Computer Science 360 – Introduction to Operating Systems
Summer 2021

Assignment #3
Due: Tuesday, July 20, 11:55 pm via push to your Gitlab.csc repository

Programming Platform

For this assignment **your code must work on the Virtualbox/Vagrant configuration you provisioned for yourself in assignment #0.** You may already have access to your own Unix system (e.g., Ubuntu, Debian, macOS with MacPorts, etc.) yet I recommend you work as much as possible while with your CSC360 virtual machine. 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. (We cannot give marks for submissions of which it is said "It worked on Visual Studio!")

Individual work

This assignment is to be completed by each individual student (i.e., no group work). Naturally you will want to discuss aspects of the problem with fellow students, and such discussion is encouraged. However, sharing of code is strictly forbidden. If you are still unsure about what is permitted or have other questions regarding academic integrity, please direct them as soon as possible to the instructor. (Code-similarity tools will be run on submitted work.) Any fragments of code found on the web and used in your solution **must be properly cited** where it is used (i.e., citation in the form of a comment giving the source of code).

Goal of this assignment

1. Write a C program implementing a simulation of *round-robin CPU scheduling* that also makes use of a *multi-level feedback queue*.

Use of gitlab.csc.uvic.ca

Each student enrolled in the course has been assigned a Git repository at gitlab.csc.uvic.ca. For example, the student having Netlink ID `grogu` would have their CSC 360 repository at this location:

`https://gitlab.csc.uvic.ca/courses/2021051/CSC360/assignments/`**`grogu`**`/csc360-coursework`

Please form the address of your repository appropriately and perform a git clone in environment you have provisioned as part of assignment #0. You are also able to access this repository by going to `https://gitlab.csc.uvic.ca` (and use your Netlink username and password to log in at that page).

What is provided to you

There is some code provided to you for completing the assignment, and in order to simplify distribution we have chosen to add this code directly into your `gitlab.csc` repository for the course. **As of 9:00 pm on Tuesday, July 6th – at the latest –** you will find this code provided to you in a directory named `a3/` within your own repo.

Your work: writing `mlfq.c`

Unlike the previous assignments involving a significant amount of systems programming, in this one your work is to write a C implementation of a round-robin CPU-scheduler simulator which also uses the organization of multi-level feedback queue. *There is no threading or synchronization required to complete this assignment.*

In effect you will be implementing a tick-by-tick simulation of this single-core CPU scheduler for a set of CPU-bound and IO-bound tasks. We will use abstract *ticks* for time rather than milliseconds or microseconds.

A starter file is provided to you as `a3/mlfq.c` which already contains significant functionality intended to reduce your work. The file has, amongst other things:

- code needed to open a test-case file;
- a function named `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 ultimately provided to arguments to these `printf` 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.

Overview of the input into, and output from, the simulator* `a3/mlfq.c`*:

Your implementation of **`a3/mlfq.c`** will accept a single argument consisting of a text file with lines, the lines together containing 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
```

Therefore an input file is *a representation of a simulation case*, where each line represents one of three possible facts. For example, the following line contained within such a file:

```
13,3,0
```

indicates that at tick 13, task 3 has been created (i.e. the 0 means creation). A line in the file 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, there will appear a line in the file such as:

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

here indicating that at tick 21, task 3 will terminate once its remaining burst's CPU ticks are scheduled. (If no such CPU ticks are remaining, then the task would be said to terminate immediately.)

As mentioned earlier in this document, your simulator is to provide a tick-by-tick simulation of a CPU scheduler. Although we have still to describe the nature of this scheduler, you may want to look at the last two pages of this document where you will find the contents of `a3/cases/text1.txt` followed by the expected simulator output for this specific simulation case. You will see in this output (i.e. very last page of this document) 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 be explained 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 assignment description.)

Nature of the scheduling queue used by* `mlfq.c`*:

You are to implement a **multi-level feedback queue**, or **MLFQ**. An MLFQ is designed to ensure the tasks requiring quick responses (i.e. which are often characterized as having short CPU bursts) are scheduled before tasks which are more compute bound (i.e. which are characterized as having long – or at least longer – CPU bursts).

Your simulation will be of an MLFQ with three different queues: one with a quantum of 2, one with a quantum of 4, and one with a quantum of 8.

- When there is a CPU scheduling event, the queue corresponding to `q=2` is examined. If it is not empty, then the task at the front is selected to run and given a quantum of 2. Otherwise the queue corresponding to `q=4` is examined, and if it is not empty, then the task at the front of this queue selected to run and given a quantum of 4. Otherwise the queue corresponding to `q=8` is examined, with the task at the front of this queue selected to run and given a quantum of 8.

- If a task selected from either the `q=2` or `q=4` queues finishes its current burst within 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 the `q=2` or `q=4` queues has a burst that exceeds its quantum, then it is interrupted (as would be the case for any round-robin algorithm), and placed into the next queue with a larger quantum. That is, a task taken from `q=2` would be placed at the end of the `q=4` queue; a task taken from `q=4` would be placed at the end of the `q=8` queue.

- In a full implementation of an MLFQ, tasks in the queues with longer quantums would be permitted to move back into queues with shorter quantums. (This would reflect the nature of real tasks in that they tend to oscillate between being CPU-bound and IO-bound.) ***However, you will not implement aging in this assignment.*** Once a task enters the `q=8` queue, it is doomed to enter and exit this queue until the task is terminated.

In `a3/mlfq.c` the three queues are global variables named `queue_1`, `queue_2`, and `queue_3` (i.e. `q=2`, `q=4`, `q=8`).

You are given a queue implementation in the files `queue.c` and `queue.h`. Please read this code carefully. However, you are not to modify this code unless given express written permission by the instructor.

What is all this about CPU-bound and IO-bound tasks?

Given this assignment works with a *simulation* of tasks rather than a *real OS workload*, we must also simulate the notion of what is means for a task to be CPU-bound or IO-bound.

More precisely, it is perfectly possible that a task in our simulator might not, at some tick, be enqueued within MFLQ as it does not currently have any CPU-tick requirement. This would be precisely the same behavior as if that task were now blocked on some IO (i.e., IO-bound).

Given this is a possibility, we cannot depend completely on the three global queues to store all information on tasks within our simulation. Therefore a global `task_table` in `a3/mlfq.c` is provided to you. When you create a task, an instance of `task_t` for that new task must be added to this table. And when the simulator detects that there CPU time is required for the task (i.e., from the input file), then the `task_t` for the task must properly enqueued into the MLFQ. If it should happen 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 a line taken from the last page of this document (i.e. sample output from a simulation based on `a3/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, although 2 ticks of that have been scheduled during some previous ticks. Also the task was retrieved from queue 1 (i.e., the queue for which quantum `q=2`).

The very next line, however, shows an important change:

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

Although the amount of CPU ticks used (i.e. scheduled) has gone up, the queue from which the task was scheduled has changed. That is, after completing tick 15, task 3 had used up the whole CPU quantum given to it, and as it still had more time left in

its burst, and according to the rules of the MLFQ, it had been enqueued after tick 15 to the next queue down (i.e. the queue for which quantum `q=4`).

And note that the `Task_t` structure given to you in `a3/queue.h` has two fields, `burst_time` and `remaining_burst_time` which your implementation must correctly update and from which can be computed the proper values for `req` and `used`.

Let us consider one more line from the sample output given at the end of this assignment description:

```
[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 turn-around time of 7 ticks. There are three things to observe here:

- Technically speaking, 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 turn-around time does not include any simulated IO 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` (i.e. definition found in `a3/queue.h`) 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 `update_task_metrics` in order update relevant tasks on a tick; note that you need not traverse all of the tasks in all the queues for this action, i.e. all tasks are already available for examination via the global `task_table` array in `a3/mlfq.c`.

Some visualization of expected schedules

Given the complexity of this assignment, you may desire the assistance of some other way of visualizing the schedules expected from the various simulations in the text files (i.e. within the `a3/cases/` subdirectory.)

To that end 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 you can use to “wrap your mind” around the problem as stated for you in this assignment.

A word about `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 that can be dispatched to the CPU. For that tick, therefore, the CPU is said to be `IDLE`.

… And so now you must complete the code!

To help guide your development of a solution to this assignment, there are many comments provided within `a3/mlfq.c` that not only state some helpful simplifying assumptions, but also indicate places in which to add functionality via `TO DO` comments. However:

- You may not add any additional source code files without the express written permission of the course instructor.

- You must not modify the code in `a3/queue.c` or `a3/queue.h` – and if you think a modification is absolutely necessary, you must pose a question on RocketChat stating the problem as you see it and the reason why you think the code must be changed. A member of the teaching team will provide an answer.

What you must submit

- All submission is via a git push of your `gitlab.csc` respository. This therefore must include:
    - Your code for `a3/mlfq.c` corresponding to your work for this assignment within the `a3/` directory;
    - a file `a3/README.md` with details of your submitted solution.

If you simply add, commit, and push all your work in `a3/` then you will – in effect – have performed submission as requested.

Evaluation

Our grading scheme is relatively simple.

- "A" grade: An exceptional submission demonstrating creativity and initiative. The **`mlfq`** program runs without any problems, and **`README.md`** clearly describes the design and implementation of your solution.
- "B" grade: A submission completing the requirements of the assignment. The **`mlfq`** program runs without any problems, and **`README.md`** clearly describes the design and implementation of your solution.
- "C" grade: A submission completing most of the requirements of the assignment. The **`mlfq`** program runs with some problems or the **`README.md`** does not describe the design and implementation of your solution.
- "D" grade: A serious attempt at completing requirements for the assignment. The **`mlfq`** program runs with serious problems.
- "F" grade: Either no submission is given, or the submission represents very little work.

Content of `cases/text1.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
```

Output from a completed simulator based on `cases/text1.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] id=0001 req=17 used=15 queue=3
[00026] id=0001 req=17 used=16 queue=3
[00027] id=0001 req=17 used=17 queue=3
[00028] id=0001 EXIT wt=9 tat=26
[00028] IDLE
```