Diary

Initial Steps

The first steps I took towards creating the server was creating an appropriate `Makefile`. This was important as rather than having to remember a long and difficult to remember command which only becomes more difficult to remember as the number of source files increases, I could type `make all` and be left with a binary to run. I decided upon the structure of having an obj folder inside my project root to contain all the .o files and to have the executable placed in the project root, as this seemed the cleanest solution to me. To keep my code as tidy as possible, I made use of multiple source files to logically separate the code.

Creating a TCP server

The next item on my agenda was the create a simple TCP server which could echo back any data sent to it. To understand how to piece together the socket API and the correct way of handling multiple threads, I read through Ian's network code and also various sources online. After this, I was able to create a TCP server which made use of threads to handle multiple connections at once. An initial flaw I encountered was when I ran the program with valgrind, I encountered multiple memory errors, so I corrected the errors in my code and made a note to ensure that I used valgrind throughout my development to find memory errors and fix memory leaks.

Reading and parsing a HTTP request

From the offset, it was clear I was going to need a way to store multiple items from the HTTP request in a data structure to keep track of them ready to form the HTTP response. As a result, I created the `http_header` struct to store data extracted from the header of the request. This was modified accordingly as more data was processed, so for example, when trying to read the file, if the file is not present then the `status` field of the `http_header` struct was changed to `404`.
Following on from this, the next decision was on how to read the request coming from the client. The method I chose was to read character by character, as this allowed me to easily detect a CRLF (\r\n) and process the header line to that point. This solution worked out for me in the end, although I had some trouble with valgrind informing me of invalid read/writes, which I corrected before moving on to the next task of sending a response.

Sending a HTTP response

Sending a response consisted of first determining what should be sent in the header of the response. It became apparent that the resource needs to be read from the server first, so it could be determined whether a `404 Not Found` response needs to be sent. After trying to read the file into a buffer, I attempted to craft and send the appropriate headers for the assessed HTTP status code. After sending the request line, the content length and the appropriate content type, the response body was also sent, which may either be a file from the server or an error message such as "404 Not Found".
This worked in the browser. It was now possible to retrieve a web page from the server. However, when I hovered over a link using Firefox the server received a connection and then crashed. After researching the symptoms, I found that Firefox has the ability to open a TCP connection speculatively to save time when the hyperlink is clicked. This would not send any data apart from opening the connection, and this cause my server to crash. After this diagnosis, I pinpointed the exact problem to be writing to the socket. After googling this, the solution seemed to be to handle

the BAD PIPE signal using , which I placed in my main method. After this, all writes which failed returned -1 and were handled accordingly.
I also encountered the server crashing when sending bad requests using netcat. The problem was if I missed out the resource, version, or both, I would receive errors because I used them later in the code without checking if they were null. To fix this, I ensured bad requests were handled appropriately and sent the correct HTTP response.

Implementing SSL

After I had a working HTTP server, I implemented SSL as one of the extensions. This was relatively straightforward, and involved wrapping the client socket in an SSL wrapper. Then I made a call to SSL_accept to handle the SSL protocol, and from then on the only change that needed to be made was to replace the `read` and `write` calls with `SSL_read` and `SSL_write`. For testing, I used a self-signed certificate which required me to add an exception in the browser, as otherwise the browser would warn me of this security issue and not allow me to view the page.

Final Thoughts

I found this exercise to be an interesting way to begin to understand network programming in C, as well as gain an understanding of the HTTP protocol. I faced various memory problems along the way, especially when dealing with strings, which I may not have faced in a language such as Java. I believe this was beneficial to my learning however, as I was able to practise my lower level programming knowledge. If I were to redo this exercise, I would consider rewriting the methods inside `send_http_response.c` as I feel these could be done in a more modular and simpler way.