OPERATING SYSTEM REPORT

Operating Systems (CSE 316)

Submitted by:

Sr. No	Registration No	Name of student	Section	Group
1	***12213334***	**SURAJ JUMAR**	K21GP	**2**

School of computer Science and Engineering

Submitted To: Mrs. Cherry Khosla

Lovely Professional University

Jalandhar, Punjab, India-144411

Question:

Write a program for multilevel queue scheduling algorithm. There must be three queues generated. There must be specific range of priority associated with every queue. Now prompt the user to enter number of processes along with their priority and burst time. Each process must occupy the respective queue with specific priority range according to its priority. Apply round robin algorithm with quantum time 4 on queue with highest priority range. Apply priority scheduling algorithm on the queue with medium range of priority and first come first serve algorithm on the queue with lowest range of priority. Each and every queue should get a quantum time of 10 seconds. CPU will keep on shifting between queues after every 10 seconds.

Introduction

Multilevel Queue (MLQ) CPU Scheduling

It may happen that processes in the ready queue can be divided into different classes where each class has its own scheduling needs. For example, a common division is a foreground (interactive) process and a background (batch) process. These two classes have different scheduling needs. For this kind of situation, Multilevel Queue Scheduling is used.

- Multiple queues: In MLQ scheduling, processes are divided into multiple queues based on their priority, with each queue having a different priority level. Higher-priority processes are placed in queues with higher priority levels, while lower-priority processes are placed in queues with lower priority levels.
- Priorities assigned: Priorities are assigned to processes based on their type, characteristics, and importance. For example, interactive processes like user input/output may have a higher priority than batch processes like file backups.
- Preemption: Preemption is allowed in MLQ scheduling, which means a higher priority process can preempt a lower priority process, and the CPU is allocated to the higher priority process. This helps ensure that high-priority processes are executed in a timely manner.
- Scheduling algorithm: Different scheduling algorithms can be used for each queue, depending on the requirements of the processes in that queue. For example, Round Robin scheduling may be used for interactive processes, while First Come First Serve scheduling may be used for batch processes.
- Feedback mechanism: A feedback mechanism can be implemented to adjust the priority of a process based on its behavior over time. For example, if an interactive process has been waiting in a lower-priority queue for a long time, its priority may be increased to ensure it is executed in a timely manner.
- Efficient allocation of CPU time: MLQ scheduling ensures that processes with higher priority levels are executed in a timely manner, while still allowing lower priority processes to execute when the CPU is idle.
- Fairness: MLQ scheduling provides a fair allocation of CPU time to different types of processes, based on their priority and requirements.
- Customizable: MLQ scheduling can be customized to meet the specific requirements of different types of processes.

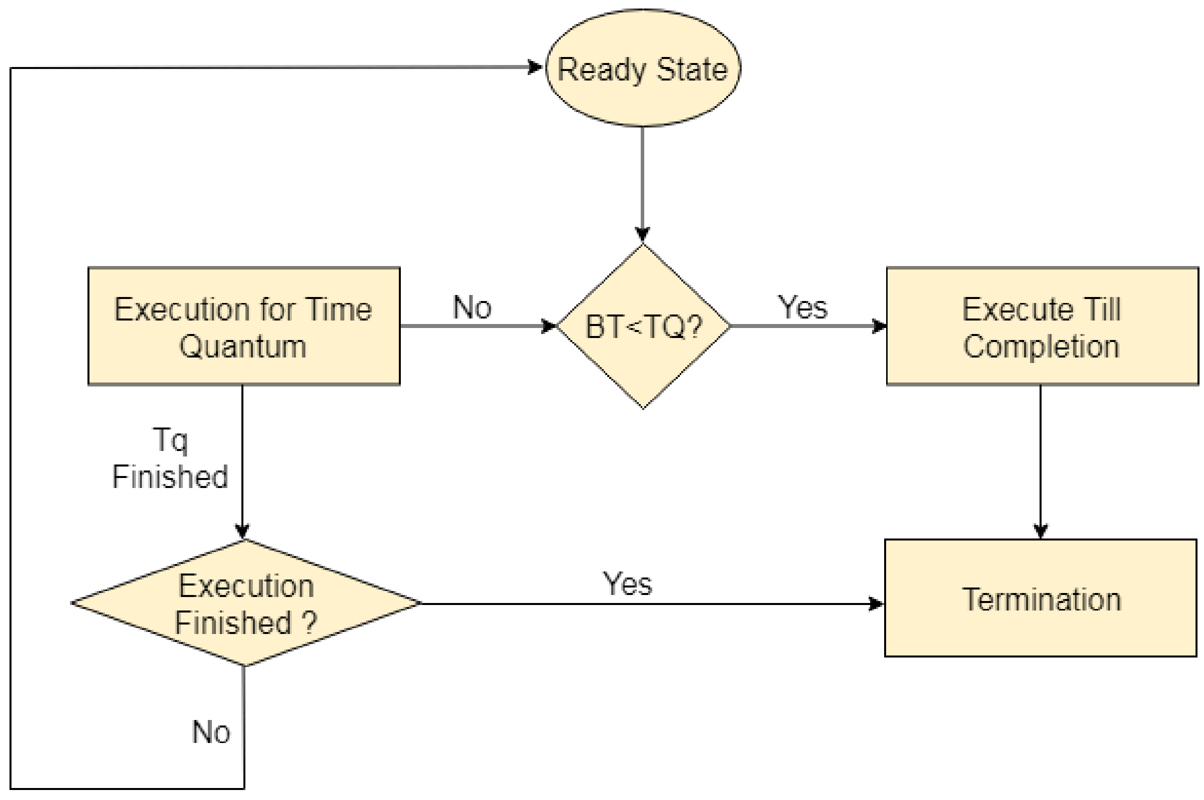
Round Robin CPU Scheduling

Round Robin CPU Scheduling is the most important CPU Scheduling Algorithm which is ever used in the history of CPU Scheduling Algorithms. Round Robin CPU Scheduling uses Time Quantum (TQ). The Time Quantum is something which is removed from the Burst Time and lets the chunk of process to be completed.

First, the processes which are eligible to enter the ready queue enter the ready queue. After entering the first process in Ready Queue is executed for a Time Quantum chunk of time. After execution is complete, the process is removed from the ready queue. Even now the process requires some time to complete its execution, then the process is added to Ready Queue.

The Ready Queue does not hold processes which already present in the Ready Queue. The Ready Queue is designed in such a manner that it does not hold non unique processes. By holding same processes Redundancy of the processes increases.

After, the process execution is complete, the Ready Queue does not take the completed process for holding.



Priority Scheduling Algorithm

In Priority scheduling, there is a priority number assigned to each process. In some systems, the lower the number, the higher the priority. While, in the others, the higher the number, the higher will be the priority. The Process with the higher priority among the available processes is given the CPU. There are two types of priority scheduling algorithm exists. One is Preemptive priority scheduling while the other is Non Preemptive Priority scheduling.

The priority number assigned to each of the process may or may not vary. If the priority number doesn't change itself throughout the process, it is called static priority, while if it keeps changing itself at the regular intervals, it is called dynamic priority.

First Come First Serve

First Come First Serve CPU Scheduling Algorithm shortly known as FCFS is the first algorithm of CPU Process Scheduling Algorithm. In First Come First Serve Algorithm what we do is to allow the process to execute in linear manner.

This means that whichever process enters process enters the ready queue first is executed first. This shows that First Come First Serve Algorithm follows First In First Out (FIFO) principle.

CODE

```
#include <iostream>
using namespace std;

struct process{
    int priority;
    int burst_time;
    int tt_time;
    int total_time=0;
};

struct queues{
    int priority_start;
    int priority_end;
    int total_time=0;
    int length = 0;
    process *p;
    bool executed = false;
};
```

```
bool notComplete(queues q[]){
    bool a=false;
    int countInc=0;
        for(int i=0;i<3;i++){
                countInc=0;
            for(int j=0;j<q[i].length;j++){
                if(q[i].p[j].burst_time != 0){
                    a=true;
                }
                else{
                    countInc+=1;
                }
            }
            if(countInc==q[i].length){

                q[i].executed = true;
            }
        }
        return a;
}
```

```
void sort_ps(queues q){
   //Queue q has to be sorted according to priority of processes
   for(int i=1;i<q.length;i++){
      for(int j=0;j<q.length-1;j++){
         if(q.p[j].priority<q.p[j+1].priority){
            process temp = q.p[j+1];
            q.p[j+1] = q.p[j];
            q.p[j] = temp;
         }
      }
   }
}


void checkCompleteTimer(queues q[]){
   bool a = notComplete(q);
   for(int i=0;i<3;i++){
      if(q[i].executed==false){
         for(int j=0;j<q[i].length;j++){
            if(q[i].p[j].burst_time!=0){
               q[i].p[j].total_time+=1;
            }
         }
         q[i].total_time+=1;
      }
   }
}
```

```
main(){

   //Initializing 3 queues
   queues q[3];
   q[0].priority_start = 7;
   q[0].priority_end = 9;
   q[1].priority_start = 4;
   q[1].priority_end = 6;
   q[2].priority_start = 1;
   q[2].priority_end = 3;

   int no_of_processes,priority_of_process,burst_time_of_process;
   //Prompt User for entering Processes and assigning it to respective
queues.
   cout<<"Enter the number of processes\n";
   cin>>no_of_processes;
   process p1[no_of_processes];

   for(int i=0;i<no_of_processes;i++){
      cout<<"Enter the priority of the process\n";
      cin>>priority_of_process;
      cout<<"Enter the burst time of the process\n";
      cin>>burst_time_of_process;
      p1[i].priority = priority_of_process;
      p1[i].burst_time = burst_time_of_process;
      p1[i].tt_time = burst_time_of_process;
      for(int j=0;j<3;j++){
      if(q[j].priority_start<=priority_of_process &&
priority_of_process<=q[j].priority_end){
         q[j].length++;
      }
      }
   }
```

```
    for(int i =0;i<3;i++){
        int len = q[i].length;
        q[i].p = new process[len];
    }



    int a=0;
    int b=0;
    int c=0;

    for(int i =0;i<3;i++){
        for(int j=0;j<no_of_processes;j++){
            if((q[i].priority_start<=p1[j].priority) &&
(p1[j].priority<=q[i].priority_end)){
                if(i==0){
                    q[i].p[a++] = p1[j];

                    }
                else if(i==1){
                    q[i].p[b++] = p1[j];
                    }
                else{
                    q[i].p[c++] = p1[j];
                    }
            }
        }
    }
```

```
a--;b--;c--;
for(int i=0;i<3;i++){
   cout<<"Queue "<<i+1<<" : \t";
   for(int j=0;j<q[i].length;j++){
      cout<<q[i].p[j].priority<<"->";
   }
   cout<<"NULL\n";
}


//While RR on multiple queues is not complete, keep on repeating
int timer = 0;
int l =-1;
int rr_timer = 4;
int counter=0;
int counterps=0;
int counterfcfs=0;
while(notComplete(q)){
   if(timer == 10){
      timer = 0;
   }
   l+=1;
   if(l>=3){
      l=l%3;
   }
```

```
//Process lth queue if its already not executed
//If its executed change the value of l
if(q[l].executed == true){
    cout<<"Queue "<<l+1<<" completed\n";
  l+=1;
  if(l>=3){
    l=l%3;
  }
  continue;
}

//Finally you now have a queue which is not completely
executed
//Process the incomplete processes over it

if(l==0){
  cout<<"Queue "<<l+1<<" in hand\n";
  //Round Robin Algorithm for q=4
  if(rr_timer == 0){
    rr_timer = 4;
  }
```

```
    if(l==0){
        cout<<"Queue "<<l+1<<" in hand\n";
        //Round Robin Algorithm for q=4
        if(rr_timer == 0){
            rr_timer = 4;
        }

        for(int i=0;i<q[l].length;i++){
            if(q[l].p[i].burst_time==0){
                counter++;
                continue;
            }
            if(counter == q[l].length){
                break;
            }
            while(rr_timer>0 && q[l].p[i].burst_time!=0 &&
timer!=10){
                cout<<"Executing queue 1 and "<<i+1<<" process for a
unit time. Process has priority of "<<q[l].p[i].priority<<"\n";
                q[l].p[i].burst_time--;
                checkCompleteTimer(q);
                rr_timer--;
                timer++;

            }
```

```
        if(timer == 10){
            break;
        }
        if(q[l].p[i].burst_time==0 && rr_timer ==0){
            rr_timer = 4;
            if(i == (q[i].length-1)){
                i=-1;
            }
            continue;
        }
        if(q[l].p[i].burst_time==0 && rr_timer > 0){
            if(i == (q[i].length-1)){
                i=-1;
            }
            continue;
        }
        if(rr_timer <= 0){
            rr_timer = 4;
            if(i == (q[i].length-1)){
                i=-1;
            }
            continue;
        }

    }
}
```

```
else if(l==1){
    cout<<"Queue "<<l+1<<" in hand\n";
    sort_ps(q[l]);
    //Priority Scheduling
    for(int i=0;i<q[l].length;i++){
        if(q[l].p[i].burst_time==0){
            counterps++;
            continue;
        }
        if(counterps == q[l].length){
            break;
        }
        while(q[l].p[i].burst_time!=0 && timer!=10){
            cout<<"Executing queue 2 and "<<i+1<<" process for a unit
time. Process has priority of "<<q[l].p[i].priority<<"\n";
            q[l].p[i].burst_time--;
            checkCompleteTimer(q);
            timer++;

        }
        if(timer == 10){
            break;
        }
        if(q[l].p[i].burst_time==0){
                continue;
        }

    }
}
```

```
else{
    cout<<"Queue "<<l+1<<" in hand\n";
    //FCFS
    for(int i=0;i<q[l].length;i++){
        if(q[l].p[i].burst_time==0){
            counterfcfs++;
            continue;
        }
        if(counterfcfs == q[l].length){
            break;
        }
        while(q[l].p[i].burst_time!=0 && timer!=10){
            cout<<"Executing queue 3 and "<<i+1<<" process for a unit
time. Process has priority of "<<q[l].p[i].priority<<"\n";
            q[l].p[i].burst_time--;
            checkCompleteTimer(q);

            timer++;
        }
        if(timer == 10){
            break;
        }
        if(q[l].p[i].burst_time==0){
                continue;
        }
```

```
        }

      }
      cout<<"Broke from queue "<<l+1<<"\n";
   }

   for(int i=0;i<3;i++){
         cout<<"\nTime taken for queue "<<i+1<<" to execute:
"<<q[i].total_time<<"\n";
      for(int j=0;j<q[i].length;j++){
         cout<<"Process "<<j+1<<" of queue "<<i+1<<" took
"<<q[i].p[j].total_time<<"\n";
      }
   }
```

```
    int sum_tt=0;
    int sum_wt=0;

    cout<<"\n\nProcess     | Turn Around Time | Waiting Time\n";
    for(int i=0;i<3;i++){
          cout<<"Queue "<<i+1<<"\n";
       for(int j=0;j<q[i].length;j++){
          cout<<"Process P"<<j+1<<"\t"<<q[i].p[j].total_time<<"\t\t
"<<q[i].p[j].total_time-q[i].p[j].tt_time<<"\n";
          sum_tt+=q[i].p[j].total_time;
          sum_wt+=q[i].p[j].total_time-q[i].p[j].tt_time;
       }
    }

    cout<<"\n The average turnaround time is :
"<<sum_tt/no_of_processes<<endl;
    cout<<"\n The average waiting time is :
"<<sum_wt/no_of_processes<<endl;

}
```

Logic Of the Code

The code is an implementation of a multi-level feedback queue scheduler algorithm using a Round Robin algorithm for the highest priority queue.

The code starts by defining two structs, process and queues. The process struct contains the attributes of a process, including its priority, burst time, turnaround time, and total time. The queues struct contains the priority range for each queue, the total time taken by each queue, the length of each queue, a pointer to an array of processes, and a flag to indicate whether the queue has executed or not.

The notComplete function is used to check if any process is remaining in the queues or not. It iterates through all the queues and their respective processes and checks if any process still has a non-zero burst time. If a process is still remaining in any of the queues, the function returns true, indicating that the scheduler algorithm needs to continue.

The sort_ps function is used to sort the processes in a queue based on their priority. It uses a bubble sort algorithm to swap the processes' positions if their priorities are not in descending order.

The checkCompleteTimer function is used to keep track of the total time taken by each process and queue. It updates the total time for each process that has not yet completed and increases the total time for each queue for every unit of time.

The main function starts by taking user input for the number of processes, their priorities, and burst times. It creates an array of processes p1 and assigns each process to its respective queue based on its priority range.

The code then initializes the Round Robin algorithm by setting the timer to 0 and the time quantum to 4. It uses a while loop to execute the processes in each queue until all processes have completed.

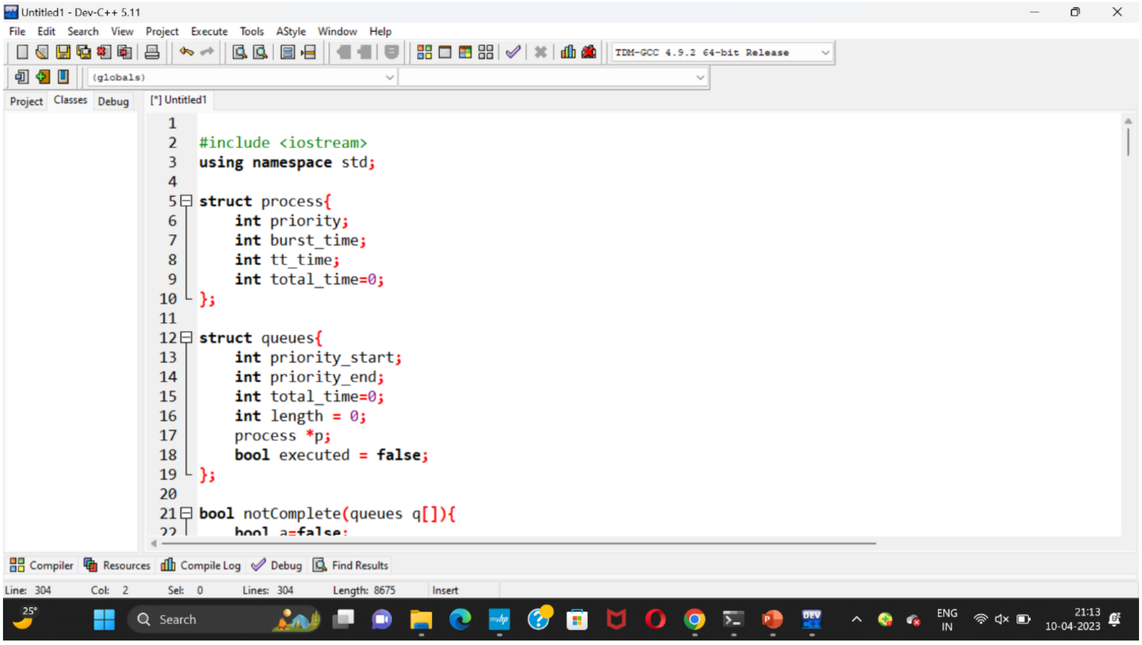
Inside the loop, it checks if the current queue has completed executing all its processes. If yes, it moves to the next queue. If not, it checks if the current queue is the highest priority queue, i.e., queue 1. If yes, it executes the Round Robin algorithm for a time quantum of 4. It executes each process in the queue until the time quantum has expired or the process has completed its burst time. If the time quantum expires, it moves to the next process in the queue. If the process has completed, it moves to the next process in the queue only if all other processes have not completed.

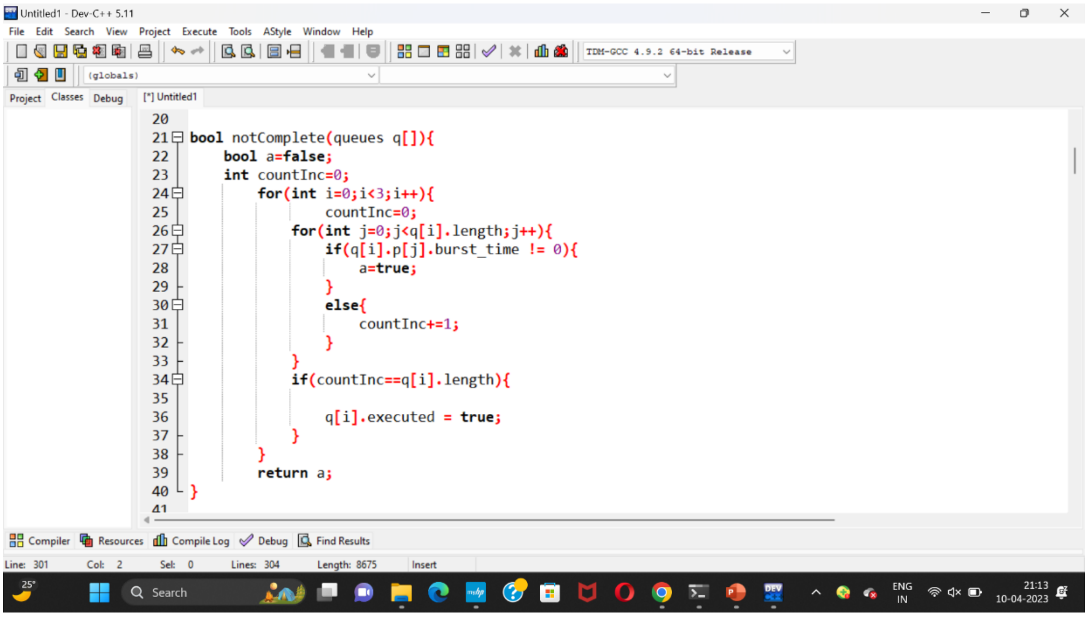
If the current queue is not the highest priority queue, it executes the processes in the queue in a First-Come-First-Serve (FCFS) order until all processes have completed.

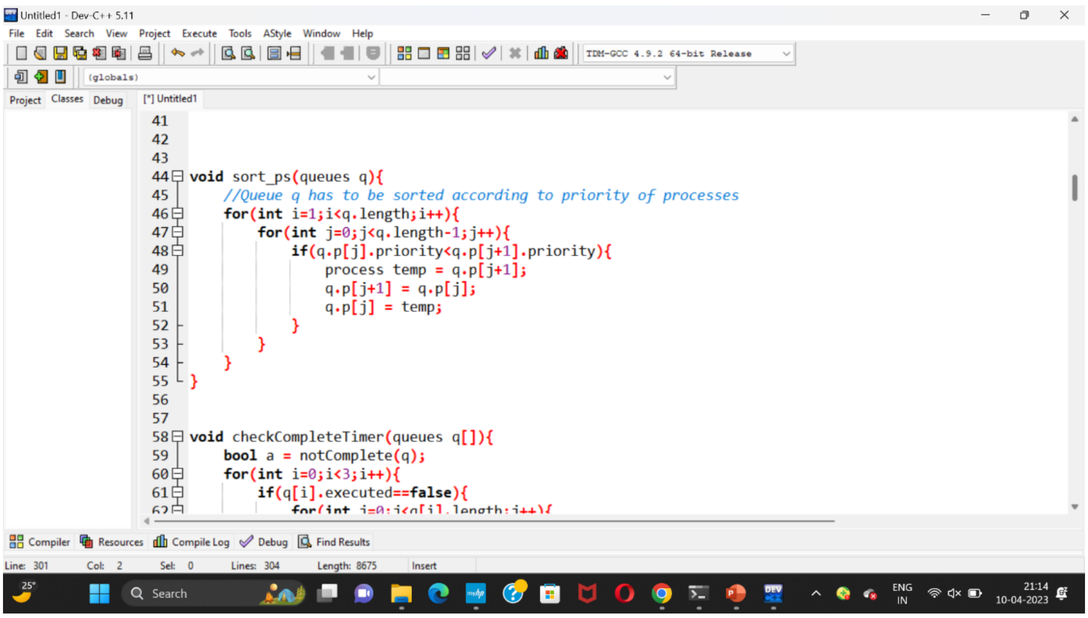
The code keeps track of the total time taken by each process and queue using the checkCompleteTimer function after each unit of time, and it prints the status of each process and queue after every execution. The code stops when all processes have completed executing.

In summary, the given code implements a multi-level feedback queue scheduler algorithm using a Round Robin algorithm for the highest priority queue and an FCFS algorithm for lower priority queues.

Screenshots







```
            if(q[i].executed==false){
                for(int j=0;j<q[i].length;j++){
                    if(q[i].p[j].burst_time!=0){
                        q[i].p[j].total_time+=1;
                    }
                }
                q[i].total_time+=1;
            }
        }
    }

main(){

    //Initializing 3 queues
    queues q[3];
    q[0].priority_start = 7;
    q[0].priority_end = 9;
    q[1].priority_start = 4;
    q[1].priority_end = 6;
    q[2].priority_start = 1;
    q[2].priority_end = 3;
```

```

    int no_of_processes,priority_of_process,burst_time_of_process;
    //Prompt User for entering Processes and assigning it to respective queues.
    cout<<"Enter the number of processes\n";
    cin>>no_of_processes;
    process p1[no_of_processes];

    for(int i=0;i<no_of_processes;i++){
        cout<<"Enter the priority of the process\n";
        cin>>priority_of_process;
        cout<<"Enter the burst time of the process\n";
        cin>>burst_time_of_process;
        p1[i].priority = priority_of_process;
        p1[i].burst_time = burst_time_of_process;
        p1[i].tt_time = burst_time_of_process;
        for(int j=0;j<3;j++){
        if(q[j].priority_start<=priority_of_process && priority_of_process<=q[j].priority_end){
            q[j].length++;
        }
        }
    }
```

```
    }

    for(int i =0;i<3;i++){
        int len = q[i].length;
        q[i].p = new process[len];
    }


    int a=0;
    int b=0;
    int c=0;

    for(int i =0;i<3;i++){
        for(int j=0;j<no_of_processes;j++){
            if((q[i].priority_start<=p1[j].priority) && (p1[j].priority<=q[i].priority_end)){
                if(i==0){
                    q[i].p[a++] = p1[j];

                }
                else if(i==1){
                    q[i].p[b++] = p1[j];
```

```
                q[i].p[b++] = p1[j];
                }
            else{
                q[i].p[c++] = p1[j];
                }
            }
        }
    }

    a--;b--;c--;
    for(int i=0;i<3;i++){
        cout<<"Queue "<<i+1<<" : \t";
        for(int j=0;j<q[i].length;j++){
            cout<<q[i].p[j].priority<<"->";
        }
        cout<<"NULL\n";
    }


    //While RR on multiple queues is not complete, keep on repeating
    int timer = 0;
    int l =-1;
```

```
    int timer = 0;
    int l =-1;
    int rr_timer = 4;
    int counter=0;
    int counterps=0;
    int counterfcfs=0;
    while(notComplete(q)){
        if(timer == 10){
            timer = 0;
        }
        l+=1;
        if(l>=3){
            l=l%3;
        }

        //Process lth queue if its already not executed
        //If its executed change the value of l
        if(q[l].executed == true){
            cout<<"Queue "<<l+1<<" completed\n";
            l+=1;
            if(l>=3){
                l=l%3;
```

```
            l+=1;
            if(l>=3){
                l=l%3;
            }
            continue;
        }

        //Finally you now have a queue which is not completely executed
        //Process the incomplete processes over it

        if(l==0){
            cout<<"Queue "<<l+1<<" in hand\n";
            //Round Robin Algorithm for q=4
            if(rr_timer == 0){
                rr_timer = 4;
            }

            for(int i=0;i<q[l].length;i++){
                if(q[l].p[i].burst_time==0){
                    counter++;
                    continue;
                }
```

```
            }
            if(counter == q[l].length){
                break;
            }
            while(rr_timer>0 && q[l].p[i].burst_time!=0 && timer!=10){
                cout<<"Executing queue 1 and "<<i+1<<" process for a unit time. Process has priority
                q[l].p[i].burst_time--;
                checkCompleteTimer(q);
                rr_timer--;
                timer++;

            }
            if(timer == 10){
                break;
            }
            if(q[l].p[i].burst_time==0 && rr_timer ==0){
                rr_timer = 4;
                if(i == (q[i].length-1)){
                    i=-1;
                }
                continue;
            }
```

```
            }
            if(q[l].p[i].burst_time==0 && rr_timer > 0){
                if(i == (q[i].length-1)){
                    i=-1;
                }
                continue;
            }
            if(rr_timer <= 0){
                rr_timer = 4;
                if(i == (q[i].length-1)){
                    i=-1;
                }
                continue;
            }

        }
    }


    else if(l==1){
        cout<<"Queue "<<l+1<<" in hand\n";
        sort_ps(q[l]);
```

```
        cout<<"Queue "<<l+1<<" in hand\n";
        sort_ps(q[l]);
        //Priority Scheduling
        for(int i=0;i<q[l].length;i++){
            if(q[l].p[i].burst_time==0){
                counterps++;
                continue;
            }
            if(counterps == q[l].length){
                break;
            }
            while(q[l].p[i].burst_time!=0 && timer!=10){
                cout<<"Executing queue 2 and "<<i+1<<" process for a unit time. Process has priority
                q[l].p[i].burst_time--;
                checkCompleteTimer(q);
                timer++;

            }
            if(timer == 10){
                break;
            }
            if(q[l].p[i].burst_time==0){
```

```cpp
                        continue;
                    }

                }
            }
            else{
                cout<<"Queue "<<l+1<<" in hand\n";
                //FCFS
                for(int i=0;i<q[l].length;i++){
                    if(q[l].p[i].burst_time==0){
                        counterfcfs++;
                        continue;
                    }
                    if(counterfcfs == q[l].length){
                        break;
                    }
                    while(q[l].p[i].burst_time!=0 && timer!=10){
                        cout<<"Executing queue 3 and "<<i+1<<" process for a unit time. Process has priority
                        q[l].p[i].burst_time--;
                        checkCompleteTimer(q);

                        timer++;
```

```cpp
                    }
                    if(timer == 10){
                        break;
                    }
                    if(q[l].p[i].burst_time==0){
                            continue;
                    }

                }

            }
            cout<<"Broke from queue "<<l+1<<"\n";
        }

        for(int i=0;i<3;i++){
            cout<<"\nTime taken for queue "<<i+1<<" to execute: "<<q[i].total_time<<"\n";
            for(int j=0;j<q[i].length;j++){
                cout<<"Process "<<j+1<<" of queue "<<i+1<<" took "<<q[i].p[j].total_time<<"\n";
            }
        }

        int sum_tt=0;
```

```cpp
                cout<<"Process "<<j+1<<" of queue "<<i+1<<" took "<<q[i].p[j].total_time<<"\n";
            }
        }

        int sum_tt=0;
        int sum_wt=0;

        cout<<"\n\nProcess      | Turn Around Time | Waiting Time\n";
        for(int i=0;i<3;i++){
                cout<<"Queue "<<i+1<<"\n";
            for(int j=0;j<q[i].length;j++){
                cout<<"Process P"<<j+1<<"\t"<<q[i].p[j].total_time<<"\t\t     "<<q[i].p[j].total_time-q[i].p[
                sum_tt+=q[i].p[j].total_time;
                sum_wt+=q[i].p[j].total_time-q[i].p[j].tt_time;
            }
        }

        cout<<"\n The average turnaround time is : "<<sum_tt/no_of_processes<<endl;
        cout<<"\n The average waiting time is : "<<sum_wt/no_of_processes<<endl;

}
```

C:\Users\hp\OneDrive\Docum

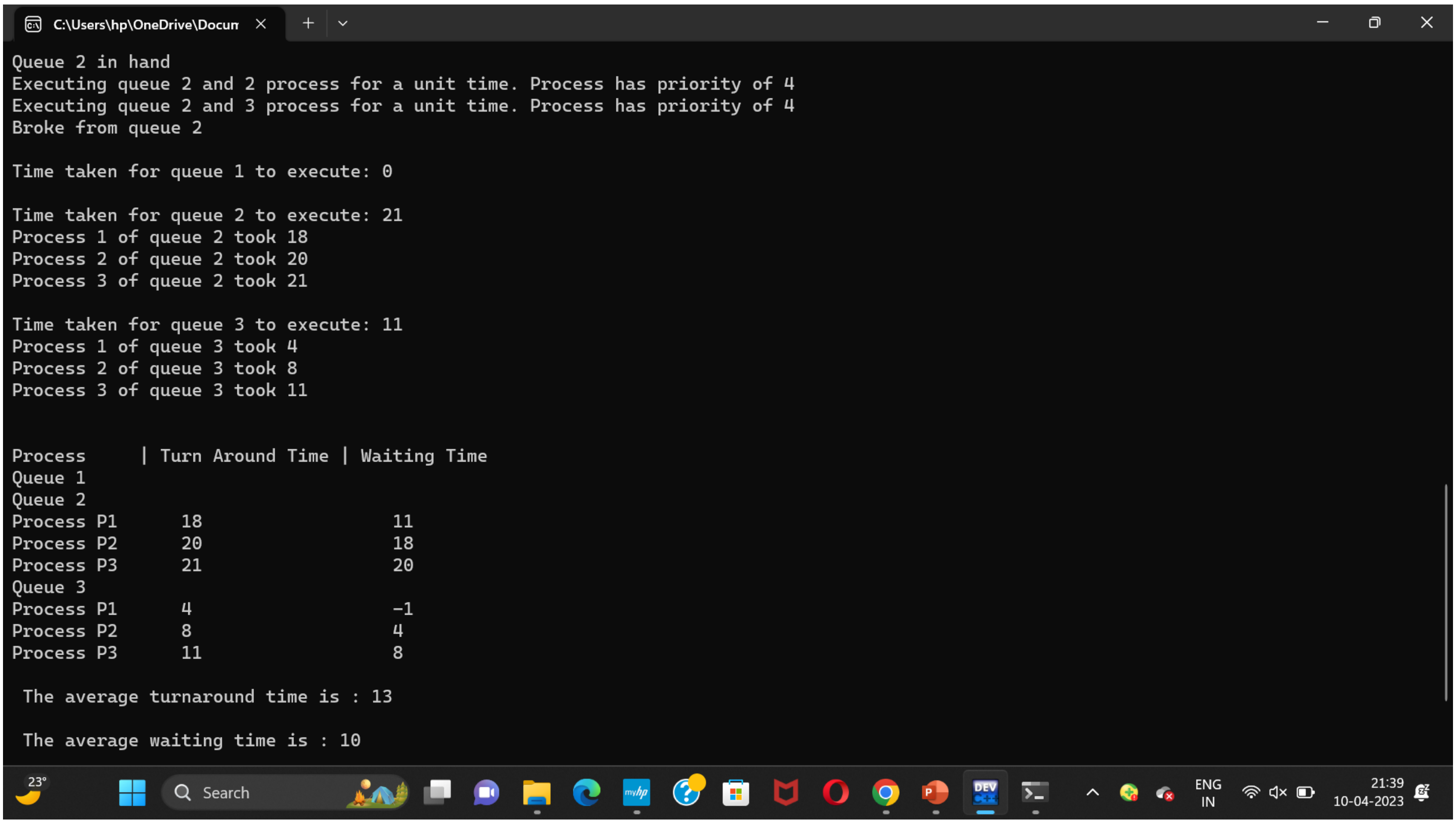
```
Enter the number of processes
6
Enter the priority of the process
4
Enter the burst time of the process
2
Enter the priority of the process
4
Enter the burst time of the process
1
Enter the priority of the process
5
Enter the burst time of the process
7
Enter the priority of the process
3
Enter the burst time of the process
5
Enter the priority of the process
2
Enter the burst time of the process
4
Enter the priority of the process
1
Enter the burst time of the process
3
Queue 1 :       NULL
Queue 2 :       4->4->5->NULL
Queue 3 :       3->2->1->NULL
Queue 1 completed
Queue 3 in hand
Executing queue 3 and 1 process for a unit time. Process has priority of 3
```

23° Search ENG IN 21:39 10-04-2023

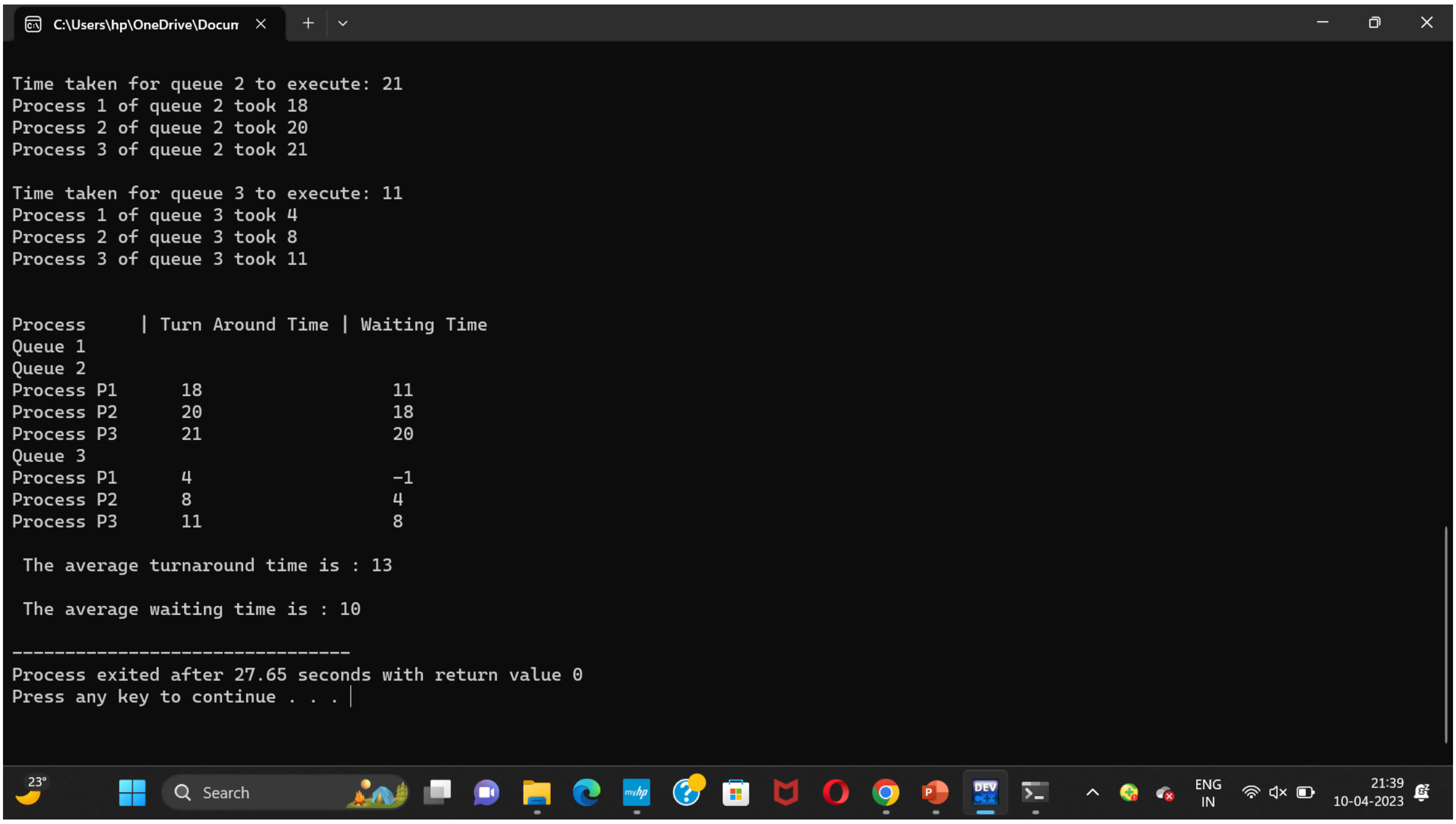
C:\Users\hp\OneDrive\Docum

```
Queue 3 in hand
Executing queue 3 and 1 process for a unit time. Process has priority of 3
Executing queue 3 and 1 process for a unit time. Process has priority of 3
Executing queue 3 and 1 process for a unit time. Process has priority of 3
Executing queue 3 and 1 process for a unit time. Process has priority of 3
Executing queue 3 and 1 process for a unit time. Process has priority of 3
Executing queue 3 and 2 process for a unit time. Process has priority of 2
Executing queue 3 and 2 process for a unit time. Process has priority of 2
Executing queue 3 and 2 process for a unit time. Process has priority of 2
Executing queue 3 and 2 process for a unit time. Process has priority of 2
Executing queue 3 and 3 process for a unit time. Process has priority of 1
Broke from queue 3
Queue 1 completed
Queue 3 in hand
Executing queue 3 and 3 process for a unit time. Process has priority of 1
Executing queue 3 and 3 process for a unit time. Process has priority of 1
Broke from queue 3
Queue 1 completed
Queue 3 completed
Queue 2 in hand
Executing queue 2 and 1 process for a unit time. Process has priority of 5
Executing queue 2 and 1 process for a unit time. Process has priority of 5
Executing queue 2 and 1 process for a unit time. Process has priority of 5
Executing queue 2 and 1 process for a unit time. Process has priority of 5
Executing queue 2 and 1 process for a unit time. Process has priority of 5
Executing queue 2 and 1 process for a unit time. Process has priority of 5
Executing queue 2 and 1 process for a unit time. Process has priority of 5
Executing queue 2 and 2 process for a unit time. Process has priority of 4
Broke from queue 2
Queue 3 completed
Queue 2 in hand
Executing queue 2 and 2 process for a unit time. Process has priority of 4
```

23° Search ENG IN 21:39 10-04-2023



```
Queue 2 in hand
Executing queue 2 and 2 process for a unit time. Process has priority of 4
Executing queue 2 and 3 process for a unit time. Process has priority of 4
Broke from queue 2

Time taken for queue 1 to execute: 0

Time taken for queue 2 to execute: 21
Process 1 of queue 2 took 18
Process 2 of queue 2 took 20
Process 3 of queue 2 took 21

Time taken for queue 3 to execute: 11
Process 1 of queue 3 took 4
Process 2 of queue 3 took 8
Process 3 of queue 3 took 11


Process      | Turn Around Time | Waiting Time
Queue 1
Queue 2
Process P1      18                   11
Process P2      20                   18
Process P3      21                   20
Queue 3
Process P1      4                    -1
Process P2      8                    4
Process P3      11                   8

 The average turnaround time is : 13

 The average waiting time is : 10
```



```
Time taken for queue 2 to execute: 21
Process 1 of queue 2 took 18
Process 2 of queue 2 took 20
Process 3 of queue 2 took 21

Time taken for queue 3 to execute: 11
Process 1 of queue 3 took 4
Process 2 of queue 3 took 8
Process 3 of queue 3 took 11


Process      | Turn Around Time | Waiting Time
Queue 1
Queue 2
Process P1      18                   11
Process P2      20                   18
Process P3      21                   20
Queue 3
Process P1      4                    -1
Process P2      8                    4
Process P3      11                   8

 The average turnaround time is : 13

 The average waiting time is : 10

--------------------------------
Process exited after 27.65 seconds with return value 0
Press any key to continue . . .
```