Name: Garvit Joshi **Roll No.:** 51

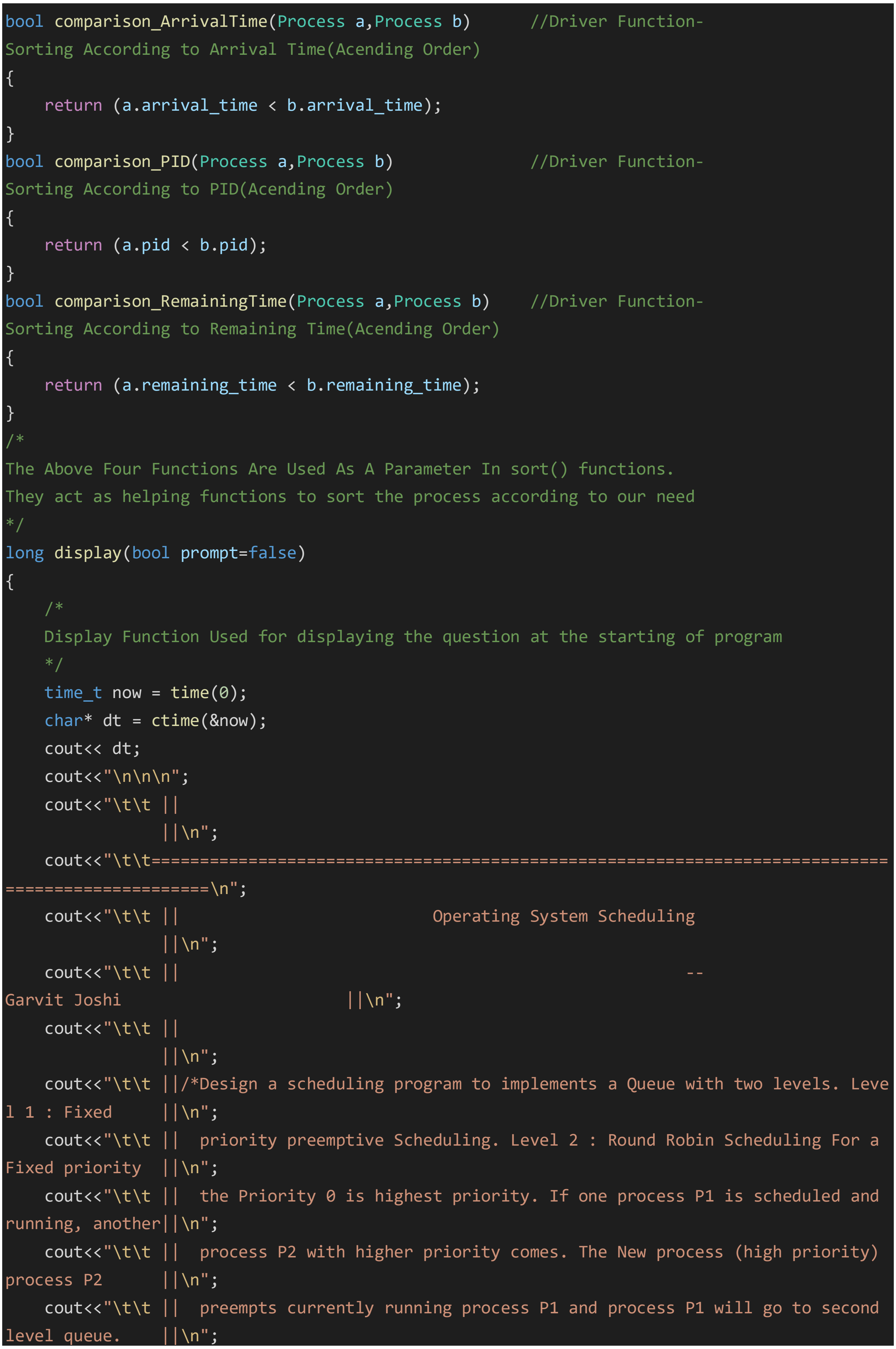
Registration no.: 11808472

Email Address: garvitjoshi9@gmail.com;

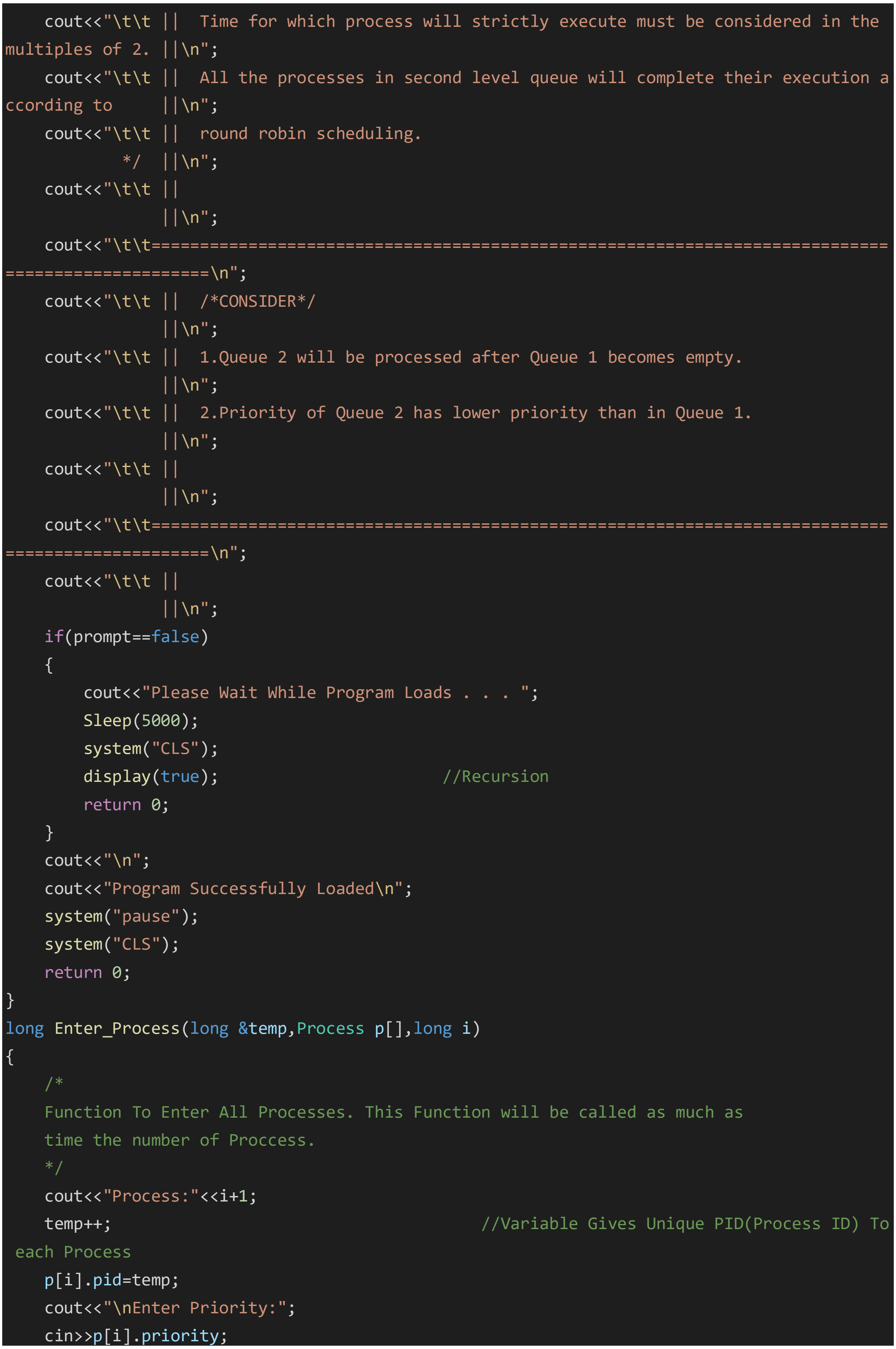
GitHub Link: https://github.com/garvit-joshi/OS_Scheduling

Code of Question 7:

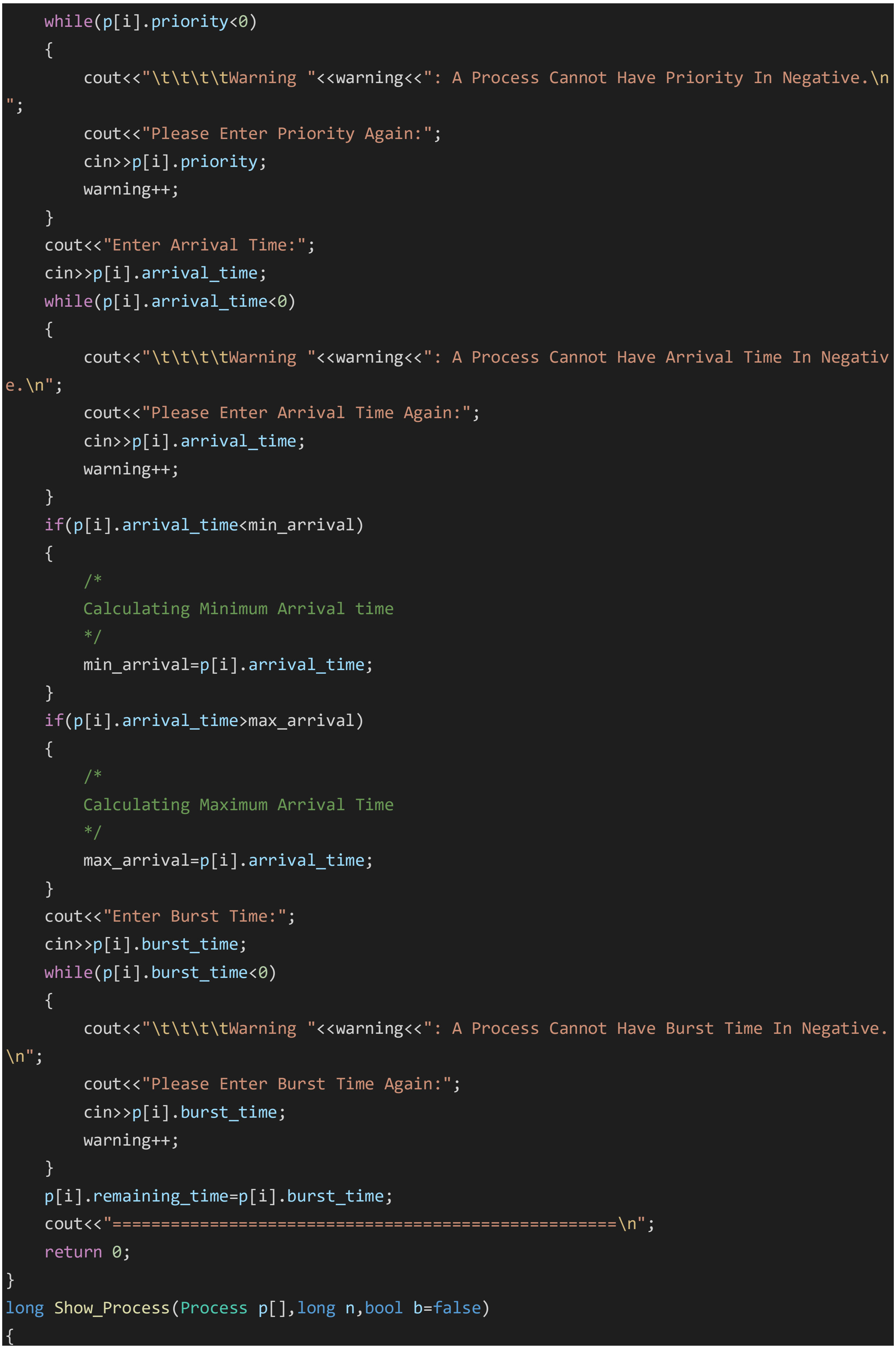
```
#include<bits/stdc++.h>
#include<iostream>
#include<windows.h>                                    //Sleep() function
using namespace std;
long max_arrival=-1,min_arrival=LONG_MAX,warning=1;     //Global Variable
/*
max_arrival will store the maximim arrival time of a process
min_arrival will store 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 sh
ould run
    long completion_time=0;                            //Time at which CPU completed the who
le 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 proc
ess 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);
}
```



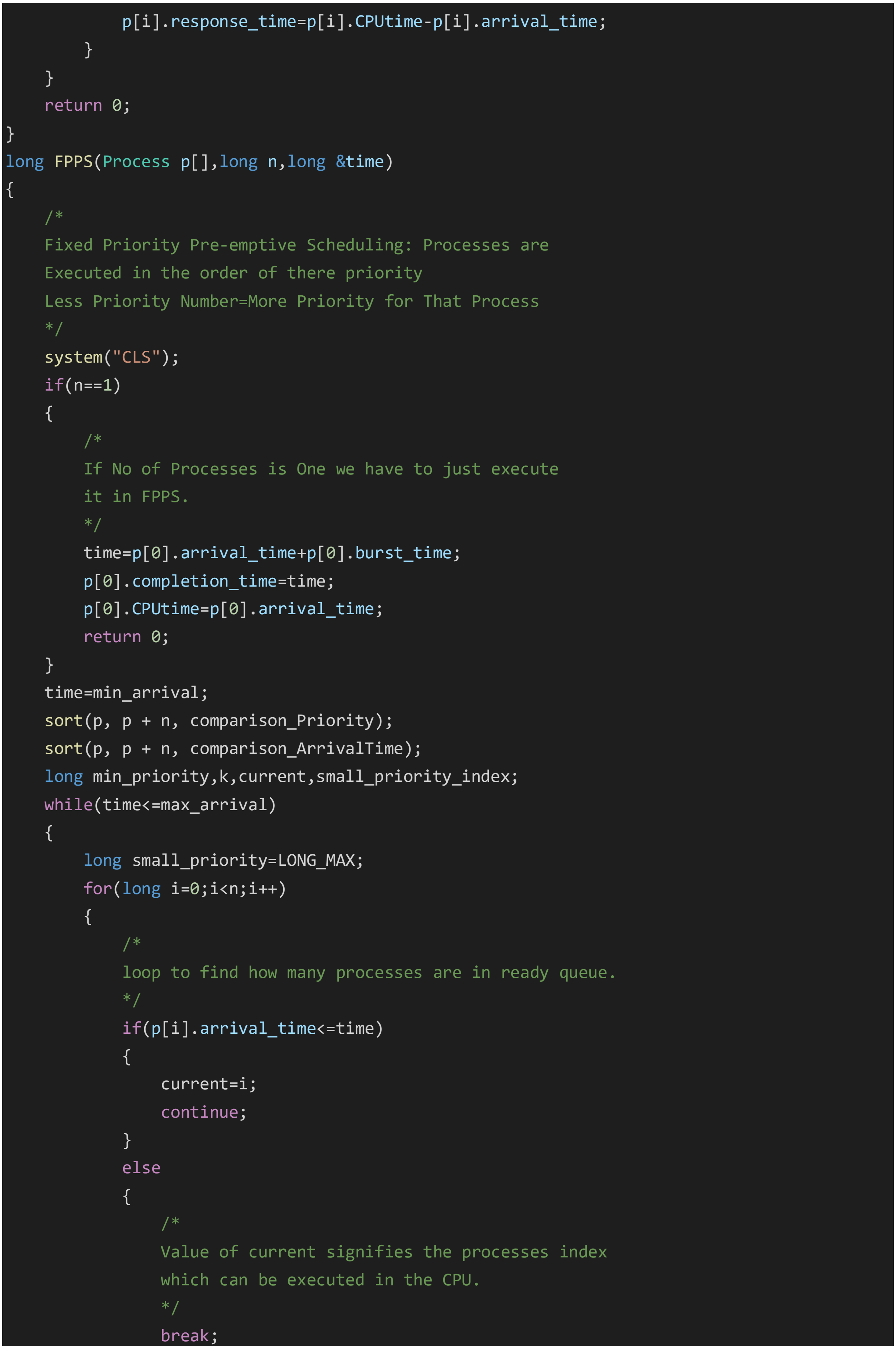
```
bool comparison_ArrivalTime(Process a,Process b)      //Driver Function-
Sorting According to Arrival Time(Acending Order)
{
    return (a.arrival_time < b.arrival_time);
}
bool comparison_PID(Process a,Process b)              //Driver Function-
Sorting According to PID(Acending Order)
{
    return (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. Leve
l 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";
```



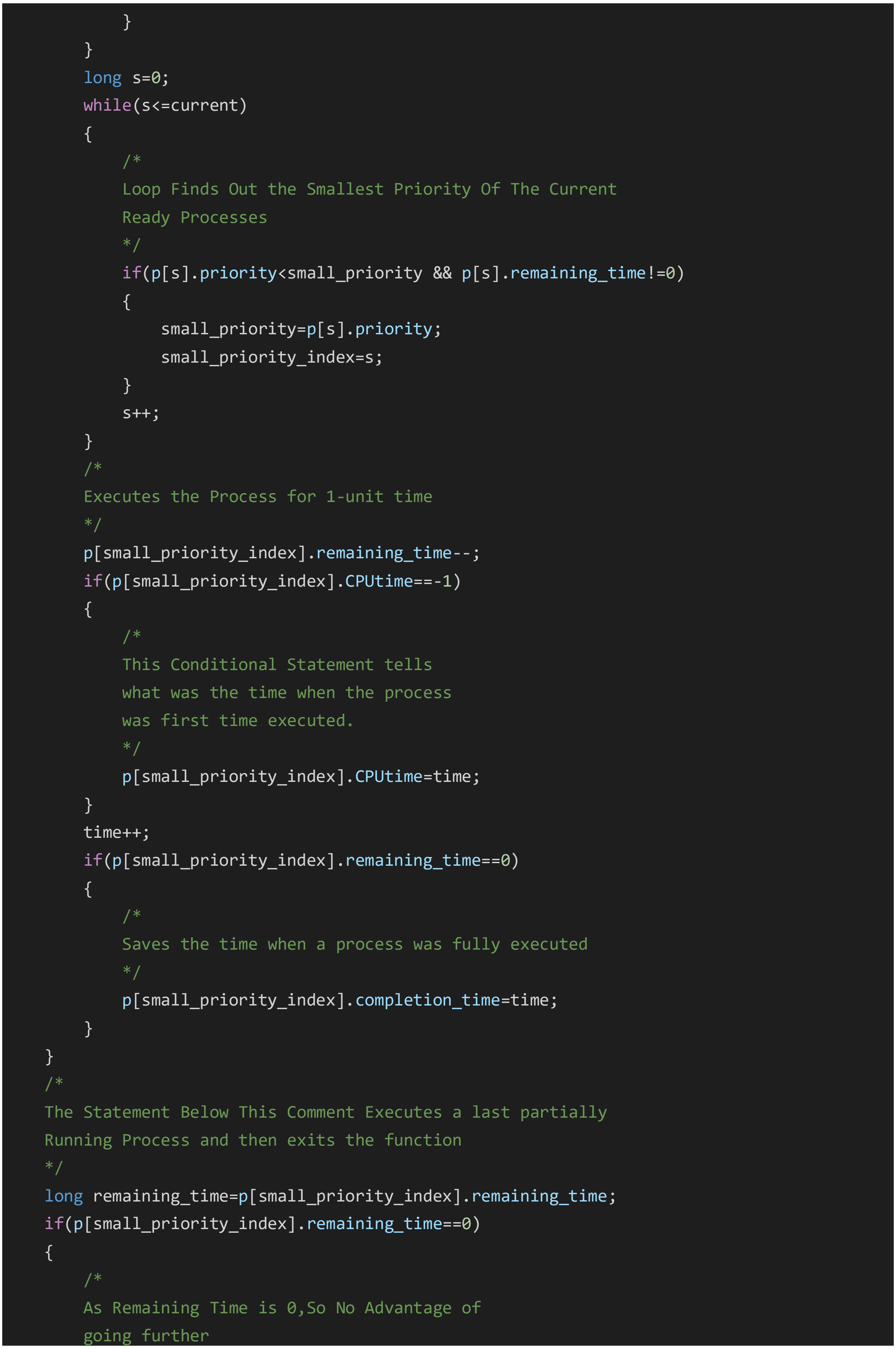
```
    cout<<"\t\t ||  Time for which process will strictly execute must be considered in the
multiples of 2. ||\n";
    cout<<"\t\t ||  All the processes in second level queue will complete their execution a
ccording to     ||\n";
    cout<<"\t\t ||  round robin scheduling.
            */  ||\n";
    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;
```



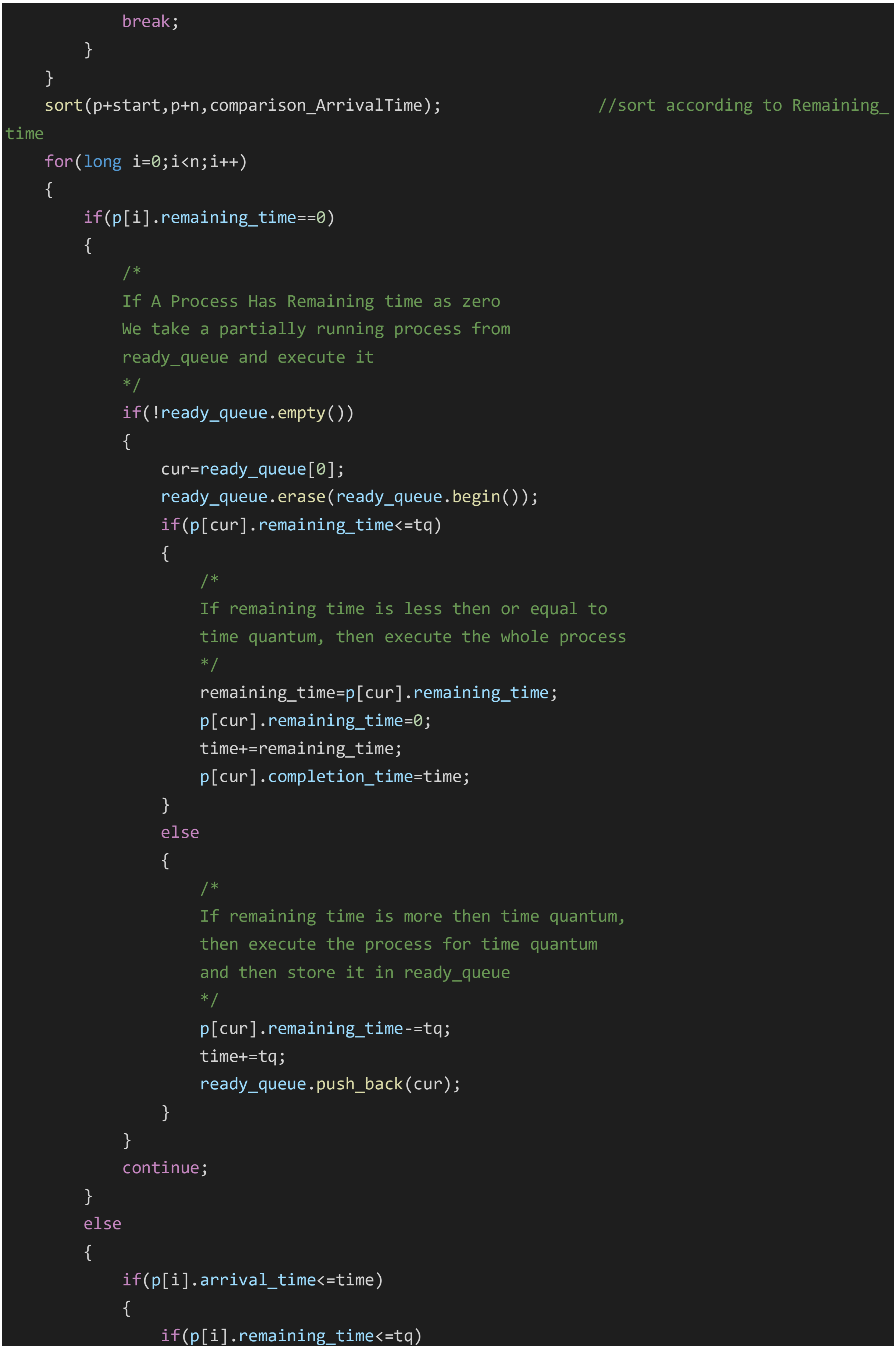
```
    while(p[i].priority<0)
    {
        cout<<"\t\t\t\tWarning "<<warning<<": A Process Cannot Have Priority In Negative.\n
";
        cout<<"Please Enter Priority Again:";
        cin>>p[i].priority;
        warning++;
    }
    cout<<"Enter Arrival Time:";
    cin>>p[i].arrival_time;
    while(p[i].arrival_time<0)
    {
        cout<<"\t\t\t\tWarning "<<warning<<": A Process Cannot Have Arrival Time In Negativ
e.\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;
```



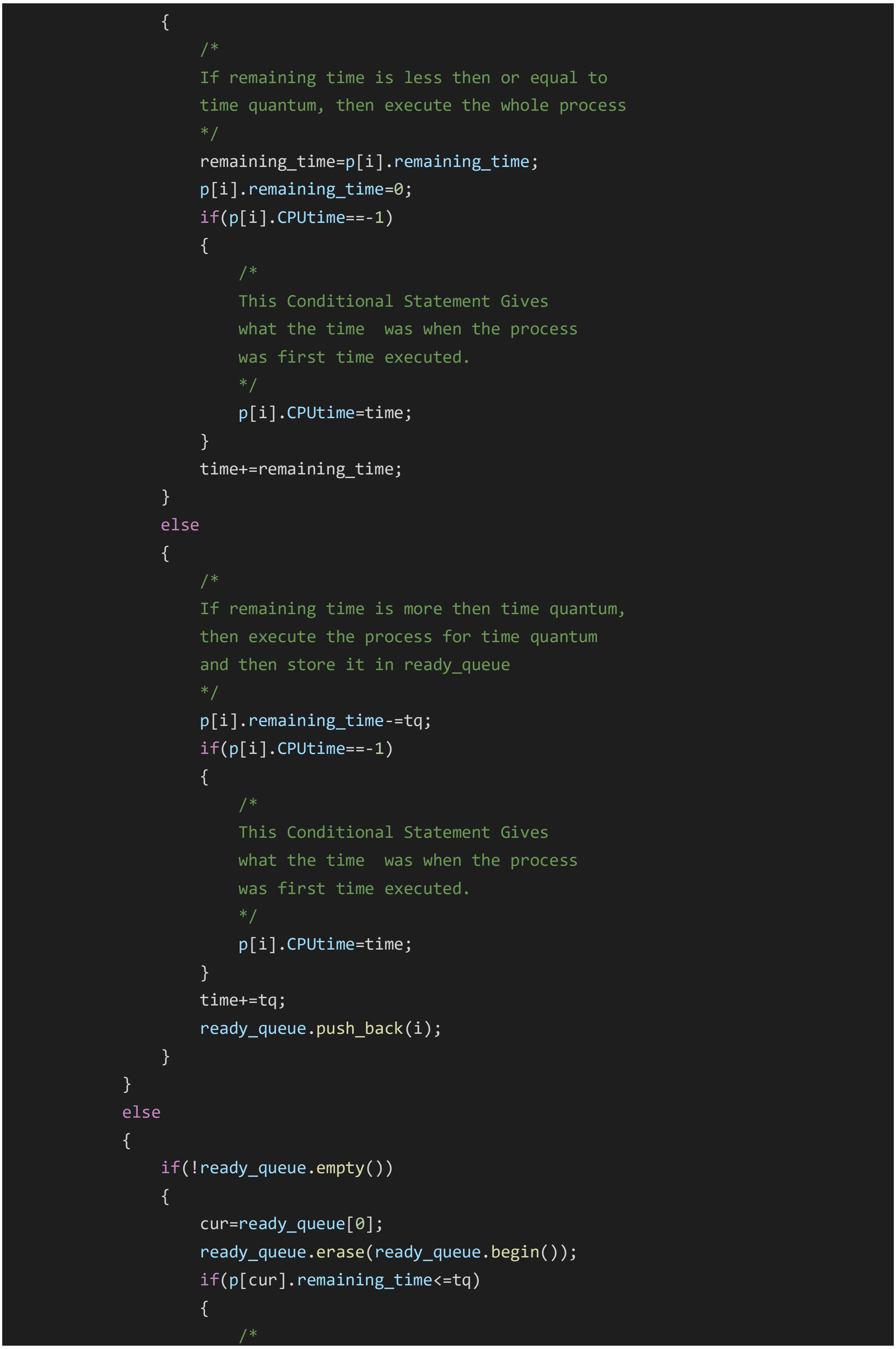
```
            p[i].response_time=p[i].CPUtime-p[i].arrival_time;
        }
    }
    return 0;
}
long FPPS(Process p[],long n,long &time)
{
    /*
    Fixed Priority Pre-emptive Scheduling: Processes are
    Executed in the order of there priority
    Less Priority Number=More Priority for That Process
    */
    system("CLS");
    if(n==1)
    {
        /*
        If No of Processes is One we have to 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=LONG_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;
```



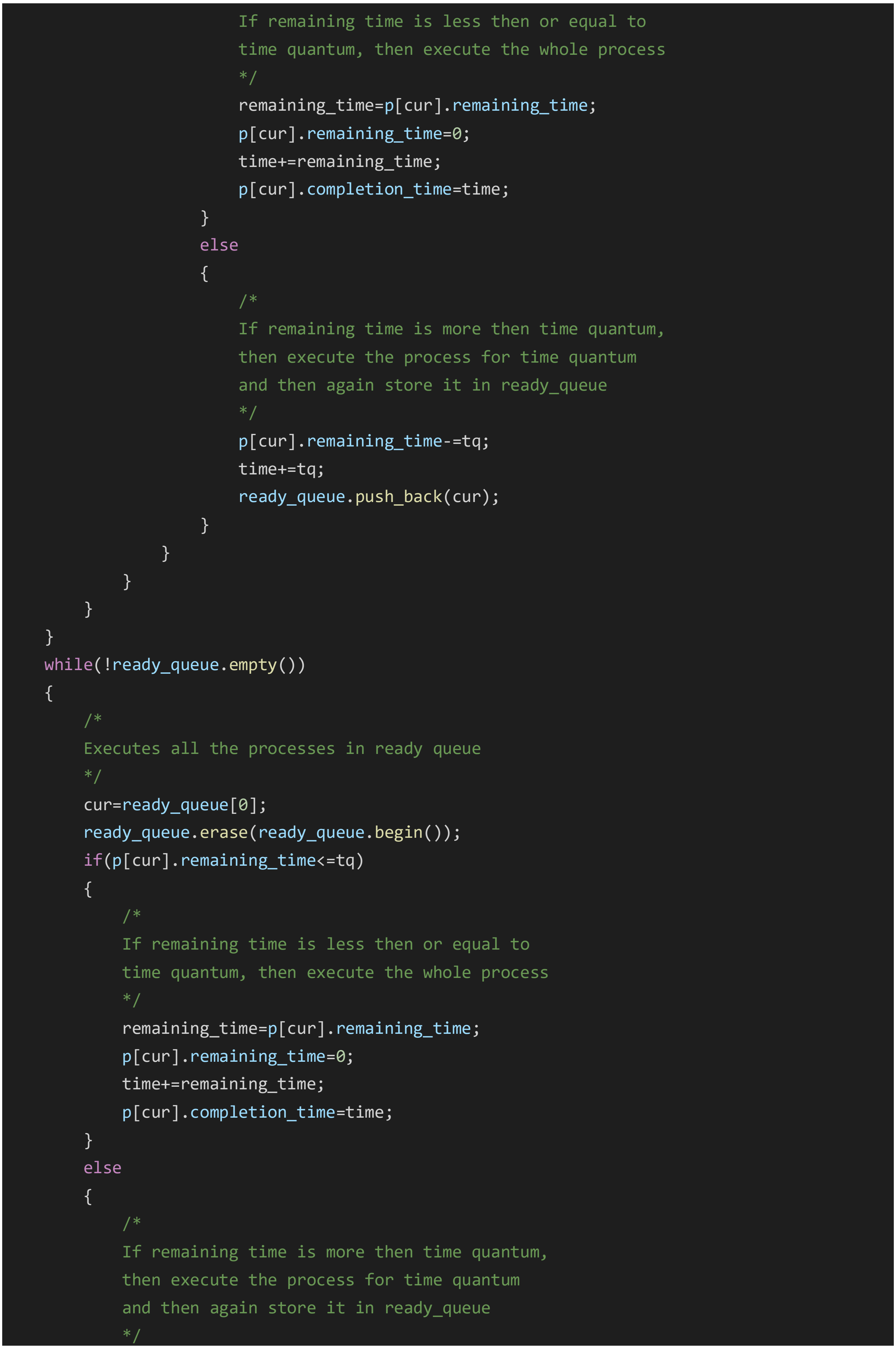
```
            }
        }
        long s=0;
        while(s<=current)
        {
            /*
            Loop Finds Out the Smallest Priority Of The Current
            Ready Processes
            */
            if(p[s].priority<small_priority && p[s].remaining_time!=0)
            {
                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
        what the time was 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;
```



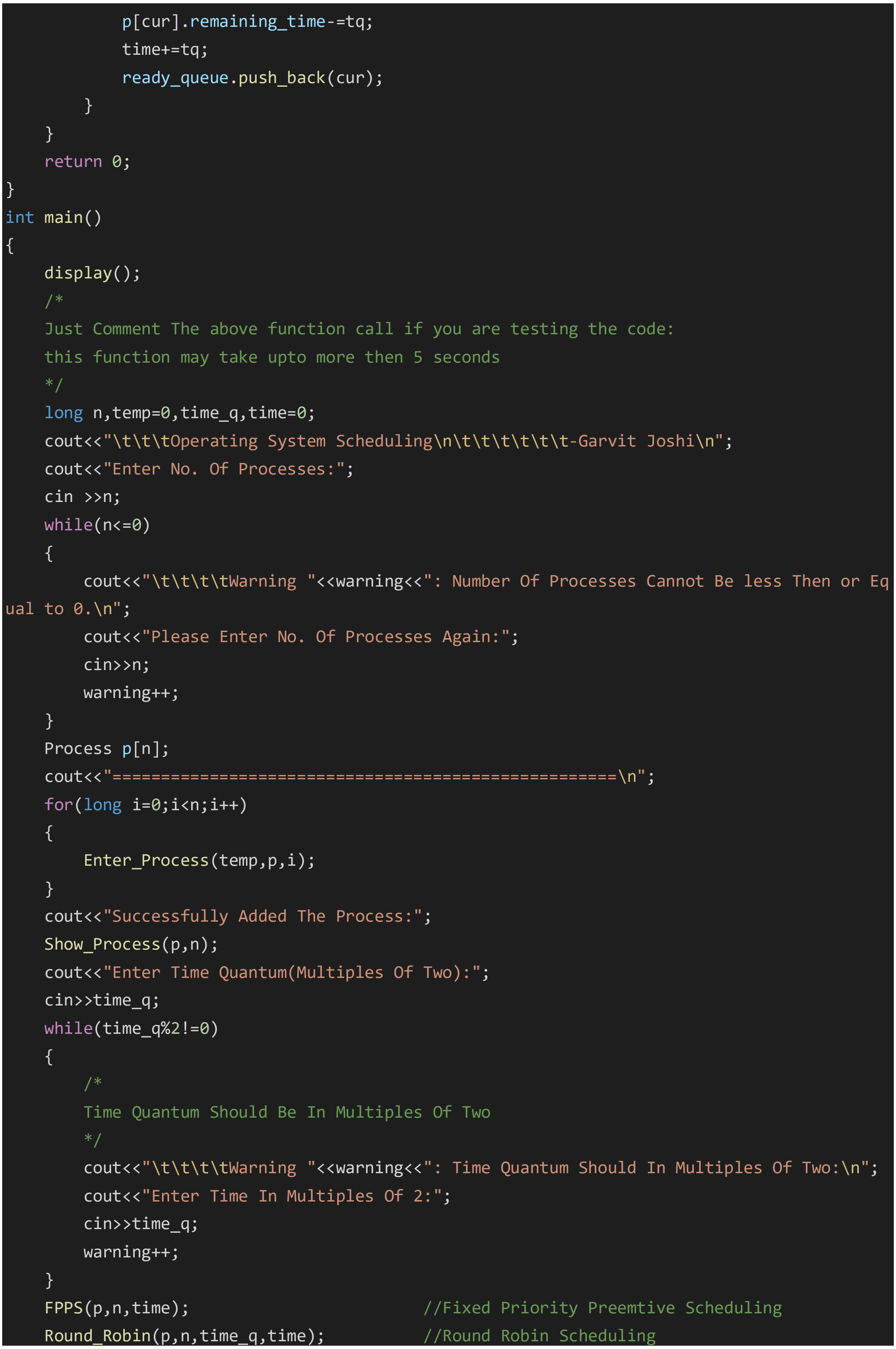
```
            break;
        }
    }
    sort(p+start,p+n,comparison_ArrivalTime);                //sort according to Remaining_
time
    for(long i=0;i<n;i++)
    {
        if(p[i].remaining_time==0)
        {
            /*
            If A Process Has Remaining time as zero
            We take a 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)
```



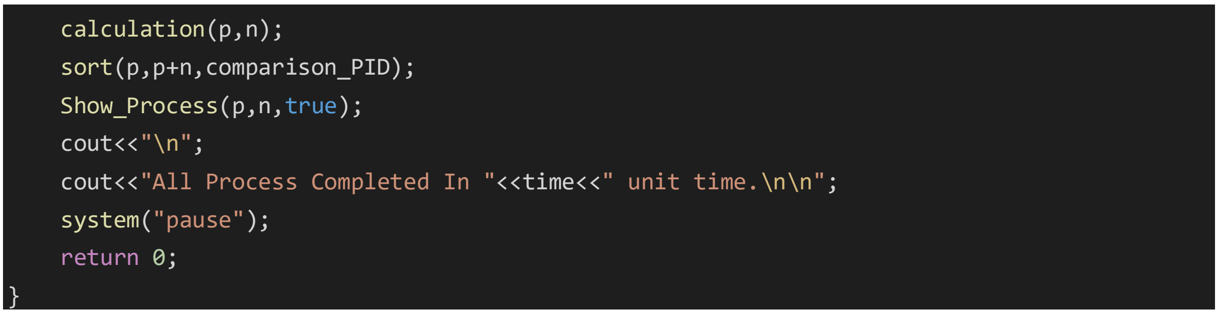
```
            {
                /*
                If remaining time is less then or equal to
                time quantum, then execute the whole process
                */
                remaining_time=p[i].remaining_time;
                p[i].remaining_time=0;
                if(p[i].CPUtime==-1)
                {
                    /*
                    This Conditional Statement Gives
                    what the time  was 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
                    what the time  was 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 Eq
ual 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:

Question 1: Explain the problem in terms of operating system concept? (Max 200 word)

Answer 1: Some Scheduling Algorithm, are non-pre-emptive in nature which means, if a process starts, the CPU executes the process until it ends. Because of this problem, if a process has a very large Burst Time, the process waiting in the queue will have to wait for a long time before they get a chance to be executed, this problem is called Starvation. This happens Mostly in First Come First Serve Scheduling.

So, the scheduling algorithm made by me uses multilevel queue scheduling algorithm, it uses two queues: the high priority queue uses modified Fixed Priority pre-emptive scheduling. The second level queue uses Round Robin Scheduling. When a process is running in Queue 1 and a high priority process comes, so the process that is running pre-empts and now it will run in second level queue i.e. Round Robin Scheduling. Round Robin Scheduling can have time quantum in multiples of two. Queue 2 can only be processed if queue 1 becomes empty.

Question 2: Write the algorithm for proposed solution of the assigned problem.

Answer 2:

```
=>Fixed_Priority_Preemtive_Scheduling (Process p[],long n, long &time)
    1. If n==1
        1.1. There Is only One process Execute it.
    2. time=minimum_arival_time of process
    3. sort all the process according to Priority
    4. sort all the process according to Arrival Time
    5. loop until time <=max_arrival
        5.1. loop from i=0 to i=n;
            5.1.1. find minimum arrival time
        5.2. loop from i=0 to minimum arrival time
```

```
            5.2.1 find smallest_priority_process whose remaining time is left
        5.3. Execute it for 1 Unit time.
        5.4.time =time +1
        5.5. If It is executed for the first time
            5.5.1.  Store the time in process structure
        5.6. If process.remaining time= 0
        5.6.1. It is completed and store its completetion time
    6. Execute the process that has run last at FPPS.
=>Round_Robin_scheduling(Process p[],long n,long tq,long &time)
    1.if n==1
        1.1 exit ,there is only one process that has been executed in FPPS
    2. sort the process according to remaining time
    3.loop from i=0 to i=n:
        3.1Find The Index Of process Which do not have remaining time as 0.
        3.2. Store that index
    4. sort the process according to arrival time
    5.loop from i=0 to i=n:
        5.1. if remaining time==0
            5.1.1. If Ready Queue is not empty
                5.1.1.1.  if  remaining time<=time_quantum
                    5.1.1.1.1 Execute the whole first process in ready queue
                5.1.1.2. else
                    5.1.1.2.1 Execute the process for time quantum
        5.2 else
```

```
            5.2.1 If arrival time<= time
                5.2.1.1.  if remaining time<=time_quantum
                    5.2.2.1.1.1 Execute the whole process
                5.2.1.2. else
                    5.2.1.2.1. Execute the process for time quantum
                    5.2.1.2.2. Put the remaining process into ready queue
            5.2.2. else
                5.2.2.1. If Ready Queue is not empty
                    5.2.2.1.1. if  remaining time<=time_quantum
                        5.2.2.1.1.1. Execute the whole first process in ready queue
                5.2.2.2. else
                    5.2.2.2.1. Execute the process for time quantum
                    5.2.2.2.2. Put the remaining process to ready queue
6. loop while ready queue does not become empty
    6.1.take the first Process from ready queue.
    6.2. if  remaining time<=time_quantum
        6.2.1. Execute the whole first process in ready queue
    6.3. else
        6.3.1. Execute the process for time quantum
        6.3.2. Put the remaining process to ready queue
7.exit.
```

Question 3: Calculate complexity of implemented algorithm. (Student must specify complexity of each line of code along with overall complexity)

Answer 3:

The C++ 11 standard sort function has complexity of O(NlogN) time in worst case, where N is no. of processes being sorted.

The Function Signifying Fixed Priority Pre-emptive Scheduling has complexity of O(T-t).

Where, T=max Arrival_Time Of Process and t=time at which the process started.

The Function Signifying Round Robin Scheduling has complexity of O(N).

Where, N= No. Of process

All Other Functions (calculation, show_process, Enter_Process) take O(N) time where N is no. of Process.

The Whole Program May have variable Frequency. The complexity of whole program may be dependent on user input. If Max_Arrival_Time of Process-Time at wich process started is greater than (Number_Of_Processes(log Number_Of_Processes) then complexity of whole code is O(T-t).

Question 4: Explain all the constraints given in the problem. Attach the code snippet of the implemented constraint.

Answer 4: The Constraints are:

1. Time Quantum should be in the multiples of two.
2. Arrival Time, Priority, Burst Time Cannot Be in Negative.
3. All the variables are of long variable type that is there range can be between -2,147,483,647 to 2,147,483,647.

If the 1. And 2. Constraints are not met then a warning is shown that tells user about constraints.

Code Snippet:

1. When Time Quantum is not given in multiples of two,

```
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++;
}
```

2. When Arrival time Is Gives Less Then 0, and

```
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++;
}
```

3. All Variables Are long in type

```
long n,temp=0,time_q,time=0;
struct Process
{
      long pid=0;
      long priority=0;
      long arrival_time=0;
      long burst_time=0;
      long completion_time=0;
      long turnaround_time=0;
      long waiting_time=0;
      long response_time=0;
      long remaining_time=0
      long CPUtime=-1;
};
```

Question 5: If you have implemented any additional algorithm to support the solution, explain the need and usage of the same.

Answer 5: Yes, I Implemented One algorithm that is

```
sort(start_Address, End_Address, Binary_Function);
```

I Used This Function to sort a structure process according to my needs(with the help of binary functions)

First Parameter: start_address: Tells the function from where to sort a process i.e. starting point

Second Parameter: End_address: Tells the function to end the sot function at desired address.

Third Parameter: Binary_Function (Optional) It tells the compiler to sort according to our need. It returns true or false value.

Example of Helper Function:

```
bool comparison_Priority(Process a, Process b)
{
    return (a.priority < b.priority);
}
sort(p, p + n, comparison_Priority);
```

Question 6: Explain the boundary conditions of the implemented code.

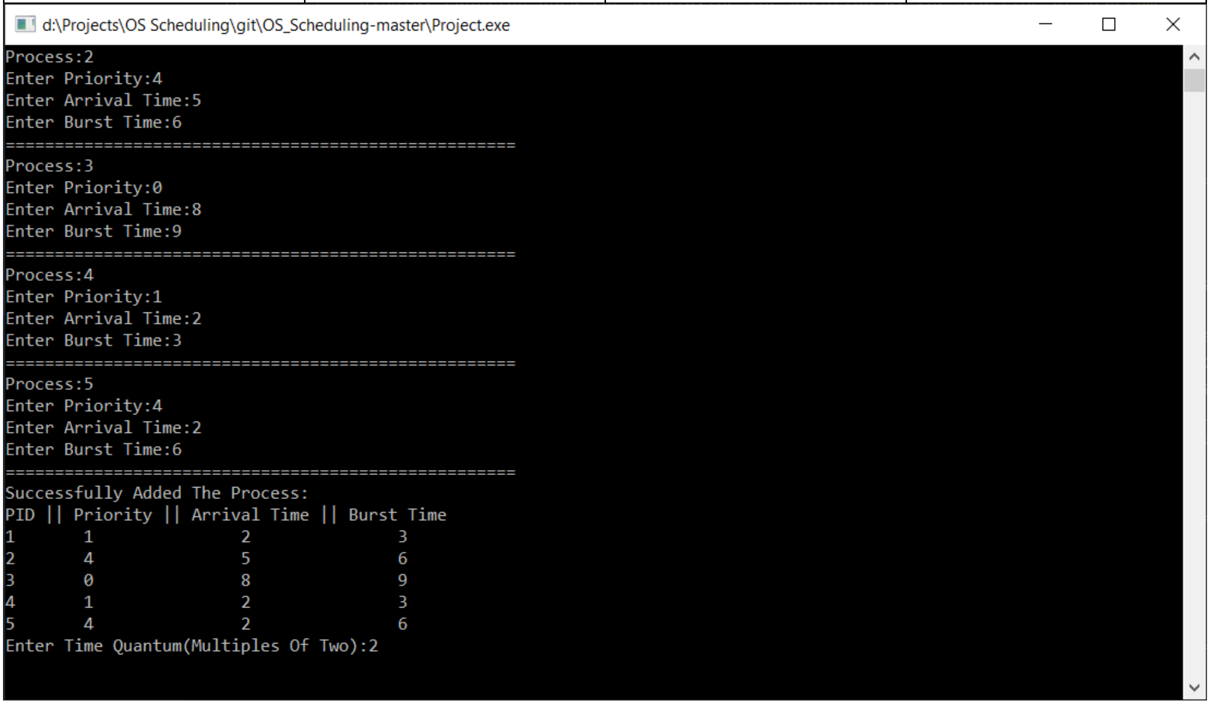
Answer 6: All the Variables can store values between -2,147,483,647 to 2,147,483,647, but I have prohibited using negative values as process cannot have arrival time, burst time, and priority in negative. So, the values can range from 0 to 2,147,483,647.

Question 7: Explain all the test cases applied on the solution of assigned problem.

Answer 7:

Input:

PID	Priority	Arrival Time	Burst Time
1	1	2	3
2	4	5	6
3	0	8	9
4	1	2	3
5	4	2	6



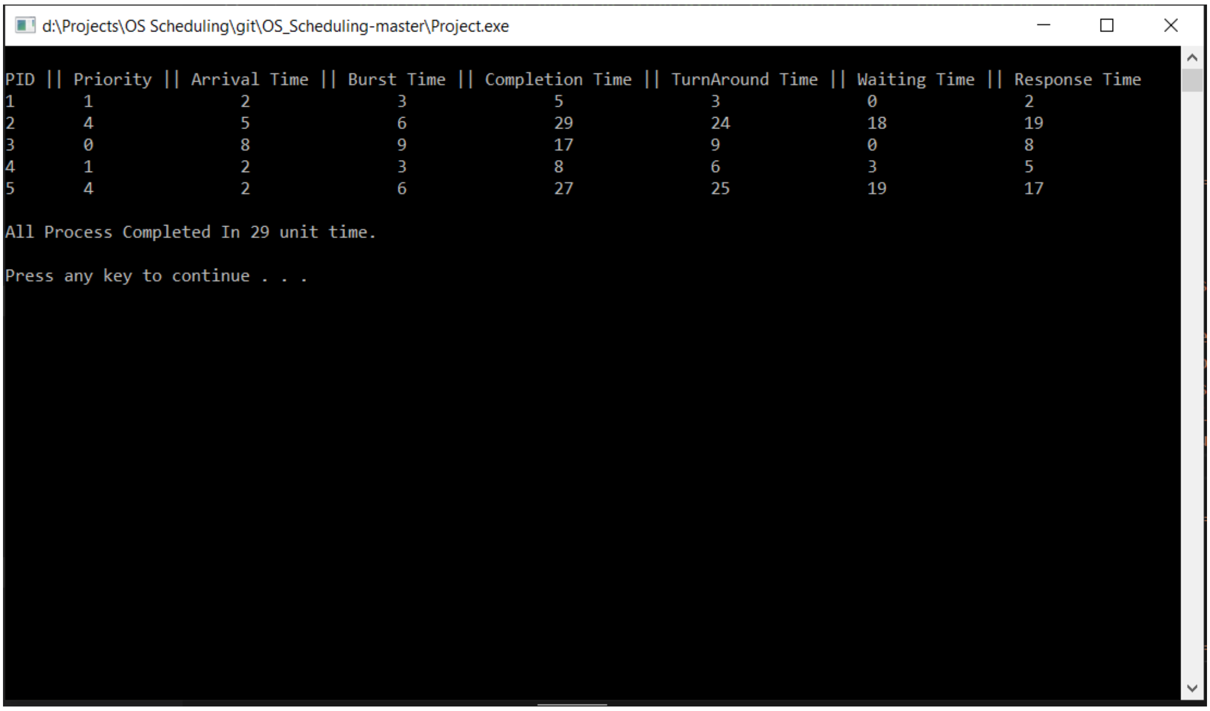
d:\Projects\OS Scheduling\git\OS_Scheduling-master\Project.exe

```
Process:2
Enter Priority:4
Enter Arrival Time:5
Enter Burst Time:6
====================================================
Process:3
Enter Priority:0
Enter Arrival Time:8
Enter Burst Time:9
====================================================
Process:4
Enter Priority:1
Enter Arrival Time:2
Enter Burst Time:3
====================================================
Process:5
Enter Priority:4
Enter Arrival Time:2
Enter Burst Time:6
====================================================
Successfully Added The Process:
PID || Priority || Arrival Time || Burst Time
1       1               2               3
2       4               5               6
3       0               8               9
4       1               2               3
5       4               2               6
Enter Time Quantum(Multiples Of Two):2
```

Output:

PID	Priority	Arrival Time	Burst Time	Completion Time	Turnaround Time	Waiting Time	Response Time
1	1	2	3	5	3	0	2
2	4	5	6	29	24	18	19
3	0	8	9	17	9	0	8
4	1	2	3	8	6	3	5
5	4	2	6	27	25	19	17

All Process Completed In 29 unit time.



d:\Projects\OS Scheduling\git\OS_Scheduling-master\Project.exe

```
PID || Priority || Arrival Time || Burst Time || Completion Time || TurnAround Time || Waiting Time || Response Time
1       1               2               3               5               3               0               2
2       4               5               6               29              24              18              19
3       0               8               9               17              9               0               8
4       1               2               3               8               6               3               5
5       4               2               6               27              25              19              17

All Process Completed In 29 unit time.

Press any key to continue . . .
```

Status: Passed

Question 8: Have you made minimum 5 revisions of solution on GitHub?

Answers 8: Yes, there are more than 50 commits in the repository

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