CS 201, Spring 2025
Homework Assignment 4

Due: 23:59, May 16, 2025

1 Introduction

In this homework, you will simulate a **Multilevel Feedback Queue (MLFQ)** CPU scheduling system using C++. You will implement your own queue logic to manage process execution, preemption, and dynamic movement across different priority levels based on runtime behavior. This simulation is designed to reinforce your understanding of queue data structures and CPU scheduling algorithms.

Multilevel Feedback Queue Scheduling is a flexible CPU scheduling strategy that allows processes to move between different queues depending on their CPU usage patterns. The scheduler typically maintains multiple queues with different priority levels and time quantums. New processes always start in the highest-priority queue and may be demoted to lower-priority queues if they consume too much CPU time.

You are given the interface of a class called `MLFQScheduler`, which coordinates the simulation. Your task is to implement the required functions to manage the scheduling of processes across three priority queues and simulate their execution based on strict priority and quantum rules.

For a basic introduction to the Multilevel Feedback Queue (MLFQ) scheduling algorithm, you may refer to:
`https://www.studytonight.com/operating-system/multilevel-feedback-queue-scheduling`.

2 Assignment Details

This assignment simulates a Multilevel Feedback Queue (MLFQ) CPU scheduler using three priority queues.

- The system contains **three queues**:
  - **Queue 1 (Q1)**: Highest priority. Uses Round-Robin (RR) scheduling with a time quantum specified by the parameter `q1`.
  - **Queue 2 (Q2)**: Medium priority. Uses Round-Robin (RR) scheduling with a time quantum specified by the parameter `q2`.
  - **Queue 3 (Q3)**: Lowest priority. Uses First-Come First-Served (FCFS) scheduling and does not use a time quantum.
- **Processes:** Each process in the system arrives with pid and burst time. There are several attributes that are helpful to track throughout the simulation:
  - **PID (Process ID)**: A unique identifier assigned to each process when it enters the system.

- **Arrival Time**: The simulation clock value at which the process is added to the system.
- **Burst Time**: The total CPU time required by the process to complete.
- **Remaining Time**: The amount of CPU time the process still needs to finish. It starts equal to the burst time and decreases during execution.
- **Used Quantum**: The amount of time the process has spent running in its current queue. This is relevant only for Q1 and Q2 and helps determine when a process should be demoted.
- **Start Time**: The first clock time when the process begins execution.
- **End Time**: The clock time when the process finishes execution.

- All processes enter the system through **Queue 1**. If a process uses up its full time quantum without completing, it is **demoted** to the next lower-priority queue.

- In a Multilevel Feedback Queue (MLFQ) scheduler, strict priority enforcement is essential for minimizing response time and simulating real-world system behavior. In this homework, the following preemption rules are applied:

  If a process is currently running from a lower-priority queue (such as Q2 or Q3), and a new process arrives in a higher-priority queue (such as Q1), the currently running process is **preempted**. The preempted process is paused and placed **back into the same queue as before**, at the back of the queue, without resetting its `used quantum` value. This ensures that the process continues consuming its originally assigned CPU time fairly, without gaining an unfair advantage from being preempted.

  The newly arrived higher-priority process begins execution immediately after the preemption. Only processes running from a lower-priority queue can be preempted by a process from a higher-priority queue; processes running from Q1 are never preempted by another process arriving into Q1.

  This preemption mechanism ensures that:

  - A running process from Q2 or Q3 is **preempted** if there is a process waiting in a higher-priority queue (Q1 for Q2, Q1 and Q2 for Q3).
  - The preempted process is **placed back into its corresponding queue**, at the end.
  - The preempted process's `used quantum` is **preserved** and continues from where it left off.
  - This maintains fairness and prevents processes from abusing the scheduler to gain extra CPU time unfairly.

  Strict priority enforcement ensures that urgent or short processes are given prompt access to CPU time, improving system responsiveness and modeling real-world operating system behavior accurately.

- **Starvation Problem:** Since higher-priority queues are always served first, long processes in lower-priority queues may suffer from **starvation**—they may never get CPU time if short tasks keep arriving. To address this issue, the scheduler can perform a **priority boost**, moving all waiting processes back to the highest-priority queue (Q1). This ensures fairness and prevents starvation in long-running simulations.

2.1 Class Implementation

Your solution must be implemented in a class called **MLFQScheduler**. Below is the required public part of the `MLFQScheduler` class. The interface for the class must be written in a file called **MLFQScheduler.h** and its implementation must be written in a file called **MLFQScheduler.cpp**. You can define additional public and private member functions and data members in this class. You are also expected to implement your own `Process` and `Queue` classes.

```
class MLFQScheduler {
public:
    MLFQScheduler(const int q1, const int q2);
    ~MLFQScheduler();

    void addProcess(const int pid, const int burstTime);
    void tick(const int timeUnits);
    void run();
    void priorityBoost();
    void printScheduler() const;
};
```

The member functions are defined as follows:

- `MLFQScheduler` Constructs the `MLFQScheduler` object by initializing three queues:
  - **Queue 1 (Q1)**: Highest-priority queue. Uses Round Robin (RR) scheduling with a time quantum of `q1`.
  - **Queue 2 (Q2)**: Medium-priority queue. Uses Round Robin (RR) scheduling with a time quantum of `q2`.
  - **Queue 3 (Q3)**: Lowest-priority queue. Uses First-Come First-Served (FCFS) scheduling. This queue does not use a time quantum.

  The simulation clock is initialized to 0, and no process is running at the beginning. Processes are always scheduled from the highest-priority non-empty queue, enforcing strict priority across levels.

- `addProcess` Adds a new process into the system. The process is inserted into the highest-priority queue (Q1) immediately upon creation (it should not wait for the next tick).

  This function takes two parameters:

- `pid`: The unique identifier for the process.
- `burstTime`: The total amount of CPU time the process requires to complete.

The arrival time of the process is recorded as the current simulation clock time. All newly arriving processes always start in Q1 regardless of their burst time.

A log message must be printed after a process is added, showing the process ID, burst time, and arrival time. You can assume that each pid given will be unique.

- `tick` Simulates the passage of time in the system by advancing the simulation clock by a specified number of time units.

  At each tick, the scheduler performs the following steps:

  - If no process is currently running, the scheduler selects the next process from the highest-priority non-empty queue (Q1 first, then Q2, then Q3).
  - The selected process begins (or resumes) execution and runs for one time unit.
  - The process's `used quantum` is incremented.

  After this:

  - If the running process completes (i.e., its remaining time becomes 0), it is removed from the system.
  - If the process is from Q1 or Q2 and it has used up its full time quantum without completing, it is **demoted** to the next lower-priority queue (Q1 Q2, Q2 Q3), and its `used quantum` is reset.
  - If the process is from Q3, it continues running until completion. Q3 processes do not have a time quantum.

  Preemption Rule: If a higher-priority queue becomes non-empty while a lower-priority process is running (e.g., a Q1 process arrives while a Q2 or Q3 process is running), the currently running process is **preempted immediately**. It is placed at the end of its original queue *without resetting* its `used quantum`. When this process is rescheduled later, it resumes execution from where it left off.

  After simulating the given number of ticks, the simulation clock is updated accordingly. A log showing the state of the scheduler at the end of the time passed since the beginning of the simulation can be printed separately by calling `printScheduler()`.

- `run` Simulates the system execution by incrementing the simulation clock until all processes have finished.

  The simulation continues until all three queues (Q1, Q2, and Q3) are empty and no process is currently running.

  If the system is already empty when `run()` is called (no processes in any queue and no running process), the statistics are immediately printed without advancing the simulation clock.

  After all processes have completed, the function prints the following statistics:

- **Average Waiting Time**: The average time processes spent waiting in queues, calculated as:

$$\text{Waiting Time} = \text{Turnaround Time} - \text{Burst Time}$$

$$\text{Average Waiting Time} = \frac{\text{Total Waiting Time}}{\text{Number of Processes}}$$

- **Average Turnaround Time**: The average total time between a process's arrival and its completion, calculated as:

$$\text{Turnaround Time} = \text{End Time} - \text{Arrival Time}$$

$$\text{Average Turnaround Time} = \frac{\text{Total Turnaround Time}}{\text{Number of Processes}}$$

- **Total Runtime**: The final simulation clock value when all processes have completed.

Finally, a log message is printed confirming that all processes have completed execution.

- **(Bonus)** `priorityBoost`
  Promotes all *waiting* processes from Queue 2 and Queue 3 to the end of Queue 1. Only **waiting processes** in Q2 and Q3 are boosted. The currently running process (if any) is **not affected**. A boosted process is inserted into Q1 and its `usedQuantum` is reset to 0. The system clock **does not advance** when a priority boost is performed. After a boost, boosted processes are inserted at the **back** of Q1. A log message must be printed after a boost, indicating how many processes were boosted and at what clock time the boost occurred.

  You must ensure that processes maintain their execution order during promotion (i.e., FCFS order is preserved within each boosted queue).

2.2 Example Test Case and Output

This section walks through an example test case and the expected output for your MLFQ scheduler. We will use a similar driver program to test your code, so make sure that your class is named **MLFQScheduler**, its interface is provided in **MLFQScheduler.h**, and the public functions are implemented as described.

Figure 1 illustrates the state of the scheduler queues throughout the simulation. Each state corresponds to the output of a single call to the `printScheduler()` function in the example test program. This visualization is provided to help you trace process transitions over time.

Example Test Program

```cpp
#include <iostream>
#include "MLFQScheduler.h"

using namespace std;

int main()
{
    MLFQScheduler scheduler(4, 8);
    scheduler.printScheduler();
    cout << endl;

    scheduler.addProcess(1, 2);
    scheduler.addProcess(2, 18);
    cout << endl;

    scheduler.printScheduler();
    cout << endl;

    scheduler.tick(1);
    scheduler.printScheduler();
    cout << endl;

    scheduler.tick(1);
    scheduler.printScheduler();
    cout << endl;

    scheduler.tick(1);
    scheduler.printScheduler();
    cout << endl;

    scheduler.addProcess(3, 5);
    cout << endl;

    scheduler.tick(3);
    scheduler.printScheduler();
    cout << endl;

    scheduler.addProcess(4, 20);
    scheduler.addProcess(5, 1);
    cout << endl;

    scheduler.tick(1);
    scheduler.printScheduler();
    cout << endl;

    scheduler.tick(1);
    scheduler.printScheduler();
```

```
    cout << endl;

    scheduler.tick(3);
    scheduler.printScheduler();
    cout << endl;

    scheduler.tick(4);
    scheduler.printScheduler();
    cout << endl;

    scheduler.tick(1);
    scheduler.printScheduler();
    cout << endl;

    scheduler.addProcess(6, 3);
    cout << endl;

    scheduler.printScheduler();
    cout << endl;

    scheduler.tick(1);
    scheduler.printScheduler();
    cout << endl;

    scheduler.priorityBoost();
    cout << endl;

    scheduler.printScheduler();
    cout << endl;

    scheduler.tick(4);
    scheduler.printScheduler();
    cout << endl;

    scheduler.tick(8);
    scheduler.printScheduler();
    cout << endl;

    scheduler.tick(8);
    scheduler.printScheduler();
    cout << endl;

    scheduler.tick(8);
    scheduler.printScheduler();
    cout << endl;

    scheduler.run();
    cout << endl;

```

```
    scheduler.tick(1);
    scheduler.printScheduler();

    return 0;
}
```

Expected Output

```
--- Clock: 0 ---
Q1:
Q2:
Q3:
Running: None

[INFO] Process P1 with burst time 2 arrived at clock 0.
[INFO] Process P2 with burst time 18 arrived at clock 0.

--- Clock: 0 ---
Q1: P1 P2
Q2:
Q3:
Running: None

--- Clock: 1 ---
Q1: P2
Q2:
Q3:
Running: P1 (Remaining Time: 1)

--- Clock: 2 ---
Q1: P2
Q2:
Q3:
Running: P1 (Remaining Time: 0)

--- Clock: 3 ---
Q1:
Q2:
Q3:
Running: P2 (Remaining Time: 17)

[INFO] Process P3 with burst time 5 arrived at clock 3.

--- Clock: 6 ---
Q1: P3
Q2:
Q3:
Running: P2 (Remaining Time: 14)
```

```
41
42 [INFO] Process P4 with burst time 20 arrived at clock 6.
43 [INFO] Process P5 with burst time 1 arrived at clock 6.
44
45 --- Clock: 7 ---
46 Q1: P4 P5
47 Q2: P2
48 Q3:
49 Running: P3 (Remaining Time: 4)
50
51 --- Clock: 8 ---
52 Q1: P4 P5
53 Q2: P2
54 Q3:
55 Running: P3 (Remaining Time: 3)
56
57 --- Clock: 11 ---
58 Q1: P5
59 Q2: P2 P3
60 Q3:
61 Running: P4 (Remaining Time: 19)
62
63 --- Clock: 15 ---
64 Q1:
65 Q2: P2 P3 P4
66 Q3:
67 Running: P5 (Remaining Time: 0)
68
69 --- Clock: 16 ---
70 Q1:
71 Q2: P3 P4
72 Q3:
73 Running: P2 (Remaining Time: 13)
74
75 [INFO] Process P6 with burst time 3 arrived at clock 16.
76
77 --- Clock: 16 ---
78 Q1: P6
79 Q2: P3 P4
80 Q3:
81 Running: P2 (Remaining Time: 13)
82
83 --- Clock: 17 ---
84 Q1:
85 Q2: P3 P4 P2
86 Q3:
87 Running: P6 (Remaining Time: 2)
88
89 [INFO] Priority boost performed at clock 17. 3 processes boosted to Q1.
```

```
--- Clock: 17 ---
Q1: P3 P4 P2
Q2:
Q3:
Running: P6 (Remaining Time: 2)

--- Clock: 21 ---
Q1: P2
Q2:
Q3:
Running: P4 (Remaining Time: 15)

--- Clock: 29 ---
Q1:
Q2: P2
Q3:
Running: P4 (Remaining Time: 11)

--- Clock: 37 ---
Q1:
Q2:
Q3: P4
Running: P2 (Remaining Time: 8)

--- Clock: 45 ---
Q1:
Q2:
Q3: P2
Running: P4 (Remaining Time: 3)

--- Final Statistics ---
Average Waiting Time: 12.1667
Average Turnaround Time: 20.3333
Total Runtime: 49
[INFO] All processes have completed execution.

--- Clock: 50 ---
Q1:
Q2:
Q3:
Running: None
```

3 Specifications

1. You **ARE NOT ALLOWED** to use STL containers such as `queue`, `vector`, or `list`. You must implement your own dynamic queue logic to manage processes within each level of the Multilevel Feedback Queue scheduler. Any submission using STL queues will receive **no points** for the relevant parts. You are free to define additional classes or helper structures, but **all queue-related logic must be implemented manually**.

2. You MUST implement the queue and any additional container (e.g., list) by yourself. You ARE NOT ALLOWED to use the data structures and related functions in the C++ standard template library (STL) or any external library. You can use the implementations discussed during the lectures.

3. Moreover, you ARE NOT ALLOWED to use any global variables or any global functions.

4. Output message for each operation MUST match the format shown in the output of the example code.

5. Your code MUST NOT have any memory leaks. You will lose points if you have memory leaks in your program even though the outputs of the operations are correct. To detect memory leaks, you may want to use Valgrind which is available at `http://valgrind.org`.

4 Submission

1. In this assignment, you must have separate interface and implementation files (i.e., separate `.h` and `.cpp` files) for your class. Your class name MUST BE `MLFQScheduler` and your file names MUST BE `MLFQScheduler.h` and `MLFQScheduler.cpp`. Note that you may write additional class(es) in your solution.

2. The code (`main` function) given above is just an example. We will test your implementation using different scenarios, which will contain different function calls. Thus, do not test your implementation only by using this example code. We recommend you to write your own driver files to make extra tests. However, you MUST NOT submit these test codes (we will use our own test code). In other words, do not submit a file that contains a function called `main`.

3. You should put all of your `.h` and `.cpp` files into a folder and zip the folder (in this zip file, there should not be any file containing a `main` function). The name of this zip file should conform to the following name convention: secX-Firstname-Lastname-StudentID.zip where X is your section number. The submissions that do not obey these rules will not be graded. Please do not use Turkish letters in your file and folder names.

4. Make sure that each file that you submit (each and every file in the archive) contains your name, section, and student number at the top as comments.

5. You are free to write your programs in any environment (you may use Linux, Windows, MacOS, etc.). On the other hand, we will test your programs on "dijkstra.ug.bcc.bilkent.edu.tr" and we will expect your programs to compile and run on the dijkstra machine. Your code will be tested by using an automated test suite that includes multiple test cases where each case corresponds to a specific number of points in the overall grade. We will provide you with example test cases by email. Thus, we strongly recommend you to make sure that your program successfully compiles and correctly works on dijkstra.ug.bcc.bilkent.edu.tr before submitting your assignment. If your current code does not fully compile on dijkstra before submission, you can try to comment out the faulty parts so that the remaining code can be compiled and tested during evaluation.

6. This assignment is due by 23:59 on Friday, May 16, 2025. You should upload your work to Moodle before the deadline. No hardcopy submission is needed. Late submissions will not be accepted (if you can upload to Moodle, then you are fine). There will be no extension to this deadline.

7. We use an automated tool as well as manual inspection to check your submissions against plagiarism. For questions regarding academic integrity and use of external tools (including generative AI tools), please refer to the course home page and the Honor Code for Introductory Programming Courses (CS 101/102/201/202) at `https://docs.google.com/document/d/1v_3ltpV_lClLsROXrMbojyuv4KrFQAm1uoz3SdC-7es/edit?usp=sharing`.

8. This homework will be graded by your TA **Sude Önder** (sude.onder@bilkent.edu.tr). Thus, you may ask your homework related questions directly to her. There will also be a forum on Moodle for questions.

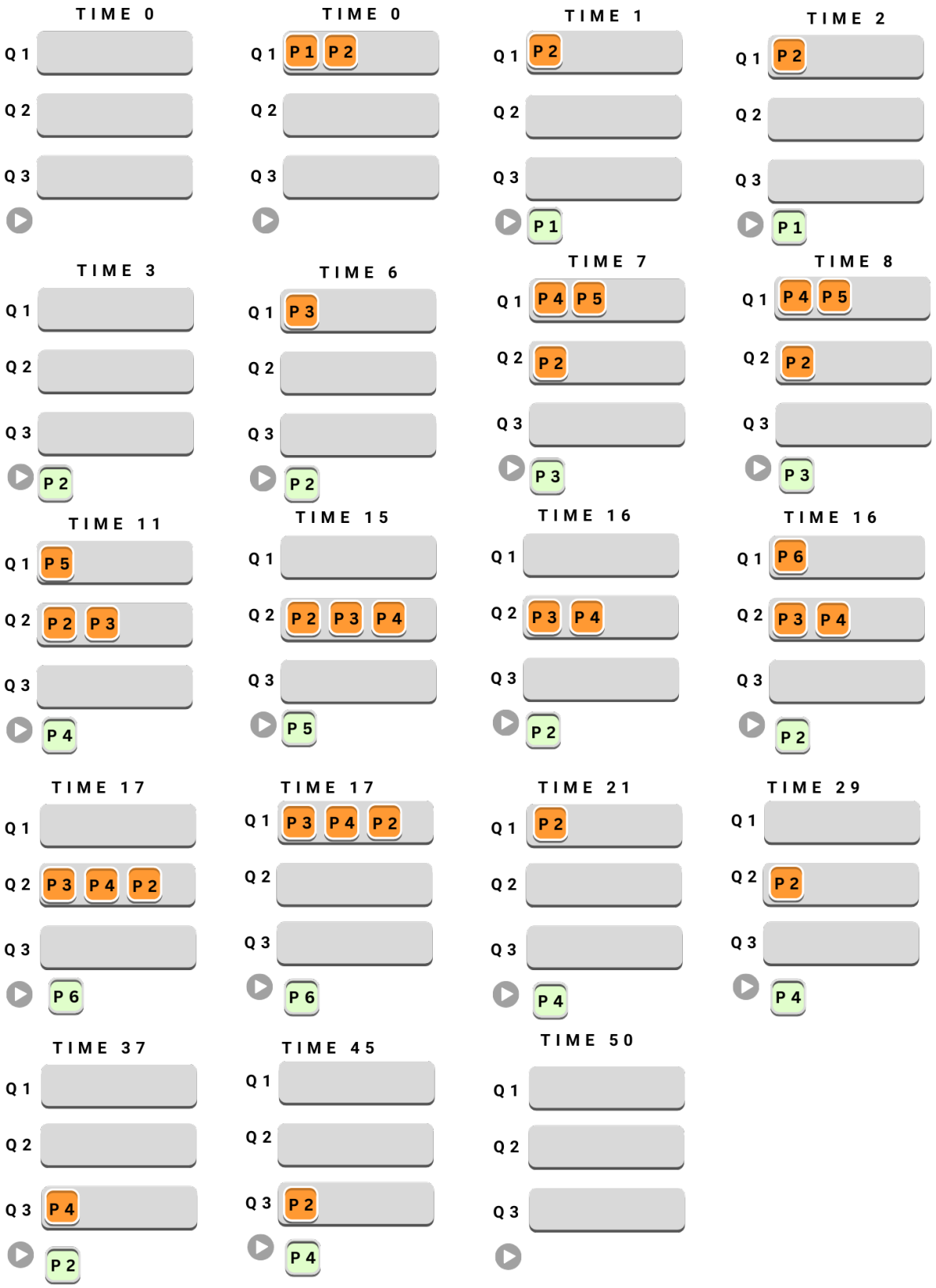

Figure 1: Queue states after each `printScheduler()` call in the example test case. Processes highlighted in green indicate the process that **run** at that clock time.