**Student Name: Aditya Dwivedi**

**Student ID: 11802083**

**Email Address: adidwivedi17@gmail.com**

**GitHub Link:** <https://github.com/adiis1/Operating-System-project>

**Ques. 10. Design a scheduler with multilevel queue having two queues which will schedule the processes on the basis of pre-emptive shortest remaining processing time first algorithm (SROT) followed by a scheduling in which each process will get 2 units of time to execute. Also note that queue 1 has higher priority than queue 2. Consider the following set of processes (for reference)with their arrival times and the CPU burst times in milliseconds.**

**------------------------------------ Process Arrival-Time Burst-Time ------------------------------------**

**P1 0 5**

**P2 1 3**

**P3 2 3**

**P4 4 1**

**-----------------------------------**

**Calculate the average turnaround time and average waiting time for each process. The input for number of processes and their arrival time, burst time should be given by the user.**

**Description:**

**Shortest Remaining Time First** is a scheduling algorithm used in Operating Systems, which can also be called as the preemptive version of the SJF scheduling algorithm. The process which has the least processing time remaining is executed first. As it is a preemptive type of schedule, it is claimed to be better than [SJF scheduling Algorithm](https://www.includehelp.com/operating-systems/sjf-shortest-job-first-scheduling-algorithm.aspx).

**Advantages:**  
SRTF algorithm makes the processing of the jobs faster than SJN algorithm, given it’s overhead charges are not counted.

**Disadvantages:**  
The context switch is done a lot more times in SRTF than in SJN, and consumes CPU’s valuable time for processing. This adds up to it’s processing time and diminishes it’s advantage of fast processing.

**Multilevel Queue Scheduling :**

* When processes can be readily categorized, then multiple separate queues can be established, each implementing whatever scheduling algorithm is most appropriate for that type of job, and/or with different parametric adjustments.
* Scheduling must also be done between queues, that is scheduling one queue to get time relative to other queues. Two common options are strict priority ( no job in a lower priority queue runs until all higher priority queues are empty ) and round-robin ( each queue gets a time slice in turn, possibly of different sizes. )
* Note that under this algorithm jobs cannot switch from queue to queue - Once they are assigned a queue, that is their queue until they finish.

**Algorithm :**

Step 1: Take input details such as No.Of Processes , Arrival Time and Burst Time

Step 2: Calculate Priority on the behalf of Burst Time if Burst Time is more value of Prioity is greater else smaller

Step 3: If the Priority is More than It is Considered as System Processes and Other wise it is considered as Student Process.

Step 4: If Priority is Same than Go to FCFS queue , else then go to RR with some TIME quantum allocated to each Process

Step 5: Calculate WT and TAT on the Behalf of Arrival Of Processes

**Source Code:**

**#include<stdio.h>**

**int main()**

**{**

**int i, limit, total = 0, x, counter = 0, time\_quantum,j;**

**int wait\_time = 0, turnaround\_time = 0,pos,z,p[10],prio[10], a\_time[10], b\_time[10], temp[10],b;**

**float average\_wait\_time, average\_turnaround\_time;**

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

**scanf("%d", &limit);**

**x = limit;**

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

**{**

**p[i]=i+1;**

**prio[i]=0;**

**printf("\nEnter total Details of Process[%d]\n", i + 1);**

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

**scanf("%d", &a\_time[i]);**

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

**scanf("%d", &b\_time[i]);**

**temp[i] = b\_time[i];**

**}**

**printf("\nEnter the Time Quantum:");**

**scanf("%d", &time\_quantum);**

**printf("\nProcess ID\t\tBurst Time\t Turnaround Time\t Waiting Time\t Priority\n");**

**for(total = 0, i = 0; x != 0;)**

**{**

**for(z=0;z<limit;z++)**

**{**

**int temp1;**

**pos=z;**

**for(j=z+1;j<limit;j++)**

**{**

**if(prio[j]<prio[pos])**

**pos=j;**

**}**

**temp1=prio[z];**

**prio[z]=prio[pos];**

**prio[pos]=temp1;**

**temp1=b\_time[z];**

**b\_time[z]=b\_time[pos];**

**b\_time[pos]=temp1;**

**temp1=a\_time[z];**

**a\_time[z]=a\_time[pos];**

**a\_time[pos]=temp1;**

**temp1=p[z];**

**p[z]=p[pos];**

**p[pos]=temp1;**

**temp1=temp[z];**

**temp[z]=temp[pos];**

**temp[pos]=temp1;**

**}**

**{**

**}**

**if(temp[i] <= time\_quantum && temp[i] > 0)**

**{**

**total = total + temp[i];**

**temp[i] = 0;**

**counter = 1;**

**}**

**else if(temp[i] > 0)**

**{**

**temp[i] = temp[i] - time\_quantum;**

**total = total + time\_quantum;**

**}**

**for(b=0;b<limit;b++)**

**{**

**if(b==i)**

**prio[b]+=1;**

**else**

**prio[b]+=2;**

**}**

**if(temp[i] == 0 && counter == 1)**

**{**

**x--;**

**printf("\nProcess[%d]\t\t%d\t\t %d\t\t %d\t\t%d", p[i], b\_time[i], total - a\_time[i], total - a\_time[i] - b\_time[i],prio[i]);**

**wait\_time = wait\_time + total - a\_time[i] - b\_time[i];**

**turnaround\_time = turnaround\_time + total - a\_time[i];**

**sumOfTAT=total - a\_time[i]+sumOfTAT;**

**sumOFWT= total - a\_time[i] - b\_time[i]+sumOFWT;**

**counter = 0;**

**}**

**if(i == limit - 1)**

**{**

**i = 0;**

**}**

**else if(a\_time[i + 1] <= total)**

**{**

**i++;**

**}**

**else**

**{**

**i = 0;**

**}**

**}**

**return 0;**

**float avg\_TAT;**

**float avg\_WT;**

**avg\_WT=sumOFWT/(float)limit;**

**avg\_TAT=sumOfTAT/(float)limit;**

**printf("\nAverage Waiting Time:%1f",avg\_WT);**

**printf("\nAverage Turn Arround Time:%1f",avg\_TAT);**

**}**

**Complexity in code:**

**PART 1:O(n)**

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

{

p[i]=i+1;

prio[i]=0;

printf("\nEnter total Details of Process[%d]\n", i + 1);

printf("Arrival Time:\t");

scanf("%d", &a\_time[i]);

printf("Burst Time:\t");

scanf("%d", &b\_time[i]);

temp[i] = b\_time[i];

}

**PART 2:O(n3)**

**for(total = 0, i = 0; x != 0;)**

**{**

**//O(n2)**

**for(z=0;z<limit;z++)**

**{**

int temp1;

pos=z;

**for(j=z+1;j<limit;j++)**

**{**

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

pos=j;

**}**

temp1=prio[z];

prio[z]=prio[pos];

prio[pos]=temp1;

temp1=b\_time[z];

b\_time[z]=b\_time[pos];

b\_time[pos]=temp1;

temp1=a\_time[z];

a\_time[z]=a\_time[pos];

a\_time[pos]=temp1;

temp1=p[z];

p[z]=p[pos];

p[pos]=temp1;

temp1=temp[z];

temp[z]=temp[pos];

temp[pos]=temp1;

**}**

{

}

if(temp[i] <= time\_quantum && temp[i] > 0)

{

total = total + temp[i];

temp[i] = 0;

counter = 1;

}

else if(temp[i] > 0)

{

temp[i] = temp[i] - time\_quantum;

total = total + time\_quantum;

**}**

**//O(n)**

**for(b=0;b<limit;b++)**

**{**

if(b==i)

prio[b]+=1;

else

prio[b]+=2;

**}**

if(temp[i] == 0 && counter == 1)

{

x--;

printf("\nProcess[%d]\t\t%d\t\t%d\t\t\t%d\t\t%d", p[i], b\_time[i], total - a\_time[i], total - a\_time[i] - b\_time[i],prio[i]);

wait\_time = wait\_time + total - a\_time[i] - b\_time[i];

turnaround\_time = turnaround\_time + total - a\_time[i];

sumOfTAT=total - a\_time[i]+sumOfTAT;

sumOFWT= total - a\_time[i] - b\_time[i]+sumOFWT;

counter = 0;

}

if(i == limit - 1)

{

i = 0;

}

else if(a\_time[i + 1] <= total)

{

i++;

}

else

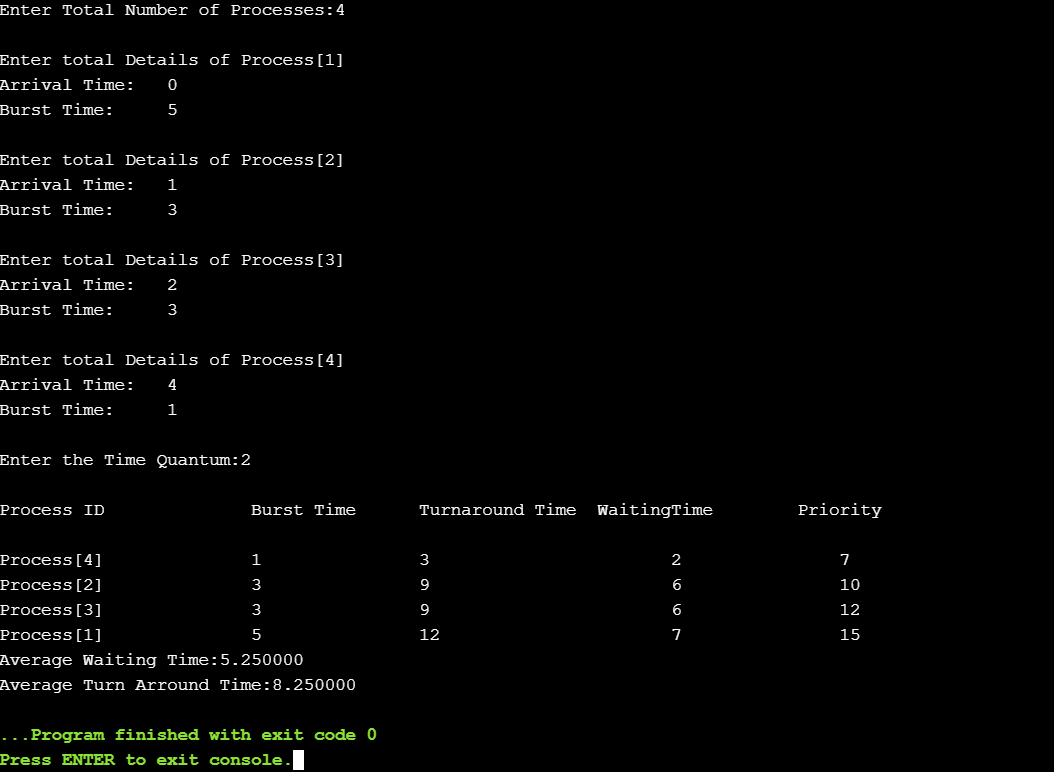
{

i = 0;

}

**}**

**Output For given Code :**



**Test cases :**

1. Pass

A screenshot of a cell phone

Description automatically generated

1. Pass

A screenshot of a cell phone

Description automatically generated

1. Pass

A screenshot of a cell phone

Description automatically generated