Assignment 08

SCS1214 – Operating Systems

MULTILEVEL QUEUE SCHEDULING

Ravija Salpitikorala – 21001685
2021/CS/168

Introduction

In this application to implement a Multilevel Queue Scheduling, we need to create 4 queues.

- q0 – 0th queue
- q1 – 1st queue
- q2 – 2nd queue
- q3 – 3rd queue

Every queue has different priorities where 0th queue has the highest priority, and the 3rd queue has the least priority and the queues in the middle has the 2nd most and the 3rd most priorities.

Each queue has its own scheduling algorithm,

- q0 – Round Robin
- q1 – Shortest Job First
- q2 - Shortest Job First
- q3 – First Come First Serve

Each queue has a 20 second time quantum where each of them must preempt and give the CPU to the processes in the next queue after every 20 seconds whether or not they have finished its process.
If the process which was running is not finished its execution, it is pushed back of its queue.

When user enters the priority and the burst time of the process it outputs the turnaround time and the waiting time of each process separately.

I implemented this program using the C++ programming language.

Note:

In this implementation the arrival time of all the processes are considered as zero.
Because we can't input processes when the execution of the program starts.
User have to enter the number of processes at the beginning of the execution.

Methodology & Program Implementation

- **Creating the struct type to store the details.**

When implementing this program, the first step is to make a struct type to store the details in variables, which we are going to use when implementing the rest of the program.

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

#define quantum 20          //Switching time between the queues
#define q_time 4            //Quantum time of the RR queue

typedef struct Process{
    int pid;                //Process ID
    int priority;           //Priority ot the Process
    int b_time;             //Burst time of the Process
    int burst_time;         //Burst time of the Process at the initial stage
    int waiting_time;       //waitinf time of the Process
    int turnaround_time;    //Completion time of the Process
}Process;
```

Also, I defined the quantum time of the queues at the beginning of the program, so I don't have to define them again and again when implementing the scheduling algorithms.

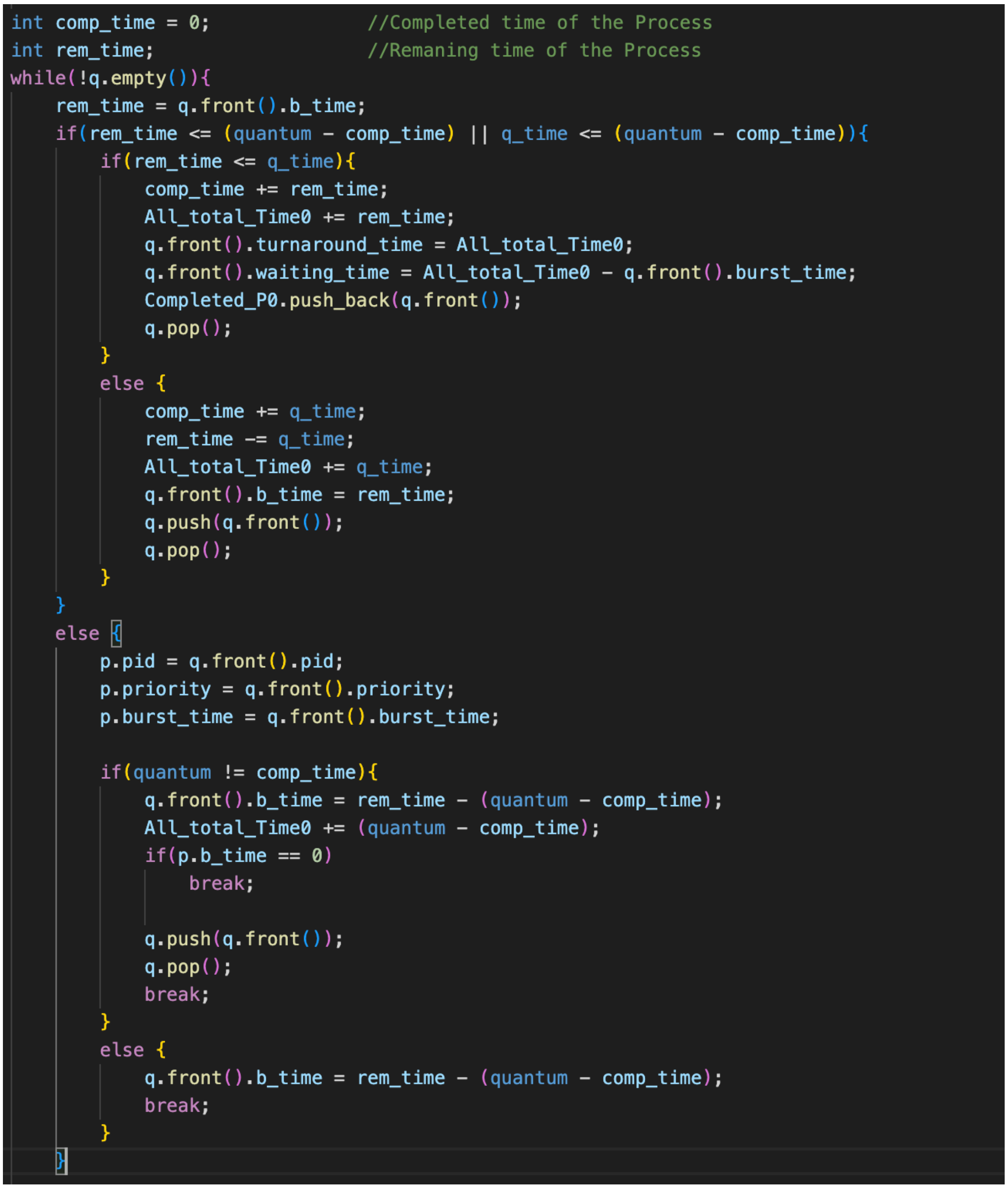
Following that, I made four vectors for each queue to contain the processes that had successfully completed, as well as a static variable to increment the time of the overall scheduling algorithm.

```
//Creating vectors to store the completed Processes
vector<Process> Completed_P0;
vector<Process> Completed_P1;
vector<Process> Completed_P2;
vector<Process> Completed_P3;

static int All_total_Time0 = 0;     //Time from the beginning of the queue scheduling to the end
```

- **Implementing the queues.**

1. Round Robin (RR)

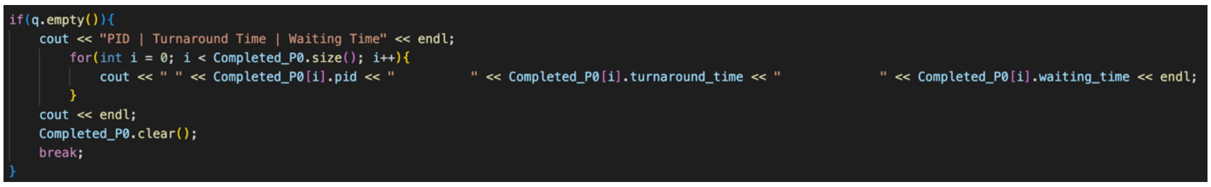
   In the Round Robin queue implementation, we define a quantum time for the queue which helps to switch between the processes (whether or not they are finished) in the queue after a certain time interval.



```
int comp_time = 0;              //Completed time of the Process
int rem_time;                   //Remaning time of the Process
while(!q.empty()){
    rem_time = q.front().b_time;
    if(rem_time <= (quantum - comp_time) || q_time <= (quantum - comp_time)){
        if(rem_time <= q_time){
            comp_time += rem_time;
            All_total_Time0 += rem_time;
            q.front().turnaround_time = All_total_Time0;
            q.front().waiting_time = All_total_Time0 - q.front().burst_time;
            Completed_P0.push_back(q.front());
            q.pop();
        }
        else {
            comp_time += q_time;
            rem_time -= q_time;
            All_total_Time0 += q_time;
            q.front().b_time = rem_time;
            q.push(q.front());
            q.pop();
        }
    }
    else {
        p.pid = q.front().pid;
        p.priority = q.front().priority;
        p.burst_time = q.front().burst_time;

        if(quantum != comp_time){
            q.front().b_time = rem_time - (quantum - comp_time);
            All_total_Time0 += (quantum - comp_time);
            if(p.b_time == 0)
                break;

            q.push(q.front());
            q.pop();
            break;
        }
        else {
            q.front().b_time = rem_time - (quantum - comp_time);
            break;
        }
    }
```

Here, conditions checks whether the burst time of the process can execute in this time iteration or not before the program switches to the next queue. And if can, above 'if' part of the code is executed and the process is pushed into the vector.
If there is not enough time left in the queue to execute the burst time of the process (before switching to the next queue) else part of the code is executed and breaks from the queue

Once the process is about to end the process execution , its burst time should be less than or equal to its Round Robing quantum Time. Hence using that condition I, Implemented a way to update *Turnaround Time* and *Waiting Time.*



```
if(q.empty()){
    cout << "PID | Turnaround Time | Waiting Time" << endl;
        for(int i = 0; i < Completed_P0.size(); i++){
            cout << " " << Completed_P0[i].pid << "          " << Completed_P0[i].turnaround_time << "             " << Completed_P0[i].waiting_time << endl;
        }
    cout << endl;
    Completed_P0.clear();
    break;
}
```

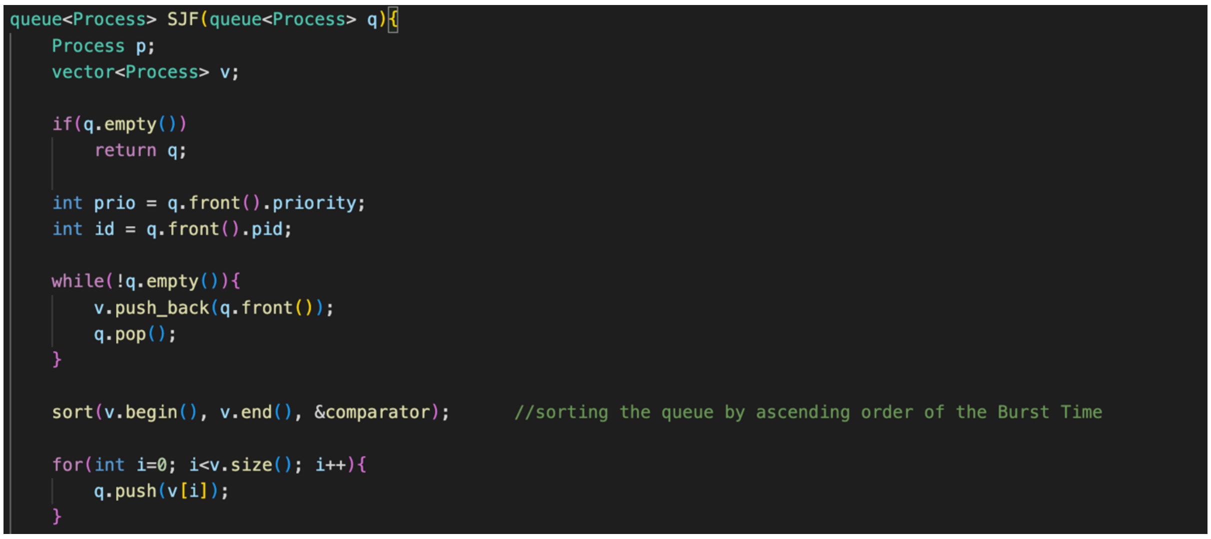
When all the processes in the queue have completed their execution, the *PID, Turnaround Time,* and *Waiting Time* of those processes are given as the output.

2. Shortest Job First (SJF)

   In the SJF queue implementation, we sort the queue by the ascending order of the burst time of the processes and treat them as FCFS queue implementation (Processes at the beginning of the queue are executed first).

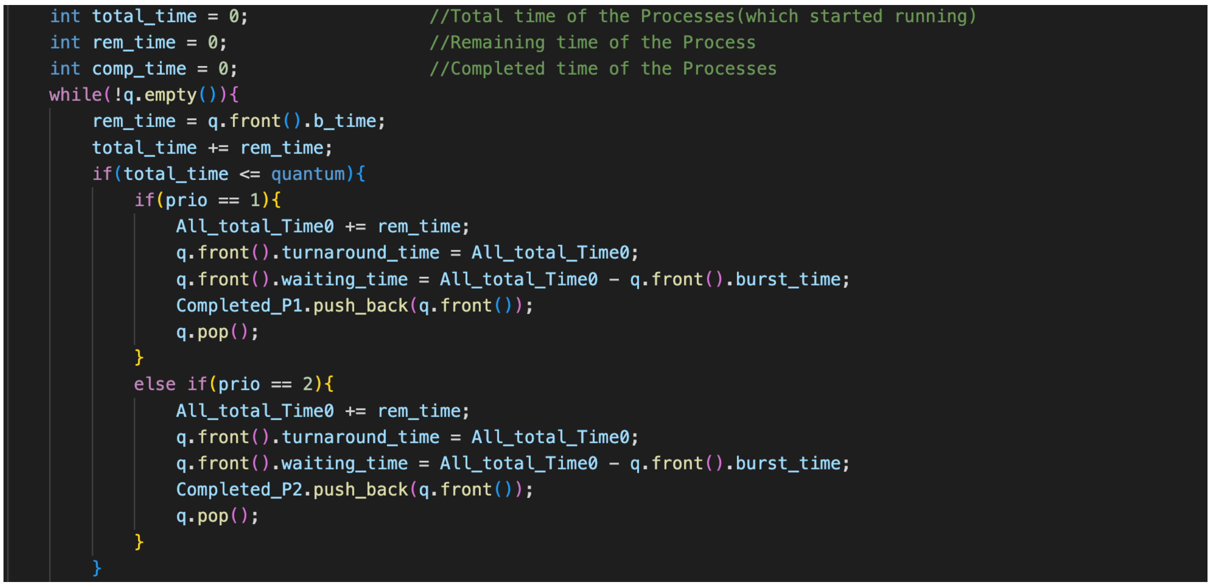
   - We are only able to do that because all processes are added to the queues prior to the scheduling algorithm starts its execution.

   Due to the existence of two SJF queues, we must go forward with the execution concerning the priority of the queues.



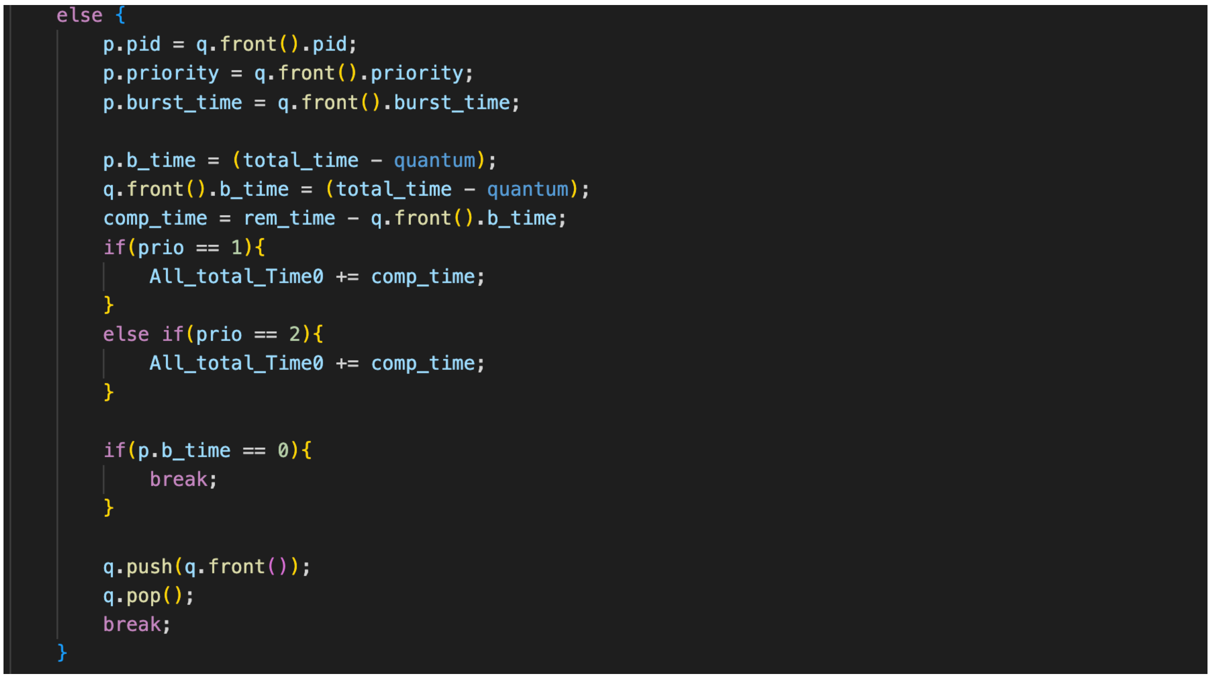
```
queue<Process> SJF(queue<Process> q){
    Process p;
    vector<Process> v;

    if(q.empty())
        return q;

    int prio = q.front().priority;
    int id = q.front().pid;

    while(!q.empty()){
        v.push_back(q.front());
        q.pop();
    }

    sort(v.begin(), v.end(), &comparator);      //sorting the queue by ascending order of the Burst Time

    for(int i=0; i<v.size(); i++){
        q.push(v[i]);
    }
```

   This part is where we sort the queue by the ascending order of the burst time of the processes.



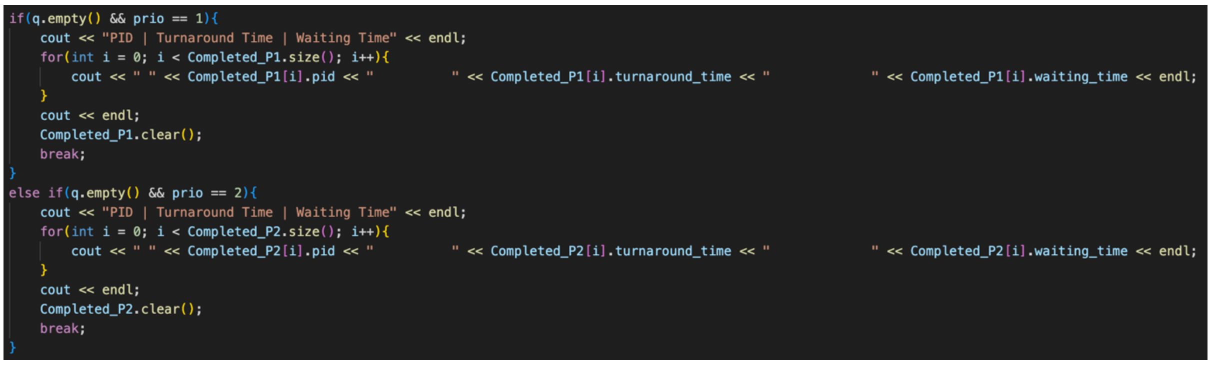
```
    int total_time = 0;                  //Total time of the Processes(which started running)
    int rem_time = 0;                    //Remaining time of the Process
    int comp_time = 0;                   //Completed time of the Processes
    while(!q.empty()){
        rem_time = q.front().b_time;
        total_time += rem_time;
        if(total_time <= quantum){
            if(prio == 1){
                All_total_Time0 += rem_time;
                q.front().turnaround_time = All_total_Time0;
                q.front().waiting_time = All_total_Time0 - q.front().burst_time;
                Completed_P1.push_back(q.front());
                q.pop();
            }
            else if(prio == 2){
                All_total_Time0 += rem_time;
                q.front().turnaround_time = All_total_Time0;
                q.front().waiting_time = All_total_Time0 - q.front().burst_time;
                Completed_P2.push_back(q.front());
                q.pop();
            }
        }
```

   If the completed time of the processes in this queue and the burst time of the next process is less that the quantum time we proceed with the above part of the code.



```
    else {
        p.pid = q.front().pid;
        p.priority = q.front().priority;
        p.burst_time = q.front().burst_time;

        p.b_time = (total_time - quantum);
        q.front().b_time = (total_time - quantum);
        comp_time = rem_time - q.front().b_time;
        if(prio == 1){
            All_total_Time0 += comp_time;
        }
        else if(prio == 2){
            All_total_Time0 += comp_time;
        }

        if(p.b_time == 0){
            break;
        }

        q.push(q.front());
        q.pop();
        break;
    }
```

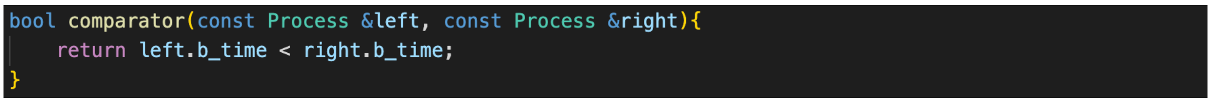
If the completed time of the processes in this queue and the burst time of the next process is less that the quantum time we proceed with the above else part of the code



```
if(q.empty() && prio == 1){
    cout << "PID | Turnaround Time | Waiting Time" << endl;
    for(int i = 0; i < Completed_P1.size(); i++){
        cout << " " << Completed_P1[i].pid << "          " << Completed_P1[i].turnaround_time << "             " << Completed_P1[i].waiting_time << endl;
    }
    cout << endl;
    Completed_P1.clear();
    break;
}
else if(q.empty() && prio == 2){
    cout << "PID | Turnaround Time | Waiting Time" << endl;
    for(int i = 0; i < Completed_P2.size(); i++){
        cout << " " << Completed_P2[i].pid << "          " << Completed_P2[i].turnaround_time << "             " << Completed_P2[i].waiting_time << endl;
    }
    cout << endl;
    Completed_P2.clear();
    break;
}
```

When all the processes in the queue have completed their execution, the *PID, Turnaround Time,* and *Waiting Time* of those processes are given as the output.
If the priority is 1, we give the output of the SJF1 queue else, we give the output of the SJF2 queue.

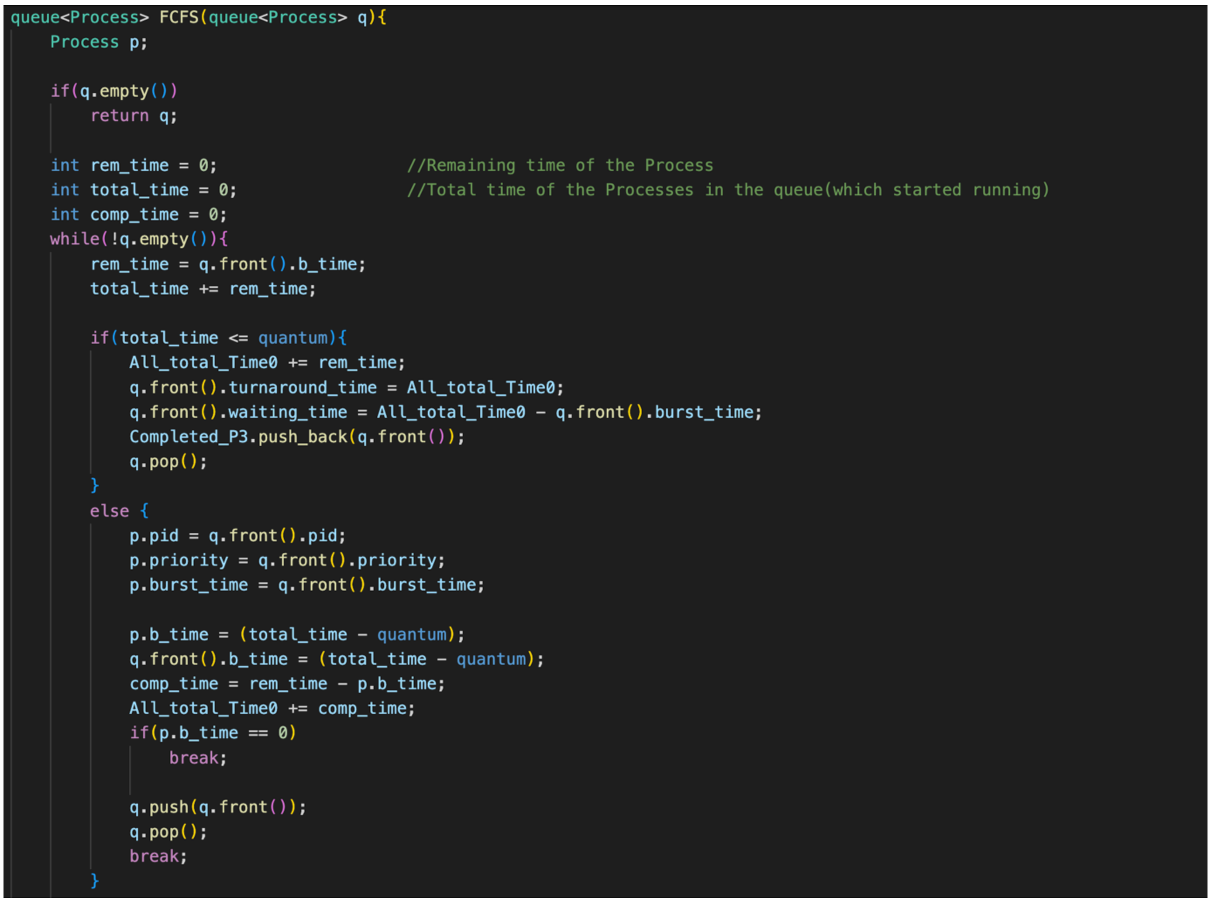
I used the following function to sort the vector of structs(process) by a variable within the struct.



```
bool comparator(const Process &left, const Process &right){
    return left.b_time < right.b_time;
}
```

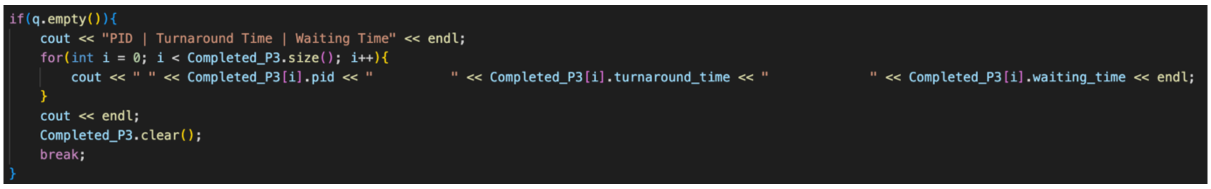
3. First Come First Serve (FCFS)

   In FCFS queue implementation, we execute the process which is in front of the queue. This is decided at the initial stage, which is when we insert the processes into the queue.



```
queue<Process> FCFS(queue<Process> q){
    Process p;

    if(q.empty())
        return q;

    int rem_time = 0;                    //Remaining time of the Process
    int total_time = 0;                  //Total time of the Processes in the queue(which started running)
    int comp_time = 0;
    while(!q.empty()){
        rem_time = q.front().b_time;
        total_time += rem_time;

        if(total_time <= quantum){
            All_total_Time0 += rem_time;
            q.front().turnaround_time = All_total_Time0;
            q.front().waiting_time = All_total_Time0 - q.front().burst_time;
            Completed_P3.push_back(q.front());
            q.pop();
        }
        else {
            p.pid = q.front().pid;
            p.priority = q.front().priority;
            p.burst_time = q.front().burst_time;

            p.b_time = (total_time - quantum);
            q.front().b_time = (total_time - quantum);
            comp_time = rem_time - p.b_time;
            All_total_Time0 += comp_time;
            if(p.b_time == 0)
                break;

            q.push(q.front());
            q.pop();
            break;
        }
```

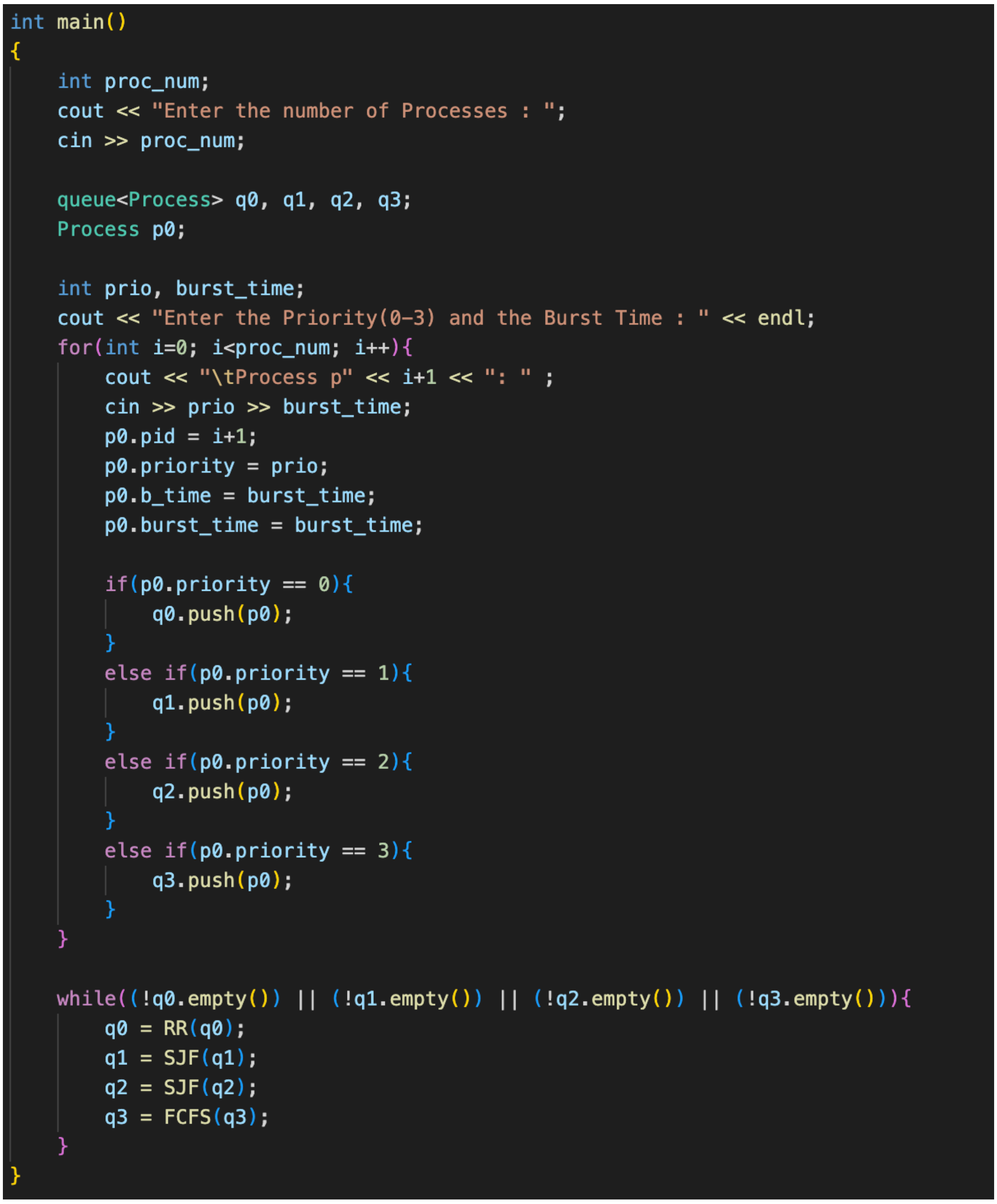
As same as in the SJF implementation, if the completed time of the processes in this queue and the burst time of the next process is less that the quantum time we proceed with the 'if' part of the code and that time is greater than the quantum time, we proceed with the 'else' part of the code.



```
if(q.empty()){
    cout << "PID | Turnaround Time | Waiting Time" << endl;
    for(int i = 0; i < Completed_P3.size(); i++){
        cout << " " << Completed_P3[i].pid << "          " << Completed_P3[i].turnaround_time << "             " << Completed_P3[i].waiting_time << endl;
    }
    cout << endl;
    Completed_P3.clear();
    break;
}
```

When all the processes in the queue have completed their execution, the *PID, Turnaround Time,* and *Waiting Time* of those processes are given as the output.

- **Creating the main() function of the program.**

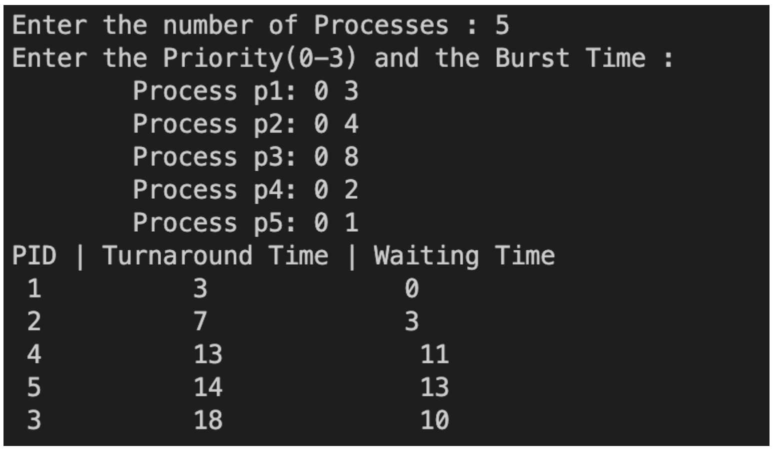


```cpp
int main()
{
    int proc_num;
    cout << "Enter the number of Processes : ";
    cin >> proc_num;

    queue<Process> q0, q1, q2, q3;
    Process p0;

    int prio, burst_time;
    cout << "Enter the Priority(0-3) and the Burst Time : " << endl;
    for(int i=0; i<proc_num; i++){
        cout << "\tProcess p" << i+1 << ": " ;
        cin >> prio >> burst_time;
        p0.pid = i+1;
        p0.priority = prio;
        p0.b_time = burst_time;
        p0.burst_time = burst_time;

        if(p0.priority == 0){
            q0.push(p0);
        }
        else if(p0.priority == 1){
            q1.push(p0);
        }
        else if(p0.priority == 2){
            q2.push(p0);
        }
        else if(p0.priority == 3){
            q3.push(p0);
        }
    }

    while((!q0.empty()) || (!q1.empty()) || (!q2.empty()) || (!q3.empty())){
        q0 = RR(q0);
        q1 = SJF(q1);
        q2 = SJF(q2);
        q3 = FCFS(q3);
    }
}
```

Here, we create four processes and ask the user to input the number of processes, priority, and the burst time of each process.
The according to the priority of each process we insert the processes into the queues.
Then we execute the above program according to the priorities of the queues (after the preemption of the q0, the 2nd most prioritized queue which is q1 is executed) until all the processes have completed their execution and all the queues are empty.

Results of the Program

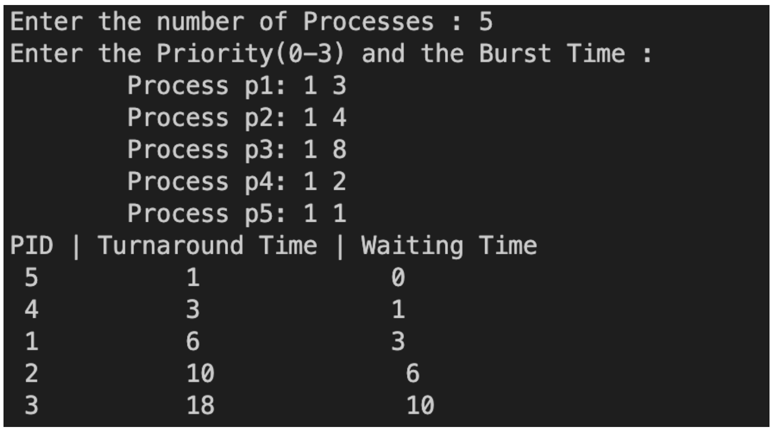
- **First let's consider each queue individually where all processes burst time < 20 sec.**
    - Round Robin Queue (q0)



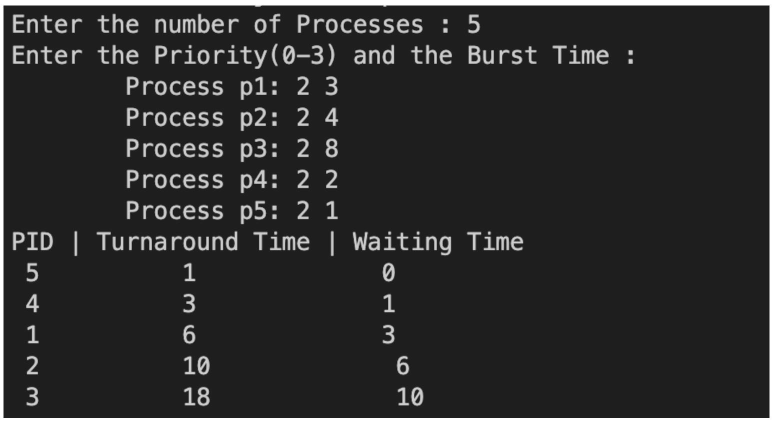
```
Enter the number of Processes : 5
Enter the Priority(0-3) and the Burst Time :
        Process p1: 0 3
        Process p2: 0 4
        Process p3: 0 8
        Process p4: 0 2
        Process p5: 0 1
PID | Turnaround Time | Waiting Time
 1          3            0
 2          7            3
 4          13             11
 5          14             13
 3          18             10
```

Average Turnaround Time = 11 sec
Average Waiting Time = 7.4 sec

- Shortest Job First Queues (q1 / q2)



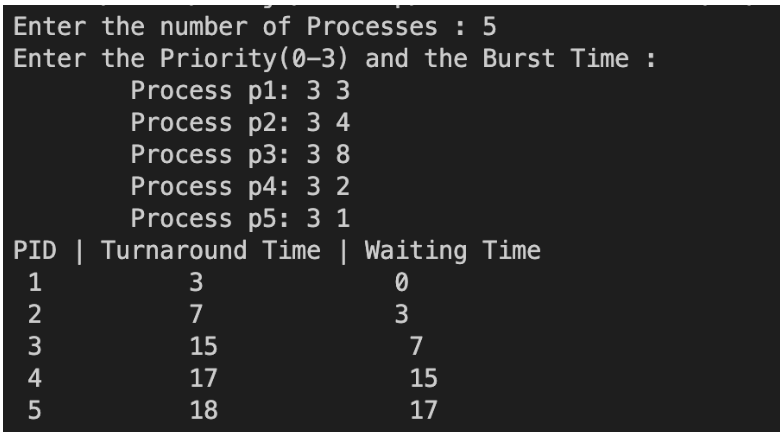
```
Enter the number of Processes : 5
Enter the Priority(0-3) and the Burst Time :
        Process p1: 1 3
        Process p2: 1 4
        Process p3: 1 8
        Process p4: 1 2
        Process p5: 1 1
PID | Turnaround Time | Waiting Time
 5          1            0
 4          3            1
 1          6            3
 2          10             6
 3          18             10
```



```
Enter the number of Processes : 5
Enter the Priority(0-3) and the Burst Time :
        Process p1: 2 3
        Process p2: 2 4
        Process p3: 2 8
        Process p4: 2 2
        Process p5: 2 1
PID | Turnaround Time | Waiting Time
 5          1            0
 4          3            1
 1          6            3
 2          10             6
 3          18             10
```

Average Turnaround Time = 7.6 sec
Average Waiting Time = 4 sec

- First Come First Serve Queue (q3)



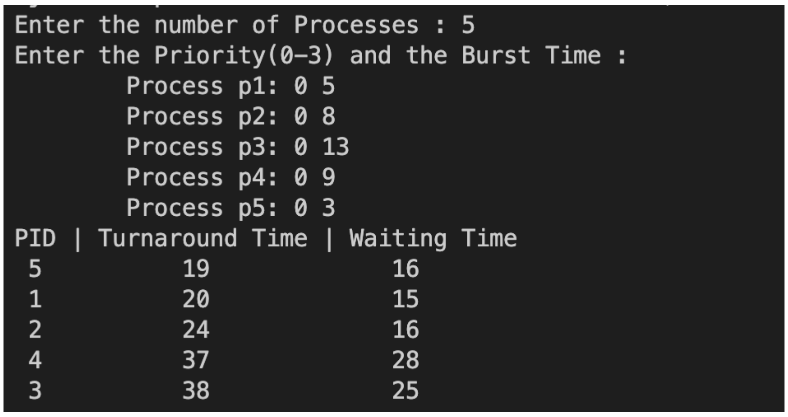
```
Enter the number of Processes : 5
Enter the Priority(0-3) and the Burst Time :
        Process p1: 3 3
        Process p2: 3 4
        Process p3: 3 8
        Process p4: 3 2
        Process p5: 3 1
PID | Turnaround Time | Waiting Time
 1          3           0
 2          7           3
 3          15           7
 4          17           15
 5          18           17
```

Average Turnaround Time = 12 sec
Average Waiting Time = 8.4 sec

According to these statistics, in SJF scheduling queue, the average turnaround time and the average waiting time is lesser than both the other queues.
Also, the average turnaround and waiting times of the RR queue is lesser than the times of FCFS queue.

- **Then consider each queue individually where all processes burst time > 20 sec.**

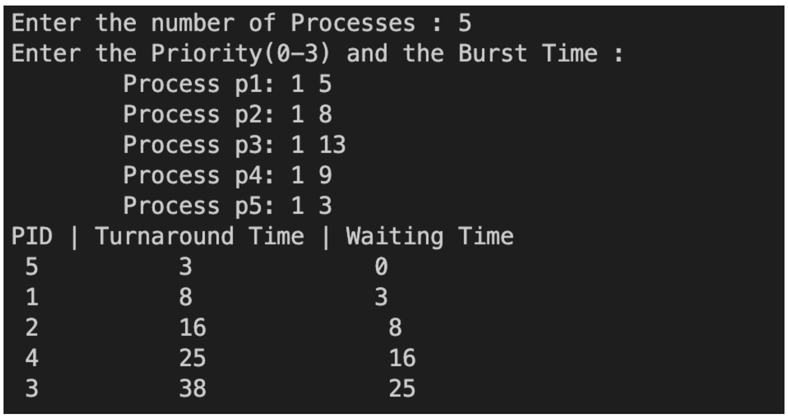
  - Round Robin Queue



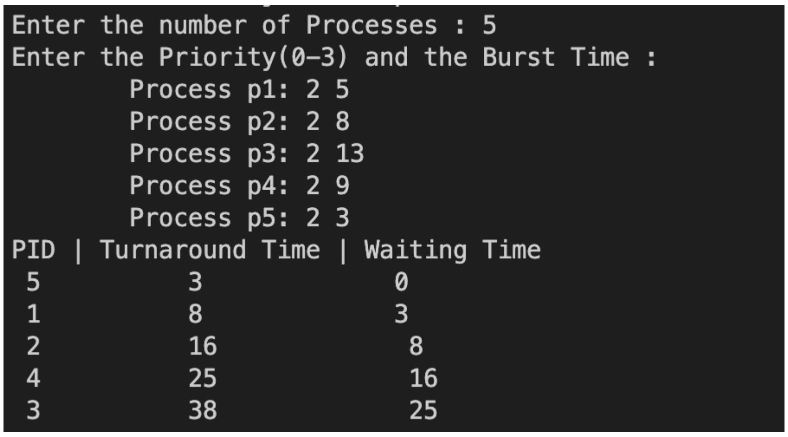
```
Enter the number of Processes : 5
Enter the Priority(0-3) and the Burst Time :
        Process p1: 0 5
        Process p2: 0 8
        Process p3: 0 13
        Process p4: 0 9
        Process p5: 0 3
PID | Turnaround Time | Waiting Time
 5          19             16
 1          20             15
 2          24             16
 4          37             28
 3          38             25
```

Average Turnaround Time = 27.6 sec
Average Waiting Time = 20 sec

- Shortest Job First Queues



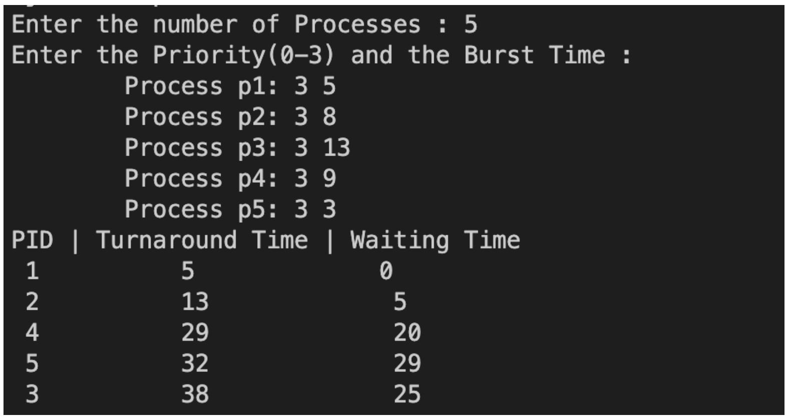
```
Enter the number of Processes : 5
Enter the Priority(0-3) and the Burst Time :
        Process p1: 1 5
        Process p2: 1 8
        Process p3: 1 13
        Process p4: 1 9
        Process p5: 1 3
PID | Turnaround Time | Waiting Time
 5         3             0
 1         8             3
 2         16             8
 4         25             16
 3         38             25
```



```
Enter the number of Processes : 5
Enter the Priority(0-3) and the Burst Time :
        Process p1: 2 5
        Process p2: 2 8
        Process p3: 2 13
        Process p4: 2 9
        Process p5: 2 3
PID | Turnaround Time | Waiting Time
 5         3             0
 1         8             3
 2         16             8
 4         25             16
 3         38             25
```

Average Turnaround Time = 18 sec
Average Waiting Time = 10.4 sec

- First Come First Serve Queue



```
Enter the number of Processes : 5
Enter the Priority(0-3) and the Burst Time :
        Process p1: 3 5
        Process p2: 3 8
        Process p3: 3 13
        Process p4: 3 9
        Process p5: 3 3
PID | Turnaround Time | Waiting Time
 1         5             0
 2         13             5
 4         29             20
 5         32             29
 3         38             25
```

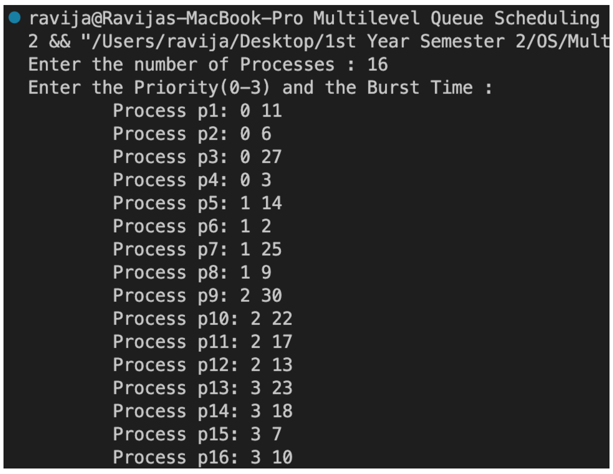
Average Turnaround Time = 23.4 sec
Average Waiting Time = 15.8 sec

In above statistics, in SJF scheduling queue, the average turnaround and waiting times are lesser than the average turnaround and waiting in both the other queues.
Even though the average turnaround and the average waiting times in RR queue is high, it allows all the processes to switch after a 4 second time quantum which means the starvation of the processes are avoided.

- **Inserting processes for all four queues at the same time**
    - Test Case 01

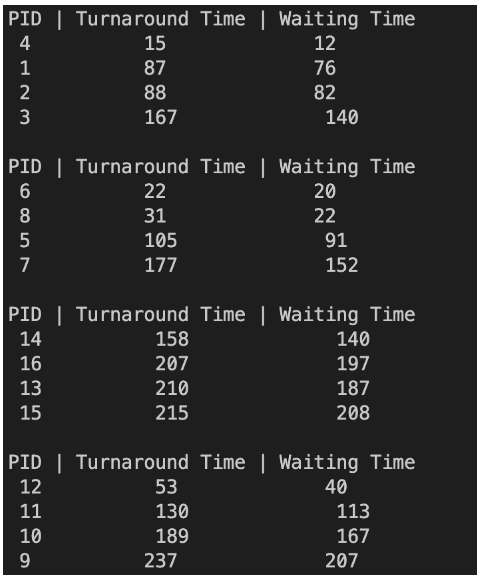
Process ID	Priority	Burst Time
1	0	11
2	0	6
3	0	27
4	0	3
5	1	14
6	1	2
7	1	25
8	1	9
9	2	30
10	2	22
11	2	17
12	2	13
13	3	23
14	3	18
15	3	7
16	3	10

Input: -



```
● ravija@Ravijas-MacBook-Pro Multilevel Queue Scheduling
  2 && "/Users/ravija/Desktop/1st Year Semester 2/OS/Mult
  Enter the number of Processes : 16
  Enter the Priority(0-3) and the Burst Time :
          Process p1: 0 11
          Process p2: 0 6
          Process p3: 0 27
          Process p4: 0 3
          Process p5: 1 14
          Process p6: 1 2
          Process p7: 1 25
          Process p8: 1 9
          Process p9: 2 30
          Process p10: 2 22
          Process p11: 2 17
          Process p12: 2 13
          Process p13: 3 23
          Process p14: 3 18
          Process p15: 3 7
          Process p16: 3 10
```

Output: -



```
PID | Turnaround Time | Waiting Time
 4          15              12
 1          87              76
 2          88              82
 3          167              140

PID | Turnaround Time | Waiting Time
 6          22              20
 8          31              22
 5          105              91
 7          177              152

PID | Turnaround Time | Waiting Time
 14          158              140
 16          207              197
 13          210              187
 15          215              208

PID | Turnaround Time | Waiting Time
 12          53              40
 11          130              113
 10          189              167
 9          237              207
```

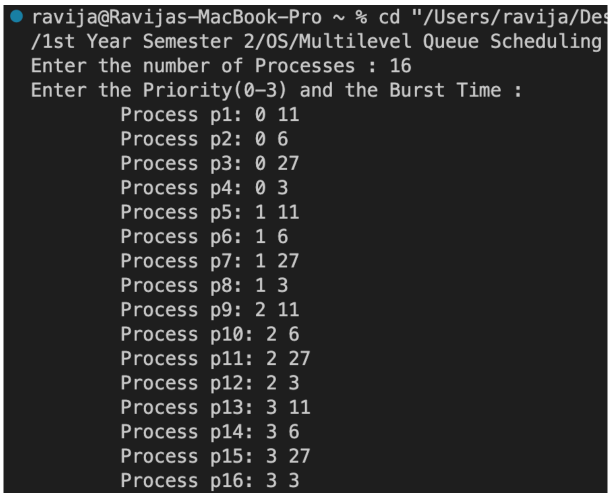
Here, there are four processes in each queue with random burst times.

Queue	Avg. Turnaround Time	Avg. Waiting Time
q0	89.25	77.5
q1	83.75	71.25
q2	152.25	131.75
q3	197.5	183

- Test Case 02

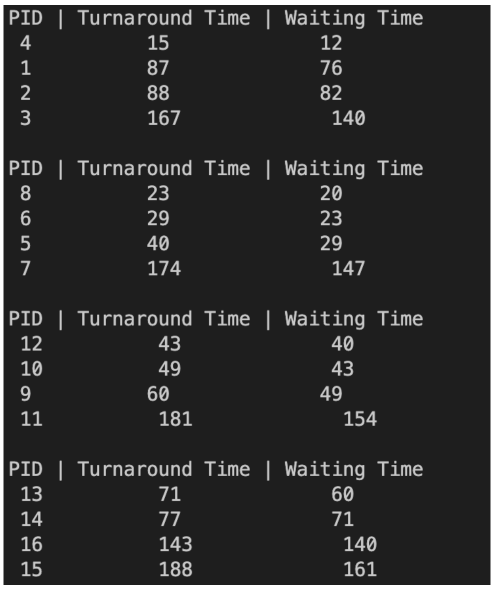
Process ID	Priority	Burst Time
1	0	11
2	0	6
3	0	27
4	0	3
5	1	11
6	1	6
7	1	27
8	1	3
9	2	11
10	2	6
11	2	27
12	2	3
13	3	11
14	3	6
15	3	27
16	3	3

Input: -



```
ravija@Ravijas-MacBook-Pro ~ % cd "/Users/ravija/Des
/1st Year Semester 2/OS/Multilevel Queue Scheduling
Enter the number of Processes : 16
Enter the Priority(0-3) and the Burst Time :
        Process p1: 0 11
        Process p2: 0 6
        Process p3: 0 27
        Process p4: 0 3
        Process p5: 1 11
        Process p6: 1 6
        Process p7: 1 27
        Process p8: 1 3
        Process p9: 2 11
        Process p10: 2 6
        Process p11: 2 27
        Process p12: 2 3
        Process p13: 3 11
        Process p14: 3 6
        Process p15: 3 27
        Process p16: 3 3
```

Output: -



```
PID | Turnaround Time | Waiting Time
 4          15              12
 1          87              76
 2          88              82
 3          167              140

PID | Turnaround Time | Waiting Time
 8          23              20
 6          29              23
 5          40              29
 7          174              147

PID | Turnaround Time | Waiting Time
 12          43              40
 10          49              43
 9          60              49
 11          181              154

PID | Turnaround Time | Waiting Time
 13          71              60
 14          77              71
 16          143              140
 15          188              161
```

Here, there are four processes in each queue with the same burst times.

Queue	Avg. Turnaround Time	Avg. Waiting Time
q0	89.25	77.5
q1	66.5	54.75
q2	83.25	71.5
q3	119.75	108

According to these statistics, Shortest Job First queue has the lowest turnaround time and the waiting time, while First Come First Serve queue having the highest turnaround time and the waiting time.

Advantages & Disadvantages

- **Advantage of the Implementation**

This implementation of Multilevel Queue Scheduling is easy to understand with each scheduling algorithm being implemented separately.

- **Disadvantage of the Implementation**

This implementation cost a lot of space (Space Complexity is high) , due to it having a struct type Process.

- **Pros and Cons of Scheduling Algorithms**

Scheduling Algorithm	***Pros***	***Cons***
Round Robin Algorithm	Fair allocation of CPU time to all processes. Preempts long running processes to ensure that all processes get a chance to execute (Solution for starvation).	May cause unnecessary context switching if the time quantum is too high. Longer waiting time for processes with higher burst times. Not ideal for real-time systems and processes with hard deadlines.
Shortest Job First Algorithm	Result in shorter average waiting time and turnaround time. Ensures that less burst time process executes first.	Requires knowledge of burst time of each and every process. May takes more longer time for long time processes (Starvation may occur due to short time processes keep arriving).
First Come First Serve	Simple and Easy to Implement. Ensures that processes are execute in the order of they arrive. Good for long run processes with no interactivity.	Can result in starvation in later arrived processes (If long burst time processes arrive first). Poor Responce Time. Doesn’t prioritize short or high-priority jobs.

Limitations of the program

In this implementation quantum time of the queues are given so we can't change it with the users will and the process quantum time of the Round Robin queue is also defined at the beginning of the implementation so it can't be changed at latter parts of the implementation.

The *Turnaround time* and the *Waiting time* of the queues are only printed once all the processes in the respective queue have finished its execution (We can't get the output when one process is finished).

The user needs to provide the burst time and priority of each process as inputs at every time we need to run the code.

The scheduling algorithm for each queue is hardcoded, which means that any change in the algorithm would require modifying the code.

Arrival time of all the processes in the program is considered as zero because after the program begins its execution, we can't input processes to the queues.
User have to provide the number of processes that has to execute as an input at the beginning of the exaction.

Conclusion

Multilevel Queue Scheduling is a technique used to manage processes with different priorities efficiently. It involves organizing processes into separate queues based on their priority levels, and each queue has its specific scheduling algorithm. This approach ensures that high-priority tasks get the necessary attention and resources they require and reduces the problem of starvation.

In terms of performance of this code, the Shortest Job First algorithm in q1 and q2 is expected to have the lowest Average Waiting time and Turnaround time, as it prioritizes the shortest jobs. The Round Robin algorithm in q0 may have a higher Waiting time, but it ensures that all processes get a fair share of CPU time. The First Come First Serve algorithm in q3 may have the highest Waiting time, as it does not prioritize short jobs.