Department of CSE

Laboratory Manual	
Course:	B.Tech.
Year & Semester:	II – II
Class:	CSE
Subject:	**OPERATING SYSTEM LAB**
Regulation:	R22

OPERATING SYSTEMS LAB SYLLABUS

(Practical Hours: 03, Credits: 02)

Implement the following programs on Linux platform using C language.

Exp.No.	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 using UNIX/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 management techniques a) Paging b) Segmentation
7	Write C programs to simulate Page replacement policies a) FCFS b) LRU c) Optimal

OPERATING SYSTEMS LABORATORY

OBJECTIVE:

This lab complements the operating systems course. Students will gain practical experience with designing and implementing concepts of operating systems such as system calls, CPU scheduling, process management, memory management, file systems and deadlock handling using C language in Linux environment.

OUTCOMES:

Upon the completion of Operating Systems practical course, the student will be able to:

1. **Understand** and implement basic services and functionalities of the operating system using system calls.
2. **Use** modern operating system calls and synchronization libraries in software/ hardware interfaces**.**
3. **Understand** the benefits of thread over process and implement synchronized programs using multithreading concepts.
4. **Analyze** and simulate CPU Scheduling Algorithms like FCFS, Round Robin, SJF, and Priority.
5. **Implement** memory management schemes and page replacement schemes.
6. **Simulate** file allocation and organization techniques.
7. **Understand** the concepts of deadlock in operating systems and implement them in multiprogramming system.

EXPERIMENT 1

1.1 OBJECTIVE

Write a C program to simulate the following non-preemptive CPU scheduling algorithms to find turnaround time and waiting time for the above problem.

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

1.2 DESCRIPTION

Assume all the processes arrive at the same time.

1.2.1 FCFS CPU SCHEDULING ALGORITHM

For FCFS scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times. The scheduling is performed on the basis of arrival time of the processes irrespective of their other parameters. Each process will be executed according to its arrival time. Calculate the waiting time and turnaround time of each of the processes accordingly.

1.2.2 SJF CPU SCHEDULING ALGORITHM

For SJF scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times. Arrange all the jobs in order with respect to their burst times. There may be two jobs in queue with the same execution time, and then FCFS approach is to be performed. Each process will be executed according to the length of its burst time. Then calculate the waiting time and turnaround time of each of the processes accordingly.

1.2.3 ROUND ROBIN CPU SCHEDULING ALGORITHM

For round robin scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times, and the size of the time slice. Time slices are assigned to each process in equal portions and in circular order, handling all processes execution. This allows every process to get an equal chance. Calculate the waiting time and turnaround time of each of the processes accordingly.

1.2.4 PRIORITY CPU SCHEDULING ALGORITHM

For priority scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times, and the priorities. Arrange all the jobs in order with respect to their priorities. There may be two jobs in queue with the same priority, and then FCFS approach is to be performed. Each process will be executed according to its priority. Calculate the waiting time and turnaround time of each of the processes accordingly.

1.3 PROGRAM

1.3.1 FCFS CPU SCHEDULING ALGORITHM

```
#include<stdio.h>
#include<conio.h>
main()
{
        int bt[20], wt[20], tat[20], i, n;
        float wtavg, tatavg;
        clrscr();
        printf("\nEnter the number of processes -- ");
                scanf("%d", &n);
        for(i=0;i<n;i++)
        {
                printf("\nEnter Burst Time for Process %d -- ", i);
                scanf("%d", &bt[i]);
        }
        wt[0] = wtavg = 0;
        tat[0] = tatavg = bt[0];
        for(i=1;i<n;i++)
        {
                wt[i] = wt[i-1] +bt[i-1];
                tat[i] = tat[i-1] +bt[i];
                wtavg = wtavg + wt[i];
                tatavg = tatavg + tat[i];
        }
        printf("\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");
```

```
        for(i=0;i<n;i++)
                printf("\n\t P%d \t\t %d \t\t %d \t\t %d", i, bt[i], wt[i], tat[i]);
        printf("\nAverage Waiting Time -- %f", wtavg/n);
        printf("\nAverage Turnaround Time -- %f", tatavg/n);
        getch();
}
```

INPUT

```
Enter the number of processes --    3
Enter Burst Time for Process  0 --    24
Enter Burst Time for Process  1 --    3
Enter Burst Time for Process  2 --    3
```

OUTPUT

PROCESS	BURST TIME	WAITING TIME	TURNAROUND TIME
P0	24	0	24
P1	3	24	27
P2	3	27	30

```
Average Waiting Time--     17.000000
Average Turnaround Time --          27.000000
```

1.3.2 SJF CPU SCHEDULING ALGORITHM

```
 #include<stdio.h>
#include<conio.h>
main()
{
int p[20], bt[20], wt[20], tat[20], i, k, n, temp;float wtavg, tatavg;
clrscr();
printf("\nEnter the number of processes -- ");
scanf("%d", &n);
for(i=0;i<n;i++)
{
p[i]=i;
printf("Enter Burst Time for Process %d -- ", i);
scanf("%d", &bt[i]);
}
for(i=0;i<n;i++)
for(k=i+1;k<n;k++)
if(bt[i]>bt[k])
{
temp=bt[i];
bt[i]=bt[k];
bt[k]=temp;
temp=p[i];
p[i]=p[k]
p[k]=temp;

}
wt[0] = wtavg = 0;
tat[0] = tatavg = bt[0];
for(i=1;i<n;i++)
{
wt[i] = wt[i-1] +bt[i-1];
tat[i] = tat[i-1] +bt[i];
wtavg = wtavg + wt[i];
tatavg = tatavg + tat[i];
}
printf("\n\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");
for(i=0;i<n;i++)
```

```
printf("\n\t P%d \t\t %d \t\t %d \t\t %d", p[i], bt[i], wt[i], tat[i]);
printf("\nAverage Waiting Time -- %f", wtavg/n);
printf("\nAverage Turnaround Time -- %f", tatavg/n);getch();
        }
```

INPUT

```
Enter the number of processes --    4
Enter Burst Time for Process  0 --    6
Enter Burst Time for Process  1 --    8
Enter Burst Time for Process  2 --    7
Enter Burst Time for Process  3 --    3
```

OUTPUT

PROCESS	BURST TIME	WAITING TIME	TURNAROUND TIME
P3	3	0	3
P0	6	3	9
P2	7	9	16
P1	8	16	24

Average Waiting Time -- 7.000000
Average Turnaround Time -- 13.000000

1.3.3 ROUND ROBIN CPU SCHEDULING ALGORITHM

```
#include<stdio.h>
main()
{
int i,j,n,bu[10],wa[10],tat[10],t,ct[10],max;
float awt=0,att=0,temp=0;
                clrscr();
                printf("Enter the no of processes -- ");
                scanf("%d",&n);

                for(i=0;i<n;i++)
                {
                        printf("\nEnter Burst Time for process %d -- ", i+1);
                        scanf("%d",&bu[i]);
                        ct[i]=bu[i];
                }
                printf("\nEnter the size of time slice -- ");
                scanf("%d",&t);
                max=bu[0];
                for(i=1;i<n;i++)
                        if(max<bu[i])
                                max=bu[i];
                for(j=0;j<(max/t)+1;j++)
                        for(i=0;i<n;i++)
                                if(bu[i]!=0)
                                        if(bu[i]<=t)
                                        {
                                                tat[i]=temp+bu[i];
                                                temp=temp+bu[i];
                                                bu[i]=0;
                                        }
                                        else
                                        {
                                                bu[i]=bu[i]-t;
                                                temp=temp+t;
                                        }
                for(i=0;i<n;i++)
                {
                        wa[i]=tat[i]-ct[i];
                        att+=tat[i];
```

```
                awt+=wa[i];
        }
        printf("\nThe Average Turnaround time is -- %f",att/n);
        printf("\nThe Average Waiting time is -- %f  ",awt/n);
        printf("\n\tPROCESS\t BURST TIME \t WAITING TIME\tTURNAROUND TIME\n");
        for(i=0;i<n;i++)
                printf("\t%d \t %d \t\t %d \t\t %d \n",i+1,ct[i],wa[i],tat[i]);
        getch();
}
```

INPUT

Enter the no of processes – 3
Enter Burst Time for process 1 – 24
Enter Burst Time for process 2 -- 3
Enter Burst Time for process 3 -- 3

Enter the size of time slice – 3

OUTPUT

The Average Turnaround time is – 15.666667
The Average Waiting time is -- 5.666667

PROCESS	BURST TIME	WAITING TIME	TURNAROUND TIME
1	24	6	30
2	3	4	7
3	3	7	10

1.3.4 PRIORITY CPU SCHEDULING ALGORITHM

```
#include<stdio.h>
main()
{
        int p[20],bt[20],pri[20], wt[20],tat[20],i, k, n, temp;
        float wtavg, tatavg;
        clrscr();
        printf("Enter the number of processes --- ");
        scanf("%d",&n);

        for(i=0;i<n;i++)
        {
                p[i] = i;
                printf("Enter the Burst Time & Priority of Process %d --- ",i);
                scanf("%d %d",&bt[i], &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;

                         }
        wtavg = wt[0] = 0;
        tatavg = tat[0] = bt[0];
```

```
        for(i=1;i<n;i++)
        {
                wt[i] = wt[i-1] + bt[i-1];
                tat[i] = tat[i-1] + bt[i];

                wtavg = wtavg + wt[i];
                tatavg = tatavg + tat[i];
        }

        printf("\nPROCESS\t\tPRIORITY\tBURST TIME\tWAITING TIME\tTURNAROUND  TIME");
        for(i=0;i<n;i++)
                printf("\n%d \t\t %d \t\t  %d \t\t %d \t\t %d ",p[i],pri[i],bt[i],wt[i],tat[i]);

        printf("\nAverage Waiting Time is --- %f",wtavg/n);
        printf("\nAverage Turnaround Time is --- %f",tatavg/n);
        getch();
}
```

INPUT

```
Enter the number of processes --    5
Enter the Burst Time & Priority of Process 0 ---  10     3
Enter the Burst Time & Priority of Process 1 ---  1      1
Enter the Burst Time & Priority of Process 2 ---  2      4
Enter the Burst Time & Priority of Process 3 ---  1      5
Enter the Burst Time & Priority of Process 4 ---  5      2
```

OUTPUT

PROCESS	PRIORITY	BURST TIME	WAITING TIME	TURNAROUND TIME
1	1	1	0	1
4	2	5	1	6
0	3	10	6	16
2	4	2	16	18
3	5	1	18	19

Average Waiting Time is --- 8.200000
Average Turnaround Time is --- 12.000000

EXPERIMENT 2

2.1 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;
}
```

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

```
**menu**
1.create dir
2.remove dir
3.read dir
Enter your choice 2
Enter dir name mkdir
Dir is removed
```

EXPERIMENT 3

1. OBJECTIVE

Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance.

2. DESCRIPTION

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a deadlock. Deadlock avoidance is one of the techniques for handling deadlocks. This approach requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this additional knowledge, it can decide for each request whether or not the process should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process. Banker's algorithm is a deadlock avoidance algorithm that is applicable to a system with multiple instances of each resource type.

3. PROGRAM

```
#include<stdio.h>
struct file
{
        int all[10];
        int max[10];
        int need[10];
        int flag;
};



void main()
{
        struct file f[10];
        int fl;
        int i, j, k, p, b, n, r, g, cnt=0, id, newr;
        int avail[10],seq[10];
        clrscr();
        printf("Enter number of processes -- ");
        scanf("%d",&n);
        printf("Enter number of resources -- ");
        scanf("%d",&r);
        for(i=0;i<n;i++)
        {
                printf("Enter details for P%d",i);
                printf("\nEnter allocation\t -- \t");
                for(j=0;j<r;j++)
                        scanf("%d",&f[i].all[j]);
                printf("Enter Max\t\t -- \t");
                for(j=0;j<r;j++)
                        scanf("%d",&f[i].max[j]);
                f[i].flag=0;
        }
        printf("\nEnter Available Resources\t -- \t");
        for(i=0;i<r;i++)
                scanf("%d",&avail[i]);

        printf("\nEnter New Request Details -- ");
        printf("\nEnter pid \t -- \t");
        scanf("%d",&id);
        printf("Enter Request for Resources \t -- \t");
```

```
    for(i=0;i<r;i++)
    {
            scanf("%d",&newr);
            f[id].all[i] += newr;



avail[i]=avail[i] - newr;
    }

    for(i=0;i<n;i++)
    {
            for(j=0;j<r;j++)
            {
                    f[i].need[j]=f[i].max[j]-f[i].all[j];
                    if(f[i].need[j]<0)
                            f[i].need[j]=0;
            }
    }
    cnt=0;
    fl=0;
    while(cnt!=n)
    {
            g=0;
            for(j=0;j<n;j++)
            {
                    if(f[j].flag==0)
                    {
                            b=0;
                            for(p=0;p<r;p++)
                            {
                                    if(avail[p]>=f[j].need[p])
                                            b=b+1;
                                    else
                                            b=b-1;
                            }
                            if(b==r)
                            {
                                    printf("\nP%d is visited",j);
                                    seq[fl++]=j;
                                    f[j].flag=1;
                                    for(k=0;k<r;k++)
                                            avail[k]=avail[k]+f[j].all[k];
                                    cnt=cnt+1;
                                    printf("(");
                                    for(k=0;k<r;k++)
                                            printf("%3d",avail[k]);
                                    printf(")");
                                    g=1;
                            }
                    }
            }
            if(g==0)
            {
                    printf("\n REQUEST NOT GRANTED -- DEADLOCK OCCURRED");
                    printf("\n SYSTEM IS IN UNSAFE STATE");
                    goto y;
            }
    }
    printf("\nSYSTEM IS IN SAFE STATE");
    printf("\nThe Safe Sequence is -- (");
```

```
                for(i=0;i<fl;i++)
                        printf("P%d ",seq[i]);
                printf(")");
              y: printf("\nProcess\t\tAllocation\t\tMax\t\t\tNeed\n");
                for(i=0;i<n;i++)
                {
                        printf("P%d\t",i);
                        for(j=0;j<r;j++)

printf("%6d",f[i].all[j]);for(j=0;j<r;j++)
                                printf("%6d",f[i].max[j]);
                        for(j=0;j<r;j++)
                                printf("%6d",f[i].need[j]);
                        printf("\n");
                }
                getch();
        }
```

INPUT

```
Enter number of processes       –       5
Enter number of resources       --      3
Enter details for P0
Enter allocation        --      0       1       0
Enter Max               --              7       5       3

Enter details for P1
Enter allocation        --      2       0       0
Enter Max               --      3       2       2

Enter details for P2
Enter allocation        --      3       0       2
Enter Max               --      9       0       2

Enter details for P3
Enter allocation        --      2       1       1
Enter Max               --      2       2       2

Enter details for P4
Enter allocation        --      0       0       2
Enter Max               --      4       3       3

Enter Available Resources  --   3   3       2
Enter New Request Details --
Enter pid       --      1
Enter Request for Resources     --      1       0       2
```

OUTPUT

```
P1 is visited(  5  3  2)
P3 is visited(  7  4  3)
P4 is visited(  7  4  5)
P0 is visited(  7  5  5)
P2 is visited( 10 5 7)
SYSTEM IS IN SAFE STATE
The Safe Sequence is -- (P1  P3  P4  P0  P2 )
```

Process	Allocation	Max	Need
P0	0 1 0	7 5 3	7 4 3
P1	3 0 2	3 2 2	0 2 0
P2	3 0 2	9 0 2	6 0 0
P3	2 1 1	2 2 2	0 1 1
P4	0 0 2	4 3 3	4 3 1

EXPERIMENT 4

4 OBJECTIVE

Write a C program to simulate producer-consumer problem using semaphores using UNIX/LINUX system calls

4.1 DESCRIPTION

Producer-consumer problem, is a common paradigm for cooperating processes. A producer process produces information that is consumed by a consumer process. One solution to the producer-consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, there must be available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and consumer processes. A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

4.1 PROGRAM

```
#include<stdio.h>
void main()
{
        int buffer[10], bufsize, in, out, produce, consume, choice=0;
        in = 0;
        out = 0;
        bufsize = 10;
        while(choice !=3)
        {
                printf("\n1. Produce \t 2. Consume \t3. Exit");
                printf("\nEnter your choice:  ");
                scanf("%d", &choice);
                switch(choice)    {
                        case 1:  if((in+1)%bufsize==out)
                                        printf("\nBuffer is Full");
                                else
                                {
                                        printf("\nEnter the value: ");
                                        scanf("%d", &produce);
                                        buffer[in] = produce;
                                        in = (in+1)%bufsize;
                                }
                                Break;
                        case 2:  if(in == out)
                                        printf("\nBuffer is Empty");
                                else
                                {
                                        consume = buffer[out];
                                        printf("\nThe consumed value is %d", consume);
                                        out = (out+1)%bufsize;
                                }
                                break;
}    }    }
```

OUTPUT

```
1. Produce      2. Consume      3. Exit
Enter your choice: 2
Buffer is Empty
1. Produce      2. Consume      3. Exit
Enter your choice: 1
Enter the value: 100
1. Produce      2. Consume      3. Exit
Enter your choice: 2
The consumed value is 100
1. Produce      2. Consume      3. Exit
Enter your choice
```

EXPERIMENT-5

Write a C program to illustrate the following IPC mechanisms
a) Pipes b) FIFOs c) Message Queues d) Shared Memory

OBJECTIVE:

Write a C program to illustrate the Pipes
IPC mechanism

PROGRAM

```
#include <stdio.h>
#include <unistd.h>
#define MSGSIZE 16
char* msg1 = "hello, world #1";
char* msg2 = "hello, world #2";
char* msg3 = "hello, world #3";

int main()
{
 char inbuf[MSGSIZE];
 int p[2], i;

 if (pipe(p) < 0)
 exit(1);

 /* continued */
 /* write pipe */

 write(p[1], msg1, MSGSIZE);
 write(p[1], msg2, MSGSIZE);
 write(p[1], msg3, MSGSIZE);

 for (i = 0; i < 3; i++) {
 /* read pipe */
 read(p[0], inbuf, MSGSIZE);
 printf("% s\n", inbuf);
 }
 return 0;
}
```

Output:

```
hello, world #1
hello, world #2
hello, world #3
```

b) Write a C program to illustrate the FIFO IPC Mechanism

PROGRAM

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

int main()
{
 int fd;

 // FIFO file path
 char * myfifo = "/tmp/myfifo";

 // Creating the named file(FIFO)
 // mkfifo(<pathname>, <permission>)
 mkfifo(myfifo, 0666);

 char arr1[80], arr2[80];
 while (1)
 {
 // Open FIFO for write only
 fd = open(myfifo, O_WRONLY);

 // Take an input arr2ing from user.
 // 80 is maximum length
 fgets(arr2, 80, stdin);

 // Write the input arr2ing on FIFO
 // and close it
 write(fd, arr2, strlen(arr2)+1);
 close(fd);

 // Open FIFO for Read only
 fd = open(myfifo, O_RDONLY);

 // Read from FIFO
 read(fd, arr1, sizeof(arr1));

 // Print the read message
 printf("User2: %s\n", arr1);

 close(fd);
 }
 return 0;
}
```

OUTPUT

```
Write data:hello
Data send is :hello
Data received is :hello
```

c) Write a C program to illustrate the Message Queue IPC mechanism

PROGRAM:

```
// 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 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);
 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;
}
```

OUTPUT

```
Write data:name9966
Data send is :name9966
```

```
// 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:

Data received is :name9966

d) Write a C program to illustrate the Shared Memory IPC mechanism

PROGRAM:

```
#include <iostream>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
using namespace std;

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

 // shmget returns an identifier in shmid
 int shmid =
shmget(key,1024,0666|IPC_CREAT);

 // shmat to attach to shared memory
 char *str = (char*)
shmat(shmid,(void*)0,0);

 cout<<"Write Data : ";
 gets(str);

 printf("Data written in memory:
%s\n",str);

 //detach from shared memory
 shmdt(str);

 return 0;
}
```

OUTPUT

Write data: hiiii
Data written in memory:hiiii

SHARED MEMORY FOR READER PROCESS

```
#include <iostream>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
using namespace std;

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

 // shmget returns an identifier in shmid
 int shmid =
shmget(key,1024,0666|IPC_CREAT);

 // shmat to attach to shared memory
 char *str = (char*)
shmat(shmid,(void*)0,0);

 printf("Data read from memory:
%s\n",str);

 //detach from shared memory
 shmdt(str);

 // destroy the shared memory
 shmctl(shmid,IPC_RMID,NULL);

 return 0;
}
```

OUTPUT

Data read from memory:hiii

EXPERIMENT 6

6.1 OBJECTIVE

6.2 Write C programs to simulate the following memory management techniques a) Paging b) Segmentation

6.3 DESCRIPTION:

6.4 A)paging

In computer operating systems, paging is one of the memory management schemes by which a computer stores and retrieves data from the secondary storage for use in main memory. In the paging memory-management scheme, the operating system retrieves data from secondary storage in same-size blocks called pages. Paging is a memory-management scheme that permits the physical address space a process to be noncontiguous. The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called frames and breaking logical memory into blocks of the same size called pages. When a process is to be executed, its pages are loaded into any available memory frames from their source.

6.5 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];

        clrscr();

        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
            {


            }
            getch();
}
```

INPUT

```
Enter the memory size – 1000
Enter the page size --        100
The no. of pages available in memory are----10
Enter number of processes --          3
Enter no. of pages required for p[1]--          4
Enter pagetable for p[1] ---          8          6          9          5

Enter no. of pages required for p[2]--          5
Enter pagetable for p[2] ---          1          4          5          7          3

Enter no. of pages required for p[3]--          5
```

OUTPUT

```
Memory is Full

Enter Logical Address to find Physical Address
Enter process no. and pagenumber and offset  --  2               3          60



The Physical Address is760
```

b) Segmentation

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 <stdlib.h>

// Global variables (Stored in Data Segment)
int globalVar = 10;
const int constGlobalVar = 20;

void functionExample() {
    // Local variables (Stored in Stack)
    int localVar = 30;
    printf("Address of localVar (Stack)        : %p\n", (void*)&localVar);
}

int main() {
    // Static variables (Stored in Data Segment)
    static int staticVar = 40;

    // Heap memory (Dynamically allocated, Stored in Heap)
    int *heapVar = (int *)malloc(sizeof(int));
    *heapVar = 50;

    printf("Address of globalVar (Data)       : %p\n", (void*)&globalVar);
    printf("Address of constGlobalVar (RO Data): %p\n", (void*)&constGlobalVar);
    printf("Address of staticVar (Data)       : %p\n", (void*)&staticVar);
    printf("Address of heapVar (Heap)         : %p\n", (void*)heapVar);

    functionExample();

    free(heapVar); // Free dynamically allocated memory
    return 0;
}
```

OUTPUT

```
Address of globalVar (Data)       : 0x601034
Address of constGlobalVar (RO Data): 0x400654
Address of staticVar (Data)       : 0x601038
Address of heapVar (Heap)         : 0x602000
Address of localVar (Stack)       : 0x7fffe52c
```

EXPERIMENT 7

7.1 OBJECTIVE

Write a C program to simulate page replacement algorithms
a) FIFO b) LRU c) Optimal

a) FIFO PROGRAM

```
#include <stdio.h>

int main() {
   int i, j, n, frames, pages[100], queue[100], front = 0, rear = 0, page_faults = 0, found;

   printf("Enter the number of pages: ");
   scanf("%d", &n);

   printf("Enter the reference string: ");
   for(i = 0; i < n; i++) {
      scanf("%d", &pages[i]);
   }

   printf("Enter the number of frames: ");
   scanf("%d", &frames);

   for(i = 0; i < frames; i++) {
      queue[i] = -1; // Initialize empty frames
   }

   printf("\nPage Replacement Process:\n");
   for(i = 0; i < n; i++) {
      found = 0;
      for(j = 0; j < frames; j++) {
         if(queue[j] == pages[i]) { // Check if page exists in frame
            found = 1;
            break;
         }
      }

      if(!found) { // Page fault occurs
         queue[rear] = pages[i];
         rear = (rear + 1) % frames; // Circular replacement
         page_faults++;
      }

      printf("Frame: ");
      for(j = 0; j < frames; j++) {
         if(queue[j] != -1)
            printf("%d ", queue[j]);
         else
            printf("- ");
      }
      printf("\n");
   }

   printf("\nTotal Page Faults: %d\n", page_faults);
   return 0;
}
```

OUTPUT
Enter the number of pages: 10
Enter the reference string: 7 0 1 2 3 0 4 2 3 0
Enter the number of frames: 3

Page Replacement Process:
Frame: 7 - -
Frame: 7 0 -
Frame: 7 0 1
Frame: 2 0 1
Frame: 2 3 1
Frame: 2 3 0
Frame: 4 3 0
Frame: 4 2 0
Frame: 4 2 3
Frame: 0 2 3

Total Page Faults: 10

[Process completed - press Enter]

b) LRU PAGE REPLACEMENT ALGORITHM

```
#include <stdio.h>

int findLRU(int time[], int frames) {
   int i, min = time[0], pos = 0;
   for(i = 1; i < frames; i++) {
      if(time[i] < min) {
         min = time[i];
         pos = i;
      }
   }
   return pos;
}

int main() {
   int i, j, n, frames, pages[100], frame[100], time[100], counter = 0, page_faults = 0, found, pos;

   printf("Enter the number of pages: ");
   scanf("%d", &n);

   printf("Enter the reference string: ");
   for(i = 0; i < n; i++) {
      scanf("%d", &pages[i]);
   }

   printf("Enter the number of frames: ");
   scanf("%d", &frames);

   for(i = 0; i < frames; i++) {
      frame[i] = -1;
   }

   printf("\nPage Replacement Process:\n");
   for(i = 0; i < n; i++) {
      found = 0;
      for(j = 0; j < frames; j++) {
         if(frame[j] == pages[i]) {
            found = 1;
            time[j] = counter++;
            break;
         }
      }

      if(!found) {
         if(i < frames) {
            pos = i;
         } else {
            pos = findLRU(time, frames);
         }
         frame[pos] = pages[i];
         time[pos] = counter++;
         page_faults++;
      }



  printf("Frame: ");
      for(j = 0; j < frames; j++) {
         if(frame[j] != -1)
```

```
    printf("%d ", frame[j]);
            else
                printf("- ");
        }
        printf("\n");
    }

    printf("\nTotal Page Faults: %d\n", page_faults);
    return 0;
}
```

INPUT

Enter the length of reference string -- 20
Enter the reference string -- 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
Enter the number of frames -- 3

OUTPUT

The Page Replacement process is --

7	-1	-1	PF No. -- 1
7	0	-1	PF No. -- 2
7	0	1	PF No. -- 3
2	0	1	PF No. -- 4
2	0	1	
2	0	3	PF No. -- 5
2	0	3	
4	0	3	PF No. -- 6
4	0	2	PF No. -- 7
4	3	2	PF No. -- 8
0	3	2	PF No. -- 9
0	3	2	
0	3	2	
1	3	2	PF No. -- 10
1	3	2	
1	0	2	PF No. -- 11
1	0	2	
1	0	7	PF No. -- 12
1	0	7	
1	0	7	

The number of page faults using LRU are 12

C) Optimal PROGRAM

PROGRAM

```
#include<stdio.h>
int n;
main()
{
        int seq[30],fr[5],pos[5],find,flag,max,i,j,m,k,t,s;
        int count=1,pf=0,p=0;
        float pfr;
        clrscr();
        printf("Enter maximum limit of the sequence: ");
        scanf("%d",&max);
        printf("\nEnter the sequence: ");
        for(i=0;i<max;i++)
                scanf("%d",&seq[i]);
                printf("\nEnter no. of frames: ");
                scanf("%d",&n);
                fr[0]=seq[0];
                pf++;
                printf("%d\t",fr[0]);

                i=1;
                while(count<n)
                {
                        flag=1;
                        p++;
                        for(j=0;j<i;j++)
                        {
                                if(seq[i]==seq[j])


                                flag=0;
                        }
                        if(flag!=0)
                        {
                                fr[count]=seq[i];
                                printf("%d\t",fr[count]);
                                count++;
                                pf++;
                        }
                        i++;
                }
printf("\n"); for(i=p;i<max;i++)
{
flag=1; for(j=0;j<n;j++)
{
if(seq[i]==fr[j])
flag=0;
}
if(flag!=0)


{
for(j=0;j<n;j++)
{
m=fr[j]; for(k=i;k<max;k++)
{
if(seq[k]==m)
{
```

```
pos[j]=k;break;

}
else
pos[j]=1;

}
}
for(k=0;k<n;k++)
{
if(pos[k]==1)
flag=0;
}
if(flag!=0)
s=findmax(pos);if(flag==0)
{
for(k=0;k<n;k++)
{
if(pos[k]==1)
{
s=k; break;
}
}
}
fr[s]=seq[i]; for(k=0;k<n;k++)
printf("%d\t",fr[k]);
pf++; printf("\n");
}
}
pfr=(float)pf/(float)max;
printf("\nThe no. of page faults are %d",pf);printf("\nPage fault rate %f",pfr);
getch();
}
int findmax(int a[])
{
int max,i,k=0; max=a[0]; for(i=0;i<n;i++)
{
if(max<a[i])
{
                                max=a[i];k=i;

                        }
                        }
                        return k;
}
```

INPUT

Enter number of page references -- 10

Enter the reference string -- 1 2 3 4 5 2 5 2 5 1 4 3
Enter the available no. of frames -- 3

OUTPUT
The Page Replacement Process is –

1	-1	-1	PF No. 1
1	2	-1	PF No. 2
1	2	3	PF No. 3
4	2	3	PF No. 4
5	2	3	PF No. 5
5	2	3	
5	2	3	
5	2	1	PF No. 6
5	2	4	PF No. 7
5	2	3	PF No. 8

Total number of page faults -- 8