Lab 5 — Multithreading

CS 205

Lab objectives:

- Correct synchronization issues within a multithreaded program with Locks
- Program a multithreaded solution to a problem

1. You will start by debugging a program with a synchronization problem.
   (a) In your pair, download and import the lab5Starter project into Eclipse. (Note that this one project archive contains both the Counter project and the Mandelbrot project.)
   (b) Open the Counter project and examine the code which creates a Counter object and two Accessors which each increment the Counter ten times. Try running the program, which may take up to a minute, and check out the output. Run the application again to see if you get the same result.
   (c) Use Lock objects to fix the program to produce the expected (and correct) output. Only lock as much code as necessary in order to get the correct output, so that you don't limit the threads' parallelism.

2. You will now parallelize a Java program which produces a Mandelbrot set image. Open the Mandelbrot project. Run the program and study the code.
   (a) You will parallelize the program by creating threads, each of which will color the pixels of contiguous rows of the `BufferedImage I`. The threads split up the work done by the outer loop of the `Mandelbrot` class' constructor. For example, for an image with 80 rows and eight available threads, the first thread handles the first 10 rows, the second thread the next 10 rows, etc.
   (b) Today's CPUs have multiple cores, each of which can efficiently run one thread. Your program should create one thread for each core available on any computer it runs on. The method call

   ```
   Runtime.getRuntime().availableProcessors()
   ```

   can be used to determine the number of available cores at runtime.

   The image that you'll be creating has 819 rows. If you only have two cores available, the first thread is responsible for 409 rows, while the second thread is responsible for 410 rows. Keep this in mind when you write the portion of your program that allocates blocks of rows to threads.
   (c) Your main thread should create the child worker threads, start the threads, and then must wait for all of the worker threads to complete, before exiting. You'll find it helpful to use an array of threads and the `join()` method that was used in the `Driver` class of the Counter program.

(d) As mentioned in the program's comments, don't assume that a `BufferedImage`'s `setRGB()` method is thread-safe. This means that only one thread can call it at a time.

Add Javadoc comments to record the members of your team in the Class file containing your `main()` method and also to document the code that you write for the lab. Export your lab into a ZIP archive and submit it in GoucherLearn.