National University of Singapore
School of Computing

CS2106 **Laboratory 6** Semester 1 11/12

Deadline

14 October, 2011, Friday, 10:00pm.

Platform

This exercise can be done using either Linux or sunfire.

Submission

Your working directory should be named `lab06-A000000X` where `A000000X` is your matriculation number.

Create a PDF file called `lab06-A000000X.pdf` (where `A000000X` is your matriculation number), that summarizes your findings in comparing round-robin and MLFQ, and put it under your working directory.

Create a tar ball named `lab06-A000000X.tar` containing your version of the source code by:

```
tar cvf lab06-A000000X.tar lab06-A000000X/*
```

and submit the tar file into the IVLE workbin named Lab 6.

Marking Scheme

This lab exercise is worth 15 marks.

0.1 marks will be deducted per minute after the deadline for late submission, and 3 marks will be deducted for EACH file and directory naming violation and format violation.

You will implement a simulation of a Multi-Level Feedback Queue (MLFQ) scheduler. A code skeleton has been given to you. You may download it from the URL `http://www.comp.nus.edu.sg/~cs2106/lab06-A000000X.tar`. Untar the downloaded file using:

```
tar xvf lab06-A000000X.tar
```

A subdirectory `lab06-A000000X` is now created. As usual, rename the directory by replacing the string "A000000X" with your matriculation number (DO IT NOW before you forget. We are ruthless in deducting marks if the directory is not named properly). Inside the directory are a `Makefile` and a set of files. You can compile the code using the `Makefile` provided.

Simulating CPU Scheduler

During class, tutorial, and midterm, we have been simulating the CPU scheduler on paper by hand, which is fine for a small example and simple algorithms, but it would be too tedious for large number of jobs and more complex scheduler, such as MLFQ.

In this lab, we will simulate CPU scheduling algorithms and study the effect of different algorithms on the response time of jobs.

You have two tasks in the lab. First, you need to complete the MLFQ scheduler in the simulation. Second, you need to compare the performance of RR and MLFQ, and report on the relative performance of the schedulers.

Skeleton Code

The skeleton code provided for you consists of three main components:

- A discrete event simulator, which is responsible for managing what event happens when. The simulator calls upon the job manager to create/terminate/block/preempt/run jobs at appropriate time. The simulator is implemented in `event.c` and `simulator.c`.
- A job manager, which is responsible for managing all jobs in the system, including creating, terminating, blocking, preempting, and running the jobs. The job manager calls upon the scheduler to decide what jobs to run. The job manager is implemented in `job.c`.
- A scheduler keeps track of jobs that are ready to run and select one to run when requested by the job manager. Scheduler implementations can be found in `job_queue.c`, `sched_rr.c` and `sched_mlfq.c`.

There are a few other basic key structures in the given code:

- A job represents a job in the system, and consists a sequence of interleaving CPU burst and I/O burst (`job_t` is defined in `job.h`). A job is uniquely identified through a job ID, and is maintained by the job manager.
- An event (a, j) consists of an *action* a on a *job* j, where action can be one of create, preempt, block, ready, terminate. (`event_t` is defined in `event.h`).

There are two queues, derived from a queue implementation from your Lab 2.

- A job queue is a FIFO queue of job IDs (see `job_queue.c`) and is used by the scheduler.
- An event queue is a FIFO queue of events and is used by the simulator.

The simulator organizes all events in the system into a event calender (see `event.h` and `event.c`), which is basically an array of event queues indexed by the time the event is supposed to occur.

The event, event calendar, and simulator are structures related to simulating the scheduler only and does NOT work the same way in a real OS.

The job, job queue, and job manager are simplified versions of the mechanisms inside an OS.

The scheduler (see `sched.h`) is a generic scheduler that consists of a series of function pointers, pointing to the actual function that do the work. This design allows very generic code to be written, and by changing the function pointers, we can change the schedulers. The scheduler is instantiated to either the round robin scheduler or the MLFQ scheduler in `main.c`.

The round-robin scheduler (see `sched_rr.c`) has been implemented for you. The MLFQ scheduler's implementation (in `sched_mlfq.c`), however, is incomplete.

Assumptions

There are several hardcoded values in the skeleton code.

Our simulation runs only for 5000 time steps. But this can be easily changed in `main.c`. Events happen after 5000 time steps are not simulated.

We also hardcoded the minimum time quantum of the scheduler to be 2 time unit. Again, this can be changed easily in `main.c`.

Jobs Input

Since we are not simulating the complete OS and are not actually executing jobs, we need a way to simulate the arrival of jobs and the CPU and I/O bursts of jobs. To do so, we specify all jobs arriving in the system in a text file, which we will pass to the scheduler from the command line.

The text file has the following format. The first line in the text file consists of one integer, which is the total number of jobs N to arrive in the system. The next N lines describes the N jobs, one line per job.

The first integer of each of this line is the job ID, ranged from 0 to $N - 1$. The second integer is the arrival time, which is an integer, indicating the time slot when the job will be created. The third integer is the number of bursts k in this job. The number of bursts must be odd, since a job always starts with a CPU burst and ends with a CPU burst. The rest of the line must consists of k integers, each indicating the burst length of the corresponding CPU or I/O burst.

The following shows a simple example of a job specification with two jobs, Job 0 and Job 1, arriving at time 1 and time 2 respectively. Job 0 is a completely CPU-bound job with a single burst of length 9 time units. Job 1 consists of a CPU burst of length 3 unit time, followed by an I/O burst of 1 unit time, and a CPU burst of 2 unit time.

```
2
0 1 1 9
1 2 3 3 1 2
```

The skeleton code given assumes that the input file is in correct format. If you do handcraft your own job specification, check carefully to make sure that it is correct before you feed it into the simulator.

To help you with obtaining a large job specification file, a job generator has been given to you (`job_generator.c`). This generator generates random I/O bound jobs and CPU bound jobs. You should read the code – it is pretty self-explanatory.

Performance Metrics

The job manager keeps tracks of various performance metrics related to job scheduling, including turnaround time, waiting time, and response time.

The definitions of turnaround time and waiting time have been discussed in class and are unambiguous. As for the definition of response time, we are adopting one which measure the time interval between the time when a job's is ready after I/O (e.g., after receiving a keystroke) and its subsequent execution (e.g., response to keystroke). For a job with multiple I/O bursts, we will have multiple such intervals. We calculate the average response time of a job as the mean of all such intervals (including the interval between arrival and first run) over the lifetime of a job.

Output

The simulator, after simulating all jobs, print the performance metrics of each job onto standard output. There is one line per job, and each line consists of four numbers (in order): the job ID, total waiting time of job, average response time, turnaround time. The last line prints the average of the waiting time, response time, turnaround time over all jobs.

Example

Suppose the sample input above is stored in a file called `JOBS`. The following shows example runs of the program.

```
./sched JOBS rr
0        5        0.00        14
1        6        1.00        12
AVG        5.50        0.50        13.00
./sched JOBS mlfq
0        3        0.00        12
1        7        1.50        13
AVG        5.00        0.75        12.50
```

Your Tasks

(10 marks) Your first task in this exercise is to complete the implementation of MLFQ scheduler with four levels of priority. To do this, you would need to first understand the code given. It is essential that you understand at least the implementation of job manager in `job.c`. Understanding fully the implementation of the round robin scheduler (`sched_rr.c`) helps.

Note that as we discussed in the lecture, there are different varations of MLFQ. What is needed for the first task is described in the file `sched_mlfq.c`, which you are expected to complete. Your code essentially should maintain the internal structures of MLFQ (the queues) and update the following job properties: (i) the assigned priority (`priority`), (ii) time quantum assigned (`max_time_quantum`), which is fixed and only changes when the priority level changes,

and (iii) time quantum remaining (`time_left`), which decreases after a job runs and is reset to full time quantum when the priority level changes.

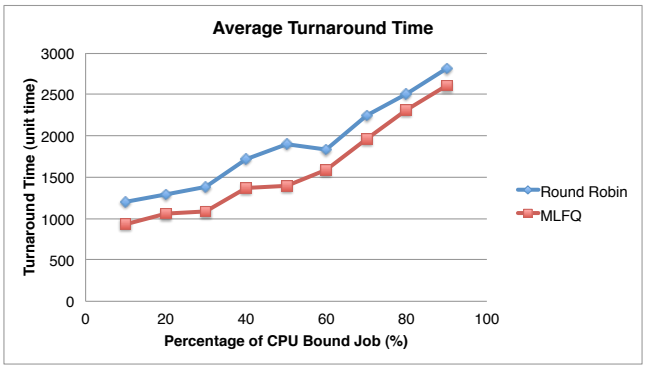
(5 marks) Your second task is compare between RR and MLFQ, in terms of the waiting time, response time, and turnaround time of jobs, and write down your observation.

The Makefile contains the script to (i) automatically generate a job configuration file with 100 jobs, with different mix of CPU bounds and I/O bounds jobs (with percentage of CPU-bound jobs ranges from 10% to 90%); (ii) feed it to the simulator for MLFQ and RR schedulers, and (iii) store the output from the simulator into a file named `xx.yy.out` where `xx` is name of the scheduler and `yy` is the percentage of the CPU-bound jobs.

To run the script above, use the command `make compare`.

Your should analyze the output stored in the *.out files. We are only interested in the last line (the average values over all jobs) of each file. To make your job easier, you can either comment out the appropriate code in job.c so that only the average is printed, or use the command `tail` with option `-1` to print out the last line of each file.

Now, use your favourite chart plotting tool to plot three charts, one for each of performance metric. Each chart should contain two data series, one for RR and one for MLFQ. The x-axis should be the percentage of CPU-bound jobs, while the y-axis is the performance metric. A sample chart (drawn with Microsoft Excel's Straight Marked Scatter) can be found in the figure below. Note that the sample chart below are plotted with random values; your charts should look different. Please remember to label your data series and your axes.



Next, look at the charts that you obtained and interpret the trend that you see. In particular, explain (i) how each performance metric changes as the number of CPU bound jobs increases, and (ii) the relative performances of RR versus MLFQ.

Put your charts and your interpretation of results in a PDF file named `lab06-A000000X.pdf`. Include the PDF file under your working directory (`lab06-A000000X`) and submit together with the tar ball. Remember, `A000000X` should be your matriculation number.

We do not expect anything more than ONE page (10 point font, single line spacing) including the charts.

(Not Graded) Now that you have an implementation of a MLFQ, it is now easy to compare different variants of MLFQ, such as one that does not increase the priority of a job that blocks, one that assigns same time quantum to all level of priorities, or one that assigns longer time quantum to jobs at higher level of priorities. You can even compare MLFQ with Linux's version of scheduler.

THE END