LOUISIANA STATE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

CSC 4103: Operating Systems
Prof. Aisha Ali-Gombe

Programming Assignment # 3: Multilevel Feedback Queue Scheduling

Assigned – March 6th, 2025 - Due Date: March 25th, 2025 @ noon
Instruction – Students are strongly encouraged to work with a PA buddy but can also to work on their own. Only one student can submit on behalf of their team
NO LATE SUBMISSIONS will be accepted

There are a variety of algorithms for process scheduling, and each has advantages and disadvantages. For this assignment you'll investigate one of the more complex (and powerful) scheduling algorithms, *Multi-level Feedback Queue Scheduling*. **Your solution must be written in C.**

First, you'll need a queue package. Feel free to write your own, but to save time, I suggest you use the `prioque.c` package I've provided on Moodle. Be sure to read the documentation in `prioque.h` and to test the priority queue package using `test-prioque.c`, to make sure you understand how it works.

Your task is to simulate a **four-level** multi-level feedback queue scheduler. Each queue in your scheduler will use round robin scheduling. The **first** level will have a small quantum to let I/O-bound processes get through quickly. The **second** level will have a medium quantum, the **third** level will have a larger quantum, etc. The four queue levels operate under a strict priority scheme, e.g., for a process in the second queue to execute, the first queue must be empty. When a process arrives in an upper-level queue while a process is executing in a lower-level queue, the lower priority process is preempted and remains in place in the same queue until it gets to execute again.

You should use the set of rules we discussed in class to determine which process executes:

<u>Rules:</u>

- New processes at placed at the end of the highest priority queue
- If Priority(A) > Priority(B), A is selected for execution
- If Priority(A) = Priority(B), use RR to schedule A and B

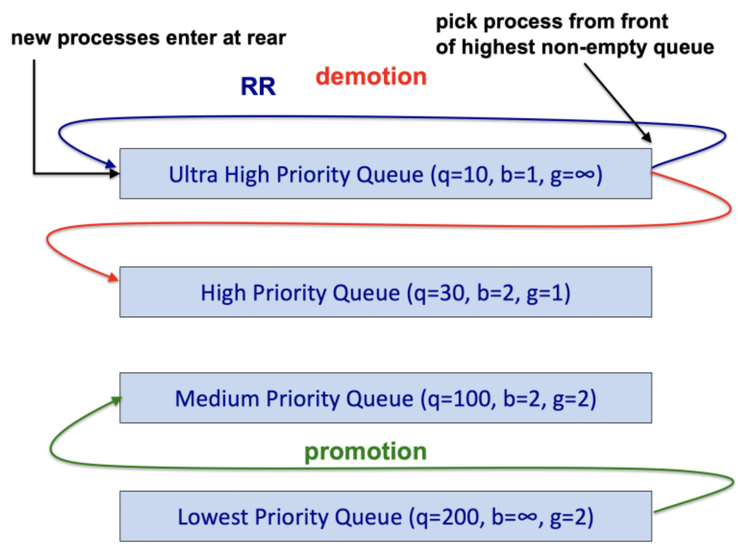
- If a higher priority process arrives, the currently executing process is preempted after the current clock tick (but stays in place in its current queue)
- If a process uses its entire quantum at a particular priority *b* times, its priority is reduced and it moves down one level
- If a process doesn't use its entire quantum at a particular priority *g* times, its priority is increased and it moves up one level

Notes on resetting *b* and *g*:

For promotion calculations, *g* can't be reset on each I/O, because the idea is that the process gets its full CPU need, does I/O, returns to ready queue, gets its full CPU need, etc., without exhausting the quantum, *g* times in a row, to be promoted. So the *g* can't be reset when the process does an I/O, or you can't track this.

For demotion calculations, doing an I/O resets *b*. The idea here is that burning through the entire quantum *b* times in a row before you do I/O means the process should be demoted. If the process does I/O before using the entire quantum, *b* is reset (i.e., it has "behaved" during this execution).

The following diagram provides concrete values for the quantum (*q*), demotion counter (*b*), and promotion counter (*g*) for each queue level:



A few more observations:

Your scheduler <u>simulates</u> MLFQS scheduling and obviously isn't part of a real operating system. A particularly unrealistic assumption is that the scheduler itself consumes no resources because complicated multi-level feedback queue scheduling can be expensive if the implementation isn't well thought out.

To keep time, your scheduler's high-level structure should look something like this:

```
read all input
clock=0
while (there is at least one unterminated process) loop
      if (processes should enter at current clock value) then
            enqueue these processes
      end if
      execute highest priority process that's ready to run for one tick
      make exit, I/O, demotion, or promotion decisions
      clock++
end loop
```

The "processes" your scheduler operates on aren't "real" processes. Instead, your schedule will read process specifications from standard input. These specifications describe compute and I/O behavior. Based on a set of process specifications, your scheduler will output scheduling decisions to standard output.

The format for input is:

TIME	**PID**	**RUN**	**I/O**	**REPEAT**
5	1000	8	20	5
200	1583	1000	10	1
1500	2120	5	20	10
1500	2120	200	30	2
2500	2450	200	100	3
3200	1060	7	20	5
3200	1060	500	50	10
3200	1060	7	20	10
4000	1201	2000	100	5
4000	1201	25	50	5
4000	1201	5	20	5

Each line of input contains information about one phase of the lifetime of a process. The TIME value is the time the process is created and placed in the highest priority ready queue. PID is the unique identifier for the process. RUN is the amount of time the process runs during this phase. I/O is the amount of time required to do I/O after running during this phase. REPEAT specifies how many times this RUN-I/O phase is repeated, **but the process shouldn't end on an I/O; one additional RUN period should be performed after the last I/O operation.** Finally, the "TIME", "PID", etc. labels are not included in the input!

Thus, a process' simulated execution looks like this in pseudocode:

```
while  (there's another (RUN, I/O, REPEAT) phase for process PID) loop
      for I in 1..REPEAT loop
            do compute for RUN time units
            do I/O for I/O time units
      end loop
      if (this is the last phase) then
            do compute for RUN time units
      end if
end loop
```

When your scheduler begins, the time should be 0. Whenever there is no process to schedule (all processes are doing I/O or no processes exist), a special process called the *null* process should execute. The scheduler should continue to increment its clock during the execution of the null process, waiting for another process to become ready. Your scheduler should exit when all processes from the input have been executed completely.

The required output format for your scheduler is described below. Please do NOT improvise--you must use the required format. All output should be directed to standard output.

When a process is created and enters the ready queue, a line like this should be generated:

`CREATE: Process 100 entered the ready queue at time 1000.`

When a process gets the CPU and enters the running state:

`RUN: Process 100 started execution from level 2 at time 1000; wants to execute for 43 ticks.`

…where "43 ticks" in this case is the time *remaining* before this process wants to do an I/O. Of course the process may not be allowed to run for 43 clock ticks in a row before being preempted.

When a process is placed into a queue (after being stripped of the CPU or completing I/O):

`QUEUED: Process 100 queued at level 2 at time 1000.`

where the level is 1, 2, 3, or 4. The preceding line will help someone looking at the behavior of your scheduler to determine when processes are being moved from higher to lower queues or vice versa.

When a process leaves the ready queues to perform I/O:

`I/O: Process 100 blocked for I/O at time 1000.`

When a process completes execution (there are no more phases of execution behavior specified):

`FINISHED:  Process 100 finished at time 1000.`

Finally, your scheduler should report the final clock time and total CPU usage of all processes (including the <<null>> process) scheduled when it exits. Use the following format:

```
Scheduler shutdown at time 85453.

Total CPU usage for all processes scheduled:

Process <<null>>:   13934 time units.
Process 100:        18843 time units.
Process 200:         1000 time units.
     …
     …
```

Use the standard `classes.csc.lsu.edu` submission procedures, as for programming assignment # 1. The name of this assignment is prog3.

Documentation quality, code quality, and of course the degree to which your solution works properly (including adherence to input/output specifications) will all be considered when assigning a grade.

Your submitted solution **must** compile cleanly to get partial credit.

This is a complicated program. Get started early and test thoroughly.