# EXPERIMENT-1

1. Write C programs to simulate the following CPU scheduling algorithms:

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

## a) 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.

**FCFS CPU SCHEDULING ALGORITHM**

```
#include<stdio.h>
main()
{
int bt[20], wt[20], tat[20], i, n;
float wtavg, tatavg;
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

### b) 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

**SJF CPU SCHEDULING ALGORITHM**

```
#include<stdio.h>
main()
{
int p[20], bt[20], wt[20], tat[20], i, k, n, temp;
float wtavg, tatavg;
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);
}
```

**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

## c) 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.

**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;
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);
}
```

**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

## d) 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.

**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;
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]);
	}
```

**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 |

## EXPERIMENT-2

2. Write programs using the I/O system calls of UNIX/LINUX operating system (open, read, write, close, fcntl, seek, stat, opendir, readdir)
   **DESCRIPTION** The contents in the source file need to be copied into destination file using system calls. The system call open() is used to open source and destination files. The system call read( ) is used to read characters from the source file and write( ) system call is used to write the characters into the destination file.

**PROGRAM**

```
#include <syscall.h>

#include <unistd.h>

#include <sys/types.h>

#include <fcntl.h>

#include <sys/uio.h>

#include <sys/stat.h>

#include <stdio.h>

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

   {

      int fd;
```

```
fd=open(argv[1], O_CREAT | O_RDONLY);
if(fd== -1)
  {
     printf("error opening the file");
  }
void *buf = (char*) malloc(120);
int count=read(fd,buf,120);
printf("count : %d",count);
printf("%s",buf);
close(fd);
int f1;
f1=open(argv[2],O_CREAT | O_WRONLY);
if(f1==-1)
  {
     printf("error opening the file");
  }
int c;
while(count=read(fd,buf,120)>0)
  {
     c=write(f1,buf,120);
  }
```

```
 if(c==-1)
   {
        printf("error writing to the file");
   }
printf("\n Successfully copied the content from one file to other");
close(f1);
}
```

How to run it?
$gcc file.c
$./a.out

**OUTPUT:**

Successfully copied the content from one file to other

**b) Program to work with system calls related to directories**

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

int main (int argc, char *argv[])
{
    if(2 != argc)
    {
        printf("\n Please pass in the directory name \n");
        return 1;
    }

    DIR *dp = NULL;
    struct dirent *dptr = NULL;
    // Buffer for storing the directory path
    char buff[128];
    memset(buff,0,sizeof(buff));

    //copy the path set by the user
    strcpy(buff,argv[1]);

    // Open the directory stream
    if(NULL == (dp = opendir(argv[1])) )
    {
        printf("\n Cannot open Input directory [%s]\n",argv[1]);
        exit(1);
    }
    else
    {
```

```
        // Check if user supplied '/' at the end of directory name.
        // Based on it create a buffer containing path to new directory name
'newDir'
        if(buff[strlen(buff)-1]=='/')
        {
           strncpy(buff+strlen(buff),"newDir/",7);
        }
        else
        {
           strncpy(buff+strlen(buff),"/newDir/",8);
        }

        printf("\n Creating a new directory [%s]\n",buff);
        // create a new directory
        mkdir(buff,S_IRWXU|S_IRWXG|S_IRWXO);
        printf("\n The contents of directory [%s] are as follows
\n",argv[1]);
        // Read the directory contents
        while(NULL != (dptr = readdir(dp)) )
        {
           printf(" [%s] ",dptr->d_name);
        }
        // Close the directory stream
        closedir(dp);
        // Remove the new directory created by us
        rmdir(buff);
        printf("\n");
    }

    return 0;
}
```

**OUTPUT:**

```
# ./directory /home/cmr/practice/linux

 Creating a new directory [/home/cmr/practice/linux/newDir/]

 The contents of directory [/home/cmr/practice/linux] are as follows
 [redhat]  [newDir]  [linuxKernel]  [..]  [ubuntu]  [.]
```

## EXPERIMENT-3

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

**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.

```
#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];
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");
        }
}
```

***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. Write a C program to simulate producer-consumer problem using semaphores.

## 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.

## PROGRAM

```
#include<stdio.h>

int 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: 3

# EXPERIMENT- 5

## 5.1 OBJECTIVE

Write C programs to illustrate the following IPC mechanisms

a) Pipes b) FIFOs c) Message Queues d) Shared Memory

## 5.2 DESCRIPTION

Inter process communication (IPC) is a mechanism which allows processes to communicate each other and synchronize their actions. The communication between these processes can be seen as a method of co-operation between them. Processes can communicate with each other using these two ways:

i. Pipes
ii. FIFOs
iii. Shared Memory
iv. Message passing

Pipes are the oldest form of UNIX System IPC and are provided by all UNIX systems. A pipe is created by calling the pipe function.

```
#include <unistd.h>
int pipe(int fd[2]);            /* Returns: 0 if OK, −1 on error */
```

Two file descriptors are returned through the *fd* argument:

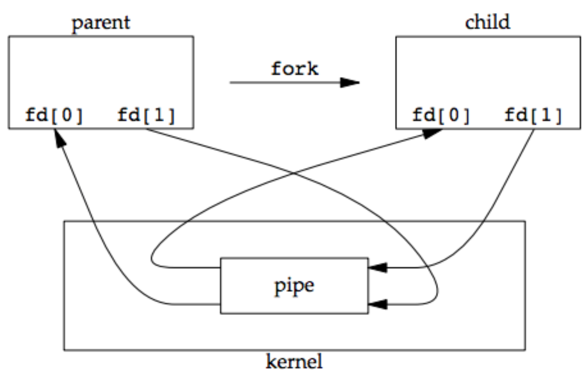
*fd[0]* is open for reading, and

*fd[1]* is open for writing.

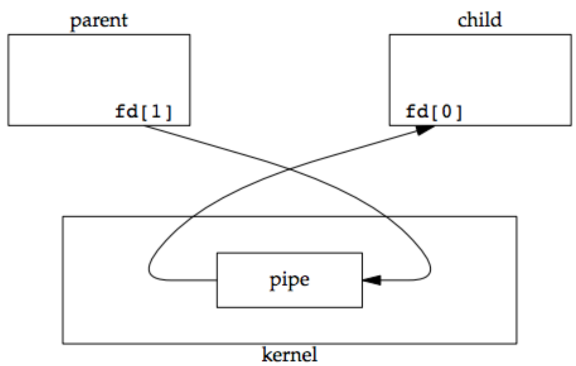
The output of *fd[1]* is the input for *fd[0]*.

POSIX.1 allows for implementations to support full-duplex pipes. For these implementations, *fd[0]* and *fd[1]* are open for both reading and writing.

The fstat function returns a file type of FIFO for the file descriptor of either end of a pipe. We can test for a pipe with the S_ISFIFO macro. The fstat function is applied to the file descriptor for the read end of the pipe, many systems store in st_size the number of bytes available for reading in the pipe, which is non portable. A pipe in a single process is next to useless. Normally, the process that calls pipe then calls fork, creating an IPC channel from the parent to the child, or vice versa. The following figure shows this scenario:




What happens after the fork depends on which direction of data flow we want. For a pipe from the parent to the child, the parent closes the read end of the pipe (*fd[0]*), and the child closes the write end (*fd[1]*). The following figure shows the resulting arrangement of descriptors.




For a pipe from the child to the parent, the parent closes fd[1], and the child closes fd[0].

Pipes are basically an IPC mechanism used for message passing between process in a system. They signify information flow between sender and reciever processes. The major differences between named and unnamed pipes are:-

- As suggested by their names, a named type has a specific name which can be given to it by the user. Named pipe if referred through this name only by the reader and writer. All instances of a named pipe share the same pipe name. On the other hand, unnamed pipes is not given a name. It is accessible through two file descriptors that are created through the function pipe(fd[2]), where fd[1] signifies the write file descriptor, and fd[0] describes the read file descriptor.
- An unnamed pipe is only used for communication between a child and it's parent process, while a named pipe can be used for communication between two unnamed process as well. Processes of different ancestry can share data through a named pipe.

**5.3 PROGRAM**

**a. IPC using pipes**

```
#include<stdio.h>
#include<fcntl.h>
#include<stdlib.h>
main()
{
  int file1,file2;
  int fd;
  char str[256];
  char temp[4]="how";
  char temp1[4];

  file1 = mkfifo("fifo_server",0666);
  if(file1<0)
   {
     printf("Unable to create a fifo");
     exit(-1);
   }
```

```
  file2 = mkfifo("fifo_client",0666);
  if(file2<0)
   {
     printf("Unable to create a fifo");
     exit(-1);
   }
  printf("fifos server and child created successfuly");
}
```

**Compile and run it . Next, open new terminal and create server.c The code of server.c is given below**

```
#include<stdio.h>
#include<fcntl.h>
main()
{
  FILE *file1;
  int fifo_server,fifo_client;
  int choice;
  char *buf;
  fifo_server = open("fifo_server",O_RDWR);
  if(fifo_server<1)
```

```
 {
   printf("Error opening file");
 }
read(fifo_server,&choice,sizeof(int));
sleep(10);
fifo_client = open("fifo_client",O_RDWR);

if(fifo_server<1)
 {
   printf("Error opening file");
 }

switch(choice)
 {
  case 1:  buf="CMRTC";
           write(fifo_client,buf,10*sizeof(char));
           printf("\n Data sent to client \n");
           break;
  case 2:  buf="HYDERABAD";
           write(fifo_client,buf,10*sizeof(char));
           printf("\nData sent to client\n");
```

```
            break;
    case 3:  buf="B.Tech CSE";
            write(fifo_client,buf,10*sizeof(char));
            printf("\nData sent to client\n");
  }
 close(fifo_server);
 close(fifo_client);
}
```

**Compile and run it . Next, open new terminal and create client.c The code of client.c is given below**

```
#include<stdio.h>
#include<fcntl.h>
#include<stdlib.h>
main()
{
 FILE *file1;
 int fifo_server,fifo_client;
 char str[256];
 char *buf;
 int choice=1;
 printf("Choose the request to be sent to server from options below");
```

```
 printf("\n\t\t Enter 1 for College Name \n \
               Enter 2 for Location \n \
               Enter 3 for Course \n");
 scanf("%d",&choice);
fifo_server=open("fifo_server",O_RDWR);
 if(fifo_server < 0)
  {
    printf("Error in opening file");
    exit(-1);
  }
 write(fifo_server,&choice,sizeof(int));
 fifo_client=open("fifo_client",O_RDWR);
  if(fifo_client < 0)
  {
    printf("Error in opening file");
  exit(-1);
  }
 buf=malloc(10*sizeof(char));
 read (fifo_client,buf,10*sizeof(char));
 printf("\n ***Reply from server is %s***\n",buf);
 close(fifo_server);
```

close(fifo_client);

}

**Now compile it and run it.**

**Input in Terminal of client:**

Choose the request to be sent to server from options below

Enter 1 for College Name

Enter 2 for Location

Enter 3 for Course

1

**Output in Server Terminal:**

Data sent to client

**Output in client's Terminal:**

CMRTC

***RESULT:***

Thus the program is executed successfully.

**b) IPC using FIFOs**

**i) Writer 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;
```

```
}
```

**ii Reader Program**

```
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
 int main()
{
   int fd1;

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

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

   char str1[80], str2[80];
   while (1)
```

```
    {
        // First open in read only and read
        fd1 = open(myfifo,O_RDONLY);
        read(fd1, str1, 80);

        // Print the read string and close
        printf("User1: %s\n", str1);
        close(fd1);

        // Now open in write mode and write
        // string taken from user.
        fd1 = open(myfifo,O_WRONLY);
        fgets(str2, 80, stdin);
        write(fd1, str2, strlen(str2)+1);
        close(fd1);
    }
    return 0;
}
```

**Note:** Execute the two programs simultaneously

**OUTPUT:**

User1 :   CMRTC

          HYDERABAD

***RESULT:***

Thus the program is executed successfully.

**C) IPC using Message Queues**

**i. MESSAGE QUEUE FOR 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;
}
```

**ii. MESSAGE QUEUE FOR 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;
}
```

**Note:** Execute the two programs simultaneously

**OUTPUT:**

*Writer side process:*

Write data : CMRTC HYDERABAD

Data send is : CMRTC HYDERABAD

Reader side Process:

Data received is : CMRTC HYDERABAD

***RESULT:***

Thus the program is executed successfully.

**d. IPC through shared memory**

i. SHARED MEMORY FOR WRITER PROCESS

```
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
 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;
}
```

ii. SHARED MEMORY FOR READER PROCESS

```
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
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;
}
```

**Note:** Execute the two programs simultaneously

**OUTPUT:**

*Writer side process:*

Write data : CMRTC HYDERABAD

Data written in memory : CMRTC HYDERABAD

*Reader side Process:*

Data read from memory : CMRTC HYDERABAD

***RESULT:***

Thus the program is executed successfully.

# EXPERIMENT-6

Write C programs to simulate the following memory management techniques

a) Paging b) Segmentation

## 6.1 OBJECTIVE

Write a C program to simulate paging technique of memory management.

## 6.2 DESCRIPTION

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.

**6.3.1 PROGRAM**

```
#include<stdio.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);
  }
}
```

**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 page table for p[1] --- 8 6 9 5

Enter no. of pages required for p[2] -- 5

Enter page table 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 is – 760

***RESULT:***

Thus the program is executed successfully.

## 6.3.2 PROGRAM: Segmentation:

```
#include<stdio.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,limit,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