Assignment 1 (CS 798: Multicore programming—Fall 2018)

Due date: October 22

If you are auditing the course, it is not necessary to do the assignment. If you are auditing and *choose* to do the assignment, please place a (∗) next to your name. I will prioritize marking assignments for enrolled students, and will mark auditors' assignments as time permits.

What to submit: your zipped code sent to me somehow (to be decided soon), and a paper copy of your other material (preferably double spaced, not handwritten).

1 Counters

This question has both implementation and theory components. For the implementation component, start by downloading the C++ code from `http://tbrown.pro/multicore/a1code.zip`. The code contains a simple counter microbenchmark (see workload_timed.cpp) and an implementation of a naive counter, as well as a placeholder for your own counter implementations (see counters_impl.h). Instructions to compile and run appear in README.md.

1.1 Easy counters

1. Implement: the *lock-based counter* and *fetch&add counter*, as described in the slides for week 1, Monday (by adding them to counters_impl.h).
   Note: you can use `pthread_spinlock_t` for your lock(s). To get fetch&add, you can either use the features in the C++ `<atomics>` header, or, if you are using a GCC compiler try `__sync_fetch_and_add`. Use the counter implementation in counter_impl.h as an example. Add your counter(s) to `workload_timed.cpp:main()`, following the examples there.

2. Experiment: (design and run a simple experiment to answer the question) how much more costly is a fetch&add than a write? Briefly describe the experiment and results (a few sentences). (Remember to make any shared variables volatile.)
   Note: it should be possible to do this with only small changes to the posted code.

3. Experiment: "how much more costly is it for many threads to fetch&add on a single address vs fetch&add on disjoint addresses?" Briefly describe the experiment and results.

1.2 Approximate counters

1. Implement: the *approximate counter* discussed (as part of the expandable hash table) in the slides for week 3, Monday.

2. Experiment: how does this improve scalability (and absolute throughput) relative to the *fetch&add counter* (with a single address)?

1.3 Sharded counters

In the first class, we discussed the possibility of sharding (or partitioning) a counter into multiple *subcounters* to avoid a concurrency bottleneck on a single memory address (read: cache line). Recall that, in the sharded counter we considered, increment does *not* return a value (only read does). In this question, we think of the counter's increment operation as having no return value.

1. Algorithm design: show how to use fine-grained locks (one lock per subcounter) to implement a linearizable sharded counter (with linearizable read operations). Do *not* worry about efficiency for this first algorithm.

2. Correctness: give a proof sketch showing the counter is linearizable.
   Hint: define linearization points for increment and read operations. Remember, you get to *choose* linearization points to make the return values "make sense," as long as the linearization point for an operation occurs *during* the operation. At a high level, you want to linearize increments and reads so that the return value of each read matches the number of increments linearized before it.
   More formally, consider any execution E of your algorithm. Let L be a linearized (sequential) execution that contains the same operations as E, except they are executed atomically in the order of their linearization points. You want to show that each operation has the same return value in L as it does in E.

3. Algorithm design: show how to implement a wait-free counter that performs *increment* in $O(1)$ steps and *read* in $O(n)$ steps, where n is the number of threads.
   (Desired counter semantics: Increments should return nothing, reads should return the number of increments performed so far. Wait-free means locks cannot be used, and every operation should make progress in a bounded number of its own steps.)
   Hint: is locking *needed* in your previous algorithm to prove that *read*s can be linearized?

4. Correctness: give a proof sketch showing this counter is linearizable.

2 Hash tables

Recall the *simplest possible hash table* from the slides for week 2, Wednesday (the algorithm on slide 4, as well as the proof sketch on slides 4-7).

1. Where does the linearizability proof sketch break down if we simply eliminate the lock and unlock calls?

2. Algorithm design: improve the hash table so it only locks once it finds a *NULL* slot.

3. Correctness: give a proof sketch for its linearizability.

4. Implement: the *simplest possible hash table* and your improved version.

5. Experiment: how much of a performance difference does the improvement make in, say, inserting 10 million or 100 million elements?
   Hint: you can use a copy of the test harness you were given for counters, but pass a hash table to `runExperiment` instead of a counter. Only small modifications should be needed from there.

3 Bonus

OPTIONAL: reproduce slide 14 of the week 2, Wednesday slides, but for *Murmurhash 3*.
`https://en.wikipedia.org/wiki/MurmurHash#MurmurHash3`