



## **PRACTICAL FILE OF OPERATING SYSTEM BTech: III Year**

Department of Computer Science & Information Technology

**Name of the Student : Akshat Shrivastava**

**Branch & section : CSIT 1**

**Roll No. : 0827CI201020**

**Year : 2022**

**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 journal as per the BTech III year syllabus prescribed by the RGPV was done by Mr. Akshat Shrivastava in 5th semester in the Laboratory of this institute during the academic year 2022- 2023

Signature of Head

Signature of the Faculty

**ACROPOLIS INSTITUTE OF TECHNOLOGY & RESEARCH, INDORE**  
**GENERAL 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 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 Management.

#### *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.

### **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 Operating System and their functionality.

## **Lab Plan**

### **Operating System**

### **CS-502**

<b>S.No</b>	<b>Name of Experiment</b>	<b>Page No.</b>
1.	Program to implement FCFS scheduling	1
2.	Program to implement SJF scheduling	6
3.	Program to implement SRTF scheduling	11
4.	Program to implement Round Robin scheduling	16
5.	Program to implement Priority scheduling	22
6.	Program to implement Banker's algorithm	28
7.	Program to implement FIFO page replacement algorithm.	34
8.	Program to implement LRU page replacement algorithm	39
9	Program to implement Disk Scheduling(FIFO) algorithm	44
10	Program to implement Disk Scheduling(SSTF) algorithm	49

**ACROPOLIS INSTITUTE OF TECHNOLOGY & RESEARCH, INDORE**

**Name of Department: CSIT**

**Name of  
Laboratory: Operating System**

**Index**

S.No.	Date of Exp.	Name of the Experiment	Page No.	Date of Submission	Grade & Sign of the Faculty
1.		FCFS Scheduling			
2.		SJF Scheduling			
3.		SRTF Scheduling			
4.		ROUND ROBIN Scheduling			
5.		PRIORITY Scheduling			
6.		Banker's Algorithm			
7.		FIFO PAGE REPLACEMENT	34		
8.		LRU PAGE REPLACEMENT			
9.		FCFS Disk Scheduling Algorithm			
10.		SSTF Disk Scheduling Algorithm			
11.					

# Experiment-1

## FCFS SCHEDULING

Name of Student: Akshat Shrivastava		Class: CSIT-1
Enrollment No: 0827CI201020		Batch: B1
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 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

(a)  $\text{Waiting time for process}(n) = \text{waiting time of process } (n-1) + \text{Burst time of process}(n-1)$

(b)  $\text{Turn around time for Process}(n) = \text{waiting time of Process}(n) + \text{Burst time for process}(n)$

Step 6: Calculate

(a)  $\text{Average waiting time} = \text{Total waiting Time} / \text{Number of process}$

(b)  $\text{Average Turnaround time} = \text{Total Turnaround Time} /$

$\text{Number of process}$

Step 7: Stop the process

**d) Program:**

```
#include <iostream>
//First Come First Serve with zero arrival time .....
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];
```

```

// int arr[n]={0};
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;
}

#include <iostream>
//First Come First Serve with varying arrival time .....
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[],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:**

```
Enter the number of process:4
Enter the process ID and burst time:1
5
Enter the process ID and burst time:2
4
Enter the process ID and burst time:3
3
Enter the process ID and burst time:4
6
p_ID completion TAT Waiting
1      5          5      0
2      9          9      5
3     12         12      9
4     18         18     12
-----
Process exited after 24.37 seconds with return value 0
Press any key to continue . . .
```

```
Enter the number of process:3
Enter the process ID
1
Enter the process ID
2
Enter the process ID
3
Enter the arrival time and burst time of Process :1
14
12
Enter the arrival time and burst time of Process :2
13
15
Enter the arrival time and burst time of Process :3
16
19
p_ID   arrival   burst completion   TAT   Waiting
1      14        12        12        -2      -14
2      13        15        27        14       -1
3      16        19        46        30       11
-----
Process exited after 19.84 seconds with return value 0
Press any key to continue . . .
```

**b) Result:**

Average Waiting Time :6.5 units

Average Turnaround Time :11 units

## Experiment-2

### SJF Scheduling

Name of Student: Akshat Shrivastava		Class: CSIT-1
Enrollment No: 0827CI201020		Batch: B1
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

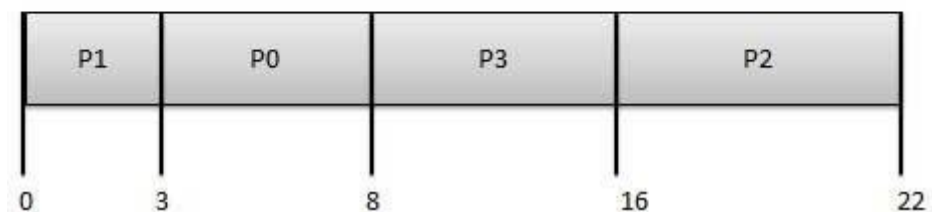
#### a) Facilities Required Doing The Experiment:

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 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 for 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

**d) Program:**

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

```

**a) Output:**

```
Enter the number of process:3
Enter the process ID and burst time (arrival time is zero) :1 23
Enter the process ID and burst time (arrival time is zero) :2 45
Enter the process ID and burst time (arrival time is zero) :3 11
p_ID    burst    TAT    Waiting
3        11       11       0
1        23       34       11
2        45       79       34
The average TAT is :41
The averageWaiting time is 15
-----
Process exited after 22.23 seconds with return value 0
Press any key to continue . . .
```

**b) Result:**

Average Waiting Time 15 units

Average Turnaround Time 41 units

## Experiment-3

### SRTF Scheduling

Name of Student: Akshat Shrivastava		Class: CSIT-1
Enrollment No: 0827CI201020		Batch: B1
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 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

**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 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 (BT<sub>n</sub>).

Step 7: If the burst time of newly arrived process (BT<sub>n</sub>) 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 process

Step 8: Stop the process

**d) Program:**

```
// C++ program to implement Shortest Remaining Time First
// Shortest Remaining Time First (SRTF)

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

struct Process {
    int pid; // Process ID
    int bt; // Burst Time
    int art; // Arrival Time
};

// Function to find the waiting time for all
// processes
void findWaitingTime(Process proc[], int n,
                     int wt[])
{
    int rt[n];

    // Copy the burst time into rt[]
    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;

    // Process until all processes gets
    // completed
    while (complete != n) {

        // Find process with minimum
        // remaining time among the
        // processes that arrives till the
        // current time`
        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 a process is found, then
        // decrease the remaining time by
        // one unit
        if (check) {
            rt[shortest]--;
            t++;
            if (rt[shortest] == 0)
                complete++;
            wt[shortest] = t - proc[shortest].art;
        }
    }
}
```

```

        if (check == false) {
            t++;
            continue;
        }

        // Reduce remaining time by one
        rt[shortest]--;

        // Update minimum
        minm = rt[shortest];
        if (minm == 0)
            minm = INT_MAX;

        // If a process gets completely
        // executed
        if (rt[shortest] == 0) {

            // Increment complete
            complete++;
            check = false;

            // Find finish time of current
            // process
            finish_time = t + 1;

            // Calculate waiting time
            wt[shortest] = finish_time -
                            proc[shortest].bt -
                            proc[shortest].art;

            if (wt[shortest] < 0)
                wt[shortest] = 0;
        }
        // Increment time
        t++;
    }
}

// Function to calculate turn around time
void findTurnAroundTime(Process proc[], int n,
                        int wt[], int tat[])
{
    // calculating turnaround time by adding
    // bt[i] + wt[i]
    for (int i = 0; i < n; i++)
        tat[i] = proc[i].bt + wt[i];
}

```

```

// Function to calculate average time
void findavgTime(Process proc[], int n)
{
    int wt[n], tat[n], total_wt = 0,
        total_tat = 0;

    // Function to find waiting time of all
    // processes
    findWaitingTime(proc, n, wt);

    // Function to find turn around time for
    // all processes
    findTurnAroundTime(proc, n, wt, tat);

    // Display processes along with all
    // details
    cout << " P\t\t"
        << "BT\t\t"
        << "WT\t\t"
        << "TAT\t\t\n";

    // Calculate total waiting time and
    // total turnaround time
    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;
}

// Driver code
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; }

```

**a) Output:**

```
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
-----
Process exited after 0.6151 seconds with return value
Press any key to continue . . .
```

**b) Result:**

Average Waiting Time 4.6 units

Average Turnaround Time 9.2 units



## Experiment-4

### ROUND ROBIN Scheduling

Name of Student: Akshat Shrivastava		Class: BE
Enrollment no.: 0827CI201020		Batch: B1
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:

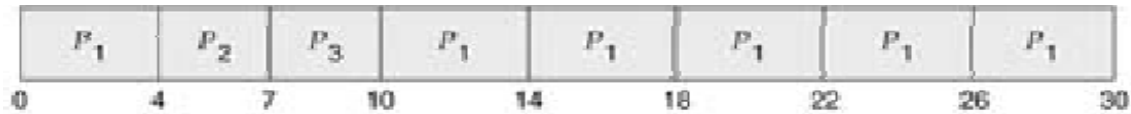
S.NO	FACILITIES REQUIRED	QUANTITY
1	System	1
2	Windows XP/7	

#### 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. 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.

	Burst
Process	Time
$P_1$	24
$P_2$	3

$P_2$	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 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

$$\text{No. of time slice for process}(n) = \text{burst time process}(n) / \text{time slice}$$

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 process

Step 8: Stop the process

**Program:**

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

    // Use for loop to enter the details of the process like Arrival time and the Burst Time
    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
    }
    // Accept the Time quant
    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], sum-at[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;
    }
}
// represents the average waiting time and Turn Around time
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();
}

```

**e. Output:**

```
Enter the Arrival and Burst time of the Process[1]
Arrival time is: 1
Burst time is: 2
Enter the Arrival and Burst time of the Process[2]
Arrival time is: 2
Burst time is: 5
Enter the Arrival and Burst time of the Process[3]
Arrival time is: 6
Burst time is: 2
Enter the Time Quantum for the process: 2
Process No      Burst Time      TAT      Waiting Time
Process No[1]   2              1
-1
Process No[3]   2              2
0
Process No[2]   5              7
2
Average Turn Around Time: 0.333333
Average Waiting Time: 3.333333
```

**d) Result:**

Average Waiting Time 3.333

Average Turnaround Time 0.333

## Experiment-5

### PRIORITY SCHEDULING

Name of Student: Akshat Shrivastava		Class:CSIT-1
Enrollment no.: 0827CI201020		Batch: B1
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:

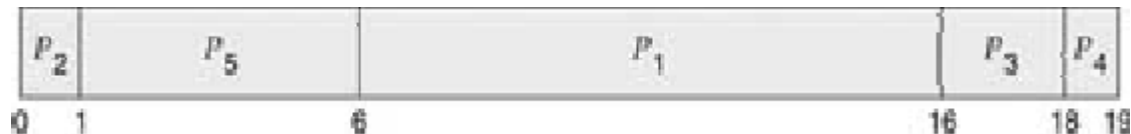
A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order.

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

	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 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 for 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

#### d) 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" <<bt[i]<<"\t\t" <<wt[i]<<"\t\t"<<tat[i];
}

```



```

avg_tat=total/n;
cout<<"\n\nAverage Waiting Time="<<avg_wt;
cout<<"\nAverage Turnaround Time="<<avg_tat;

    return 0;

}

```

### a) Output

```

input
P[1]
Burst Time:1 2
Priority:
P[2]
Burst Time:1 4
Priority:
P[3]
Burst Time:4 2
Priority:
P[4]
Burst Time:1 4
Priority:
P[5]
Burst Time:1 4
Priority:
Process    Burst Time    Waiting Time    Turnaround Time
P[1]        1              0                1
P[3]        4              1                5
P[2]        1              5                6
P[4]        1              6                7
P[5]        1              7                8
Average Waiting Time=3
Average Turnaround Time=5

```

### b) Result:

AverageWaitingTime...3.....

AverageTurnaroundTime...5.....

## Experiment-6

### BANKER ALGORITHM

Name of Student: Akshat Shrivastava		Class: CSIT-1	
Enrollment No: 0827CI201020		Batch: B1	
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 of time.

**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.

**Program:**

```
#include<iostream>
using namespace std;
class Bankers{
public:
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()
{
    Bankers b;
    int np=100;

    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>>b.alloc[i][j];

    cout<<"\nEnter the requirement data : \n";

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

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

    cin>>b.maxi[i][j];

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

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

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

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

```

```

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

cin>>b.avail[i];

int ex_it=nr;

    int flg;

    bool completed[np];

while(10)

    {

for(int i=0;i<np;i++)
{
            if(!completed[i] && b.check_safety(i,nr)
            {
for(int j=0;j<nr;j++)

b.avail[j]+=b.alloc[i][j];
            }
            completed[i]=true;
        }
flg=b.check(completed,np);

ex_it--;

if(flg==1 || ex_it==0)

        break;

    }

cout<<"\n\nThe final availability matrix \n";

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

cout<<b.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;  
  
}
```

**a) Output:**

```
Enter the no of processes : 5  
Enter the no of resources : 3  
Enter the allocation data :  
0 1 0  
2 0 0  
3 0 2  
2 1 1  
0 0 2  
  
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 5  
----- Result -----  
There is no deadlock
```

**a) Result:**

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

## Experiment-7

### FIFO PAGE REPLACEMENT

Name of Student: Akshat Shrivastava		Class: CSIT-1
Enrollment No: 0827CI201020		Batch: B1
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.

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

7	7	7	2	2	2	4	4	4	0	0	0	7	7	7
	0	0	0	3	3	3	2	2	2	1	1	1	0	0
		1	1	1	0	0	0	3	3	3	2	2	2	1

page frames

- 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

**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 queue

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

**d) Program:**

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

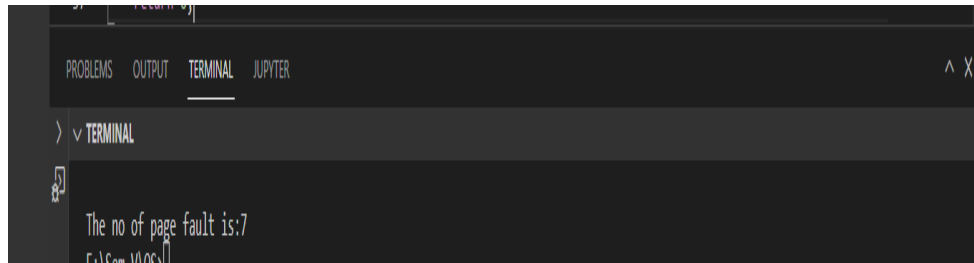
int pageFault(int page[],int n,intno_frame)
{
    int pagefault=0;

    vector<int> v1;
    int i;
    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());
            }
            v1.push_back(page[i]);
            pagefault++;
        }
    }
    return pagefault;
}

int main()
{
    int page[]={7,0,1,2,0,3,0,4,2,3,0,3,2};
    int n=13;
    int no_frame=4;
    cout<<pageFault(page,n,no_frame);
    return 0;
}
```



**a) Output:**

A screenshot of a terminal window from an IDE. The terminal has tabs for 'PROBLEMS', 'OUTPUT', 'TERMINAL', and 'JUPYTER'. The 'TERMINAL' tab is active. The output text in the terminal is 'The no of page fault is:7'.

```
PROBLEMS OUTPUT TERMINAL JUPYTER
> v TERMINAL
The no of page fault is:7
```

**b) Result:**

No. of page faults. 7

## Experiment-8

### LRU PAGE REPLACEMENT

Name of Student: Akshat Shrivastava		Class:CSIT-1	
Enrollment No: 0827CI201020		Batch: B1	
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 again in 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 longest time. 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 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

**d) Program:**

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

using namespace std;

int pageFault(int pages[],int n,intmem_capacity)
{
    int pagefault=0;
    vector<int> v1;

    for(int i=0;i<=n;i++)
    {
        auto it=find(v1.begin(),v1.end(),pages[i]);
        if(it==v1.end())
        {
            if(v1.size()==mem_capacity)
            {
                v1.erase(v1.begin());
            }
            v1.push_back(pages[i]);
            pagefault++;
        }
        else
        {
            v1.erase(it);
            v1.push_back(pages[i]);
        }
    }
}
```

```

    }
    return pagefault;
}

int main()
{

    int pages[]={7,0,1,2,0,3,0,4,2,3,0,3,2};
    int n=sizeof(pages)/sizeof(pages[0]); //no of pages

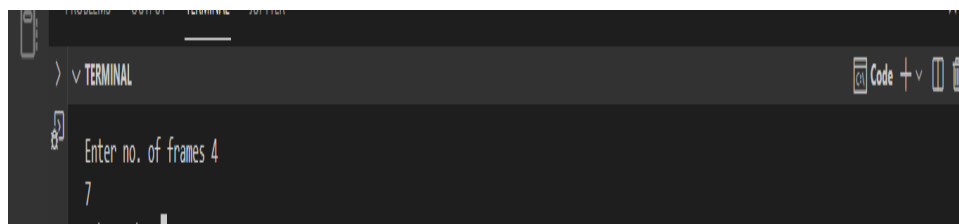

    int mem_capacity;
    cout<<"Enter no. of frames";
    cin>>mem_capacity;

    cout<<pageFault(pages,n,mem_capacity);

    return 0;
}

```

**d) OUTPUT:**



**e) Result:**

No. of pages faults 7.

## Experiment-9

### FCFS Disk Scheduling Algorithm

Name of Student: Akshat Shrivastava		Class: CSIT-1
Enrollment No: 0827CI201020		Batch: B1
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

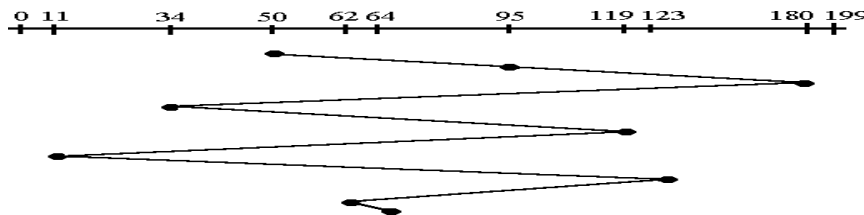
### FACILITIES REQUIRED

#### 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 in the 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 track 11 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;
int main()
{
    int a[100];
    int n;

    cout<<"Enter no. of service request:";
    cin>>n;

    for(int i=0;i<n;i++)
    {
        cin>>a[i];
    }

    int start;
    cout<<"Enter start position of arm:";
    cin>>start;

    int distance,current;
    int seek_count=0;

    for(int i=0;i<n;i++)
    {
        current=a[i];
        distance=abs(current-start);
        seek_count=seek_count+distance;
        start=current;
    }
    cout<<"Seek count is:"<<seek_count<<endl;
    cout<<"Average seek count is:"<<seek_count/n<<endl;

    cout<<"The sequence:";
    for(int i=0;i<n;i++)
    {
        cout<<a[i]<<" ";
    }
}
```

**d) Output:**

```
E:\sem-V\OS> E:\sem-V\OS> g++ disk_scheduling_fcfs.cpp -o disk_scheduling_fcfs.exe E:\sem-V\OS> disk_scheduling_fcfs
Enter no. of service request:8
176 79 34 60 92 11 41 114
Enter start position of arm: 50
Seek count is:510
Average seek count is:63
The sequence:176 79 34 60 92 11 41 114
```

**Result:**

Total Head Movement Required Serving All Requests ...7.....

## Experiment-10

### SSTF Disk Scheduling Algorithm

Name of Student: Akshat Shrivastava		Class: CSIT-1
Enrollment No: 0827CI201020		Batch: B1
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

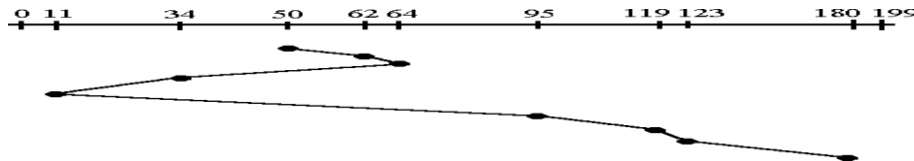
### FACILITIES REQUIRED

#### 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 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.





**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,intpos,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,intpos,int n)
{
    /* 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;

}

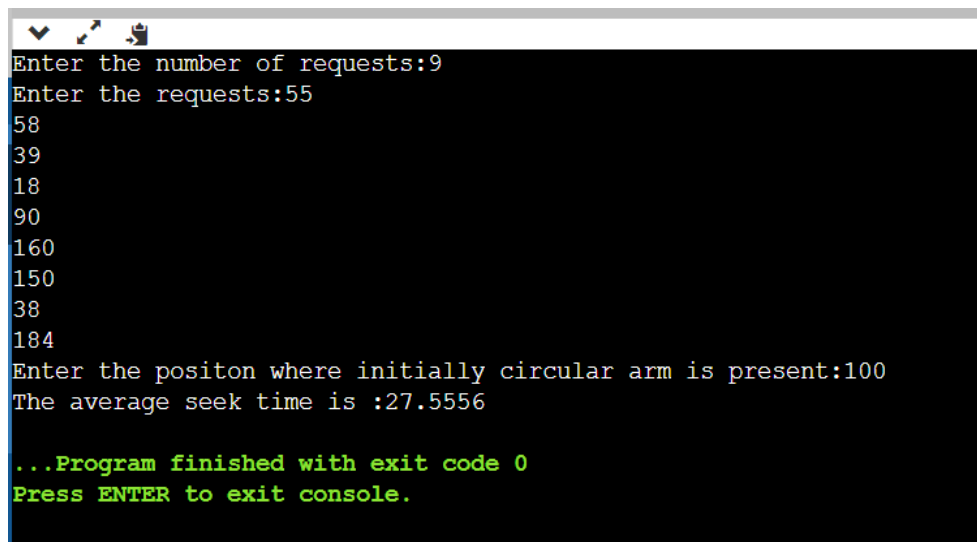
int main()
{
    int n,positom;
    cout<<"Enter the number of requests:";
    cin>>n;

    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;
}
```

**a) Output:**

A screenshot of a console window showing the execution of a C++ program. The program prompts the user to enter the number of requests, which is 9. It then prompts for the requests, which are 55, 58, 39, 18, 90, 160, 150, 38, and 184. The program then prompts for the initial position of the circular arm, which is 100. The output shows the average seek time as 27.5556. The program finishes with exit code 0 and prompts the user to press ENTER to exit the console.

```
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.
```

**b) Result:**

Total Head Movement Required Serving All Request: 27.5556

## 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 dynamiclinking 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?