CSC 360 Summer 2023
Assignment 3

Programming Environment

For this assignment, your code must work in the JupyterHub environment (`https://jhub.csc.uvic.ca`). Even if you already have access to your own Unix system, we recommend you work as much as possible with the JupyterHub environment. Bugs in systems programming tend to be platform-specific and something that works perfectly at home may end up crashing on a different computer-language library configuration.

Individual Work

This assignment is to be completed by each individual student (i.e. no group work). You are encouraged to discuss aspects of the problem with your classmates, however **sharing of code is strictly forbidden**. If you are unsure about what is permitted or have other questions regarding academic integrity, please ask the instructor. Code-similarity tools will be run on submitted programs. Any fragments of code found or generated on the web and is used in your solution must be properly cited (citation in the form of a comment in your code).

Assignment Objectives

Write a C program `mlfq.c` which implements a simulation of round-robin CPU scheduling which uses a multi-level feedback queue and a boosting mechanism which prevents tasks from waiting indefinitely (starvation). There is no threading or synchronization required to complete this assignment.

You will be implementing a tick-by-tick simulation of a single-core CPU scheduler for a set of CPU-bound and IO-bound tasks. We will be using abstract ticks (incrementing a tick variable) rather than actually counting milliseconds or microseconds.

The starter file is provided to you which already contains functionality intended to reduce your work. Amongst other things, the file contains

- code needed to open a test-case file
- a function called `read_instruction()` which uses the open file to read out the three integers corresponding to the current line in the file
- `printf()` statements with the correct formatting needed for simulator output, although your code must ensure the correct values are provided to the arguments of these print statements
- code to detect when all test-case lines have been read from the file into the program and that the simulator loop should terminate

This assignment is quite different than the previous assignments. Rather than a significant amount of systems programming, you will be writing a simulator. The difficulty of this assignment will not necessarily be any new C code, but rather understanding how the parts of the given code work together.

Submission

Submit your `mlfq.c` file containing your programming solution to Brightspace.

Multi-level Feedback Queue Specifications

A multi-level feedback queue (MLFQ) is designed to ensure that the tasks require quick responses (which are often characterized as having short CPU bursts) are scheduled before tasks which are more compute bound (which are characterized as having longer CPU bursts).
Your simulation will be of an MLFQ with three different queues: `queue_1` with a quantum of 2, `queue_2` with a quantum of 4, and `queue_3` with a quantum of 8.

- When a new task arrives, it is enqueued onto `queue_1`.
- When there is a CPU scheduling event, `queue_1` is examined. If it is not empty, then the task at the front of this queue is selected to run and given a quantum of 2. Otherwise, `queue_2` is examined, then the task at the front of this queue is selected to run and given a quantum of 4. Otherwise, the `queue_3` is examined, and the task at the front of this queue is selected to run and given a time quantum of 8.
- If a task selected from either `queue_1` or `queue_2` finishes its current burst within the quantum provided to it, then it is placed at the back of the queue from which it was taken.
- However, if a task selected from either `queue_1` or `queue_2` has a burst that exceeds its quantum, then it is interrupted (as would be the case for any round-robin algorithm), and it is placed into the next queue with a larger quantum. That is, a task taken from `queue_1` would be placed at the end of `queue_2`, and a task taken from `queue_2` would be placed at the end of `queue_3`.
- The currently running task may be preempted. That is, if the scheduler discovers that there is a task on a higher-priority queue than the current task, it must preempt the current task. The current task is put to the back of the queue from which it was taken, and the higher-priority task becomes the current task.
- Without any mechanism which allows processes to leave `queue_3`, a process may have to wait indefinitely. For example, a task in `queue_3` might never get a chance to resume if it is preempted by a series of new tasks on a queue with a higher priority. To prevent this, the CPU must boost tasks to to the top queue periodically to give them a chance to compete for CPU time with other tasks. More details about the boost mechanism are provided later in this document.

You are given a queue implementation in the files `queue.c` and `queue.h`. Please read this code carfully and do not modify this code since we will be using the given version of `queue.c` to evalulate your submission of `mlfq.c`.

Input and Output of the MLFQ Simulator

Your implementation of `mlfq.c` will accept a single argument consisting of a text file. The lines of the text file contain the information about tasks for a particular simulation case. For example, to run the first test case, the following would be entered at the command line:

```
./mlfq cases/test1.txt
```

Each line of the simulation case text file represents one of three possibilities: task creation, task requires CPU time, and task termination request.

For example, a line such as

```
13, 3, 0
```

indicates that at tick 13, task 3 has been created (0 means creation).

A line such as

```
14, 3, 6
```

indicates that at tick 14, task 3 initiates an action that will require 6 ticks of CPU time. There may be many such CPU-tick lines for a task contained in the simulation case.

Lastly, a line such as

```
21, 3, -1
```

indicates that at tick 21, task 3 will terminate once its remaining CPU bursts' ticks are scheduled. If no such CPU ticks are remaining, then the task would terminate immediately.

The following example shows the contents of the provided `cases/test1.txt` and the expected simulator output for this case. You will see in this output that:

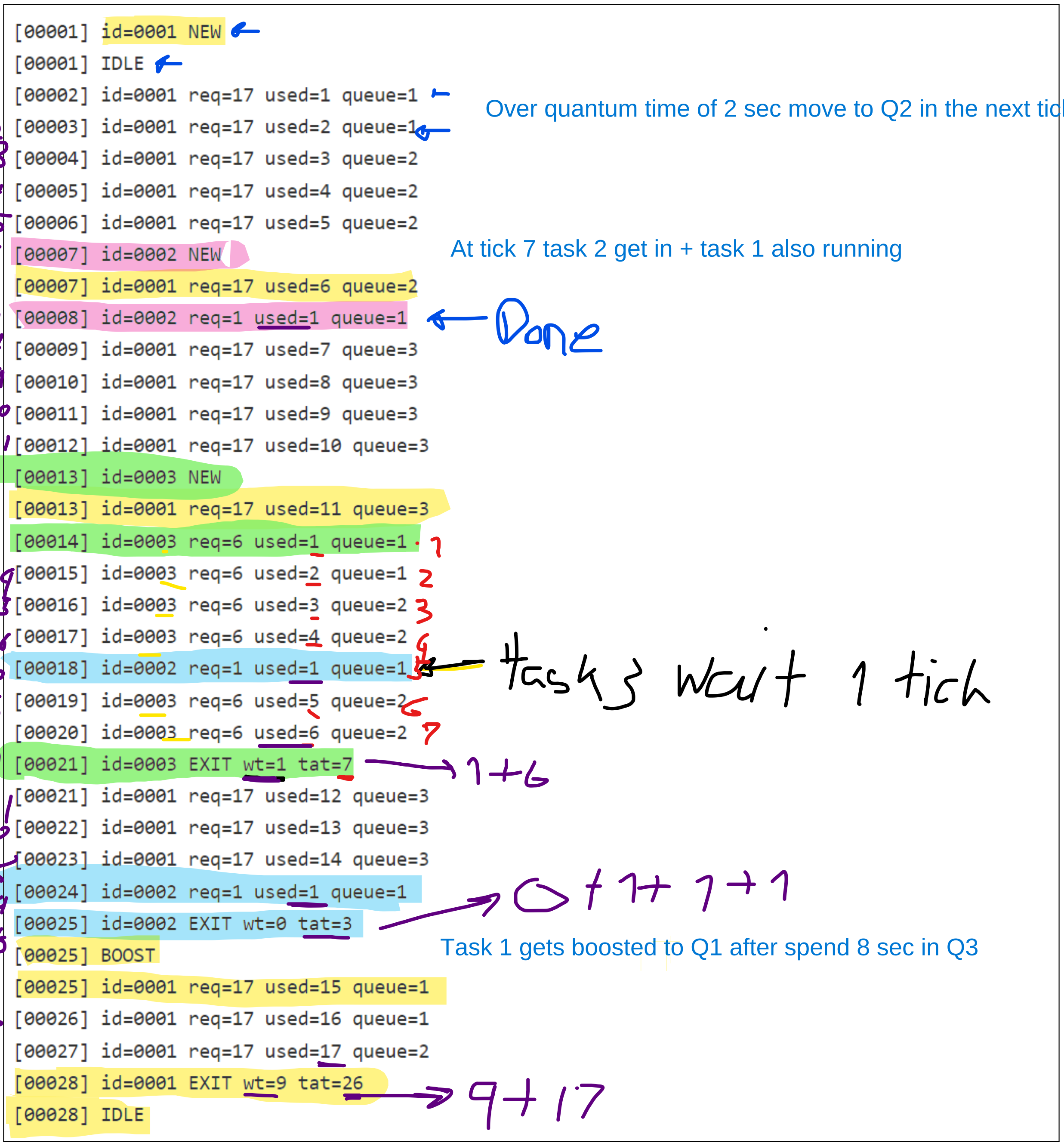
- The tick at which a task enters the system is shown (i.e. each simulated tick appears on its own line).
- The currently-running task at the given tick is displayed on a line, with some statistics about that task at the start of the tick. The meaning of these statistics will explained further later in this document.
- If there is no task to run at a tick, then the simulator outputs `IDLE` for that tick. A note on what `IDLE` means is given later in this document.
- When a boost occurs, a message indicating the tick number and boost message is displayed.

Contents of `cases/test1.txt`:

```
1,1,0
2,1,17
7,2,0
8,2,1
13,3,0
14,3,6
18,2,1
21,3,-1
24,2,1
25,2,-1
28,1,-1
```

Cannot run at the same uploaded tick

Output of a completed simulation of `cases/test1.txt`



```
[00001] id=0001 NEW
[00001] IDLE
[00002] id=0001 req=17 used=1 queue=1
[00003] id=0001 req=17 used=2 queue=1
[00004] id=0001 req=17 used=3 queue=2
[00005] id=0001 req=17 used=4 queue=2
[00006] id=0001 req=17 used=5 queue=2
[00007] id=0002 NEW
[00007] id=0001 req=17 used=6 queue=2
[00008] id=0002 req=1 used=1 queue=1
[00009] id=0001 req=17 used=7 queue=3
[00010] id=0001 req=17 used=8 queue=3
[00011] id=0001 req=17 used=9 queue=3
[00012] id=0001 req=17 used=10 queue=3
[00013] id=0003 NEW
[00013] id=0001 req=17 used=11 queue=3
[00014] id=0003 req=6 used=1 queue=1
[00015] id=0003 req=6 used=2 queue=1
[00016] id=0003 req=6 used=3 queue=2
[00017] id=0003 req=6 used=4 queue=2
[00018] id=0002 req=1 used=1 queue=1
[00019] id=0003 req=6 used=5 queue=2
[00020] id=0003 req=6 used=6 queue=2
[00021] id=0003 EXIT wt=1 tat=7
[00021] id=0001 req=17 used=12 queue=3
[00022] id=0001 req=17 used=13 queue=3
[00023] id=0001 req=17 used=14 queue=3
[00024] id=0002 req=1 used=1 queue=1
[00025] id=0002 EXIT wt=0 tat=3
[00025] BOOST
[00025] id=0001 req=17 used=15 queue=1
[00026] id=0001 req=17 used=16 queue=1
[00027] id=0001 req=17 used=17 queue=2
[00028] id=0001 EXIT wt=9 tat=26
[00028] IDLE
```

Over quantum time of 2 sec move to Q2 in the next tick

At tick 7 task 2 get in + task 1 also running

Done

task 3 wait 1 tick

1+6

0+1+1+1

Task 1 gets boosted to Q1 after spend 8 sec in Q3

9+17

Adding and Maintaining Tasks

Since this assignment works with a simulation of tasks rather than a real OS workload, we also simulate the notion of what it means for a task to be CPU-bound or I/O-bound. More precisely, it is possible that a task in our simulator might not, at some tick, be enqueued within the MLFQ as it does not currently have any CPU-tick requirement. This would be the same behavior as if that task were now blocked on some I/O.

Therefore, we cannot completely depend on the three global queues to store all information on tasks within our simulation. In `mlfq.c`, you are given a global array of `task_t` (the definition of this type is in `queue.h`) called `task_table`. It is initialized to the max number of tasks, so when a task arrives in the simulator, you do not need to create a new task (i.e. no `malloc` needed); you can just modify the entries within the array. When the simulator detects that CPU time is required for the task from the input file, then the `task_t` for the task must be properly enqueued into the MLFQ. If the task was scheduled and as a result has completed all of its required CPU ticks, then that task will not return to the MLFQ but it will still exist in the global `task_table`.

Task Metrics

Consider the following line from the provided output of the simulation of `cases/test1.txt`:

```
[00015] id=0003 req=6 used=2 queue=1
```

At tick 15, the simulator has scheduled task 3 which currently has a most-recent CPU-tick requirement of 6 ticks, and 2 ticks have already been scheduling during some previous ticks. Also, the task was retrieved from `queue_1` (i.e. the queue with quantum = 2).

The very next line in the output shows an important change:

```
[00016] id=0003 req=6 used=3 queue=2
```

Although the amount of CPU ticks used has gone up, the queue from which the task was scheduled has changed. That is, after completing tick 15, task 3 has used up the whole CPU quantum given to it. Since it still had more time left in its burst, according to the rules of the MLFQ, it had been enqueued after tick 15 to the next queue down, which is `queue_2` (i.e. the queue with quantum = 4).

Consider the following line of the sample output:

```
[00021] id=0003 EXIT wt=1 tat=7
```

This indicates that at tick 21, task 3 exited the system having spent one tick waiting in an MLFQ queue and where the task had a turnaround time of 7 ticks. There are three things to observe here:

- The turnaround time for a process in an OS is the sum of the ticks for which the process was scheduled the CPU plus the ticks for which it was waiting on an MLFQ queue. Note that turnaround time does not include any simulated I/O time.
- In order to keep track of the total number of CPU ticks granted to a task, you must correctly increment the field named `total_execution_time` for the task in its instance of `task_t` when the task is scheduled.
- In order to keep track of the total number of ticks for which a task is in the MLFQ, on each tick, your implementation must correctly increment the field named `total_wait_time` for all relevant tasks in the task table. A relevant task is one for which the `remaining_burst_time` is non-zero. That is, by definition, such a task will be in one of the MLFQ queues. You must write code within the `update_task_metrics()` function in order to update relevant tasks on a tick. Note that you do not need to traverse all the tasks in all the queues for this action. All tasks are already available for examination via the global `task_table` array.

Visualization of Expected Schedules

Given the complexity of this assignment, you may want to visualize the schedules expected from the various simulations in the text files. You will find one PDF per test file with a visualization of the schedule. Note that this visualization does not include all of the information that would appear within the simulator's output. However, the diagrams in the PDFs can be another resource that you can use to understand the problem as described in this assignment.

IDLE

It may be the case that at some tick, there are either no tasks in the task table, or the only tasks that are not yet terminated have `remaining_burst_time` equal to zero. By definition, this would mean there are no tasks in the MLFQ, and therefore no tasks can be dispatched to the CPU. For that tick, the CPU is said to be IDLE.

Preemption

The current task (the one in the variable `current_task`) may be preempted. If the scheduler discovers that there is a task on a higher-priority queue than the current task, the higher-priority task must preempt the current task. The current task is enqueued back onto the queue from which it was taken, and the higher-priority task becomes the current task.

- Example 1: Task A is taken from `queue_2` and is made the current task. In the next tick, Task B arrives in the `queue_1`. The scheduler must take Task A and enqueue it onto `queue_2` and make Task B the current task.
- Example 2: Task C is taken from `queue_1` and made the current task. In the next tick, Task D arrives in `queue_1`. Since Task C was taken from `queue_1`, and Task D is also in `queue_1`, there is no preemption and nothing needs to be done.

Boosting

The program must prevent tasks from being starved for CPU time. Starvation could occur if a task is relegated to the bottom queue, and then a large number of new short tasks are added. These new tasks could occupy the upper queues and run one after another, with the CPU never getting a chance to resume the task on the bottom queue.

To prevent this, you must implement a boost mechanic. The `boost()` function will be called just before the scheduler. It will move all the tasks from the bottom queue (`queue_3`) to the top queue (`queue_1`) then move all tasks from the middle queue (`queue_2`) to the top queue. Boosting, like adding or exiting tasks, takes no CPU ticks. This boost function should run periodically, as indicated by `BOOST_INTERVAL`, which is set to 25. That is, a boost should occur on tick 25, 50, 75, and so on.

If there is a current task, it should continue to be the current task, but its remaining time quantum should be reduced to 2 if it is currently greater than 2, or left alone if it is 2 or less. That is, it should be treated as if it had been dequeued from `queue_1` (its queue data field should be updated accordingly). Since the current task has its queue set to `queue_1`, there is no way for it to be preempted by any other task during the boost. Tasks that are neither on a queue, nor are the current task are not affected by the boost at all.

- Example 1: Current Task E was taken from `queue_3` and has a remaining quantum of 5. The boost occurs. Task E's queue variable is updated to 1, and its remaining quantum is set to 2.
- Example 2: Current Task F was taken from `queue_3` and has a remaining quantum of 1. The boost occurs. Task F's queue variable is updated to 1, and its remaining quantum remains at 1.
- Example 3: Task G is not the current task and is not on a queue - it exists only on the `task_table`. The boost occurs. None of Task G's variable values are changed. In particular, its `current_queue` variable is unchanged.

Completing the Code

To help guide your development of a solution to this assignment, there are many comments provided within `mlfq.c` that state some helpful assumptions and also indicate places in which to add functionality via `TO DO` comments. However, you may not add any additional source code files and you may not modify the code in `queue.c` or `queue.h`.