Concurrent Programming
COMP 409, Winter 2021

Assignment 1

Due date: Tuesday, February 9, 2021
6pm

General Requirements

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.**

There must be no data races: all shared variable access must be properly protected by synchronization. Any variable that is written by one thread and read or written by another should only be accessed within a synchronized block (protected by the same lock), or marked as volatile. At the same time, avoid unnecessary use of synchronization or use of volatile. 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.

Your assignment submission **must** include a separate text file, `declaration.txt` stating "This assignment solution represents my own efforts, and was designed and written entirely by me". Assignments without this declaration, or for which this assertion is found to be untrue are not accepted. In the latter case it will also be referred to the academic integrity office.

Questions

1. You and k of your friends are cleaning out a zombie invasion. You've secured a large building consisting of just one very large room and k doors to the street. Zombies move slowly, so it's easy to control in general. You only have one weapon though, so you've set up the following protocol. Each of your k friends controls one of the k doors; they let in individual zombies, keeping count of how many have entered. You stand in the center, and eliminate the zombies that have entered as fast as you can, keeping track of how many you have removed. **15**

   You don't want too many zombies in the room for obvious reasons, but keeping track of the total while you're busy is too difficult. To find the total you need to radio each of your friends individually, one by one, to find out how many each has let in. The resulting sum must be correct and true while you make a decision on what to do. If there are too many you need to ensure no new zombies enter until you've had opportunity to reduce their numbers—tell each of your friends to not let in zombies for a while, and check again in a bit. Only once the total number is below a reasonable threshold should you allow zombies back in, again only by radio-ing each friend individually.

   Simulate this as a multi-threaded program. You should have $k + 1$ threads, one representing you and one for each friend/door. Each friend thread should let in a zombie with a 10% chance every 10ms, keeping track of the number she admitted. The thread representing you has a 40% probability of removing a zombie once every 10ms. You should check the total every 1s, and if it is below n then everything is ok, otherwise no new zombies should be admitted until you've reduced the number to below $n/2$.

   Provide a Java program `zombie.java` that takes two integer command-line parameters $k \geq 1$ and $n \geq 2$. Use `synchronized` to coordinate thread interaction and ensure shared data is properly protected by synchronization; do not use operations or data structures in the `java.util.concurrent` package or sub-packages. Your solution should allow for maximal concurrency—operations should not be serialized unless necessary. Run your program for a few minutes with $n = 5$, $n = 10$, $n = 100$. What is your throughput (zombies eliminated/second)?

2. A virtual terrain may be represented by a discretized 2D space, with each discrete location given a height value. Many techniques exist for generating random virtual terrains. One conceptually simple approach is to use *fault-lines*. Given a rectangular boundary for the terrain, a point on each of two different boundary edges is selected, and a line is drawn between them. The height of all locations within the terrain on one side of the line are then raised by the same (randomly selected) amount. This process is repeating many times, producing a (quite rugged, noisy) terrain. **20**

   The goal here is to use multiple threads efficiently to generate a terrain using the fault-line approach. Define an integer $w \times h$ grid of height values for a given an input w and h, all heights initialized to 0. Once initialized, use t threads to modify height values: each thread chooses random entry/exit points to define a fault-line and a random height adjustment (within a $[0, 10]$ range), and then adds the height adjustment value to every point on one side of the line[1] (whether you include the points on the line itself and which side to adjust is arbitrary/unimportant). Each thread does this as fast as possible, with the whole simulation ending when a total of k fault-lines have been created.

   It should not be possible to lose an attempted height adjustment—two threads adjusting the height of the same point should not interfere with each other. It should, however, be possible for two different points to be changed simultaneously by different threads.

   After the total k fault-lines have been processed, only one thread should be used to compute and emit a pixel image (`.png`) of the result: determine a global min/max, and map the range to a reasonable spread of colour (RGB) values.

   Your program, `fault.java` should accept command-line arguments, w, h, t, and k. Choose w, h and a $k > 8$ such that the program typically takes measureable time (at least 10's of milliseconds, preferably 100's of ms or more) to run with $t = 1$. Add timing code (using `System.currentTimeMillis`) to time the program from the point the threads begin (or about to begin) work to the point when all threads have completed their work (but before the image is generated). Once all threads have completed, the time taken in milliseconds should be emitted as console output and the image output to a file, named `outputimage.png`.

   Plot performance (speedup) versus the number of threads (t), keeping w, h, k fixed (ensure $k \geq t$). Given the randomness, you will need still to do several runs at each thread-count (at least 5) averaging the timings (you may opt to discard the initial run as a cache-warmup). Provide a graph of the relative *speedup* of your multithreaded versions over the single-threaded version for 2, 3, and 4 threads, and briefly comment on characteristics of your speedup data. You should be able to observe some speedup for some number(s) of threads, but perhaps not all values. This of course does require you do experiments on a multi-core machine. Note that achieving speedup is not necessarily trivial: beware of sequential bottlenecks from different methods and APIs you may use or implement.

   Example/template code that creates an image of a given size and writes it to a file is provided with this assignment description.

   Provide your source code (`fault.java`), and as a *separate document* your performance plot with a brief textual explanation of your results (why do you get the speedup curve you get).

[1]Note: given a vector $p_0 \to p_1$ you can decide whether a point p_2 is "left" or "right" of that line by computing:

$$(p_1.x - p_0.x)(p_2.y - p_0.y) - (p_2.x - p_0.x)(p_1.y - p_0.y)$$

If the result is > 0 then p_2 is "left" of $p_0 \to p_1$, or if the result is < 0 then p_2 is "right" of $p_0 \to p_1$, or if the result is $= 0$ then p_2 is colinear with $p_0 \to p_1$.

What to hand in

Submit your declaration and assignment files 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. Avoid .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. $\overline{\mathbf{35}}$