ACROPOLIS INSTITUTE OF TECHNOLOGY & RESEARCH, INDORE

ACROPOLIS
Enlightening Wisdom

PRACTICAL FILE OF OPERATING SYSTEM

BTech: III Year

Department of Computer Science & Information Technology

Name of the Student : Abhishek Malviya

Branch & section: CSIT – 01

Roll No.: 0827CI201007

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 journal as per the BTech III year syllabus prescribed by the RGPV was done by Mr Abhishek Malviya in 5^{th} 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, 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/O subsystem, 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

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

CSIT-502

ACROPOLIS INSTITUTE OF TECHNOLOGY & RESEARCH, INDORE

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

<u>Index</u>

S.No.	Date of Exp.	Name of the Experiment	Page No.	Date of Submission	Grade & Sign of the Faculty
1.					
2.					
3.					
4.					
5.					
6.					
7.					
8.					
9.					
10.					
11.					

Experiment-1

FCFS SCHEDULING

Name of Student: Abhishek Malviya			Class: CI-1
Enrollment No:0827CI201007			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

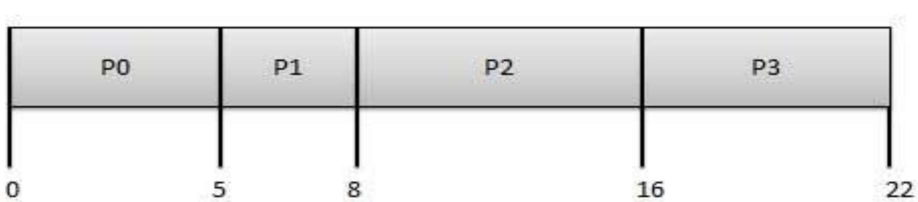
a) **Facilities Required Doing The Experiment:**

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) 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

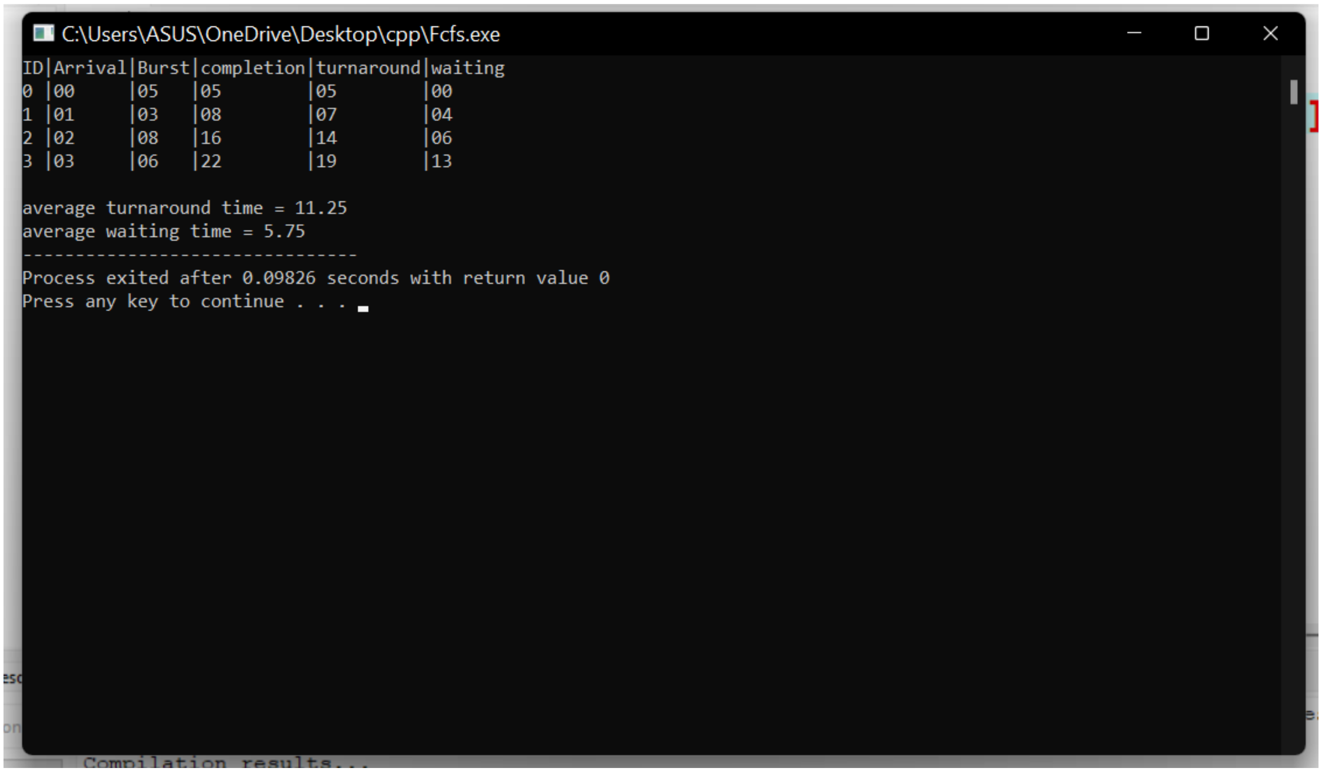
(a) Average waiting time = Total waiting Time / Number of process
(b) Average Turnaround time = Total Turnaround Time / Number of process

Step 7: Stop the process

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

class FCFS{
    public:

    void FunCompetionTime(int completiontime[],int bursttime[],int NoOfProcess){
    completiontime[0]=bursttime[0];
    for(int i=1; i<NoOfProcess; i++){
        completiontime[i]=completiontime[i-1]+bursttime[i];
    }

    }

    void FunTurnAroundTime(int completiontime[], int arrivaltime[], int turnaroundtime[],
    int NoOfProcess){
    for(int i=0; i<NoOfProcess; i++){
        turnaroundtime[i]=completiontime[i]-arrivaltime[i];
    }
    }

    void FunWaitingTime(int bursttime[], int turnaroundtime[], int waitingtime[], int
    NoOfProcess){

    for(int i=0; i<NoOfProcess; i++){
        waitingtime[i]=turnaroundtime[i] - bursttime[i];
    }

    }

};


int main(){
int NoOfProcess=4;
float avgwaiting=0.0,avgturnaroundtime=0.0;
int bursttime[]={5,3,8,6}, completiontime[NoOfProcess], turnaroundtime[NoOfProcess],
    waitingtime[NoOfProcess], arrivaltime[NoOfProcess]={0,1,2,3};

    FCFS table1;
```

```
table1.FunCompetionTime(completiontime, bursttime, NoOfProcess);
table1.FunTurnAroundTime(completiontime, arrivaltime, turnaroundtime,
NoOfProcess);
table1.FunWaitingTime(bursttime, turnaroundtime, waitingtime, NoOfProcess);

cout<<"ID"<<"|"<<"Arrival"<<"|"<<"Burst"<<"|"<<"completion"<<"|"<<"turnaround"<
<"|"<<"waiting"<<endl;

for(int i=0; i<NoOfProcess; i++){
cout<<i<<" |";
cout<<setw(2)<<setfill('0')<<arrivaltime[i]<<"     |";
cout<<setw(2)<<setfill('0')<<bursttime[i]<<"   |";
cout<<setw(2)<<setfill('0')<<completiontime[i]<<"        |";
cout<<setw(2)<<setfill('0')<<turnaroundtime[i]<<"        |";
cout<<setw(2)<<setfill('0')<<waitingtime[i]<<endl;
}


for(int i=0; i<NoOfProcess; i++){
avgwaiting = avgwaiting + waitingtime[i];
avgturnaroundtime = avgturnaroundtime + turnaroundtime[i];
}
cout<<endl<<"average turnaround time = "<< avgturnaroundtime/NoOfProcess;
cout<<endl<<"average waiting time = "<<avgwaiting/NoOfProcess;



return 0;
}
```

e)Output



d) Result:

Average Waiting Time 11.25 .

Average Turnaround Time 5.75 .

Experiment-2

SJF Scheduling

Name of Student: Abhishek Malviya		Class: CI-1
Enrollment No: 0827CI201007		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 SJF CPU Scheduling Algorithm.

FACILITIES REQUIRED

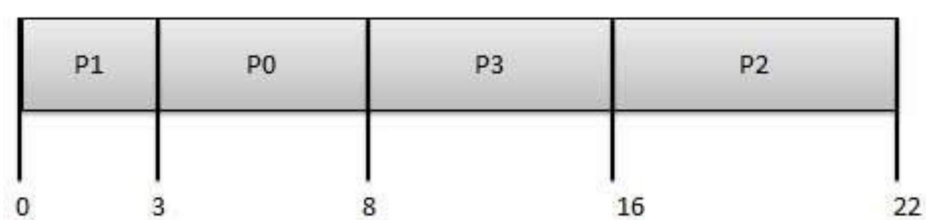
a) Facilities Required Doing The Experiment:

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

(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 process

Step 8: Stop the process

d) Program:

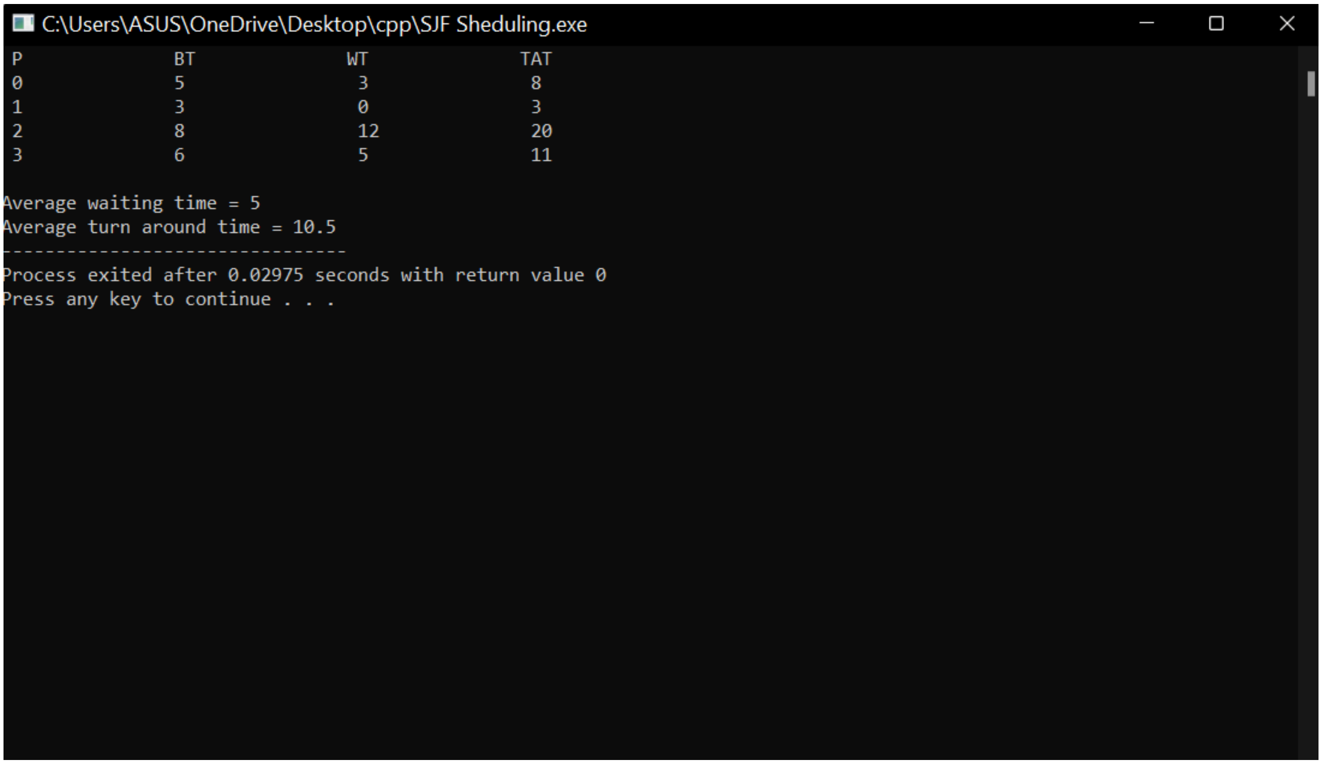
```
// C++ program to implement Shortest Remaining Time First
// Shortest Remaining Time First (SRTF)

#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 (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[] = { { 0, 5, 0 }, { 1 ,3, 1 },
                       { 2, 8, 2 }, { 3, 6, 3}};
    int n = sizeof(proc) / sizeof(proc[0]);

    findavgTime(proc, n);
    return 0;
}
```

e) Output:



```
C:\Users\ASUS\OneDrive\Desktop\cpp\SJF Sheduling.exe
P              BT              WT              TAT
0              5               3               8
1              3               0               3
2              8               12              20
3              6               5               11

Average waiting time = 5
Average turn around time = 10.5
--------------------------------
Process exited after 0.02975 seconds with return value 0
Press any key to continue . . .
```

f) Result:

Average Waiting Time = 5

Average Turnaround Time = 10.5

Experiment-3

SRTF Scheduling

Name of Student: Abhishek Malviya		Class: CI-1
Enrolment No: 0827CI201007		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 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 (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

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

(f) Average Turnaround time = Total Turnaround Time / Number of process

Step 8: Stop the process

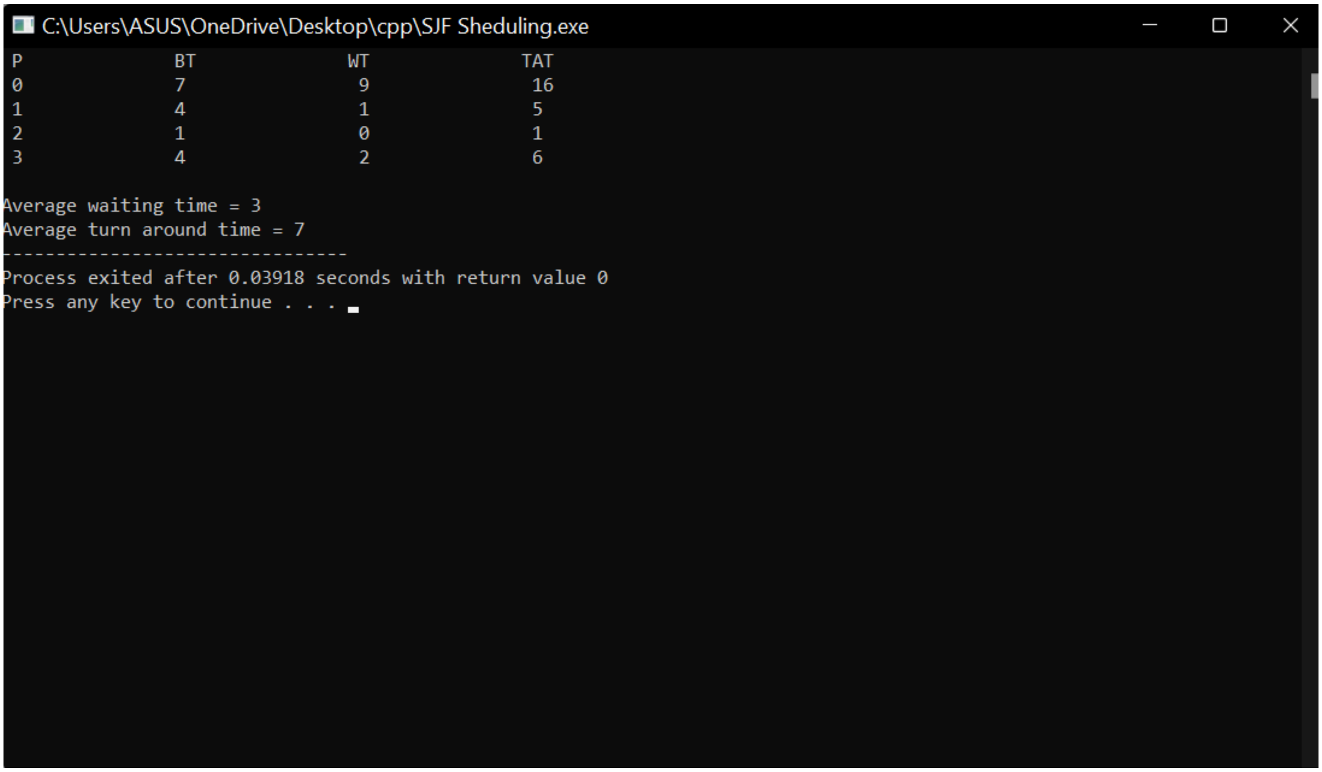
d) Program:

```
// 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 (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[] = { { 0, 7, 0 }, { 1, 4, 2 },
                                        { 2, 1, 4 }, { 3, 4, 5} };
    int n = sizeof(proc) / sizeof(proc[0]);

    findavgTime(proc, n);
    return 0;
}
```

e) Output:



f) Result:

Average Waiting Time =3

Average Turnaround Time = 7

Experiment-4

ROUND ROBIN Scheduling

Name of Student: Abhishek Malviya		Class: CI-1
Enrolment No: 0827CI201007		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 Round Robin scheduling.

FACILITIES REQUIRED

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

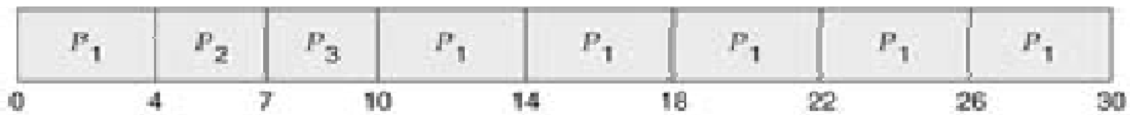
S.NO	FACILITIES REQUIRED	QUANTITY
1	**System**	**1**
2	**Windows XP/7**	

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.

Process	Burst 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 slice

Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time

Step 4: Calculate the no. of time slices for each process where

No. of time slice for process(n) = burst time process(n)/time slice

Step 5: If the burst time is less than the time slice then the no. of time slices =1.

Step 6: Consider the ready queue is a circular Q, calculate

(a) Waiting time for process(n) = waiting time of process(n-1)+ burst time of process(n-1) + the time difference in getting the CPU from process(n-1)
(b) 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

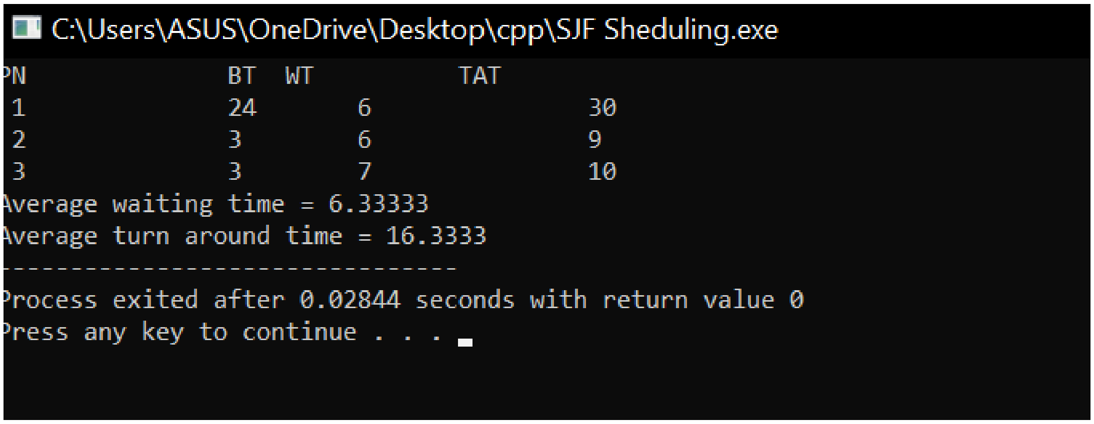
Program:

```
// C++ program for implementation of RR scheduling
#include<iostream>
using namespace std;

// Function to find the waiting time for all
// processes
void findWaitingTime(int processes[], int n,
                        int bt[], int wt[], int quantum)
{
    // Make a copy of burst times bt[] to store remaining
    // burst times.
    int rem_bt[n];
    for (int i = 0 ; i < n ; i++)
        rem_bt[i] = bt[i];

    int t = 0; // Current time

    // Keep traversing processes in round robin manner
    // until all of them are not done.
    while (1)
    {
        bool done = true;

        // Traverse all processes one by one repeatedly
        for (int i = 0 ; i < n; i++)
        {
            // If burst time of a process is greater than 0
            // then only need to process further
            if (rem_bt[i] > 0)
            {
                done = false; // There is a pending process

                if (rem_bt[i] > quantum)
                {
                    // Increase the value of t i.e. shows
                    // how much time a process has been processed
                    t += quantum;

                    // Decrease the burst_time of current process
                    // by quantum
                    rem_bt[i] -= quantum;
                }

                // If burst time is smaller than or equal to
                // quantum. Last cycle for this process
                else
                {
                    // Increase the value of t i.e. shows
                    // how much time a process has been processed
                    t = t + rem_bt[i];
```

```
                    // Waiting time is current time minus time
                    // used by this process
                    wt[i] = t - bt[i];

                    // As the process gets fully executed
                    // make its remaining burst time = 0
                    rem_bt[i] = 0;
                }
            }
        }

        // If all processes are done
        if (done == true)
        break;
    }
}

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

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

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

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

    // Display processes along with all details
    cout << "PN\t "<< " \tBT "
        << " WT " << " \tTAT\n";

    // Calculate total waiting time and total turn
    // around time
    for (int i=0; i<n; i++)
    {
        total_wt = total_wt + wt[i];
        total_tat = total_tat + tat[i];
        cout << " " << i+1 << "\t\t" << bt[i] <<"\t "
            << wt[i] <<"\t\t " << tat[i] <<endl;
    }

    cout << "Average waiting time = "
        << (float)total_wt / (float)n;
    cout << "\nAverage turn around time = "
        << (float)total_tat / (float)n;
```

```
}

// Driver code
int main()
{
	// process id's
	int processes[] = { 1, 2, 3};
	int n = sizeof processes / sizeof processes[0];

	// Burst time of all processes
	int burst_time[] = {24, 3, 3};

	// Time quantum
	int quantum = 2;
	findavgTime(processes, n, burst_time, quantum);
	return 0;
}
```

d) Output:



```
C:\Users\ASUS\OneDrive\Desktop\cpp\SJF Sheduling.exe
PN              BT   WT          TAT
 1              24       6                30
 2              3        6                9
 3              3        7                10
Average waiting time = 6.33333
Average turn around time = 16.3333
--------------------------------
Process exited after 0.02844 seconds with return value 0
Press any key to continue . . .
```

e) Result:

Average Waiting Time = 6.33333

Average Turnaround Time 16.3333

Experiment-5

PRIORITY SCHEDULING

Name of Student:		Class: CI-1
Enrollment No: 0827CI201007		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 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

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

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

(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 process

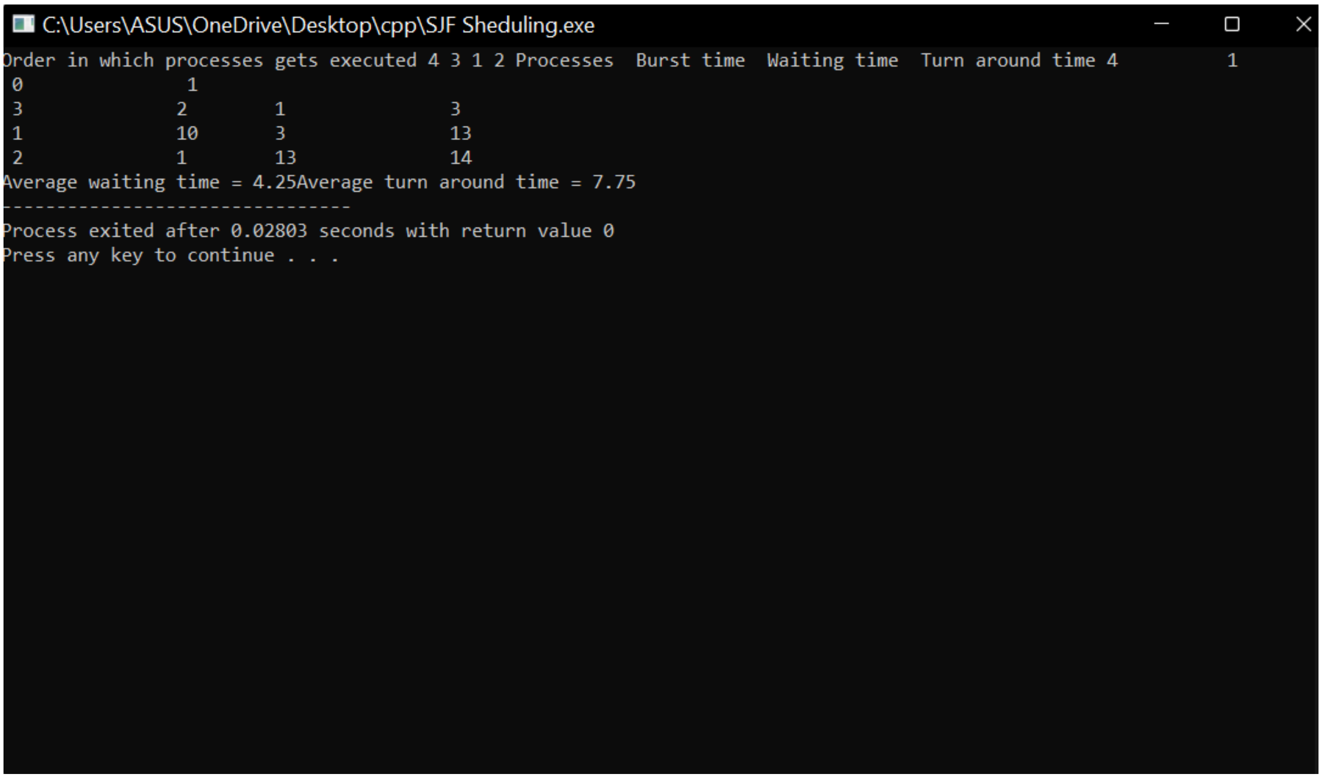
Step 8: Stop the process

d) Program:

```
#include<bits/stdc++.h>
using namespace std;
struct Process {
  int pid; // Process ID
  int bt; // CPU Burst time required
  int priority; // Priority of this process
};
// sorting the Process acc. to the priority
bool compare(Process a, Process b) {
  return (a.priority > b.priority);
}
void waitingtime(Process pro[], int n, int wt[]) {
  // Initial waiting time for a process is 0
  wt[0] = 0;
  // calculating waiting time
  for (int i = 1; i < n ; i++ )
    wt[i] = pro[i-1].bt + wt[i-1] ;
}
 // Function to calculate turn around time
void turnarround( Process pro[], int n, int wt[], int tat[]) {
  // calculating turnaround time by adding
  // bt[i] + wt[i]
  for (int i = 0; i < n ; i++)
    tat[i] = pro[i].bt + wt[i];
}
//Function to calculate average time
void avgtime(Process pro[], int n) {
  int wt[n], tat[n], total_wt = 0, total_tat = 0;
  //Function to find waiting time of all processes
  waitingtime(pro, n, wt);
```

```
  //Function to find turn around time for all processes
  turnarround(pro, n, wt, tat);
  //Display processes along with all details
  cout << "
Processes "<< " Burst time " << " Waiting time " << " Turn around time
";
  // Calculate total waiting time and total turn
  // around time
  for (int i=0; i<n; i++) {
    total_wt = total_wt + wt[i];
    total_tat = total_tat + tat[i];
    cout << " " << pro[i].pid << "\t\t" << pro[i].bt << "\t " << wt[i] << "\t\t " << tat[i] <<endl;
  }
  cout << "
Average waiting time = " << (float)total_wt / (float)n;
  cout << "
Average turn around time = " << (float)total_tat / (float)n;
}
void scheduling(Process pro[], int n) {
  // Sort processes by priority
  sort(pro, pro + n, compare);
  cout<< "Order in which processes gets executed
";
  for (int i = 0 ; i < n; i++)
    cout << pro[i].pid <<" " ;
  avgtime(pro, n);
}
// main function
int main() {
  Process pro[] = {{1, 10, 2}, {2, 5, 0}, {3, 8, 1}};
  int n = sizeof pro / sizeof pro[0];
  scheduling(pro, n);
  return 0;
}
```

e) Output:



f) Result:

Average Waiting Time = 4.25

Average Turnaround Time 7.75

Experiment-6

BANKER ALGORITHM

<table>
<tr><td colspan="3">Name of Student: Abhishek Malviya</td><td>Class: CI-1</td></tr>
<tr><td colspan="3">Enrollment No: 0827CI201007</td><td>Batch:2020-24</td></tr>
<tr><td>Date of Experiment</td><td colspan="2">Date of Submission</td><td>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 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.

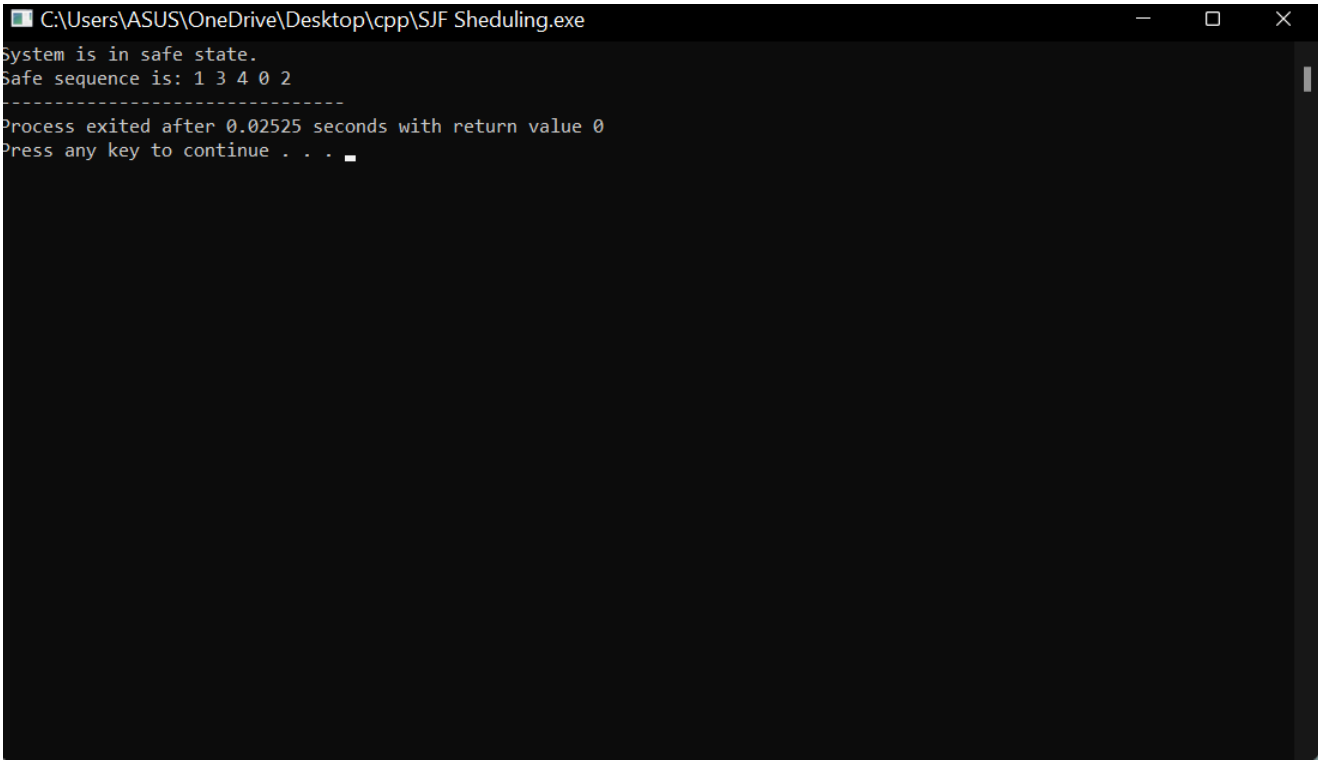
d) Program:

```
// C++ program to illustrate Banker's Algorithm
#include<iostream>
using namespace std;

// Number of processes
const int P = 5;

// Number of resources
const int R = 3;

// Function to find the need of each process
void calculateNeed(int need[P][R], int maxm[P][R],
                   int allot[P][R])
{
    // Calculating Need of each P
    for (int i = 0 ; i < P ; i++)
        for (int j = 0 ; j < R ; j++)

            // Need of instance = maxm instance -
            //                    allocated instance
            need[i][j] = maxm[i][j] - allot[i][j];
}

// Function to find the system is in safe state or not
bool isSafe(int processes[], int avail[], int maxm[][R],
            int allot[][R])
{
    int need[P][R];

    // Function to calculate need matrix
    calculateNeed(need, maxm, allot);

    // Mark all processes as infinish
    bool finish[P] = {0};

    // To store safe sequence
    int safeSeq[P];

    // Make a copy of available resources
    int work[R];
    for (int i = 0; i < R ; i++)
        work[i] = avail[i];

    // While all processes are not finished
    // or system is not in safe state.
    int count = 0;
    while (count < P)
    {
        // Find a process which is not finish and
        // whose needs can be satisfied with current
        // work[] resources.
        bool found = false;
```

```
		for (int p = 0; p < P; p++)
		{
			// First check if a process is finished,
			// if no, go for next condition
			if (finish[p] == 0)
			{
				// Check if for all resources of
				// current P need is less
				// than work
				int j;
				for (j = 0; j < R; j++)
					if (need[p][j] > work[j])
						break;

				// If all needs of p were satisfied.
				if (j == R)
				{
					// Add the allocated resources of
					// current P to the available/work
					// resources i.e.free the resources
					for (int k = 0 ; k < R ; k++)
						work[k] += allot[p][k];

					// Add this process to safe sequence.
					safeSeq[count++] = p;

					// Mark this p as finished
					finish[p] = 1;

					found = true;
				}
			}
		}

		// If we could not find a next process in safe
		// sequence.
		if (found == false)
		{
			cout << "System is not in safe state";
			return false;
		}
	}

	// If system is in safe state then
	// safe sequence will be as below
	cout << "System is in safe state.\nSafe"
		" sequence is: ";
	for (int i = 0; i < P ; i++)
		cout << safeSeq[i] << " ";

	return true;
}

// Driver code
int main()
{
```

```
    int processes[] = {0, 1, 2, 3, 4};

    // Available instances of resources
    int avail[] = {3, 3, 2};

    // Maximum R that can be allocated
    // to processes
    int maxm[][R] = {{7, 5, 3},
                     {3, 2, 2},
                     {9, 0, 2},
                     {2, 2, 2},
                     {4, 3, 3}};

    // Resources allocated to processes
    int allot[][R] = {{0, 1, 0},
                      {2, 0, 0},
                      {3, 0, 2},
                      {2, 1, 1},
                      {0, 0, 2}};

    // Check system is in safe state or not
    isSafe(processes, avail, maxm, allot);

    return 0;
}
```

e) Output:



f) Result:

The Sequence Is: 1 3 4 0 2

Experiment-7

FIFO PAGE REPLACEMENT

<table>
<tr><td colspan="2">Name of Student: Abhishek Malviya</td><td colspan="2">Class: CI-1</td></tr>
<tr><td colspan="2">Enrollment No: 0827CI201007</td><td colspan="2">Batch:2020-24</td></tr>
<tr><td>Date of Experiment</td><td colspan="2">Date of Submission</td><td>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 implement page replacement algorithm FIFO.

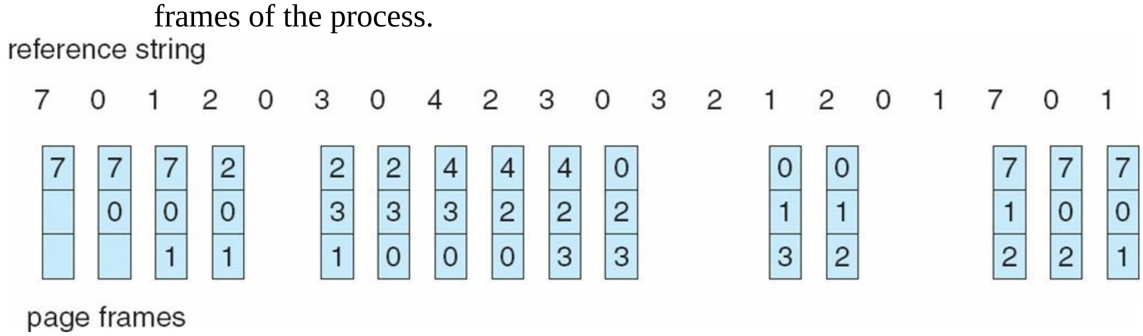
FACILITIES REQUIRED

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 $\Rightarrow$ 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:

```
// C++ implementation of FIFO page replacement
// in Operating Systems.
#include<bits/stdc++.h>
using namespace std;

// Function to find page faults using FIFO
int pageFaults(int pages[], int n, int capacity)
{
	// To represent set of current pages. We use
	// an unordered_set so that we quickly check
	// if a page is present in set or not
	unordered_set<int> s;

	// To store the pages in FIFO manner
	queue<int> indexes;

	// Start from initial page
	int page_faults = 0;
	for (int i=0; i<n; i++)
	{
		// Check if the set can hold more pages
		if (s.size() < capacity)
		{
			// Insert it into set if not present
			// already which represents page fault
			if (s.find(pages[i])==s.end())
			{
				// Insert the current page into the set
				s.insert(pages[i]);

				// increment page fault
				page_faults++;

				// Push the current page into the queue
				indexes.push(pages[i]);
			}
		}

		// If the set is full then need to perform FIFO
		// i.e. remove the first page of the queue from
		// set and queue both and insert the current page
		else
		{
			// Check if current page is not already
			// present in the set
			if (s.find(pages[i]) == s.end())
			{
				// Store the first page in the
				// queue to be used to find and
				// erase the page from the set
				int val = indexes.front();

				// Pop the first page from the queue
				indexes.pop();
```

```
                    // Remove the indexes page from the set
                    s.erase(val);

                    // insert the current page in the set
                    s.insert(pages[i]);

                    // push the current page into
                    // the queue
                    indexes.push(pages[i]);

                    // Increment page faults
                    page_faults++;
                }
            }
        }

        return page_faults;
}

// Driver code
int main()
{
        int pages[] = {7, 0, 1, 2, 0, 3, 0, 4,
                    2, 3, 0, 3, 2};
        int n = sizeof(pages)/sizeof(pages[0]);
        int capacity = 4;
        cout << " number of pageFaults arev = "<<pageFaults(pages, n, capacity);
        return 0;
}
```

e) Output:

Output

```
no. of page faults =7
```

f) Result:

No. of page faults 7.

Experiment-8

LRU PAGE REPLACEMENT

Name of Student: Abhishek Malviya		Class: CI-1
Enrollment No: 0827CI201007		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 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

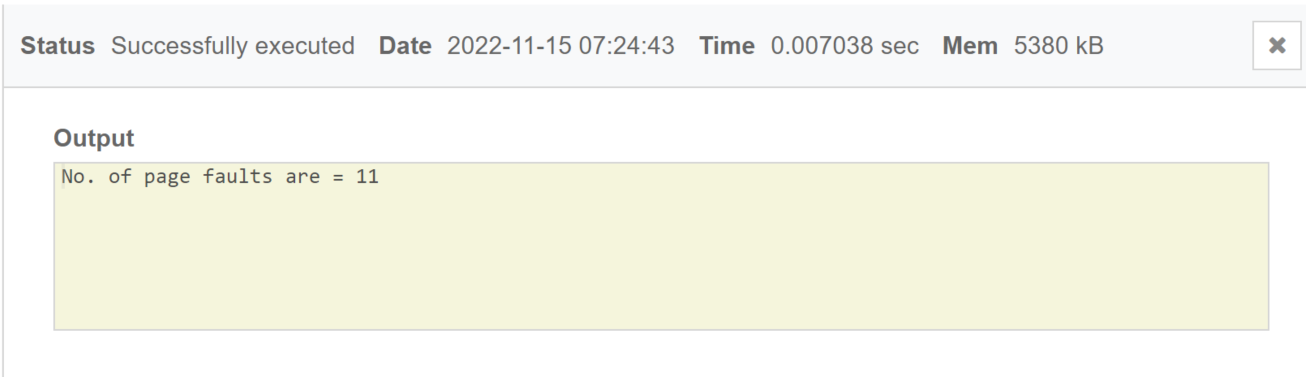
Step 5: When the page fault occurs replace page present at the bottom of the stack

d) Program:

```
//C++ implementation of above algorithm
#include<bits/stdc++.h>
using namespace std;

// Function to find page faults using indexes
int pageFaults(int pages[], int n, int capacity)
{
	// To represent set of current pages. We use
	// an unordered_set so that we quickly check
	// if a page is present in set or not
	unordered_set<int> s;

	// To store least recently used indexes
	// of pages.
	unordered_map<int, int> indexes;

	// Start from initial page
	int page_faults = 0;
	for (int i=0; i<n; i++)
	{
		// Check if the set can hold more pages
		if (s.size() < capacity)
		{
			// Insert it into set if not present
			// already which represents page fault
			if (s.find(pages[i])==s.end())
			{
				s.insert(pages[i]);

				// increment page fault
				page_faults++;
			}

			// Store the recently used index of
			// each page
			indexes[pages[i]] = i;
		}
```

```
			// If the set is full then need to perform lru
			// i.e. remove the least recently used page
			// and insert the current page
			else
			{
				// Check if current page is not already
				// present in the set
				if (s.find(pages[i]) == s.end())
				{
					// Find the least recently used pages
					// that is present in the set
					int lru = INT_MAX, val;
					for (auto it=s.begin(); it!=s.end(); it++)
					{
						if (indexes[*it] < lru)
						{
							lru = indexes[*it];
							val = *it;
						}
					}

					// Remove the indexes page
					s.erase(val);

					// insert the current page
					s.insert(pages[i]);

					// Increment page faults
					page_faults++;
				}

				// Update the current page index
				indexes[pages[i]] = i;
			}
		}

		return page_faults;
}

// Driver code
int main()
{
		int pages[] = {1,2,3,2,1,5,2,1,6,2,5,6,3,1,3,6,1,2,4,3};

		int n = sizeof(pages)/sizeof(pages[0]);
		int capacity = 3;
		cout << pageFaults(pages, n, capacity);
		return 0;
}
```

OUTPUT:



d) Result:

No. of pages faults 11.

Experiment-9

FCFS Disk Scheduling Algorithm

<table>
<tr><td colspan="4">Name of Student: Abhishek Malviya</td><td colspan="2">Class: CI-1</td></tr>
<tr><td colspan="4">Enrollment No: 0827CI201007</td><td colspan="2">Batch:2020-24</td></tr>
<tr><td>Date of Experiment</td><td colspan="3">Date of Submission</td><td colspan="2">Submitted on:</td></tr>
<tr><td colspan="3">Remarks by faculty:</td><td colspan="3">Grade:</td></tr>
<tr><td colspan="2">Signature of student:</td><td colspan="4">Signature of Faculty:</td></tr>
</table>

OBJECTIVE OF THE EXPERIMENT

To implement FCFS Disk Scheduling Algorithm

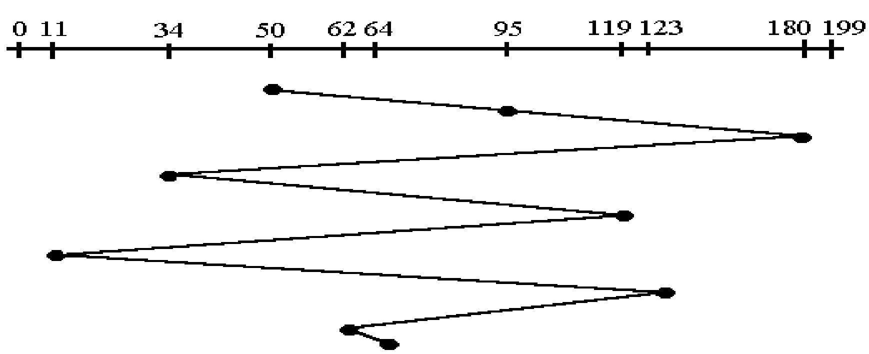
FACILITIES REQUIRED

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)

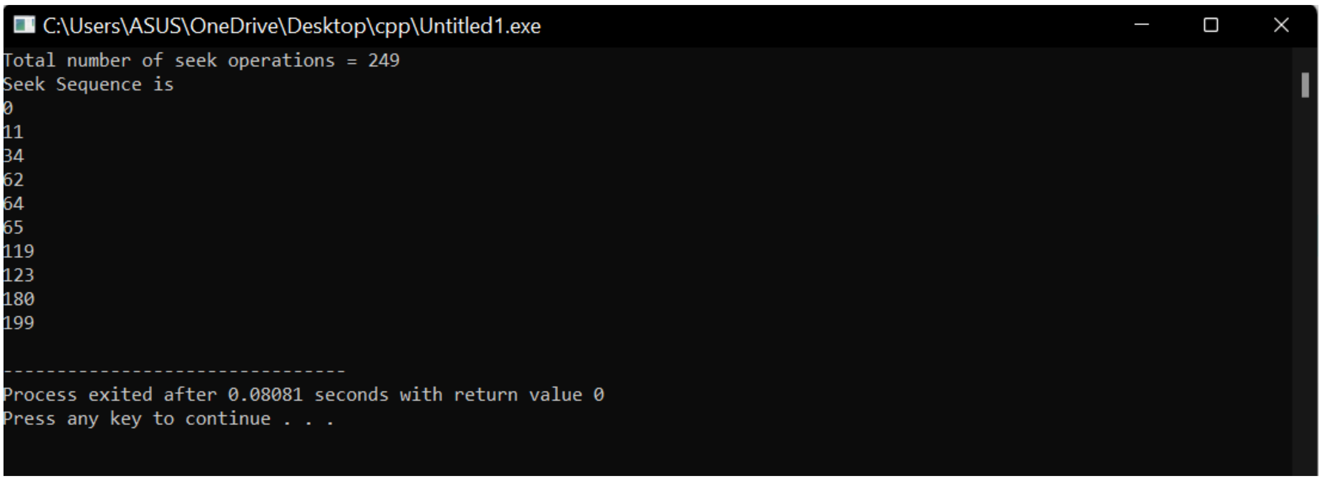
Step 3: Calculate the total head movement required to serve all request.

Program:

```
// C++ program to demonstrate
// FCFS Disk Scheduling algorithm

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

int size = 8;

void FCFS(int arr[], int head)
{
	int seek_count = 0;
	int distance, cur_track;

	for (int i = 0; i < size; i++) {
		cur_track = arr[i];

		// calculate absolute distance
		distance = abs(cur_track - head);

		// increase the total count
		seek_count += distance;

		// accessed track is now new head
		head = cur_track;
	}

	cout << "Total number of seek operations = "
		<< seek_count << endl;

	// Seek sequence would be the same
	// as request array sequence
	cout << "Seek Sequence is" << endl;

	for (int i = 0; i < size; i++) {
		cout << arr[i] << endl;
	}
}

// Driver code
int main()
{

	// request array
	int arr[size] = { 0,11,34,50,62,64,65,119,123,180,199 };
	int head = 50;
```

```
        FCFS(arr, head);

        return 0;
}
```

d) Output:



```
C:\Users\ASUS\OneDrive\Desktop\cpp\Untitled1.exe
Total number of seek operations = 249
Seek Sequence is
0
11
34
62
64
65
119
123
180
199

--------------------------------
Process exited after 0.08081 seconds with return value 0
Press any key to continue . . .
```

e) Result:

Total Head Movement Required Serving All Requests 249.

Experiment-10

SSTF Disk Scheduling Algorithm

<table>
<tr><td colspan="3">Name of Student: Abhishek Malviya</td><td>Class: CI-1</td></tr>
<tr><td colspan="3">Enrollment No: 0827CI201007</td><td>Batch:2020-24</td></tr>
<tr><td>Date of Experiment</td><td colspan="2">Date of Submission</td><td>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 implement SSTF Disk Scheduling Algorithm

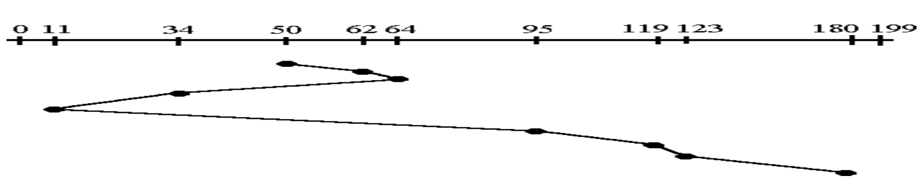
FACILITIES REQUIRED

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 head position

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

d) Program:

```
// C++ program for implementation of
// SSTF disk scheduling
#include <bits/stdc++.h>
using namespace std;

// Calculates difference of each
// track number with the head position
void calculatedifference(int request[], int head,
                                                    int diff[][2], int n)
{
        for(int i = 0; i < n; i++)
        {
                diff[i][0] = abs(head - request[i]);
        }
}

// Find unaccessed track which is
// at minimum distance from head
int findMIN(int diff[][2], int n)
{
        int index = -1;
        int minimum = 1e9;

        for(int i = 0; i < n; i++)
        {
                if (!diff[i][1] && minimum > diff[i][0])
                {
                        minimum = diff[i][0];
                        index = i;
                }
        }
        return index;
}

void shortestSeekTimeFirst(int request[],
```

```
                                        int head, int n)
{
	if (n == 0)
	{
		return;
	}

	// Create array of objects of class node
	int diff[n][2] = { { 0, 0 } };

	// Count total number of seek operation
	int seekcount = 0;

	// Stores sequence in which disk access is done
	int seeksequence[n + 1] = {0};

	for(int i = 0; i < n; i++)
	{
		seeksequence[i] = head;
		calculatedifference(request, head, diff, n);
		int index = findMIN(diff, n);
		diff[index][1] = 1;

		// Increase the total count
		seekcount += diff[index][0];

		// Accessed track is now new head
		head = request[index];
	}
	seeksequence[n] = head;

	cout << "Total number of seek operations = "
		<< seekcount << endl;
	cout << "Seek sequence is : " << "\n";

	// Print the sequence
	for(int i = 0; i <= n; i++)
	{
		cout << seeksequence[i] << "\n";
	}
}

// Driver code
int main()
{
	int n = 10;
	int proc[n] = { 0,11,34,62,64,65,119,123,180,199 };

	shortestSeekTimeFirst(proc, 50, n);

	return 0;
}
```

e) Output:

```
C:\Users\ASUS\OneDrive\Desktop\cpp\Untitled1.exe
Total number of seek operations = 279
Seek sequence is :
50
62
64
65
34
11
0
119
123
180
199

--------------------------------
Process exited after 0.07261 seconds with return value 0
```

f) Result:

Total Head Movement Required Serving All Requests = 279.

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?