# Experiment-1

## FCFS SCHEDULING

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Name of Student : Atharva Sharma | | | | Class : CI-1 | |
| Enrollment No : 0827CI201044 | | | | 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.

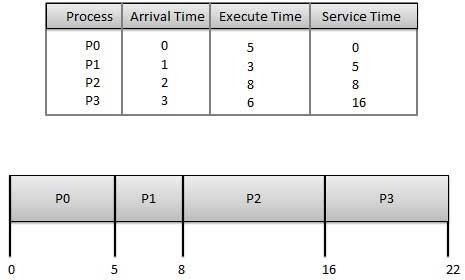
## FACILITIES REQUIRED

1. **Facilities Required Doing The Experiment:**

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

## Concept of FCFS:

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



## 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

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

Step 6: Calculate

1. Average waiting time = Total waiting Time / Number of process
2. Average Turnaround time = Total Turnaround Time / Number of process Step 7: Stop the process

## Program :

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

## using namespace std;

## class fcfs

## {

## public :

## 

## void completion\_time( int burst\_time [] ,int n ,int cmp\_time[])

## {

## cmp\_time[0] = burst\_time[0];

## 

## for(int i=1 ; i<n; i++)

## {

## cmp\_time[i] = cmp\_time[i-1]+burst\_time[i];

## }

## 

## }

## 

## void turnaround\_time(int cmp\_time[] ,int n ,int arr\_time[] , int turn\_time[] )

## {

## for(int k=0; k<n ; k++)

## {

## turn\_time[k] = cmp\_time[k] - arr\_time[k];

## }

## 

## }

## 

## void waiting\_time(int burst\_time[] ,int n ,int waiting\_time[] , int turn\_time[] )

## {

## 

## for(int k=0; k<n ; k++)

## {

## waiting\_time[k] = turn\_time[k] - burst\_time[k];

## }

## 

## }

## 

## 

## };

## int main()

## {

## fcfs ob;

## 

## int bt[4] = {3,4,6,7};

## int ct[4];

## int tt[4];

## int wt[4];

## int at[4] ={3,1,0,2};

## sort(at ,at+4) ;

## 

## ob.completion\_time(bt , 4 , ct);

## ob.turnaround\_time(ct , 4 , at, tt);

## ob.waiting\_time(bt , 4 , wt , tt);

## 

## 

## for(int i = 0 ; i< 4 ;i++)

## {

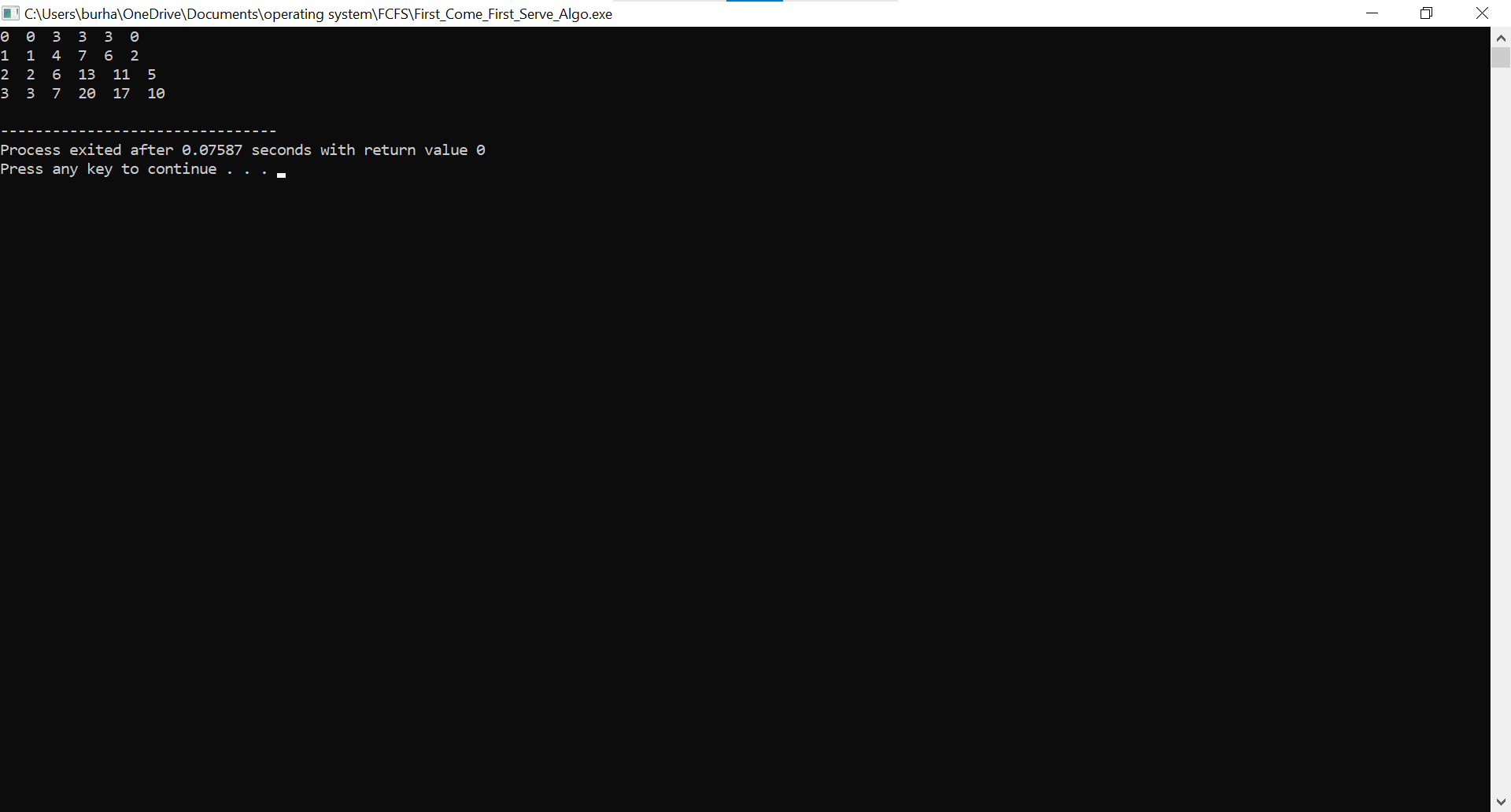
## cout<<i<<" "<<at[i]<<" "<<bt[i]<<" "<<ct[i]<<" "<<tt[i]<<" "<<wt[i]<<endl;

## }

## return 0;

## }

1. **Output:**



## Result:

Average Waiting Time ……..

Average Turnaround Time ……….

# Experiment-2

## SJF Scheduling

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Name of Student: Atharva Sharma | | | | Class: CI-1 | |
| Enrollment No: 0827CI201044 | | | | 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.

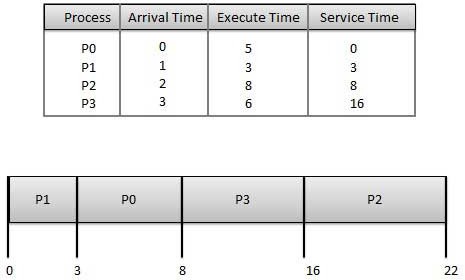
## FACILITIES REQUIRED

1. **Facilities Required Doing The Experiment:**

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

## Concept of SJF:

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



## 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

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

Step 7: Calculate

1. Average waiting time = Total waiting Time / Number of process
2. Average Turnaround time = Total Turnaround Time / Number of process Step 8: Stop the process

## Program:

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

## using namespace std;

## struct data

## {

## int pid;

## int bt;

## int at;

## int wt;

## int tat;

## };

## bool cmp( data x , data y)

## {

## return ( x.bt<y.bt);

## }

## 

## 

## int main()

## {

## data a[30];

## 

## int i,k,n;

## float wtavg , tatavg;

## 

## cout<<"Enter the Number of process"<<endl;

## cin>>n;

## for(i=0;i<n;i++)

## {

## a[i].pid =i;

## a[i].at=0;

## cout<<"Enter The Burst time for the process "<<i<<endl;

## cin>>a[i].bt;

## }

## 

## sort( a , a+n , cmp);

## a[0].wt =0;

## a[0].tat = a[0].bt;

## 

## for(i=1;i<n;i++)

## {

## a[i].wt = a[i-1].wt + a[i-1].bt;

## a[i].tat = a[i-1].tat + a[i].bt;

## }

## cout<<endl<<"P.id\tBt\tWt\tTat"<<endl;

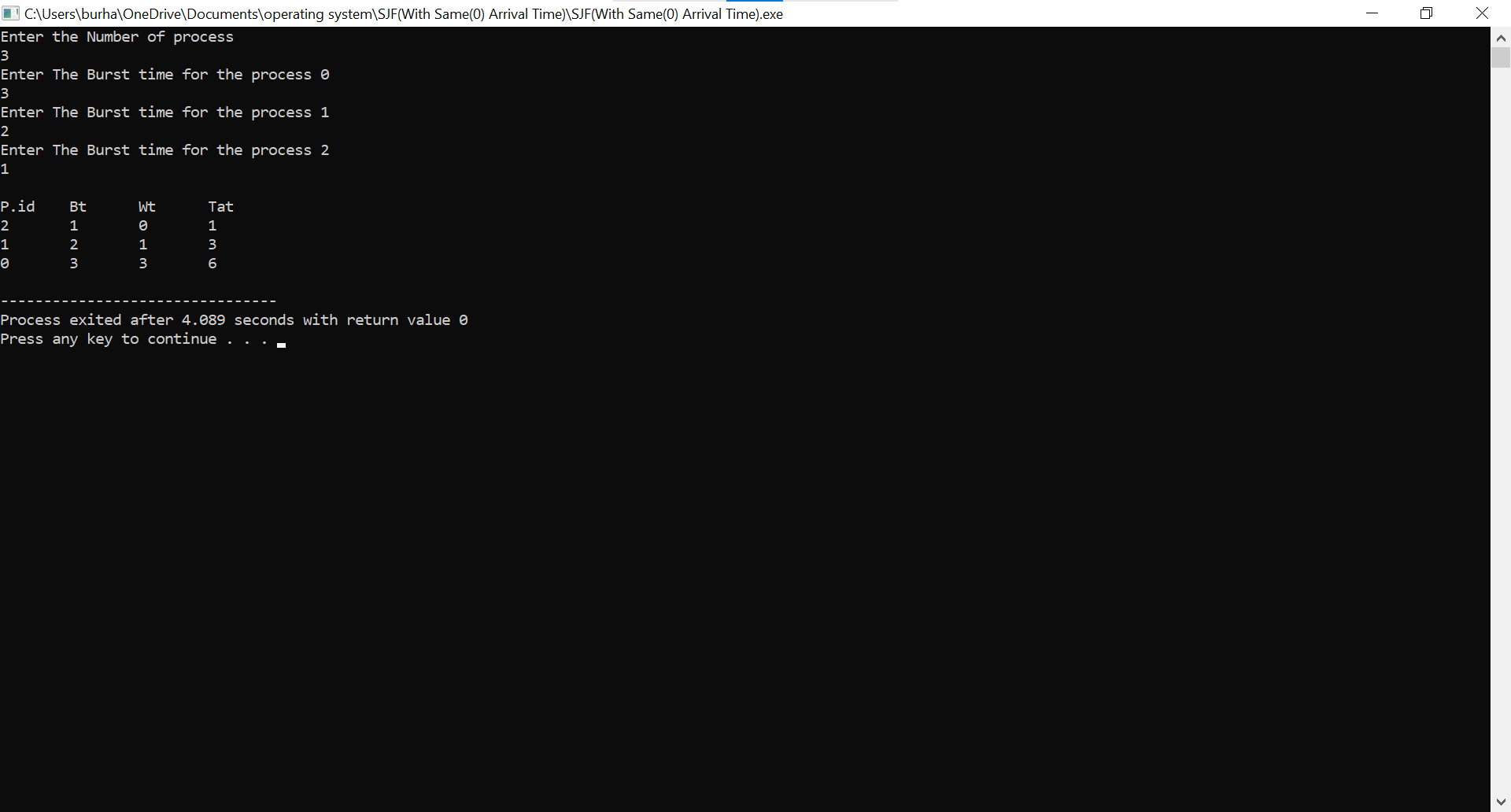
## for(i=0;i<n;i++)

## {

## cout<<a[i].pid<<"\t"<<a[i].bt<<"\t"<<a[i].wt<<"\t"<<a[i].tat<<endl;

## }

1. **Output:**



## Result:

Average Waiting Time ……..

Average Turnaround Time

# Experiment-3

## ROUND ROBIN Scheduling

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Name of Student: Atharva Sharma | | | | Class: CI-1 | |
| Enrollment No: 0827CI201044 | | | | 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

1. **Facilities Required To Do The Experiment:**

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

## Concept Of Round Robin Scheduling:

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

|  |  |
| --- | --- |
|  | Burst |
| Process | Time |
|  | 24 |
|  | 3 |

|  |  |
| --- | --- |
|  | 3 |
| Average |  |



## Algorithm:

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue and time quantum (or) time slice Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time Step 4: Calculate the no. of time slices for each process where

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

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

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

* 1. 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)

* 1. 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

1. Average waiting time = Total waiting Time / Number of process
2. Average Turnaround time = Total Turnaround Time / Number of process Step 8: Stop the process

## Program:

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

## using namespace std;

## void RoundRobin() {

## int n, qt;

## cout << "enter number of processes" << endl;

## cin >> n;

## int p[n];

## int Bt[n];

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

## p[i] = i + 1;

## }

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

## cout << "enter burst of processes" << endl;

## cin >> Bt[i];

## }

## cout << "enter quantum time" << endl;

## cin >> qt;

## cout << "Ready Queue :- " << endl;

## int tot = 0;

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

## tot = tot + Bt[i];

## }

## int x = 0;

## while (x <= tot) {

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

## if (Bt[i] > qt) {

## cout << p[i] << " ";

## x = x + qt;

## Bt[i] = Bt[i] - qt;

## } else if (Bt[i] <= qt and Bt[i] != 0) {

## x = x + qt;

## cout << p[i] << " ";

## Bt[i] = 0;

## }

## }

## }

## }

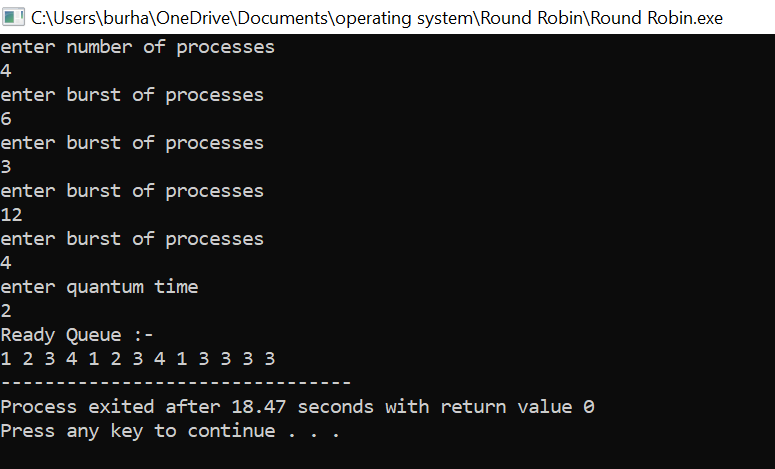
## int main() {

## RoundRobin();

## return 0;

## }

1. **Output:**



## Result:

Average Waiting Time ……..

Average Turnaround Time ……….

# Experiment-4

## BANKER ALGORITHM

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Name of Student: Atharva Sharma | | | | Class: CI-1 | |
| Enrollment No: 0827CI201044 | | | | 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 deadlock avoidance & Prevention by using Banker’s Algorithm.

## FACILITIES REQUIRED

1. **Facilities Required To Do The Experiment:**

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

## Concept Of BANKER’S Algorithm:

The Banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation of predetermined maximum possible 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.

## Algorithm:

1. Start the program.
2. Get the values of resources and processes.
3. Get the avail value.
4. After allocation find the need value.
5. Check whether it’s possible to allocate.
6. If it is possible then the system is in safe state.
7. Else system is not in safety state.
8. If the new request comes then check that the system is in safety.
9. Or not if we allow the request.
10. Stop the program.

## Program:

## #include<iostream>

## using namespace std;

## int alloc[50][50];

## int maxi[50][50];

## int need[50][50];

## int avail[50];

## int check\_safety(int j,int nr)

## {

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

## {

## if(need[j][i]>avail[i])

## return 0;

## }

## return 1;

## }

## int check(bool a[],int n)

## {

## for(int i=0;i<n;i++)

## {

## if(a[i]==false)

## return 0;

## }

## return 1;

## }

## int main()

## {

## int np=100;

## int nr=100;

## cout<<"\nEnter the no of processes : ";

## cin>>np;

## cout<<"\nEnter the no of resources : ";

## cin>>nr;

## cout<<"\nEnter the allocation data : \n";

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

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

## cin>>alloc[i][j];

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

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

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

## cin>>maxi[i][j];

## 

## 

## 

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

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

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

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

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

## cin>>avail[i];

## int ex\_it=nr;

## int flg;

## bool completed[np];

## while(10)

## {

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

## {

## if(!completed[i] && check\_safety(i,nr))

## {

## 

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

## avail[j]+=alloc[i][j];

## }

## 

## completed[i]=true;

## }

## flg=check(completed,np);

## ex\_it--;

## if(flg==1 || ex\_it==0)

## break;

## }

## 

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

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

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

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

## if(flg==1)

## cout<<"There is no deadlock";

## else

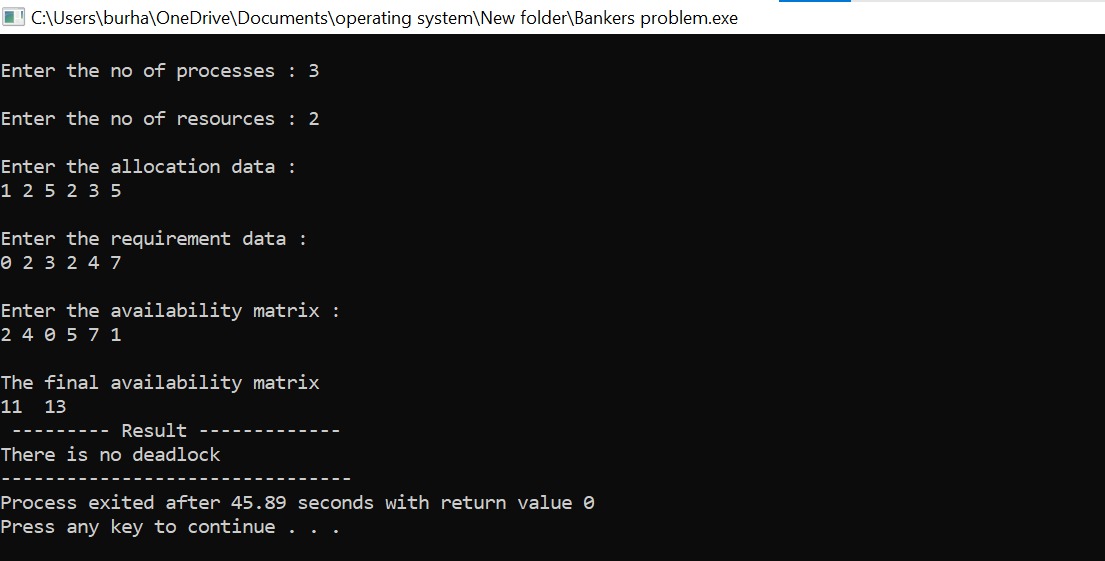
## cout<<"Sorry there is a possibility of deadlock";

## 

## return 0;

## }

1. **Output:**



## Result:

The Sequence Is:

# Experiment-5

## FIFO PAGE REPLACEMENT

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Name of Student: Atharva Sharma | | | | Class: CI-1 | |
| Enrollment No: 0827CI201044 | | | | 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 FIFO.

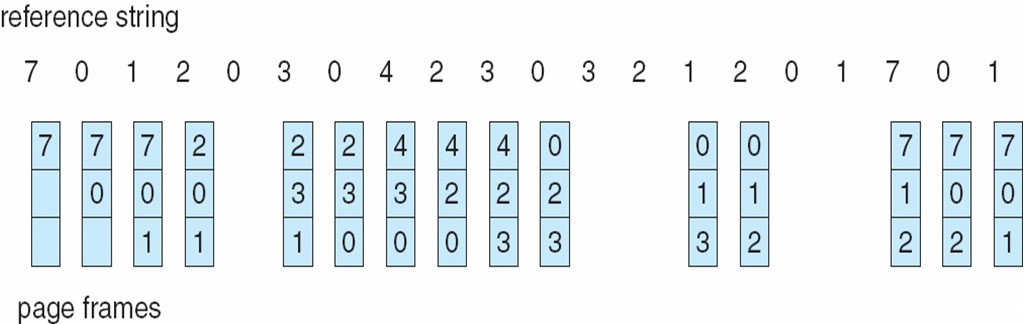
## FACILITIES REQUIRED

1. **Facilities Required To Do The Experiment:**

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

## Concept Fifo Page Replacement:

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



* FIFO Replacement manifests Belady’s Anomaly: more frames  more page faults
* Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5() 3 Frames:-9 page fault

4 Frames: - 10 page fault

## Algorithm:

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

Step 2: When the page is required replace the page at the head of the queue Step 3: Now the new page is inserted at the tail of the queue

## Program:

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

## using namespace std;

## int PageFaults(int pages[], int n, int frames)

## {

## 

## unordered\_set <int> set;

## 

## queue <int> indexes;

## 

## 

## int countPageFaults = 0;

## for (int i=0; i < n; i++)

## {

## 

## if (set.size() < frames)

## {

## 

## if (set.find(pages[i])==set.end())

## {

## set.insert(pages[i]);

## 

## countPageFaults++;

## 

## 

## indexes.push(pages[i]);

## }

## }

## else

## {

## 

## if (set.find(pages[i]) == set.end())

## {

## 

## int val = indexes.front();

## indexes.pop();

## 

## 

## set.erase(val);

## 

## 

## set.insert(pages[i]);

## indexes.push(pages[i]);

## 

## 

## countPageFaults++;

## }

## }

## }

## return countPageFaults;

## }

## int main()

## {

## int pages[] = {7,0,1,2,0,3,0,4,2,3,0,3,2};

## int n = sizeof(pages)/sizeof(pages[0]);

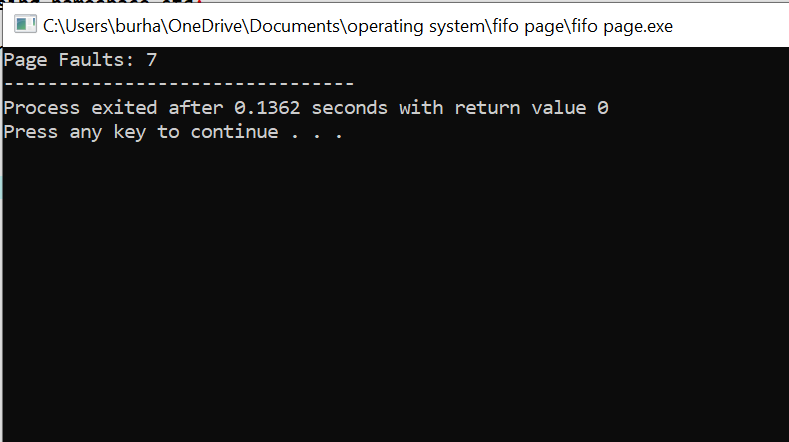
## int frames = 4;

## cout << "Page Faults: " << PageFaults(pages, n, frames);

## return 0;

## }

1. **Output:**



## Result:

No. of page faults......

# Experiment-6

## LRU PAGE REPLACEMENT

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Name of Student: Atharva Sharma | | | | Class: CI-1 | |
| Enrollment No: 0827CI201044 | | | | 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.

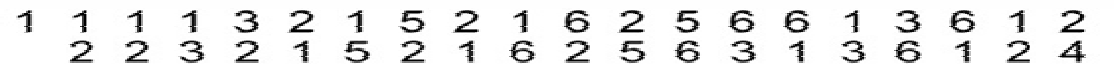
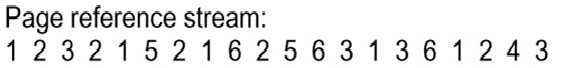
## FACILITIES REQUIRED

1. **Facilities Required To Do The Experiment:**

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

## Concept of LRU Algorithm:

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



## Algorithm:

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

Step 2: When the page is required replace the page at the head of the queue Step 3: Now the new page is inserted at the tail of the queue

Step 4: Create a stack

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

## Program:

## #include <iostream>

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

## using namespace std;

## int pageFault(int pages[], int n, int memcapacity){

## int pagefault = 0;

## vector <int> v;

## int i;

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

## auto it= find(v.begin(), v.end(), pages[i]);

## if(it==v.end()){

## if(v.size()==memcapacity){

## v.erase(v.begin());

## }

## v.push\_back(pages[i]);

## pagefault++;

## }

## else{

## v.erase(it);

## v.push\_back(pages[i]);

## }

## }

## return pagefault;

## }

## int main() {

## int pages[] = {7,0,1,2,0,3,0,4,2,3,0,3,2};

## int n=13;

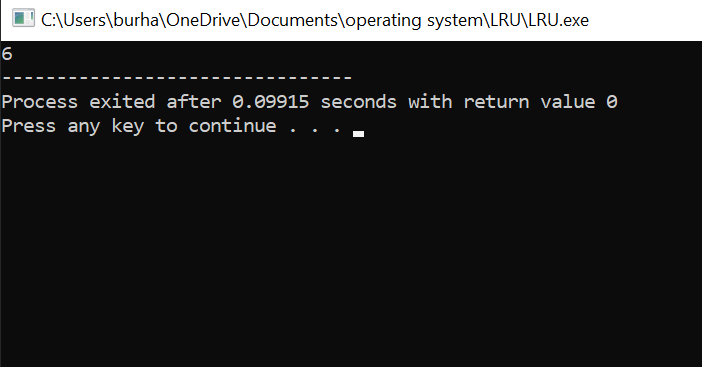
## int memcapacity = 4;

## cout<<pageFault(pages, n, memcapacity);

## return 0;

## }

1. **Output:**



## Result:

No. of pages faults …….

# Experiment-7

## FCFS Disk Scheduling Algorithm

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Name of Student: Atharva Sharma | | | | Class: CI-1 | |
| Enrollment No: 0827CI201044 | | | | 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 FCFS Disk Scheduling Algorithm

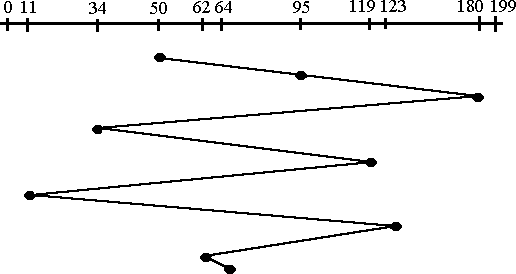
## FACILITIES REQUIRED

* 1. **Facilities Required To Do The Experiment:**

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

## Concept of FCFS Disk Scheduling Algorithm:

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



## Algorithm:

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

Step 2: Move the head to the request in FIFO order (Serve the request first that came first) Step 3: Calculate the total head movement required to serve all request.

## Program:

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

## using namespace std;

## class Diskscheduling

## {

## public :

## float FCFS ( int arr[] , int n , int init)

## {

## float ans = abs (init - arr[0]);

## for(int i=1;i<n;i++)

## {

## ans = ans + abs(arr[i]-arr[i-1]);

## }

## 

## return (ans/n);

## }

## };

## int main()

## {

## int n;

## cout<<"Enter the No. of disk"<<endl;

## cin>>n;

## int disk[n];

## cout<<"Enter the disk sequence"<<endl;

## for(int i=0;i<n;i++)

## {

## cin>>disk[i];

## }

## 

## cout<<"Enter the Position of Head"<<endl;

## int init;

## cin>>init;

## 

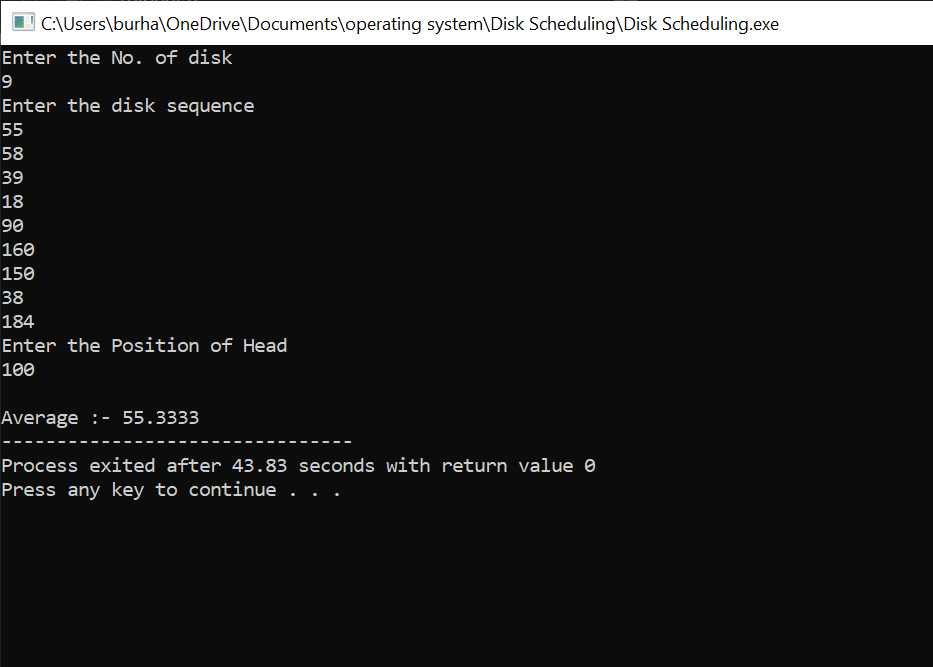
## Diskscheduling obj;

## cout<<endl<<"Average :- "<<obj.FCFS(disk,n,init);

## 

## }

* 1. **Output:**



## Result:

Total Head Movement Required Serving All Requests ………

# Experiment-8

## SSTF Disk Scheduling Algorithm

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Name of Student: Atharva Sharma | | | | Class: CI-1 | |
| Enrollment No: 0827CI201044 | | | | 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 SSTF Disk Scheduling Algorithm

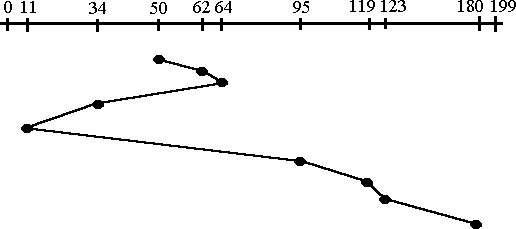
## FACILITIES REQUIRED

1. **Facilities Required To Do The Experiment:**

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

## Concept of SSTF Disk Scheduling Algorithm:

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



## Algorithm:

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

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

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

## Program:

#include<iostream>

#include<conio.h>

#include<math.h>

using namespace std;

int main()

{

int queue[100],t[100];

int head,seek=0,n,i,j,temp;

float avg;

cout<<"Enter the size of Queue: ";

cin>>n;

cout<<"Enter the Queue: ";

for(i=0;i<n;i++)

cin>>queue[i];

cout<<"Enter the initial head position: ";

cin>>head;

for(i=1;i<n;i++)

t[i]=abs(head-queue[i]);

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

for(j=i+1;j<n;j++){

if(t[i]>t[j]){

temp=t[i];

t[i]=t[j];

t[j]=temp;

temp=queue[i];

queue[i]=queue[j];

queue[j]=temp;

}

}

}

for(i=1;i<n-1;i++){

seek=seek+abs(head-queue[i]);

head=queue[i];

}

cout<<"Total Seek Time is "<<seek<<endl;

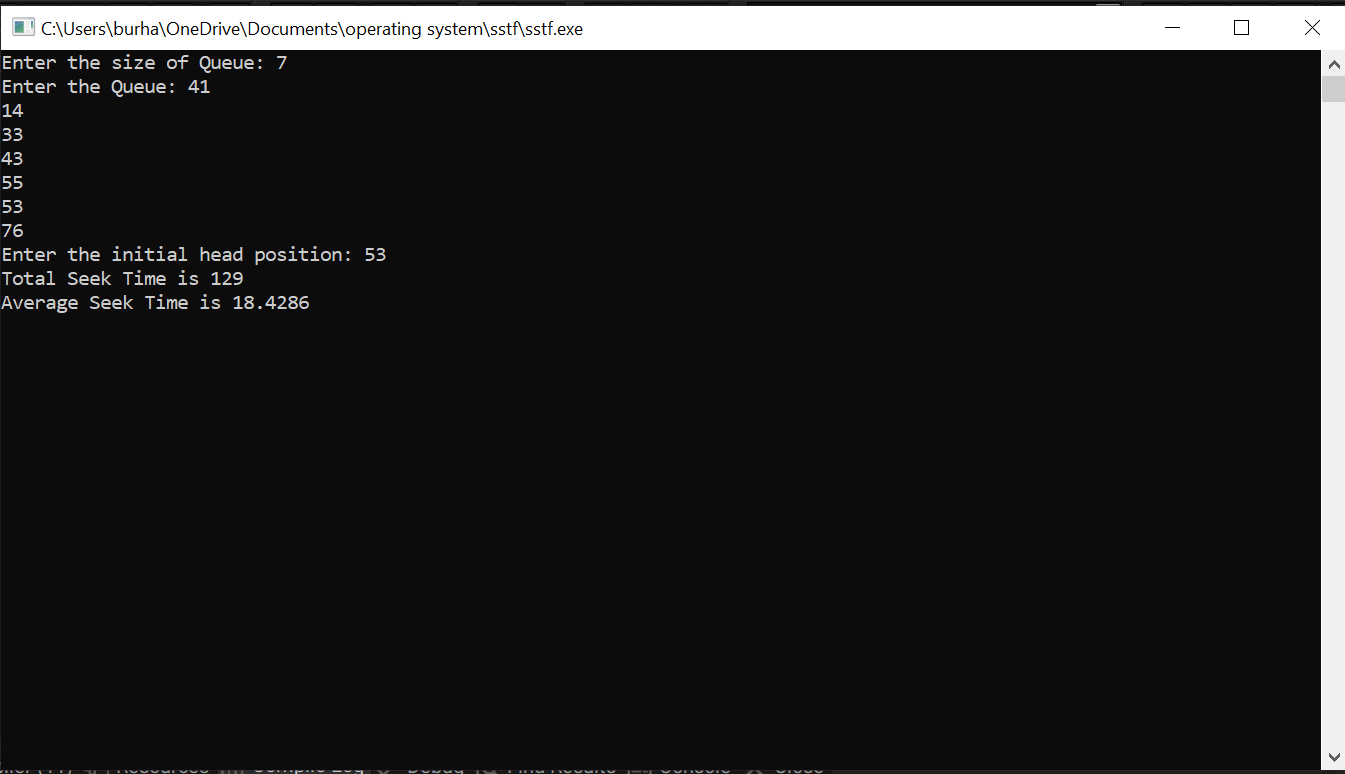
avg=seek/(float)n;

cout<<"Average Seek Time is "<<avg;

return 0;

}

1. **Output:**

****

## Result:

Total Head Movement Required Serving All Requests …….

# Experiment-9

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| P1 | P2 | P3 | P2 | P4 | P1 |

## SRTF Scheduling

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Name of Student: Atharva Sharma | | | | Class: CI-1 | |
| Enrollment No: 0827CI201044 | | | | 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

1. **Facilities Required To Do The Experiment:**

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

## Concept Of SRTF Scheduling:

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

Example of Preemptive SJF

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Process | Arrival Time |  | Burst Time |  | |
| P1 | 0.0 |  | 7 |  |  |
| P2 | 2.0 |  | 4 |  |  |
| P3 | 4.0 |  | 1 |  |  |
| P4 | 5.0 |  | 4 |  |  |
| SJF (preemptive) |  |  |  |  |  |
|  |  |  |  |  |  |  |
| 0 | 2 | 4 | 5 | 7 | 11 | 16 |

## Algorithm:

Step 1: Start the process

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

Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time Step 4: For each process in the ready Q, Accept Arrival time

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

Step 5: Set the waiting time of the first process in Sorted Q as ‘0’.

Step 6: After every unit of time compare the remaining time of currently executing process (RT) and Burst time of newly arrived process (BTn).

Step 7: If the burst time of newly arrived process (BTn) is less than the currently executing process (RT) the processor will preempt the currently executing process and starts executing newly arrived process

Step 7: Calculate

1. Average waiting time = Total waiting Time / Number of process
2. Average Turnaround time = Total Turnaround Time / Number of process Step 8: Stop the process

## Program:

## #include<iostream>

## using namespace std;

## int main()

## {

## int a[10],b[10],x[10];

## int waiting[10],turnaround[10],completion[10];

## int i,j,smallest,count=0,time,n;

## double avg=0,tt=0,end;

## cout<<"\nEnter the number of Processes: "; //input

## cin>>n;

## for(i=0; i<n; i++)

## {

## cout<<"\nEnter arrival time of process: "; //input

## cin>>a[i];

## }

## for(i=0; i<n; i++)

## {

## cout<<"\nEnter burst time of process: "; //input

## cin>>b[i];

## }

## for(i=0; i<n; i++)

## x[i]=b[i];

## b[9]=9999;

## for(time=0; count!=n; time++)

## {

## smallest=9;

## for(i=0; i<n; i++)

## {

## if(a[i]<=time && b[i]<b[smallest] && b[i]>0 )

## smallest=i;

## }

## b[smallest]--;

## if(b[smallest]==0)

## {

## count++;

## end=time+1;

## completion[smallest] = end;

## waiting[smallest] = end - a[smallest] - x[smallest];

## turnaround[smallest] = end - a[smallest];

## }

## }

## cout<<"Process"<<"\t"<< "burst-time"<<"\t"<<"arrival-time" <<"\t"<<"waiting-time" <<"\t"<<"turnaround-time"<< "\t"<<"completion-time"<<endl;

## for(i=0; i<n; i++)

## {

## cout<<"p"<<i+1<<"\t\t"<<x[i]<<"\t\t"<<a[i]<<"\t\t"<<waiting[i]<<"\t\t"<<turnaround[i]<<"\t\t"<<completion[i]<<endl;

## avg = avg + waiting[i];

## tt = tt + turnaround[i];

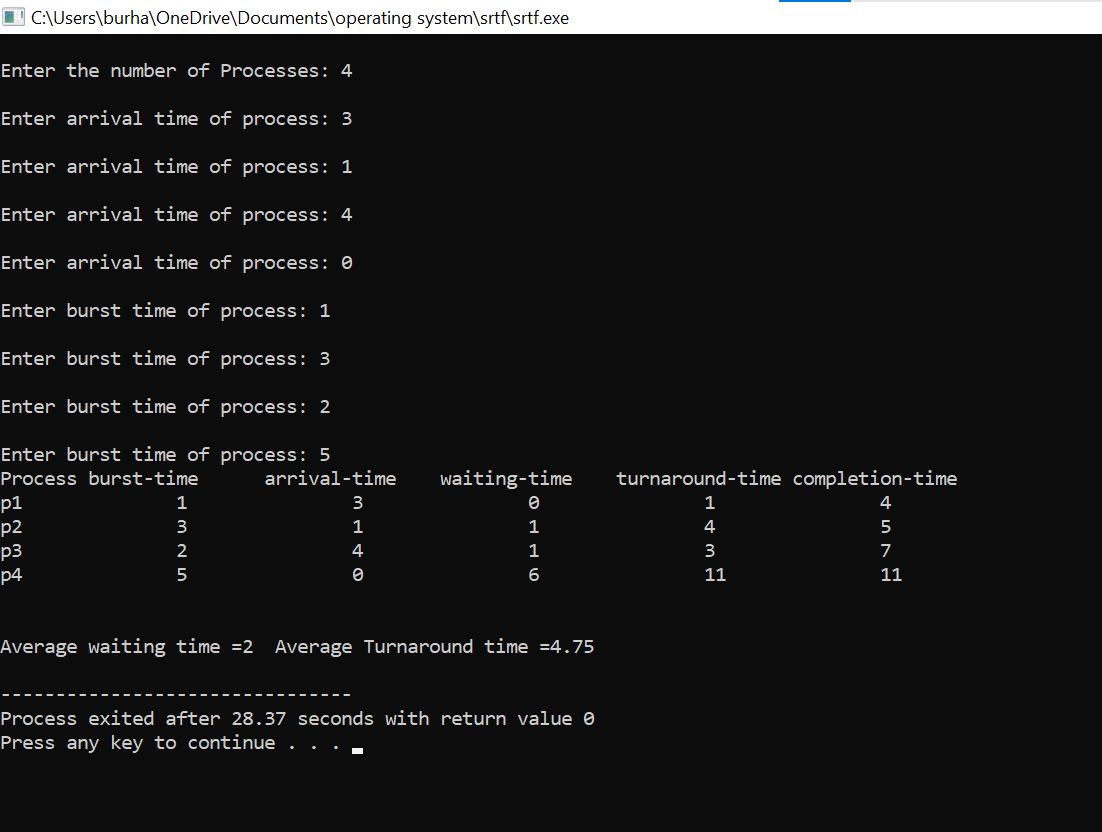
## }

## cout<<"\n\nAverage waiting time ="<<avg/n;

## cout<<" Average Turnaround time ="<<tt/n<<endl;

## }

1. **Output:**

****

## Result:

Average Waiting Time ……..

Average Turnaround Time …

# Experiment-10

## PRIORITY SCHEDULING

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Name of Student: Atharva Sharma | | | | Class: CI-1 | |
| Enrollment No: 0827CI201044 | | | | 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

1. **Facilities Required To Do The Experiment:**

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

## Concept Of Priority Scheduling:

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

An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa.

As an example, consider the following set of processes, assumed to have arrived at time 0, in the order

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Burst |  | Waiting | Turnaround |
| Process | Time | Priority | Time | Time |
|  | 10 | 3 | 6 | 16 |
|  | 1 | 1 | 0 | 1 |
|  | 2 | 4 | 16 | 18 |
|  | 1 | 5 | 18 | 19 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 5 | 2 | 1 | 6 |
| Average | - | - | 8.2 | 12 |



## Algorithm:

Step 1: Start the process

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

Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time Step 4: Sort the ready queue according to the priority number.

Step 5: Set the waiting of the first process as ‘0’ and its burst time as its turn around time Step 6: For each process in the Ready Q calculate

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

Step 7: Calculate

1. Average waiting time = Total waiting Time / Number of process
2. Average Turnaround time = Total Turnaround Time / Number of process Step 8: Stop the process

## Program:

#include<iostream>

using namespace std;

int main() {

int bt[20],p[20],wt[20],tat[20],pr[20],i,j,n,total=0,pos,temp,avg\_wt,avg\_tat;

cout<<"Enter Total Number of Process:"<<endl;

cin>>n;

cout<<"Enter Burst Time and Priority"<<endl;

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

cin>>bt[i];

cin>>pr[i];

p[i]=i+1;

}

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

pos=i;

for (j=i+1;j<n;j++) {

if(pr[j]<pr[pos])

pos=j;

}

temp=pr[i];

pr[i]=pr[pos];

pr[pos]=temp;

temp=bt[i];

bt[i]=bt[pos];

bt[pos]=temp;

temp=p[i];

p[i]=p[pos];

p[pos]=temp;

}

wt[0]=0;

for (i=1;i<n;i++) {

wt[i]=0;

for (j=0;j<i;j++)

wt[i]+=bt[j];

total+=wt[i];

}

avg\_wt=total/n;

total=0;

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

tat[i]=bt[i]+wt[i];

total+=tat[i];

}

avg\_tat=total/n;

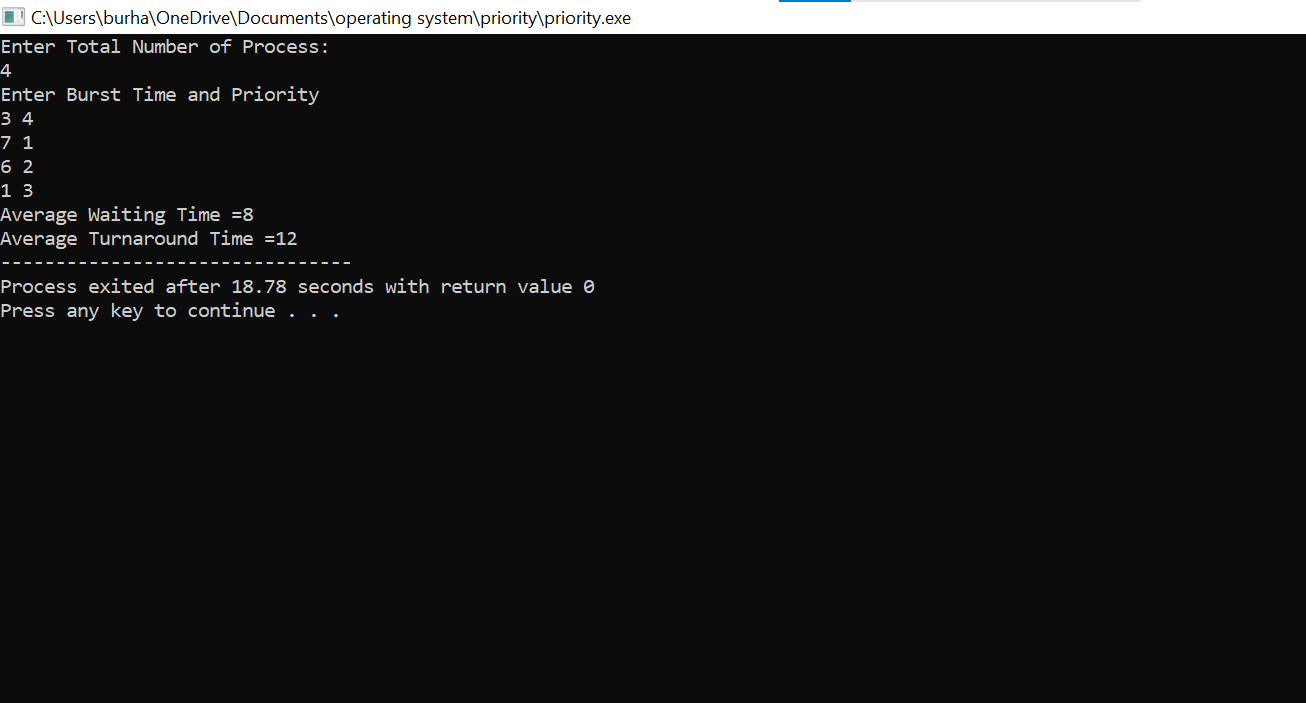
cout<<"Average Waiting Time ="<<avg\_wt<<endl;

cout<<"Average Turnaround Time ="<<avg\_tat;

return 0;

}

1. **Output:**

****

## Result:

Average Waiting Time ……..

Average Turnaround Time ……….