Concurrent Programming

COMP 409, Winter 2025

Assignment 3

Due date: Tuesday, March 18, 2025
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 <u>very generously</u> 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. Design a concurrent resizable array with `Object get(int i)` and `void set(int i,Object o)` operations, as well as a parameter-less constructor that initializes the array to size 20. For resizing you only need to support increasing the array size in increments of 10, and resizing should be automatic based an attempt to access the array one past the end of the array (you do not need to account for or protect against accesses beyond that limit).

   Come up with two implementations, one using blocking synchronization, and one using a *lock-free* strategy.

   (a) Your first implementation, `q1a.java`, should be based on only using atomic reads and writes of primitive data, as well as blocking synchronization (such as via either `synchronized`, or `java.util.concurrent.locks.ReentrantLock`). For a size n array, however, you may only use $o(n)$ (note that is little-o) data to control synchronization. **4**

   (b) Your second implementation, `q1b.java`, should be *lock-free*. Use only atomic reads and writes of primitive data, as well as the various Java implementations of CAS, TS, FA, in the `java.util.concurrent.atomic` package (but do not use the array classes in that package). **6**

   Your designs do not necessarily have to allow *all* n values to be accessed concurrently, but should allow multiple threads to concurrently read and write different array elements in $O(1)$ while the array is not in the process of being resized. Resizing should of course not lose or corrupt the prior array state.

   Which design has better performance? Write a driver program `q1.java` that accepts two parameters k and m and tests performance with four threads, each thread doing m operations consisting of $100 - k\%$ of the time reading or writing (50/50) an existing array element, and $k\%$ of the time accessing one past **2** the end of the array. Select reasonable values of k and m for testing and in a *separate document* briefly describe your design strategy for both implementations and relate it to the performance data you gather.

2. This problem requires you explore the use of *thread pools* in Java. You have probably encountered the problem of *bracket matching*, wherein you need to verify that the opening and closing brackets in a string are each matched in a properly nested manner. Sequentially this is straightforward: characters are processed sequentially with a *counter* starting at 0, incremented on an opening bracket, decremented on a closing bracket, and verifying that the counter is never negative and is 0 at the end. **8**

   This can be solved in parallel using a divide-and-conquer property of bracket matching. Suppose we divide the string into two (left and right) pieces and check bracket matching independently in both. In the simplest case, if both substrings are individually properly matched then the concatenation is also matched. However, we cannot trivially conclude a bracket mismatch if the substrings don't have balanced brackets themselves—even if the right substring contains too many closing brackets, for example, the left substring might have enough opening brackets to balance it.

   We can generalize this by assuming the bracket verification of each substring generates a triple, (b, f, m), where b is a boolean for whether brackets are properly matched, f is the counter result on that sequence, and m is the minimum counter value on that sequence. For example, as base cases, a single character '(' has the triple $(\textit{false}, 1, 1)$, ')' has the triple $(\textit{false}, -1, -1)$, and a non-bracket character has the triple $(\textit{true}, 0, 0)$,

   Given (b_1, f_1, m_1) and (b_2, f_2, m_2) from the left and right substring, we can compute the triple for the concatenated string as (b, f, m) where,

$$
\begin{aligned}
b &= (b_1 \ \wedge \ b_2) \ \vee \ ((f_1 + f_2 = 0) \ \wedge \ (m_1 \geq 0) \ \wedge \ (f_1 + m_2 \geq 0)) \\
f &= f_1 + f_2 \\
m &= \mathit{min}(m_1, f_1 + m_2)
\end{aligned}
$$

   Using the `newFixedThreadPool(int)` static factory method of the `java.util.concurrent.Executors` class, construct a thread pool to do parallel *bracket matching* using this technique. Your program, `q2.java` should be launched as "`java q2 n t s`", where n is the string length, t is the number of threads to use in the pool, and s is an optional random-seed parameter (if not provided seed from `System.currentTimeMillis`). Template code is provided in `Bracket.java` to generate the initial array of characters, each of which is either an opening bracket '[', closing bracket ']', or non-bracket character '*'.

   Output of your program should consist of two lines. The first line should be the time (in milliseconds) of the program (excluding array construction, sequential validation, I/O). The second line should show two boolean values separated by a space—the first is the final boolean conclusion of your parallel check (the final b value), and the second is the result of calling `Bracket.verify()`.

   Your code should generate (relative) speedup for some number of threads on some input size. Find an n for which that is true, and in a *separate document* show a speedup curve for at least 3 different thread (t) values, including $t = 1$ as a baseline. State your value of n, and give a *brief* explanation for your data.

What to hand in

Submit your declaration, code, and performance/design explanations 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. $\overline{\textbf{20}}$

Programming assessment

	Mastery	Proficient	Developing	Beginning
Correctness	The solution works correctly on all inputs and meets all specifications.	The solution meets most of the specifications; minor errors exist.	The solution is incorrect in many instances.	The solution does not run or is mostly incorrect.
Readability	Well organized according to course expectations and easy to follow without additional context.	Mostly organized according to course expectations and easy to follow for someone with context.	Readable only by someone who knows what it is supposed to be doing.	Poorly organized and very difficult to read.
Algorithm Design	The choice of algorithms, data structures, or implementation techniques is very appropriate to the problem.	The choice of algorithms, data structures, or implementation techniques is mostly appropriate to the problem.	The choice of algorithms, data structures, or implementation techniques is mostly inappropriate to the problem.	Fails to present a coherent algorithm or solution.
Documentation	The solution is well documented according to course expectations.	The solution is well documented according to course expectations.	The solution documentation lacks relevancy or disagrees with course expectations.	The solution lacks documentation.
Performance	The solution meets all performance expectations	The solution meets most performance expectations	The solution meets few performance expectations	The solution is not performant.