ASSIGNMENT 4

CS 361: Systems Programming / Spring 2023

Description

In this assignment, you will implement an HTTP server.

Notation

Line change

Line changes in HTTP are composed of the following characters: `\r\n`

This document denotes line changes with the character ▼

End of message

Messages **<u>do not</u>** use a NULL terminator '\0'. They end by not having any more data. This document denotes end of message with the character █

This character denotes "end of data", meaning "there is no data here", not even a NULL terminator character. Nothing.

Spaces

Spaces are part of the HTTP protocol. This document denotes spaces that are part of the protocol with the character •

HTTP requests

The HTTP protocol will be covered in class in detail. HTTP requests have the format:

```
<METHOD>•<URL>•HTTP/1.1▼
<HEADERS>▼
▼
<BODY>█
```

Method

Your server will support two methods: `GET` and `POST`. `POST` only works for the URL `write` (see below), `GET` works for all other URLs. `<BODY>` is only present for `POST` requests, and is empty for `GET` requests.

HTTP responses

The HTTP protocol will be covered in class in detail. HTTP responses have the format:

```
HTTP/1.1•<CODE>▼
<HEADERS>▼
▼
<BODY>█
```

200 OK

This response indicates that the resource was found, and the body has the data requested. It uses the mandatory header `Content-Length: <LENGTH>` with the length of the body, in bytes. Example:

```
HTTP/1.1•200•OK▼
Content-Length:•4▼
▼
pong█
```

400 Bad Request

This response indicates that the request was invalid, and does not send any data on the body. Example:

```
HTTP/1.1•400•Bad•Request█
```

404 Not Found

This response indicates that the resource requested was not found on the server. Example:

```
HTTP/1.1•404•Not•Found█
```

Supported URLs

Your server will support the following URLs:

- `/ping`
    - Return an HTTP response with the contents pong. More details about HTTP responses below. Example interaction:

Request	Response

`GET•/ping•HTTP/1.1▼`
`<HEADERS>▼`
`▼`
`█`
	`HTTP/1.1•200•OK▼`
	`Content-Length:•4▼`
	`▼`
	`pong█`

- /echo
  - Return an HTTP response in which the contents are the headers. Example:

Request	Response
GET•/echo•HTTP/1.1▼ Header: Value▼ ▼ █	HTTP/1.1•200•OK▼ Content-Length:•13▼ ▼ Header: Value█

- /write
  - Send some data to the server using POST. The server will keep that data for future read requests (see below).
  - Write requests have a mandatory Content-Length header, indicating the size of the data being posted.
  - The server should reply back with the data sent by the client. Example:

Request	Response
POST•/write•HTTP/1.1▼ <HEADERS>▼ Content-Length:•24▼ <HEADERS>▼ ▼ This is the data to save█	HTTP/1.1•200•OK▼ Content-Length:•24▼ ▼ This is the data to save█

- /read
  - Reads data saved earlier by a /write request
  - If there is no previous write request, read should return the contents <empty>.
  - Example:

Request	Response
GET•/read•HTTP/1.1▼ <HEADERS>▼ ▼ █	HTTP/1.1•200•OK▼ Content-Length:•7▼ ▼ <empty>█
POST•/write•HTTP/1.1▼ <HEADERS>▼ Content-Length:•24▼ <HEADERS>▼ ▼ This is the data to save█	HTTP/1.1•200•OK▼ Content-Length:•24▼ ▼ This is the data to save█
GET•/read•HTTP/1.1▼ <HEADERS>▼ ▼ █	HTTP/1.1•200•OK▼ Content-Length:•24▼ ▼ This is the data to save█

- `/stats`
    - Returns a page containing statistics about the execution of the server:
        - Requests: Number of requests served so far resulting in a 200 OK response
        - Header bytes: Number of bytes sent as header so far
        - Body bytes: Number of bytes sent as body so far
        - Errors: Number of requests resulting in errors so far
        - Error bytes: Number of bytes sent when handling errors
    - Note that the body uses regular line changes '\n'.
    - Example:

Request	Response
GET•/stats•HTTP/1.1▼ <HEADERS>▼ ▼ █	HTTP/1.1•200•OK▼ Content-Length:•66▼ ▼ Requests: 0 Header bytes: 0 Body bytes: 0 Errors: 0 Error bytes: 0█
GET•/stats•HTTP/1.1▼ <HEADERS>▼ ▼ █	HTTP/1.1•200•OK▼ Content-Length:•68▼ ▼ Requests: 1 Header bytes: 39 Body bytes: 66 Errors: 0 Error bytes: 0█

- Any other URL
    - Open the file specified by the URL and send the contents of that file to the client
    - The file should be in the same directory in which the server process was launched from
    - If the file is not found, the server should return a 404 error response (see above).
    - Example:

Request	Response
GET•/index.html•HTTP/1.1▼ <HEADERS>▼ ▼ █	HTTP/1.1•200•OK▼ Content-Length:•469▼ ▼ <contents of file>█

Limits

Headers should be only up to 1024 bytes. Longer headers should be trimmed at 1024 bytes. Posted data should be up to 1024 bytes long. Longer posted data should be trimmed at 1024 bytes.

You can expect up to 10 clients attempting to connect before dropping new connections.

Implementation guide

Doing the following will result in a simpler implementation:

- Save all HTTP requests and responses as global variables
  - (e.g., `static const char[] error404 = <ERROR 404 here>;`)
- Use a global variable buffer to read the client request, so all functions can have direct access to it
- Use a global variable buffer to keep the header, together with an int that tracks the size of the header
- Use a global variable buffer to keep the body, together with an int that tracks the size of the body
- Use a single function to send data to the client, by reading the buffers. This will make it easy to count bytes and requests.
- Use one function to handle each different type of requests described above. Functions should be small and fit comfortably in one screen.
- Strings may not be NULL terminated (i.e., they don't end in the character '\0'). Functions like `strstr` expects NULL terminated strings.
- Be sure to check that sending data does indeed send all the data you want. If not, be sure to send the remainder of the data.
- You should use functions from previous assignments (e.g., `snprintf` and `strtok` from A2)
- Don't attempt to transfer the whole file at once, send it in increments. Use a buffer to send each increment.
- Tests 9 and 10 require you to use a data-structure to keep track of the status of multiple file transfers. You can uses arrays, lists, structs, etc.
- Some useful functions:
  - `memcpy` copies memory between buffers as needed https://linux.die.net/man/3/memcpy
  - `strcmp` checks if NULL-terminated strings are equal https://linux.die.net/man/3/strcmp
  - `strncmp` checks if two strings have the same prefix https://linux.die.net/man/3/strcmp

Entry Point

You will modify file `a4.c` with your solution and implement each function as defined above.

Due Date and Resubmission Policy

This assignment is due on **April 1st 2023** (Saturday) at **5pm CST**. There is no late policy.

The code and date used for your submission is defined by the last commit to your Git repository.

To resubmit this assignment, you are required to have attendance to Lab Sessions 7, 8, and 9. You can resubmit your assignment until **April 8th 2023** (following Saturday) at **5pm CST**. Together with your resubmission, you will have to submit a written description of what you changed from the original submission (on Gradescope).

Bonus Points

This assignment has a total of **25% bonus points**:

- 10% can be earned by using Piazza as described in the syllabus. Your posts should be public, tagged with the `assignment4` label, and non-anonymous to the instructors to count towards the bonus. You can claim bonus points through **a Gradescope quiz**.
- 10% can be earned by completing Lab Sessions 7, 8, and 9.

Submission and Grading

This assignment is submitted through Github, and has an automatic grade component of 100%. You can check your current grade at any point by submitting your code and checking the autograder. The automatic grade is determined by 10 tests, that will check if your submission outputs the expected result. Each test is worth 10%.

Errors and Omissions

If you find an error or an omission, please post it on Piazza as soon as you find it.

Hardcoding and Academic Integrity

Any hardcoding will result in a 0% grade. Hardcoding is when you submit code that detects which test is being run, and simply outputs the expected result. For instance, detecting that test 22 is running, and replacing the usual execution of your submission with `printf("expected result").`

The academic integrity policy described in the syllabus applies to this assignment. You are responsible for writing all the code that you submit. We will use an automatic tool that detects plagiarism on all submitted code, and we will investigate all instances where plagiarism is more than likely.

Please refer to the syllabus for the full academic integrity policy.