

Experiment-1

FCFS SCHEDULING

Name of Student:AKASH JAIN		Class: CSIT-1
Enrollment No: 0827CI201017		Batch:2020-2024
Date of Experiment	Date of Submission	Submitted on:
Remarks by faculty:		Grade:
Signature of student:		Signature of Faculty:

OBJECTIVE OF THE EXPERIMENT

To write c++ program to implement the FCFS SCHEDULING.

FACILITIES REQUIRED

a) Facilities Required Doing The Experiment:

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

b) Concept of FCFS:

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

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



c) Algorithm:

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue

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

d) Program:

```
//First Come First Serve with zero arrival time .....

#include <iostream>
using namespace std;
class FCFS
{
public:
void completion(int procesid[],int n,int bursttime[],int comp[])
{
    comp[0]=bursttime[0];
    for(int i=1;i<n;i++)
    {
        comp[i]=comp[i-1]+bursttime[i];
    }
}
void turnaround_time(int procesid[],int n,int bursttime[],int tat[],int comp[])
{
    for(int i=0;i<n;i++)
    {
        tat[i]=comp[i];
    }
}
void waiting_time(int procesid[],int n,int bursttime[],int tat[],int wait[])
{
    for(int i=0;i<n;i++)
    {
        wait[i]=tat[i]-bursttime[i];
    }
}
void display(int procesid[],int comp[],int tat[],int wait[],int n)
{
    cout<<"p_ID completion TAT Waiting"<<endl;
    for(int i=0;i<n;i++)
    {
        cout<<procesid[i]<<"    " <<comp[i]<<"    " <<tat[i]<<"    " <<wait[i]<<endl;
    }
}
};
int main()
{
    FCFS f;
    int n;
    cout<<"Enter the number of process:";
```

```

cin>>n;
int procesid[n];
int bursttime[n];
// 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;
}

```

//First Come First Serve with vrying arrival time

```

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

void turnaround_time(int procesid[],int n,int bursttime[],int tat[],int comp[],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;  
}
```

e) Output:

```
//First Come First Serve with zero arrival time .....
```

```
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
8
Enter the process ID and burst time:4
7
p_ID completion TAT Waiting
1      5         5      0
2      9         9      5
3     17        17      9
4     24        24     17
```

```
//First Come First Serve with vrying arrival time .....
```

```
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
0
5
Enter the arrival time and burst time of Process :2
1
6
Enter the arrival time and burst time of Process :3
2
8
p_ID    arrival    burst completion    TAT    Waiting
1        0         5         5         5         0
2        1         6        11        10         4
3        2         8        19        17         9
```

f) Result:

Average Waiting Time : 7.75 , 4.33

Average Turnaround Time : 13.75 , 10.66

Experiment-2

SJF Scheduling

Name of Student: AKASH JAIN		Class: CSIT-1
Enrollment No: 0827CI201017		Batch: 2020-2024
Date of Experiment	Date of Submission	Submitted on:
Remarks by faculty:		Grade:
Signature of student:		Signature of Faculty:

OBJECTIVE OF THE EXPERIMENT

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

FACILITIES REQUIRED

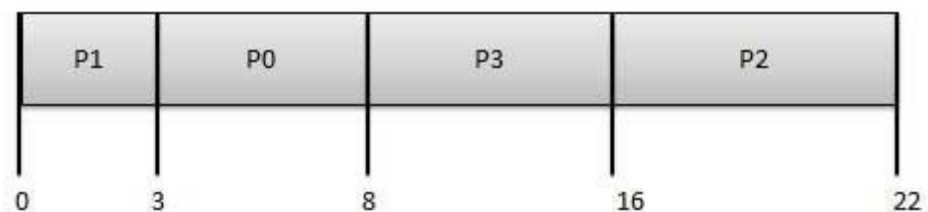
a) Facilities Required Doing The Experiment:

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

b) Concept of SJF:

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

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



c) Algorithm:

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue

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

```
#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;
}

```

e) 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
```

f)

f) Result:

Average Waiting Time : 15

Average Turnaround Time: 41

Experiment-3

SRTF Scheduling

Name of Student:AKASH JAIN		Class: CSIT-1
Enrollment No:0827CI201017		Batch: 2020-2024
Date of Experiment	Date of Submission	Submitted on:
Remarks by faculty:		Grade:
Signature of student:		Signature of Faculty:

OBJECTIVE OF THE EXPERIMENT

To write c program to implement SRTF scheduling.

FACILITIES REQUIRED

a) Facilities Required To Do The Experiment:

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

b) Concept Of SRTF Scheduling:

- Associate with each process the length of its next CPU burst. Use these lengths 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_n).

Step 7: If the burst time of newly arrived process (BT_n) 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:

```
// 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" << "BT\t" << "WT\t" << "TAT\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" << proc[i].bt << "\t " << wt[i]
    << "\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;
}

```

e) 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

f) Result:

Average Waiting Time: 4.6

Average Turnaround Time: 9.2

Experiment-4

ROUND ROBIN Scheduling

Name of Student:AKASH JAIN		Class: CSIT-1
Enrollment No:0827CI201017		Batch:2020-2024
Date of Experiment	Date of Submission	Submitted on:
Remarks by faculty:		Grade:
Signature of student:		Signature of Faculty:

OBJECTIVE OF THE EXPERIMENT

To write c program to implement Round Robin scheduling.

FACILITIES REQUIRED

a) Facilities Required To Do The Experiment:

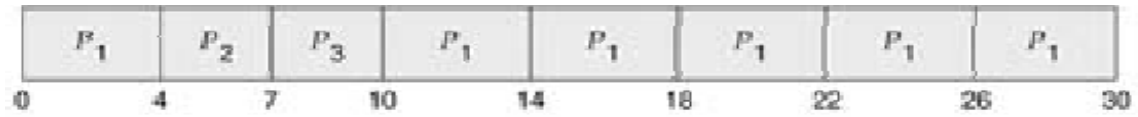
S.NO	FACILITIES REQUIRED	QUANTITY
1	System	1
2	Windows XP/7	

b) Concept Of Round Robin Scheduling:

This Algorithm is designed especially for time-sharing systems. A small unit of time, called time slices or **quantum** is defined. All runnable processes are kept in a circular queue. The CPU scheduler goes around this queue, allocating the CPU to each process for a time interval of one quantum. New processes are added to the tail of the queue. The CPU scheduler picks the first process from the queue, sets a timer to interrupt after one quantum, and dispatches the process. 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

d) Program:

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

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

return 0;
}
```

e) Output:

```
Enter the total number of process=4
Enter the Arrival time, Burst time for All the processes
Arrival time for process 1:0
Burst time for process 1:5
Arrival time for process 2:2
Burst time for process 2:8
Arrival time for process 3:4
Burst time for process 3:9
Arrival time for process 4:5
Burst time for process 4:3
Enter the value of time QUANTUM:
2

Process :Turnaround Time:Waiting Time

Process{4}      :      10      :      7
Process{1}      :      16      :     11
Process{2}      :      20      :     12
Process{3}      :      21      :     12

Average waiting time 10.5
Average turn around time 16.75
```

f) Result:

Average Waiting Time:16.75

Average Turnaround Time:10.5

Experiment-5

PRIORITY SCHEDULING

Name of Student:AKASH JAIN		Class: CSIT-1
Enrplment No:0827CI201017		Batch:2020-2024
Date of Experiment	Date of Submission	Submitted on:
Remarks by faculty:		Grade:
Signature of student:		Signature of Faculty:

OBJECTIVE OF THE EXPERIMENT

To write c program to implement Priority scheduling.

FACILITIES REQUIRED

a) Facilities Required To Do The Experiment:

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

b) Concept Of Priority Scheduling:

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

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

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

Step 7: Calculate

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

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

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\tBurst Time \tWaiting Time\tTurnaround Time";
for(i=0;i<n;i++)
{
    tat[i]=bt[i]+wt[i]; total+=tat[i];
    cout<<"\nP["<<p[i]<<"]\t\t "<<bt[i]<<"\t\t "<<wt[i]<<"\t\t"<<tat[i];
}
avg_tat=total/n;
cout<<"\n\nAverage Waiting Time="<<avg_wt;
cout<<"\n\nAverage Turnaround Time="<<avg_tat;

return 0;
}

```

e) Output:

```
Enter Total Number of Process:4
Enter Burst Time and Priority
P[1]
Burst Time:5
Priority:4
P[2]
Burst Time:7
Priority:2
P[3]
Burst Time:2
Priority:1
P[4]
Burst Time:9
Priority:3
Process  Burst Time    Waiting Time    Turnaround Time
P[3]           2           0             2
P[2]           7           2             9
P[4]           9           9            18
P[1]           5          18            23
Average Waiting Time=7
Average Turnaround Time=13
```

f) Result:

Average Waiting Time:7

Average Turnaround Time:13

Experiment-6

BANKER ALGORITHM

Name of Student:AKASH JAIN		Class: CSIT-1
Enrollment No:0827CI201017		Batch:2020-2024
Date of Experiment	Date of Submission	Submitted on:
Remarks by faculty:		Grade:
Signature of student:		Signature of Faculty:

OBJECTIVE OF THE EXPERIMENT

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

FACILITIES REQUIRED

a) Facilities Required To Do The Experiment:

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

b) Concept Of BANKER'S Algorithm:

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

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

```
#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<<"\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;

}

```

e) 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 4 3 1 2 2 6 0 0 0 1 1 4 3 1
Enter the availability matrix :
3 3 2
The final availability matrix
10 4 7
----- Result -----
There is no deadlock
-----
```

f) Result:

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

Experiment-7

FIFO PAGE REPLACEMENT

Name of Student: AKASH JAIN		Class: CSIT-1	
Enrollment No: 0827CI201017		Batch: 2020-2024	
Date of Experiment	Date of Submission		Submitted on:
Remarks by faculty:		Grade:	
Signature of student:		Signature of Faculty:	

OBJECTIVE OF THE EXPERIMENT

To implement page replacement algorithm FIFO.

FACILITIES REQUIRED

a) Facilities Required To Do The Experiment:

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

b) Concept Fifo Page Replacement:

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

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

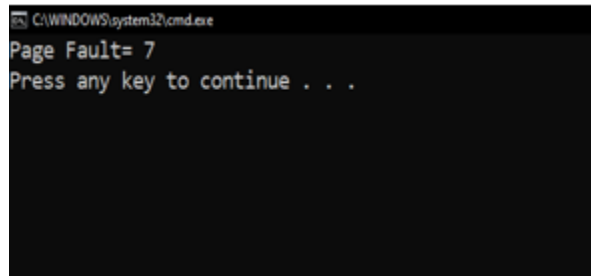
Step 2: When the page is required replace the page at the head of the queue

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

d) Program:

```
#include<bits/stdc++.h>
using namespace std;
int pageFaults(int pages[], int n, int c)
{
    unordered_set<int> v; queue<int> indexes;
    int page_faults = 0;
    for (int i=0; i<n; i++)
    {
        if (v.size() < c)
        {
            if (v.find(pages[i])==v.end())
            {
                v.insert(pages[i]); page_faults++; indexes.push(pages[i]);
            }
        }
        else
        {
            if (v.find(pages[i]) == v.end())
            {
                int val = indexes.front();
                indexes.pop();
                v.erase(val);
                v.insert(pages[i]);
                indexes.push(pages[i]);
                page_faults++;
            }
        }
    }
    return page_faults;
}
int main()
{
    int pages[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2};
    int n = 13;
    int capacity = 4;
    cout <<"Page Fault= "<< pageFaults(pages, n, capacity);
    return 0;
}
```

e) Output:



```
C:\WINDOWS\system32\cmd.exe
Page Fault= 7
Press any key to continue . . .
```

f) Result:

No. of page faults: 7

Experiment-8

LRU PAGE REPLACEMENT

Name of Student:AKASH JAIN		Class: CSIT-1	
Enrollment No:0827CI201017		Batch:2020-2024	
Date of Experiment	Date of Submission		Submitted on:
Remarks by faculty:		Grade:	
Signature of student:		Signature of Faculty:	

OBJECTIVE OF THE EXPERIMENT

To implement page replacement algorithm LRU.

FACILITIES REQUIRED

a) Facilities Required To Do The Experiment:

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

b) Concept of LRU Algorithm:

Pages that have been heavily used in the last few instructions will probably be heavily used 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

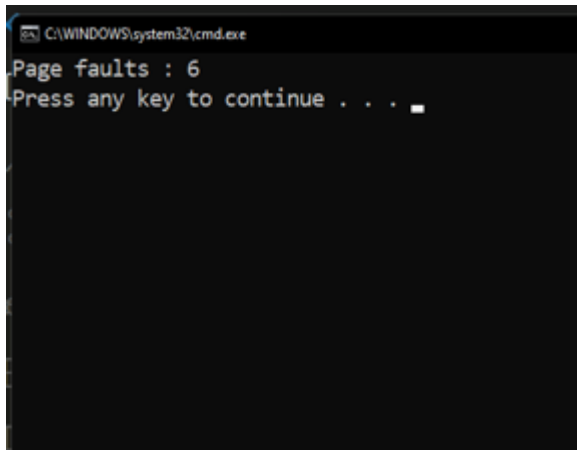
Step 5: When the page fault occurs replace page present at the bottom of the stack

d) Program:

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

int main()
{
    int capacity = 4;
    int arr[] = { 1,2,3,2,1,5,2,1,6,2,5,6,3,1,3,6,1,2,4,3 };
    deque<int> q(capacity);
    int count=0;
    int page_faults=0; deque<int>::iterator itr; q.clear();
    for(int i:arr)
    {
        itr = find(q.begin(),q.end(),i);
        if(!(itr != q.end()))
        {
            ++page_faults; if(q.size() == capacity)
            {
                q.erase(q.begin()); q.push_back(i);
            }
        }
        else{ q.push_back(i);
        }
    }
    else
    {
        q.erase(itr); q.push_back(i);
    }
}
cout<<"Page faults : "<<page_faults;
}
```

e) OUTPUT:

A screenshot of a Windows command prompt window. The title bar at the top reads "C:\WINDOWS\system32\cmd.exe". The command prompt shows two lines of text: "Page faults : 6" and "Press any key to continue . . .". The cursor is positioned at the end of the second line.

```
C:\WINDOWS\system32\cmd.exe
Page faults : 6
Press any key to continue . . .
```

f) Result:

No. of pages faults : 6

Experiment-9

FCFS Disk Scheduling Algorithm

Name of Student:AKASH JAIN		Class: CSIT-1
Enrollment No:0827CI201017		Batch:2020-2024
Date of Experiment	Date of Submission	Submitted on:
Remarks by faculty:		Grade:
Signature of student:		Signature of Faculty:

OBJECTIVE OF THE EXPERIMENT

To implement FCFS Disk Scheduling Algorithm

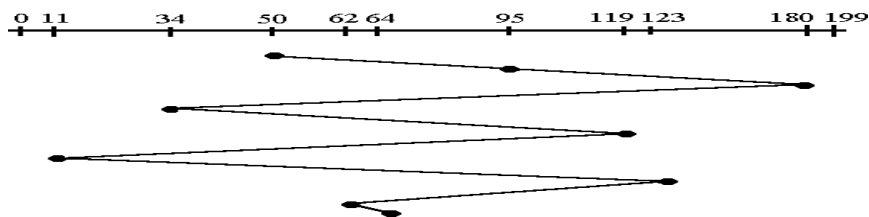
FACILITIES REQUIRED

a) Facilities Required To Do The Experiment:

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

b) Concept of FCFS Disk Scheduling Algorithm:

All incoming requests are placed at the end of the queue. Whatever number that is next 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

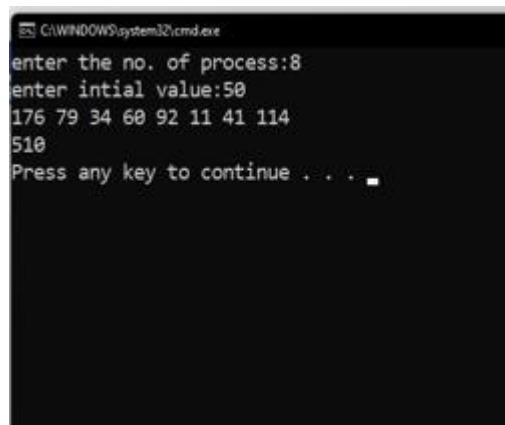
Step 2: Move the head to the request in FIFO order (Serve the request first that came first)

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

d) Program:

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

e) Output:



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

f) Result:

Total Head Movement Required Serving All Requests: 7

Experiment-10

SSTF Disk Scheduling Algorithm

Name of Student: AKASH JAIN		Class: CSIT-1
Enrollment No: 0827CI201017		Batch: 2020-2024
Date of Experiment	Date of Submission	Submitted on:
Remarks by faculty:		Grade:
Signature of student:		Signature of Faculty:

OBJECTIVE OF THE EXPERIMENT

To implement SSTF Disk Scheduling Algorithm

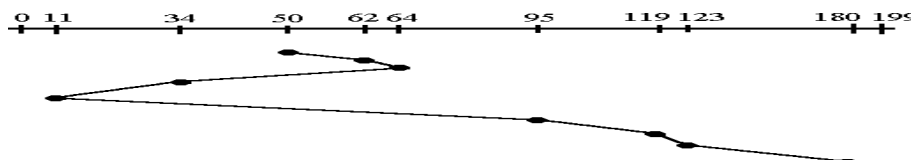
FACILITIES REQUIRED

a) Facilities Required To Do The Experiment:

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

b) Concept of SSTF Disk Scheduling Algorithm:

In this case request is serviced according to next shortest distance. Starting at 50, the 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:

```
#include <bits/stdc++.h>
using namespace std;
int minDiff(int *req, int pos, int n)
{
    int newpos;
    int mini = INT_MAX;
    int diff;
    for (int i = 0; i < n; i++)
    {
        if (req[i] != -1)
        {
            diff = abs(pos - req[i]);
            if (mini > diff)
            {
                mini = diff; newpos = i;
            }
        }
    }
    return newpos;
}
float SSTF(int *req, int pos, int n)
{
    int posi = pos;
    float total = 0;
    for (int i = 0; i < n; i++)
    {
        int index = minDiff(req, posi, n);
        // cout<<"diff: "<<abs(pos-req[index]); total += abs(posi - req[index]);
        posi = req[index]; req[index] = -1;
    }
    float avg = total / n; return avg;
}

int main()
{
    int n, positom;
    cout << "Enter the number of requests:";
    cin >> n;
    int req[n];
    cout << "Enter the requests:";
    for (int i = 0; i < n; i++)
    {
        cin >> req[i];
    }
}
```

```
    }  
    cout << "Enter the position where initially circular arm is present:";  
    cin >> positom;  
    cout << SSTF(req, positom, n);  
    return 0;  
}
```

e) Output:

```
C:\WINDOWS\system32\cmd.exe
Enter the number of requests:9
Enter the requests:55
58
39
18
90
160
150
38
184
Enter the positon where initially circular arm is present:100
27.5556
Press any key to continue . . .
```

f) Result:

Total Head Movement Required Serving All Requests : 27.55

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?