4. The string [] array is write into a file close it.
5. Then the first is opened for read only mode and read the characters and displayed it and close the file.
6. Stop the program.

Program

```
#include<sys/stat.h>
#include<stdio.h>
#include<fcntl.h>
#include<sys/types.h>
int main()
{
   int n,i=0;
   int f1,f2;
   char c,strin[100];
   f1=open("data",O_RDWR|O_CREAT|O_TRUNC);
   while((c=getchar())!='\n')
   {
      strin[i++]=c;

   }
   strin[i]='\0';
   write(f1,strin,i);
   close(f1);
   f2=open("data",O_RDONLY);
   read(f2,strin,0);
   printf("\n%s\n",strin);
   close(f2);
  return 0;

}
```

Output:

```
Hai
Hai
```

b) Aim: C program using lseek

Theory:

lseek is a system call that is used to change the location of the read/write pointer of a file descriptor. The location can be set either in absolute or relative terms.

Syntax : off_t lseek(int fildes, off_t offset, int whence);

int fildes : The file descriptor of the pointer that is going to be moved.

off_t offset : The offset of the pointer (measured in bytes).

int whence : Legal values for this variable are provided at the end which are SEEK_SET (Offset is to be measured in absolute terms), SEEK_CUR (Offset is to be measured relative to the current location of the pointer), SEEK_END (Offset is to be measured relative to the end of the file)

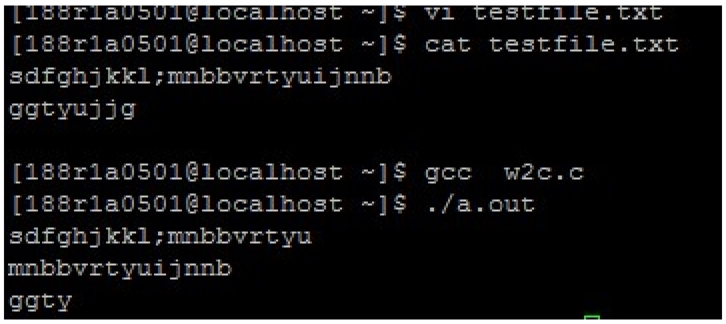
Algorithm:

1. Start the program
2. Open a file in read mode
3. Read the contents of the file
4. Use lseek to change the position of pointer in the read process
5. Stop

Program:

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

int main()
{
        int file=0;
        if((file=open("testfile.txt",O_RDONLY)) < -1)
                return 1;

        char buffer[19];
        if(read(file,buffer,19) != 19)  return 1;
        printf("%s\n",buffer);

        if(lseek(file,10,SEEK_SET) < 0) return 1;

        if(read(file,buffer,19) != 19)  return 1;
        printf("%s\n",buffer);

        return 0;
}
```

Output:



```
[188r1a0501@localhost ~]$ vi testfile.txt
[188r1a0501@localhost ~]$ cat testfile.txt
sdfghjkkl;mnbbvrtyuijnnb
ggtyujjg

[188r1a0501@localhost ~]$ gcc  w2c.c
[188r1a0501@localhost ~]$ ./a.out
sdfghjkkl;mnbbvrtyu
mnbbvrtyuijnnb
ggty
```

c) Aim: C program using opendir(), closedir(), readdir()

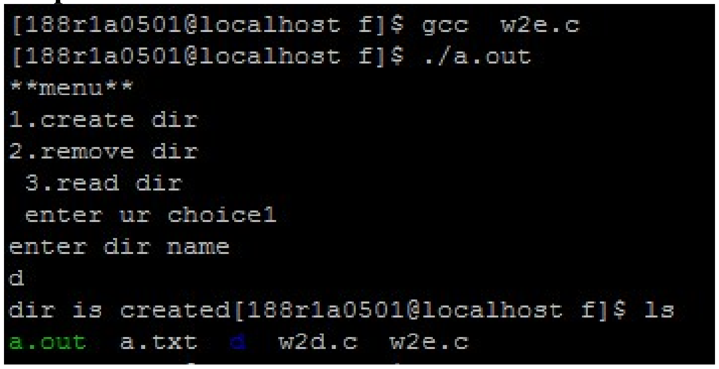
Theory:
The following are the various operations using directories

1. Creating directories.
   Syntax : int mkdir(const char *pathname, mode_t mode);
2. The 'pathname' argument is used for the name of the directory.
3. Opening directories
   Syntax : DIR *opendir(const char *name);
4. Reading directories.
   Syntax: struct dirent *readdir(DIR *dirp);
5. Removing directories.
   Syntax: int rmdir(const char *pathname);
6. Closing the directory.
   Syntax: int closedir(DIR *dirp);
7. Getting the current working directory.
   Syntax: char *getcwd(char *buf, size_t size);

Algorithm:
1. Start the program
2. Print a menu to choose the different directory operations
3. To create and remove a directory ask the user for name and create and remove the same respectively.
4. To open a directory check whether directory exists or not. If yes open the directory .If it does not exists print an error message.
5. Finally close the opened directory.
6. Stop

Program:
```
#include<stdio.h>
#include<fcntl.h>
#include<dirent.h>
main()
{
char d[10]; int c,op; DIR *e;
struct dirent *sd;
printf("**menu**\n1.create dir\n2.remove dir\n 3.read dir\n enter ur choice");
scanf("%d",&op);
switch(op)
{
case 1: printf("enter dir name\n"); scanf("%s",&d);
c=mkdir(d,777);
if(c==1)
printf("dir is not created");
 else
printf("dir is created"); break;
case 2: printf("enter dir name\n"); scanf("%s",&d);
c=rmdir(d);
if(c==1)
printf("dir is not removed");
 else
printf("dir is removed"); break;
case 3: printf("enter dir name to open");
scanf("%s",&d);
e=opendir(d);
if(e==NULL)
printf("dir does not exist"); else
{
printf("dir exist\n"); while((sd=readdir(e))!=NULL) printf("%s\t",sd->d_name);
}
closedir(e);
break;
}
}
```

Output:



```
[188r1a0501@localhost f]$ gcc  w2e.c
[188r1a0501@localhost f]$ ./a.out
**menu**
1.create dir
2.remove dir
 3.read dir
 enter ur choice1
enter dir name
d
dir is created[188r1a0501@localhost f]$ ls
a.out  a.txt  d  w2d.c  w2e.c
```

WEEK -3

Write a C program to simulate Bankers Algorithm for Deadlock Avoidance and Prevention

a) Aim

Write a C program to simulate the Bankers Algorithm for Deadlock Avoidance.

Data structures

1. n- Number of process, m-number of resource types.
2. Available: Available[j]=k, k – instance of resource type Rj is available.
3. Max: If max [i, j]=k, Pi may request at most k instances resource Rj.
4. Allocation: If Allocation [i, j]=k, Pi allocated to k instances of resource Rj
5. Need: If Need[I, j]=k, Pi may need k more instances of resource type Rj,
6. Need [I, j] =Max [I, j]-Allocation [I, j];

Safety Algorithm

1. Work and Finish be the vector of length m and n respectively, Work=Available and Finish[i] =False.
2. Find an i such that both
3. Finish[i] =False
4. Need<=Work
5. If no such I exist go to step 4.
6. work=work+Allocation, Finish[i] =True;
7. If Finish [1] =True for all I, then the system is in safe state.

Resource request algorithm

1. Let Request i be request vector for the process Pi, If request i=[j]=k, then process Pi wants k instances of resource type Rj.
2. If Request<=Need I go to step 2. Otherwise raise an error condition.
3. If Request<=Available go to step 3. Otherwise Pi must since the resources are available.
4. Have the system pretend to have allocated the requested resources to process Pi by modifying the state as follows;
5. Available=Available-Request I;
6. Allocation I =Allocation+Request I;
7. Need i=Need i-Request I;

If the resulting resource allocation state is safe, the transaction is completed and process Pi is allocated its resources. However, if the state is unsafe, the Pi must wait for Request i and the old resource-allocation state is restore.

Algorithm:

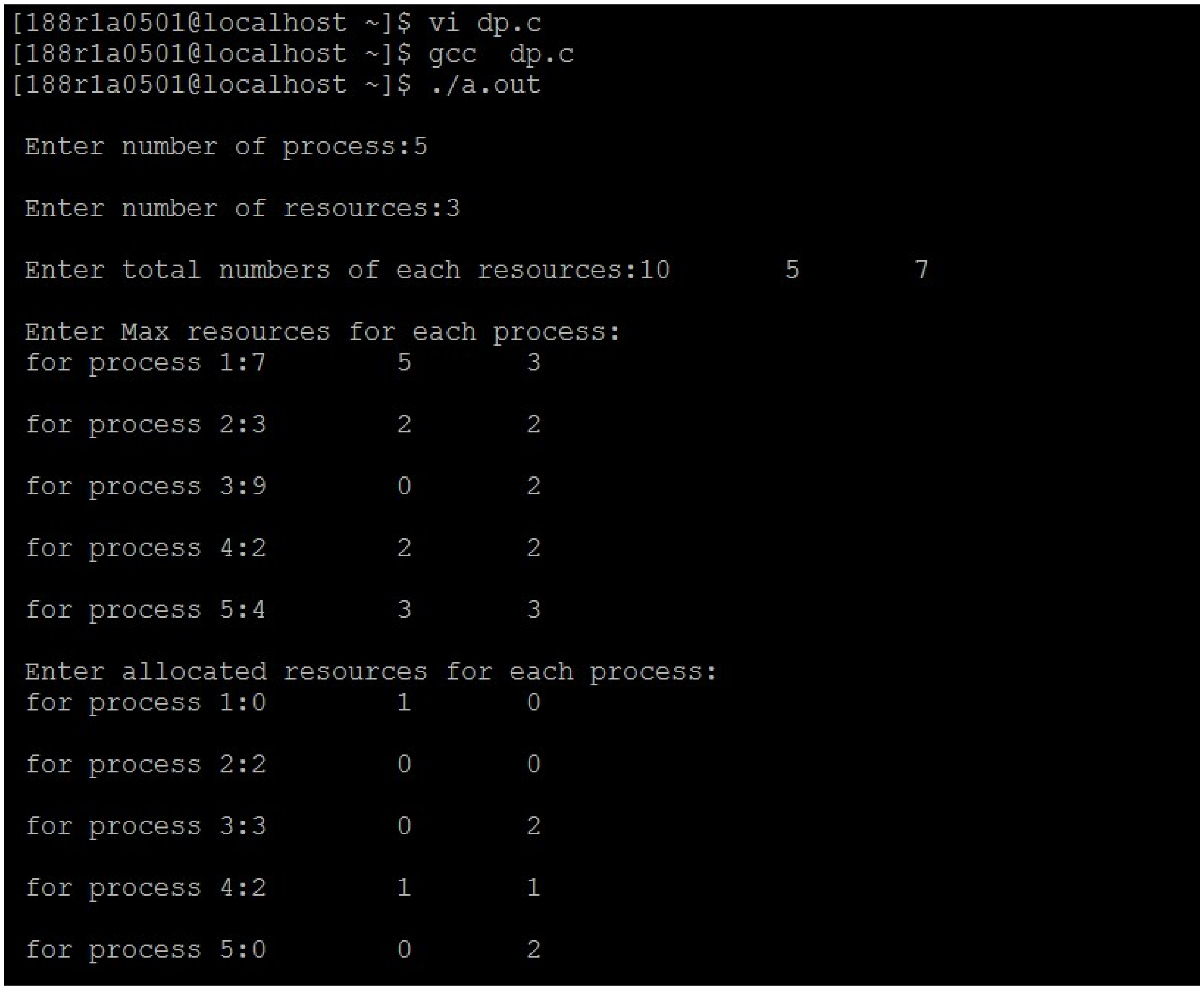
1. Start the program.
2. Get the values of resources and processes.
3. Get the avail value.
4. After allocation find the need value.
5. Check whether it is possible to allocate.
6. If it is possible then the system is in safe state.
7. Else system is not in safety state.
8. If the new request comes then check that the system is in safety.
9. Or not if we allow the request.
10. Stop the program.

Program:

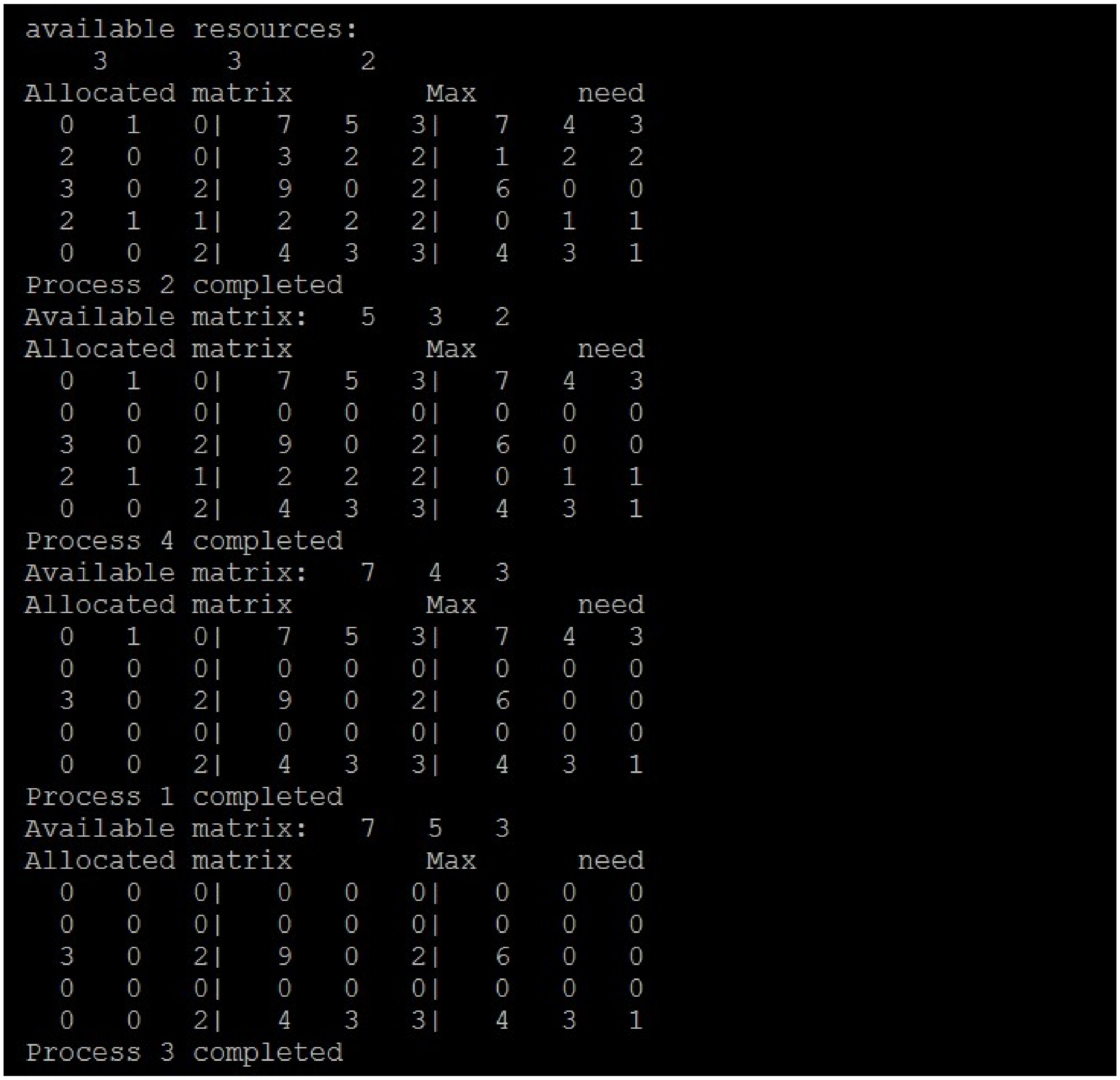
```
#include<stdio.h>

int  main ()
{
  int allocated[15][15], max[15][15], need[15][15], avail[15], tres[15],
    work[15], flag[15];
  int pno, rno, i, j, prc, count, t, total;
  count = 0;
  //clrscr ();

  printf ("\n Enter number of process:");
  scanf ("%d", &pno);
  printf ("\n Enter number of resources:");
  scanf ("%d", &rno);
  for (i = 1; i <= pno; i++)
    {
      flag[i] = 0;
    }
  printf ("\n Enter total numbers of each resources:");
  for (i = 1; i <= rno; i++)
    scanf ("%d", &tres[i]);


  printf ("\n Enter Max resources for each process:");
  for (i = 1; i <= pno; i++)
    {
      printf ("\n for process %d:", i);
      for (j = 1; j <= rno; j++)
        scanf ("%d", &max[i][j]);
    }


  printf ("\n Enter allocated resources for each process:");
  for (i = 1; i <= pno; i++)
    {
      printf ("\n for process %d:", i);
      for (j = 1; j <= rno; j++)
        scanf ("%d", &allocated[i][j]);


    }
printf ("\n available resources:\n");
  for (j = 1; j <= rno; j++)
    {
      avail[j] = 0;
      total = 0;
      for (i = 1; i <= pno; i++)
        {
          total += allocated[i][j];
        }
      avail[j] = tres[j] - total;
      work[j] = avail[j];
      printf ("     %d \t", work[j]);
    }



  do
    {

      for (i = 1; i <= pno; i++)
        {
          for (j = 1; j <= rno; j++)
            {
              need[i][j] = max[i][j] - allocated[i][j];

        }
  }



}
```

```
printf ("\n Allocated matrix        Max       need");
      for (i = 1; i <= pno; i++)
        {
          printf ("\n");
          for (j = 1; j <= rno; j++)
            {
              printf ("%4d", allocated[i][j]);
            }
          printf ("|");
          for (j = 1; j <= rno; j++)
            {
              printf ("%4d", max[i][j]);
            }
          printf ("|");
          for (j = 1; j <= rno; j++)
            {
              printf ("%4d", need[i][j]);
            }
        }

      prc = 0;

      for (i = 1; i <= pno; i++)
        {
          if (flag[i] == 0)
            {
              prc = i;

              for (j = 1; j <= rno; j++)
                {
                  if (work[j] < need[i][j])
                    {
                      prc = 0;
                      break;
                    }
                }
            }
if (prc != 0)
            break;
        }
      if (prc != 0)
        {
          printf ("\n Process %d completed", i);
          count++;
          printf ("\n Available matrix:");
          for (j = 1; j <= rno; j++)
            {
              work[j] += allocated[prc][j];
              allocated[prc][j] = 0;
              max[prc][j] = 0;
              flag[prc] = 1;
              printf ("  %d", work[j]);
            }
        }

    }
  while (count != pno && prc != 0);

  if (count == pno)
    printf ("\nThe system is in a safe state!!");
  else
    printf ("\nThe system is in an unsafe state!!");
return 0;

}
```

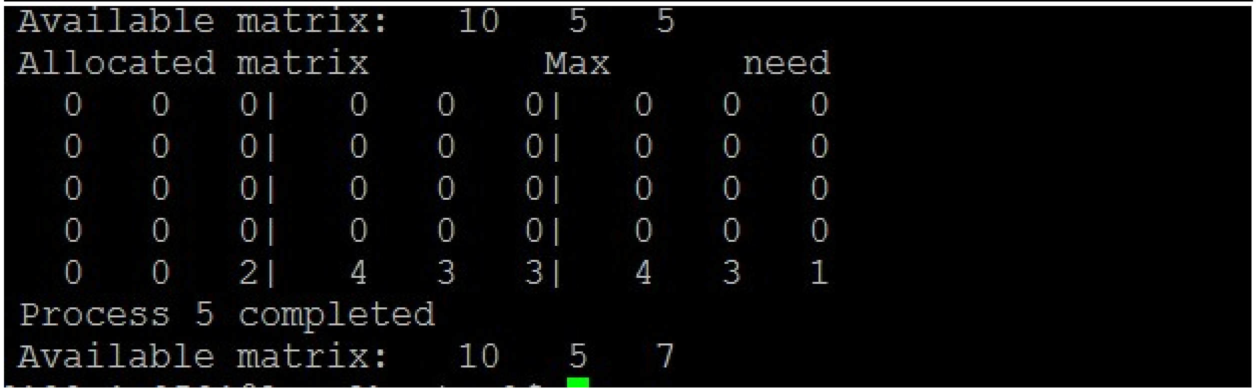
Output:



```
[188r1a0501@localhost ~]$ vi dp.c
[188r1a0501@localhost ~]$ gcc  dp.c
[188r1a0501@localhost ~]$ ./a.out

 Enter number of process:5

 Enter number of resources:3

 Enter total numbers of each resources:10       5       7

 Enter Max resources for each process:
 for process 1:7        5       3

 for process 2:3        2       2

 for process 3:9        0       2

 for process 4:2        2       2

 for process 5:4        3       3

 Enter allocated resources for each process:
 for process 1:0        1       0

 for process 2:2        0       0

 for process 3:3        0       2

 for process 4:2        1       1

 for process 5:0        0       2
```



```
available resources:
    3       3       2
Allocated matrix        Max       need
  0   1   0|   7   5   3|   7   4   3
  2   0   0|   3   2   2|   1   2   2
  3   0   2|   9   0   2|   6   0   0
  2   1   1|   2   2   2|   0   1   1
  0   0   2|   4   3   3|   4   3   1
Process 2 completed
Available matrix:   5   3   2
Allocated matrix        Max       need
  0   1   0|   7   5   3|   7   4   3
  0   0   0|   0   0   0|   0   0   0
  3   0   2|   9   0   2|   6   0   0
  2   1   1|   2   2   2|   0   1   1
  0   0   2|   4   3   3|   4   3   1
Process 4 completed
Available matrix:   7   4   3
Allocated matrix        Max       need
  0   1   0|   7   5   3|   7   4   3
  0   0   0|   0   0   0|   0   0   0
  3   0   2|   9   0   2|   6   0   0
  0   0   0|   0   0   0|   0   0   0
  0   0   2|   4   3   3|   4   3   1
Process 1 completed
Available matrix:   7   5   3
Allocated matrix        Max       need
  0   0   0|   0   0   0|   0   0   0
  0   0   0|   0   0   0|   0   0   0
  3   0   2|   9   0   2|   6   0   0
  0   0   0|   0   0   0|   0   0   0
  0   0   2|   4   3   3|   4   3   1
Process 3 completed
```



```
Available matrix:   10   5   5
Allocated matrix        Max       need
  0   0   0|   0   0   0|   0   0   0
  0   0   0|   0   0   0|   0   0   0
  0   0   0|   0   0   0|   0   0   0
  0   0   0|   0   0   0|   0   0   0
  0   0   2|   4   3   3|   4   3   1
Process 5 completed
Available matrix:   10   5   7
```

b) Aim

Write a C program to simulate Bankers Algorithm for Deadlock Prevention

Algorithm:

1. Start
2. Attacking Mutex condition : never grant exclusive access. but this may not be possible for several resources.
3. Attacking preemption: not something you want to do.
4. Attacking hold and wait condition : make a process hold at the most 1 resource at a time.make all the requests at the beginning. All or nothing policy. If you feel,retry. eg. 2-phase locking 34
5. Attacking circular wait: Order all the resources. Make sure that the requests are issued in the correct order so that there are no cycles present in the resource graph. Resources numbered 1 ... n. Resources can be requested only in increasing order. ie. you cannot request a resource whose no is less than any you may be holding.
6. Stop

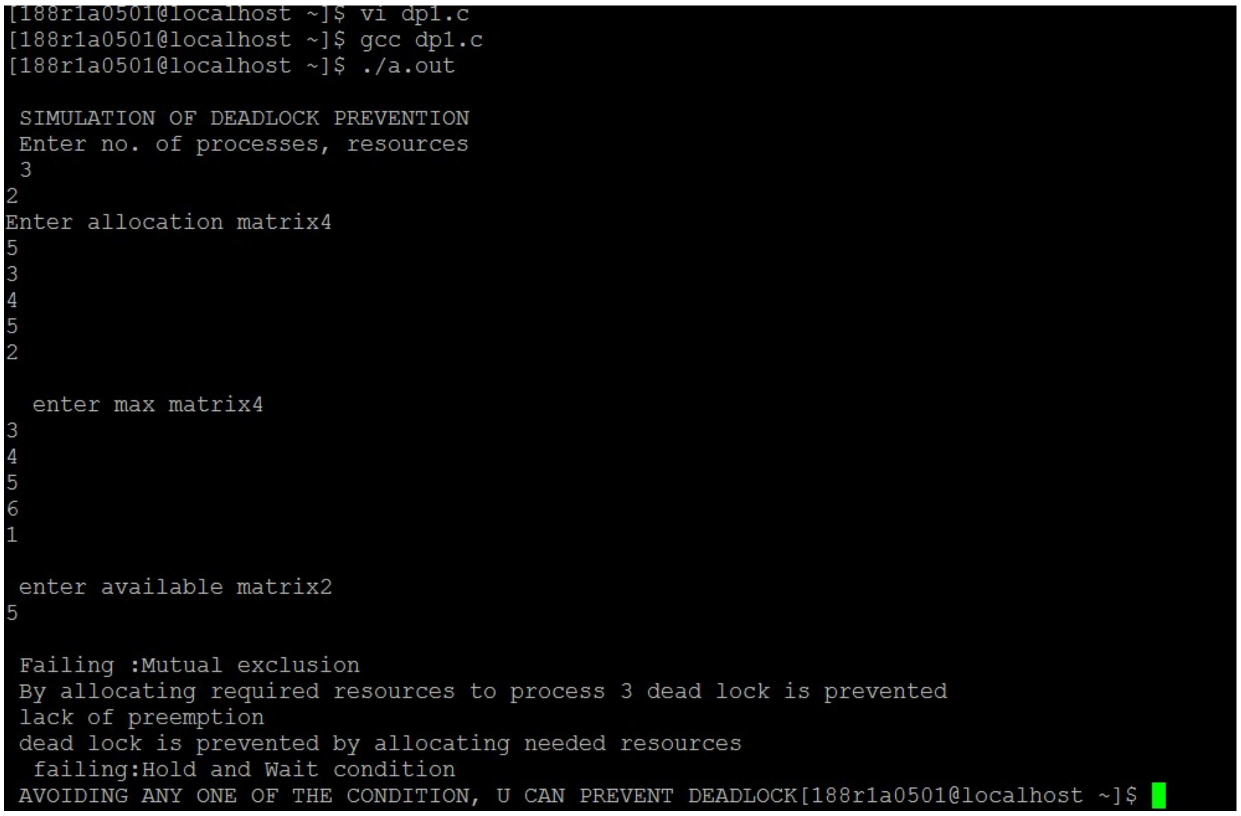
Program:

```
#include<stdio.h>

int max[10][10],alloc[10][10],need[10][10],avail[10],i,j,p,r,finish[10]={0},flag=0;
main( )
{

printf("\n SIMULATION OF DEADLOCK PREVENTION \n ");
printf("Enter no. of processes, resources\n ");
scanf("%d%d",&p,&r);
printf("Enter allocation matrix");
for(i=0;i<p;i++)
for(j=0;j<r;j++)
scanf("%d",&alloc[i][j]);
printf("\n  enter max matrix");
for(i=0;i<p;i++) /*reading the maximum matrix and availale matrix*/
for(j=0;j<r;j++)
scanf("%d",&max[i][j]);
printf(" \n enter available matrix");
for(i=0;i<r;i++)
scanf("%d",&avail[i]);
```

```
for(i=0;i<p;i++)
for(j=0;j<r;j++)
need[i][j]=max[i][j]-alloc[i][j];
fun(); /*calling function*/
if(flag==0)
{if(finish[i]!=1)
{
printf("\n Failing :Mutual exclusion");
for(j=0;j<r;j++)
{ /*checking for mutual exclusion*/
if(avail[j]<need[i][j])
avail[j]=need[i][j];
}fun();
printf("\n By allocating required resources to process %d dead lock is prevented ",i);
printf("\n lack of preemption");
for(j=0;j<r;j++)
{
if(avail[j]<need[i][j])
avail[j]=need[i][j];
alloc[i][j]=0;
}
fun( );
printf("\n dead lock is prevented by allocating needed resources");

printf(" \n failing:Hold and Wait condition ");
for(j=0;j<r;j++)
{ /*checking hold and wait condition*/
if(avail[j]<need[i][j])
avail[j]=need[i][j];
}
fun( );
printf("\n AVOIDING ANY ONE OF THE CONDITION, U CAN PREVENT DEADLOCK");
}
}
}
fun()
{
while(1)
{
for(flag=0,i=0;i<p;i++)
```

```
{
if(finish[i]==0)
{
for(j=0;j<r;j++)
{
if(need[i][j]<=avail[j])
continue;
else
break;
}
if(j==r)
{
for(j=0;j<r;j++)
avail[j]+=alloc[i][j];
flag=1;
finish[i]=1;
}
}
}
```

Output:



```
[188r1a0501@localhost ~]$ vi dp1.c
[188r1a0501@localhost ~]$ gcc dp1.c
[188r1a0501@localhost ~]$ ./a.out

 SIMULATION OF DEADLOCK PREVENTION
 Enter no. of processes, resources
 3
2
Enter allocation matrix4
5
3
4
5
2

  enter max matrix4
3
4
5
6
1

 enter available matrix2
5

 Failing :Mutual exclusion
 By allocating required resources to process 3 dead lock is prevented
 lack of preemption
 dead lock is prevented by allocating needed resources
  failing:Hold and Wait condition
 AVOIDING ANY ONE OF THE CONDITION, U CAN PREVENT DEADLOCK[188r1a0501@localhost ~]$
```

WEEK-4

Write a C program to implement the Producer – Consumer problem using semaphores using UNIX/LINUX system calls.

Aim:

Write a C program to implement the Producer – Consumer problem using semaphores using UNIX/LINUX system calls.

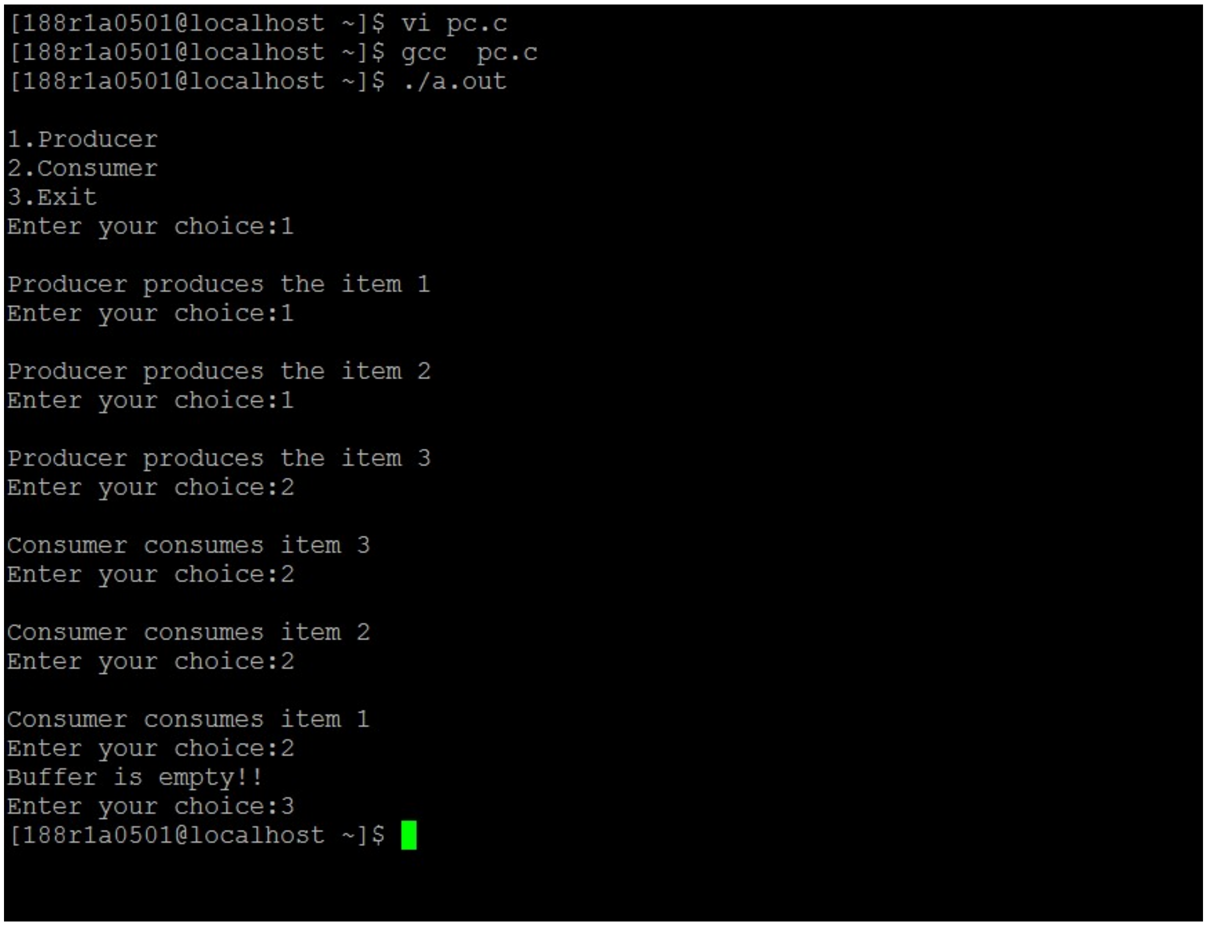
Algorithm:

1. The Semaphore mutex, full & empty are initialized.
2. In the case of producer process
3. Produce an item in to temporary variable.
   If there is empty space in the buffer check the mutex value for enter into the critical section.
   If the mutex value is 0, allow the producer to add value in the temporary variable to the buffer.
4. In the case of consumer process
   i) It should wait if the buffer is empty
   ii) If there is any item in the buffer check for mutex value, if the mutex==0, remove item from buffer
   iii) Signal the mutex value and reduce the empty value by 1.
   iv) Consume the item.
5. Print the result

Program:

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

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

int main ()
{
 int n;
 void producer ();
 void consumer ();
 int wait (int);
 int signal (int);
 printf ("\n1.Producer\n2.Consumer\n3.Exit");
 while (1)
   {
     printf ("\nEnter 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;
          }
   }

 return 0;
}

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 ("\nConsumer consumes item %d", x);
  x--;
  mutex = signal (mutex);
}
```

Output:



```
[188r1a0501@localhost ~]$ vi pc.c
[188r1a0501@localhost ~]$ gcc  pc.c
[188r1a0501@localhost ~]$ ./a.out

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: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
[188r1a0501@localhost ~]$
```

Week: 5

Write C programs to illustrate the following IPC mechanisms

Aim: Write C programs to illustrate the following IPC mechanisms

ALGORITHM:

1. Start the program.
2. Declare the variables.
3. Read the choice.
4. Create a piping processing using IPC.
5. Assign the variable lengths
6. "*strcpy*" the message lengths.
7. To join the operation using IPC .
8. Stop the program

Program : (PIPE PROCESSING)

```
#include <unistd.h>
#include <stdlib.h> #include <stdio.h>
#include <string.h> #define MSG_LEN 64 int main(){
Int result;
int fd[2];
char  message[MSG_LEN];
 char recvd_msg[MSG_LE]; result = pipe (fd);
//Creating a pipe//fd[0] is for reading and fd[1] is for writing if (result < 0){
perror("pipe "); exit(1);
}
strncpy(message,"Linux  World!!  ",MSG_LEN);  result=write(fd[1],message,strlen(message));  if (result
< 0){
perror("write"); exit(2);
}
strncpy(message,"Understanding ",MSG_LEN); result=write(fd[1],message,strlen(message)); if
(result < 0){
perror("write"); exit(2);
}
strncpy(message,"Concepts of ",MSG_LEN); result=write(fd[1],message,strlen(message)); if (result <
0){

perror("write");  exit(2);
}

strncpy(message,"Piping ", MSG_LEN);  result=write(fd[1],message,strlen(message));  if (result  <  0){
perror("write"); exit(2);
}

result=read(fd[0],recvd_msg,MSG_LEN); if (result < 0){
perror("read"); exit(3);
}

printf("%s\n",recvd_msg); return 0;
}
```

a) FIFO

Program:

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

#include <linux/stat.h>

#define FIFO_FILE       "MYFIFO"

int main(void)
{
        FILE *fp;
        char readbuf[80];

        /* Create the FIFO if it does not exist */
        umask(0);
        mknod(FIFO_FILE, S_IFIFO|0666, 0);

        while(1)
        {
                fp = fopen(FIFO_FILE, "r");
                fgets(readbuf, 80, fp);
                printf("Received string: %s\n", readbuf);
                fclose(fp);
        }

        return(0);
}
```

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

#define FIFO_FILE       "MYFIFO"

int main(int argc, char *argv[])
{
        FILE *fp;

        if ( argc != 2 ) {
                printf("USAGE: fifoclient [string]\n");
                exit(1);
        }

        if((fp = fopen(FIFO_FILE, "w")) == NULL) {
                perror("fopen");
                exit(1);
        }
        fputs(argv[1], fp);

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

C Program for Message Queue (Writer Process)

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

// structure for message queue
struct mesg_buffer {
   long  msg_type;
   char msg_text[100];
} message;

int main()
{
   key_t key;
   int msgid;
  // ftok to generate unique key
   key = ftok("progfile", 65);
 // msgget creates a message queue
   // and returns identifier
   msgid = msgget(key, 0666 | IPC_CREAT);
   message.mesg_type = 1;

   printf("Write Data : ");
   gets(message.mesg_text);

   // msgsnd to send message
   msgsnd(msgid, &message, sizeof(message), 0);

   // display the message
   printf("Data send is : %s \n", message.mesg_text);

   return 0;
}
```

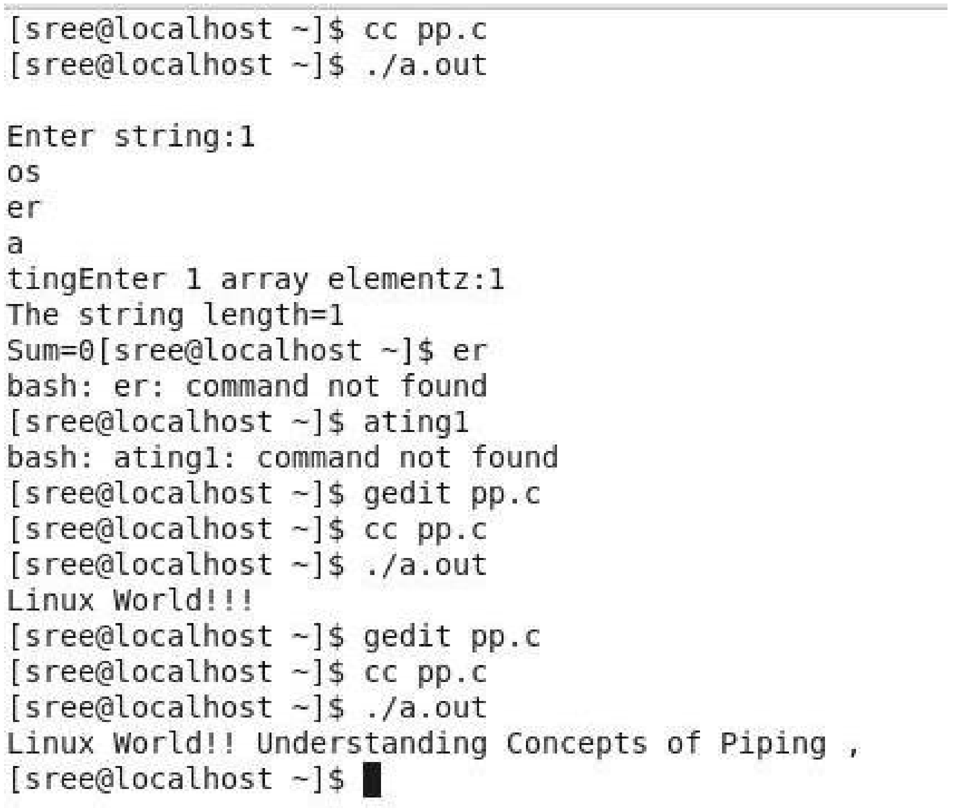
C Program for Message Queue (Reader Process)

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

// structure for message queue
struct mesg_buffer {
   long mesg_type;
```

```
    char mesg_text[100];
} message;

int main()
{
    key_t key;
    int msgid;

    // ftok to generate unique key
key = ftok("progfile", 65);



// msgget creates a message queue  and returns identifier
    msgid = msgget(key, 0666 | IPC_CREAT);
    // msgrcv to receive message
    msgrcv(msgid, &message, sizeof(message), 1, 0);

    // display the message
    printf("Data Received is : %s \n",
                    message.mesg_text);

    // to destroy the message queue
    msgctl(msgid, IPC_RMID, NULL);

    return 0;
}
```

C Program for Message Queue (Reader Process)

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>
  // structure for message queue
struct mesg_buffer {
    long  mesg_type;
    char mesg_text[100];
} message;

int main()
{
    key_t key;
    int msgid;
    // ftok to generate unique key
    key = ftok("progfile", 65);

    // msgget creates a message queue
    // and returns identifier
    msgid = msgget(key, 0666 | IPC_CREAT);

    // msgrcv to receive message
    msgrcv(msgid, &message, sizeof(message), 1, 0);

    // display the message
    printf("Data Received is : %s \n",
                    message.mesg_text);

    // to destroy the message queue
    msgctl(msgid, IPC_RMID, NULL);


    return 0;
}
```
OUTPUT: Thus the Piping process using IPC program was executed and verified successfully



```
[sree@localhost ~]$ cc pp.c
[sree@localhost ~]$ ./a.out

Enter string:1
os
er
a
tingEnter 1 array elementz:1
The string length=1
Sum=0[sree@localhost ~]$ er
bash: er: command not found
[sree@localhost ~]$ ating1
bash: ating1: command not found
[sree@localhost ~]$ gedit pp.c
[sree@localhost ~]$ cc pp.c
[sree@localhost ~]$ ./a.out
Linux World!!!
[sree@localhost ~]$ gedit pp.c
[sree@localhost ~]$ cc pp.c
[sree@localhost ~]$ ./a.out
Linux World!! Understanding Concepts of Piping ,
[sree@localhost ~]$ █
```

Week: 6
Aim: Write C programs to simulate the following memory management techniques

a) Paging

AIM: To write a C program to implement memory management using paging technique.

ALGORITHM:

```
Step1 : Start the program.
Step2 : Read the base address, page size, number of pages and memory unit.
Step3 : If the memory limit is less than the base address display the memory limit is less than limit.
Step4 : Create the page table with the number of pages and page address.
Step5 : Read the page number and displacement value.
Step6 : If the page number and displacement value is valid, add the displacement value with the address
        corresponding to the page number and display the result.
Step7 : Display the page is not found or displacement should be less than page size.
Step8 : Stop the program.
```

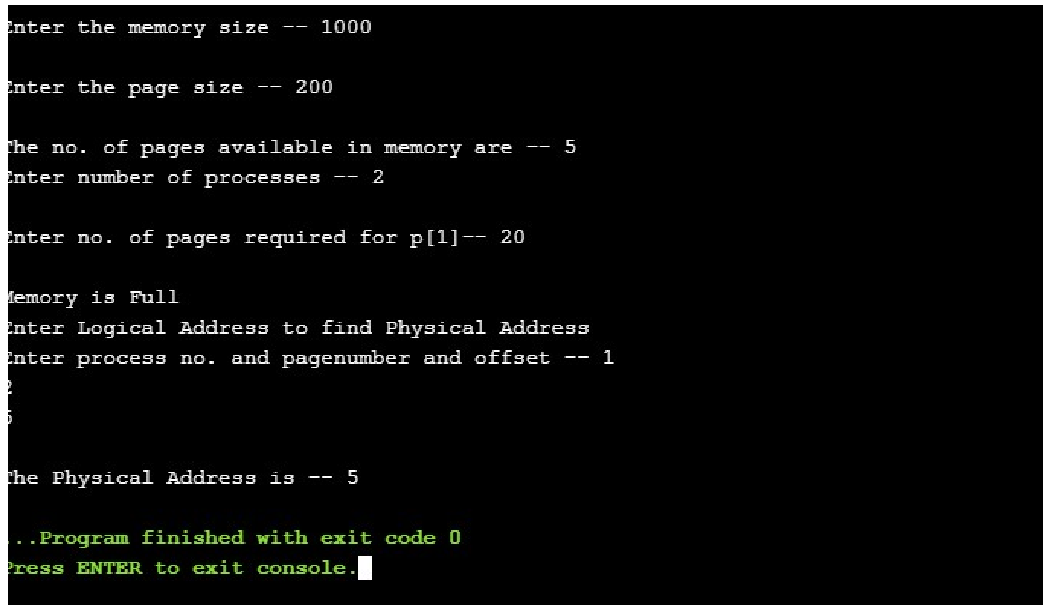
Program:

```
#include<stdio.h>

#include<conio.h>

main()

{

int ms, ps, nop, np, rempages, i, j, x, y, pa, offset; int s[10], fno[10][20];

printf("\nEnter the memory size -- ");

scanf("%d",&ms);

printf("\nEnter the page size -- ");

scanf("%d",&ps);

nop = ms/ps;

printf("\nThe no. of pages available in memory are -- %d ",nop);

printf("\nEnter number of processes -- ");

scanf("%d",&np);
```

```
rempages = nop; for(i=1;i<=np;i++)
{

printf("\nEnter no. of pages required for p[%d]-- ",i);
scanf("%d",&s[i]);
if(s[i] >rempages)
{
printf("\nMemory is Full");
break;
}
rempages = rempages - s[i];
printf("\nEnter pagetable for p[%d] --- ",i);
for(j=0;j<s[i];j++)
scanf("%d",&fno[i][j]);
}

printf("\nEnter Logical Address to find Physical Address ");
printf("\nEnter process no. and pagenumber and offset -- ");
scanf("%d %d %d",&x,&y, &offset);
if(x>np || y>=s[i] || offset>=ps)
printf("\nInvalid Process or Page Number or offset");
else
{
pa=fno[x][y]*ps+offset;
printf("\nThe Physical Address is -- %d",pa);
}
```

```
getch();

}
```

OUTPUT:



```
nter the memory size -- 1000

nter the page size -- 200

he no. of pages available in memory are -- 5
nter number of processes -- 2

nter no. of pages required for p[1]-- 20

emory is Full
nter Logical Address to find Physical Address
nter process no. and pagenumber and offset -- 1


he Physical Address is -- 5

..Program finished with exit code 0
ress ENTER to exit console.
```

b) Segmentation

Aim: To write a C program to implement memory management using segmentation

Algorithm:

Step1 : Start the program.
Step2 : Read the base address, number of segments, size of each segment, memory limit.Step3 : If memory address is less than the base address display "invalid memory limit".
Step4 : Create the segment table with the segment number and segment address and display it.Step5 : Read the segment number and displacement.
Step6 : If the segment number and displacement is valid compute the real address and display the same.
Step7 :Stop the program.

Program:

```
#include<stdio.h>
#include<conio.h>
struct list
{
int seg;
int base;
int limit;
struct list *next;
} *p;
void insert(struct list *q,int base,int limit,int seg)
{
if(p==NULL)
{
p=malloc(sizeof(Struct list));
p->limit=limit;
p->base=base;
p->seg=seg;
p->next=NULL;
}
else
{
while(q->next!=NULL)
{
Q=q->next;
Printf("yes")
}
q->next=malloc(sizeof(Struct list));
q->next ->limit=limit;
q->next ->base=base;
```

```
q->next ->seg=seg;
```

```
q->next ->next=NULL;
}
}
int find(struct list *q,int seg)
{
while(q->seg!=seg)
{
q=q->next;
}
return q->limit;
}
int search(struct list *q,int seg)
{
while(q->seg!=seg)
{
q=q->next;
}
return q->base;
}
main()
{


p=NULL;
int seg,offset,limit,base,c,s,physical;
printf("Enter segment table/n");
printf("Enter -1 as segment value for termination\n");
do
{
printf("Enter segment number");
scanf("%d",&seg);
if(seg!=-1)
{
printf("Enter base value:");
scanf("%d",&base);

printf("Enter value for limit:");
scanf("%d",&limit);
insert(p,base,lmit,seg);
}
}
while(seg!=-1)
printf("Enter offset:");
```

```
scanf("%d",&offset);
printf("Enter bsegmentation number:");
scanf("%d",&seg);
c=find(p,seg);
s=search(p,seg);
if(offset<c)
{
physical=s+offset;
printf("Address in physical memory %d\n",physical);
}
else
{
printf("error");
}
```

OUTPUT:

```
Enter segment table
Enter -1 as segmentation value for termination
Enter segment number:1
Enter base value:2000
Enter value for limit:100
Enter segment number:2
Enter base value:2500
Enter value for limit:100
Enter segmentation number:-1
Enter offset:90
Enter segment number:2
Address in physical memory 2590

Enter segment table
Enter -1 as segmentation value for termination
Enter segment number:1
Enter base value:2000
Enter value for limit:100
Enter segment number:2
Enter base value:2500
Enter value for limit:100
Enter segmentation number:-1
Enter offset:90
Enter segment number:1
Address in physical memory 20
```