ACROPOLIS INSTITUTE OF TECHNOLOGY & RESEARCH, INDORE

ACROPOLIS
Enlightening Wisdom

PRACTICAL FILE OF OPERATING SYSTEM

Bachelor of Technology: III Year

Department of Computer Science & Information Technology

Name of the Student :- ABHIDATT SHARMA

Branch & section :- CSIT-1

Roll No. :- 0827CI201005

Year :- III

ACROPOLIS INSTITUTE OF TECHNOLOGY & RESEARCH, INDORE

Department of Computer Science & Information Technology

Certificate

This is to certify that the experimental work entered in this journal as per the BTech III year syllabus prescribed by the RGPV was done by Mr. / Ms. ………ABHIDATT SHARMA……… in …5… semester in the Laboratory of this institute during the academic year July 2022 - Dec 2022

Signature of Head   Signature of the Faculty

SYLLABUS

CSIT-502 – Operating System

Branch: Computer Science Information Technology V Semester

Course: CSIT 502 Operating System

Unit I

Introduction to Operating Systems, Evaluation of OS, Types of operating Systems, system protection, Operating system services, Operating System structure, System Calls and System Boots, Operating System design and implementation, Spooling and Buffering.

Unit II

Basic concepts of CPU scheduling, Scheduling criteria, Scheduling algorithms, algorithm evaluation, multiple processor scheduling. Process concept, operations on processes, threads, inter process communication, precedence graphs, critical section problem, semaphores, classical problems of synchronization,

Unit III

Deadlock problem, deadlock characterization, deadlock prevention, deadlock avoidance, deadlock detection, recovery from deadlock, Methods for deadlock handling. Concepts of memory management, logical and physical address space, swapping, Fixed and Dynamic Partitions, Best-Fit, First-Fit and Worst Fit Allocation, paging, segmentation, and paging combined with segmentation.

Unit IV

Concepts of virtual memory, Cache Memory Organization, demand paging, page replacement algorithms, allocation of frames, thrashing, demand segmentation, Role of Operating System in Security, Security Breaches, System Protection, and Password Managment.

Unit V

Disk scheduling, file concepts, File manager, File organization, access methods, allocation methods, free space managements, directory systems, file protection, file organization & access mechanism, file sharing implement issue, File Management in Linux, introduction to distributed systems.

ABHIDATT SHARMA

HARDWARE REQUIREMENTS:

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

SOFTWARE REQUIREMENTS:

Linux: Ubuntu / OpenSUSE / Fedora / Red Hat / Debian / Mint OS
WINDOWS: 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 OperatingSystem and their functionality.

Lab Plan

Operating System

CSIT-502

S.No	Name of Experiment	Page No.
1.	Program to implement FCFS scheduling	
2.	Program to implement SJF scheduling	
3.	Program to implement SRTF scheduling	
4.	Program to implement Round Robin scheduling	
5.	Program to implement Priority scheduling	
6.	Program to implement Banker's algorithm	
7.	Program to implement FIFO page replacement algorithm.	
8.	Program to implement LRU page replacement algorithm	
9	Program to implement Disk Scheduling (FIFO)algorithm	
10	Program to implement Disk Scheduling (SSTF)algorithm	

ACROPOLIS INSTITUTE OF TECHNOLOGY & RESEARCH, INDORE

Computer Science and Information Technology

Operating System

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	3		
2.		Program to implement SJF scheduling	8		
3.		Program to implement SRTF scheduling	13		
4.		Program to implement Round Robin scheduling	18		
5.		Program to implement Priority scheduling	22		
6.		Program to implement Banker's algorithm	26		
7.		Program to implement FIFO page replacement algorithm.	30		
8.		Program to implement LRU page replacement algorithm	33		
9.		Program to implement Disk Scheduling(FIFO)algorithm	36		
10.		Program to implement Disk Scheduling(SSTF)algorithm	39		

Experiment-1

FCFS SCHEDULING

Name of Student: ABHIDATT SHARMA		Class: CSIT-1
Enrollment No:0827CI201005		Batch: 2020-24
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

Facilities Required Doing The Experiment:

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

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

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: 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

- Waiting time for process(n)= waiting time of process (n-1) + Burst time of process(n-1
- Turn around time for Process(n)= waiting time of Process(n)+ Burst time forprocess(n)

Step 6: Calculate

- Average waiting time = Total waiting Time / Number of process
- Average Turnaround time = Total Turnaround Time / Number of process

Step 7: Stop the process

Program:

```
#include <iostream>
#include <bits/stdc++.h>
using namespace std;
class fcfs
{
public:
      int pid;int at;int bt;int ct;int wt;int tat;
};
bool cmp(fcfs x, fcfs y)
{
    return x.at < y.at;
}
void waiting_time(fcfs a[], int n)
{
    for (int i = 0; i < n; i++)
    {
        a[i].wt = a[i].tat - a[i].bt;
    }
    cout << "Waiting time:" << endl;
    for (int i = 0; i < n; i++)
    {
        cout << a[i].wt << " ";
    }
}
void turnaround_time(fcfs a[], int n)
{
    for (int i = 0; i < n; i++)
    {
        a[i].tat = a[i].ct - a[i].at;
    }
    cout << "Turnaround time\n";
    for (int i = 0; i < n; i++)
    {
        cout << a[i].tat << " ";
    }
    cout << endl;
}
void cmp_time(fcfs a[], int n)
{
```

```
    a[0].ct = a[0].bt;
    for (int i = 1; i < n; i++)
    {
        a[i].ct = a[i - 1].ct + a[i].bt;
    }
    cout << "completion time\n";
    for (int i = 0; i < n; i++)
    {
        cout << a[i].ct << " ";
    }
    cout << endl;
}
int main()
{
    int n;
    cout << "Enter the no. of process:";
    cin >> n;
    fcfs a[n];
    for (int i = 0; i < n; i++)
    {
        a[i].pid = i + 1;
        cout << "Enter arrival time for " << i + 1 << " process:";
        cin >> a[i].at;
        cout << "Enter burst time for " << i + 1 << " process:";
        cin >> a[i].bt;
    }
    sort(a, a + n, cmp);
    cmp_time(a, n);
    turnaround_time(a, n);
    waiting_time(a, n);
}
```

Output:

```
PS D:\OS lab> & .\"FCFS.exe"
Enter the no. of process:4
Enter arrival time for 1 process:0
Enter burst time for 1 process:5
Enter arrival time for 2 process:1
Enter burst time for 2 process:3
Enter arrival time for 3 process:2
Enter burst time for 3 process:8
Enter arrival time for 4 process:3
Enter burst time for 4 process:6
completion time
5 8 16 22
Turnaround time
5 7 14 19
Waiting time:
0 4 6 13
```

Result:

Average Waiting Time : 13.25

Average Turnaround Time: 30.75

Experiment-2

SJF Scheduling

Name of Student: ABHIDATT SHARMA		Class: CSIT-1
Enrollment No: 0827CI201005		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

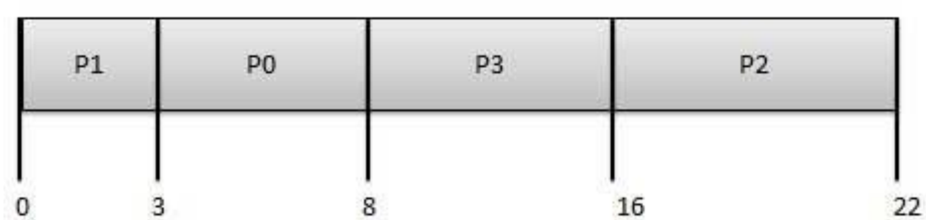
Facilities Required Doing The Experiment:

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

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



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: For each process in the ready queue, calculate

- Waiting time for process(n)= waiting time of process (n-1) + Burst time of process(n-1)
- Turnaround time for Process(n)= waiting time of Process(n)+ Burst time forprocess(n)

Step 7: Calculate

- Average waiting time = Total waiting Time / Number of process
- Average Turnaround time = Total Turnaround Time / Number of process

Step 8: Stop the process

Program:

```
#include<iostream>
using namespace std;
void sort(int process[],int n,int a[],int b[])
{
int temp;
for(int i=0;i<n;i++)
{
for(int j=i+1;j<n;j++)
{
if(b[i]>b[j])
{
temp=a[i];
a[i]=a[j];
a[j]=temp;

temp=b[i];
b[i]=b[j];
b[j]=temp;
}}}}
int main()
{
int n,temp,tt=0,min,d,i,j;
float atat=0,awt=0,stat=0,swt=0;

cout<<"enter no of process"<<endl;
cin>>n;
int process[n],a[n],b[n],e[n],tat[n],wt[n];
for(i=0;i<n;i++)
{
cin>>process[i];
}
cout<<"enter arival time ";
for(i=0;i<n;i++)
{
cin>>a[i];
}
cout<<"enter brust time ";
for(i=0;i<n;i++)
```

```
{
cin>>b[i];
}
sort(process,n,a,b);
min=a[0];
for(i=0;i<n;i++)
{
if(min>a[i])
{
min=a[i];
d=i;
}
}
tt=min;
e[d]=tt+b[d];
tt=e[d];

for(i=0;i<n;i++)
{
if(a[i]!=min)
{
e[i]=b[i]+tt;
tt=e[i];
}
}
for(i=0;i<n;i++)
{
tat[i]=e[i]-a[i];
stat=stat+tat[i];
wt[i]=tat[i]-b[i];
swt=swt+wt[i];
}
atat=stat/n;
awt=swt/n;
cout<<"Process  Arrival-time(s)  Burst-time(s)  Waiting-time(s)  Turnaround-time(s)\n";
for(i=0;i<n;i++)
{
cout<<process[i]<<"            "<<a[i]<<"             "<<b[i]<<"               "<<wt[i]<<"
    "<<tat[i]<<endl;
```

```
}
cout<<"Average Waiting Time:"<<awt<<endl;
cout<<"Average TurnAround Time"<<atat;
}
```

Output:

```
enter no of process
5
1
2
3
4
5
enter arival time 1
2 3
3
2
enter brust time 2
1
8
9
6
Process  Arrival-time(s)  Burst-time(s)  Waiting-time(s)  Turnaround-time(s)
1            2                1                1               2
2            1                2                0               2
3            2                6                2               8
4            3                8                7               15
5            3                9                15               24
Average Waiting Time:5
Average TurnAround Time10.2
```

Result:

Average Waiting Time :5

Average Turnaround Time : 10.2

Experiment-3

SRTF Scheduling

<table>
<tr><td colspan="2">Name of Student: ABHIDATT SHARMA</td><td colspan="2">Class: CSIT-1</td></tr>
<tr><td colspan="2">Enrollment No:0827CI201005</td><td colspan="2">Batch: 2020-2024</td></tr>
<tr><td>Date of Experiment</td><td>Date of Submission</td><td colspan="2">Submitted on:</td></tr>
<tr><td colspan="2">Remarks by faculty:</td><td colspan="2">Grade:</td></tr>
<tr><td>Signature of student:</td><td colspan="3">Signature of Faculty:</td></tr>
</table>

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

FACILITIES REQUIRED
Facilities Required To Do The Experiment:

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

Concept Of SRTF Scheduling:

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.
- Two schemes:
- 1. non pre- emptive – once CPU given to the process it cannot be preempted until completes 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

Process	Arrival Time	Burst Time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

SJF (preemptive)

P1	P2	P3	P2	P4	P1

0 2 4 5 7 11 16

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: 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 to

Highest 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

- Average waiting time = Total waiting Time / Number of process
- Average Turnaround time = Total Turnaround Time / Number of process

Step 8: Stop the process

Program:

```
#include <bits/stdc++.h>
using namespace std;
struct Process {
int pid; // Process ID
int bt; // Burst Time
int art; // Arrival Time
};
void findWaitingTime(Process proc[], int n,
int wt[])
{
int rt[n];

for (int i = 0; i < n; i++)
rt[i] = proc[i].bt;
int complete = 0, t = 0, minm = INT_MAX;
int shortest = 0, finish_time;
bool check = false;
while (complete != n) {
for (int j = 0; j < n; j++) {
if ((proc[j].art <= t) &&
(rt[j] < minm) && rt[j] > 0) {
minm = rt[j];
shortest = j;
check = true;
}
}
if (check == false) {
t++;
continue;
}
rt[shortest]--;
minm = rt[shortest];
if (minm == 0)
minm = INT_MAX;
if (rt[shortest] == 0) {
complete++;
check = false;
finish_time = t + 1;
wt[shortest] = finish_time -
proc[shortest].bt -
proc[shortest].art;
if (wt[shortest] < 0)
wt[shortest] = 0;
}
t++;
```

```
}
}
void findTurnAroundTime(Process proc[], int n,
int wt[], int tat[])
{
for (int i = 0; i < n; i++)
tat[i] = proc[i].bt + wt[i];
}
void findavgTime(Process proc[], int n)
{
int wt[n], tat[n], total_wt = 0,
total_tat = 0;
findWaitingTime(proc, n, wt);
findTurnAroundTime(proc, n, wt, tat);
cout << " P\t\t"
<< "BT\t\t"
<< "WT\t\t"
<< "TAT\t\t\n";
for (int i = 0; i < n; i++) {
total_wt = total_wt + wt[i];
total_tat = total_tat + tat[i];
cout << " " << proc[i].pid << "\t\t"
<< proc[i].bt << "\t\t " << wt[i]
<< "\t\t " << tat[i] << endl;
}
cout << "\nAverage waiting time = "
<< (float)total_wt / (float)n;
cout << "\nAverage turn around time = "
<< (float)total_tat / (float)n;
}
int main()
{
Process proc[] = { { 1, 6, 2 }, { 2, 2, 5 },
{ 3, 8, 1 }, { 4, 3, 0}, {5, 4, 4} };
int n = sizeof(proc) / sizeof(proc[0]);
findavgTime(proc, n);
return 0;
}
```

Output:

```
C:\WINDOWS\system32\cmd.exe
 P              BT              WT              TAT
 1              6                7               13
 2              2                0               2
 3              8                14              22
 4              3                0               3
 5              4                2               6

Average waiting time = 4.6
Average turn around time = 9.2
Press any key to continue . . .
```

Result:

Average Waiting Time …4.6…..

Average Turnaround Time …9.2…….

Experiment-4

ROUND ROBIN Scheduling

<table>
<tr><td colspan="2">Name of Student: ABHIDATT SHARMA</td><td>Class: CSIT-1</td></tr>
<tr><td colspan="2">Enrollment No: 0827CI201005</td><td>Batch: 2020-2024</td></tr>
<tr><td>Date of Experiment</td><td>Date of Submission</td><td>Submitted on:</td></tr>
<tr><td colspan="2">Remarks by faculty:</td><td>Grade:</td></tr>
<tr><td>Signature of student:</td><td colspan="2">Signature of Faculty:</td></tr>
</table>

OBJECTIVE OF THE EXPERIMENT

To write c program to implement Round Robin scheduling.

FACILITIES REQUIRED

Facilities Required To Do The Experiment:

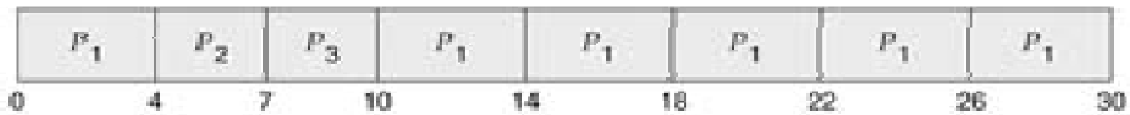
S.NO	FACILITIES REQUIRED	QUANTITY
1	System	1
2	Windows XP/7	

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. If the 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.

Process	Burst Time
P_1	24
P_2	3

P_3	3
Average	



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

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

Step 7: Calculate

- Average waiting time = Total waiting Time / Number of process
- Average Turnaround time = Total Turnaround Time / Number of process

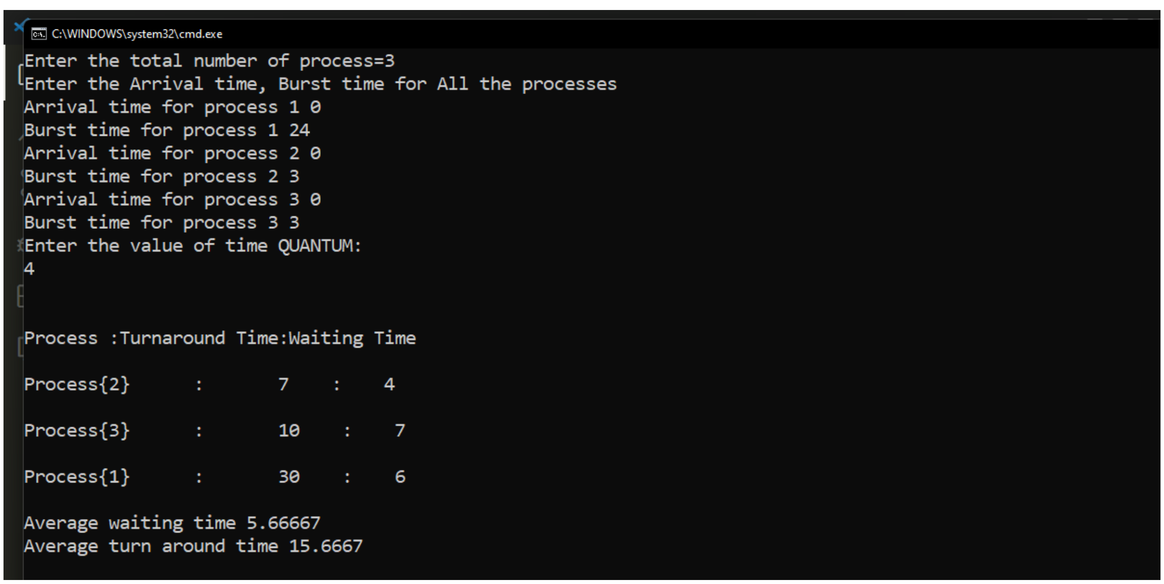
Step 8: Stop the process

Program:

```
#include <iostream>
using namespace std;
int main()
{
    int i, n, time, remain, temps = 0, time_quantum;
    int wt = 0, tat = 0;
    cout << "Enter the total number of process=";
    cin >> n;

    remain = n;
    int at[n];
    int bt[n];
    int rt[n];
    cout << "Enter the Arrival time, Burst time for All the processes" << endl;
    for (i = 0; i < n; i++)
    {
        cout << "Arrival time for process " << i + 1 << " ";
        cin >> at[i];
        cout << "Burst time for process " << i + 1 << " ";
        cin >> bt[i];
        rt[i] = bt[i];
    }
    cout << "Enter the value of time QUANTUM:" << endl;
    cin >> time_quantum;
    cout << "\n\nProcess\t:Turnaround Time:Waiting Time\n\n";
    for (time = 0, i = 0; remain != 0;)
    {
        if (rt[i] <= time_quantum && rt[i] > 0)
        {
            time += rt[i];
            rt[i] = 0;
            temps = 1;
        }
        else if (rt[i] > 0)
        {
            rt[i] -= time_quantum;
            time += time_quantum;
        }
        if (rt[i] == 0 && temps == 1)
        {
            remain--;
```

```
            cout << "Process{" << i + 1 << "}"
                << "\t:\t" << time - at[i] << "    :    " << time - at[i] - bt[i] << "\n";
            cout << endl;
            wt += time - at[i] - bt[i];
            tat += time - at[i];
            temps = 0;
        }
        if (i == n - 1)
            i = 0;
        else if (at[i + 1] <= time)
            i++;
        else
            i = 0;
    }
    cout << "Average waiting time " << wt * 1.0 / n << endl;
    cout << "Average turn around time " << tat * 1.0 / n << endl;

    return 0;
}
```

Output:



```
C:\WINDOWS\system32\cmd.exe
Enter the total number of process=3
Enter the Arrival time, Burst time for All the processes
Arrival time for process 1 0
Burst time for process 1 24
Arrival time for process 2 0
Burst time for process 2 3
Arrival time for process 3 0
Burst time for process 3 3
Enter the value of time QUANTUM:
4


Process :Turnaround Time:Waiting Time

Process{2}      :       7    :    4

Process{3}      :       10    :    7

Process{1}      :       30    :    6

Average waiting time 5.66667
Average turn around time 15.6667
```

Result:

Average Waiting Time:5.66667
Average Turnaround Time :15.6667

Experiment-5

PRIORITY SCHEDULING

Name of Student: ABHIDATT SHARMA		Class: CSIT-1
Enrollment No: 0827CI201005		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

Facilities Required To Do The Experiment:

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

Concept Of Priority Scheduling:

A priority is associated with each process, and the CPU is allocated to the process with the highest 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 the order

Process	Burst Time	Priority	Waiting Time	Turnaround 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

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 turn around time

Step 6: For each process in the Ready Q calculate

- Waiting time for process(n)= waiting time of process (n-1) + Burst time of process(n-1)
- Turn around time for Process(n)= waiting time of Process(n)+ Burst time forprocess(n)

Step 7: Calculate

- Average waiting time = Total waiting Time / Number of process
- Average Turnaround time = Total Turnaround Time / Number of process

Step 8: Stop the process

Program:

```
#include<iostream>
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<<"\nP["<<i+1<<"]\n";
cout<<"Burst Time:";
cin>>bt[i];
cout<<"Priority:";
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;      //average waiting time
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;
}
```

Output:

```
C:\WINDOWS\system32\cmd.exe
Enter Total Number of Process:4

Enter Burst Time and Priority

P[1]
Burst Time:10
Priority:3

P[2]
Burst Time:1
Priority:1

P[3]
Burst Time:2
Priority:4

P[4]
Burst Time:1
Priority:5

Process     Burst Time          Waiting Time    Turnaround Time
P[2]              1                   0                 1
P[1]              10                  1                 11
P[3]              2                   11                13
P[4]              1                   13                14

Average Waiting Time=6
Average Turnaround Time=9
```

Result:

Average Waiting Time …6…..

Average Turnaround Time …9…….

Experiment-6

BANKER ALGORITHM

Name of Student: ABHIDATT SHARMA			Class: CSIT-1
Enrollment No:0827CI201005			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

Facilities Required To Do The Experiment:

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

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 amountsof 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.

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.

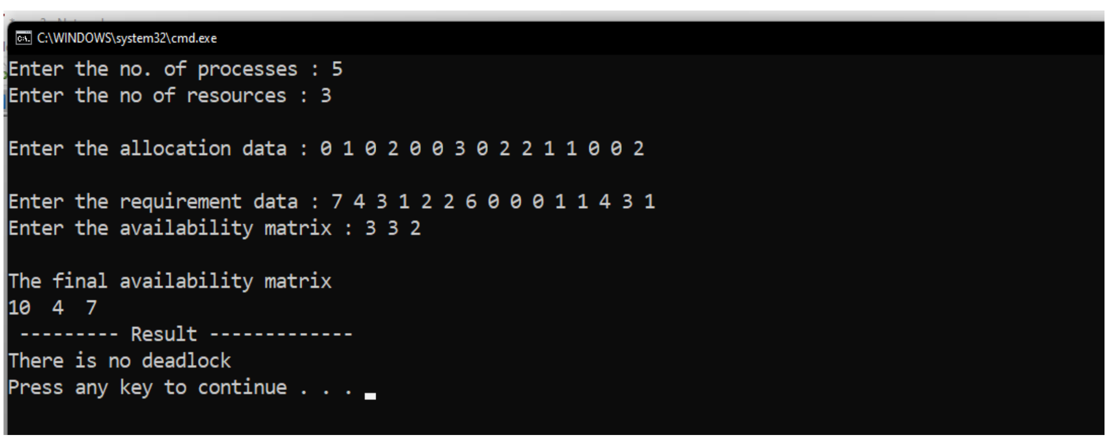
Program:

```
#include <bits/stdc++.h>
using namespace std;
int alloc[50][50];
int maxi[50][50];
int need[50][50];
int avail[50];
class Test
{
public:
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()
{
        Test t1;
        int np = 100;
        int nr = 100;
        cout << "Enter the no. of processes : ";
        cin >> np;
        cout << "Enter the no of resources : ";
        cin >> nr;
        cout << "\nEnter the allocation data : ";
        for (int i = 0; i < np; i++)
              for (int j = 0; j < nr; j++)
                        cin >> alloc[i][j];
        cout << "\nEnter the requirement data : ";
        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 << "Enter the availability matrix :";
        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] && t1.check_safety(i, nr))
          {
                        for (int j = 0; j < nr; j++)
                                    avail[j] += alloc[i][j];
```

```
      }
      completed[i] = true;
    }
    flg = t1.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;
}
```

Output:



Result:

The Sequence Is: P1 -> P3 -> P4 -> P0 -> P2

Experiment-7

FIFO PAGE REPLACEMENT

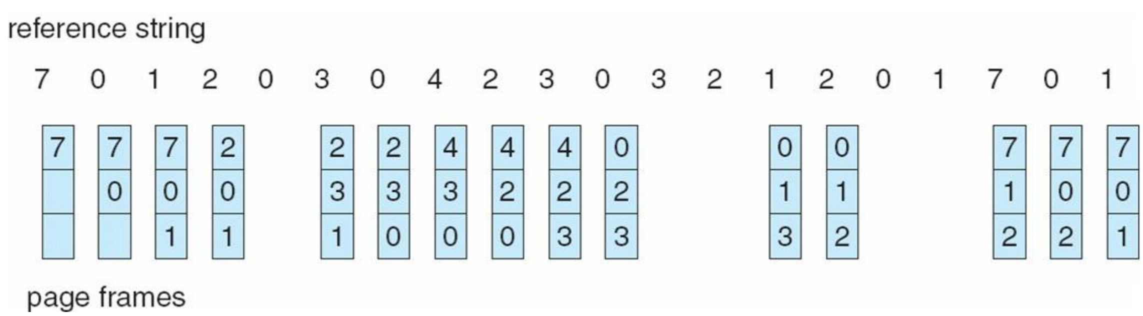
Name of Student: ABHIDATT SHARMA		Class: CSIT-1
Enrollment No:0827CI201005		Batch: 2020-2024
Date of Experiment	Date of Submission	Submitted on:
Remarks by faculty:		Grade:
Signature of student:		Signature of Faculty:

OBJECTIVE OF THE EXPERIMENT
To implement page replacement algorithm FIFO.

FACILITIES REQUIRED
Facilities Required To Do The Experiment:

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

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
.



- FIFO Replacement manifests Belady's Anomaly:

  more frames $\Rightarrow$ 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

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 queue

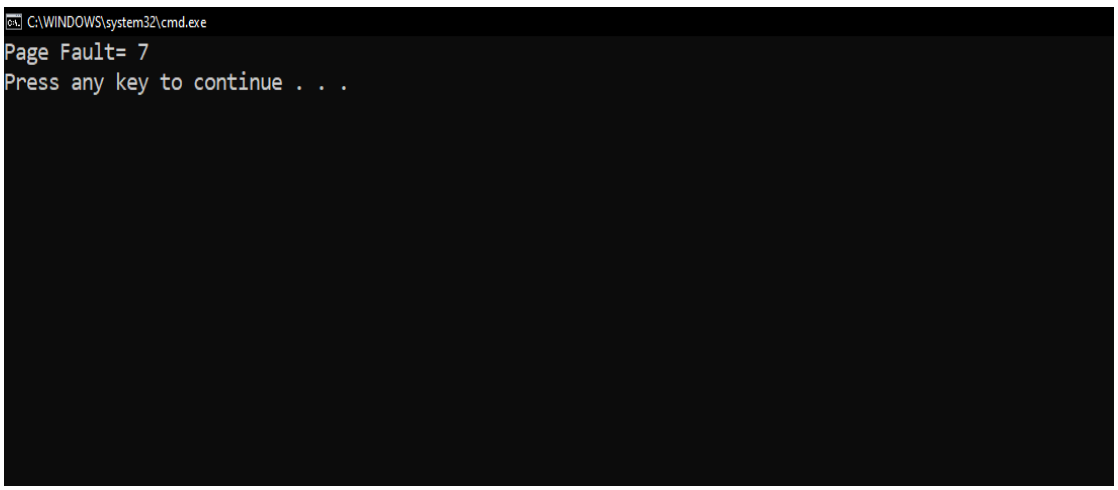
Step 3: Now the new page is inserted at the tail of the queue

Program:

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

int pageFaults(int pages[], int n, int c)
{

  unordered_set<int> v;
  queue<int> indexes;

  int page_faults = 0;
  for (int i=0; i<n; i++)
  {

  if (v.size() < c)
  {
     if (v.find(pages[i])==v.end())
     {
           v.insert(pages[i]);
            page_faults++;
           indexes.push(pages[i]);
     }
  }
  else
  {
    if (v.find(pages[i]) == v.end())
   {
        int val = indexes.front();
        indexes.pop();
        v.erase(val);
        v.insert(pages[i]);
        indexes.push(pages[i]);
        page_faults++;
    }
}
```

```
    }

    return page_faults;
  }



  int main()
  {
    int pages[] = {7, 0, 1, 2, 0, 3, 0, 4,2, 3, 0, 3, 2};
    int n = 13;
    int capacity = 4;
    cout <<"Page Fault= "<< pageFaults(pages, n, capacity);
    return 0;
  }
```

Output:



Result:

No. of page faults: 7

Experiment-8

LRU PAGE REPLACEMENT

Name of Student: ABHIDATT SHARMA		Class: CSIT-1
Enrollment No: 0827CI201005		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

Facilities Required To Do The Experiment:

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

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
```

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 queue

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

Step 4: Create a stack

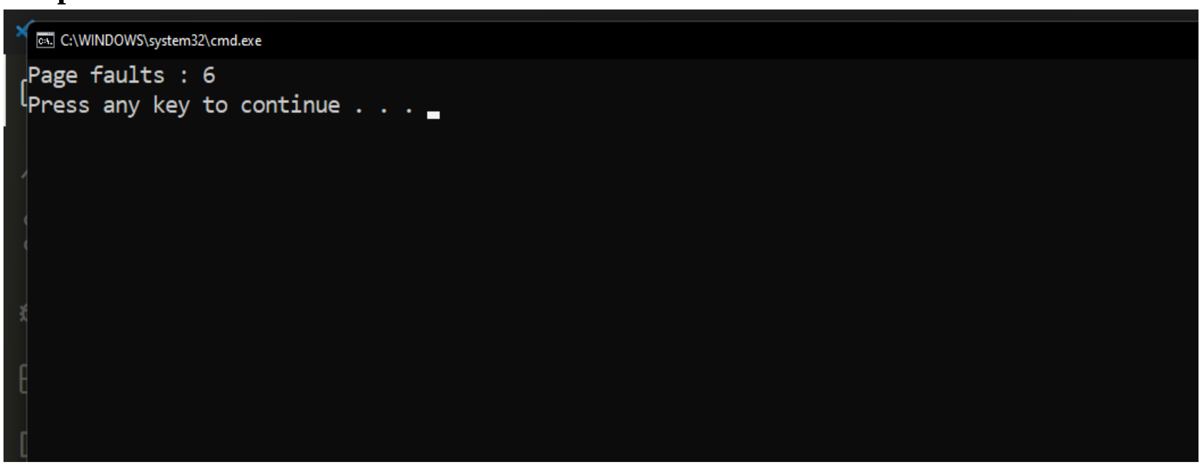
Step 5: When the page fault occurs replace page present at the bottom of the stack

Program:

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

int main()
{
  int capacity = 4;
  int arr[] = {1,2,3,2,1,5,2,1,6,2,5,6,3,1,3,6,1,2,4,3};

  deque<int> q(capacity);
  int count=0;
  int page_faults=0;
  deque<int>::iterator itr;
  q.clear();
  for(int i:arr)
  {
    itr = find(q.begin(),q.end(),i);
    if(!(itr != q.end()))
    {
      ++page_faults;
      if(q.size() == capacity)
      {
        q.erase(q.begin());
        q.push_back(i);
      }
      else{
        q.push_back(i);

      }
    }
```

```
    else
    {
      q.erase(itr);
      q.push_back(i);
    }
  }
  cout<<"Page faults : "<<page_faults;
}
```

Output:



Result:

No. of pages faults :6

Experiment-9

FCFS Disk Scheduling Algorithm

Name of Student: ABHIDATT SHARMA			Class: CSIT-1
Enrollment No:0827CI201005			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

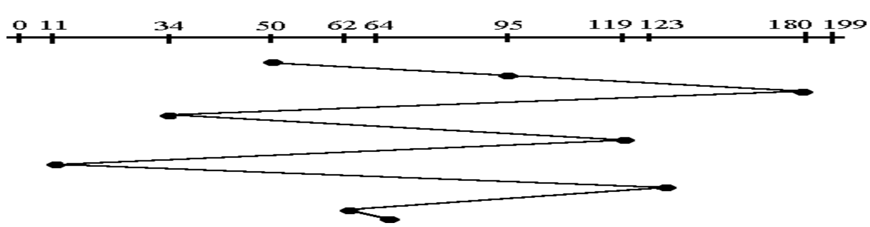
FACILITIES REQUIRED

Facilities Required To Do The Experiment:

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

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.



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.

Program:

```
#include<iostream>
using namespace std;
int main()
{
    int initial;
    int n;
    int seek;

    cout<<"enter the no. of process:";
    cin>>n;
    cout<<"enter intial value:";
     cin>>initial;
    int arr[n];
    for(int i=0;i<n;i++)
       cin>>arr[i];
    seek=abs(initial-arr[0]);
    for(int i=1;i<n;i++)
    {
        seek=seek+abs(arr[i-1]-arr[i]);
    }
    cout<<seek;
}
```

Output:

```
C:\WINDOWS\system32\cmd.exe
enter the no. of process:8
enter intial value:50
176 79 34 60 92 11 41 114
510
Press any key to continue . . .
```

Result:

Total Head Movement Required Serving All Requests …7……

Experiment-10

SSTF Disk Scheduling Algorithm

<table>
<tr><td colspan="3">Name of Student: ABHIDATT SHARMA</td><td colspan="2">Class: CSIT-1</td></tr>
<tr><td colspan="3">Enrollment No: 0827CI201005</td><td colspan="2">Batch: 2020-2024</td></tr>
<tr><td>Date of Experiment</td><td colspan="3">Date of Submission</td><td>Submitted on:</td></tr>
<tr><td colspan="2">Remarks by faculty:</td><td colspan="3">Grade:</td></tr>
<tr><td colspan="2">Signature of student:</td><td colspan="3">Signature of Faculty:</td></tr>
</table>

OBJECTIVE OF THE EXPERIMENT

To implement SSTF Disk Scheduling Algorithm

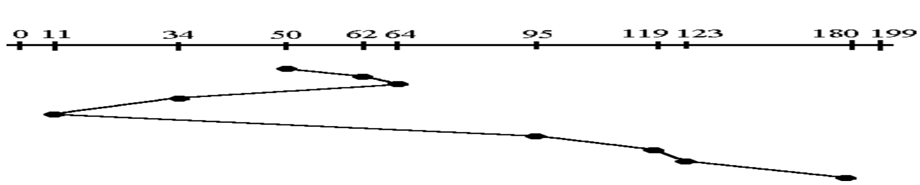
FACILITIES REQUIRED

Facilities Required To Do The Experiment:

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

Concept of SSTF Disk Scheduling Algorithm:

In this case request is serviced according to next shortest distance. Starting at 50, the next shortest distance would be 62 instead of 34 since it is only 12 tracks away from 62 and 16 tracks 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 seems to 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 since the distance will always be greater.



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 head position

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

Program:

```
#include <bits/stdc++.h>
using namespace std;
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;
                }
        }
    }
    return newpos;
}
float SSTF(int *req, int pos, int n)
{

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

Output:

```
C:\WINDOWS\system32\cmd.exe
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
27.5556
Press any key to continue . . .
```

Result:

Average seek time:27.5

FAQ’s

1. What are different types of schedulers?
2. Explain types of Operating System?
3. Explain performance criteria for the selection of schedulers?
4. Explain priority based preemptive scheduling algorithm?
5. What is thread?
6. Explain different types of thread?
7. What is kernel level thread?
8. What is user level thread?
9. What is memory management?
10. Explain Belady's Anomaly.
11. What is a binary semaphore? What is its use?
12. What is thrashing?
13. List the Coffman's conditions that lead to a deadlock.
14. What are turnaround time and response time?
15. What is the Translation Lookaside Buffer (TLB)?
16. When is a system in safe state?
17. What is busy waiting?
18. Explain the popular multiprocessor thread-scheduling strategies.
19. What are local and global page replacements?
20. In the context of memory management, what are placement and replacement algorithms?
21. In loading programs into memory, what is the difference between load-time dynamic linking and run-time dynamic linking?
22. What are demand- and pre-paging?
23. Paging a memory management function, while multiprogramming a processor management functions, are the two interdependent?
24. What has triggered the need for multitasking in PCs?
25. What is SMP?
26. List out some reasons for process termination.
27. What are the reasons for process suspension?
28. What is process migration?
29. What is an idle thread?
30. What are the different operating systems?
31. What are the basic functions of an operating system?