Concurrent Programming
COMP 409, Winter 2018

Assignment 1

Due date: Wednesday, February 7, 2018
6pm

These instructions require you use Java. All code should be well-commented, in a professional style, with appropriate variables names, indenting, etc. Your code must be clear and readable. **Marks will be very generously deducted for bad style or lack of clarity.**

In this assignment you may use only basic synchronization constructs; in Java, this includes the `synchronized` and `volatile` keywords, `wait`/`notify`, and the `java.util.concurrent.atomic.*` classes, as well as basic Thread methods, such as `start`, `join`, `sleep`, `currentThread`, `yield`.

All shared variable access must be properly protected by synchronization—there must be *no race conditions,* but also no unnecessary use of synchronization. Unless otherwise specified, your programs should aim to be efficient, and exhibit high parallelism, maximizing the ability of threads to execute concurrently. Please stick closely to the described input and output formats.

1. One or two threads draw random circles in a single image, pixel by pixel (ie not using any built-in ellipse or circle-drawing facility). If there are 2 threads, their drawings should not interfere with each other other, but certainly many random circles will not overlap, and thus it should be possible to sometimes draw them concurrently. **15**

   Implement a simulation that allows the threads to render their circles as concurrently as possible. Repeatedly, each thread independently and randomly selects a center and radius and tries to draw a circle (wrap left/right and up/down). It should not be possible for two overlapping circles (or parts thereof) to be drawn at the same, but there should be no restriction on drawing two non-overlapping circles concurrently. Note that "overlapping" includes intersecting and proper nesting.

   Template code is provided that constructs a blank image of given dimensions, and is able to write out the image as a `.png` file. The program accepts 3 command-line arguments: r, c, and a flag. The r parameter gives the maximum radius of circle to draw, the c number of circles that should be drawn, and the flag indicates whether to use two threads or just one (false for one thread, true for two).

   Add timing code (using `System.currentTimeMillis`) to time the actual work of the thread(s) (ie not including the initial image creation, or the file I/O). The program should emit as output a single integer (long) value, the time taken in milliseconds.

   Now, plot performance versus r. For some reasonable r value, choose a count c such that with the flag false (ie one thread) it takes at least a few hundred ms (preferable a second or two) of timed work. Retain the same c value and time the system with the flag true and false while varying r—try at least 5 values of r, varying from small up to well over half the image size. Note that when timing the program you should execute the exact same execution scenario several times (at least 5), discarding the first timing (as cache warmup), and averaging the rest of the values.

   Provide a graph of the relative speedup of your multithreaded version over the single-threaded version for the different choices of r. You should be able to observe speedup for some value of r, but perhaps not all values. This of course does require you do experiments on a multi-core machine.

   Either as text included in the plot file, or as a separate `.txt` file (and specifically *not* just as code comments), briefly explain your results in relation to your synchronization strategy.

2. This question disallows lock-based synchronization—may not use `synchronized`, nor `mutex`es, and **15**

must rely entirely on basic atomicity. You must still avoid race conditions of course.

One thread first constructs a binary tree. The nodes in the tree contain a single floating point number as data, and the tree itself is doubly-linked—parent nodes have pointers to their left/right children, and child nodes have a reference back to their parent (which also indicate whether they are a left or right child). The tree data should be random, but initialized so an in-order traversal is guaranteed to be in order. The root pointer should be stored as a global/static variable.

Once the tree is constructed, launch 2 additional threads. All of these threads (repeatedly) traverse the tree in a depth-first, in-order fashion. In doing so, they rely solely on the child and parent pointer values, and do not maintain a stack.

Two of the threads just read the data found in the tree nodes, recording the in-order sequence in a private string as a series of space-separated values. After reading a value, and before entering a child or returning to the parent, the thread sleeps for 5–20ms (value randomly selected). Once a tree traversal is complete, it records a newline in its private string and restarts.

The third thread also traverses the tree, but makes random decisions to go left/right, and instead of reading data it modifies the tree structure. As it descends it keeps track of the data values of the in-order predecessor and successor (so it can create new data properly ordered). At each node it reaches it has a 10% chance of deleting the left child/subtree (if there is one) and a 10% chance of deleting the right child/subtree (if there is one). If there is no left child, it has a 40% chance of creating a new one, and similarly if there is no right child it has a 40% chance of creating a new one. Newly created nodes have data values chosen to respect the in order property of the in-order traversal. Once it creates or deletes a node, this thread sleeps for 1–5ms, and restarts from the root.

After 5 seconds of execution in a thread it should terminate. The modifying thread should join with the the two threads reading, and emit their private strings, predicated by "A" (first thread) or "B" (second thread).

3. Let us investigate the cost of synchronization using a simple micro-benchmark. Build a program which does, and separately times, each following: **5**

   (a) In a loop from 0 to `Integer.MAX_VALUE/x`, increment an static integer value.
   (b) In a loop from 0 to `Integer.MAX_VALUE/x`, increment a volatile static integer value.
   (c) In a loop from 0 to `Integer.MAX_VALUE/x`, increment a static integer value, within a `synchronized` block.
   (d) Using 2 threads, increment a static volatile integer value from 0 to `Integer.MAX_VALUE/x`, within a `synchronized` block.
   (e) Using 2 threads, increment a static integer value from 0 to `Integer.MAX_VALUE/x`

   Choose an x which is tolerable, but has non-trivial runtime (at least a couple of seconds in one of the single-threaded cases). Report the performance, as an average of 7 runs, discarding the first run.

What to hand in

Submit your assignment to *MyCourses*. Note that clock accuracy varies, and late assignments will not be accepted without a medical note: **do not wait until the last minute**. Assignments must be submitted on the due date **before 6pm**.

Where possible hand in only **source code** files containing code you write. Do not submit compiled binaries or .class files. For any written answer questions, submit either an ASCII text document or a .pdf file *with all fonts embedded*. Do not submit .doc or .docx files. Images (plots or scans) are acceptable in all common graphic file formats.
This assignment is worth 10% of your final grade. **35**