CSCI-GA.3033-017 Special Topic: Multicore Programming
Lab 2 Assignment (Part 1: Implementation)

This is the second of four labs in which you'll be building a multithreaded data server. This key-value server stores a (potentially large) set of key strings, each of which maps to a value string. It will operate over the HTTP protocol, allowing data to be read (GET), written (POST), and even removed (DELETE). To make things interesting, it will also compute and store the MD5 hash of the value every key in the system. This lab will challenge you to complete three tasks:

1. Implement reader-writer lock(s) to improve concurrency.
2. Implement thread pool to handle requests.
3. Use provided HTTP 1.1 parser/generator: http://cs.nyu.edu/courses/fall17/CSCI-GA.3033-017/httpreq.zip

Further information, hints, tips, and tricks will be / were given in Lecture 6. Specific requirements are as follows; everything else is up to you. Please **strongly** consider documenting the design you create for structuring your classes and their methods, including the functionality you're delegating to each class and method, in a document for your and our reference in coding and grading, respectively. It can only help you write better-structured code the first time.

1. Your lab must utilize reader-writer lock(s) in some way to improve concurrency in the ThreadSafeKVStore from Lab 1. Ambitious students may wish to use multiple ThreadSafeKVStores dividing the key space in some way, each with associated lock(s), but this is not required. Using pthread reader-writer lock(s) instead of implementing your own is encouraged.
2. You must implement a thread pool. Your program will take the same command line argument as last time: a single integer specifying the number of threads to spawn. Your threads will now be thread pool workers, capable of handling an HTTP 1.1 request (parsed using a class you will create for this purpose). You will probably want to implement a thread-safe queue in which to place threads that are idle and can be used for incoming requests (how about your queue from Lab 1?). As discussed in Lecture 6, it's recommended that the incoming connections themselves be passed directly to a thread for handling, including reading/writing requests/responses and manipulating the underlying ThreadSafeKVStore(s).
3. You must find and/or implement a class that can compute the MD5 hash of a byte sequence that you can use in your program. Please be aware of the terms of the licenses on available code, and don't use something with virality that requires you to publicly open-source your own code! I don't recommend you implement your very own MD5 hashing class unless you really, really want to.

4. You must implement a class capable of reading a very restricted subset of HTTP 1.1 requests, parsing what they contain, and constructing a response. The requests to be handled:
    a. `GET /key HTTP/1.1`: Search the K-V store for `key`, and return it in a `200 OK` response if it exists. Return a `404 Not Found` response if it does not exist.
    b. `POST /key HTTP/1.1`: Insert the `key` with the value found in the POST request body. Return a `200 OK`; an empty response body is acceptable. If the key already exists, replace it, and still return `200 OK`. To make this process more computationally expensive, also compute the MD5 hash of the value and store it, either in the same appropriately typed ThreadSafeKVStore as the value itself, or in a separate store. Be aware of concurrency concerns, including whatever invariants you define for the globally visible key-value pair and the value's globally visible MD5 hash. Note that you need not provide a way to get this MD5 hash out again in this phase of the lab.
    c. `DELETE /key HTTP/1.1`: Delete the key from the K-V store. Return `200 OK` if it was there, or `404 Not Found` if it did not exist.

    See Appendix 1 for a recommended (but not required) skeleton of what your server should look like.

Testing: You will learn to use the `httperf` tool to test your implementation; further details will be provided in Lecture 7.

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: November 13, 2017, 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 "lab2". Please also send me and the grader 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.

Appendix A: Suggested (*Not* Required) Code Structure

- **Main function:** responsible for parsing command line argument(s), creating one or two ThreadSafeKVStores (one for actual keys/values, one for keys/MD5 hashes of values, or one that maps keys to both values and MD5 hashes of values), initializing and calling ThreadPoolServer.
- **ThreadPoolServer:** class to create threads, pass them the ThreadSafeKVStores, create threads-safe thread pool/queue, creating a listening socket, and waiting for requests. Each time a connection request arrives, it can pull an idle thread out of the thread pool/queue and pass it the connection. Its responsibility ends at that point.
- **HTTPReq/HTTPResp:** Parses HTTP requests and generates HTTP responses (provided)
- **Threads:** Each individual thread in the pool may be an instance of a class if desired.