**Name:** Garvit Joshi

**Registration no.:** 11808472

**Email Address:** [garvitjoshi9@gmail.com](mailto:garvitjoshi9@gmail.com)

**GitHub Link:** <https://github.com/garvit-joshi/OS_Scheduling>

**Code:**

#include<bits/stdc++.h>

#include<iostream>

#include<windows.h>                                    //Sleep() function

using namespace std;

long max\_arrival=-1,min\_arrival=INT\_MAX,warning=1;     //Global Variable

/\*

max\_arrival will store the maximim arrival time of a process

min\_arrival will store the minimul arrival time of a process

warning will store the no. of warning that came in during

program execution(due to user input and constraints)

\*/

struct Process

{

    long pid=0;                                       //Process ID

    long priority=0;                                  //the Priority 0 is highest priority

    long arrival\_time=0;                              //Time At Which Process Came

    long burst\_time=0;                                //The Total Time for which process should run

    long completion\_time=0;                           //Time at which CPU completed the whole process

    long turnaround\_time=0;                           //Turn\_Around\_Time=Completetion\_Time-Arrival\_Time

    long waiting\_time=0;                              //Waiting\_Time=Turn\_Around\_Time-Burst\_Time

    long response\_time=0;                             //RT=CPU got Process first time-Arrival Time

    long remaining\_time=0;                            //Time For Which Process Is Remaining to be Executed

    long CPUtime=-1;                                  //Stores When Process got CPU for first time

};

vector<long> ready\_queue;                             //for round robin(stores all the process Index no. for which remaining time is left)

bool comparison\_Priority(Process a, Process b)        //Driver Function-Sorting According to Priority

{

    return (a.priority < b.priority);

}

bool comparison\_ArrivalTime(Process a,Process b)      //Driver Function-Sorting According to Arrival Time(Acending Order)

{

    return (a.arrival\_time < b.arrival\_time);

}

bool comparison\_PID(Process a,Process b)              //Driver Function-Sorting According to PID(Acending Order)

{

    return (a.pid < b.pid);

}

bool comparison\_RemainingTime(Process a,Process b)    //Driver Function-Sorting According to Remaining Time(Acending Order)

{

    return (a.remaining\_time < b.remaining\_time);

}

/\*

The Above Four Functions Are Used As A Parameter In sort() functions.

They act as helping functions to sort the process according to our need

\*/

long display(bool prompt=false)

{

    /\*

    Display Function Used for displaying the question at the starting of program

    \*/

    time\_t now = time(0);

    char\* dt = ctime(&now);

    cout<< dt;

    cout<<"\n\n\n";

    cout<<"\t\t ||                                                                                         ||\n";

    cout<<"\t\t=================================================================================================\n";

    cout<<"\t\t ||                          Operating System Scheduling                                    ||\n";

    cout<<"\t\t ||                                                    --Garvit Joshi                       ||\n";

    cout<<"\t\t ||                                                                                         ||\n";

    cout<<"\t\t ||/\*Design a scheduling program to implements a Queue with two levels. Level 1 : Fixed     ||\n";

    cout<<"\t\t ||  priority preemptive Scheduling. Level 2 : Round Robin Scheduling For a Fixed priority  ||\n";

    cout<<"\t\t ||  the Priority 0 is highest priority. If one process P1 is scheduled and running, another||\n";

    cout<<"\t\t ||  process P2 with higher priority comes. The New process (high priority) process P2      ||\n";

    cout<<"\t\t ||  preempts currently running process P1 and process P1 will go to second level queue.    ||\n";

    cout<<"\t\t ||  Time for which process will strictly execute must be considered in the multiples of 2. ||\n";

    cout<<"\t\t ||  All the processes in second level queue will complete their execution according to     ||\n";

    cout<<"\t\t ||  round robin scheduling.                                                            \*/  ||\n";

    cout<<"\t\t ||                                                                                         ||\n";

    cout<<"\t\t=================================================================================================\n";

    cout<<"\t\t ||  /\*CONSIDER\*/                                                                           ||\n";

    cout<<"\t\t ||  1.Queue 2 will be processed after Queue 1 becomes empty.                               ||\n";

    cout<<"\t\t ||  2.Priority of Queue 2 has lower priority than in Queue 1.                              ||\n";

    cout<<"\t\t ||                                                                                         ||\n";

    cout<<"\t\t=================================================================================================\n";

    cout<<"\t\t ||                                                                                         ||\n";

    if(prompt==false)

    {

        cout<<"Please Wait While Program Loads . . . ";

        Sleep(5000);

        system("CLS");

        display(true);                       //Recursion

        return 0;

    }

    cout<<"\n";

    cout<<"Program Successfully Loaded\n";

    system("pause");

    system("CLS");

    return 0;

}

long Enter\_Process(long &temp,Process p[],long i)

{

    /\*

    Function To Enter All Processes. This Function will be called as much as

    time the number of Proccess.

    \*/

    cout<<"Process:"<<i+1;

    temp++;                                      //Variable Gives Unique PID(Process ID) To each Process

    p[i].pid=temp;

    cout<<"\nEnter Priority:";

    cin>>p[i].priority;

    while(p[i].priority<0)

    {

        cout<<"\t\t\t\tWarning "<<warning<<": A Process Cannot Have Priority In Negative.\n";

        cout<<"Please Enter Priority Again:";

        cin>>p[i].priority;

        warning++;

    }

    cout<<"Enter Arrival Time:";

    cin>>p[i].arrival\_time;

    while(p[i].arrival\_time<0)

    {

        cout<<"\t\t\t\tWarning "<<warning<<": A Process Cannot Have Arrival Time In Negative.\n";

        cout<<"Please Enter Arrival Time Again:";

        cin>>p[i].arrival\_time;

        warning++;

    }

    if(p[i].arrival\_time<min\_arrival)

    {

        /\*

        Calculating Minimum Arrival time

        \*/

        min\_arrival=p[i].arrival\_time;

    }

    if(p[i].arrival\_time>max\_arrival)

    {

        /\*

        Calculating Maximum Arrival Time

        \*/

        max\_arrival=p[i].arrival\_time;

    }

    cout<<"Enter Burst Time:";

    cin>>p[i].burst\_time;

    while(p[i].burst\_time<0)

    {

        cout<<"\t\t\t\tWarning "<<warning<<": A Process Cannot Have Burst Time In Negative.\n";

        cout<<"Please Enter Burst Time Again:";

        cin>>p[i].burst\_time;

        warning++;

    }

    p[i].remaining\_time=p[i].burst\_time;

    cout<<"====================================================\n";

    return 0;

}

long Show\_Process(Process p[],long n,bool b=false)

{

    if(b==false)

    {

        /\*

        By Default This Conditional Statement Will Work,

        It Will Only Show PID,Priority,Arrival Time,Burst Time

        \*/

        cout<<"\nPID || Priority || Arrival Time || Burst Time\n";

        for(long i=0;i<n;i++)

        {

            cout<<p[i].pid<<"\t"<<p[i].priority<<"\t\t"<<p[i].arrival\_time<<"\t\t"<<p[i].burst\_time<<"\n";

        }

    }

    else if(b==true)

    {

        /\*

        This Works when the function call is called with a third

        parameter which must be true

        \*/

        cout<<"\nPID || Priority || Arrival Time || Burst Time || Completion Time || TurnAround Time || Waiting Time || Response Time\n";

        for(long i=0;i<n;i++)

        {

            cout<<p[i].pid<<"\t"<<p[i].priority<<"\t\t"<<p[i].arrival\_time<<"\t\t"<<p[i].burst\_time<<"\t\t"<<p[i].completion\_time<<"\t\t"<<p[i].turnaround\_time<<"\t\t"<<p[i].waiting\_time<<"\t\t"<<p[i].CPUtime<<"\n";

        }

    }

    return 0;

}

long calculation(Process p[],long n)

{

    /\*

    Function Calculates TurnAround Time,Waiting Time,

    Response Time.

    \*/

    for(long i=0;i<n;i++)

    {

        if(p[i].burst\_time==0)

        {

            p[i].turnaround\_time=0;

            p[i].waiting\_time=0;

            p[i].response\_time=0;

        }

        else

        {

            p[i].turnaround\_time=p[i].completion\_time-p[i].arrival\_time;

            p[i].waiting\_time=p[i].turnaround\_time-p[i].burst\_time;

            p[i].response\_time=p[i].CPUtime-p[i].arrival\_time;

        }

    }

    return 0;

}

long FPPS(Process p[],long n,long &time)

{

    /\*

    Fixed Priority Preemtive Scheduling:Processes are

    Executed in the oreder of there priority

    Less Priority Number=More Priority For That Process

    \*/

    system("CLS");

    if(n==1)

    {

        /\*

        If No Of Processes is One OS we have o just execute

        it in FPPS.

        \*/

        time=p[0].arrival\_time+p[0].burst\_time;

        p[0].completion\_time=time;

        p[0].CPUtime=p[0].arrival\_time;

        return 0;

    }

    time=min\_arrival;

    sort(p, p + n, comparison\_Priority);

    sort(p, p + n, comparison\_ArrivalTime);

    long min\_priority,k,current,small\_priority\_index;

    while(time<=max\_arrival)

    {

        long small\_priority=INT\_MAX;

        for(long i=0;i<n;i++)

        {

            /\*

            loop to find how many processes are in ready queue.

            \*/

            if(p[i].arrival\_time<=time)

            {

                current=i;

                continue;

            }

            else

            {

                /\*

                Value of current signifies the processes index

                which can be executed in the CPU.

                \*/

                break;

            }

        }

        long s=0;

        while(s<=current)

        {

            /\*

            Loop Finds Out The Smallest Priority Of The Current

            Ready Processes

            \*/

            if(p[s].priority<small\_priority && p[s].remaining\_time!=0)

            {

                small\_priority=p[s].priority;

                small\_priority\_index=s;

            }

            s++;

        }

        /\*

        Executes The Process for 1 unit time

        \*/

        p[small\_priority\_index].remaining\_time--;

        if(p[small\_priority\_index].CPUtime==-1)

        {

            /\*

            This Conditional Statement tells

            what was the time when the process

            was first time executed.

            \*/

            p[small\_priority\_index].CPUtime=time;

        }

        time++;

        if(p[small\_priority\_index].remaining\_time==0)

        {

            /\*

            Saves the time when a process was fully executed

            \*/

            p[small\_priority\_index].completion\_time=time;

        }

    }

    /\*

    The Statement Below This Comment Executes a last partially

    Running Process and then exits the function

    \*/

    long remaining\_time=p[small\_priority\_index].remaining\_time;

    if(p[small\_priority\_index].remaining\_time==0)

    {

        /\*

        As Remaining Time is 0,So No Advantage of

        going further

        \*/

        return 0;

    }

    p[small\_priority\_index].remaining\_time=0;

    if(p[small\_priority\_index].CPUtime==-1)

    {

        /\*

        This Conditional Statement Gives

        tells what was the time when the process

        was first time executed.

        \*/

        p[small\_priority\_index].CPUtime=time;

    }

    time+=remaining\_time;

    if(p[small\_priority\_index].remaining\_time==0)

    {

        /\*

        Gives the time when a process was fully executed

        \*/

        p[small\_priority\_index].completion\_time=time;

    }

    return 0;

}

long Round\_Robin(Process p[],long n,long tq,long &time)    //Round Robin Scheduling

{

    if(n==1)

    {

        /\*

        If there is only one process, the Process has been executed in FPPS

        \*/

        return 0;

    }

    /\*Round Robin Scheduling\*/

    long start=-1,remaining\_time=-1,cur=-1;

    sort(p,p+n,comparison\_RemainingTime);              //sort according to Remaining\_time

    for(long i=0;i<n;i++)

    {

        /\*

        Finds the index of Process which does

        not have remaining time as 0.

        \*/

        if(p[i].remaining\_time==0)

        {

            continue;

        }

        else

        {

            start=i;

            break;

        }

    }

    sort(p+start,p+n,comparison\_ArrivalTime);                //sort according to Remaining\_time

    for(long i=0;i<n;i++)

    {

        if(p[i].remaining\_time==0)

        {

            /\*

            If A Process Has Remaining time as zero

            We take a partially running process from

            ready\_queue and execute it

            \*/

            if(!ready\_queue.empty())

            {

                cur=ready\_queue[0];

                ready\_queue.erase(ready\_queue.begin());

                if(p[cur].remaining\_time<=tq)

                {

                    /\*

                    If remaining time is less then or equal to

                    time quantum, then execute the whole process

                    \*/

                    remaining\_time=p[cur].remaining\_time;

                    p[cur].remaining\_time=0;

                    time+=remaining\_time;

                    p[cur].completion\_time=time;

                }

                else

                {

                    /\*

                    If remaining time is more then time quantum,

                    then execute the process for time quantum

                    and then store it in ready\_queue

                    \*/

                    p[cur].remaining\_time-=tq;

                    time+=tq;

                    ready\_queue.push\_back(cur);

                }

            }

            continue;

        }

        else

        {

            if(p[i].arrival\_time<=time)

            {

                if(p[i].remaining\_time<=tq)

                {

                    /\*

                    If remaining time is less then or equal to

                    time quantum, then execute the whole process

                    \*/

                    remaining\_time=p[i].remaining\_time;

                    p[i].remaining\_time=0;

                    if(p[i].CPUtime==-1)

                    {

                        /\*

                        This Conditional Statement Gives

                        tells what was the time when the process

                        was first time executed.

                        \*/

                        p[i].CPUtime=time;

                    }

                    time+=remaining\_time;

                }

                else

                {

                    /\*

                    If remaining time is more then time quantum,

                    then execute the process for time quantum

                    and then store it in ready\_queue

                    \*/

                    p[i].remaining\_time-=tq;

                    if(p[i].CPUtime==-1)

                    {

                        /\*

                        This Conditional Statement Gives

                        tells what was the time when the process

                        was first time executed.

                        \*/

                        p[i].CPUtime=time;

                    }

                    time+=tq;

                    ready\_queue.push\_back(i);

                }

            }

            else

            {

                if(!ready\_queue.empty())

                {

                    cur=ready\_queue[0];

                    ready\_queue.erase(ready\_queue.begin());

                    if(p[cur].remaining\_time<=tq)

                    {

                        /\*

                        If remaining time is less then or equal to

                        time quantum, then execute the whole process

                        \*/

                        remaining\_time=p[cur].remaining\_time;

                        p[cur].remaining\_time=0;

                        time+=remaining\_time;

                        p[cur].completion\_time=time;

                    }

                    else

                    {

                        /\*

                        If remaining time is more then time quantum,

                        then execute the process for time quantum

                        and then again store it in ready\_queue

                        \*/

                        p[cur].remaining\_time-=tq;

                        time+=tq;

                        ready\_queue.push\_back(cur);

                    }

                }

            }

        }

    }

    while(!ready\_queue.empty())

    {

        /\*

        Executes all the processes in ready queue

        \*/

        cur=ready\_queue[0];

        ready\_queue.erase(ready\_queue.begin());

        if(p[cur].remaining\_time<=tq)

        {

            /\*

            If remaining time is less then or equal to

            time quantum, then execute the whole process

            \*/

            remaining\_time=p[cur].remaining\_time;

            p[cur].remaining\_time=0;

            time+=remaining\_time;

            p[cur].completion\_time=time;

        }

        else

        {

            /\*

            If remaining time is more then time quantum,

            then execute the process for time quantum

            and then again store it in ready\_queue

            \*/

            p[cur].remaining\_time-=tq;

            time+=tq;

            ready\_queue.push\_back(cur);

        }

    }

    return 0;

}

int main()

{

    display();

    /\*

    Just Comment The above function call if you are testing the code:

    this function may take upto more then 5 seconds

    \*/

    long n,temp=0,time\_q,time=0;

    cout<<"\t\t\tOperating System Scheduling\n\t\t\t\t\t\t-Garvit Joshi\n";

    cout<<"Enter No. Of Processes:";

    cin >>n;

    while(n<=0)

    {

        cout<<"\t\t\t\tWarning "<<warning<<": Number Of Processes Cannot Be less Then or Equal to 0.\n";

        cout<<"Please Enter No. Of Processes Again:";

        cin>>n;

        warning++;

    }

    Process p[n];

    cout<<"====================================================\n";

    for(long i=0;i<n;i++)

    {

        Enter\_Process(temp,p,i);

    }

    cout<<"Successfully Added The Process:";

    Show\_Process(p,n);

    cout<<"Enter Time Quantum(Multiples Of Two):";

    cin>>time\_q;

    while(time\_q%2!=0)

    {

        /\*

        Time Quantum Should Be In Multiples Of Two

        \*/

        cout<<"\t\t\t\tWarning "<<warning<<": Time Quantum Should In Multiples Of Two:\n";

        cout<<"Enter Time In Multiples Of 2:";

        cin>>time\_q;

        warning++;

    }

    FPPS(p,n,time);                        //Fixed Priority Preemtive Scheduling

    Round\_Robin(p,n,time\_q,time);          //Round Robin Scheduling

    calculation(p,n);

    sort(p,p+n,comparison\_PID);

    Show\_Process(p,n,true);

    cout<<"\n";

    cout<<"All Process Completed In "<<time<<" unit time.\n\n";

    system("pause");

    return 0;

}

**Questions:**