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 data server. It will operate over an approximation of the HTTP protocol, allowing data to be read (GET) and written (POST). In this stage, you'll be starting slow, with a multi-threaded key-value store. 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 sockets and the HTTP protocol 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.

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. You needn't template this one; just use `std::string` for your keys and values. Your job is to do the following:

1. 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>`
2. Create a `ThreadSafeKVStore` class. Inside it, add the necessary state for a thread-safe key-value store, namely a single hashmap and whatever mutex or mutices is necessary to synchronize accesses to the hashmap.
3. Implement several (thread-safe) methods:
   a. **`int insert(const std::string key, const std::string 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 -1 if some fatal error occurs (which may not be possible at this point). Return 0 on success.
   b. **`int lookup(const std::string key, std::string& value)`**
      Return 0 if the key is present, -1 otherwise. If the key is present, fill the value variable (passed by reference) with the associated value.
   c. **`int remove(const std::string key)`**
      Delete the key-value pair with key key from the hashmap, if it exists. Do nothing if it does not exist. Return 0 on success; return -1 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).

4. In your `main()` function, instantiate `ThreadSafeKVStore`, then spawn the requested number of pthreads, and pass `ThreadSafeKVStore` to each one. Each thread should perform 10,000 iterations of a test. In each iteration, the threads should:
    a. With probability 10%, insert a new key-value pair. The key should be of the form "`userN`", where N is an integer between 0 and 10,000,000. The value should be a string with 8 to 256 characters, uniformly distributed. The contents of the value are unimportant.
    b. With probability 10%, delete a key from the hash map. The thread should keep track of the keys it inserted in a `std::vector` or `std::unordered_set`, and pick a random key to delete.
    c. With probability 80%, look up an existing key-value pair. Do *not* consider it a fatal error if the key-value pair is no longer present! Remember, another thread may have deleted this pair.
5. The program should track 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. If you want to guarantee your code's thread safety by adding a test managing the keys and values inserted and deleted by different threads, by all means do so, but you will not be penalized for not having one.

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 two source code units, one for the main function and thread function(s), and one for the `ThreadSafeKVStore` 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 Thursday. 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 13, 2016, 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 an email once you have committed and pushed your final submission with the tag (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.