**A Simple Scheduler**

This is part 1 of a two part assignment in which you will implement a simulation of a process scheduler using Java threads.

The simulation will consist of:

* A thread simulating the OS kernel (and will be an instance of the SystemSimulator class)
* A set of threads simulating the processes managed by the kernel. Each thread will be an instance of the Job class, and will simulate a single process.
* Each Job will contain an instance of a class implementing the JobWorkable interface. The execution of this object will simulate the work done by the process it is simulating.
* At least one factory that generates JobWorkables. [I'm providing you one example, WorkFactory, but you may want a factory that is an instance of an extension of that class.]
* A thread simulating users periodically submitting new processes to the kernel to be executed (this will be an instance of the Submittor class). An input file controls how often this thread will submit a new process to the kernel, and the duration of that process's simulated CPU burst.

In this assignment, you will implement a simple first-come, first-serve (FCFS) scheduler for (simulated) processes that do no (simulated) I/O. In other words, each process will consist of a single (simulated) CPU burst. Instances of the class named Job will be used to simulate the processes. Job will be a subclass of Thread. To test your code I will provide my own factory to your Submittor to see how well your scheduler executes my simulated processes. The end result should be a Gannt chart based on the actual running times of the threads.

**Mutual Exclusion for Kernel and Jobs**

Your simulation should be of an old-school, single procossor, single-threaded system. Because we're actually running the simulation on modern hardware with multiple cores and multi-threading, we've got to manage things carefully. In particular, you should manage it so that whenever the kernel thread is running, there is no Job thread running, and vice versa. A fairly straightforward way to accomplish this is to have the behavior between the kernel and each of the Jobs mimic that of a producer/consumer using a bounded buffer of size 1. To do this my solution uses Java's ReentrantLocks and their associated Condition objects. These work very much like the monitors (and their condition variables) discussed in the textbook. I use Java's *synchronized*methodology elsewhere, but with ReentrantLocks I can create a separate Condition object for each Job. All the Jobs and the kernel hold the same lock, L. Consequently, only one of them can be running at a time. By using *await()* and*signal()*on the Conditions you can pass control back and forth between the kernel and the Jobs.

For an excellent discussion of ReentrantLocks and Conditions, please see Jeff Friesen's [article](http://www.javaworld.com/article/2078848/java-concurrency/java-101-the-next-generation-java-concurrency-without-the-pain-part-2.html) at www.javaworld.com.

Here's an outline of how this works.

1. The kernel (a SystemSimulator object) has a ReentrantLock, *singleThreadMutex* and has invoked *lock()*on it in the kernel's constructor.
2. Each Job has a Condition, *myCondition*, created in the Job constructor via   
   myCondition = s.getSingleThreadMutex().newCondition();
3. When it is time to start a Job, the kernel invokes a *start()* on it, but the first line of a Job's *run()* method is  
   myOS.getSingleThreadMutex().lock();  
   so the Job thread immediately blocks, because the kernel thread already holds that lock.
4. The next line executed by the kernel is something like  
   theNewJob.getMyCondition().await();   
   This works like a *wait()*in a synchronized method--it temporarily yields the lock to the thread associated with that Condition, but the thread executing the *await()* (the kernel, in this case) immediately blocks. The Job, though, is now free to run.
5. When we want the Job to return control to the kernel (for example, when the Job is terminating) the Job invokes a *signal()* on the same Condition and then either terminates or invokes an *await()*on the same Condition.

To implement the ready queue for a FCFS scheduler, you might consider using the java.util.concurrent.ConcurrentLinkedQueue class, though you could also simply use an ArrayList and synchronize the appropriate methods.

To complete the implementation of this project, start by examining the code that I have provided. Note which methods are available for each class. I have marked with "TO\_DO" all the places that you need to complete the coding.

**Recommended "Path to Victory":**

1. You might start by creating a class that implements the JobWorkable interface. You'll need that for WorkFactory.java to compile.
2. Complete GanntChart.java and test it.
3. Complete TO\_DO in SystemSimulator.java
4. Complete TO\_DO in Submittor.java
5. Complete FCFSScheduler.java

My boilerplate code for the driver, [RunScheduler.java](http://www.emunix.emich.edu/~evett/OS/Assignments/Scheduler/RunScheduler.java), is complete. If you'd like to alter the file, that is fine, but I will execute your program by running the main() of this file.

**Input File:**

The input file must be named "scheduleInput.txt". Each line of this file is of the form:

*jobID delayTillSubmission CPUburst*

The first argument will be an integer, and must somehow be used in the name of the Job (how is up to you). The second argument is the number of milliseconds that should elapse from the submission of the Job corresponding to the previous input line until this Job is submitted to the SystemSimulator via AddNewProcess. (If this is the first line of the file, then *delayTillSubmission*should be measured from the start time of the SystemSimulator itself. The *CPUburst* is the number of milliseconds that this Job should run until it terminates. Such time, of course, excludes any spent on the ready queue.

Here is a sample input file, [scheduleInput.txt](http://www.emunix.emich.edu/~evett/OS/Assignments/Scheduler/scheduleInput.txt).

**Output:**

The output must end with some kind of a Gannt chart that identifies when each job was running. The bottom of this [sample output](http://www.emunix.emich.edu/~evett/OS/Assignments/Scheduler/sampleOutput.txt) provides an example, though your method could be different. The Gannt chart should start with the execution of the first job to arrive, and end with the termination of the last job specified in the input file. Note that it is possible that the Gannt chart must be able to show "idle" periods (if they exist), where no jobs are currently running, but the input file specifies that more are yet to arrive.

For example, suppose the input file was:

1 0 200  
2 300 300  
3 300 300  
4 100 300

(Note that job #1 should finish at around time 200, but job #2 doesn't arrive until time 300.) Then the resulting Gannt chart might look something like (because you are using real-time timers, the numbers may fluctuate slightly):

GANNT CHART:

BurstStart BurstEnd JOB   
 0 1 IDLE   
 2 211 1   
 211 302 IDLE   
 302 608 2   
 609 910 3   
 911 1212 4   
 1212 1212 FINISHED

**Video Discussing How to Attack this Project**

Can be found at [this link](http://www.emunix.emich.edu/~evett/OS/Online/simpleSchedulerProjectWeb/index.html).

**To Hand In:**

Submit to the assignment's drop box a zip file containing

1. All of your source code (make sure your name is in the headers of any files you have created or modified).

The source code can either be all of your .java files, or the Eclipse folder you used.

1. The testing input files you used, along with an output file consisting of the output generated by your program, including the Gannt chart. If your program does not run, or there are run-time errors, provide an explanation here.