Overview :

The aim of this project is to code with socket programming, multi-threading, and the HTTP protocol. Your task is to write a basic HTTP server that responds correctly to a limited set of GET requests. The HTTP server must return valid response headers for a range of files and paths. Your HTTP Server must be written in C. Submissions that do not compile and run on a cloud VM, may receive zero marks. You must write your own HTTP handling code; you may not use existing HTTP libraries.

Details:

Your task is to write a simple HTTP Server. There is only a limited range of content that needs to be served, and it only needs to respond to GET requests for static files. You can assume that GET requests are no longer than 2 kB in size; note that this is larger than a typical MTU, and so you must be able to read in a multi-packet request.

Example content that needs to be served is:

Content	File Extension
HTML	.html
JPEG	.jpg
CSS	.css
JavaScript	.js

For these extensions, the correct MIME type should be reported (se https://mimetype.io/all-types). For all other extensions, or if there is no extension like Makefile, you can return a MIME type of application/octet-stream. You can assume that the requests are plain ASCII; you do not need to deal with Internationalized Resource Identifiers [https://en.wikipedia.org/wiki/Internationalized_Resource_ Identifier] and you do not need to deal with %-encoding [https://en.wikipedia.org/wiki/Percentencoding].

The minimum requirement is that the server implements HTTP 1.0, and as such, your server does not have to support pipelining or persistent connections. Your server should be able to handle multiple incoming requests by making use of Pthreads (or similar concurrent programming technique) to process and respond to each incoming request.

Your server program must use the following command line arguments:

protocol number: 4 for IPv4 and 6 for IPv6

port number

string path to root web directory

The port number will be the port the server will listen for connections on. The string path to the root web directory points to the directory that contains the content that is to be served. For example: /home/compproject/website

All paths requested by a client should be treated as relative to the path specified on the command line. For example, if a request was received with the path /css/style.css it would be expected to be found at: /home/compproject/website/css/style.css

The server must support responding with either a 200 response containing the requested file, or a 404 response if the requested file is not found. You do not need to handle requests with invalid headers (but the program should never crash), but paths to non-existent files may be requested, and if they are a 404 should be returned. Response headers should be valid and must included at a minimum:

Http Status

Content-type

<u>Program execution / command line arguments</u>

To run your server program on your VM prompt, type:

./server [protocol number] [port number] [path to web root]

where:

[protocol number] is 4 for IPv4 or 6 for IPv6

[port number] is a valid port number (e.g., 8080), and

[path to web root] is a valid absolute path (e.g., /home/compproject/website)

Task 1. *Server runs and sends valid responses*

Code must run on the marking VM without crashing (e.g., seg faulting) regardless of the inputs. Server sends a valid HTTP 200 response in reply to a GET request for an HTML file located in web root directory (not a sub-directory).

Server sends a valid HTTP 404 response in reply to a GET request for a file in the web root directory that does not exist or cannot be opened for reading.

GET requests using path components ../ should return a 404 error.

Task 2. *Server MIME types and paths*

Server sends a valid HTTP 200 response with the correct MIME type in reply to a GET request for a file located in web root directory.

Server sends a valid HTTP 200 response with the correct MIME type in reply to a GET request with a path below the web root for any of the specified file types (e.g. GET /css/style.css HTTP/1.0)

Task 3. *IPv6*

The server can accept connections and perform Tasks 1 and 2 using IPv6.

Task 4. *Build quality*

Running *make clean && make -B && ./server* should execute the submission. The server need not exit. The automated test script will kill it at the end of the tests.

Task 5. *Parallel downloads*

Server uses Pthreads (or similar concurrent programming technique, or epoll) to process incoming requests and sending responses. Do not *fork* multiple processes.
Server can process and respond to at least five HTTP requests concurrently. There is no need to limit this to only five.

__**Submission:**__

All code must be written in C (e.g., it should not be a C-wrapper over non C-code) and cannot use any external libraries, except standard libraries as noted below.

You may use standard libraries (e.g., to print, read files, create sockets, manipulate threads etc.). In particular, you may find *sendfile* useful. If you choose to use this, then provide a comment of at least two lines explaining the benefits of *sendfile* over the alternatives, in terms of both code simplicity and performance. Your code must compile and run on the provided VMs.
The repository must contain a Makefile which produces an executable named "*server*", along with all source files required to compile the executable. Place the Makefile at the root of your repository, and ensure that running make places the executable there too.

If you have implemented IPv6, include the line
#define IMPLEMENTS_IPV6

If you have implemented the multithreading extension, include the line

#define MULTITHREADED

Your server should not shut down by itself. SIGINT (like CTRL-C) will be used to terminate your server between test cases. You may notice that a port and interface which has been bound to a socket sometimes cannot be reused until after a timeout. To make your testing and our marking easier, please override this behaviour by placing the following lines before the bind() call:

```
int enable = 1;
if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &enable, sizeof(int)) < 0) {
perror("setsockopt");
exit(1); }
```

Make sure that all source code is committed and pushed. Executable files (that is, all files with the executable bit which are in your repository) will be removed before marking. Hence, ensure that none of your source files have the executable flag set. (You can verify this by cloning your repo onto your VM, and using ls -l.)

__**Testing:**__

You have access to several test cases and their expected outputs in a README file that I'll provide. However, these test cases are not exhaustive and will not cover all edge cases. Hence, you are also encouraged to write more tests to further test your own implementation. I will test the server myself via Continuous Integration Testing and will report what should be changed.