**LOVELY PROFESSIONAL UNIVERSITY**

**Faculty of Technology and Sciences**

**School of Computer Science and Engineering**

**Name of the faculty member:** Chhavi Ralhan

**Course Code:** CSE316

**Course Title:** Operating System

**Student Name:** Abhay Chauhan

**Student ID: 11705360**

**Email Address: Aakash.ayush@gmail.com**

**GitHub Link-** https://github.com/Abhay-Chauhan/os

**Code:**

#include<stdio.h>

void pmt();

void rr();

struct rrbn

{

char name;

int at,bt,wt,tt,rt;

int completed;

}p[10];

int n;

int q[10]; //queue

int front=-1,rear=-1;

void q2(int i)

{

if(rear==10)

printf("overflow");

rear++;

q[rear]=i;

if(front==-1)

front=0;

}

int q1()

{

if(front==-1)

printf("underflow");

int temp=q[front];

if(front==rear)

front=rear=-1;

else

front++;

return temp;

}

int isInQueue(int i)

{int k;

for(k=front;k<=rear;k++)

{

if(q[k]==i)

return 1;

}

return 0;

}void sortByArrival()

{

struct rrbn temp;

int i,j;

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

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

{

if(p[i].at>p[j].at)

{

temp=p[i];

p[i]=p[j];

p[j]=temp;

}

}

}

int main()

{

printf("\t\*\*\* Fixed priority pmt Scheduling \*\*\*\n");

pmt();

printf("\n \n");

printf("\t\*\*\* Round Robin Scheduling \*\*\*\n");

rr();

}

void pmt()

{

int num;

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

scanf("%d",&num);

if(num<=0)

printf("ENTER A MINIMUM OF 1 PROCESS \n");

sortByArrival();

int id[num],bt[num],wt[num],tat[num],p[num],i,j,temp;

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

{

printf("Enter process %d id: ",i+1);

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

printf("Enter process %d burst time: ",i+1);

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

bt[i]=bt[i]\*2;

printf("Enter process %d priority: ",i+1);

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

}

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

{

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

{

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

{

temp=p[i];

p[i]=p[j];

p[j]=temp;

temp=bt[i];

bt[i]=bt[j];

bt[j]=temp;

temp=id[i];

id[i]=id[j];

id[j]=temp;

}

}

wt[i]=0;

}

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

{

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

{

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

}

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

}

float avwt=0,avtat=0;

printf("Process\tP\tBT\tWT\tTAT\n");

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

{

printf("%d\t%d\t%d\t%d\t%d\n",id[i],p[i],bt[i],wt[i],tat[i]);

avwt=avwt+wt[i];

avtat=avtat+tat[i];

}

printf("Average Waiting Time: %f\n",avwt/num);

printf("\nAverage Turnaround Time: %f",avtat/num);

}

void rr()

{

int i,j,time=0,sum\_bt=0,tq;

char c;

float avgwt=0;

printf("Enter no of processes: ");

scanf("%d",&n);

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

{

p[i].name=c;

printf("\nEnter arrival time [process] %c: ",p[i].name);

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

printf("Enter burst time [process] %c: ",p[i].name);

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

p[i].bt=p[i].bt\*2;

p[i].rt=p[i].bt;

p[i].completed=0;

sum\_bt+=p[i].bt;

}

printf("\nEnter time quantum: ");

scanf("%d",&tq);

if(tq<=0)

printf("ENTER A MINIMUM OF 1 TIME QUANTUM \n");

sortByArrival();

q2(0);

printf("Process execution order: ");

for(time=p[0].at;time<sum\_bt;)

{

i=q1();

if(p[i].rt<=tq)

{

time+=p[i].rt;

p[i].rt=0;

p[i].completed=1;

printf(" %c ",p[i].name);

p[i].wt=time-p[i].at-p[i].bt;

p[i].tt=time-p[i].at;

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

{

if(p[j].at<=time && p[j].completed!=1&& isInQueue(j)!=1)

{

q2(j);

}

}

}

else

{

time+=tq;

p[i].rt-=tq;

printf(" %c ",p[i].name);

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

{

if(p[j].at<=time && p[j].completed!=1&&i!=j&& isInQueue(j)!=1)

{

q2(j);

}

}

q2(i);

}

}

printf("\nName\tArrival Time\tBurst Time\tWaiting Time\tTurnAround Time\t");

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

{avgwt+=p[i].wt;

printf("\n%c\t\t%d\t\t%d\t\t%d\t\t%d\t\t%f",p[i].name,p[i].at,p[i].bt,p[i].wt,p[i].tt);

}

printf("\nAverage waiting time:%f\n",avgwt/n);

}

**EXPLANATION:**

**Fixed priority preemptive scheduling** policy is widely adopted for periodic task scheduling in embedded real-time operating systems. However, there are few methods or criteria for period assignment of periodic tasks. In this paper, a period assignment method which combines schedulability analysis and quality of service considerations is proposed. Periodic task sets with ranges for the periods of individual tasks are considered. This method is designed to enlarge the tasks' quality of service while maintaining schedulability of the system. High priority tasks are given preference to be assigned with a short period. Talking about **Round-robin,** It is one of the simplest scheduling algorithms for processes in an operating system, which assigns time slice to each process in equal portions and in circular order, handling all processes without priority (also known as cyclic executive). Round-robin scheduling is both simple and easy to implement, and starvation -free. Round-robin scheduling can also be applied to other scheduling problems, such as data packet scheduling in computer networks.

**FIXED PRIORITY PREEMPTIVE SCHEDULING**

The operating system assigns a fixed priority rank to every process, and the scheduler arranges the processes in the ready queue in order of their priority. Lower-priority processes get interrupted by incoming higher-priority processes.

* Overhead is not minimal, nor is it significant.
* FPPS has no particular advantage in terms of throughput over FIFO scheduling.
* If the number of rankings is limited, it can be characterized as a collection of FIFO queues, one for each priority ranking. Processes in lower-priority queues are selected only when all of the higher-priority queues are empty.
* Waiting time and response time depend on the priority of the process. Higher-priority processes have smaller waiting and response times.
* Deadlines can be met by giving processes with deadlines a higher priority.
* Starvation of lower-priority processes is possible with large numbers of high-priority processes queuing for CPU time.

**Advantages**

* The priority of a process can be selected based on memory requirement, time requirement or user preference. For example, a high end game will have better graphics, that means the process which updates the screen in a game will have higher priority so as to achieve better graphics performance.

**Disadvantages:**

* A second scheduling algorithm is required to schedule the processes which have same priority.
* In preemptive priority scheduling, a higher priority process can execute ahead of an already executing lower priority process. If lower priority process keeps waiting for higher priority processes, starvation occurs.

### ROUND-ROBIN SCHEDULING

The scheduler assigns a fixed time unit per process, and cycles through them. If process completes within that time-slice it gets terminated otherwise it is rescheduled after giving a chance to all other processes.

* RR scheduling involves extensive overhead, especially with a small time unit.
* Balanced throughput between FCFS/ FIFO and SJF/SRTF, shorter jobs are completed faster than in FIFO and longer processes are completed faster than in SJF.
* Good average response time, waiting time is dependent on number of processes, and not average process length.
* Because of high waiting times, deadlines are rarely met in a pure RR system.
* Starvation can never occur, since no priority is given. Order of time unit allocation is based upon process arrival time, similar to FIFO.
* If Time-Slice is large it becomes FCFS /FIFO or if it is short then it becomes SJF/SRTF.

**Advantages**:

* Each process is served by the CPU for a fixed time quantum, so all processes are given the same priority.
* Starvation doesn't occur because for each round robin cycle, every process is given a fixed time to execute. No process is left behind.

**Disadvantages:**

* The throughput in RR largely depends on the choice of the length of the time quantum. If time quantum is longer than needed, it tends to exhibit the same behavior as FCFS.
* If time quantum is shorter than needed, the number of times that CPU switches from one process to another process, increases. This leads to decrease in CPU efficiency.

**ALGORITHM:**

**For Fixed Priority Preemptive Scheduling:**

Step 1: Start the process

Step 2: Accept no. of process, process id, burst time of process and priority.

Step 3: Calculate

(a) 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)

(b) 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 4: Calculate

(a) Average waiting time = Total waiting Time / Number of process

(b) Average Turnaround time = Total Turnaround Time / Number of process.

Step 5: Stop the process.

**For Round Robin Scheduling**

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

(a) 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)

(b) 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).

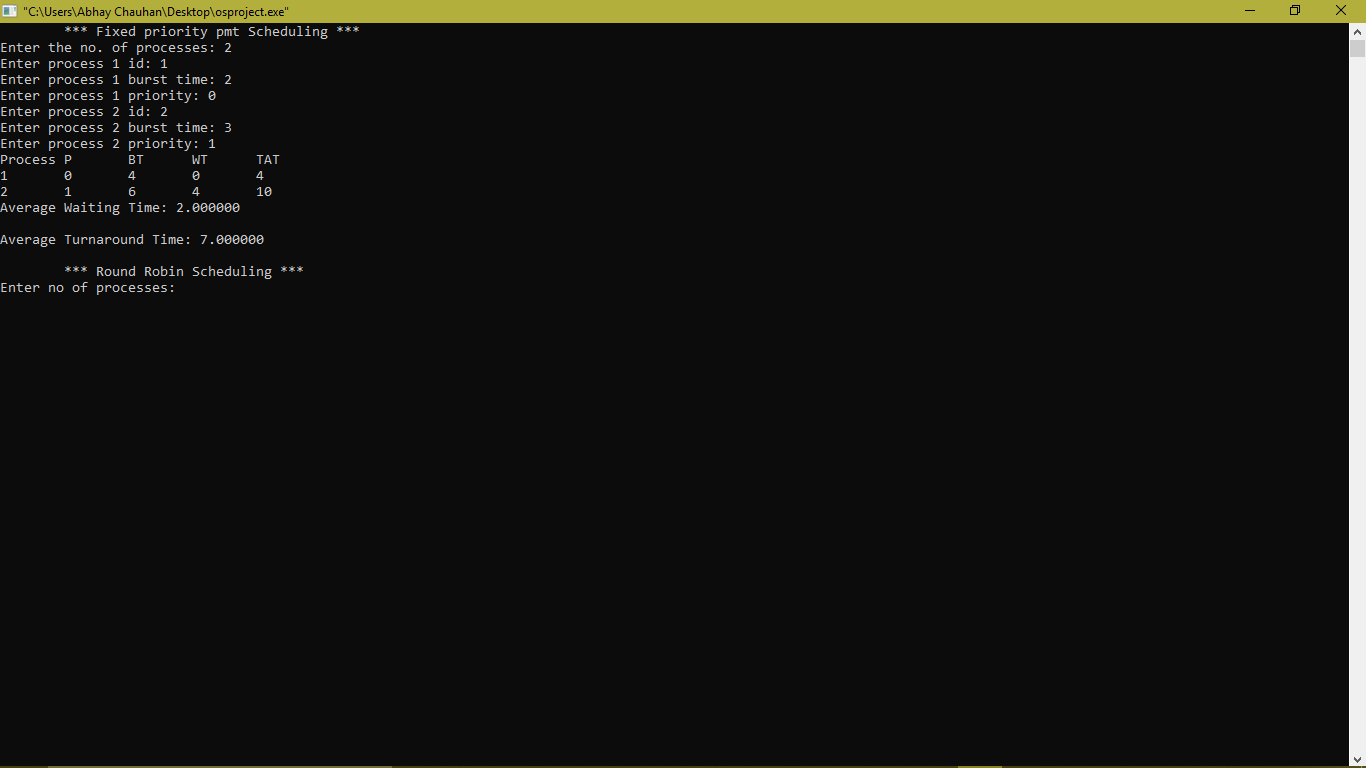
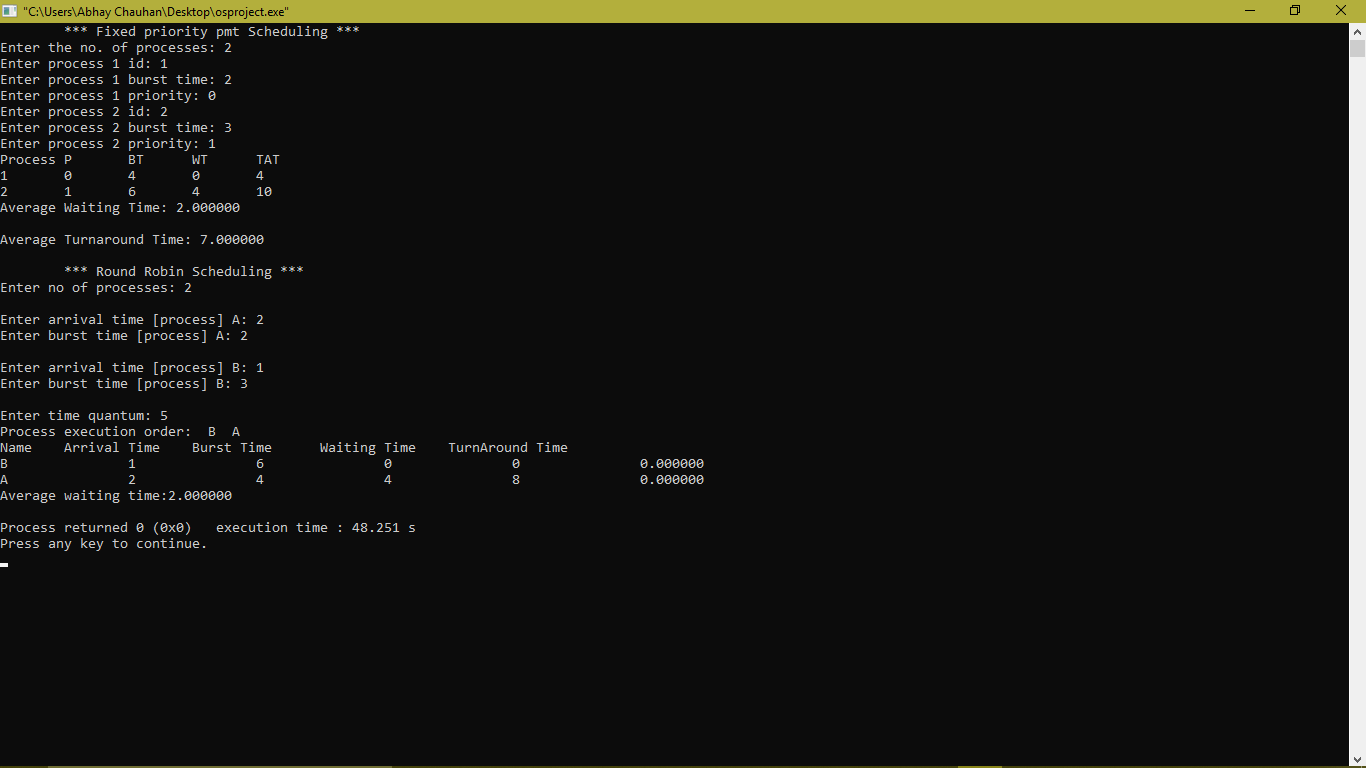
Step 7: Calculate

(a) Average waiting time = Total waiting Time / Number of process

(b) Average Turnaround time = Total Turnaround Time / Number of process

Step 8: Stop the process

**COMPLEXITITY:** The time complexity is Big O(1) .

**CODE SNIPPET:**

**TEST CASES:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Scenario** | **Test Step** | **Expected Result** | **Actual Outcome** | **Result** |
| Verify min no. of processes is 1 | Run the program & enter no of process in negative value or 0. | Display message “ENTER A MINIMUM OF 1 PROCESS” | Message is displayed | Pass |
| Verify if min no. of process in priority scheduling is less than 1 wil the program run. | Run the program & enter no of process in negative value or 0. | Terminate priority preemptive scheduling function | Function Terminated | Pass |
| Verify min. of time quantum is 1. | Run the program & enter time quantum in negative value or 0. | Display message “ENTER A MINIMUM OF 1 TIME QUANTUM” | Message is displayed | Pass |
| Verify if min time quantum in round robin scheduling is less than 1 wil the program run. | Run the program & enter time quantum in negative value or 0. | Terminate round robin scheduling function | Function Terminated | Pass |