**Air University Aerospace & Aviation**

**Kamra Campus**



**[Scheduling Algorithm]**

**Project Report**

**OS (225L)**

**205128 Hafsa Naeem**

**Objective:**

Code and implement all four-scheduling algorithm in C++ Language.

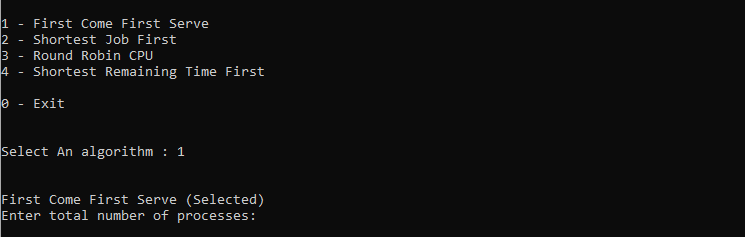
**Scope:**

A user can select which scheduling algorithm he/she wants to use.

**Code:**

|  |
| --- |
| while(i!=0){  cout << "\n\n1 - First Come First Serve" << endl;  cout << "2 - Shortest Job First" << endl;  cout << "3 - Round Robin CPU" << endl;  cout << "4 - Shortest Remaining Time First" <<endl;  cout << "\n0 - Exit" <<endl;  cout << "\n\nSelect An algorithm : ";  cin >> i;  if(i==1){  cout << "\n\nFirst Come First Serve (Selected)" << endl;  fcfs();  }  if(i==2){  cout << "\n\nShortest Job First (Selected)" << endl;  sjf();  }  if(i==3){  cout << "\n\nRound Robin CPU (Selected)" << endl;  rr();  }  if(i==4){  cout << "\n\nShortest Remaining Time First (Selected)" <<endl;  srtf();  }    } |

**Output:**



**Scheduling:**

The activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy. By switching the CPU among processes, operating system can make the computer more effective.  
There are four major scheduling algorithms, which are listed below:

* First Come First Serve (FCFS)
* Shortest Job First (SJF)
* Round Robin (RR)
* Shortest Remaining Time First (SRTF)

**First Come First Serve:**

It automatically executes queued requests and processes in order of their arrival. It is the easiest and simplest CPU scheduling algorithm. In this type of algorithm, processes which requests the CPU first get the CPU allocation first.

**Advantages:**

* It is simple and easy to understand.
* Processes arrived first will get the CPU for their execution.

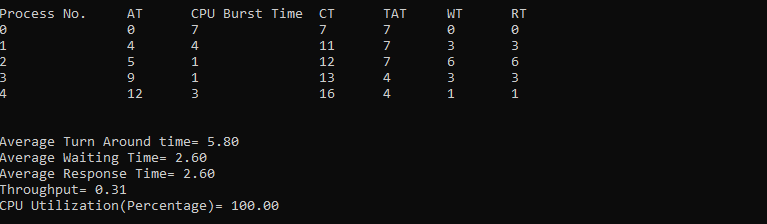
**Disadvantages:**

* The process with less execution time suffers i.e. waiting time is often quite long.
* Favors CPU Bound process then I/O bound process.
* Here, first process will get the CPU first, other processes can get CPU only after the current process has finished its execution. Now, suppose the first process has large burst time, and other processes have less burst time, then the processes will have to wait more unnecessarily, this will result in more average waiting time, i.e., Convey effect.
* This effect results in lower CPU and device utilization.
* FCFS algorithm is particularly troublesome for time-sharing systems, where it is important that each user get a share of the CPU at regular intervals.

**Code:**

|  |
| --- |
| int fcfs(){  int n;  cout<<"Enter total number of processes: ";  cin>>n;  float sum\_tat=0,sum\_wt=0,sum\_rt=0;  int length\_cycle,total\_idle\_time=0;  float cpu\_utilization;  cout << fixed << setprecision(2);  for(int i=0;i<n;i++)  {  cout<<"\nEnter Process" <<i<< "Arrival Time: ";  cin >> ps2[i].at;  ps2[i].pid=i;  }  for(int i=0;i<n;i++)  {  cout<<"\nEnter Process" <<i<< "Burst Time: ";  cin >> ps2[i].bt;  }  //sort  sort(ps2,ps2+n, comparatorAT);  //calculation  for(int i=0;i<n;i++)  {  ps2[i].start\_time = (i==0) ? ps2[i].at : max(ps2[i].at, ps2[i-1].ct);  ps2[i].ct = ps2[i].start\_time + ps2[i].bt;  ps2[i].tat = ps2[i].ct-ps2[i].at;  ps2[i].wt = ps2[i].tat-ps2[i].bt;  ps2[i].rt=ps2[i].wt;    sum\_tat += ps2[i].tat;  sum\_wt += ps2[i].wt;  sum\_rt += ps2[i].rt;  total\_idle\_time += (i==0) ? 0 : (ps2[i].start\_time - ps2[i-1].ct);  }  length\_cycle = ps2[n-1].ct - ps2[0].start\_time;  //sort so that process ID in output comes in Original order (just for interactivity)  sort(ps2,ps2+n, comparatorPID);  //Output  cout<<"\nProcess No.\tAT\tCPU Burst Time\tCT\tTAT\tWT\tRT\n";  for(int i=0;i<n;i++)  cout<<i<<"\t\t"<<ps2[i].at<<"\t"<<ps2[i].bt<<"\t\t"<<ps2[i].ct<<"\t"<<ps2[i].tat<<"\t"<<ps2[i].wt<<"\t"<<ps2[i].rt<<endl;  cout<<endl;  cpu\_utilization = (float)(length\_cycle - total\_idle\_time)/ length\_cycle;  cout<<"\nAverage Turn Around time= "<< sum\_tat/n;  cout<<"\nAverage Waiting Time= "<<sum\_wt/n;  cout<<"\nAverage Response Time= "<<sum\_rt/n;  cout<<"\nThroughput= "<<n/(float)length\_cycle;  cout<<"\nCPU Utilization(Percentage)= " << cpu\_utilization\*100;  return 0;  } |

**Output:**



**Shortest Job First:**

Shortest job first is a scheduling algorithm in which the process with the smallest execution time is selected for execution next. Shortest job first can be either preemptive or non-preemptive. Owing to its simple nature, shortest job first is considered optimal. It also reduces the average waiting time for other processes awaiting execution.

**Advantages:**

* Shortest jobs are favored.
* It is provably optimal; in that it gives the minimum average waiting time for a given set of processes.

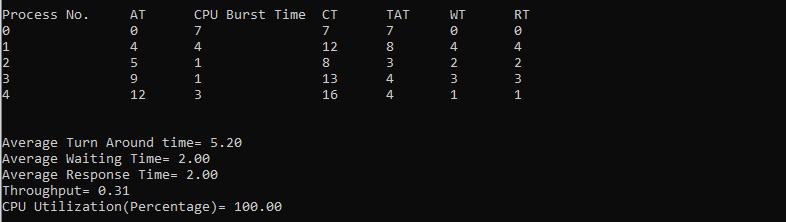
**Disadvantages:**

* SJF may cause starvation, if shorter processes keep coming. This problem is solved by aging.
* It cannot be implemented at the level of short-term CPU scheduling.

**Code:**

|  |
| --- |
| int sjf(){  int n;  bool is\_completed[100]={false},is\_first\_process=true;  int current\_time = 0;  int completed = 0;;  cout<<"Enter total number of processes: ";  cin>>n;  int sum\_tat=0,sum\_wt=0,sum\_rt=0,total\_idle\_time=0,prev=0,length\_cycle;  float cpu\_utilization;  int max\_completion\_time,min\_arrival\_time;    cout << fixed << setprecision(2);  for(int i=0;i<n;i++)  {  cout<<"\nEnter Process " <<i<< " Arrival Time: ";  cin >> ps4[i].at;  ps4[i].pid=i;  }    for(int i=0;i<n;i++)  {  cout<<"\nEnter Process " <<i<< " Burst Time: ";  cin >> ps4[i].bt;  }      while(completed!=n)  {  //find process with min. burst time in ready queue at current time  int min\_index = -1;  int minimum = INT\_MAX;  for(int i = 0; i < n; i++) {  if(ps4[i].at <= current\_time && is\_completed[i] == false) {  if(ps4[i].bt < minimum) {  minimum = ps4[i].bt;  min\_index = i;  }  if(ps4[i].bt== minimum) {  if(ps4[i].at < ps4[min\_index].at) {  minimum= ps4[i].bt;  min\_index = i;  }  }  }  }  if(min\_index==-1)  {  current\_time++;  }  else  {  ps4[min\_index].start\_time = current\_time;  ps4[min\_index].ct = ps4[min\_index].start\_time + ps4[min\_index].bt;  ps4[min\_index].tat = ps4[min\_index].ct - ps4[min\_index].at;  ps4[min\_index].wt = ps4[min\_index].tat - ps4[min\_index].bt;  ps4[min\_index].rt = ps4[min\_index].wt;  // ps4[min\_index].rt = ps4[min\_index].start\_time - ps4[min\_index].at;    sum\_tat +=ps4[min\_index].tat;  sum\_wt += ps4[min\_index].wt;  sum\_rt += ps4[min\_index].rt;  total\_idle\_time += (is\_first\_process==true) ? 0 : (ps4[min\_index].start\_time - prev);    completed++;  is\_completed[min\_index]=true;  current\_time = ps4[min\_index].ct;  prev= current\_time;  is\_first\_process = false;  }  }    //Calculate Length of Process completion cycle  max\_completion\_time = INT\_MIN;  min\_arrival\_time = INT\_MAX;  for(int i=0;i<n;i++)  {  max\_completion\_time = max(max\_completion\_time,ps4[i].ct);  min\_arrival\_time = min(min\_arrival\_time,ps4[i].at);  }  length\_cycle = max\_completion\_time - min\_arrival\_time;  //Output  cout<<"\nProcess No.\tAT\tCPU Burst Time\tCT\tTAT\tWT\tRT\n";  for(int i=0;i<n;i++)  cout<<i<<"\t\t"<<ps4[i].at<<"\t"<<ps4[i].bt<<"\t\t"<<ps4[i].ct<<"\t"<<ps4[i].tat<<"\t"<<ps4[i].wt<<"\t"<<ps4[i].rt<<endl;  cout<<endl;    cpu\_utilization = (float)(length\_cycle - total\_idle\_time)/ length\_cycle;  cout<<"\nAverage Turn Around time= "<< (float)sum\_tat/n;  cout<<"\nAverage Waiting Time= "<<(float)sum\_wt/n;  cout<<"\nAverage Response Time= "<<(float)sum\_rt/n;  cout<<"\nThroughput= "<<n/(float)length\_cycle;  cout<<"\nCPU Utilization(Percentage)= " << cpu\_utilization\*100;  return 0;  } |

**Output:**



**Round Robin:**

The name of this algorithm comes from the round-robin principle, where each person gets an equal share of something in turns. It is the oldest, simplest scheduling algorithm, which is mostly used for multitasking.

In Round-robin scheduling, each ready task runs turn by turn only in a cyclic queue for a limited time slice. A time quantum is provided to the CPU in this algorithm. This algorithm also offers starvation free execution of processes.

**Advantages:**

* Every process gets an equal share of the CPU.
* RR is cyclic in nature, so there is no starvation.

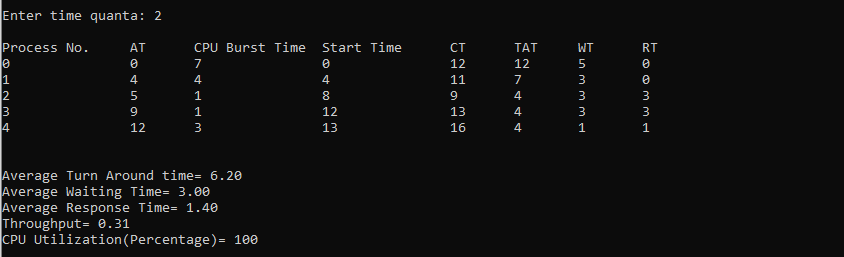
**Disadvantages:**

* Setting the quantum too short, increases the overhead and lowers the CPU efficiency, but setting it too long may cause poor response to short processes.
* Average waiting time under the RR policy is often long.

**Code:**

|  |
| --- |
| int rr(){  int n,index;  int cpu\_utilization;  queue<int> q;  bool visited[100]={false},is\_first\_process=true;  int current\_time = 0,max\_completion\_time;  int completed = 0,tq, total\_idle\_time=0,length\_cycle;  cout<<"Enter total number of processes: ";  cin>>n;  float sum\_tat=0,sum\_wt=0,sum\_rt=0;  cout << fixed << setprecision(2);  for(int i=0;i<n;i++)  {  cout<<"\nEnter Process " <<i<< " Arrival Time: ";  cin >> ps3[i].at;  ps3[i].pid=i;  }    for(int i=0;i<n;i++)  {  cout<<"\nEnter Process " <<i<< " Burst Time: ";  cin >> ps3[i].bt;  ps3[i].bt\_remaining= ps3[i].bt;  }    cout<<"\nEnter time quanta: ";  cin>>tq;    //sort structure on the basis of Arrival time in increasing order  sort(ps3,ps3+n,comparatorATrr);    q.push(0);  visited[0] = true;    while(completed != n)  {  index = q.front();  q.pop();    if(ps3[index].bt\_remaining == ps3[index].bt)  {  ps3[index].start\_time = max(current\_time,ps3[index].at);  total\_idle\_time += (is\_first\_process == true) ? 0 : ps3[index].start\_time - current\_time;  current\_time = ps3[index].start\_time;  is\_first\_process = false;    }  if(ps3[index].bt\_remaining-tq > 0)  {  ps3[index].bt\_remaining -= tq;  current\_time += tq;  }  else  {  current\_time += ps3[index].bt\_remaining;  ps3[index].bt\_remaining = 0;  completed++;  ps3[index].ct = current\_time;  ps3[index].tat = ps3[index].ct - ps3[index].at;  ps3[index].wt = ps3[index].tat - ps3[index].bt;  ps3[index].rt = ps3[index].start\_time - ps3[index].at;  sum\_tat += ps3[index].tat;  sum\_wt += ps3[index].wt;  sum\_rt += ps3[index].rt;  }  //check which new Processes needs to be pushed to Ready Queue from Input list  for(int i = 1; i < n; i++)  {  if(ps3[i].bt\_remaining > 0 && ps3[i].at <= current\_time && visited[i] == false)  {  q.push(i);  visited[i] = true;  }  }  //check if Process on CPU needs to be pushed to Ready Queue  if( ps3[index].bt\_remaining> 0)  q.push(index);    //if queue is empty, just add one process from list, whose remaining burst time > 0  if(q.empty())  {  for(int i = 1; i < n; i++)  {  if(ps3[i].bt\_remaining > 0)  {  q.push(i);  visited[i] = true;  break;  }  }  }  } //end of while    //Calculate Length of Process completion cycle  max\_completion\_time = INT\_MIN;    for(int i=0;i<n;i++)  max\_completion\_time = max(max\_completion\_time,ps3[i].ct);    length\_cycle = max\_completion\_time - ps3[0].at; //ps3[0].start\_time;    cpu\_utilization = (float)(length\_cycle - total\_idle\_time)/ length\_cycle;  //sort so that process ID in output comes in Original order (just for interactivity- Not needed otherwise)  sort(ps3, ps3+n , comparatorPIDrr);  //Output  cout<<"\nProcess No.\tAT\tCPU Burst Time\tStart Time\tCT\tTAT\tWT\tRT\n";  for(int i=0;i<n;i++)  cout<<i<<"\t\t"<<ps3[i].at<<"\t"<<ps3[i].bt<<"\t\t"<<ps3[i].start\_time<<"\t\t"<<ps3[i].ct<<"\t"<<ps3[i].tat<<"\t"<<ps3[i].wt<<"\t"<<ps3[i].rt<<endl;  cout<<endl;  cout<<"\nAverage Turn Around time= "<< (float)sum\_tat/n;  cout<<"\nAverage Waiting Time= "<<(float)sum\_wt/n;  cout<<"\nAverage Response Time= "<<(float)sum\_rt/n;  cout<<"\nThroughput= "<<n/(float)length\_cycle;  cout<<"\nCPU Utilization(Percentage)= " << cpu\_utilization\*100;  return 0;  } |

**Output:**



**Shortest Remaining Time First:**

Shortest Remaining Time First (SRTF) is the preemptive version of Shortest Job First (SJF) algorithm, where the processor is allocated to the job closest to completion. The process with the lowest burst time remaining, among the list of available processes in the ready queue, is allocated the CPU.

**Advantages:**

SRTF algorithm makes the processing of the jobs faster than SJF algorithm, given its overhead charges are not counted.

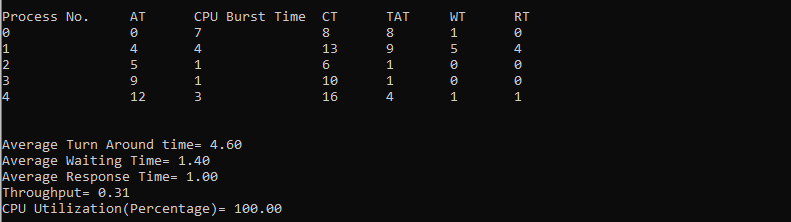
**Disadvantages:**

The context switch is done a lot more times in SRTF than in SJF, and consumes CPU’s valuable time for processing.

**Code:**

|  |
| --- |
| int srtf(){  int n;  float bt\_remaining[100];  bool is\_completed[100]={false},is\_first\_process=true;  int current\_time = 0;  int completed = 0;;  float sum\_tat=0,sum\_wt=0,sum\_rt=0,total\_idle\_time=0,length\_cycle,prev=0;  float cpu\_utilization;  int max\_completion\_time,min\_arrival\_time;    cout << fixed << setprecision(2);  cout<<"Enter total number of processes: ";  cin>>n;  for(int i=0;i<n;i++)  {  cout<<"\nEnter Process" <<i<< "Arrival Time: ";  cin >> ps[i].at;  ps[i].pid=i;  }    for(int i=0;i<n;i++)  {  cout<<"\nEnter Process" <<i<< "Burst Time: ";  cin >> ps[i].bt;  bt\_remaining[i]= ps[i].bt;  }  while(completed!=n)  {  //find process with min. burst time in ready queue at current time  int min\_index = -1;  int minimum = INT\_MAX;  for(int i = 0; i < n; i++) {  if(ps[i].at <= current\_time && is\_completed[i] == false) {  if(bt\_remaining[i] < minimum) {  minimum = bt\_remaining[i];;  min\_index = i;  }  if(bt\_remaining[i]== minimum) {  if(ps[i].at < ps[min\_index].at) {  minimum= bt\_remaining[i];;  min\_index = i;  }  }  }  }  if(min\_index==-1)  {  current\_time++;  }  else  {  if(bt\_remaining[min\_index] == ps[min\_index].bt)  {  ps[min\_index].start\_time = current\_time;  total\_idle\_time += (is\_first\_process==true) ? 0 : (ps[min\_index].start\_time - prev);  is\_first\_process=false;  }  bt\_remaining[min\_index] -= 1;  current\_time++;  prev=current\_time;  if(bt\_remaining[min\_index] == 0)  {  ps[min\_index].ct = current\_time;  ps[min\_index].tat = ps[min\_index].ct - ps[min\_index].at;  ps[min\_index].wt= ps[min\_index].tat - ps[min\_index].bt;  ps[min\_index].rt = ps[min\_index].start\_time - ps[min\_index].at;    sum\_tat +=ps[min\_index].tat;  sum\_wt += ps[min\_index].wt;  sum\_rt += ps[min\_index].rt;  completed++;  is\_completed[min\_index]=true;  }  }  }  //Calculate Length of Process completion cycle  max\_completion\_time = INT\_MIN;  min\_arrival\_time = INT\_MAX;  for(int i=0;i<n;i++)  {  max\_completion\_time = max(max\_completion\_time,ps[i].ct);  min\_arrival\_time = min(min\_arrival\_time,ps[i].at);  }  length\_cycle = max\_completion\_time - min\_arrival\_time;  //Output  cout<<"\nProcess No.\tAT\tCPU Burst Time\tCT\tTAT\tWT\tRT\n";  for(int i=0;i<n;i++)  cout<<i<<"\t\t"<<ps[i].at<<"\t"<<ps[i].bt<<"\t\t"<<ps[i].ct<<"\t"<<ps[i].tat<<"\t"<<ps[i].wt<<"\t"<<ps[i].rt<<endl;  cout<<endl;    cpu\_utilization = (float)(length\_cycle - total\_idle\_time)/ length\_cycle;  cout<<"\nAverage Turn Around time= "<< (float)sum\_tat/n;  cout<<"\nAverage Waiting Time= "<<(float)sum\_wt/n;  cout<<"\nAverage Response Time= "<<(float)sum\_rt/n;  cout<<"\nThroughput= "<<n/(float)length\_cycle;  cout<<"\nCPU Utilization(Percentage)= " << cpu\_utilization\*100;  return 0;  } |

**Output:**



## 