Assignment 1
CSE 130: Principles of Computer System Design, Fall 2019

Due: Sunday, October 27 at 9:00PM

Goals

The goal for Assignment 1 is to implement a simple single-threaded HTTP server. The server will respond to simple GET and PUT commands to read and write (respectively) "files" named by 27-character ASCII names. The server will persistently store files in a directory on the server, so it can be restarted or otherwise run on a directory that already has files. As usual, you must have a design document and writeup along with your `README.md` in your `git` repository. Your code must build `httpserver` using `make`.

Programming assignment: 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).

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.

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.*, overall design and detailed design of HTTP handling functions but not of the full server), 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; you have to implement this yourself. You may use standard networking (and file system) system calls, but not any `FILE *` calls except for printing to the screen (*e.g.*, error messages). Note that string functions like `sprintf()` and `sscanf()` aren't `FILE *` calls.

Your code may be either C or C++, but all source files must have a `.cpp` suffix and be compiled by `clang++` with no errors or warnings using the following flags: `-std=gnu++11 -Wall -Wextra -Wpedantic -Wshadow`

HTTP protocol

The HTTP protocol is used by clients and servers for a significant amount of web communication. It's designed to be simple, easy to parse (in code), and easy to read (by a human). Your server will need to send and receive files via `http`, so the best approach may be to reuse your code for copying data between file descriptors (which you should be well-acquainted with after `asgn0`!).

Your server will need to be able to parse simple HTTP headers; you're encouraged to use string functions (but not `FILE *` functions) to do this. The `http` protocol that you need to implement is very simple. The client sends a request to the server that either asks to send a file from client to server (PUT) or fetch a file from server to client (GET).

An HTTP header consists of one or more lines of (ASCII) text, followed by a blank (empty) line. The first line of the header is a *single line* specifying the action. A PUT header looks like this (note the blank line at the end):

```
PUT ABCDEFabcdef012345XYZxyz-mm HTTP/1.1
Content-Length: 460

```

The newlines are encoded as `\r\n`, and the data being sent immediately follows the header. The sent data may include any bytes of data, including NUL (`\0`). The `Content-Length` header line is optional; the client may send it to indicate how much data follows the header. if it's included, the server should stop reading data after the specified number of bytes, and look for another header (if the client hasn't closed the connection). If there's no `Content-Length` header, the server copies data until the client closes the connection (`read()` reads end-of-file). For the above example, the name that the server will bind to the data is `ABCDEFabcdef012345XYZxyz-mm`.

For GET, the header looks like this. There's no `Content-Length` header because the client doesn't know the length of the content (again, notice the blank line at the end):

```
GET ABCDEFarqdeXYZxyzf012345-ab HTTP/1.1

```

All valid resource names in this assignment *must* be 27 ASCII characters long, and must consist *only* of the upper and lower case letters of the (English) alphabet (52 characters), the digits 0–9 (10 characters), and dash (-) and underscore (_), for a total of 64 possible characters that may be used. If a request includes an invalid name, the server must fail the request and respond accordingly.

The server must respond to a PUT or GET with a "response", which is a response header optionally followed by data. An example response header looks like this:

```
HTTP/1.1 200 OK\r\n
```

The `200` is a status code—200 means "OK". The `OK` message is an informational description of the code. For example, the `404` status code could say "File not found". The server must fill in the appropriate status code and message. For a response to a GET request, the server must provide a `Content-Length:` line in the header, similar to the one shown in the PUT request. The header is followed by a blank line and, for a GET response, by the data that the client has requested. As before, the data may include *any* data, including `NUL` bytes.

You can find a list of HTTP status codes at:
`https://en.wikipedia.org/wiki/List_of_HTTP_status_codes`.

The only status codes you'll *need* to implement are 200 (OK), 201 (Created), 400 (Bad Request), 403 (Forbidden), 404 (Not Found), and 500 (Internal Server Error). You may use additional status codes if you like, but these are the only required ones. Look at the link above to determine when to use each one.

Your server will need to be able to handle malformed and erroneous requests and respond appropriately, without crashing. Note that a "bad" name is *not* the same thing as a valid name that doesn't correspond to an existing file. You may assume that a header will be no longer than 4 KiB, though the data that follows it (for a PUT) may be much longer. Similarly, response headers will be less than 4 KiB, but the data may be (much) longer.

HTTP server

Your server binary must be called `httpserver`. Your HTTP server is a single-threaded server that will listen on a user-specified port and respond to HTTP PUT and GET requests on that port. The address to listen to and the port number are specified on the command line. (Yes, it *is* necessary to specify the server address, since your computer has multiple Internet addresses, including `localhost`.)

The first argument to `httpserver` is the address of the HTTP server to contact, which may be specified as a hostname or an IP address; your software must handle either one. The second, optional, argument to `httpserver` is the port number on which to listen. If there's no second argument, assume the standard HTTP port, port 80.

Your server will use the directory in which it's run to `write(2)` files that are PUT, and `read(2)` files for which a GET request is made. **All file I/O for user data must be done via `read()` and `write()`.** As with Assignment 0, you may not allocate more than 32 KiB of buffer space for your program.

Testing your code

You should test your code on your own system. You can run the server on `localhost` using a port number above 1024 (*e.g.*, 8888). Come up with requests you can make of your server, and try them using `curl(1)`. See if this works! `curl` is very reliable, so errors are likely to involve your code.

You might also consider cloning a new copy of your repository (from GITLAB@UCSC) to a clean directory to see if it builds properly, and runs as you expect. That's an easy way to tell if your repository has all of the right files in it. You can then delete the newly-cloned copy of the directory on your local machine once you're done with it.

We are providing a service on GITLAB@UCSC that will allow you to run a *subset* of the tests that we'll run on your code for each assignment. You will be able to run this test from the GITLAB@UCSC server at most twice per day (days start at midnight), and (of course) you can only run it on commits that have been pushed to GITLAB@UCSC. Running these tests is completely optional, and we'll go over how to do it in section during the first week of class.

The GITLAB@UCSC test will cover at least half of the functionality points for this assignment, **but there will be additional test cases not covered by this service**, so you should still do your own testing.

README and Writeup

As for previous assignments, your repository must include (`README.md`) and (`WRITEUP.pdf`). 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 1, please answer the following question:

- What fraction of your design and code are there to handle errors properly? How much of your time was spent ensuring that the server behaves "reasonably" in the face of errors?
- List the "errors" in a request message that your server must handle. What response code are you returning for each error?
- What happens in your implementation if, during a PUT with a `Content-Length`, the connection is closed, ending the communication early?
- Does endianness matter for the HTTP protocol? Why or why not?

Submitting your assignment

All of your files for Assignment 1 must be in the `asgn1` 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 `asgn1` directory (*i.e.*, object files).
- Your assignment builds in `asgn1` using `make` to produce `httpserver`.
- All required files (`DESIGN.pdf`, `README.md`, `WRITEUP.pdf`) are present in `asgn1`.

If the repository meets these minimum requirements for Assignment 1, 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. **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.**

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.

Hints

- Start early on the design. This is a more difficult program than `dog`!
- Go to section on October 14th or 16th for details on the code you need to set up an HTTP server connection. While you'll need this code for your server (obviously), you can "include" it in your design with a simple line that says "set up the HTTP server connection at address X and port Y".
- You'll need to use (at least) the system calls `socket`, `bind`, `listen`, `accept`, `connect`, `send`, `recv`, `open`, `read`, `write`, `close`. The last four calls should be familiar from Assignment 0, and `send` and `recv` are very similar to `write` and `read`, respectively. You might also want to investigate `dprintf(3)` for printing to a file descriptor and `sscanf(3)` for parsing data in a string (*i.e.*, a buffer). You should read the `man` pages or other documentation for these functions. Don't worry about the complexity of opening a socket; we'll discuss it in section (see above). **You may not use any calls for operating on files or network sockets other than those above.**
- Test your server using an existing Web client (we recommend `curl(1)`. Make sure you test error conditions as well as "normal" operation.
- Aggressively check for and report errors via a response. If your server runs into a problem well into sending data in response to a `GET`, you may not be able to send an error header (the header may have been sent long ago). Instead, you should just close the connection. Normally, however, responses are your server's way of notifying the client of an error.
- The commit whose ID you submit on the Google form must contain the following files:

  ```
  README.md DESIGN.pdf Makefile WRITEUP.pdf
  ```

  It may *not* contain any `.o` files. You may, if you wish, include the "source" files for your `DESIGN.pdf` and/or `WRITEUP.pdf` in your repo, but you don't have to. After running `make`, your directory must contain `httpserver`. Your source files must be `.cpp` files (and `.h` files, if needed).
- If you need help, use online documentation such as `man` pages and documentation on `Makefiles`. If you still need help, ask the course staff. You should be familiar with the rules on academic integrity *before* you start the assignment.

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 (15%); functionality (40%); 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.