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. It will operate over an approximation of the HTTP protocol, allowing data to be read (GET) and written (POST). 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. Implement HTTP 1.1 GET/POST/DELETE frontend to field requests.

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 strongly 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. As discussed in Lecture 6, it's recommended that incoming connections be passed directly to a thread for handling, including reading/writing requests/responses and manipulating the underlying ThreadSafeKVStore.
3. 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`.

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 of this assignment for a transcript of sample request/response pairs. See Appendix 2 for a recommended (but not required) skeleton of what your HTTP request/response framework should look like.

Testing: You will learn to use the httperf tool to test your implementation; further details will be provided in Lecture 7. We may also provide a testing framework to verify components of your lab, like your ThreadSafeKVStore, at least one week before the Lab 2 deadline.

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 3, 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 "lab2". 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.

Appendix 1: Request-Response transcripts

Request	Possible Response	Possible Response

`GET /user1024 HTTP/1.1`
`Some random header`
`Another random header`
[blank line]
	`HTTP/1.1 200 OK`
	`Content-Length: 5`
	[blank line]
	`abcde`
		`HTTP/1.1 404 Not found`
		`Content-Length: 0`
		[blank line]

`POST /user2048 HTTP/1.1`
`Some random header`
`Content-Length: 10`
`Another random header`
[blank line]
`Abcdefghij`
	`HTTP/1.1 200 OK`
	`Content-Length: 0`
	[blank line]	

`DELETE /user65536 HTTP/1.1`
`Some random header`
`Another random header`
`[blank line]`
	`HTTP/1.1 200 OK`
	`Content-Length: 0`
	`[blank line]`
		`HTTP/1.1 404 Not found`
		`Content-Length: 0`
		`[blank line]`

Appendix B: Suggested Code Structure

- **Main function:** responsible for parsing command line argument(s), creating ThreadSafeKVStore, initializing and calling ThreadPoolServer
- **ThreadPoolServer:** class to create threads, pass them the ThreadSafeKVStore, 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.
- **HTTPRequestParser:** Reads the full text from an HTTP request and parses it. Should return the type of request (GET, POST, or DELETE), perhaps as an enum member. If a GET, should return the key; if a POST, the key and value; if a DELETE, the key. Needs to (1) parse the first line of the request, specifying the type of request, (2) parse the header lines looking for a Content-Length field and saving it, and (3) find where the body starts (after a blank line), and read Content-Length bytes out of the body.
- **HTTPResponseBuilder:** Builds a response with (1) a given status code (eg, 200 or 404), (2) a given status message (eg, “OK” or “Not found”), (3) and a given body (which may be empty). It needs to add at least a Content-Length field to the header, as required by the HTTP 1.1 spec. Students may choose to add additional headers if desired.
- **Threads:** Each individual thread in the pool may be an instance of a class if desired.