Concurrent Programming

COMP 409, Winter 2024

Assignment 3

Due date: Wednesday, March 13, 2024
Midnight (23:59:59)

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. This means all shared variable access must be properly protected by synchronization: any memory location 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. A simple computer game allows some number of characters to move around a 2D space discretized into an $m \times m$ grid, with each character occupying a grid cell, and characters following an 8-way movement model (i.e., they can move up, down, left, right, but also in the diagonal directions). **15**

   Characters begin at spawn points, n of which are placed at random, non-overlapping points on the perimeter of the map. Characters then move around the grid. Each character chooses a random goal point no more than 10 cells away from their current location (and clipped to the interior of the map), builds a movement plan consisting of the ordered sequence of steps they plan to make, and then enacts that plan. After each step in their plan a character pauses for 20 ms before making another step. Once at their destination they choose a new destination and repeat the process.

   A character's movement plan consists of the sequence of cells forming as "straight" a line as possible between their current and destination location. For this use the *Bresenham* algorithm to compute the series of steps forming a "straight" line to their goal, given the 8-way movement model. There are many online sources on how to compute a Bresenham line.

   Characters must not touch; if two characters end up occupying the same cell at the same time then both should be disappear. The number of characters will thus eventually reduce to 1 (or 0) as the simulation progresses. To compensate, new characters are created at the spawn points every s ms.

   The movement of each character implies different tasks, consisting of occasional movement planning, as well as regular, individual movement. For this question you will explore and compare the impact of using a *thread pool* to manage the simulation. You will thus need two versions of your code: a non-pooled version, `q1n.java`, in which each character is controlled by a separate thread, and `q1p.java`, in which character actions are managed by a thread pool. The design of the thread pool version is up to you—you can use one or multiple pools, of any type available in Java, or customized. You must, however,

have some way of limiting the the maximum number of threads that can be active in your simulation at any one time to a parameter, t.

Your applications should both accept parameters $m\ n\ s\ t$. For testing, let the application run and terminate it after ≈10 s, emitting the total sum of successul moves (steps) made over all the characters. Establish a baseline using your non-pooled version, picking a large m, $n \approx m$, and $s = 100$, and see how many moves are made on average over a few runs of your simulation. Now, try a couple more simulations, one with $s = 50$ and one with $s = 20$, and note the total number moves made (average over a few runs).

Compare corresponding tests using your pooled version. For this you will also need to explore different values of t. How does performance vary, and what impact does the number of threads per character have?

In a separate document `q1.txt`, briefly describe your pooling strategy, give your experimental data, and offer some explanation of your results.

2. In class we discussed the design of a (lock-free) elimination stack. The design has some overhead though, and whether there is really a performance advantage over a simpler lock-free stack design is not necessarily obvious. In this question you will compare performance of an elimination stack to a lock-free stack, and attempt to discover what, if any, performance and context parameters make use of an elimination stack viable. **10**

   First, implement a baseline lock-free stack. Your stack must support two operations, PUSH and POP, and should be capable of reusing stack nodes (such as by re-PUSH-ing after POP-ing) without losing data or corrupting the stack.

   Next, in the same code-base, implement a lock-free elimination stack. It should have the same properties (can reuse nodes without error), but must make use of an elimination (exchanger) array as a pre-filter on stack operations. You can either implement your own exchanger, following the design described in class (or the text), or the exchanger available in Java. The size of the elimination array (e) and the max delay (or number of spin-iterations if you implement the exchanger yourself) used to wait for an elimination partner (w) should be parameters.

   Your stack must be tested by starting t threads that attempt PUSH or POP operations on the stack, each operation being either a `push` or `pop`, selected with even odds. Whenever a node is popped a thread should immediately set the object's `next` field to be null. A thread may PUSH either a newly allocated node, or use an old, previously popped node. To support the latter it should retain the last 50 nodes it has popped, and when performing a PUSH, if it has any old nodes stored, then 50% of the time it should reuse an old node. After each push/pop operation, a thread sleeps for a random time, 0–s ms. Each thread will attempt to perform n operations (PUSHes should always succeed, but some POPs may fail if the stack is empty).

   Your program should be invoked with arguments ordered as:

   `java q2` $x\ t\ n\ s\ e\ w$

   where x is a 0 if a regular lock-free stack is used, and 1 if an elimination stack is to be used. As output, once all threads have completed their n operations, you program should emit one line, consisting of the time (in ms) taken by the simulation, a space, and the total number of nodes remaining on the stack.

   Choose a large $n > 1000$ so the simulation runs for a measurable time with $x = 0$. Experiment with choices of $t > 1$, $s < 40$, $e \geq 1$, and $w \geq 1$, considering at least 2 distinct values for each (holding others constant). In a *separate document* `q2.txt`, give some data and briefly describe how each of these parameters affects the relative performance of the base lock-free stack versus the elimination stack. Can you find a combination of parameters for which the elimination stack performs better than its underlying lock-free version?

What to hand in

Submit your declaration and `q1n.java`, `q1p.java`, `q1.txt`, `q2.java`, and `q2.txt` files to *MyCourses*. Note that clock accuracy varies, and late assignments will not be accepted without a special permission: **do not wait until the last minute**. Assignments must be submitted on the due date **before midnight**.

Where possible hand in only **source code** files containing code you write. Do not submit compiled binaries or .class files, but do include a `readme.txt` of how to execute your program if it is not trivial and obvious. 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. **25**