**LAB-01**

1. **FCFS Scheduling Algorithms**

**First come first serve** (FCFS) scheduling algorithm simply schedules the jobs according to their arrival time. The job which comes first in the ready queue will get the CPU first. The lesser the arrival time of the job, the sooner will the job get the CPU. FCFS scheduling may cause the problem of starvation if the burst time of the first process is the longest among all the jobs.

**Advantages of FCFS**

* Simple
* Easy
* First come, First serve

# Disadvantages of FCFS

1. The scheduling method is non preemptive, the process will run to the completion.
2. Due to the non-preemptive nature of the algorithm, the problem of starvation may occur.
3. Although it is easy to implement, but it is poor in performance since the average waiting time is higher as compare to other scheduling algorithms.

**Codeof FCFS**

*// C++ program for implementation of FCFS scheduling algorithm*

#include<iostream>

using namespace std;

void findWaitingTime(int processes[], int n,

int bt[], int wt[])

{

wt[0] = 0;

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

wt[i] = bt[i-1] + wt[i-1] ;

}

void findTurnAroundTime( int processes[], int n,

int bt[], int wt[], int tat[])

{

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

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

}

void findavgTime( int processes[], int n, int bt[])

{

int wt[n], tat[n], total\_wt = 0, total\_tat = 0;

findWaitingTime(processes, n, bt, wt);

findTurnAroundTime(processes, n, bt, wt, tat);

cout << "Processes "<< " Burst time "

<< " Waiting time " << " Turn around time\n";

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;

}

int main()

{

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

int n = sizeof processes / sizeof processes[0];

int burst\_time[] = {10, 5, 8};

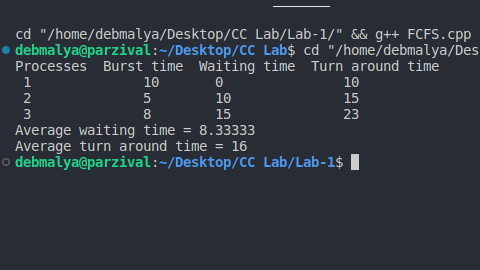
findavgTime(processes, n, burst\_time);

cout<<"\t"<<endl;

return 0;

}

**Output**

****

1. **Round Robin Scheduling Algorithm**

Round Robin scheduling algorithm is one of the most popular scheduling algorithm which can actually be implemented in most of the operating systems. This is the **preemptive version** of first come first serve scheduling. The Algorithm focuses on Time Sharing. In this algorithm, every process gets executed in a **cyclic way**. A certain time slice is defined in the system which is called time **quantum**. Each process present in the ready queue is assigned the CPU for that time quantum, if the execution of the process is completed during that time then the process will **terminate** else the process will go back to the **ready queue** and waits for the next turn to complete the execution.

**Advantages of Round Robin Scheduling Algorithm**

1. It can be actually implementable in the system because it is not depending on the burst time.
2. It doesn't suffer from the problem of starvation or convoy effect.
3. All the jobs get a fare allocation of CPU.

**Disadvantages of Round Robin Scheduling Algorithm**

1. The higher the time quantum, the higher the response time in the system.
2. The lower the time quantum, the higher the context switching overhead in the system.
3. Deciding a perfect time quantum is really a very difficult task in the system.

**Code of Round Robin**

*// C++ program for implementation of RR scheduling algorithm*

#include<iostream>

using namespace std;

void findWaitingTime(int processes[], int n,

int bt[], int wt[], int quantum)

{

int rem\_bt[n];

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

rem\_bt[i] = bt[i];

int t = 0;

while (1)

{

bool done = true;

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

{

if (rem\_bt[i] > 0)

{

done = false;

if (rem\_bt[i] > quantum)

{

t += quantum;

rem\_bt[i] -= quantum;

}

else

{

t = t + rem\_bt[i];

wt[i] = t - bt[i];

rem\_bt[i] = 0;

}

}

}

if (done == true)

break;

}

}

void findTurnAroundTime(int processes[], int n,

int bt[], int wt[], int tat[])

{

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

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

}

void findavgTime(int processes[], int n, int bt[],

int quantum)

{

int wt[n], tat[n], total\_wt = 0, total\_tat = 0;

findWaitingTime(processes, n, bt, wt, quantum);

findTurnAroundTime(processes, n, bt, wt, tat);

cout << " PN\t "<< " \tBT "

<< " WT " <<" \t\t TAT\n";

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;

}

int main()

{

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

int n = sizeof processes / sizeof processes[0];

int burst\_time[] = {10, 5, 8};

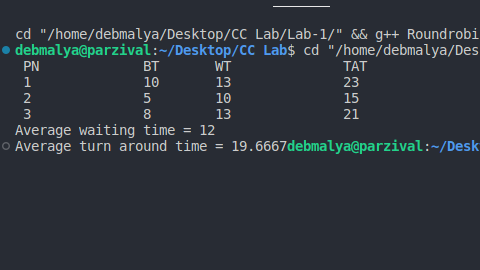
int quantum = 2;

findavgTime(processes, n, burst\_time, quantum);

return 0;

}

**Output**



## Non-Preemptive SJF

In non-preemptive scheduling, once the CPU cycle is allocated to process, the process holds it till it reaches a waiting state or terminated.

**Advantages of Non -Preemptive SJF:**

1. Short processes are handled very quickly.
2. The system also requires very little overhead since it only makes a decision when a process completes or a new process is added.
3. When a new process is added the algorithm only needs to compare the currently executing process with the new process, ignoring all other processes currently waiting to execute.

**Disadvantages of Non -Preemptive SJF:**

1. Like shortest job first, it has the potential for process starvation.
2. Long processes may be held off indefinitely if short processes are continually added.

**Code of Non-Preemptive SJF**

*// C++ program for implementation of SJF scheduling algorithm*

#include<iostream>

using namespace std;

int main()

{

int n,temp,tt=0,min,d,i,j;

float atat=0,awt=0,stat=0,swt=0;

cout<<"enter no of process"<<endl;

cin>>n;

int a[n],b[n],e[n],tat[n],wt[n];

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

{

cout<<"enter arrival time ";

cin>>a[i];

}

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

{

cout<<"enter burst time ";

cin>>b[i];

}

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

{

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

{

if(b[i]>b[j])

{

temp=a[i];

a[i]=a[j];

a[j]=temp;

temp=b[i];

b[i]=b[j];

b[j]=temp;

}

}

}

min=a[0];

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

{

if(min>a[i])

{

min=a[i];

d=i;

}

}

tt=min;

e[d]=tt+b[d];

tt=e[d];

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

{

if(a[i]!=min)

{

e[i]=b[i]+tt;

tt=e[i];

}

}

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

{

tat[i]=e[i]-a[i];

stat=stat+tat[i];

wt[i]=tat[i]-b[i];

swt=swt+wt[i];

}

atat=stat/n;

awt=swt/n;

cout<<"Process Arrival-time(s) Burst-time(s) Waiting-time(s) Turnaround-time(s)\n";

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

{

cout<<"P"<<i+1<<" "<<a[i]<<" "<<b[i]<<" "<<wt[i]<<" "<<tat[i]<<endl;

}

cout<<"awt="<<awt<<" atat="<<atat;

}

**Output**