**Question:**

Design a scheduling program to implements a Queue with two levels:

Level 1 : Fixed priority preemptive Scheduling

Level 2: Round Robin Scheduling

For a Fixed priority preemptive Scheduling (Queue1), the Priority 0 is highest priority. If one process P1 is scheduled and running, another process P2 with higher priority comes. The New process (high priority) process P2 preempts currently running process P1 and process P1 will go to second level queue. Time for which process will strictly execute must be considered in the multiples of 2. All the processes in second level queue will complete their execution according to round robin scheduling.

Consider: 1. Queue 2 will be processed after Queue 1 becomes empty.

2. Priority of Queue 2 has lower priority than in Queue 1.

**Description:**

Fixed Priority Preemptive Scheduling is a way of scheduling which tells in which order the processes will run in the CPU using the priority assigned to them. In this, we compare the priority of the processes and the process having the highest priority takes the CPU. In case the CPU already has a process then it gets replaced by the process with higher priority and the process with the lower priority is placed in the queue. If two or more process have same arrival time then it schedules according to the First Come First Serve Scheduling. It is mainly used in real time systems. In this type of scheduling, there is a problem of starvation.

Round Robin Scheduling is a way of scheduling the processes by giving them equal time in the CPU in a circular manner, i.e., if the process does not get completed in the giving time then it will be placed in the queue again until the process gets executed completely. The maximum time for which a process can take CPU is known as Time Quantum. It is always Pre-emptive in nature. This type of scheduling is simple, easy to implement and starvation-free in nature.

In this question, we are scheduling the process using Fixed Priority Preemptive Scheduling and Round Robin Scheduling. Firstly, we apply the Fixed Priority Preemptive Scheduling to the processes. If the process gets executed completely, then no problem but if it is not executed completely then we will place them in another queue and apply Round Robin Scheduling to it till all the process get executed completely.

**Algorithm:**

In this problem, I have used complex algorithm for the calculation of the completion time, turn around time, waiting time, average turn around time and average waiting time using two types of scheduling.

for(total\_time=0,counter=0;left\_process!=0;)

{

if(remaining\_time[counter]<=time\_quantum && remaining\_time[counter]>0)

{

total\_time+=remaining\_time[counter];

remaining\_time[counter]=0;

flag=1;

}

else if(remaining\_time[counter]>0)

{

remaining\_time[counter]-=time\_quantum;

total\_time+=time\_quantum;

}

if(remaining\_time[counter]==0 && flag==1)

{

left\_process--;

wait\_time+=total\_time-arrival\_time[counter]-burst\_time[counter];

turnaround\_time+=total\_time-arrival\_time[counter];

flag=0;

}

if(counter==n-1)

{

counter=0;

}

else if(arrival\_time[counter+1]<=total\_time)

{

counter++;

}

else

{

counter=0;

}

}

average\_wait\_time = wait\_time / no\_of\_process;

average\_turnaround\_time = turnaround\_time / no\_of\_process;

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

{

largest = 9;

for(i = 0; i < no\_of\_process; i++)

{

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

{

largest = i;

}

}

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

process\_queue[largest].current\_time = total\_time;

process\_queue[largest].wait\_time = process\_queue[largest].current\_time -process\_queue[largest].arrival\_time - process\_queue[largest].burst\_time;

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

process\_queue[largest].status = 1;

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

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

}

average\_wait\_time = wait\_time / no\_of\_process;

average\_turnaround\_time = turnaround\_time / no\_of\_process;

**Complexity:**

Here in the algorithm we see that there is nested loop, i.e., nested for loop is contributing to the complexity.

So, we can say that the complexity of the mentioned code is **‘n2’** which is the square of number of processes.

Or we can say that the complexity of the given problem depends on the square of the number of processes that are present for execution.

**Code:**

The code is implemented using the concepts of system call.

#include<stdio.h>

#include<conio.h>

struct process

{

int arrival\_time, burst\_time, current\_time, wait\_time, turnaround\_time, priority,status;

}process\_queue[10];

int no\_of\_process;

void Sorting\_of\_Arrival\_Time()

{

struct process temp;

int i, j;

for(i = 0; i < no\_of\_process - 1; i++)

{

for(j = i + 1; j < no\_of\_process; 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()

{

printf("\nPress 1 for Round Robin Scheduling");

printf("\nPress 2 for Priority Scheduling");

printf("\nEnter your choice :: ");

int choice;

scanf("%d",&choice);

switch(choice)

{

case 1:

{

int counter,j,n,total\_time,left\_process,flag=0,time\_quantum;

int wait\_time=0,turnaround\_time=0,arrival\_time[10], burst\_time[10],remaining\_time[10];

printf("\nEnter no of process :: ");

scanf("%d",&n);

left\_process=n;

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

{

printf("\nProcess%d\n",counter+1);

printf("Enter Arrival time :: ");

scanf("%d",&arrival\_time[counter]);

printf("Enter Burst time :: ");

scanf("%d",&burst\_time[counter]);

remaining\_time[counter]=burst\_time[counter];

}

printf("Enter Time Quantum ::");

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

printf("\n\nProcess\t|Arrival Time\t|Burst Time\t|Completion Time|Turnaround Time|Waiting Time\n\n");

for(total\_time=0,counter=0;left\_process!=0;)

{

if(remaining\_time[counter]<=time\_quantum && remaining\_time[counter]>0)

{

total\_time+=remaining\_time[counter];

remaining\_time[counter]=0;

flag=1;

}

else if(remaining\_time[counter]>0)

{

remaining\_time[counter]-=time\_quantum;

total\_time+=time\_quantum;

}

if(remaining\_time[counter]==0 && flag==1)

{

left\_process--;

printf("P%d\t|\t%d\t|\t%d\t|\t%d\t|\t%d\t|\t%d\n",counter+1,arrival\_time[counter],burst\_time[counter],total\_time,total\_time-arrival\_time[counter],total\_time-arrival\_time[counter]-burst\_time[counter]);

wait\_time+=total\_time-arrival\_time[counter]-burst\_time[counter];

turnaround\_time+=total\_time-arrival\_time[counter];

flag=0;

}

if(counter==n-1)

{

counter=0;

}

else if(arrival\_time[counter+1]<=total\_time)

{

counter++;

}

else

{

counter=0;

}

}

printf("\nAverage Waiting Time= %f\n",wait\_time\*1.0/n);

printf("Avg Turnaround Time = %f",turnaround\_time\*1.0/n);

break;

}

case 2:

{

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

float wait\_time = 0, turnaround\_time = 0, average\_wait\_time,

average\_turnaround\_time;

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

scanf("%d", &no\_of\_process);

for(i = 0; i < no\_of\_process; i++)

{

printf("\nProcess%d\n", i+1);

printf("Enter Arrival time :: ");

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

printf("Enter Burst time :: ");

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;

}

Sorting\_of\_Arrival\_Time();

process\_queue[9].priority = -10000;

printf("\nProcess Name|Arrival Time|Burst Time\t|Priority\t| Completion Time|Turn Around Time|Waiting Time");

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

{

largest = 9;

for(i = 0; i < no\_of\_process; i++)

{

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

{

largest = i;

}

}

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

process\_queue[largest].current\_time = total\_time;

process\_queue[largest].wait\_time = process\_queue[largest].current\_time - process\_queue[largest].arrival\_time - process\_queue[largest].burst\_time;

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

process\_queue[largest].status = 1;

wait\_time = wait\_time + process\_queue[largest].wait\_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\t\t%d\t\t%d", largest+1, process\_queue[largest].arrival\_time,process\_queue[largest].burst\_time, process\_queue[largest].priority,process\_queue[largest].current\_time, process\_queue[largest].turnaround\_time,process\_queue[largest]. wait\_time,process\_queue[largest].current\_time);

}

average\_wait\_time = wait\_time / no\_of\_process;

average\_turnaround\_time = turnaround\_time / no\_of\_process;

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

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

break;

}

}

getch();

}

**Test Cases:**

**Case 1:**

**A screenshot of a computer

Description automatically generated**

**Case 2:**

A screenshot of a cell phone

Description automatically generated