HTTP Web Server - Diary

George Taylor - 1522968

***Abstract*—This is the diary section for the HTTP Web Server assignment as part of the Networks module. It aims to give an insight into the process and issues I occurred during the development of the server.**

02/10/17

I have begun researching socket programming in C, I feel that it will be simplest to start the project in a multi-threaded fashion as I feel it will be easier than trying to adapt the server to accept multiple clients later on in the project.

I believe I have settled on an appropriate architecture for the system:

- The prgram starts, sets itself up as a daemon process and then enters an infinite loop that will listen for and accept client connections.
- WHen a client connects, fork the process and handle the connection in the new child process, return to listening for clients in the parent process.
- When the client communications finish terminate the child process.

At this stage I don't understand how the process can communicate with multiple clients when only one port has been given at the very start of the process. I remember learning about ephemeral ports at some point in my education so this is something I will have to research as I may need to handle this.

03/10/17

I have been able to successfully open and close a port and return a simple "Hello World" page. A this point the server ignores the content of the request method, however, I have created the relevant framework and methods to handle the parsing and response to requests.

This has been tested with simple curl requests to ensure the server is accepting the connection and it is now able to display the contents of the request.

07/10/17

When a client connects all of the details pertaining to that client are stored in a structure, and when the request is received it will be parsed into a `HTTP_request` structure. The request can then be passed through a collection of helper methods that will return an appropriate `HTTP_response` structure. The response can then be processed by the server and sent to the client. Separating the system in this way means that the helper functions can be responsible for dealing with requests and generating the appropriate responses without having to worry about the networking side, and the server functions can manage the networking without concerning itself with the contents of the request and response.

I have also been able to successfully implement forking a child process to handle parallel client connections. This may need to be looked at again when the server is set up to run as a daemon as I believe this will require another level of forking.

13/10/17

The server is now able to minimally parse the requests in order to determine the method and URI of the request. This can now be used to return a directory listing in the case that the URI refers to a directory. The links returned in the listing are currently broken as I will need to use the `Referer` field in the header to determine the actual path of the requested resource.

At all stages where I have needed to create a buffer to store a temporary value (such as when storing directory paths and file names) I have used a buffer size that seems appropriate to the situation. At some point later in the project I would like to replace these by more rigorous values that match with the standards used in the POSIX file system.

[15:09]

I have now been able to parse the `Host` and `Referer` header fields allowing me to accurately produce a directory listing with working links that allow the client to navigate through the file system.

[23:04]

After realising that the current set up of returning a page based solely on a directory path was causing an issue for scalability (I believe that every request should return a page not just a bunch of content) I have decided that when the user requests a directory `<directory>`, a redirect response to `<directory>/index.html` will be given. This means that whenever an `index.html` page is requested that doesn't exist the server can generate one with a directory listing. While this adds an extra layer of messaging required from a response it means that the flow of information between the client and server is more consistent and therefore more maintainable. Because of this it is now possible to serve any file in exactly the same way based on a check for the presence of an `index.html` file.

This new system works fairly well but when tested in Google Chrome it produces unreliable results. I believe this is due to the additional complexity of the requests that Google Chrome uses as it works seamlessly in Mozilla Firefox.

17/10/17

The server now correctly and reliably serves a directory listing and will redirect successfully if necessary. I have come to realise that my decision to separate serving a response from the generation of the response will not be appropriate for serving large files as they would need to be loaded into memory in their entirety. I have therefore changed the structure of the system such that the responses are now served to the client by the handling functions. This means that each function can serve the response in a manner appropriate to their task. This means that the helpers will no longer generate response structures but will instead receive a client structure hat they can use to write to.

19/10/17

The server is now able to successfully serve files of any size to the client by streaming them in chunks, thus eliminating the issue of loading large files into memory. After making so many structural changes to the code there was a lot of redundant and deprecated code lying around so I have done a significant clean up and added a considerable amount of error checking, however, there are still a lot of potential errors that I am not handling appropriately.

20/10/17

The server has undergone a major cleanup ensuring it operates as smoothly as possible. The process can now be started on any port, can accept any directory as the root, and can be started as a daemon process. A significant amount of extra error checking has been added (though still not enough) and several HTTP responses have been included to be more compliant with the RFC.

22/10/17

Receiving messages now works by polling the socket until there is something to be received. This allows the process to exit without the call to `recv(2)` blocking. This will eventually be extended to all blocking method calls on the sockets. Once a poll indicates that there are bytes to be read the server will make `recv(2)` requests until no bytes are returned.

23/10/17

The server has undergone another thorough clean up and a lot of error checking has been introduced to `server.c`. This will need repeating for the helper functions

28/10/17

The architecture has now been changed such that when the user requests a directory it will not redirect to an `index.html` but will instead just serve the listing straight from the directory. I made this change because it allows the user to receive a directory listing even if the directory contains an `index.html`.

I have also been able to successfully implement persistent connections. This allows a client to send up to `CONNECTION_REQUEST_LIMIT` requests to the server provided there is not a gap of more than `CONNECTION_TIMEOUT` milliseconds. This allows the server to operate more efficiently as it removes the need for the overheads when setting up and closing a connection.

I also discovered that the system had a fairly colossal memory leak. The receiving buffer was being allocated memory in the `serveClient()` method, the call to `receive()` would then `realloc(3)` the memory in order to grow the buffer to fit the entire request. However, the pointer to the start of the buffer was not being updated with the result of the call to `realloc(3)`. This lead to a very confusing memory leak because there would be times when the pointer didn't update (in the case of an in-place `realloc(3)`) and the original pointer would be `free(3)`'d successfully. Whereas if the pointer was updated it would result in LLVM reporting an invalid free and a memory leak. To fix this the `receive()` method now takes a pointer to the pointer that starts the buffer, this means that the contents of this pointer can be updated to wherever the new request buffer starts.

Terminating the program has been tidied significantly and polling has been introduced wherever necessary in order to ensure the successful termination of the program.

[19:53]

I have added the ability to provide response headers allowing the server to comply more closely with RFC 2616. This is particularly important as it means I can now provide a `keep-alive` response when handling persistent connections.

29/10/17

I'm not quite sure how it happened but I somehow managed to introduce a fairly major 'double free' error. I found it was caused by freeing uninitialised variables in the `request` struct. This should not have caused a crash however, so I can only assume this was some unwanted side effect of the undefined behaviour resulting from `free(3)`.

30/10/17

The server now sends the response content type as a header field and it has been tested on the SoCS lab machines.

[13:46]

I have added the ability to respond to `HEAD` requests and added several extra response headers including the current time and date formatted in accordance with RFC 7231. As a result of the well structured architecture this was a simple task as I only had to add conditions on whether to send a message body.

[15:45]

I have fixed an issue that was occurring when header fields in the `HTTP_request` struct were being freed while they were still filled with garbage data. This was leading to seemingly non-deterministic invalid pointer errors.

[17:30]

Fixed a pesky bug whereby images were being corrupted by a buffer overflow and added the ‘Content-length’ header to all error messages in order to prevent them from timing out.