CSCI-GA.3033-017 Special Topic: Multicore Programming
Lab 1 Assignment

This is the first of four labs in which you'll be building a multithreaded server solving problems using genetic algorithms. In this stage, you'll be starting slow, with a thread-safe key-value store and a thread-safe queue. Some of you discovered that a hashmap provided an O(1) lookup solution for Lab 0; in this lab, you'll all be using a hashmap to store the underlying data. You won't be implementing anything with genetic algorithms yet.

A ***hashmap*** uses a mathematical function called a hashing function to turn any variable-length key (for example, a string) into a semi-unique hash with a fixed number of bits. An ideal hashing function would produce a unique hash for any possible key, but because the hash can have fewer bits than the key that is hashed, there necessarily must be collisions in a real hashing function. A well-designed hashing function minimizes these collisions, so a hashmap can be considered to have amortized constant-time (O(1)) lookups, insertions, and deletions.

Thread-Safe Key-Value Store

This time, you won't need to implement your own data structure. Just use `std::unordered_map` if you're using C++; otherwise, your implementation job is going to be much steeper. You'll be combining `std::unordered_map` with pthreads to make a thread-safe key-value store. Your job is to do the following:

Create a program that takes one argument with `-n`, the number of threads to create. I recommend using `getopt` or `getopt_long`. I.e.:

```
./lab1 -n <N>
```

Create a templated `ThreadSafeKVStore<K, V>` class. Inside it, add the necessary state for a thread-safe key-value store, namely a single hashmap and whatever synchronization primitive(s) (mutices, semaphors, condition variables) is/are necessary to synchronize accesses to the hashmap.

Implement several (thread-safe) methods:

1. **`bool insert(const K key, const V value)`**
   Should insert the key-value pair if the key doesn't exist in the hashmap, or update the value if it does. Only return false if some fatal error occurs (which may not be possible at this point). Return true on success.
2. **`bool accumulate(const K key, const V value)`**
   Like insert(), but if the key-value pair already exists, *accumulate* (i.e., add) the new value to the existing value. This of course means that the templated V type must support the + operator.

Because addition on some data types (like strings) is not commutative, make sure that your accumulation is in the form
`stored_value = existing_stored_value + new_value;`
The return values should mirror those for insert().

3. **`bool lookup(const K key, V& value)`**
   Return true if the key is present, false otherwise. If the key is present, fill the value variable (passed by reference) with the associated value.
4. **`bool remove(const K key)`**
   Delete the key-value pair with key key from the hashmap, if it exists. Do nothing if it does not exist. Return true on success; return false if there is some fatal error. The key not existing is a "success" condition, not an error (why? Think about the invariants that the `remove()` operation implies).

Thread-Safe FIFO Queue

Implement a templated, thread-safe FIFO queue class called `ThreadSafeListenerQueue<T>` on top of a `std::list`. It should do whatever you think is the right thing to make it safe to insert into and remove from the queue from different threads. You should also make it possible to "listen" to the queue, in other words block until there's something in the queue that can be safely removed. Your queue should implement at least the following functions:

1. **`bool push(const T element)`**
   Should push the element onto the front of the list, so that it will be the last of the items currently on the queue to be removed. Only return false if some fatal error occurs (which may not be possible at this point). Return true on success.
2. **`bool pop(T& element)`**
   Pop the least-recently inserted element from the queue and fill in the passed-by-reference variable element with its contents, if the queue was not empty. Return true if this was successful; return false if the queue was empty (a non-fatal error) or if some fatal error occurred.
3. **`bool listen(T& element)`**
   Similar to pop(), but block until there is an element to be popped. Return true if an element was returned, or false only if some fatal error occurred.

Testing

In your `main()` function, instantiate `ThreadSafeKVStore<std::string, int32_t>` and `ThreadSafeListenerQueue<int32_t>`, then spawn the requested number of pthreads, and pass the `ThreadSafeKVStore` and `ThreadSafeListenerQueue` to each one. Each thread should perform 10,000 iterations of a test. In each iteration, the threads should:

1. With probability 20%, accumulate (do not insert) a new key-value pair. The key should be of the form "`userN`", where N is an integer between 0 and 500. The value should be a uniformly selected signed 32-bit integer between -256 and 256. In each thread, keep track of the running sum of *all* values inserted into the `ThreadSafeKVStore`.
2. With probability 80%, look up an existing key-value pair (keep track of the keys . *Do* consider it a fatal error if the key-value pair is no longer present! Another thread should not have deleted this pair.
3. After each thread finishes its work, it should push the sum of all elements accumulated into the key-value store by that thread into the queue.
4. The main function should, after launching all of the threads, listen on the queue, and continue listening until it has received as many elements off the queue as there were threads. It should sum all of the elements removed from the queue.
5. Finally, the main function should iterate the `std::unordered_map` underlying the key-value store (provide a thread-**un**safe way to access the begin() and end() iterators for the underlying map), and add all of its values together. That sum must match the sum of the elements popped from the queue, which in turn represents the sum of all of the elements accumulated into the key-value store. If this sum does not match, you have a synchronization error somewhere!
6. The program should track and display the time:
    a. For each thread to complete (each thread should print its completion time).
    b. Between launching the first thread and the final thread terminating.

Notes:

1. Please do what is necessary to ensure that the threads have seeded their random number and key generators with different seeds.
2. The tests you will implement (to the above spec) will ensure that your data structures are truly thread-safe. Make sure that you check not only the adherence to this spec, but the correctness of the results!

Deliverables: You're expected to produce at least two things:

1. The source code for an executable that implements the spec above. I recommend having at least three source code units, one for the main function and thread function(s), one for the `ThreadSafeKVStore` class, and one for the `ThreadSafeListeningQueue` class. You may further modularize the assignment if necessary.
2. A `README` that explains how I can compile and run your code. I would prefer that you make a simple Makefile so I can make your code. Otherwise, provide a `build.sh` that invokes g++, or at worst, put the gcc/g++ command that should be used to compile your code in the `README`.

Grading: This lab will contribute at most 25% to your lab score. It will be graded on functionality, adherence to the specification, thread safety, code style, and commenting. Proper thread safety will be given an inordinate share of the grade, for obvious reasons.

Getting Help: If you're struggling, your first recourse should be the class mailing list. *Important:* you should solicit your fellow students for high-level help, such as "how can I iterate through a vector?". You should not ask for specific help with your code, such as "why doesn't the following code work?". If you're still stuck in a week, come to my office hours on Wednesdays. See the first lecture for the policy on collaboration with other students (tl;dr: don't, outside of the mailing list). As is university policy, instances of cheating will be taken very seriously. If you believe there's an omission or error in this document, you're welcome to email me directly.

Due date: October 15, 2018, by 11:59:59pm EDT. See the first lecture for the late policy.

Submission: Push your code to your **private** MulticoreProgramming repository in a folder entitled "lab1". Please also send me (cmitchell@cs.nyu.edu) and the grader (TBA) an email once you have committed and pushed your final submission with the hash (a 7+-character hexadecimal string) of that commit; I will use the timestamp GitHub records for that commit to determine whether your submission is on-time or late.