CS 143A/CSE 104 - Principles of Operating Systems Project: CPU Scheduling Simulator

Kyle Benson

February 14, 2014

1 Introduction

In this project, you will implement the scheduling algorithms we discussed in class and perform some analysis on them. You are given a framework with a nice GUI to help visualize what is happening with the algorithms and aid in your implementing/debugging. All you have to do is implement the required algorithms and then run them with the given simulation parameters.

Please let me know if you encounter any problems that you believe to be a bug in this code. I have tested most of it, but could have easily missed something subtle.

2 Interacting with the Simulator

This section describes how to define simulation parameters and run the system.

2.1 Running the Simulator

We have two options for how to run the simulator: through the `Makefile` or with Eclipse.

If you choose Eclipse, you should be able to simply import the `CPUSchedulingSimulator` folder, located in the `CPUSchedulingSimulator.zip` file found on the course website next to this document, as an "Existing project into workspace". Run the `MainApp` file to open the simulator GUI. If you have trouble with Eclipse, try adding the `.jar` files located in `src/lib` to your build path.

If you choose to use the `Makefile` (on a Unix-like system), navigate to the `CPUSchedulingSimulator` folder. Run `make classes` to compile the source code. As the output directs you, run `make run` to run the program.

When the simulator window opens, select your parameters (described below) and press the *Play* button to begin. Press it again to pause the simulator.

2.2 Options

The following options are found in the drop-down menu labeled `Options`.

2.2.1 Choosing Algorithms

From the options menu, you may select which of the available algorithms will handle scheduling. You can switch between them while the simulator is running.

2.2.2 Speed

You may also select the frames-per-second (FPS) of the simulator. This allows you to slow the simulator down to view finer details of your algorithm, or speed it up to complete the simulation faster.

2.2.3 Preemptive

For processes implementing the `OptionallyPreemptiveSchedulingAlgorithm` interface, described later, you may set their preemptive value to true (checked) or false (unchecked).

2.2.4 Show Hidden

This displays all of the processes, including those that have not yet arrived.

2.3 Quantum

At the bottom of the screen, next to the Play/Pause button, is a text field where you enter the quantum (in CPU cycles) for use with the Round Robin algorithm. You must choose the quantum value (by entering it in the box) *before* you select the algorithm from the drop-down menu.

2.4 Generating Processes

When you run the simulator, it will by default generate 50 processes with:

- CPU burst times between 1 and 99
- Delay times between 0 and 69
- Priorities between 0 and 9

The delay is the how long after the previous process arrival does this process arrive. So if P1 arrives at 12 and P2 arrives at 22, P2 has a delay of 10. Note that this means the first process may not arrive at time 0. Make sure you account for this if necessary.

You also have the option of loading an input file that defines all of the processes. This is done through selecting `File` → `Open Data Source...`. These files have the following format for each line:

```
burst delay priority
```

where each of these is an integer text value and is separated by a space, with no leading or trailing spaces. I have not tested exceeding the bounds specified above for these values, so do so at your own risk!

3 Programming

This section describes the scheduling algorithms that you will implement. Each section (including the two extra credit ones) is worth **20 points**. I have included skeleton code for each algorithm, so you only need to fill in the specified functions, adding additional members and functions as necessary.

IMPORTANT: replace my name with all of the full names of each of your group members, as well as their student ID numbers and your group number (assigned based on what slot you sign up for. See Section 5).

Note that working `.class` files for each algorithm are included with the source code. Feel free to keep a separate copy of the project and run it (without compiling) as described in Section 2.1 to compare your algorithm with the solution. Note: Eclipse may try to build the source code and overwrite the class files, so you may need to `touch` them so that it thinks they are up-to-date.

The most important source code file, reproduced on the next page for your convenience, is the `SchedulingAlgorithm.java` interface that defines all of the methods that your scheduling algorithm classes must implement. This allows you to write any number of algorithm classes implementing this interface and have the GUI automatically populate its algorithm list and properly interact with each of them.

```
/** SchedulingAlgorithm.java
 *
 * A scheduling algorithm to be used with Jim Weller's simulator.
 *
 * Adapted for CS 143A - Principles of Operating Systems
 *
 * @author: Kyle Benson
 * Winter 2013
 *
 */
package com.jimweller.cpuscheduler;

public interface SchedulingAlgorithm {
    /** Add the new job to the correct queue.*/
    public void addJob(Process p);

    /** Returns true if the job was present and was removed. */
    public boolean removeJob(Process p);

    /** Returns the next process that should be run by the CPU, null if none available.*
    public Process getNextJob(long currentTime);

    /** Returns true if the current job is finished or there is no such job. */
    public boolean isJobFinished();

    /** Return a human-readable name for the algorithm. */
    public String getName();

    /** Transfer all the jobs in the queue of a SchedulingAlgorithm to another, such as
        when switching to another algorithm in the GUI */
    public void transferJobsTo(SchedulingAlgorithm otherAlg);
}
```

The second most important source code file is `BaseSchedulingAlgorithm.java`. Extend this class in each of yours (already done in the provided skeleton code classes) in order to inherit some of the methods defined in this class and avoid rewriting them. In particular, note the `isJobFinished()` function and the `activeJob` member, which are useful for keeping track of the currently running job and determining whether to preempt it or not.

The third most important source code file is `Process.java`. Feel free to use any of the public functions in the latter half of the source file (from `getResponseTime()` onwards). In particular, note the `getBurstTime()` function, which returns the remaining CPU burst time of the specified Process. Some of the other functions will likely prove useful for the extra credit algorithm, if you choose to implement it.

I recommend first going through the code and inserting return statements as necessary to make it at least compile. That way you can work on one algorithm at a time, compiling and running it to check your progress.

3.1 First-come First-served

A simple FCFS algorithm. Don't worry about breaking ties.

3.2 Single-queue Priority

This algorithm should run the job with the highest priority (lowest priority number). It should implement the `OptionallyPreemptiveSchedulingAlgorithm` interface and preempt currently running processes only if this value is set to true.

3.3 Shortest Job First

This algorithm should run the job with the *shortest remaining time* first. Similar to the single-queue priority algorithm, this should implement the `OptionallyPreemptiveSchedulingAlgorithm` interface and preempt currently running processes only if this value is set to true.

Hint: if you design your single-queue priority algorithm properly, you should be able to implement this algorithm very easily by simply extending `PrioritySchedulingAlgorithm` and overriding a single helper method that you wrote (you *did* write a helper method, right?).

3.4 Round Robin

I recommend doing this algorithm last, as it is by far the most complicated. You must maintain state between each call to `getNextJob(int currentTime)`, which can be difficult if you are not careful. Use a default time quantum of 10. Note that the skeleton code includes some methods for setting/getting the current quantum. Don't modify them as they are correctly interacting with the GUI currently.

Carefully set up your data structures for managing the algorithms. Remember that you should place newly arrived processes and ones being switched out at *the end of the waiting queue.*

3.5 Extra Credit 1: Multilevel Priority

In this algorithm, you will implement put jobs in three different queues, depending on their priority:

- Processes with priorities 0-3 will go in queue 1
- Processes with priorities 4-6 will go in queue 2
- Processes with priorities 7-9 will go in queue 3

For each queue, the next job to run will be picked in the following manner:

- **Queue 1:** Round Robin with a default quantum of 10
- **Queue 2:** Round Robin with a quantum of twice whatever Queue 1's current quantum is
- **Queue 3:** First-come first-served

If there are jobs available in Queue 1, pick from that according to its scheduling algorithm. If not, try picking from Queue 2. If it is also empty, pick from Queue 3.

To get full credit on this portion, you *must* use good object-oriented programming techniques. Specifically, you should be using the other algorithm classes that you implemented rather than copying and pasting code.

Similar to the single-queue priority algorithm, this should implement the `OptionallyPreemptiveSchedulingAlgorith` interface and preempt currently running processes only if this value is set to true.

Note: if you choose the Makefile option and implement this algorithm, you will need to add the Java file name to the `SCHEDULING_ALGORITHMS` variable at the top of the Makefile.

Note: have this algorithm extend the `RoundRobinSchedulingAlgorithm` that you already implemented so that you can change the value for quantum through the simulator GUI.

3.6 Extra Credit 2: Memory Constraints

This is not an actual scheduling algorithm, but is rather an extension to the other scheduling algorithms. To receive these extra credit points, you must extend *all* of the scheduling algorithms. For this portion, add a total memory constraint to your system and an associated memory usage for each process. You can set the total system memory any way you want, but doing so through the GUI (similar to the way the time quantum is done) is preferred. For the memory usage of each process, you may add an additional field to the input file. The amount of currently available system memory, total system memory, and the memory requirements for each process should be displayed in the GUI somewhere.

Your scheduling algorithms should only choose from processes whose memory requirements can be met by the system's total constraints. You do not need to optimize for memory usage or guarantee any form of fairness (these are non-trivial topics), but simply respect the given constraints on the system. This means that a process requiring more memory than is currently available will not be able to run until that much memory becomes available. When a process is run for the first time, deduct its

memory requirement from the total available. When a process completes its execution, add its memory requirement back to the total available *before* choosing the next process to run. How to admit processes for consideration and implement this ready queue is entirely up to you. This would be a good place for practicing with moving processes between queues (new, ready, running, etc.).

This will not be trivial as you will have to modify code outside the scheduling algorithms, such as the classes representing Processes, etc. The description of this portion of the assignment is intentionally vague: the goal here is for you to gain experience digging deeper into the source code of a system that isn't fully documented and extending it to support a feature that wasn't planned for in the original design. You will all likely end up with many different solutions to this problem, several of which will be "correct". As before, you must follow good software engineering practices to receive full credit. This includes adding the necessary attributes and functions to the `SchedulingAlgorithm` interface (or other more abstract classes) whenever possible, rather than copy/pasting code across each of concrete classes you implemented. You will be asked to document and explain your final design during the project demo, so be prepared to answer questions about why you made certain decisions in your architecture.

4 Analysis

To further your understanding of how these different scheduling algorithms compare and why some might be used over others, you will perform some in-depth analysis of their performance over a variety of simulation parameters. To do this, you will first write a small script in whatever language you choose (scripting language recommended) to generate process data files, as described in Section 2.4. This should be very easy and you should make it easy to specify the parameters that you are asked to vary so that you can quickly create new process files. Generate 20-100 processes for the analysis in this section. The more processes, the more accurate your results, but the harder to debug with the GUI.

Each portion of this section asks you to vary the average job burst time and compare the results of each. To accomplish this, generate your values from a *Normal Distribution*, where you set the mean to be the value requested and the standard deviation to be one sixth of the range of possible values. If your chosen library asks for variance instead of standard deviation, just convert it to standard deviation and follow the above instructions. If this paragraph looks like Martian-speak to you, go brush up on your probability on Wikipedia, though you shouldn't really have to for the purposes of this assignment.

To save the statistics after completing a simulation, select `File` → `Save Statistics...`.

For each of the algorithms, collect the results of the given parameters, import them into Excel (or whatever), and create *well-labeled* graphs for the turnaround and waiting times (y-axis) with respect to the given parameter. If the priority isn't specified, as well as for the delay time, use the distribution described in 2.4.

4.1 First-come First-served

Run with average CPU bursts of 20, 40, 60, and 80. How does the average burst size affect turnaround time and waiting time? Is this result intuitive? Explain.

4.2 Single-queue Priority

Run with average CPU bursts of 30 and 70 and average priorities of 2 and 7 (4 combinations). Based on your results, which would you expect to have a higher turnaround time: job priorities being proportional to their CPU burst, or inversely proportional? What about for waiting time?

Repeat the experiments, plot the charts, and answer the questions for both preemptive and non-preemptive versions. Bar charts should work fine for this problem.

4.3 Shortest Job First

Run with average CPU bursts of 20, 40, 60, and 80. How does the average burst size affect turnaround time and waiting time? Is this result intuitive? Explain.

Repeat the experiments, plot the charts, and answer the questions for both preemptive and non-preemptive versions. Which results in a better turnaround time and why do you think that is the case?

4.4 Round Robin

Run with average CPU bursts of 20, 50, and 80 and quantums of 5, 10, 20. Is it better to use larger quantums with longer-running jobs? Explain.

4.5 Extra Credit 1: Multilevel Priority

Tweak your policies on each of the 3 queues by choosing different algorithms, different quantums, choosing preemptive vs. non-preemptive, etc. What combinations did you find to perform best with the default job generator built into the simulator? Feel free to share your high scores with others on Piazza, but do not post the combination of algorithms and parameters you used until after the project demos.

4.6 Extra Credit 2: Memory Constraints

Set your simulator to have 100 units (your choosing) of memory. Run each of the above scheduling algorithms on jobs with an average memory need of 10, 20, and 40. How do the algorithms perform for different memory needs? Which algorithms perform best?

Now set the simulator to have 200 units of memory. Run the scheduling algorithms again and comment on the differences with more total available memory.

Try varying CPU bursts as well and discuss the results. Which scheduling algorithms perform better with higher/lower certain CPU burst values?

5 Demonstrations

You will demonstrate your code to the class staff during finals week. We will post a sign-up sheet as the time nears for each group to sign up for a ten-minute block. We may ask *any* group member about *any* piece of code, so make sure that *each* member knows *all* of the written code and results from analysis in detail!

6 Bugs and potential pitfalls

The simulator is not perfect and contains a few bugs that have been noted in the past. Please post to Piazza if you find any strange behavior so we can debug and resolve issues as a group. Patches to fix these bugs are always welcome and may be rewarded with extra credit!

1. It's easy to mess up the quantum timer for the Round Robin algorithm: carefully check (count) that it actually counts down for the number of cycles you expect it to before your demo! In particular, note that a process runs for one cycle as soon as it turns red to represent it being selected next by the scheduling algorithm. You will lose points if it isn't perfect!

2. Switching scheduling algorithms: this causes some issues in the GUI such as resetting the preemption flag without updating the GUI's checkbox. You would have to uncheck/recheck it in that case. It is often best to close the whole GUI and re-open it when switching to a new algorithm.

3. Resetting/selecting new data source: Try re-selecting the scheduling algorithm after selecting a new data source if you are having problems.

4. We had permission issues in Windows (pre-8) that required putting the input file in the `data` folder and saving the output statistics to it as well.

5. A potential bug exists with Mac users. JRE 1.7 on Mac did not work with scanning the algorithms in Winter 2013.

7 Acknowledgements

Thanks to Jim Weller for providing the framework and GUI for the scheduling simulator.