Mini Project Report
Computer Architecture and Operating Systems
EEX5563/EEX5564

BY

J C C Jeewandara

Submitted to

Department of Electrical and Computer Engineering

Faculty of Engineering Technology

The Open University of Sri Lanka

On

17-12-2023

Table of Content

List of Figures

1) INRODUCTION

CPU scheduling is an essential aspect of operating systems as it manages the execution of processes that share the central processing unit (CPU) of a computer. Since the CPU can only handle one task at a time, the challenge lies in efficiently handling multiple processes that compete for its attention. The key role of CPU scheduling algorithms is to determine the order in which these processes are given access to the CPU. These algorithms aim to optimize system performance by minimizing process waiting times and maximizing CPU utilization. There are two main types of scheduling strategies: preemptive and non-preemptive. Preemptive strategies allow a running process to be interrupted and replaced by a higher-priority task. Priority assignment, process arrival and completion times, and response time are crucial factors in the development of effective scheduling algorithms. The ultimate objective is to ensure a fair and efficient allocation of CPU time, which contributes to improved system responsiveness and overall throughput.

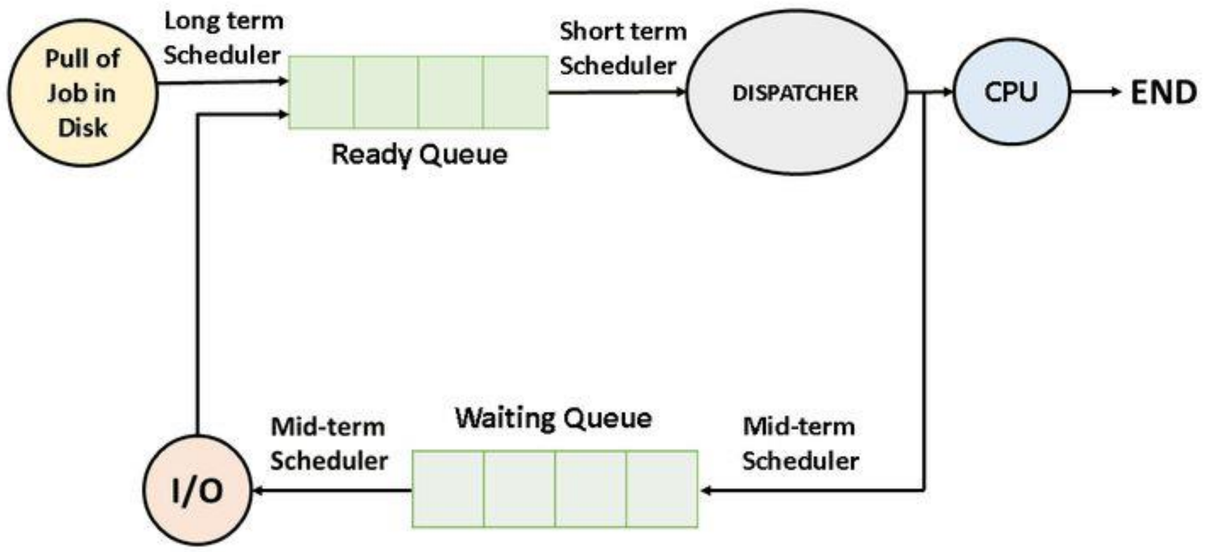


Fig 1 – CPU scheduling process

2) TYPES OF SCHEDULING

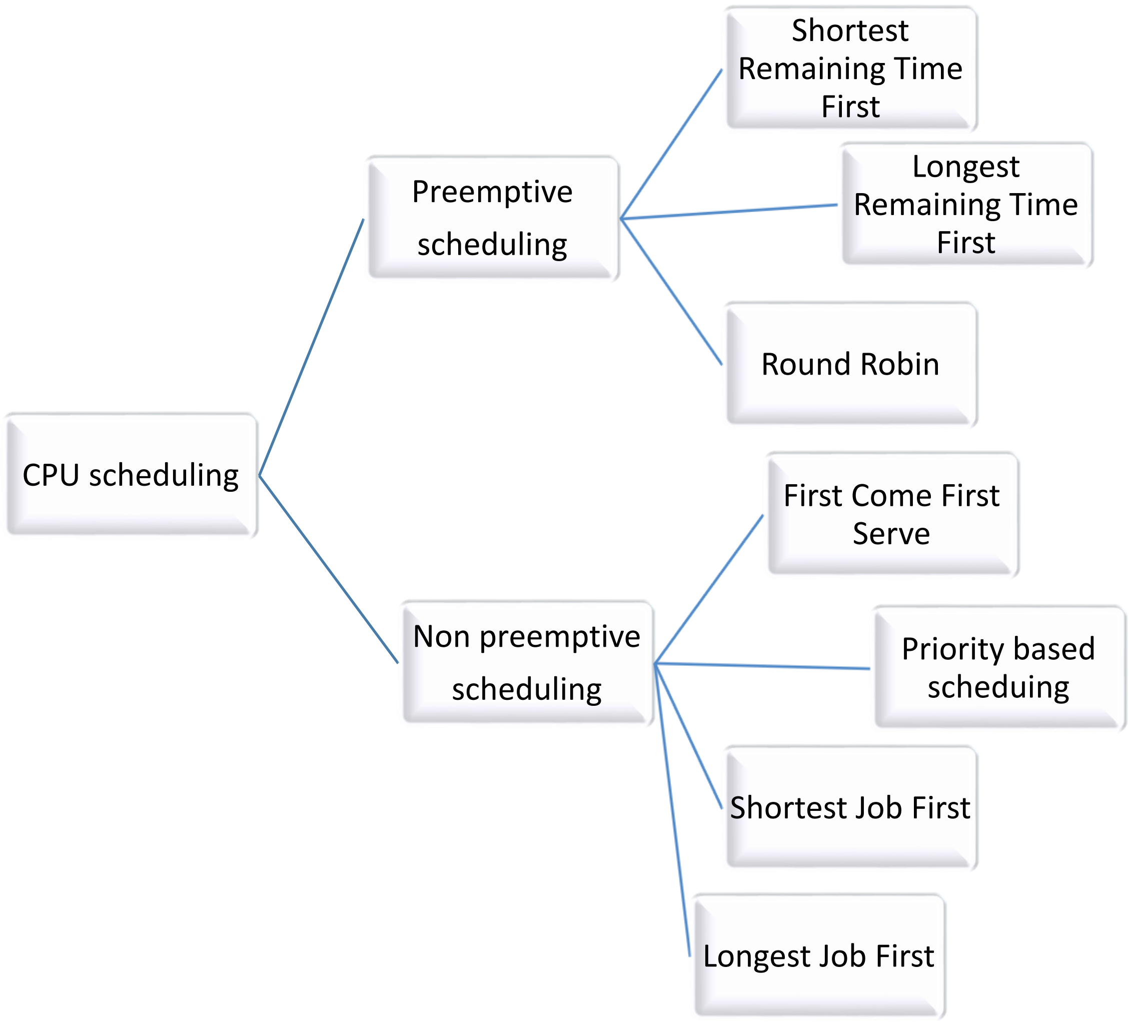

CPU scheduling
Preemptive scheduling
Shortest Remaining Time First
Longest Remaining Time First
Round Robin
Non preemptive scheduling
First Come First Serve
Priority based scheduling
Shortest Job First
Longest Job First

Fig 2 – Scheduling Methods

3) MULTILEVEL QUEUE SCHEDULING

Multilevel queue scheduling is an efficient CPU scheduling algorithm that categorizes processes into multiple queues based on their priority levels. Each queue is associated with its own unique scheduling algorithm. The assignment of processes to a particular queue is determined by their priority or other relevant characteristics. The queues are processed in a predetermined order, and each queue may have its own distinct scheduling policy, such as round-robin or first-come-first-served. By prioritizing the execution of processes in higher-priority queues over those in lower-priority queues, this algorithm strikes a balance between responsiveness and fairness. To illustrate, let's consider a system with three priority levels: high, medium, and low. The high-priority queue could utilize a round-robin scheduling algorithm, the medium-priority queue might employ a priority scheduling algorithm, and the low-priority queue could adopt a first-come-first-served algorithm. This approach ensures that high-priority tasks are promptly addressed while still allowing lower-priority tasks to receive attention, thereby preventing potential starvation.

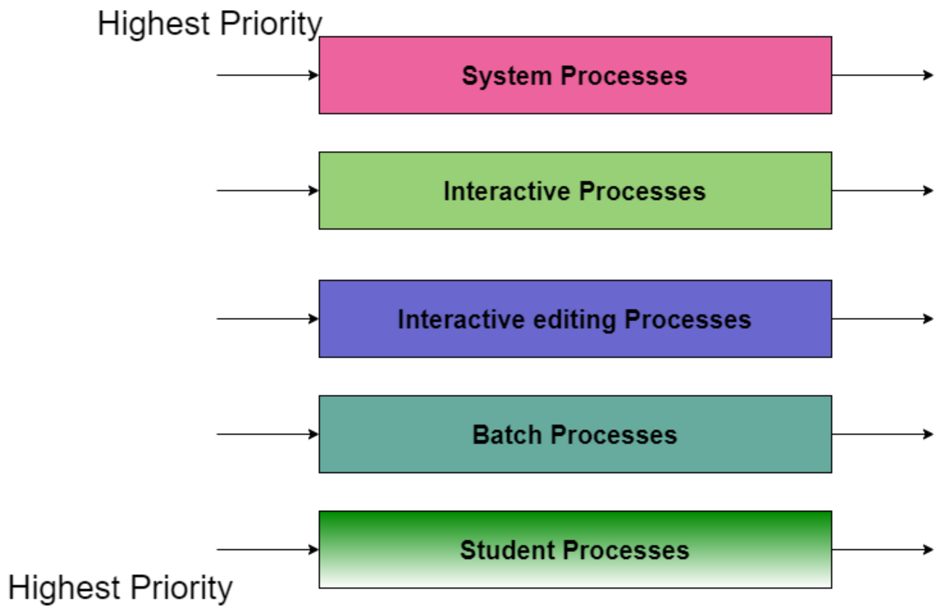


Fig 3 – Multilevel queue scheduling algorithm

4) **Multilevel Feedback Queue Scheduling**

Multilevel Feedback Queue Scheduling is an operating system scheduling algorithm that efficiently distributes system resources, such as CPU time, among multiple processes. This algorithm utilizes a combination of several First-In-First-Out (FIFO) queues, each with a distinct priority level assigned to it. The priority level is determined based on various characteristics of the process, including its age, CPU burst time, and I/O burst time. Initially, each process is assigned to the highest priority queue. The processes in this queue are granted CPU time in a round-robin fashion, meaning that each process is allocated a fixed time slice to execute. If a process completes its execution within the given time slice, it is removed from the queue. However, if it fails to complete its execution, it is demoted to the next lower priority queue.

As a process moves to a lower priority queue, it is allocated a longer time slice. This is because it is assumed that the process requires more CPU time to finish its task. If a process continuously utilizes the CPU for an extended period, it may be further demoted to an even lower priority queue. This prevents the process from monopolizing the CPU time and allows other processes to execute concurrently.

When a process necessitates I/O operations, it is transferred to a distinct I/O queue. After the I/O operation concludes, the process is relocated to the topmost priority queue to finalize its execution. In the absence of I/O operations, a process will persist in the priority queue until it concludes its execution. The multilevel feedback queue scheduling algorithm exhibits adaptability and can be customized to cater to the precise requirements of the system. By enabling shorter processes to swiftly complete their execution and preventing lengthier processes from dominating the CPU time, it can enhance the overall performance of the system.

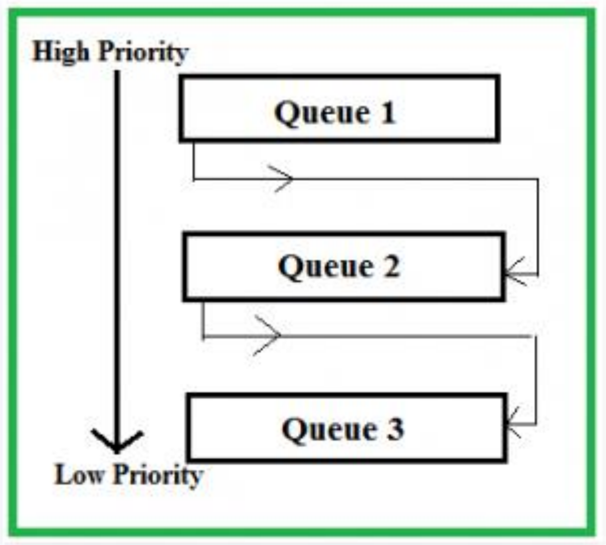


Fig 4 - Multilevel feedback queue scheduling algorithm

5) PROBLEM STATEMENT

CPU scheduling can give rise to a situation known as starvation, wherein a process or a group of processes are unable to obtain the necessary CPU time for execution, resulting in prolonged periods of waiting or inactivity. This occurs when the scheduling algorithm consistently favors certain processes, disregarding others for an extended duration. Essentially, the starved processes remain indefinitely or for an unreasonably long time in the ready queue, unable to make any progress. In a system lacking appropriate mechanisms to prevent starvation, lower-priority processes may suffer neglect if higher-priority processes consistently demand CPU time. This can have a detrimental impact on the overall performance and fairness of the system. Starvation becomes a concern particularly in systems employing priority-based scheduling algorithms, where higher-priority tasks may monopolize the CPU, leaving lower-priority tasks waiting indefinitely.

So to overcome this problem and improve efficieny I proposed **Multilevel Feedback Queue** sheduling is the most suitable one . Multilevel Feedback Queue Scheduling is a preferred approach to tackle the issue of starvation in CPU scheduling. This algorithm utilizes multiple queues with varying priority levels, constantly adapting the priorities of processes based on their behavior. By gradually shifting processes that consume excessive CPU time from higher-priority queues to lower-priority queues, it prevents any single process from monopolizing the CPU and provides an opportunity for lower-priority processes to execute. This adaptability guarantees equitable allocation of resources and responsiveness, thereby making it a highly effective solution for systems with diverse process requirements. Additionally, it mitigates the potential risk of prolonged waiting times for specific tasks.

The time quantum of the queues is determined by the following algorithms. The proposed algorithm divides the processes into three queues: q1, q2, and q3. The first queue, q1, has a short time quantum using the **Round Robin** scheduling algorithm. This allows the execution of short processes within the first queue itself. The time quantum for q1 is set to 8, meaning that processes with a burst time of 8 or less will be executed within this queue. The burst time of these processes is then added to a variable.This variable is used to calculate the time quantum for the second and third queues. If a process has a burst time of 8, it enters the second queue. Within the second queue, processes are arranged and executed using the **Round Robin** scheduling algorithm with a time quantum of 12. If a process's burst time exceeds the calculated time quantum, it is moved to the third queue, where it is executed using the **First Come First Serve** scheduling algorithm.Approximately 75% of the processes are executed in the second queue, while the remaining 25% are executed in the third queue.

6) <u>DISCRIPTION ABOUT APPLIED SHEDULING ALOGRITHEMS</u>

- Round Robin

The Round Robin (RR) scheduling algorithm is a widely utilized preemptive CPU scheduling technique that aims to allocate fair execution time to all processes within a system. In this method, each process is assigned a fixed time slice, also known as a quantum, and the CPU scheduler iterates through the ready queue, granting each process the CPU for its designated quantum. If a process completes its execution within the allocated time quantum, it is removed from the queue. However, if it fails to do so, it is placed at the end of the queue to await its next turn. The Round Robin algorithm ensures that no single process dominates the CPU for an extended duration, thereby promoting fairness and preventing starvation. Although it may result in longer turnaround times for specific tasks, its simplicity and capability to handle various types of processes have made it a widely employed algorithm in time-sharing systems and environments where equitable access to CPU resources is of utmost importance.

Why it is suitable :

- The preemptive nature and fixed time slices assigned to each process in the Round Robin scheduling algorithm effectively reduce starvation.
- Fairness is ensured in Round Robin by providing each process with a turn to execute within a specified time quantum.
- Round Robin prevents any single process from monopolizing the CPU for an extended period.
- The rotation of processes in Round Robin helps distribute resources evenly.
- Round Robin is suitable for environments where equal access to CPU resources is a priority and minimizes the risk of prolonged waiting times for any particular task.

- First Come First Serve

The First Come First Serve (FCFS) scheduling algorithm is a direct and non-preemptive approach employed in CPU scheduling. In FCFS, processes are executed in the sequence they enter the ready queue, with the initial process to arrive being granted CPU time first. Once a process commences execution, it persists until it reaches completion without any interruptions. Although FCFS is uncomplicated to implement, it can result in the "convoy effect," where shorter processes are delayed behind longer ones that arrived earlier. This algorithm has the potential to lead to inefficient CPU utilization and extended average waiting times, particularly in scenarios involving a combination of short and long processes. Despite its simplicity, FCFS is not always the most optimal selection for system performance in dynamic environments where process execution times vary.

Why it is suitable:

- The First Come First Serve (FCFS) scheduling algorithm does not prioritize processes based on their urgency or priority, which can lead to long waiting times for processes that arrive later.
- FCFS is not as effective as some other scheduling algorithms in preventing starvation.
- FCFS is often used in conjunction with the Round Robin scheduling algorithm in a multilevel scheduling environment.
- In a multilevel scheduling system, processes are categorized into different queues with different priority levels.
- FCFS is used within each priority level to ensure fairness among processes with similar priorities.
- Round Robin is employed to switch between queues, preventing processes in higher-priority queues from completely starving processes in lower-priority queues.

- The combination of FCFS and Round Robin in a multilevel scheduling system aims to achieve fairness within priority levels.
- The multilevel scheduling system also provides responsiveness and prevents any single priority level from monopolizing the CPU for an extended period.
- This approach helps strike a balance between fairness and efficiency in handling a diverse set of processes.
- By using FCFS within each priority level and Round Robin between queues, the scheduler can effectively manage processes with varying execution characteristics.

7) <u>EXAMPLE FOR METHODS</u>

Here I will give how those two algorithm works with example.

Round Robin with quantum time 2

Process	Arrival Time	Burst Time
A	0	5
B	1	4
C	2	2
D	4	1

Gantt chart

P1	P2	P3	P1	P4	P2	P1

0 2 4 6 8 9 11 12

First Come First Serve

Process	Arrival Time	Burst Time	Waiting Time
A	0	5	5
B	1	3	4
C	2	2	6
D	3	4	7

Gantt chart

A	B	C	D

0 5 8 10 14

8) ADVANTAGES OF MULTILEVEL FEEDBACK SHEDULING

❖ MLFQ scheduling offers the flexibility to assign different priorities to processes based on their behavior and resource requirements. This allows for dynamic changes in priority as per the needs of the system

.

❖ MLFQ scheduling optimizes resource utilization by giving preference to processes with shorter burst times. By executing these processes first, it reduces the average waiting time and turnaround time of processes, leading to efficient resource utilization.

❖ MLFQ scheduling ensures that processes with lower priority do not suffer from starvation. It guarantees that every process receives a fair share of CPU time, irrespective of its priority, thus preventing any process from being neglected.

❖ MLFQ scheduling is a proficient algorithm for managing both interactive and batch processes. It prioritizes interactive processes to enhance user experience while ensuring that batch processes are completed within a reasonable timeframe.

❖ Implementing MLFQ scheduling is relatively straightforward. It requires only a few priority queues and a set of rules for transferring processes between them. This simplicity makes it easier to implement and maintain in operating systems.

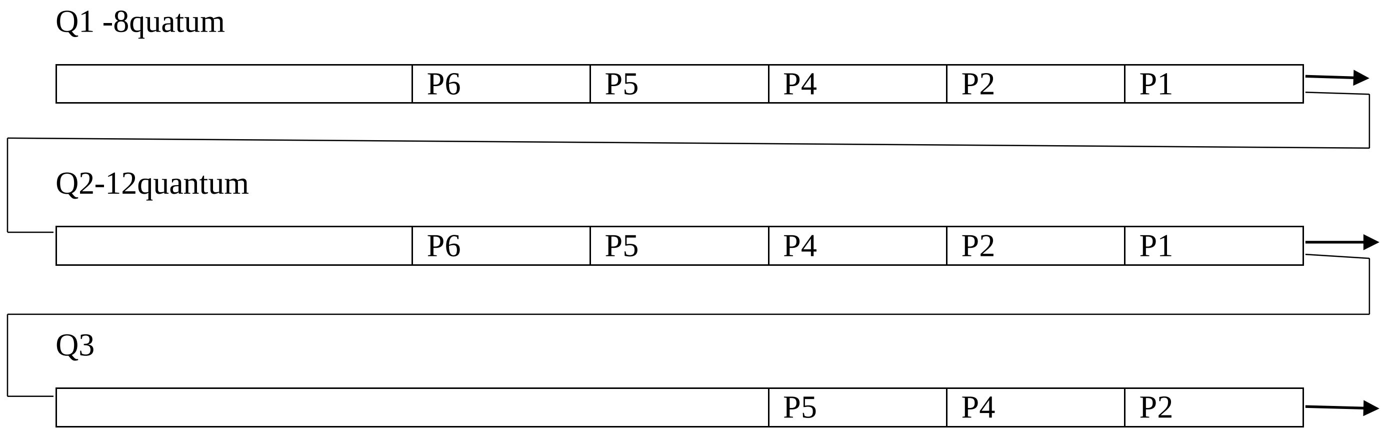
❖ MLFQ scheduling strikes a harmonious balance between priority-based and time-based scheduling approaches. This equilibrium makes it a popular choice in modern operating systems, as it combines the advantages of both approaches to achieve efficient scheduling.

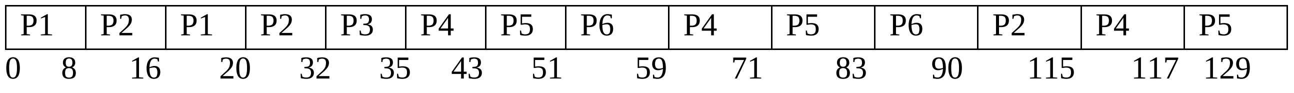
9) SYSTEM CALCULATION AND TESTING

I test my method using this values. I planned to run three steps of given six process. First process is running with round robin with 8 quantum time and second also running with round robin with 12 quantum time and third will run on first come first serve scheduling method.

Process	Arrival Time	Burst Time
P1	0	12
P2	5	45
P3	24	3
P4	30	22
P5	33	32
P6	40	15

Gantt chart





Turnaround time = Completion time – Arrival time

Waiting time = Turnaround time – Burst time

Process	Arrival Time	Burst Time	Completion Time	Turnaround Time	Waiting Time
P1	0	12	20	20	8
P2	5	45	115	30	0
P3	24	3	35	11	8
P4	30	22	117	87	65
P5	33	32	129	96	64
P6	40	15	90	50	35

Average waiting time = Total waiting time/ No of process

= (8+0+8+65+64+35)/6 = 30

Average turnaround time = Total turnaround time time/ No of process

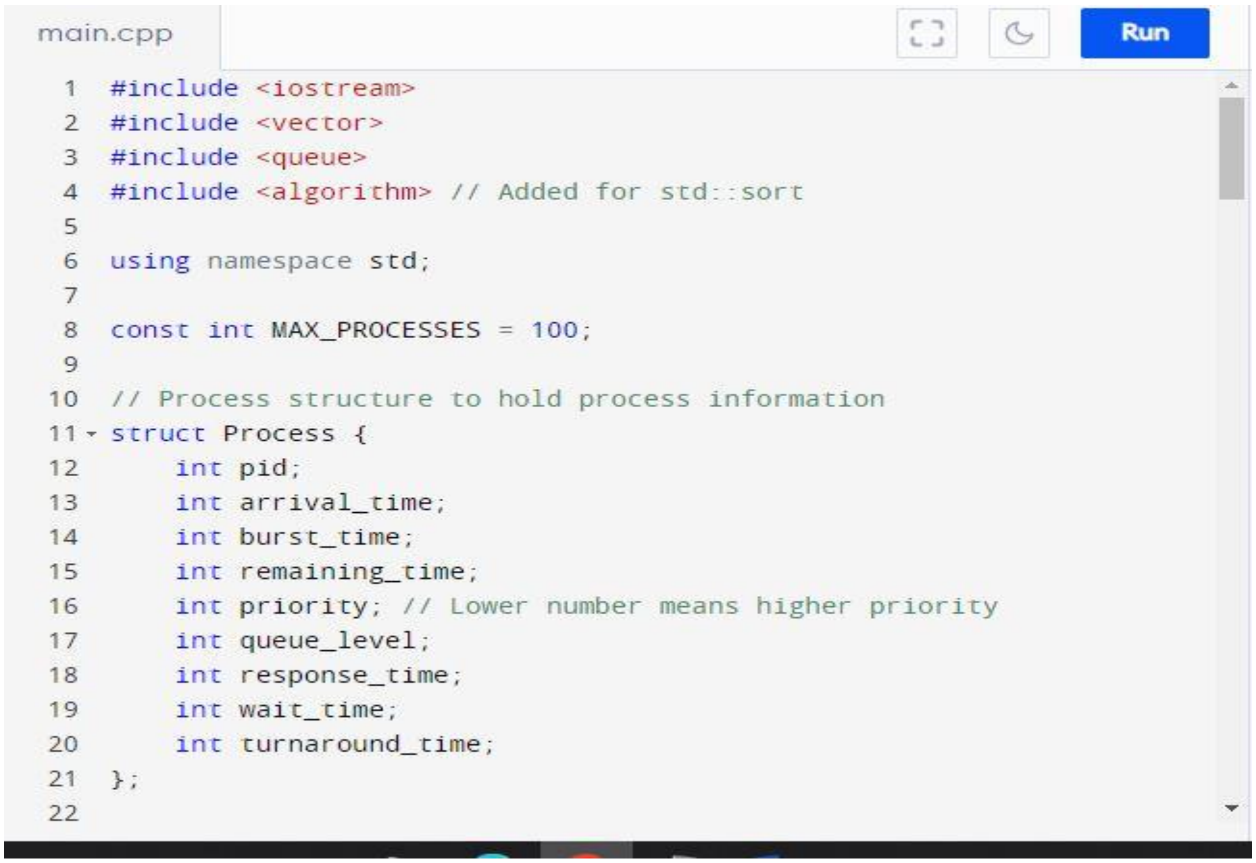
= (20+30+11+87+96+50)/6 = 49

Code and results

```
#include <iostream>

#include <vector>

#include <queue>

#include <algorithm> // Added for std::sort


using namespace std;


const int MAX_PROCESSES = 100;


// Process structure to hold process information

struct Process {

    int pid;

    int arrival_time;

    int burst_time;
    int remaining_time;
    int priority; // Lower number means higher priority
    int queue_level;
    int response_time;
    int wait_time;
    int turnaround_time;
};

// Function to sort the processes by their arrival time
bool sortByArrivalTime(Process a, Process b) {
    return a.arrival_time < b.arrival_time;
}

// Function to simulate the MFQ scheduling algorithm
void simulateMFQ(vector<Process>& processes, int num_queues, int time_quantum[]) {
    // Initialize the queues
    queue<Process> queues[num_queues];
    for (int i = 0; i < num_queues; i++) {
        queue<Process> q;
        queues[i] = q;
    }

    // Add all processes to the first queue
    for (int i = 0; i < processes.size(); i++) {
        processes[i].queue_level = 0;
        queues[0].push(processes[i]);
    }
```

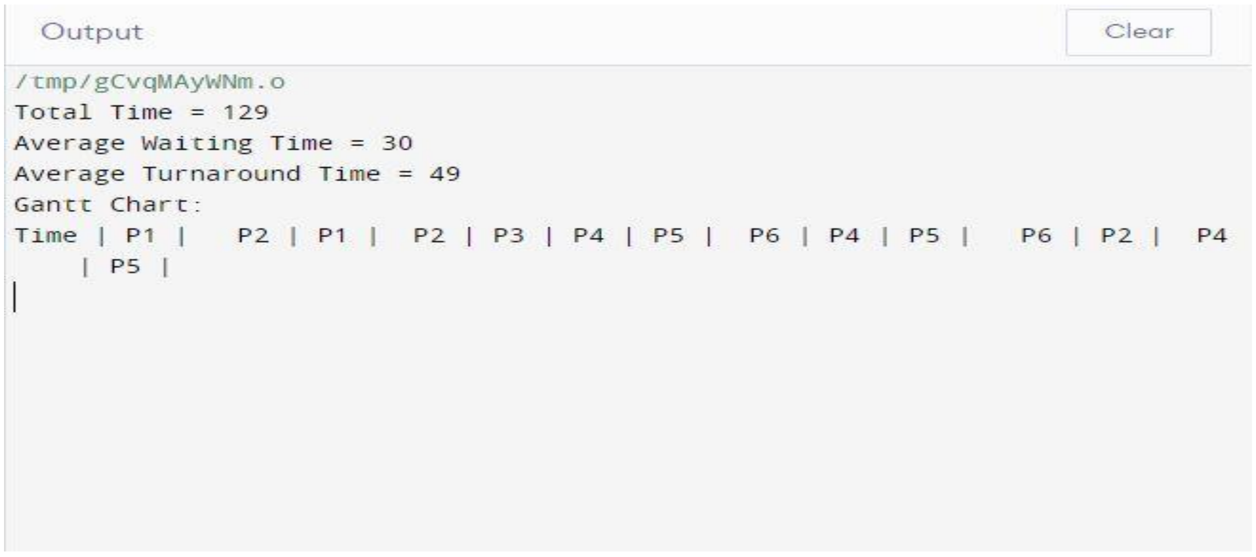
```
int current_time = 0;
int total_wait_time = 0;
int total_turnaround_time = 0;
int total_response_time = 0;
int completed_processes = 0;

vector<pair<int, int>> ganttChart;

cout << "Gantt Chart:\n";

while (completed_processes < processes.size()) {
    bool found_process = false;

    for (int i = 0; i < num_queues; i++) {
        if (!queues[i].empty()) {
            found_process = true;

            Process& current_process = queues[i].front();
            queues[i].pop();

            if (current_process.remaining_time == current_process.burst_time) {
                current_process.response_time = current_time - current_process.arrival_time;
                total_response_time += current_process.response_time;
            }

            int execution_time = min(time_quantum[i], current_process.remaining_time);
            ganttChart.emplace_back(make_pair(current_process.pid, current_time));
            ganttChart.emplace_back(make_pair(-1, current_time + execution_time));
```

```
                current_time += execution_time;
                current_process.remaining_time -= execution_time;

                if (current_process.remaining_time > 0) {
                    queues[current_process.queue_level].push(current_process);
                } else {
                    total_wait_time += current_time - current_process.arrival_time -
        current_process.burst_time;
                    total_turnaround_time += current_time - current_process.arrival_time;
                    completed_processes++;
                }

                break;
            }
        }

        if (!found_process) {
            current_time++;
        }
    }

    // Display the Gantt chart
    for (auto& entry : ganttChart) {
        if (entry.first == -1) {
            cout << "|   ";
        } else {
            cout << "| P" << entry.first << " ";
        }
    }
    cout << "|\n";


    // Display the final time of the system
    cout << "\nFinal Time: " << current_time << endl;
    // Calculate and display average wait time, turnaround time, and response time
    cout << "\nAverage Wait Time: " << static_cast<double>(total_wait_time) / processes.size()
<< endl;
    cout << "Average Turnaround Time: " << static_cast<double>(total_turnaround_time) /
processes.size() << endl;
    cout << "Average Response Time: " << static_cast<double>(total_response_time) /
processes.size() << endl;
}
int main() {
    // Example usage with the provided processes
    vector<Process> processes = {
        {1, 0, 12, 12, 1, 0, 0, 0, 0},
        {2, 5, 45, 45, 1, 0, 0, 0, 0},
        {3, 24, 3, 3, 2, 0, 0, 0, 0},
        {4, 30, 22, 22, 2, 0, 0, 0, 0},
        {5, 33, 32, 32, 0, 0, 0, 0, 0},
        {6, 40, 15, 15, 2, 0, 0, 0, 0}
    };


    int num_queues = 3;
    int time_quantum[] = {8, 12, 1};  // -1 indicates FCFS for the last queue
    sort(processes.begin(), processes.end(), sortByArrivalTime);
    simulateMFQ(processes, num_queues, time_quantum);
   return 0;
}
```



```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm> // Added for std::sort

using namespace std;

const int MAX_PROCESSES = 100;

// Process structure to hold process information
struct Process {
    int pid;
    int arrival_time;
    int burst_time;
    int remaining_time;
    int priority; // Lower number means higher priority
    int queue_level;
    int response_time;
    int wait_time;
    int turnaround_time;
};

```

Fig 5 – Simulation code



```
/tmp/gCvqMAyWNm.o
Total Time = 129
Average Waiting Time = 30
Average Turnaround Time = 49
Gantt Chart:
Time | P1 |   P2 | P1 |   P2 | P3 | P4 | P5 |   P6 | P4 | P5 |    P6 | P2 |   P4
    | P5 |
```

Fig 6 - Results

10) <u>RUNING PROCESS</u>

1) Initialize the Queues:

The MFQS algorithm utilizes multiple queues, each with its own priority level. The configuration of the number of queues and the time quantum for each queue should be determined based on the specific requirements of the system.

2) Assign Processes to the Appropriate Queue:

Upon the arrival of a new process, it is assigned to the initial queue. If the process surpasses its allocated time quantum in the current queue, it is then transferred to the next queue. This process continues until the process is completed or it reaches the final queue.

3) Execute Processes in Each Queue:

The execution of processes takes place in a sequential manner, starting from the highest priority queue and moving downwards to the lower priority queues. Each process is executed for the duration of its assigned time quantum. If a process finishes its execution before the time quantum elapses, it is removed from the queue. Conversely, if a process fails to complete its execution within the time quantum, it is demoted to the next lower priority queue.

4) Adjust the Priorities:

The priorities of the queues can be dynamically adjusted based on the prevailing system conditions. For instance, in the event of system overload, the priority of the lower priority queues can be reduced to allocate more resources to the higher priority queues.

5) Repeat Steps 2-4:

The aforementioned steps are repeated iteratively until all processes have completed their execution.

6) Terminate the Process:

Once a process has finished its execution, it is removed from the queue and terminated.

7) Perform Housekeeping:

At the conclusion of each time quantum, the system undertakes various housekeeping tasks, including updating the priorities of the queues and transferring processes between queues as necessary.

11) <u>CONCLUSION</u>

The experiments conducted have clearly demonstrated the effectiveness of dynamically generating time quantum and implementing the First-Come-First-Serve (FCFS) scheduling policy followed by Round Robin (RR) in subsequent queues. This combination significantly contributes to the improved utilization of CPU and resources. Through the exploration of different combinations of jobs and scheduling policies, it has been found that the Round Robin algorithm, especially when preceded by FCFS in earlier queues, promotes fair CPU usage and aids in reducing the average waiting time and turnaround time. The Round Robin algorithm is renowned for its fairness in allocating CPU time to processes, ensuring that each process receives an equal share in a cyclic manner. This characteristic guarantees that no process is unfairly favored, resulting in a more equitable utilization of resources. By introducing FCFS before RR in the multi-level feedback queue (MLFQ) design, the risk of starvation is further mitigated.

Starvation occurs when certain processes are consistently neglected or delayed in execution, leading to prolonged waiting times. The combination of FCFS and RR helps to address this issue. FCFS, as the initial scheduling policy, prioritizes processes with longer waiting times, reducing the likelihood of starvation. Subsequently, RR contributes to fairness in CPU allocation, preventing any single process from monopolizing resources for an extended period. So, the dynamic generation of time quantum, along with the thoughtful integration of FCFS and RR in the MLFQ design, not only optimizes CPU and resource utilization but also effectively reduces the risk of starvation. This approach ensures a balanced and fair execution of processes, resulting in improved system performance and responsiveness. As further enhancements and realistic analyses are incorporated, the MLFQ can be customized to accommodate the diverse nature of processes submitted to the system, making it a robust solution for modern operating systems.