Laboratory Manual

Operating System
(CS-405)

For

Second Year Students CSE
Department: Computer Science &Engineering

INSTITUTE OF TECHNOLOGY & MANAGEMENT
GWALIOR • MP • INDIA

Department of Computer Science and Engineering

Vision of CSE Department:

The department envisions to nurture students to become technologically proficient, research competent and socially accountable for the welfare of the society.

Mission of the CSE Department:

I. To provide high quality education through effective teaching-learning process emphasizing active participation of students.

II. To build scientifically strong engineers to cater to the needs of industry, higher studies, research and startups.

III. To awaken young minds ingrained with ethical values and professional behaviors for the betterment of the society.

Programme Educational Objectives:

Graduates will be able to

I. Our engineers will demonstrate application of comprehensive technical knowledge for innovation and entrepreneurship.

II. Our graduates will employ capabilities of solving complex engineering problems to succeed in research and/or higher studies.

III. Our graduates will exhibit team-work and leadership qualities to meet stakeholder business objectives in their careers.

IV. Our graduates will evolve in ethical and professional practices and enhance socioeconomic contributions to the society.

INSTITUTE OF TECHNOLOGY
& MANAGEMENT
GWALIOR • MP • INDIA

Program Outcomes (POs):

Engineering Graduates will be able to:

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

INSTITUTE OF TECHNOLOGY & MANAGEMENT
GWALIOR • MP • INDIA

Course Outcomes

Operating System(CS 405)

CO1 :	To Understand and apply the basic knowledge of operating systems like kernel, shell, and types of Operating systems.
CO2 :	To analyse various synchronisation algorithm & Process scheduling algorithms (FCFS, SJF, RR, and SRTF) on the basis on Turnaround time and waiting time.
CO3 :	To Apply page replacement algorithms like(LRU,FIFO,Optimal) to resolve the issues in virtual memory,and understand various memory management techniques.
CO4 :	Design the concept of disk management and analyse different disk scheduling algorithms (FCFS, SSTF, SCAN etc.) for better utilization of external memory and apply file management operations.
CO5 :	Installation and Evaluation of the various features of different OS like UNIX, Linux, windows, android,ubuntu etc.

Course	Course Outcomes	CO Attainment	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2	PSO3
CO1	To Understand and apply the basic knowledge of operating systems like kernel, shell, and types of Operating systems.		2	1											2		
CO2	To analyse various synchronisation algorithm & Process scheduling algorithms (FCFS, SJF, RR, and SRTF) on the basis on Turnaround time and waiting time.		2	2	1	1	1				1	1			2	1	
CO3	To Apply page replacement algorithms like(LRU,FIFO,Optimal) to resolve the issues in virtual memory,and understand various memory management techniques.		2	2	1	1	1				1	1			2	1	
CO4	Design the concept of disk management and analyse different disk scheduling algorithms (FCFS, SSTF, SCAN etc.) for better utilization of external memory and apply file management operations.		2	1		1	1					1			1		
CO5	Installation and Evaluation of the various features of different OS like UNIX, Linux, windows, android,ubuntu etc.		2	1	1	1	1				1	1			2		

INSTITUTE OF TECHNOLOGY & MANAGEMENT
GWALIOR • MP • INDIA

LIST OF PROGRAMS

Program-1

Write a C program to simulate the CPU scheduling algorithm First Come First Serve (FCFS)

DESCRIPTION

To calculate the average waiting time using the FCFS algorithm first the waiting time of the first process is kept zero and the waiting time of the second process is the burst time of the first process and the waiting time of the third process is the sum of the burst times of the first and the second process and so on. After calculating all the waiting times the average waiting time is calculated as the average of all the waiting times. FCFS mainly says first come first serve the algorithm which came first will be served first.

ALGORITHM

Step 1: Start the process
Step 2: Accept the number of processes in the ready Queue
Step 3: For each process in the ready Q, assign the process name and the burst time
Step 4: Set the waiting of the first process as 0‘and its burst time as its turnaround time
Step 5: for each process in the Ready Q calculate
a).Waiting time (n) = waiting time (n-1) + Burst time (n-1)
b). Turnaround time (n)= waiting time(n)+Burst time(n)
Step 6: Calculate
a)Average waiting time = Total waiting Time / Number of process
b)Average Turnaround time= Total Turnaround Time/Number of process

Step 7: Stop the process

PROGRAM

```
#include<stdio.h>
#include<conio.h>
void main()
{
   int n,bt[20],wt[20],tat[20],avwt=0,avtat=0,i,j;
   clrscr();
   printf("Enter total number of processes:");
   scanf("%d",&n);     printf("\nEnter Process Burst Time\n");
  for(i=0;i<n;i++)
   {
      printf("P[%d]:",i+1);         scanf("%d",&bt[i]);
   }
   wt[0]=0;     for(i=1;i<n;i++)
{         wt[i]=0;
for(j=0;j<i;j++)
wt[i]+=bt[j];
   }
   printf("\nProcess\t\tBurst Time\tWaiting Time\tTurnaround Time");
   for(i=0;i<n;i++)
   {
      tat[i]=bt[i]+wt[i];
     avwt+=wt[i];
    avtat+=tat[i];
     printf("\nP[%d]\t\t%d\t\t%d\t\t%d",i+1,bt[i],wt[i],tat[i]);
   }
   avwt/=i;     avtat/=i;
   printf("\n\nAverage Waiting Time:%d",avwt);
   printf("\nAverage Turnaround Time:%d",avtat);
   getch();
}
```

Program-2

Write a C program to simulate the CPU scheduling algorithm Shortest Job First (SJF)

DESCRIPTION

To calculate the average waiting time in the shortest job first algorithm the sorting of the process based on their burst time in ascending order then calculate the waiting time of each process as the sum of the bursting times of all the process previous or before to that process.

ALGORITHM

Step 1: Start the process
Step 2: Accept the number of processes in the ready Queue
Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time
Step 4: Start the Ready Q according the shortest Burst time by sorting according to lowest to highest burst time.
Step 5: Set the waiting time of the first process as ‗0' and its turnaround time as its burst time.
Step 6: Sort the processes names based on their Burt time
Step 7: For each process in the ready queue, calculate
a)Waiting time(n)= waiting time (n-1) + Burst time(n-1)
b)Turnaround time (n)= waiting time(n)+Burst time(n)
Step 8: Calculate
c) Average waiting time = Total waiting Time / Number of process

d) Average Turnaround time=Total Turnaround Time/Number of process

Step 9: Stop the process

PROGRAM

```
#include<stdio.h>
#include<conio.h>
void main()
{
 int i,n,p[10],min,k=1,btime=0;
int bt[10],temp,j,arr[10],wait[10],turn[10],ta=0,sum=0;
float wavg=0,tavg=0,tsum=0,wsum=0;

clrscr();
 printf("\nEnter the No. of processe:");
scanf("%d",&n);
printf("Enter the arrival time and burst time of the processes:\n");
for(i=0;i<n;i++)
scanf(" %d %d",&arr[i],&bt[i]);
for(i=0;i<n;i++)
{
 for(j=0;j<n;j++)
{
if(arr[i]<arr[j])
{
 temp=p[j];
p[j]=p[i]; p[i]=temp;
temp=arr[j];
arr[j]=arr[i];
arr[i]=temp; temp=bt[j];
bt[j]=bt[i];
bt[i]=temp; }
 }
}
for(j=0;j<n;j++)
 {
btime=btime+bt[j];
min=bt[k];
for(i=k;i<n;i++)
{
if (btime>=arr[i] && bt[i]<min)
 {
temp=p[k];
p[k]=p[i]; p[i]=temp;
temp=arr[k];
arr[k]=arr[i]; arr[i]=temp;
temp=bt[k];
bt[k]=bt[i];
bt[i]=temp; } }
```

```
k++; } wait[0]=0;
for(i=1;i<n;i++) {
sum=sum+bt[i-1];
wait[i]=sum-arr[i];
wsum=wsum+wait[i]; }
wavg=(wsum/n);
for(i=0;i<n;i++)
{ ta=ta+bt[i];
turn[i]=ta-arr[i];
tsum=tsum+turn[i];
} tavg=(tsum/n); printf("**************************");
printf("\n RESULT:-");
printf("\nProcess\t Burst\t Arrival\t Waiting\t Turn-around" );
for(i=0;i<n;i++)
printf("\n p%d\t %d\t %d\t\t %d\t\t\t%d",p[i],bt[i],arr[i],wait[i],turn[i]);
printf("\n\nAVERAGE WAITING TIME : %f",wavg);
printf("\nAVERAGE TURN AROUND TIME : %f",tavg);
getch();
}
```

Program-3

Write a C program to simulate the CPU scheduling algorithm Round Robin(RR) Scheduling

DESCRIPTION

To aim is to calculate the average waiting time. There will be a time slice, each process should be executed within that time-slice and if not it will go to the waiting state so first check whether the burst time is less than the timeslice. If it is less than it assign the waiting time to the sum of the total times. Ifit is greater than the burst-time then subtract the time slot from the actual burst time and increment it by time-slot and the loop continues until all the processes are completed.

ALGORITHM

Step 1: Start the process
Step 2: Accept the number of processes in the ready Queue and time quantum (or) time slice
Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time
Step 4: Calculate the no. of time slices for each process where No. of time slice for process (n) = burst time process (n)/time slice
Step 5: If the burst time is less than the time slice then the no. of time slices =1.
Step 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)Turnaround time for process(n) = waiting time of process(n) + burst time of process(n)+ the time difference in getting CPU from process(n).
Step 7: Calculate
c) Average waiting time = Total waiting Time / Number of process

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

PROGRAM

```
#include<stdio.h>
#include<conio.h>
void main()
{
 int i, NOP, sum=0,count=0, y, quant, wt=0, tat=0, at[10], bt[10], temp[10];
 float avg_wt, avg_tat;
 clrscr();
printf(" Total number of process in the system: ");
    scanf("%d", &NOP);
        y = NOP; // Assign the number of process to variable y
 for(i=0; i<NOP; i++)

 {
 printf("\n Enter the Arrival and Burst time of the Process[%d]\n", i+1);
 printf(" Arrival time is: \t");  // Accept arrival time
  scanf("%d", &at[i]);
printf(" \nBurst time is: \t"); // Accept the Burst time
 scanf("%d", &bt[i]);
 temp[i] = bt[i]; // store the burst time in temp array
  }
  printf("Enter the Time Quantum for the process: \t");
scanf("%d", &quant);
  // Display the process No, burst time, Turn Around Time and the waiting time
  printf("\n Process No \t\t Burst Time \t\t TAT \t\t Waiting Time ");
for(sum=0, i = 0; y!=0; )
  {
  if(temp[i] <= quant && temp[i] > 0) // define the conditions
  {
   sum = sum + temp[i];
   temp[i] = 0;
            count=1;
   }
     else if(temp[i] > 0)
   {
      temp[i] = temp[i] - quant;
      sum = sum + quant;
   }
      if(temp[i]==0 && count==1)
   {
        y--; //decrement the process no.
  printf("\nProcess No[%d] \t\t %d\t\t\t\t %d\t\t\t %d", i+1, bt[i], sumat[i], sum-at[i]-
bt[i]);
      wt = wt+sum-at[i]-bt[i];
tat = tat+sum-at[i];
      count =0;
```

```
    }
     if(i==NOP-1)
    {
        i=0;
    }
      else if(at[i+1]<=sum)
    {
        i++;
    }
    else
    {
        i=0;
    }
  }
   avg_wt = wt * 1.0/NOP;     avg_tat = tat * 1.0/NOP;
 printf("\n Average Turn Around Time: \t%f", avg_wt);
printf("\n Average Waiting Time: \t%f", avg_tat);
 getch();
   }
```

Program-4

Write a C program to simulate the CPU Priority Scheduling Algorithm

DESCRIPTION

To calculate the average waiting time in the priority algorithm, sort the burst times according to their priorities and then calculate the average waiting time of the processes. The waiting time of each process is obtained by summing up the burst times of all the previous processes

ALGORITHM

Step 1: Start the process
Step 2: Accept the number of processes in the ready Queue
Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time
Step 4: Sort the ready queue according to the priority number.
Step 5: Set the waiting of the first process as 0 and its burst time as its turnaround time
Step 6: Arrange the processes based on process priority
Step 7: For each process in the Ready Q calculate
Step 8: for each process in the Ready Q calculate
a)Waiting time(n)= waiting time (n-1) + Burst time(n-1)
b)Turnaround time (n)= waiting time(n)+Burst time(n)
Step 9: Calculate
b) Average waiting time = Total waiting Time / Number of process

c) Average Turnaround time=Total Turnaround Time/Number of process Print the results in an order.

Step10: Stop

PROGRAM

```
#include<stdio.h>
#include<conio.h>
 void main()
{
   int bt[20],p[20],wt[20],tat[20],pr[20],i,j,n,total=0,pos,temp,avg_wt,avg_tat;
   clrscr();
   printf("Enter Total Number of Process:");
scanf("%d",&n);
 printf("\nEnter Burst Time and Priority\n");
for(i=0;i<n;i++)
   {
         printf("\nP[%d]\n",i+1);
printf("Burst Time:");
scanf("%d",&bt[i]);
printf("Priority:");
scanf("%d",&pr[i]);
          p[i]=i+1;
            }
   for(i=0;i<n;i++)
   {
          pos=i;
           for(j=i+1;j<n;j++)
          {
               if(pr[j]<pr[pos])
                    pos=j;
          }
          temp=pr[i];
pr[i]=pr[pos];
pr[pos]=temp;
temp=bt[i];
bt[i]=bt[pos];
bt[pos]=temp;
temp=p[i];
p[i]=p[pos];
p[pos]=temp;
  }
   wt[0]=0;
 for(i=1;i<n;i++)
   {
          wt[i]=0;
          for(j=0;j<i;j++)
wt[i]+=bt[j];
total+=wt[i];
   }
   avg_wt=total/n;
```

```
   total=0;
   printf("\nProcess\t    Burst Time    \tWaiting Time\tTurnaround Time");
   for(i=0;i<n;i++)
   {
         tat[i]=bt[i]+wt[i];
total+=tat[i];
               printf("\nP[%d]\t\t  %d\t\t    %d\t\t\t%d",p[i],bt[i],wt[i],tat[i]);
   }
   avg_tat=total/n;
   printf("\n\nAverage Waiting Time=%d",avg_wt);
   printf("\nAverage Turnaround Time=%d\n",avg_tat);
         getch();
}
```

Program-5

Write a C program to simulate the Page Replacement Algorithms

DESCRIPTION

Page replacement algorithms are an important part of virtual memory management and it helps the OS to decide which memory page can be moved out making space for the currently needed page. However, the ultimate objective of all page replacement algorithms is to reduce the number of page faults.

A) FIFO (First In First Out)

This is the simplest page replacement algorithm. In this algorithm, the operating system keeps track of all pages in the memory in a queue, the oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

ALGORITHM

1. Start the process
2. Read number of pages n
3. Read number of pages no
4. Read page numbers into an array a[i]
5. Initialize avail[i]=0 .to check page hit
6. Replace the page with circular queue, while re-placing check page availability in the frame Place avail[i]=1 if page is placed in the frame Count page faults
7. Print the results.
8. Stop the process

PROGRAM

```
#include<stdio.h>
void main()
{
int i,j,n,a[50],frame[10],no,k,avail,count=0;
printf("\n enter the length of the Reference string:\n");
scanf("%d",&n);
printf("\n enter the reference string:\n");
for(i=1;i<=n;i++)
scanf("%d",&a[i]);
 printf("\n enter the number of Frames:");
scanf("%d",&no);
 for(i=0;i<no;i++)
{
 frame[i]= -1; } j=0;
 printf("ref string\t            page frames\n");
for(i=1;i<=n;i++)
{
printf("%d\t\t",a[i]);
avail=0;
for(k=0;k<no;k++)
if(frame[k]==a[i])
{ avail=1; } if (avail==0) {
frame[j]=a[i]; j=(j+1)%no;
count++;
for(k=0;k<no;k++)
printf("%d\t",frame[k]);
  printf("\n\n");
}
printf("Page Fault Is %d",count);
getch();
}
```

B) L R U (Least Recently Used)

In this algorithm page will be replaced which is least recently used

ALGORITHM

1. Start the process
2. Declare the size
3. Get the number of pages to be inserted
4. Get the value
5. Declare counter and stack
6. Select the least recently used page by counter value
7. Stack them according the selection.
8. Display the values
9. Stop the process

PROGRAM

```
#include<stdio.h> #include<conio.h>
int fr[3];

void main() { void display(); int
p[12]={2,3,2,1,5,2,4,5,3,2,5,2},i,j,fs[3]; int
index,k,l,flag1=0,flag2=0,pf=0,frsize=3;

clrscr();

for(i=0;i<3;i++) {
fr[i]=-1; }
for(j=0;j<12;j++) {
flag1=0,flag2=0;
for(i=0;i<3;i++) {
if(fr[i]==p[j]) {
flag1=1;  flag2=1;
break; } }
if(flag1==0) {
for(i=0;i<3;i++)
{ if(fr[i]==-1) { fr[i]=p[j];
flag2=1; break;
} } } if(flag2==0) { for(i=0;i<3;i++)
fs[i]=0; for(k=j-1,l=1;l<=frsize-
1;l++,k--)
{ for(i=0;i<3;i++) {
if(fr[i]==p[k]) fs[i]=1;
} } for(i=0;i<3;i++) {
if(fs[i]==0) index=i; }
```

```
fr[index]=p[j];  pf++; }
display(); } printf("\n
no of page faults
:%d",pf+frsize);
getch(); } void
display()
{ int i; printf("\n");
for(i=0;i<3;i++)  printf("\t%d",fr[i]);
}
```

C) Optimal

In this algorithm, pages are replaced which would not be used for the longest duration of time in the future. This algorithm will give us less page fault when compared to other page replacement algorithms

ALGORITHM

1. Start Program
2. Read Number of Pages and Frames
3. Read Each Page Value
4. Search for Page in the Frames
5. If Not Available Allocate Free Frame

6 .If No Frames Is Free Replace the Page with the Page That Is Lastly Used

7. Print Page Number of Page Faults
8. Stop process.

PROGRAM

```
#include<stdio.h>
 #include<conio.h>
int fr[3], n, m; void
display();  void main()
{ int i,j,page[20],fs[10];
int max,found=0,lg[3],index,k,l,flag1=0,flag2=0,pf=0;

float pr; clrscr(); printf("Enter length of the reference string: ");

scanf("%d",&n);

printf("Enter the reference string: ");
for(i=0;i<n;i++)  scanf("%d",&page[i]);
printf("Enter no of frames: ");
scanf("%d",&m);  for(i=0;i<m;i++)
fr[i]=-1;

pf=m;
for(j=0;j<n;j++)
 { flag1=0;
flag2=0;
for(i=0;i<m;i++)
{ if(fr[i]==page[j])
{ flag1=1;
flag2=1;
break;
}
```

```
} if(flag1==0) {
for(i=0;i<m;i++) {
if(fr[i]==-1) {
fr[i]=page[j];
flag2=1;  break; }
} } if(flag2==0) {
for(i=0;i<m;i++)
lg[i]=0;

for(i=0;i<m;i++) {
for(k=j+1;k<=n;k++) {

if(fr[i]==page[k]) {
lg[i]=k-j; break;
} }
}

found=0;
for(i=0;i<m;i++) {
if(lg[i]==0) {
index=i;  found =1;
break; } }
if(found==0) {
max=lg[0];  index=0;
for(i=0;i<m;i++) {

if(max<lg[i]) {
max=lg[i];
index=i; }
} } fr[index]=page[j];  pf++; } display(); } printf("Number of page faults :%d\n", pf);
pr=(float)pf/n*100; printf("Page fault rate = %f \n", pr);  getch(); } void display() {
int i;
for(i=0;i<m;i++)

printf("%d\t",fr[i]);
printf("\n");
 }
```

Program-6

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

DESCRIPTION

Producer consumer problem is a synchronization problem. There is a fixed size buffer where the producer produces items and 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. 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>
#include<stdlib.h>
int mutex = 1; int full
= 0;
int empty = 10, x = 0; void
produce()
{
      --mutex;
     ++full;          --
empty;
      x++;
          printf("\nProducer produces""item %d",x);
      ++mutex;
}
void consume()
{
      --mutex;
     --full;   ++empty;
     printf("\nConsumer consumes ""item %d",x);          x--;
      ++mutex;
} int main() {        int n, i;
     printf("\n1. Press 1 for Producer"
                  "\n2. Press 2 for Consumer"
                  "\n3. Press 3 for Exit");
       for (i = 1; i > 0; i++)
     {
                   printf("\nEnter your choice:");
               scanf("%d", &n);
     switch (n)
               {
```

```
                case 1:
                 if ((mutex == 1)
                 && (empty != 0))
                {
                 produce();
                }
                else
                {
                   printf("Buffer is full!");
                }
                break;
        case 2:
                if ((mutex == 1)
        && (full != 0))

{
consume();
} else  {
printf("Buffer is empty!");
} break; case 3: exit(0);
break;
     }
     }
     }
```

Program 7

To Write a C program to simulate Reader writer problem

DESCRIPTION

There is a shared resource which should be accessed by multiple processes.There are two types of processes in this context.They are reader and writer.Any number of readers can read from the shared resource simultaneously, but only one writer can write to the shared resource.When a writer is writing data to the resource, no other process can access the resource.A writer cannot write to the resource if there are non zero number of readers accessing the resource at that time.

PROGRAM

```
#include<stdio.h>
#include<pthread.h>
#include<semaphore.h>

sem_t mutex;
sem_t db;
int readercount=0;
pthread_t reader1,reader2,writer1,writer2;
void *reader(void *);
void *writer(void *);
main()
{
sem_init(&mutex,0,1);
sem_init(&db,0,1);
while(1)
{
pthread_create(&reader1,NULL,reader,"1");
pthread_create(&reader2,NULL,reader,"2");
pthread_create(&writer1,NULL,writer,"1");
pthread_create(&writer2,NULL,writer,"2");
}
}
void *reader(void *p)
{
printf("prevoius value %dn",mutex);
sem_wait(&mutex);
printf("Mutex acquired by reader %dn",mutex);
readercount++;
if(readercount==1) sem_wait(&db);
```

```
sem_post(&mutex);
printf("Mutex returned by reader %dn",mutex);
printf("Reader %s is Readingn",p);
//sleep(3);
sem_wait(&mutex);
printf("Reader %s Completed Readingn",p);
readercount--;
if(readercount==0) sem_post(&db);
sem_post(&mutex);
}

void *writer(void *p)
{
printf("Writer is Waiting n");
sem_wait(&db);
printf("Writer %s is writingn ",p);
sem_post(&db);
//sleep(2);
}
```

Program-8

To Write a C program to implement Dining philosopher problem

DESCRIPTION

There are some Philosophers whose work is just thinking and eating. Let there are 5 (for example) philosophers. They sat at a round table for dinner. To complete dinner each must need two Forks (spoons). But there are only 5 Forks available (Forks always equal to no. of Philosophers) on table. They take in such a manner that, first take left Fork and next right Fork. But problem is they try to take at same time. Since they are trying at same time, Fork 1, 2, 3, 4, 5 taken by Philosopher 1, 2, 3, 4, 5 respectively (since they are left side of each). And each one tries to ta ke right side Fork. But no one found available Fork. And also that each one thinks that someone will release the Fork and then I can eat. This continuous waiting leads to Dead Lock situation.

PROGRAM

```
#include<stdio.h>
#include<semaphore.h>
#include<pthread.h>

#define N 5
#define THINKING 0
#define HUNGRY 1
#define EATING 2
#define LEFT (ph_num+4)%N
#define RIGHT (ph_num+1)%N

sem_t mutex;
sem_t S[N];

void * philospher(void *num);
void take_fork(int);
void put_fork(int);
void test(int);

int state[N];
int phil_num[N]={0,1,2,3,4};

int main()
{
   int i;
   pthread_t thread_id[N];
   sem_init(&mutex,0,1);
   for(i=0;i<N;i++)
```

```
        sem_init(&S[i],0,0);
    for(i=0;i<N;i++)
    {
        pthread_create(&thread_id[i],NULL,philospher,&phil_num[i]);
        printf("Philosopher %d is thinkingn",i+1);
    }
    for(i=0;i<N;i++)
        pthread_join(thread_id[i],NULL);
}

void *philospher(void *num)
{
    while(1)
    {
        int *i = num;
        sleep(1);
        take_fork(*i);
        sleep(0);
        put_fork(*i);
    }
}

void take_fork(int ph_num)
{
    sem_wait(&mutex);
    state[ph_num] = HUNGRY;
    printf("Philosopher %d is Hungryn",ph_num+1);
    test(ph_num);
    sem_post(&mutex);
    sem_wait(&S[ph_num]);
    sleep(1);
}

void test(int ph_num)
{
    if (state[ph_num] == HUNGRY && state[LEFT] != EATING && state[RIGHT] !=
EATING)
    {
        state[ph_num] = EATING;
        sleep(2);
        printf("Philosopher %d takes fork %d and %dn",ph_num+1,LEFT+1,ph_num+1);
        printf("Philosopher %d is Eatingn",ph_num+1);
        sem_post(&S[ph_num]);
    }
}
```

```
void put_fork(int ph_num)
{
   sem_wait(&mutex);
   state[ph_num] = THINKING;
   printf("Philosopher %d putting fork %d and %d downn",ph_num+1,LEFT+1,ph_num+1);
   printf("Philosopher %d is thinkingn",ph_num+1);
   test(LEFT);
   test(RIGHT);
   sem_post(&mutex);
}
```

Program-9

Write a program to implementation and compare various page replacement algorithm.

Algorithm for FIFO Page Replacement

- Step 1. Start to traverse the pages.
- Step 2. If the memory holds fewer pages, then the capacity else goes to step 5.
- Step 3. Push pages in the queue one at a time until the queue reaches its maximum capacity or all page requests are fulfilled.
- Step 4. If the current page is present in the memory, do nothing.
- Step 5. Else, pop the topmost page from the queue as it was inserted first.
- Step 6. Replace the topmost page with the current page from the string.
- Step 7. Increment the page faults.
- Step 8. Stop

PROGRAM

```
#include <stdio.h>
int main()
{
    int referenceString[10], pageFaults = 0, m, n, s, pages, frames;
    printf("\nEnter the number of Pages:\t");
    scanf("%d", &pages);
    printf("\nEnter reference string values:\n");
    int(m = 0; m < pages; m++)
    {
        printf("Value No. [%d]:\t", m + 1);
        scanf("%d", &referenceString[m]);
    }
    printf("\n What are the total number of frames:\t");
    {
        scanf("%d", &frames);
    }
    inttemp[frames];
    for(m = 0; m < frames; m++)
    {
        temp[m] = -1;
    }
    for(m = 0; m < pages; m++)
    {
        s = 0;
        for(n = 0; n < frames; n++)
        {
            if(referenceString[m] == temp[n])
            {
```

```
                s++;
                pageFaults--;
            }
        }
        pageFaults++;
        if((pageFaults <= frames) && (s == 0))
        {
            temp[m] = referenceString[m];
        }
        else if(s == 0)
        {
            temp[(pageFaults - 1) % frames] = referenceString[m];
        }
        printf("\n");
        for(n = 0; n < frames; n++)
        {
            printf("%d\t", temp[n]);
        }
    }
    printf("\nTotal Page Faults:\t%d\n", pageFaults);
    return 0;
}
```

Algorithm for Optimal Page Replacement

Step 1: Push the first page in the stack as per the memory demand.
Step 2: Push the second page as per the memory demand.
Step 3: Push the third page until the memory is full.
Step 4: As the queue is full, the page which is least recently used is popped.
Step 5: repeat step 4 until the page demand continues and until the processing is over.
Step 6: Terminate the program.

PROGRAM

```
#include<stdio.h>
int main()
{
   int no_of_frames, no_of_pages, frames[10], pages[30], temp[10], flag1, flag2, flag3, i, j, k,
pos, max, faults = 0;
   printf("Enter number of frames: ");
   scanf("%d", &no_of_frames);

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

   printf("Enter page reference string: ");

   for(i = 0; i < no_of_pages; ++i){
      scanf("%d", &pages[i]);
   }

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

   for(i = 0; i < no_of_pages; ++i){
      flag1 = flag2 = 0;

      for(j = 0; j < no_of_frames; ++j){
         if(frames[j] == pages[i]){
               flag1 = flag2 = 1;
               break;
            }
      }

      if(flag1 == 0){
         for(j = 0; j < no_of_frames; ++j){
            if(frames[j] == -1){
               faults++;
```

```
                frames[j] = pages[i];
                flag2 = 1;
                break;
              }
            }
          }

          if(flag2 == 0){
           flag3 =0;

            for(j = 0; j < no_of_frames; ++j){
             temp[j] = -1;

             for(k = i + 1; k < no_of_pages; ++k){
             if(frames[j] == pages[k]){
             temp[j] = k;
             break;
             }
             }
            }

            for(j = 0; j < no_of_frames; ++j){
             if(temp[j] == -1){
             pos = j;
             flag3 = 1;
             break;
             }
            }

            if(flag3 ==0){
             max = temp[0];
             pos = 0;

             for(j = 1; j < no_of_frames; ++j){
             if(temp[j] > max){
             max = temp[j];
             pos = j;
             }
             }
            }
frames[pos] = pages[i];
faults++;
          }

          printf("\n");
```

```
            for(j = 0; j < no_of_frames; ++j){
                printf("%d\t", frames[j]);
            }
        }

        printf("\n\nTotal Page Faults = %d", faults);

        return 0;
    }
```

Least Recently Used (LRU)

LRU page replacement algorithm works on the concept that the pages that are heavily used in previous instructions are likely to be used heavily in next instructions. And the page that are used very less are likely to be used less in future. Whenever a page fault occurs, the page that is least recently used is removed from the memory frames. Page fault occurs when a referenced page in not found in the memory frames.

PROGRAM

```
#include<stdio.h>
 int findLRU(int time[], int n){
int i, minimum = time[0], pos = 0;
 for(i = 1; i < n; ++i){
if(time[i] < minimum){
minimum = time[i];
pos = i;
}
}
return pos;
}

int main()
{
   int no_of_frames, no_of_pages, frames[10], pages[30], counter = 0, time[10], flag1, flag2,
i, j, pos, faults = 0;
printf("Enter number of frames: ");
scanf("%d", &no_of_frames);
printf("Enter number of pages: ");
scanf("%d", &no_of_pages);
printf("Enter reference string: ");
   for(i = 0; i < no_of_pages; ++i){
    scanf("%d", &pages[i]);
   }

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

   for(i = 0; i < no_of_pages; ++i){
    flag1 = flag2 = 0;

    for(j = 0; j < no_of_frames; ++j){
    if(frames[j] == pages[i]){
    counter++;
    time[j] = counter;
```

```
    flag1 = flag2 = 1;
    break;
    }
     }

     if(flag1 == 0){
for(j = 0; j < no_of_frames; ++j){
     if(frames[j] == -1){
     counter++;
     faults++;
     frames[j] = pages[i];
     time[j] = counter;
     flag2 = 1;
     break;
     }
     }
     }

     if(flag2 == 0){
     pos = findLRU(time, no_of_frames);
     counter++;
     faults++;
     frames[pos] = pages[i];
     time[pos] = counter;
     }

     printf("\n");

     for(j = 0; j < no_of_frames; ++j){
     printf("%d\t", frames[j]);
     }
}
printf("\n\nTotal Page Faults = %d", faults);

    return 0;
}
```

Program -10

Write a program to implementation and compare various disk and drum scheduling algorithm.

DESCRIPTION

Disk Scheduling is the process of deciding which of the cylinder request is in the ready queue is to be accessed next. The access time and the bandwidth can be improved by scheduling the servicing of disk I/O requests in good order.

- **Access Time:** The access time has two major components: Seek time and Rotational Latency.
- **Seek Time**: Seek time is the time for disk arm to move the heads to the cylinder containing the desired sector.
- **Rotational Latency:** Rotational latency is the additional time waiting for the disk to rotate the desired sector to the disk head.
- **Bandwidth:** The disk bandwidth is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer.

ALGORITHM:

1. Input the maximum number of cylinders and work queue and its head starting position.
2. First Come First Serve Scheduling (FCFS) algorithm – The operations are performed in order requested.
3. There is no reordering of work queue.
4. Every request is serviced, so there is no starvation.
5. The seek time is calculated.
6. Shortest Seek Time First Scheduling (SSTF) algorithm – This algorithm selects the request with the minimum seek time from the current head position.
7. Since seek time increases with the number of cylinders traversed by the head, SSTF chooses the pending request closest to the current head position.
8. The seek time is calculated.
9. SCAN Scheduling algorithm – The disk arm starts at one end of the disk, and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk.

10. At the other end, the direction of head movement is reversed, and servicing continues.

11. The head continuously scans back and forth across the disk.

12. The seek time is calculated.

13. Display the seek time and terminate the program

PROGRAM :

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

void fcfs(int noq, int qu[10], int st)
{
 int i,s=0;
 for(i=0;i<noq;i++)
 {
  s=s+abs(st-qu[i]);
  st=qu[i];
  }
 printf("\n Total seek time :%d",s);
}

void sstf(int noq, int qu[10], int st, int visit[10])
{
 int min,s=0,p,i;
 while(1)
{
  min=999;
  for(i=0;i<noq;i++)
   if (visit[i] == 0)
    {
      if(min > abs(st - qu[i]))
       {
        min = abs(st-qu[i]);
        p = i;
        }
      }
 if(min == 999)
  break;
  visit[p]=1;
  s=s + min;
  st = qu[p];
  }
 printf("\n Total seek time is: %d",s);
 }
```

```
void scan(int noq, int qu[10], int st, int ch)
{
 int i,j,s=0;
 for(i=0;i<noq;i++)
 {
  if(st < qu[i])
  {
   for(j=i-1; j>= 0;j--)
   {
    s=s+abs(st - qu[j]);
    st = qu[j];
    }
   if(ch == 3)
   {
   s = s + abs(st - 0);
   st = 0;
   }
  for(j = 1;j < noq;j++)
  {
   s= s + abs(st - qu[j]);
   st = qu[j];
   }
  break;
 }
}
printf("\n Total seek time : %d",s);
}

int main()
{
 int n,qu[20],st,i,j,t,noq,ch,visit[20];
 printf("\n Enter the maximum number of cylinders : ");
 scanf("%d",&n);
 printf("enter number of queue elements");
 scanf("%d",&noq);
 printf("\n Enter the work queue");
 for(i=0;i<noq;i++)
 {
  scanf("%d",&qu[i]);
  visit[i] = 0;
  }
 printf("\n Enter the disk head starting posision: \n");
 scanf("%d",&st);
 while(1)
 {
  printf("\n\n\t\t MENU \n");
```

```
 printf("\n\n\t\t 1. FCFS \n");
 printf("\n\n\t\t 2. SSTF \n");
 printf("\n\n\t\t 3. SCAN \n");
 printf("\n\n\t\t 4. EXIT \n");
 printf("\nEnter your choice: ");
 scanf("%d",&ch);
 if(ch > 2)
  {
  for(i=0;i<noq;i++)
  for(j=i+1;j<noq;j++)
  if(qu[i]>qu[j])
  {
   t=qu[i];
   qu[i] = qu[j];
   qu[j] = t;
   }
  }
  switch(ch)
   {
   case 1: printf("\n FCFS \n");
           printf("\n*****\n");
           fcfs(noq,qu,st);
           break;

   case 2: printf("\n SSTF \n");
           printf("\n*****\n");
           sstf(noq,qu,st,visit);
           break;
   case 3: printf("\n SCAN \n");
           printf("\n*****\n");
           scan(noq,qu,st,ch);
           break;
   case 4: exit(0);
}                              }}
```