Assignment 1
CSE 130: Principles of Computer Systems Design, Spring 2020

Due: Monday, May 4 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”, and the command HEAD to provide information on existing files. Files are identified by names composed of up to 27 ASCII characters. 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 **should** commit your design document *before* you commit the code specified by the design document. You're encouraged to do the design in pieces (*e.g.*, overall design and detailed design of HTTP handling functions but not of the full server), leaving the code for the parts that aren't well-specified for later, after designing them. 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.). 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 will be provided with a sample starter code that contains the networking system calls you need to create a server-side socket. You may use standard file 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 must be C, all source files must have a `.c` suffix and be compiled by `gcc` with no errors or warnings using the following flags: `-Wall -Wextra -Wpedantic -Wshadow`. We also encourage you to reuse code from the previous assignment, ideally building a library of functions that you can reference.

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 requests; 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). The HEAD command should behave as GET except that the file is not sent.

An HTTP request consists of one or more lines of (ASCII) text, followed by a blank (empty) line. The first line of the request is a *single line* specifying the action. It is followed by one or more headers, which consist of key-value pairs separated by a colon (`:`), one per line. For this assignment most headers can be ignored, but you will need to process the "Content-Length" header. A PUT request to create a file to be named `ABCDEFabcdef012345XYZxyz-mm` looks like this (note the blank line at the end, indicated by two adjacent `\r\n`, and the slash (`/`) preceding the file name):

```
PUT /ABCDEFabcdef012345XYZxyz-mm HTTP/1.1\r\nContent-Length: 460\r\n\r\n
```

The newlines are encoded as `\r\n`, and the data being sent immediately follows the request. The sent data may include any bytes of data, including NUL (`\0`). The `Content-Length` header is always present in a PUT request; the client sends it to indicate how much data follows the request. After reaching the end of the request (the `\r\n\r\n`), the server should read the specified number of bytes, send the response, end the connection and wait for the next another request.

The data read should be saved to a file with the provided name. If such a file exists it should be replaced. For the above example, the name that the server will use to identify the data is `ABCDEFabcdef012345XYZxyz-mm`. Notice the name *does not* include the slash!

For GET, the request looks like this. There's no `Content-Length` header because there is no data following this request (again, notice the blank line at the end). Note that a real request might have other headers that you will ignore:

```
GET /ABCDEFarqdeXYZxyzf012345-ab HTTP/1.1\r\n\r\n
```

For HEAD, the request looks like this. It is the same as for GET, except for the command in the first line. The same observation about other headers still applies.

```
HEAD /ABCDEFarqdeXYZxyzf012345-ab HTTP/1.1\r\n\r\n
```

All valid resource names in this assignment *must* be up to 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. The resource names are always preceded in the requests by a slash (`/`), which is not part of the resource name and thus does not count towards the 27 characters limit. If a request includes an invalid name, the server must fail the request and respond accordingly.

The server must respond to a PUT, GET or HEAD request with a "response", which is a response line followed by necessary headers and optionally followed by data. An example response looks like this:

```
HTTP/1.1 200 OK\r\nContent-Length: 0\r\n\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. The server must always provide a `Content-Length` header; if there's no content following the header (as is usually the case for any request that's not a valid GET or HEAD), content length is 0. For a GET request for a valid file, the server must provide the correct `Content-Length` and follow the response (after the `\r\n\r\n`) with the data that the client has requested. As before, the data may include *any* data, including `NUL` bytes. For a

HEAD request, the response will be the same, including the `Content-Length` with the correct value, but the data will be absent.

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 request will be no longer than 4 KiB, though the data that follows it (for a PUT) may be much longer. Similarly, responses 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, GET and HEAD requests on that port. The port number is specified on the command line as a mandatory argument, so to run the server on port 8080 the user would type `./httpserver 8080` on the command line.

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 or HEAD request is made (even if HEAD does not send the file, you need to get its content length to build the response). **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.

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 (`https://forms.gle/8fz525tqBRsV6jRQ8`). 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 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`, `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 response (the response 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 `.c` files (and `.h` files, if needed).

- If you need help, use online documentation such as `man` pages and documentation on `Makefiles`. Check the questions on Piazza, and ask there if you don't find what you need. 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 (20%); functionality (70%); 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.

Starter Code

Here is a link to the HTTP server starter code:
https://git.ucsc.edu/mdcovarr/cse130-section/-/blob/master/week-4/server.c