PRACTICAL FILE OF OPERATING SYSTEM

BTech: III Year

Department of Computer Science & Information Technology

Name of the Student : ANJALI PATEL

Branch & section : CSIT – 01

Roll No. : 0827CI201027

Year : III Year

Department of Computer Science & Information Technology

AITR, Indore,

ACROPOLIS INSTITUTE OF TECHNOLOGY & RESEARCH, INDORE

Department of Computer Science & Information Technology

Certificate

This is to certify that the experimental work entered in this journalas per the BTech III year syllabus prescribed by the RGPV was done by Mr. / Ms. Anjali Patel in V semester in the Laboratory of this institute during the academic year July 2022 – Dec 2022.

Signature of Head Signature of the Faculty

ACROPOLIS INSTITUTE OF TECHNOLOGY & RESEARCH, INDOREGENERAL INSTRUCTIONS FOR LABORATORY CLASSES

DO'S

- Without Prior permission do not enter into the Laboratory.
- While entering into the LAB students should wear their ID cards.
- The Students should come with proper uniform.
- Students should maintain silence inside the laboratory.
- After completing the laboratory exercise, make sure to shutdown the system properly.

DONT'S

- Students bringing the bags inside the laboratory.
- Students using the computers in an improper way.
- Students scribbling on the desk and mishandling the chairs.
- Students using mobile phones inside the laboratory.
- Students making noise inside the laboratory.

SYLLABUS

CS-502 – Operating System

Branch: Computer Science & Information Technology V Semester

Course: CSIT 502 Operating System

Unit I

Introduction to System Programs & Operating Systems, Evolution of Operating System (mainframe, desktop, multiprocessor, Distributed, Network Operating System, Clustered & Handheld System), Operating system services, Operating system structure, System Call & System Boots, Operating system design & Implementations, System protection, Buffering & Spooling. Types of Operating System: Bare machine, Batch Processing, Real Time, Multitasking& Multiprogramming, time-sharing system.

Unit II

File: concepts, access methods, free space managements, allocation methods, and directory systems, protection, organization ,sharing & implementation issues, Disk & Drum Scheduling, I/0 devices organization, I/0 devices organization, I/0 buffering, I/O Hardware, Kernel I/Osubsystem, Transforming I/O request to hardware operations. Device Driver: Path managements,Sub module, Procedure, Scheduler, Handler, Interrupt Service Routine. File system in Linux & Windows

Unit III

Process: Concept, Process Control Blocks (PCB), Scheduling criteria Preemptive & non Preemptive process scheduling, Scheduling algorithms, algorithm evaluation, multiple processor scheduling, real time scheduling, operations on processes, threads; inter process communication, precedence graphs, critical section problem, semaphores, and classical problems of synchronization. Deadlock: Characterization, Methods for deadlock handling, deadlock prevention, deadlock avoidance, deadlock detection, recovery from deadlock, Process Management in Linux.

Unit IV

Memory Hierarchy, Concepts of memory management, MFT & MVT, logical and physical address space, swapping, contiguous and non-contiguous allocation, paging, segmentation, and paging combined with segmentation. Structure & implementation of Page table. Concepts of virtual memory, Cache Memory Organization, demand paging, page replacement algorithms, allocation of frames, thrashing, and demand segmentation.

Unit V

Distributed operating system:-Types, Design issues, File system, Remote file access, RPC, RMI, Distributed Shared Memory(DSM), Basic Concept of Parallel Processing & Concurrent Programming Security & threats protection: Security violation through Parameter, Computer Worms & Virus, Security Design Principle, Authentications, Protection Mechanisms. introduction to Sensor network and parallel operating system. Case study of Unix, Linux & Windows

HARDWARE REQUIREMENTS:

Processors - 2.0 GHz or
HigherRAM - 256 MB or
Higher Hard Disk - 20 GB
or Higher

SOFTWARE REQUIREMENTS:

Linux: Ubuntu / OpenSUSE / Fedora / Red Hat / Debian /
Mint OSWINDOWS: XP/7
Linux could be loaded in individual PCs.

RATIONALE:

The purpose of this subject is to cover the underlying concepts Operating System .This syllabus provides a comprehensive introduction of Operating System, Process Management, Memory Management, File Management and I/O management.

PREREQUISITE:

The students should have general idea about Operating System Concept, types of Operating System and their functionality.

Lab Plan

Operating SystemCSIT-502

S.No	Name of Experiment	Page No.
1.	Program to implement FCFS scheduling	8
2.	Program to implement SJF scheduling	15
3.	Program to implement SRTF sceduling	20
4.	Program to implement Round Robin scheduling	25
5.	Program to implement Priortiy scheduling	29
6.	Program to implement Banker's algorithm	34
7.	Program to implement FIFO page replacement algorithm.	40
8.	Program to implement LRU page replacement algorithm	43
9	Program to implement Disk Scheduling(FIFO) algorithm	46

ACROPOLIS INSTITUTE OF TECHNOLOGY & RESEARCH, INDORE

Name of Department--------
----Name of Laboratory-----

Index

S.No.	Date of Exp.	Name of the Experiment	Page No.	Date of Submission	Grade & Sign of the Faculty
1.		Program to implement FCFS scheduling	8-14		
2.		Program to implement SJF scheduling	15-19		
3.		Program to implement SRTF scheduling	20-24		
4.		Program to implement Round Robin scheduling	25-28		
5.		Program to implement Priority scheduling	29-33		
6.		Program to implement Banker's algorithm	34-39		
7.		Program to implement FIFO page replacement algorithm.	40-42		
8.		Program to implement LRU page replacement algorithm	43-45		
9.		Program to implement Disk Scheduling(FIFO) algorithm	46-48		

Experiment-1

FCFS SCHEDULING

Name of Student: Anjali Patel		Class: CSIT-1
Enrollment No: 0827CI201027		Batch 2020-2024
Date of Experiment	Date of Submission	Submitted on:
Remarks by faculty:	Grade:	
Signature of student:	Signature of Faculty:	

OBJECTIVE OF THE EXPERIMENT
To write c++ program to implement the FCFS SCHEDULING.

FACILITIES REQUIRED

a) **Facilities Required Doing The Experiment:**

S.NO	FACILITIES REQUIRED	QUANTITY
1	**System**	**1**
2	**WINDOWS XP/7**	

b) **Concept of FCFS:**

- Jobs are executed on first come, first serve basis.
- Easy to understand and implement.
- Poor in performance as average wait time is high.

Process	Arrival Time	Execute Time	Service Time
P0	0	5	0
P1	1	3	5
P2	2	8	8
P3	3	6	16

c) 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 timeStep 4: Set the waiting of the first process as ‘0’ and its burst time as its turn around time Step 5: for each process in the Ready Q calculate

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

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

Step 6: Calculate

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

(b) Average Turnaround time = Total Turnaround Time / Number of processStep 7: Stop the process

d) Program:

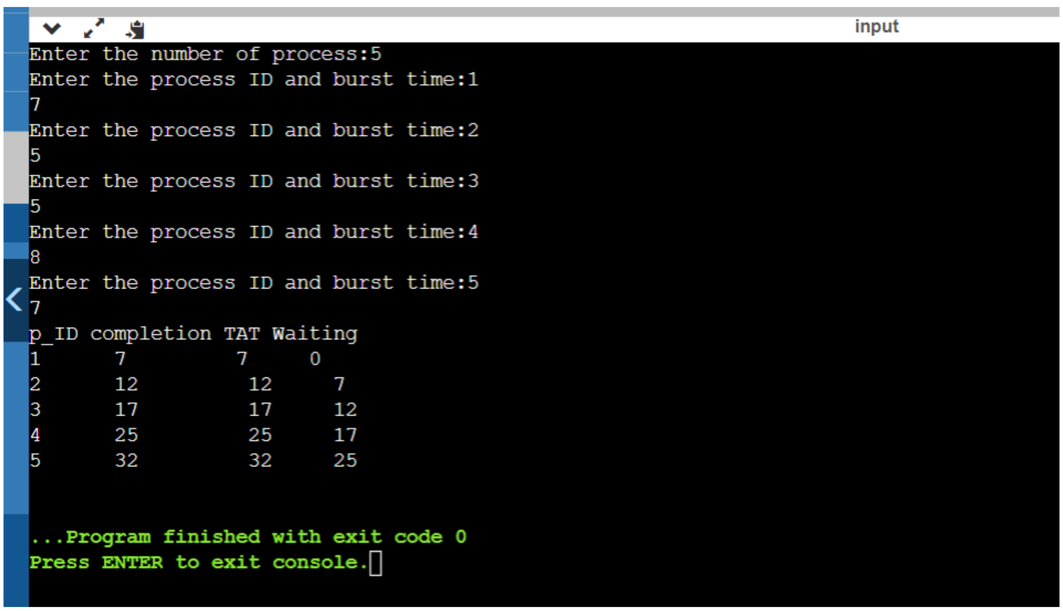
First Come First Serve with zero arrival time

```
#include <iostream>
using namespace std;
class FCFS
{
public:
void completion(int procesid[],int n,int bursttime[],int comp[])
{
   comp[0]=bursttime[0];
   for(int i=1;i<n;i++)
   {
      comp[i]=comp[i-1]+bursttime[i];
   }
}
```

```
void turnaround_time(int procesid[],int n,int bursttime[],int tat[],int comp[])
{

    for(int i=0;i<n;i++)
    {
        tat[i]=comp[i];
    }
}
void waiting_time(int procesid[],int n,int bursttime[],int tat[],int wait[])
{
    for(int i=0;i<n;i++)
    {
        wait[i]=tat[i]-bursttime[i];
    }
}
void display(int procesid[],int comp[],int tat[],int wait[],int n)
{
    cout<<"p_ID completion TAT Waiting"<<endl;
    for(int i=0;i<n;i++)
    {
        cout<<procesid[i]<<"      " <<comp[i]<<"         "<<tat[i]<<"     "<<wait[i]<<endl;
    }
}
};
int main()
{
    FCFS f;
      int n;
      cout<<"Enter the number of process:";
      cin>>n;
      int procesid[n];
      int bursttime[n];

      for(int i=0;i<n;i++)
      {
          cout<<"Enter the process ID and burst time:";
          cin>>procesid[i]>>bursttime[i];
      }
      int comp[n];
      int tat[n];
```

```
int wait[n];

f. completion( procesid, n, bursttime,comp);

f.turnaround_time( procesid, n, bursttime, tat,comp);
f.waiting_time( procesid, n, bursttime, tat,wait);
f.display( procesid,comp,tat, wait, n);


return 0;
}
```

e) Output:




```
Enter the number of process:5
Enter the process ID and burst time:1
7
Enter the process ID and burst time:2
5
Enter the process ID and burst time:3
5
Enter the process ID and burst time:4
8
Enter the process ID and burst time:5
7
p_ID completion TAT Waiting
1      7         7     0
2      12        12     7
3      17        17     12
4      25        25     17
5      32        32     25


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

f) Result:

Average Waiting Time …12.2…..

Average Turnaround Time ……18.6….

First Come First Serve with varying arrival time

```
#include <iostream>
class FCFS
{
public:
void completion(int procesid[],int n,int bursttime[],int comp[])
{
    comp[0]=bursttime[0];
    for(int i=1;i<n;i++)
    {
        comp[i]=comp[i-1]+bursttime[i];
    }
}


void turnaround_time(int procesid[],int n,int bursttime[],int tat[],int comp[],int arrivaltime[])
{

    for(int i=0;i<n;i++)
    {
        tat[i]=comp[i]-arrivaltime[i];
    }
}
void waiting_time(int procesid[],int n,int bursttime[],int tat[],int wait[])
{
    for(int i=0;i<n;i++)
    {
        wait[i]=tat[i]-bursttime[i];
    }
}
void display(int procesid[],int comp[],int tat[],int wait[],int n,int arrivaltime[],int bursttime[])
{
    cout<<"p_ID    arrival   burst completion  TAT       Waiting"<<endl;
    for(int i=0;i<n;i++)
    {
        cout<<procesid[i]<<"       "<<arrivaltime[i]<<"       "<<bursttime[i]<<"        " <<comp[i]<<"
"<<tat[i]<<"            "<<wait[i]<<endl;
    }
}
```

```
};
int main()
{
    FCFS f;
      int n;
      cout<<"Enter the number of process:";
      cin>>n;
      int procesid[n];
      int bursttime[n];
       int arrivaltime[n];
      for(int i=0;i<n;i++)
      {
          cout<<"Enter the process ID "<<endl;
          cin>>procesid[i];
      }
      for(int i=0;i<n;i++)
      {
          cout<<"Enter the arrival time and burst time of Process :"<<i+1<<endl;
          cin>>arrivaltime[i]>>bursttime[i];
      }
      int comp[n];
      int tat[n];
      int wait[n];

     f. completion( procesid, n, bursttime,comp);


      f.turnaround_time( procesid, n, bursttime, tat,comp,arrivaltime);
   f.waiting_time( procesid, n, bursttime, tat,wait);
  f.display( procesid,comp,tat, wait, n,arrivaltime,bursttime);


    return 0;
}
```

a) **Output:**

```
3
Enter the process ID
4
Enter the arrival time and burst time of Process :1
0
5
Enter the arrival time and burst time of Process :2
1
3
Enter the arrival time and burst time of Process :3
2
8
Enter the arrival time and burst time of Process :4
3
6
p_ID     arrival    burst completion  TAT        Waiting
1        0          5          5          5           0
2        1          3          8          7           4
3        2          8          16          14           6
4        3          6          22          19           13


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

b) **Result:**

Average Waiting Time …5.75…..

Average Turnaround Time …11.25…….

Experiment-2

SJF Scheduling

Name of Student: Anjali Patel		Class: CSIT-1
Enrollment No: 0827CI201027		Batch 2020-2024
Date of Experiment	Date of Submission	Submitted on:
Remarks by faculty:	Grade:	
Signature of student:	Signature of Faculty:	

OBJECTIVE OF THE EXPERIMENT

To write c++ program to implement SJF CPU Scheduling Algorithm.

FACILITIES REQUIRED

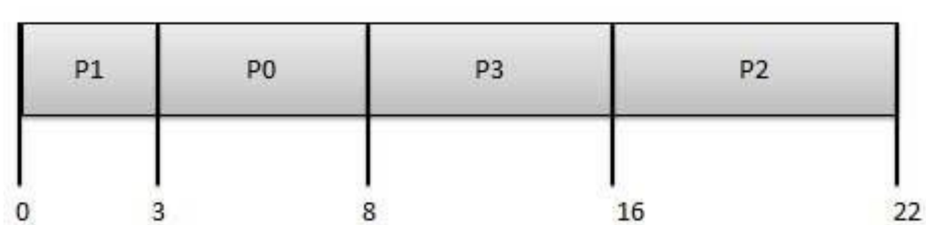
a) Facilities Required Doing The Experiment:

S.NO	FACILITIES REQUIRED	QUANTITY
1	**System**	**1**
2	**WINDOWS XP/7**	

b) Concept of SJF:

- Best approach to minimize waiting time.
- Processer should know in advance how much time process will take.

Process	Arrival Time	Execute Time	Service Time
P0	0	5	0
P1	1	3	3
P2	2	8	8
P3	3	6	16



c) 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

timeStep 4: Start the Ready Q according the shortest Burst time by sorting according to

lowest tohighest burst time.

Step 5: Set the waiting time of the first process as ‘0’ and its turnaround time as its burst time.
Step 6: For each process in the ready queue, calculate

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

(d) Turnaround time for Process(n)= waiting time of Process(n)+ Burst time for process(n)

Step 7: Calculate

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

(d) Average Turnaround time = Total Turnaround Time / Number

of processStep 8: Stop the process

d) Program:

```
#include<iostream>

using namespace std;


struct process
   {
      int id,bursttime,completion,wait ,tat;
   };

   bool cmp(process x,process y)
```

```
{
    return (x.bursttime<y.bursttime);
}


void turnaround_time( int n,process array[])
{
  array[0].tat=array[0].bursttime;
   for(int i=1;i<n;i++)
   {
     array[i].tat=array[i-1].tat+array[i].bursttime;
   }
}
void waiting_time(process array[],int n)
{
   array[0].wait=0;
   for(int i=1;i<n;i++)
   {
      array[i].wait=array[i-1].wait+array[i-1].bursttime;
   }
}
/*void completion(int procesid[],int n,int bursttime[],int comp[])
{

   for(int i=1;i<n;i++)
   {
      comp[i]=;
   }
}*/
void display(int n,process array[])
{
   cout<<"p_ID    burst   TAT         Waiting"<<endl;
   for(int i=0;i<n;i++)
   {
      cout<<array[i].id<<"      "<<array[i].bursttime<<"        "<<array[i].tat<<"
"<<array[i].wait<<endl;
   }
}


int main()
{
```

```
    int n;
    float avftat,avgwait;
   long long int totaltat=0,toalwait=0;

    cout<<"Enter the number of process:";
    cin>>n;

    process array[n];

    for(int i=0;i<n;i++)
    {
        cout<<"Enter the process ID and burst time (arrival time is  zero) :";
        cin>>array[i].id>>array[i].bursttime;
    }
    sort(array,array+n,cmp);



    turnaround_time( n, array);
  waiting_time( array, n);
//  completion( procesid, n, bursttime,comp);
 display( n,array);
 for(int i=0;i<n;i++)
 {
    totaltat+=array[i].tat;
 }
  for(int i=0;i<n;i++)
 {
    toalwait+=array[i].wait;
 }
 cout<<"The average TAT is :"<<totaltat/n;
 cout<<"\n The averageWaiting time is "<<toalwait/n;
   return 0;
}
```

e) Output:

C:\Users\HP\Documents\SJF.exe

```
Enter the number of process:4
Enter the process ID and burst time (arrival time is  zero) :1
5
Enter the process ID and burst time (arrival time is  zero) :2
3
Enter the process ID and burst time (arrival time is  zero) :3
8
Enter the process ID and burst time (arrival time is  zero) :4
6
p_ID     burst   TAT           Waiting
2       3         3           0
1       5         8           3
4       6         14           8
3       8         22           14
The average TAT is :11
 The averageWaiting time is 6
--------------------------------
Process exited after 26.22 seconds with return value 0
Press any key to continue . . .
```

f) Result:

Average Waiting Time …6…..

Average Turnaround Time …11…….

Experiment-3

SRTF Scheduling

Name of Student: Anjali Patel		Class: CSIT-1
Enrollment No:0827CI201027		Batch: 2020-2024
Date of Experiment	Date of Submission	Submitted on:
Remarks by faculty:	Grade:	
Signature of student:	Signature of Faculty:	

OBJECTIVE OF THE EXPERIMENT
To write c++ program to implement SRTF scheduling.

FACILITIES REQUIRED

a) **Facilities Required To Do The Experiment:**

S.NO	FACILITIES REQUIRED	QUANTITY
1	**System**	**1**
2	**Windows XP/7**	

b) **Concept Of SRTF Scheduling:**

- Associate with each process the length of its next CPU burst. Use these lengths toschedule the process with the shortest time.
- Two schemes:
- 1. non preemptive – once CPU given to the process it cannot be preempted untilcompletes its CPU burst.
- 2. Preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is known as the Shortest-Remaining-Time-First (SRTF).

Example of Preemptive SJF

P1	P2	P3	P2	P4	P1

Process Arrival Time Burst Time

P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

SJF (preemptive)

0 2 4 5 7 11 16

c) 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

timeStep 4: For each process in the ready Q, Accept Arrival time

Step 4: Start the Ready Q according the shortest Burst time by sorting according to lowest

toHighest burst time.

Step 5: Set the waiting time of the first process in Sorted Q as ‘0’.
Step 6: After every unit of time compare the remaining time of currently executing process (RT) and Burst time of newly arrived process (BTn).
Step 7: If the burst time of newly arrived process (BTn) is less than the currently executing process (RT) the processor will preempt the currently executing process and starts executing newly arrived process
Step 7: Calculate

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

(d) Average Turnaround time = Total Turnaround Time / Number

of processStep 8: Stop the process

d) Program:

```
#include<iostream>

using namespace std;
    int main()
    {
       int a[10],b[10],x[10];
       int waiting[10] ,  turnaround[10] , completion[10];
       int i,j,smallest,count=0,time,n;
       double avg=0,tt=0,end;

       cout<<"\nEnter the number of Processes: ";  //input
       cin>>n;
       for(i=0; i<n; i++)
       {
          cout<<"\nEnter arrival time of process: ";  //input
          cin>>a[i];
       }
       for(i=0; i<n; i++)
       {
          cout<<"\nEnter burst time of process: ";  //input
          cin>>b[i];
       }
       for(i=0; i<n; i++)
          x[i]=b[i];

       b[9]=9999;
       for(time=0; count!=n; time++)
       {
          smallest=9;
          for(i=0; i<n; i++)
          {
             if(a[i]<=time && b[i]<b[smallest] && b[i]>0 )
                smallest=i;
          }
          b[smallest]--;

          if(b[smallest]==0)
          {
             count++;
             end=time+1;
             completion[smallest] = end;
             waiting[smallest] = end - a[smallest] - x[smallest];
```

```
            turnaround[smallest] = end - a[smallest];
        }
    }
    cout<<"Process"<<"\t"<< "burst-time"<<"\t"<<"arrival-time" <<"\t"<<"waiting-time"
<<"\t"<<"turnaround-time"<< "\t"<<"completion-time"<<endl;
    for(i=0; i<n; i++)
    {

cout<<"p"<<i+1<<"\t\t"<<x[i]<<"\t\t"<<a[i]<<"\t\t"<<waiting[i]<<"\t\t"<<turnaround[i]<
<"\t\t"<<completion[i]<<endl;
        avg = avg + waiting[i];
        tt = tt + turnaround[i];
    }

    cout<<"\n\nAverage waiting time ="<<avg/n;
    cout<<"  Average Turnaround time ="<<tt/n<<endl;

}
```

e) Output:

```
Enter arrival time of process: 0

Enter arrival time of process: 1

Enter arrival time of process: 2

Enter arrival time of process: 3

Enter arrival time of process: 4

Enter burst time of process: 5

Enter burst time of process: 3

Enter burst time of process: 1

Enter burst time of process: 2

Enter burst time of process: 3
Process burst-time      arrival-time    waiting-time    turnaround-time completion-time
p1              5               0               9               14              14
p2              3               1               1               4               5
p3              1               2               0               1               3
p4              2               3               2               4               7
p5              3               4               3               6               10


Average waiting time =3  Average Turnaround time =5.8
```

f) Result:

Average Waiting Time =3

Average Turnaround Time = 5.8

Experiment-4

ROUND ROBIN Scheduling

Name of Student: Anjali Patel		Class: CSIT-1
Enrollment No: 0827CI201027		Batch 2020-2024
Date of Experiment	Date of Submission	Submitted on:
Remarks by faculty:	Grade:	
Signature of student:	Signature of Faculty:	

OBJECTIVE OF THE EXPERIMENT
To write c program to implement Round Robin scheduling.

FACILITIES REQUIRED

a) **Facilities Required To Do The Experiment:**

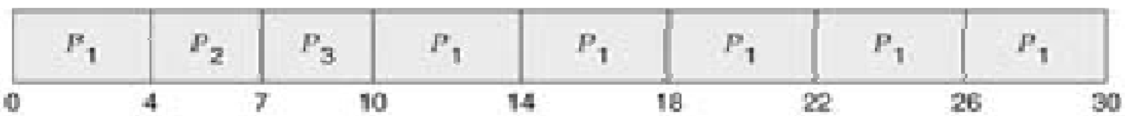
S.NO	FACILITIES REQUIRED	QUANTITY
1	**System**	**1**
2	**Windows XP/7**	

b) **Concept Of Round Robin Scheduling:**

This Algorithm is designed especially for time-sharing systems. A small unit of time, called time slices or **quantum** is defined. All runnable processes are kept in a circular queue. The CPU scheduler goes around this queue, allocating the CPU to each process for a time interval of one quantum. New processes are added to the tail of the queue. The CPU scheduler picks the first process from the queue, sets a timer to interrupt after one quantum, and dispatches the process. Ifthe process is still running at the end of the quantum, the CPU is preempted and the process is added to the tail of the queue. If the process finishes before the end of the quantum, the process itself releases the CPU voluntarily Every time a process is granted the CPU, a **context switch** occurs, this adds overhead to the process execution time.

	Burst
Process	Time
P_1	24
P_2	3

P_3	3
Average	



c) Algorithm:

Step 1: Start the process

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

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

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

Step 7: Calculate

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

(h) Average Turnaround time = Total Turnaround Time / Number

of processStep 8: Stop the process

d) Program:

```
#include <iostream>
using namespace std;
int main()
{
  int n,to=0;
  cout<<"Enter number of process"<<endl;
  cin>>n;
  int ar[n][2];
  cout<<"Enter the burst time of each process "<<endl;
  for(int i=0;i<n;i++)
  {
     cin>>ar[i][1];
     to=to+ar[i][1];
     ar[i][0]=i+1;
  }
  cout<<"Total = "<<to<<endl<<endl;
  int tq;
  cout<<"Enter time quantum"<<endl;
  cin>>tq;
  int num=0;
  cout<<"**Gantt Chart**"<<endl<<endl;
  while(to>0)
  {
     if(ar[num][1]>=tq)
     {
        ar[num][1]=ar[num][1]-tq;
        to=to-tq;

        cout<<num+1<<"  ";
     }
     else if(ar[num][1]<tq && ar[num][1]>0)
     {

        to=to-ar[num][1];
        ar[num][1]=0;

        cout<<num+1<<"  ";
     }
        num++;
        if(num==n)
```

```
        {num=0;}

    }

 return 0;}
```

e) Output:-

```
C:\Users\HP\Documents\roundrobin.exe
Enter number of process
5
Enter the burst time of each process
5
3
1
2
3
Total = 14

Enter time quantum
2
**Gantt Chart**

1  2  3  4  5  1  2  5  1
--------------------------------
Process exited after 10.79 seconds with return value 0
Press any key to continue . . .
```

a) Result:

Average Waiting Time ……..

Average Turnaround Time ……….

Experiment-5

PRIORITY SCHEDULING

Name of Student: Anjali Patel		Class: CSIT -1
Enrollment No:0827CI201027		Batch: 2020-2024
Date of Experiment	Date of Submission	Submitted on:
Remarks by faculty:		Grade:
Signature of student:		Signature of Faculty:

OBJECTIVE OF THE EXPERIMENT
To write c program to implement Priority scheduling.

FACILITIES REQUIRED

a) **Facilities Required To Do The Experiment:**

S.NO	FACILITIES REQUIRED	QUANTITY
1	**System**	**1**
2	**Windows XP/7**	

b) **Concept Of Priority Scheduling:**

A priority is associated with each process, and the CPU is allocated to the process with thehighest priority. Equal-priority processes are scheduled in FCFS order.
An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the(predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa.
As an example, consider the following set of processes, assumed to have arrived at time 0, in theorder

	Burst		Waiting	Turnaround
Process	Time	Priority	Time	Time
P_1	10	3	6	16
P_2	1	1	0	1
P_3	2	4	16	18

P_4	1	5	18	19
P_5	5	2	1	6
Average	-	-	8.2	12

c) 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

timeStep 4: Sort the ready queue according to the priority number.

Step 5: Set the waiting of the first process as '0' and its burst time as its turn around

timeStep 6: For each process in the Ready Q calculate

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

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

Step 7: Calculate

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

(j) Average Turnaround time = Total Turnaround Time / Number

of processStep 8: Stop the process

d) Program:

```
#include<iostream>
#include<bits/stdc++.h>
using namespace std;

    int main()
    {
        int bt[20],p[20],wt[20],tat[20],pr[20],i,j,n,total=0,pos,temp,avg_wt,avg_tat;
        cout<<"Enter Total Number of Process:";
        cin>>n;

        cout<<"\nEnter Burst Time and Priority\n";
        for(i=0;i<n;i++)
        {
            cout<<(i+1)<<")";
            cin>>bt[i];
            cin>>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;
    cout<<"\nProcess\t    Burst Time    \tWaiting Time\tTurnaround Time";
    for(i=0;i<n;i++)
    {
        tat[i]=bt[i]+wt[i];
        total+=tat[i];
        cout<<"\nP["<<p[i]<<"]\t\t  "<<bt[i]<<"\t\t    "<<wt[i]<<"\t\t\t"<<tat[i];
    }
    avg_tat=total/n;
    cout<<"\n\nAverage Waiting Time="<<avg_wt;
    cout<<"\nAverage Turnaround Time="<<avg_tat;

    return 0;
}
```

e) **Output:**

```
Enter Total Number of Process:5

Enter Burst Time and Priority
1)4
2
2)3
3
3)3
1
4)2
5
5)4
4

Process     Burst Time          Waiting Time    Turnaround Time
P[3]              3                  0                3
P[1]              4                  3                7
P[2]              3                  7                10
P[5]              4                  10               14
P[4]              2                  14               16

Average Waiting Time=6
Average Turnaround Time=10

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

f) **Result:**

Average Waiting Time = 6

Average Turnaround Time =10

Experiment-6

BANKER ALGORITHM

Name of Student: Anjali Patel		Class: CSIT-1
Enrollment No: 0827CI201027		Batch 2020-2024
Date of Experiment	Date of Submission	Submitted on:
Remarks by faculty:	Grade:	
Signature of student:	Signature of Faculty:	

OBJECTIVE OF THE EXPERIMENT

To write c program to implement deadlock avoidance & Prevention by using Banker's Algorithm.

FACILITIES REQUIRED

a) **Facilities Required To Do The Experiment:**

S.NO	FACILITIES REQUIRED	QUANTITY
1	System	1
2	Windows XP/7	

b) **Concept Of BANKER'S Algorithm:**

The Banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation of predetermined maximum possible amounts of all resources, and then makes an "s-state" check to test for possible deadlock conditions for all other pending activities, before deciding whether allocation should be allowed to continue.

- Always keep so many resources that satisfy the needs of at least one client
- Multiple instances.
- Each process must a priori claim maximum use.
- When a process requests a resource it may have to wait.
- When a process gets all its resources it must return them in a finite amount oftime.

c) **Algorithm:**

1. Start the program.
2. Get the values of resources and processes.
3. Get the avail value.

4. After allocation find the need value.
5. Check whether it’s possible to allocate.
6. If it is possible then the system is in safe state.
7. Else system is not in safety state.
8. If the new request comes then check that the system is in safety.
9. Or not if we allow the request.
10. Stop the program.

d) Program:

```
#include<iostream>

using namespace std;


int alloc[50][50];


int maxi[50][50];


int need[50][50];


int avail[50];


int check_safety(int j,int nr)


{


      for(int i=0;i<nr;i++)


      {
```

```
        if(need[j][i]>avail[i])
        return 0;
    }
    return 1;
}
int check(bool a[],int n)
{
    for(int i=0;i<n;i++)
    {
        if(a[i]==false)
        return 0;
    }
    return 1;
}
```

```
int main()
{
    int np=100;s
    int nr=100;


    cout<<"\nEnter the no of processes : ";
    cin>>np;
    cout<<"\nEnter the no of resources : ";
    cin>>nr;
    cout<<"\nEnter the allocation data : \n";
    for(int i=0;i<np;i++)
    for(int j=0;j<nr;j++)
    cin>>alloc[i][j];
cout<<"\nEnter the requirement data : \n";
```

```
for(int i=0;i<np;i++)

for(int j=0;j<nr;j++)

cin>>maxi[i][j];

//Calculation of need matrix

for(int i=0;i<np;i++)

for(int j=0;j<nr;j++)

need[i][j]=maxi[i][j]-alloc[i][j];

cout<<"\nEnter the availability matrix : \n";

for(int i=0;i<nr;i++)

cin>>avail[i];

int ex_it=nr;

int flg;

bool completed[np];

while(10)
```

```
{
    for(int i=0;i<np;i++)
    {
        if(!completed[i] && check_safety(i,nr))
        {
            for(int j=0;j<nr;j++)
            avail[j]+=alloc[i][j];
        }
        completed[i]=true;
    }
    flg=check(completed,np);
    ex_it--;
    if(flg==1 || ex_it==0)
    break;
}
cout<<"\nThe final availability matrix \n";
```

```
for(int i=0;i<nr;i++)

cout<<avail[i]<<"  ";

cout<<"\n --------- Result ------------- \n";

if(flg==1)

cout<<"There is no deadlock";

else

cout<<"Sorry there is a possibility of deadlock";

return 0;

}
```

e) Output:

```
Enter the requirement data :
7
5
3
3
2
2
9
0
2
2
2
2
4
3
3

Enter the availability matrix :
3
3
2

The final availability matrix
7  4  6
 --------- Result -------------
There is no deadlock

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

f) Result:

The Sequence Is:-

Experiment-7

FIFO PAGE REPLACEMENT

Name of Student: Anjali Patel			Class: CSIT-1
Enrollment No: 0827CI201027			Batch 2020-2024
Date of Experiment	Date of Submission		Submitted on:
Remarks by faculty:		Grade:	
Signature of student:	Signature of Faculty:		

OBJECTIVE OF THE EXPERIMENT

To implement page replacement algorithm FIFO.

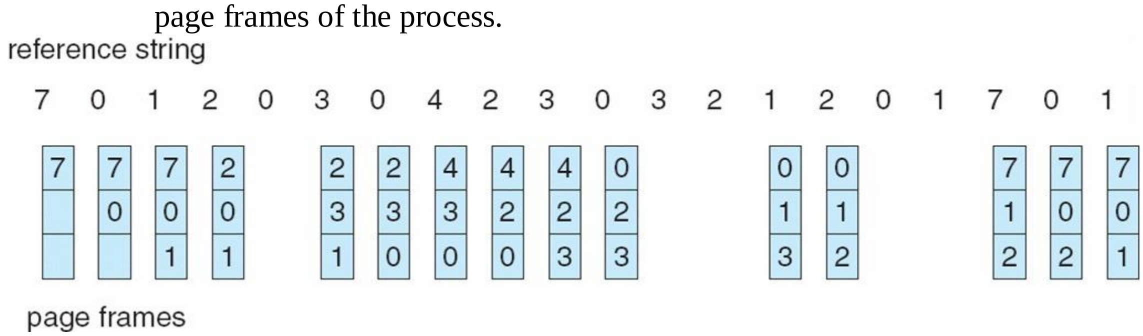
FACILITIES REQUIRED

a) **Facilities Required To Do The Experiment:**

S.NO	FACILITIES REQUIRED	QUANTITY
1	**System**	**1**
2	**Windows XP/7**	

b) **Concept Fifo Page Replacement:**

- Treats page frames allocated to a process as a circular buffer:
- When the buffer is full, the oldest page is replaced. Hence first-in, first-out: A frequently used page is often the oldest, so it will be repeatedly paged out by FIFO. Simple to implement: requires only a pointer that circles through the page frames of the process.



- FIFO Replacement manifests

  Belady's Anomaly:more frames

  more page faults

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3,

  4, 5()3 Frames:-9 page fault

  4 Frames: - 10 page fault

c) Algorithm:

Step 1: Create a queue to hold all pages in memory

Step 2: When the page is required replace the page at the head of the

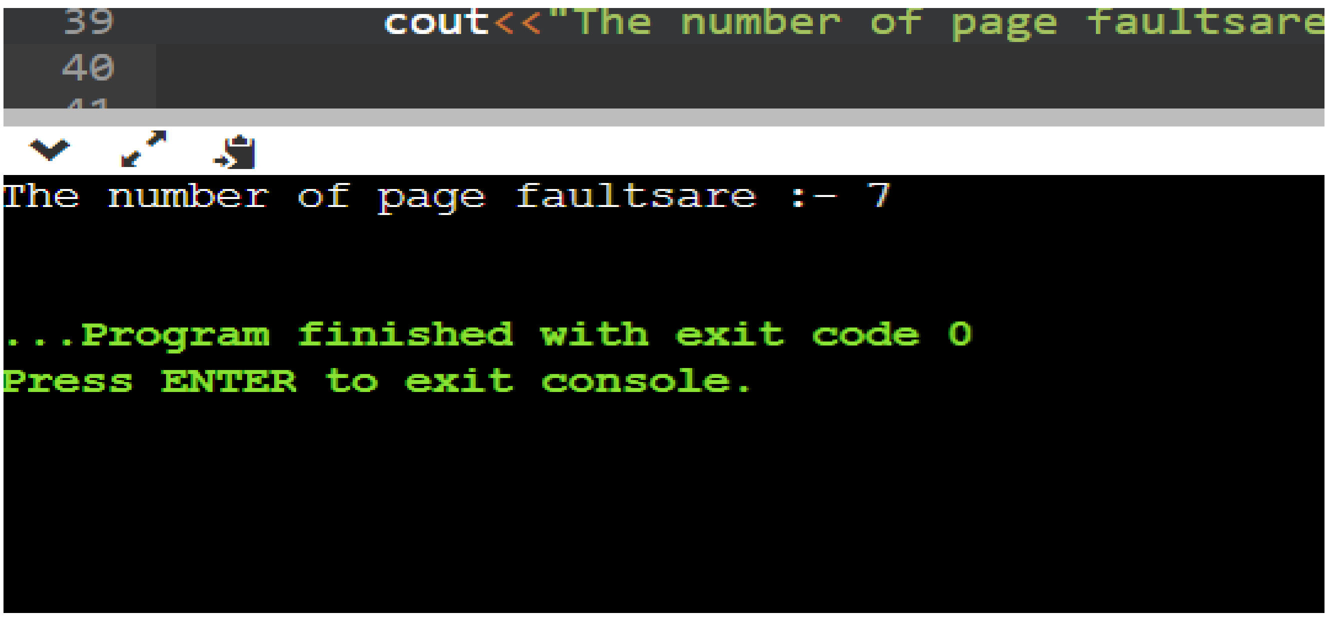
queueStep 3: Now the new page is inserted at the tail of the queue

d) Program:

```
int page_fault_FIFO(int pages[] , int n , int c)
{
   int pagefault=0;
   vector<int> v;
   int i;
   int index=0;
   for(int i=0;i<n;i++)
   {
      auto it = std::find(v.begin() , v.end(),pages[i]);
      if(it==v.end())
      {
         if(v.size()==c)
         {

            v.erase(v.begin()+(index));
         }
        // v.push_back(pages[i]);
        v.insert(v.begin(),index);
        index++;
        index=index%c;
         pagefault++;
      }
      else
      {v.erase(it);
         v.push_back(pages[i]);
      }
   }
   return pagefault;
}
int main()
```

```
{
   int page[]={7,0,1,2,0,3,0,4,2,3,0,3,2};
   int n=13;
   int memocapacity=4;
   cout<<page_fault_FIFO(page,n,memocapacity)<<endl;



   return 0;
}
```

e) Output:



f) Result:

No. of page faults :- ..7....

Experiment-8

LRU PAGE REPLACEMENT

Name of Student: Anjali Patel	Class: CSIT-1
Enrollment No: 0827CI201027	Batch 2020-2024

Date of Experiment	Date of Submission	Submitted on:

Remarks by faculty:	Grade:
Signature of student:	Signature of Faculty:

OBJECTIVE OF THE EXPERIMENT
To implement page replacement algorithm LRU.

FACILITIES REQUIRED

a) Facilities Required To Do The Experiment:

S.NO	FACILITIES REQUIRED	QUANTITY
1	**System**	**1**
2	**Windows XP/7**	

b) Concept of LRU Algorithm:

Pages that have been heavily used in the last few instructions will probably be heavily used againin the next few. Conversely, pages that have not been used for ages will probably remain unused for a long time. when a page fault occurs, throw out the page that has been unused for the longesttime. This strategy is called LRU (Least Recently Used) paging.

Page reference stream:

1	2	3	2	1	5	2	1	6	2	5	6	3	1	3	6	1	2	4	3
1	1	1	1	3	2	1	5	2	1	6	2	5	6	6	1	3	6	1	2
	2	2	3	2	1	5	2	1	6	2	5	6	3	1	3	6	1	2	4
		3	2	1	5	2	1	6	2	5	6	3	1	3	6	1	2	4	3
*	*	*			*			*		*		*	*				*	*	*

LRU
Total 11 page faults

c) **Algorithm:**

Step 1: Create a queue to hold all pages in memory

Step 2: When the page is required replace the page at the head of the

queueStep 3: Now the new page is inserted at the tail of the queue

Step 4: Create a stack

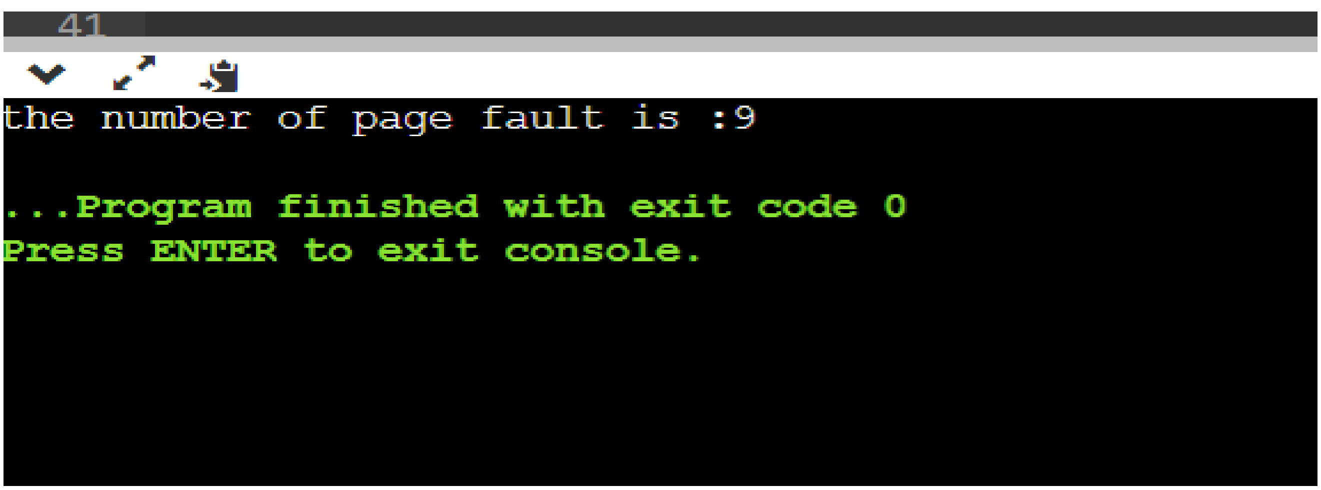
Step 5: When the page fault occurs replace page present at the bottom of the stack

d) **Program:**

```
#include<bits/stdc++.h>
using namespace std;


int pageFault(int page[],int n,int no_frame)
{
   int pagefault=0;
   vector<int> v1;
   int i;
   int j=0;
   for(i=0;i<n;i++)
   {
      auto it=find(v1.begin(),v1.end(),page[i]);
      if(it==v1.end())
      {
         if(v1.size()==no_frame)
         {
         v1.erase(v1.begin()+(j));
         }
         v1.push_back(page[i]);
         j++;
         j=j%no_frame;
         pagefault++;
      }
```

```
        else
        {
            v1.erase(it);
            v1.push_back(page[i]);
        }
    }
    return pagefault;
}


int main()
{
    int page[]={7,0,1,2,0,3,0,4,2,3,0,3,2};    //page refrence string
    int n=13;
    int no_frame=4;

    cout<<"the number of page fault is"<<pageFault(page,n,no_frame);



    return 0;
}
```

e) OUTPUT:



f) Result:

No. of pages faults :- …9….

Experiment-9

FCFS Disk Scheduling Algorithm

Name of Student: Anjali Patel		Class: CSIT-1
Enrollment No: 0827CI201027		Batch 2020-2024
Date of Experiment	Date of Submission	Submitted on:
Remarks by faculty:		Grade:
Signature of student:		Signature of Faculty:

OBJECTIVE OF THE EXPERIMENT

To implement FCFS Disk Scheduling Algorithm

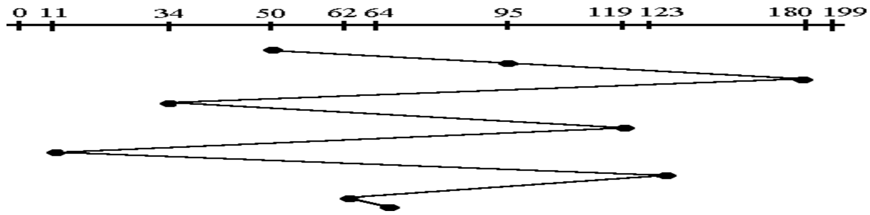
FACILITIES REQUIRED

a) **Facilities Required To Do The Experiment:**

S.NO	FACILITIES REQUIRED	QUANTITY
1	System	1
2	Windows XP/7	

b) **Concept of FCFS Disk Scheduling Algorithm:**

All incoming requests are placed at the end of the queue. Whatever number that is next inthe queue will be the next number served. Using this algorithm doesn't provide the best results. To determine the number of head movements you would simply find the number of tracks it took to move from one request to the next. For this case it went from 50 to 95 to 180 and so on. From 50 to 95 it moved 45 tracks. If you tally up the total number of tracks you will find how many tracks it had to go through before finishing the entire request. In this example, it had a total head movement of 640 tracks. The disadvantage of this algorithm is noted by the oscillation from track 50 to track 180 and then back to track11 to 123 then to 64. As you will soon see, this is the worse algorithm that one can use.



c) Algorithm:

Step 1: Create a queue to hold all requests in disk

Step 2: Move the head to the request in FIFO order (Serve the request first that came

first)Step 3: Calculate the total head movement required to serve all request.

d) Program:

```
#include<iostream>

using namespace std;

float FIFO(int req[],int pos,int n)
{
   float total=0;
   for(int i=0;i<n;i++)
   {
      total+=abs(pos-req[i]);
      pos=req[i];
   }

   float avg=total/n;
   return avg;
}


int main()
{
```

```
   int n,pos;
  cout<<"Enter the number of requests:";
  cin>>n;
 int req[n];
 cout<<"Enter the requests:";
 for(int i=0;i<n;i++)
 {
     cin>>req[i];
 }
 cout<<"Enter the positon where initially circular arm is present:";
 cin>>pos;
 cout<<FIFO(req,pos,n);
   return 0;
}
```

e) Output:

```
Enter the number of requests:8
Enter the requests:98
183
41
122
14
124
65
67
Enter the positon where initially circular arm is present:53
The average seek time is :79

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

f) Result:

Total Head Movement Required Serving All Requests :-…7……

Experiment-10

SSTF Disk Scheduling Algorithm

Name of Student: Anjali Patel		Class: CSIT-1
Enrollment No: 0827CI201027		Batch 2020-2024
Date of Experiment	Date of Submission	Submitted on:
Remarks by faculty:	Grade:	
Signature of student:	Signature of Faculty:	

OBJECTIVE OF THE EXPERIMENT

To implement SSTF Disk Scheduling Algorithm

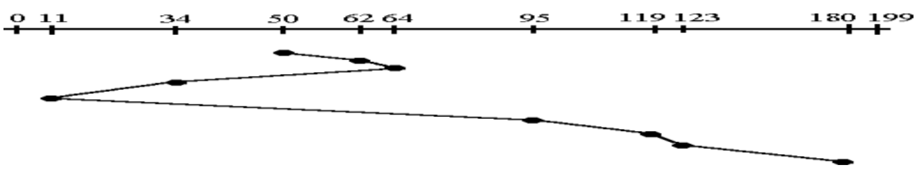
FACILITIES REQUIRED

a) **Facilities Required To Do The Experiment:**

S.NO	FACILITIES REQUIRED	QUANTITY
1	System	1
2	Windows XP/7	

b) **Concept of SSTF Disk Scheduling Algorithm:**

In this case request is serviced according to next shortest distance. Starting at 50, the nextshortest distance would be 62 instead of 34 since it is only 12 tracks away from 62 and 16tracks away from 34. The process would continue until all the process are taken care of. For example the next case would be to move from 62 to 64 instead of 34 since there are only 2 tracks between them and not 18 if it were to go the other way. Although this seemsto be a better service being that it moved a total of 236 tracks, this is not an optimal one. There is a great chance that starvation would take place. The reason for this is if there were a lot of requests close to each other the other requests will never be handled sincethe distance will always be greater.



c) Algorithm:

Step 1: Create a queue to hold all requests in disk

Step 2: Calculate the shortest seek time every time before moving head from current headposition

Step 3: Calculate the total head movement required to serve all request.

d) Program:

```
#include <bits/stdc++.h>

using namespace std;

// vector<int>:: iterator it;

int minDiff(int *req,int pos,int n)
{
int newpos;
int mini=INT_MAX;
int diff;
for(int i=0;i<n;i++)
{
if(req[i]!=-1)
{
diff=abs(pos-req[i]);
if(mini>diff)
{
mini=diff;
newpos=i;
}
}
}
//  cout<<"request choosen :"<<req[newpos];
return newpos;
}


float SSTF(int *req,int pos,int n)
{

// sort(req.begin(),req.end());
/* vector<int> :: iterator it;
while(!req.empty())
{
it=req.find(req.begin(),req.end());
int temp=req[it];
```

```
if(abs(req[it]-req[it-1])<(req[it+1]-req[it]))
{

total+=abs(req[it]-req[it-1]);
req.erase(req.begin()+it);

}
else
{
total+=abs(req[it+1]-req[it]);
req.erase(req.begin()+it);
}
}*/
int posi=pos;
float total=0;

for(int i=0;i<n;i++)
{
int index=minDiff(req,posi,n);
// cout<<"diff: "<<abs(pos-req[index]);
total+=abs(posi-req[index]);
posi=req[index];
req[index]=-1;

}

float avg=total/n;
return avg;

}

int main()
{
int n,positom;
cout<<"Enter the number of requests:";
cin>>n;

int req[n];
cout<<"Enter the requests:";
for(int i=0;i<n;i++)
{
// int r;
cin>>req[i];
//  req.push_back(r);
}
cout<<"Enter the positon where initially circular arm is present:";
cin>>positom;
cout<<SSTF(req,positom,n);
return 0;
```

e) **Output:**

```
Enter the number of requests:9
Enter the requests:55
58
39
18
90
160
150
38
184
Enter the positon where initially circular arm is present:100
The average seek time is :27.5556

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

f) **Result:**

Total Head Movement Required Serving All Requests :- …….