Multilevel Feedback Queues (MLFQ)

The commands you must handle are repeated from the previous project.

In this project you will build a Multilevel Feedback Queue simulated scheduler in userland. Your scheduler will read a data file containing information simulating timer interrupts, job arrivals, job terminations, etc. Your output will be graded against known good results. Any deviation (even by a single space) may be counted as wrong, so **follow the specification completely**.

You will be able to run tests against my grading script to debug any textual differences.

Requirements Embedded in the Specification

To encourage you to read this specification thoroughly, certain project requirements are embedded *inline* in this document. Failure to implement requirements will result in points off.

Choice of Language

You may use C or C++. I encourage you to use C++ to take advantage of `deque`.

Use of Precise Integer Types

Please use `int32_t` rather than `int`, etc. This is not a requirement but I do encourage it.

MLFQ in OSTEP

Refer to chapter 8 of OSTEP. Reminder (adapted for this project):

- Rule 1: If Priority(A) > Priority(B), A runs (B doesn't).
- Rule 2: If Priority(A) = Priority(B), A & B run in round-robin fashion.
- Rule 3: When a job enters the system, it is placed at the highest priority (the topmost queue).
- Rule 4: Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).
- Rule 5: After some time period S, move all the jobs in the system to the topmost queue.

In the above, references to Priority equates to which queue a process is on.

Command Line Options

Your program will take just one argument - the name of the data file to execute.

It is an error if the file is not given *or* cannot be opened for reading. If this error occurs, you must emit an explanatory message using `cerr`. The error message strings are:

- "Missing input file name." and
- "Input file failed to open."

If these errors occur, your program must terminate with a return code of 1. You can check the return code of a program using:

```
$ echo $?
```

Platform

This project can be coded natively on the Mac, you don't need to use a Linux VM.

This project must be coded under WSL on Windows or on a Linux machine.

Input

Your program will take a data file as its command line argument. It contains *comma separated value* lines with the following syntax:

opcode	argument 1	argument 2	meaning
newjob	NAME		A new job arrives with the given NAME *
finish			The currently running job has terminated - it is an error if the system is idle
interrupt			A timer interrupt has occurred - the currently running job's quantum is over

opcode	argument 1	argument 2	meaning
block			The currently running job has become blocked
unblock	NAME		The named job becomes unblocked - it is an error if it was not blocked
runnable			Print information about the jobs in the runnable queue
running			Print information about the currently running job
blocked			Print information about the jobs on the blocked queue
epoch			An Epoch has elapsed. Process as per MLFQ algorithm - this is not in your previous project

* Note that this project's "newjob" does not take a PRIORITY. This is different from your previous project. A new job is assigned to the topmost queue as per the MLFQ algorithm.

My test input is guaranteed to have correct syntax so you do not have to check for errors.

Note that if you create your own tests (you should), you can add comments by appending a comma plus your comments at the end of an input line. Don't put commas in your comments. See this for an example.

For example:

```
newjob,A,A runs.
newjob,B
newjob,C
interrupt,A goes to 1. B runs.
```

Lines 1 and 4 have comments.

NO MAGIC NUMBERS

You must not use any magic numbers. Any numeric literals should be `const` variables with **good descriptive names**. The exceptions to this could be 0 and 1.

The Queues

PAY ATTENTION TO THIS!

There shall be a total of 4 queues named 0 through 3.

Each queue is managed with Round Robin.

The quantum is 10ms. The milliseconds as units is, of course, meaningless as this is a simulation.

The Epoch is defined as 1000ms. But once again, for the purpose of this simulation, time is kind of meaningless.

Operation

newjob

A new job is entered into the system. Its name is given. Assume all job names are unique. A new job's arrival does not cause a rescheduling unless the system was idle. Your response:

```
New job: C added.
```

finish

The currently running job has completed and should be removed from the system.

```
Job: B completed.
```

If the system is idle, it is an error.

Note that errors of this kind are emitted to `cout` and not `cerr`.

```
Error. System is idle.
```

interrupt

The currently running task has completed its quantum. Adjust your bookkeeping. The scheduler needs to run again.

In a real system, this would happen when a process has consumed 10ms of CPU time. Another process would be given a turn.

```
Job: C scheduled.
```

It is an error if `interrupt` is received when the system is idle.

```
Error. System is idle.
```

block

The currently running task has become blocked. Perhaps it is asking for an I/O, for example.

```
Job: A blocked.
```

It is an error if the system is idle, since no process was running that could become blocked.

```
Error. System is idle.
```

unblock

The named job has become unblocked. Return it to the queue it came from and reschedule.

```
Job: A has unblocked.
```

It is an error if the named job was not blocked.

```
Error. Job: C not blocked.
```

Unblocked jobs return to the runnables on the queue on which they were last found as per MLFQ. The scheduler to ensure Rule 1 is maintained.

runnable

The runnables, if any, are listed. Note this command's output is quite different from the previous project. Example:

```
Runnables:
NAME    QUEUE
B       0
A       0
C       2
Z       3
G       3
T       3
```

These must be listed in the order they would be scheduled.

running

The running task is described (if system is not idle). Example:

```
Running:
NAME    QUEUE
H       0
```

blocked

The blocked tasks are listed, if any. Example:

```
Blocked:
NAME    QUEUE
I       0
K       2
J       2
```

epoch

An epoch has expired when you receive an "epoch" command. Handle this as per the MLFQ algorithm. Note that "epoch" does **not** cause a rescheduling. That is, a process running when you receive an "epoch" will remain running but will also be marked as coming from the initial highest priority queue.

Here is a sample output from "epoch":

```
Job: C lifted up.
Job: B lifted up.
Job: A lifted up.
```

"epoch" should produce a line for every process in the system.

Picking the Next Task to Schedule

When a rescheduling is required, pick the next task to run according to the MLFQ algorithm.

```
Job: C scheduled.
```

Running the tests

The results of the stride schedule are deterministic. Given the same input, you will get the same output. Therefore, automatic testing is possible.

I provide a `bash` script for testing purposes. The script takes two command line options.

option	required?	argument	purpose
a	no	progname	specifies name of your executable - defaults to ./a.out
i	yes	testroot	specifies the root name of the test to run

The folder containing the test script is meant to contain the folder `tests` where the test data is actually stored. So, if you specify `-i test1`, the actual data file will resolve to `tests/test1.input.txt`.

You will be given only *some* of the tests I will use for grading. Some, I hold in reserve to test corner cases, etc. If you pass all the tests you are given, you have a reasonable change of a high score but this is not a certainty.

Feel free to read the source code of the test script to see an example of `bash` scripting.

What to Turn In

See here.

Setting Expectations

For the purpose of setting your expectations only, my solution is 295 lines long including comments, asserts, and some calls to `cout` spread over several lines.

As usual, the specification is longer than the solution :(.

These are the includes from my solution, provided for your interest purposes only. You don't have to follow this example:

```
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include <deque>
#include <vector>
# include <iomanip>
```

Remember:

- no magic numbers.
- Constants should be `const`.
- No warnings! I will be compiling your code with:

```
g++ -Wall --pedantic -g -std=c++11 your_code.cpp
```

You might want to do the same.