Assignment 3
CMPE 105: Principles of Computer System Design, Winter 2019

Due: March 1 at 5:00PM

Goals

The goals for Assignment 3 are to modify the HTTP server that you already implemented for Assignment 2 to support the use of a key-value server, rather than the file system, for storing objects that are PUT on the server, and to make some other minor changes that will be listed below.

As usual, you must have a design document and writeup along with your `README.md` in your `git` repository. Your code only needs to build `httpserver` using `make`.

Programming assignment: adding key-value store to multithreaded HTTP server

Design document

Before writing code for this assignment, as with every other assignment, you must write up a design document. Your design document must be called `DESIGN.pdf`, and must be in PDF. You can easily convert other document formats, including plain text, to PDF. Scanned-in design documents are fine, as long as they're legible and they reflect the code you *actually* wrote.

Your design should describe the design of your code in enough detail that a knowledgeable programmer could duplicate your work. This includes descriptions of the data structures you use, non-trivial algorithms and formulas, and a description of each function with its purpose, inputs, outputs, and assumptions it makes about inputs or outputs.

In particular, this assignment involves *serializing* data—transforming data into a format that can be stored and later loaded. The *format* is vitally important. Please describe, in detail, the format used by your key-value store, including (but not limited to), for the hash table entries, the sizes and fields in each entry, and the order in which the fields appear. Consider this from the perspective of someone who is given your design doc and a copy of a persisted key-value store data file: they should be able to decipher the file and read data out of it.

Write your design document *before* you start writing code. It'll make writing code a lot easier. Also, if you want help with your code, the first thing we're going to ask for is your design document. We're happy to help you with the design, but we can't debug code without a design any more than you can.

You **must** commit your design document *before* you commit the code specified by the design document. You're welcome to do the design in pieces (*e.g.*, making the small changes first), as long as you don't write code for the parts that aren't well-specified. We **expect** you to commit multiple versions of the design document; your commit should specify *why* you changed the design document if you do this (*e.g.*, "original approach had flaw X", "detailed design for module Y", etc.). **If you commit code before it's designed, or you commit your design a few minutes before the working code that the design describes, you will lose points.** We want you to get in the habit of designing components before you build them.

Program functionality

You may not use standard libraries for HTTP or the key-value store; you have to implement them yourself. You may use standard networking (and file system) system calls, but not any `FILE *` or `iostream` calls except for printing to the screen (*e.g.*, error messages). You may use open-source hash functions (such as `cityhash`), but you must cite your sources.

Your code may be either C or C++, but all source files must have a `.cpp` suffix and be compiled by `clang++`.

Details on the HTTP functionality your server must support are available in Assignment 1, but your server must now also support the `Content-Length:` header which specifies the length of the data in the message in bytes. We expect that you'll build on your server code from Assignment 2 for Assignment 3. Remember, however, that your code for this assignment must be developed in `asgn3`, so copy it there before you start.

Content-Length header

Your HTTP server must support the `Content-Length:` header. This header is widely used to specify the length of the *data* in a GET or PUT message. Doing so is important for the key-value store because it may need to store object length for blocks, so it should know this information ahead of time. It also becomes possible to response to put requests after all data is received.

The content length header looks like this, and comes *on its own header line* and *before the* `\r\n\r\n` that ends the header.


```
Content-Length:  800\r\n
```

The length specifies the number of bytes that follow `\r\n\r\n` in the message; this doesn't include the bytes in `\r\n\r\n`. The bytes may have any value (including 0). With content-length, and example request looks like:

```
PUT 12345678abcdef...  HTTP/1.1\r\n Content-Length:  800\r\n\r\n
```

where `1234567890abcdef...` is an httpname from previous assignments. An example response looks like:

```
HTTP/1.1 200 OK\r\n Content-Length:  800\r\n\r\n
```

Note that content-length only needs to be present in a PUT request header, and in a GET response header. A GET request header and a PUT response header both do not transfer additional file data with them, and so content-length isn't necessary.

The content-length header will allow your server to send the HTTP response header for PUT requests *after* it receives all file data and stores it to persistent storage. **This is vital functionality** since if the server is killed after the client gets the response, we expect the data to be correctly persisted and available to GET requests when the server is restarted.

Key-value store

Your HTTP server from Assignment 2 used the underlying file system for storage. This can be somewhat slow. Instead, for Assignment 3, you're going to store objects in a single file, organized as a *key-value store* (KVS). This means that, within the single file, your program must keep track of variable-sized objects and which blocks of the file belong to which objects.

Your key value store should support (at least) these functions:

```
ssize_t kvwrite(const uint8_t * object_name, size_t length,
                size_t offset, const uint8_t * data)
ssize_t kvread (const uint8_t * object_name, size_t length,
                size_t offset, uint8_t * data)
ssize_t kvinfo (const uint8_t * object_name, ssize_t length)
```

You may, of course, write functions underneath these that help them work. For example, we suggest writing a version of these functions that requires that the request be block-aligned (a block is 4 KiB). If you feel your design could be improved by modifying the interface or design, you may do so, as long as you *carefully* document it.

The name of the key-value store is provided to `httpserver` via the `-f` *`filename`* argument. This specifies a file that will contain *all* data for *all* objects, named by `httpname`, that your server will save from PUT requests and serve for GET requests. If the file doesn't yet exist, `httpserver` must create it and do any initialization necessary. If the file exists, and contains objects, we should be able to GET them—that is, data saved during PUT requests should be available if we restart the server (just as it is now). The server will respond to PUT requests *only* after it has received all object data and persisted that data within the key-value store file. Once the client gets the response header, we should be able to kill the server and bring it back, and see the object data from the last PUT.

Since you're storing all of these objects in a single file, you'll need to have some organization. There are many approaches to designing this, however we suggest:

- The file is organized into two parts, a hash table-like *index*, and a data region, where you pre-allocate the first part of the file to have the hash table, where each entry in the hash table contains information necessary to locate a block of an object within the key-value store.
- The hash table is keyed by object name and block number, where the object name is 20 bytes (remember, a name is 40 hex characters, but that can be converted to 20 bytes).
- Each entry in the table contains the name (20 bytes), a 4 byte block number (indicating the block of the object), a 4 byte object length, and a 4 byte number that points to the data for this block, for a total of 32 bytes per entry.
- When allocating a block for an object (which you will need to do when updating an object), keep track of the current end of the key-value store file. You can use this to decide where the next data block should go. **Keep in mind** that this must be thread-safe, otherwise two different objects could both use the same data block!
- The entry for the 0'th block of an object should store the overall length of the object. This means that you won't necessarily use the object length field every time. This is fine. You may use this field for other (non-zeroth) blocks if you think you can improve the efficiency of your system, but it is not required.

- The `kvread` and `kvwrite` functions read and write data respectively for a given object. The range of bytes read may start some offset within a block, and the range may cover multiple blocks and end not block aligned. Your KVS should be able to handle this.
- The `kvinfo` function returns the length of an object (needed for content-length) when the `length` argument is -1. If the length argument is greater than or equal to 0, it *sets* the length of the object to the length argument. When setting the length, if the object does not exist, it creates it. When getting the length, if the object does not exist, it returns -2 (same as read; see below).

You will have a maximum of 800,000 objects and 800,000 blocks in your server—yes, that's a lot, but it's far fewer than "real-world" servers need to support. This means you could have one 800,000-block objects, or 800,000 one-block objects, or anything in between. Daniel will go over suggested approaches in section the week of February 18—**it's vitally important that you attend section for hints on how to do this assignment**.

Keep in mind that object length might not be a multiple of 4096 bytes, so your key-value store will have to keep track of each object's length. For read, if `kvread` specifies a length that would read beyond the end of an object, read as many bytes as you can and return the number of bytes actually read. If the *offset* is beyond the end of the object, return -1. If the object doesn't exist at all (the name is invalid), return -2.

Please note that, while this is the *suggested* design, you may tweak it, or implement a different one, as long as you *clearly* document the interface and implementation. Your key-value store code should be able to stand on its own, without the `httpserver` around it, and should allow any code interacting with it to read a range of object data, write a range of object data, create an object, change the length of an object, and determine the length of an object. Note that the above design hits each of these requirements (`kvread` and `kvwrite` for reading and writing data, and `kvinfo` for creating, changing the length of an object, and determining the length).

You may want to write a small program to load data into this file, or to dump data out. If you do so, you may turn in the code for these as well, but **this is optional, not required**. However, you'll find that it's a very good idea to have standalone code that you can use to examine your key-value store file.

As with Assignment 2, your design and implementation for Assignment 3 must be thread-safe, but you need not worry about data consistency—that is, two threads updating the same object should not cause race-conditions in the organization of the KVS and in the hash table, but may cause the object data itself to contain partial data from each update.

README and Writeup

Your repository must also include a README file (`README.md`) writeup (`WRITEUP.pdf`). The README may be in either plain text or have MarkDown annotations for things like bold, italics, and section headers. **The file must always be called `README.md`**; plain text will look "normal" if considered as a MarkDown document. You can find more information about Markdown at `https://www.markdownguide.org`.

The `README.md` file should be short, and contain any instructions necessary for running your code. You should also list limitations or issues in `README.md`, telling a user if there are any known issues with your code.

Your `WRITEUP.pdf` is where you'll describe the testing you did on your program and answer any short questions the assignment might ask. The testing can be unit testing (testing of individual functions or smaller pieces of the program) or whole-system testing, which involves running your code in particular scenarios.

For Assignment 3, please answer the following questions:

- Why use `size_t` (64 bits) for parameters, and then only support 32 bit object lengths internally?
- Perform the following tests twice, once using your code from Assignment 2 and once using Assignment 3:
    - Write a simple program or shell script to write 750,000 small objects, each 100 bytes long, to your `httpserver`. The objects may be empty, or they may be copies of the same data.
    - Start `httpserver`.
    - Time how long it takes to complete this task.
    - Write up the results, and explain them.
    - If it takes too long to complete for either task (about an hour), you may reduce the number of requests. However, you should explain *why* you were unable to finish in the allotted time.

Submitting your assignment

All of your files for Assignment 3 must be in the `asgn3` directory in your `git` repository. When you push your repository to GitLab@UCSC, the server will run a program to check the following:

- There are no "bad" files in the `asgn3` directory (*i.e.*, object files).
- Your assignment builds in `asgn3` using `make` to produce `httpserver`.
- All required files (`DESIGN.pdf`, `README.md`, `WRITEUP.pdf`) are present in `asgn3`.

If the repository meets these minimum requirements for Assignment 3, there will be a green check next to your commit ID in the GitLab@UCSC Web GUI. If it doesn't, there will be a red X. **It's OK to commit and push a repository that doesn't meet minimum requirements for grading.** However, we will only *grade* a commit that meets these minimum requirements.

Note that the *minimum* requirements say nothing about correct functionality—the green check only means that the system successfully ran `make` and that all of the required documents were present, with the correct names.

If you commit the file containing key-value pairs at any time, you will lose 40 points on the assignment. These are likely to be large files, and should never be committed!

You must submit the commit ID you want us to grade via Google Form, linked to the assignment page on Canvas. This must be done before the assignment deadline.

Hints

- Start early on the design. This program builds on Assignment 2. If you didn't get Assignment 2 to work, please see Prof. Miller or Daniel ASAP for help getting it to work.
- Reuse your code from Assignment 2. No need to cite this; we expect you to do so. But you must do your work in the `asgn3` directory.
- Go to section on February 19–20 for additional help with the program. This is especially the case if you don't understand something in this assignment!
- You'll need to use (at least) the system calls from Assignment 2. You may not need any additional system calls.
- Work through the design for the key-value store *carefully* before you start to implement it. Make sure you've answered any questions you might have in advance.
- Think about how you will access parts of the key-value store file. When you look up a block, for example, you'll need to access entries in the hash table. You probably don't want to read the entire hash table at once, but instead only the entry that you need to check during a lookup.
- Note that we do not require you to support deleting!
- Aggressively check for and report errors. Transfers may be aborted on errors.

Grading

As with all of the assignments in this class, we will be grading you on *all* of the material you turn in, with the ***approximate*** distribution of points as follows: design document (35%); coding practices (20%); functionality (35%); writeup (10%).

If you submit a commit ID without a green checkmark next to it or modify `.gitlab-ci.yml` in any way, your maximum grade is 5%. Make sure you submit a commit ID with a green checkmark.

Extra Credit

Extra credit is only valid if you get at least 80 points on the main part of the assignment. You must submit a commit ID to the Google Form that contains the "regular credit" assignment. You should include the extra credit commit ID for multiple packets on one connection in your writeup. We will grade the Google Form commit ID for the extra credit for speed.

I feel the need—the need for speed!

We will give **15 points** to the fastest `httpserver` in the class, and **8 points** to the next 5 fastest servers. Obviously, this only applies to servers that successfully finish the speed test. If you don't finish the speed test, or don't finish in the top 6, there's no penalty.

Hint: profile the parts of your KVS that are slow. How can you improve concurrency? How can you reduce the time required during calls to `kvread` and `kvwrite`?

