Concurrent Programming

COMP 409, Winter 2020

Assignment 3

Due date: Thursday, March 19, 2020
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.

1. 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. **20**

   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. If both substrings are individually properly matched, then the concatenation is also matched. However, even if not it might still match if the right substring contains too many closing brackets, but the left substring leaves enough opening brackets to balance it, while still never dipping below 0.

   We can generalize this by assuming the bracket verification of each substring generates a triple, (ok, f, m), where *ok* 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 $(\mathit{false}, 1, 1)$, ')' has the triple $(\mathit{false}, -1, -1)$, and a non-bracket character has the triple $(\mathit{true}, 0, 0)$,

   Given (ok_1, f_1, m_1) and (ok_2, f_2, m_2) from the left and right substring, we can compute the triple for the concatenated string as (ok, f, m) where,

   $$\begin{aligned} ok &= (ok_1 \ \wedge \ ok_2) \ \vee \ ((f_1 + f_2 = 0) \ \wedge \ (m_1 \geq 0) \ \wedge \ (f_1 + m_2 \geq 0)) \\ f &= f_1 + f_2 \\ m &= min(m_1, f_1 + m_2) \end{aligned}$$

   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, `q1.java` should be launched as "`java q1 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 *ok* 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.

2. The text goes through the construction of a lock-free, elimination stack. The baseline lock-free stack in the text, however, assumes garbage collection solves the ABA problem. The goal in this question is to build an elimination stack, properly backed by a lock-free stack that does not suffer from ABA concerns with or without garbage collection. **20**

Your stack needs to support two operations, PUSH and POP. Your stack must be capable of reusing nodes/data (re-PUSH-ing after POP-ing), and it should not be possible to lose data or otherwise corrupt the stack.

Implement a lock-free stack that cannot suffer from an ABA problem. Extend it with an elimination stack based on an elimination array. You can follow the general design in the text, but rather than build your own exchanger make use of the `java.util.concurrent.Exchanger` class. The size of the elimination array and the timeout used to wait for an elimination partner should be parameters.

Your stack must be tested by starting p threads that then repeatedly perform PUSH or POP operations on the stack, randomly choosing one operation or the other with equal probability. After popping a value a thread should immediately set the object's `next` field to be null. A thread may PUSH either a new value, or an old, previously popped value. To support the latter it should retain the last 20 items it popped, and when performing a PUSH, if it has any old values then 50% of the time it should randomly select a previously popped node to re-push. After each push/pop operation, a thread sleeps for a random time, 0–d ms. Each thread should also keep track of how many pushes it does and how many pops successfully returned actual data.

Your program should be invoked with 5 integer arguments as:

```
java q2 p d n t e
```

Where $p > 1$ represents the number of threads to use, $d \geq 0$ represents the upper bound for the random delay between each thread operation, and n the total number of operations each thread attempts to do. The last 2 parameters are specific to the elimination stack design: $t \geq 0$ represents the timeout factor used in the elimination stack, and $e > 0$ is the size of the elimination array. All times are in milliseconds.

Choose an $n > 1000$ and a relatively brief d, such that execution takes at least several seconds with $t = 0$.

Once all threads are done, your program should emit a time in milliseconds measuring the entire concurrent simulation on one line. A second line of output contains three numbers separated by spaces: the first value should be the total number of pushes done by all threads, and the second should be the total number of successful pops done by all threads, and the third the total number of number of nodes remaining in the stack.

Fix t at a non-0 value, but much less than d. In a *separate document*, show performance for each combination of $p \in \{4, 8, 16\}$ and $e \in \{1, 2, 4\}$. Is your elimination array being used effectively? Give a *brief* textual explanation for your data.

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. $\overline{\textbf{40}}$