WebServer

This assignment is extremely important – (nearly) every assignment after this one uses this one!

If you have bugs or missing features in this, you will need to fix them before you can continue on to new assignments. This is very typical in software development outside of school.

You must submit .c files. Any other file type will be ignored. Especially ".o" files.

You must not zip or otherwise compress your assignment. Brightspace will allow you to submit multiple files.

You must submit buildable .c files for credit.

Introduction

A web server is a program that connects to a port (networking), listens for requests and sends the appropriate file to the requestor. Web servers must respond to requests and create a response using the "HTTP protocol (https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol). For our web server, we will use only the most critical portions of the protocol:

A request (which you will receive) will consist of the word "GET", a space, a path to the file, a space and then the version of HTTP which is implemented (you can ignore this) all on the first line. There will be other lines to the request – you can ignore them.

A response (which you will generate) will consist of: the string "HTTP/1.0 ", followed by code (200 for OK or 404 for file not found), a new line, then "Content-Length: " + the number of bytes of the response followed by ***2*** new lines.

An example session follows. Red is the request, blue is the response:

```
GET /test2.html HTTP/1.1
Host: localhost:8000
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:72.0) Gecko/20100101
Firefox/72.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/
*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Connection: keep-alive
Upgrade-Insecure-Requests: 1
HTTP/1.1 200 OK
Content-Length:6

test2
```

Note a few things:

1) The "GET" has the file path, but it always starts with a / - you will need to remove that.
2) Before you output the data, you have to know the content-length. The POSIX function "stat" will get you that.
3) There are two carriage return/linefeed combinations between the HTTP response header and the data

Details

Create a new program. Your program should take a path as a parameter (and fail with a usage message if there is no parameter). That path will be the path that your web server will use to serve files. For example – let's say that the path is /home/mphipps. If I try to access localhost:8000/secret/data.html, your web server will look at: /home/mphipps/secret and try to serve data.html. If the file is there, it should return it with a "200 OK" message, as above. If the file does not exist, you should return a "404 Not Found" message with a pre-designed response (could be a file or could be a hard coded message).

Your program should bind/listen/accept on port 8000 (port 80, which is the usual for web applications, will be blocked by your firewall). You must use getaddrinfo.

Your web server should launch a new thread for each request (accept). Parse the "GET" and determine the path. Find the file on disk and return it with a "200 OK" if it exists. Return a "404" if it does not exist.

You must use POSIX commands for files (no fopen/fclose/etc) and pthreads for threading. Your web server should be an infinite loop of waiting for and processing requests.

Hints:

Don't wait – start this soon. This code isn't **long** but it is complex.

My code was around 100 lines. Yours should not be much longer or shorter.

HTTP requires \r\n as line endings.

I put all of the network setup in a single function – it either returns a socket or it exits the program.

I used strtok_r for parsing the headers. Don't use strtok – it is not thread safe.

On Windows/WSL (maybe others), quitting the program doesn't free the binding immediately. If you try to run the program again, the bind() may fail. If you let it sit for a bit (30 seconds or so), then the bind() will work again.

Rubric	Poor	OK	Good	Great
Comments	None/Excessive (0)	"What" not "Why", few (5)	Some "what" comments or missing some (7)	Anything not obvious has reasoning (10)
Variable/Function naming	Single letters everywhere (0)	Lots of abbreviations (5)	Full words most of the time (8)	Full words, descriptive (10)
Structure	Globals everywhere, indentation doesn't match braces {}, no helper functions (0)	Any 2 of: Too many globals Indentation wrong Missing helper functions (3)	Any 1 of: Too many globals Indentation wrong Missing helper functions (7)	Few/no globals, indentation correct, helper functions(10)
Threading	No threading (0)			One thread per request (10)
Networking	No networking (0)	Some attempt at setup/listening (5)		Setup, Listening and teardown work correctly, uses getaddr (20)
File Handling	None/uses "f" style functions (0)		Uses open/close/read (5)	Uses open/close/read/stat (10)
HTTP – 404	No attempt at HTTP (0)			Generates 404 when a file is missing with a reasonable message (10)

HTTP - 200	No attempt at HTTP (0)			Responds with the contents of a file with a 200 OK response when the file exists (10)
HTTP – parse header	No attempt at HTTP (0)			Parses header, finds “GET”, gets filename (10)