Concurrent Programming
COMP 409, Winter 2019

Assignment 3

Due date: Wednesday, March 20, 2019
6pm

In this assignment you will evaluate and compare the performance of a lock-free priority queue over a blocking version.

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.**

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. First implement a blocking version of a bucket-based priority queue. This should follow the design given in section 15.2 of the text (assume priority values are integers between 0 and 9 inclusive). Use only blocking synchronization in your design, and do not use any built-in concurrent data structures. **10**

   Provide a program `q1.java` that takes 4 parameters: t q d n. Argument $t \geq 1$ represents the numbers of threads used, q is a floating point number ($0 \leq q \leq 1$) used as a probability of doing a `deleteMin` as opposed to an `add`, $d \geq 0$ represents the (nominal) random delay between each thread operation, and n is the total number of operations each thread attempts to do.

   Threads should have sequential (0-based) id values. Threads each repeatedly choose to either add or remove things to or from the queue, selecting to add if a random floating point value is less than q, and trying to delete from the queue if it is greater. Between each operation (including a failed delete because the queue is empty) a thread sleeps for $d$ms.

   Each `add` operation requires a random datum, and a random priority. The random data consists of a pair of characters, the first character being the thread id number, and the second being a random choice of alphabetic character (A-Z), and the priority value. Note that this data should be encapsulated as an object, and not just stored in the queue as a String or other primitive. Your wrapper objects must be reused if possible—each thread should keep references to last 10 objects they deleted from the queue and preferentially reuse those wrapper objects (with new random data/priority) when adding.

   In addition, each thread should retain an ordered record of the data (string and priorities) they added or removed (use "*" for a failed `deleteMin`). This includes timestamps for when the operation was performed.

   Choose an $n > 1000$ and a relatively brief d, such that execution takes at least several seconds. Set q to 0.5 and adjust it if necessary so deletion failures are present, but not dominating.

   Merge the sorted list of operations for each thread to form a single ordered list. Emit as final output the first 1000 operations in the merged list. On each line print the operation showing the time-stamp, thread id, operation, and data (including priority), each separated by a space. For example,

```
10090232030 1 del *        At time 10090232030 thread 1 failed to delete
10090232033 0 add 0S 7     At time 10090232033 thread 0 added "0S" with priority 7
10090232071 0 add 0B 2
10090232149 2 del 0B 2
...
```

2. Now, create a program `q2.java` as above, same input, output and behaviour, but based on a *lock-free* exchanger stack rather than a blocking stack. You must construct this yourself; do not use any built-in stacks or Java's built-in exchanger. **15**

3. Examine your output from both implementations. Does your output always have a sequential equivalent execution? Why or why not? Give a brief explanation of your results in both cases. **4**

4. Now compare performance of the two implementations. Disable the output and add timing code to measure the duration of the simulation itself, from the time the threads are started up until all threads complete. Execute each version for $t \in \{1, 2, 3, 4\}$, and compute an average running time over at least 5 times (discard the largest time). **6**

   In a separate document from your code, use a plot or table to show the relative performance of your two implementations. Include a brief discussion of your timing data. Lock-free implementations are intended to be substantially faster. Do you agree?

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**