OS lab Assignment IV

Varun Khadayate

NMIMS

# Definition

**Shortest Job First (SJF)** is an algorithm in which the process having the smallest execution time is chosen for the next execution. This scheduling method can be preemptive or non-preemptive. It significantly reduces the average waiting time for other processes awaiting execution. The full form of SJF is Shortest Job First.

# Characteristics of SJF method

* It is associated with each job as a unit of time to complete.
* This algorithm method is helpful for batch-type processing, where waiting for jobs to complete is not critical.
* It can improve process throughput by making sure that shorter jobs are executed first, hence possibly have a short turnaround time.
* It improves job output by offering shorter jobs, which should be executed first, which mostly have a shorter turnaround time.

# Non-preemptive SJF

## Implementation

1. Sort all the process according to the arrival time.
2. Then select that process which has minimum arrival time and minimum Burst time.
3. After completion of process make a pool of process which after till the completion of previous process and select that process among the pool which is having minimum Burst time.
4. After that compute the following
   1. Completion Time: Time at which process completes its execution.
   2. Turn Around Time: Time Difference between completion time and arrival time. Turn Around Time = Completion Time – Arrival Time
   3. Waiting Time(W.T): Time Difference between turnaround time and burst time.  
      Waiting Time = Turn Around Time – Burst Time

## Code

#include<iostream>

#include<string.h>

using namespace std;

int main()

{

int i,n,p[10]={1,2,3,4,5,6,7,8,9,10},min,k=1,btime=0,bt[10],temp,j,at[10],wt[10],tt[10],ta=0,sum=0;

float wavg=0,tavg=0,tsum=0,wsum=0;

printf("Enter the No. of processes :");

scanf("%d",&n);

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

{

printf("\nEnter the arrival time of process[%d]::",i+1);

scanf(" %d",&at[i]);

printf("\nEnter the burst time of process[%d]::",i+1);

scanf(" %d",&bt[i]);

printf("\n");

}

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

{

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

{

if(at[i]<at[j])

{

temp=p[j];

p[j]=p[i];

p[i]=temp;

temp=at[j];

at[j]=at[i];

at[i]=temp;

temp=bt[j];

bt[j]=bt[i];

bt[i]=temp;

}

}

}

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

{

btime=btime+bt[j];

min=bt[k];

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

{

if (btime>=at[i] && bt[i]<min)

{

temp=p[k];

p[k]=p[i];

p[i]=temp;

temp=at[k];

at[k]=at[i];

at[i]=temp;

temp=bt[k];

bt[k]=bt[i];

bt[i]=temp;

}

}

k++;

}

wt[0]=0;

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

{

sum=sum+bt[i-1];

wt[i]=sum-at[i];

wsum=wsum+wt[i];

}

wavg=(wsum/n);

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

{

ta=ta+bt[i];

tt[i]=ta-at[i];

tsum=tsum+tt[i];

}

tavg=(tsum/n);

printf("\n RESULT:-");

printf("\nP#\t AT\t BT\t WT\t TAT" );

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

{

printf("\np%d\t %d\t %d\t %d\t %d",p[i],at[i],bt[i],wt[i],tt[i]);

}

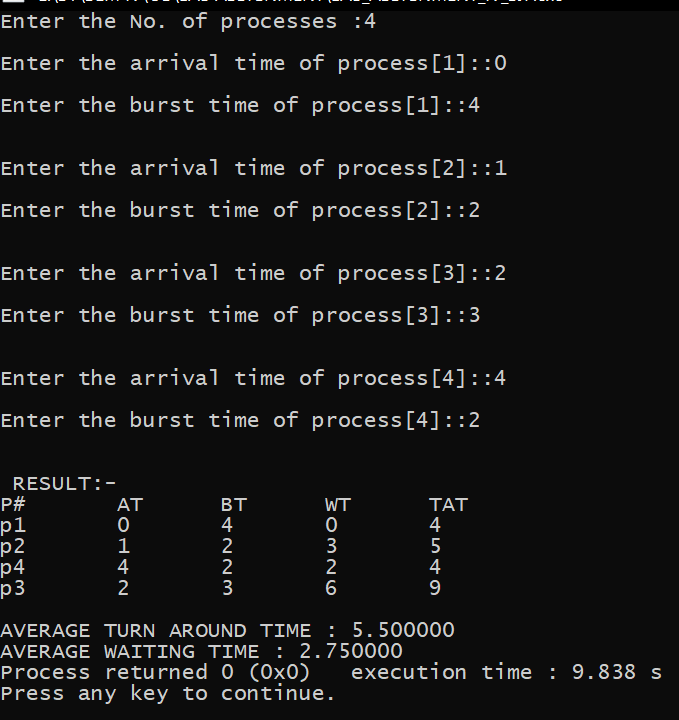
printf("\n\nAVERAGE TURN AROUND TIME : %f",tavg);

printf("\nAVERAGE WAITING TIME : %f",wavg);

return 0;

}

## OUTPUT



## Via Formula and Steps check

Text, letter

Description automatically generated

## Advantages of SJF

* This algorithm is very simple to implement.
* The aging technique is implemented to reduce the starvation of lower priority processes.

## Disadvantages of SJF

* Starvation or indefinite blockage of the lower priority processes.
* Since this is a non-preemptive implementation, the waiting time is comparatively higher.
* The average turnaround time is higher as compared to the preemptive priority scheduling algorithm.
* If a system failure occurs, all the unfinished lower priority jobs get vanished from the system.

# Preemptive SJF

## Implementation

1. Traverse until all process gets completely executed.
   1. Find process with minimum remaining time at every single time lap.
   2. Reduce its time by 1.
   3. Check if its remaining time becomes 0.
   4. Increment the counter of process completion.
   5. Completion time of current process = current\_time +1.
   6. Calculate waiting time for each completed process.

wt[i]= Completion time - arrival\_time-burst\_time

* 1. Increment time lap by one.

1. Find turnaround time (waiting\_time+burst\_time).

## Code

#include<iostream>

#include<string.h>

using namespace std;

struct process

{

char process\_name;

int arrival\_time, burst\_time, ct, waiting\_time, turnaround\_time, priority;

int status;

}process\_queue[10];

int limit;

void Arrival\_Time\_Sorting()

{

struct process temp;

int i, j;

for(i = 0; i < limit - 1; i++)

{

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

{

if(process\_queue[i].arrival\_time > process\_queue[j].arrival\_time)

{

temp = process\_queue[i];

process\_queue[i] = process\_queue[j];

process\_queue[j] = temp;

}

}

}

}

int main()

{

int i, time = 0, burst\_time = 0, largest;

char c;

float wait\_time = 0, turnaround\_time = 0, average\_waiting\_time, average\_turnaround\_time;

printf("\nEnter Total Number of Processes:\t");

scanf("%d", &limit);

for(i = 0, c = 'A'; i < limit; i++, c++)

{

process\_queue[i].process\_name = i;

printf("\nEnter Details For Process[%d]:\n", process\_queue[i].process\_name);

printf("Enter Arrival Time:\t");

scanf("%d", &process\_queue[i].arrival\_time );

printf("Enter Burst Time:\t");

scanf("%d", &process\_queue[i].burst\_time);

printf("Enter Priority:\t");

scanf("%d", &process\_queue[i].priority);

process\_queue[i].status = 0;

burst\_time = burst\_time + process\_queue[i].burst\_time;

}

Arrival\_Time\_Sorting();

process\_queue[9].priority = -9999;

printf("\nProcess Name\tArrival Time\tBurst Time\tPriority\tWaiting Time");

for(time = process\_queue[0].arrival\_time; time < burst\_time;)

{

largest = 9;

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

{

if(process\_queue[i].arrival\_time <= time && process\_queue[i].status != 1 && process\_queue[i].priority > process\_queue[largest].priority)

{

largest = i;

}

}

time = time + process\_queue[largest].burst\_time;

process\_queue[largest].ct = time;

process\_queue[largest].waiting\_time = process\_queue[largest].ct - process\_queue[largest].arrival\_time - process\_queue[largest].burst\_time;

process\_queue[largest].turnaround\_time = process\_queue[largest].ct - process\_queue[largest].arrival\_time;

process\_queue[largest].status = 1;

wait\_time = wait\_time + process\_queue[largest].waiting\_time;

turnaround\_time = turnaround\_time + process\_queue[largest].turnaround\_time;

printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d", process\_queue[largest].process\_name, process\_queue[largest].arrival\_time, process\_queue[largest].burst\_time, process\_queue[largest].priority, process\_queue[largest].waiting\_time);

}

average\_waiting\_time = wait\_time / limit;

average\_turnaround\_time = turnaround\_time / limit;

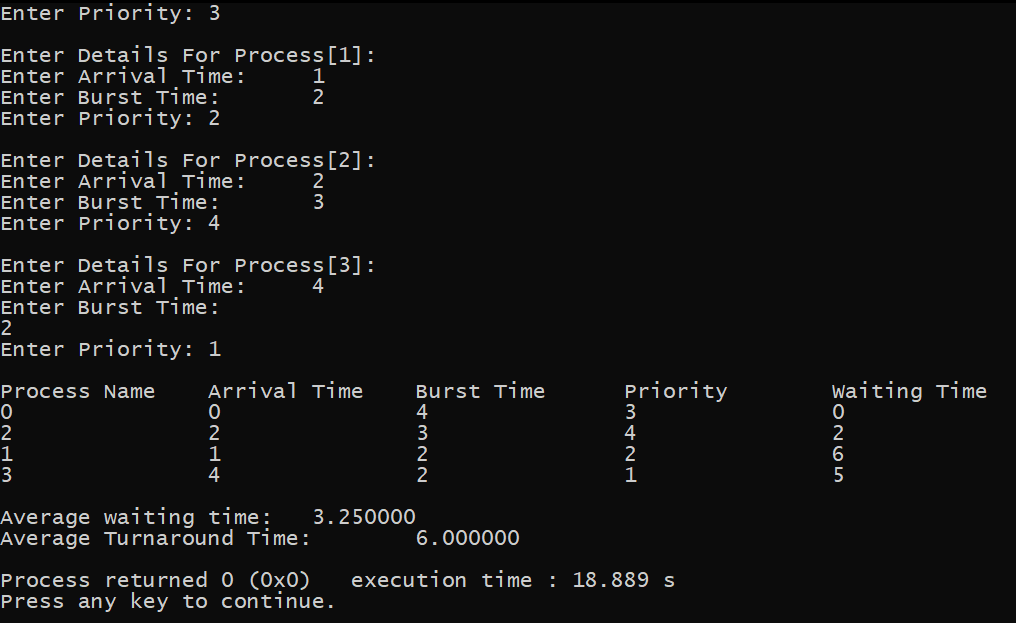
printf("\n\nAverage waiting time:\t%f\n", average\_waiting\_time);

printf("Average Turnaround Time:\t%f\n", average\_turnaround\_time);

return 0;

}

## OUTPUT



## Advantages of SJF

* Preemptive priority scheduling is much more efficient as compared to the non-preemptive version.
* This priority job scheduling algorithm is quite simple to implement.
* The aging technique is implemented to reduce the starvation of lower priority processes.
* The average turnaround time and waiting time is efficient.

## Disadvantages of SJF

* Indefinite blockage of the lower priority jobs.
* For a system failure occurs, the unfinished lower priority jobs are removed from the system and cannot be recovered.