Web Server Design Description

After initializing and binding to a socket used to communicate with the client, our server runs in a while loop while listening for connection requests. The server accepts each request and forks a child process to handle it by calling a function called serve_request(). The client web browser sends HTTP request messages to the server, and each child process in the server reads one request message into a buffer, prints the request message to the server Terminal, and determines which particular file is being requested via a function called parse_request(), which extracts the filename from the request message header. In order to allow for case-insensitive filename requests, the filename is passed into a function called get_true_filename(), which returns the exact case-sensitive name of the file stored on the server if the file is found.

If no matching file is found, the server sends a 404 error message to the client through the send_404() function. Otherwise, the server gathers information about the requested file, including its size and file type, and calls send_header() to send the header of the HTTP response message (with fields for Content-Type, Content-Length, and Date) to the client. Then, the actual data content of the requested file is read from the local directory and sent to the client in 4096-byte chunks (the reason for sending the HTTP response data in chunks is explained in the next section). Our web server can support .html, .htm, .txt, .jpg, .jpeg, and .gif files, so files of these types will be directly displayed in the client's browser. Any other file type is treated as binary data, so the browser will prompt the client to download the file to the local disk. After sending the HTTP response message, each child process frees all dynamically allocated memory, closes its file descriptors, and exits. The server's parent process, however, continues to fork and serve more HTTP requests. The server only shuts down completely when the termination signal Ctrl-C is entered into the Terminal running it.

Difficulties Encountered and Solutions

One major difficulty was discovering the reason for receiving Segmentation Faults in our web server program whenever the client web browser requested a large binary file. The program would crash because we attempted to store the entire HTTP response message in a single locally declared buffer; the data would exceed the amount of memory available in the stack, causing program abortion via Segmentation Faults. To solve this issue, we declared a smaller buffer and sent chunks of the HTTP response data incrementally by using a while loop that overwrites the data inside the buffer and sends a new chunk every iteration. This solution works because the client browser receives the Content-Length field in the HTTP response header, so it knows when the last chunk of the response message is received.

Another difficulty was understanding why our web server would sometimes receive automatic HTTP request messages asking for a file named 'favicon.ico' even though the client did not request for it specifically. Initially, our server program would crash after receiving multiple HTTP requests for 'favicon.ico' along with the request for the originally requested file. We learned that a favicon is a file containing a small icon that some web browsers (particularly

Google Chrome) display next to the address bar, and we handled requests for such files by responding with 404 file-not-found messages. This solution prevented the web browser from further asking for a favicon, and our program then functioned properly regardless of whether or not a favicon request was sent to it.

Manual for Compiling/Running Web Server

1. Preparation: All four files included in the submission tarball (webserver.tar.gz) should be inside the current directory. If not, place the tarball in the current directory and then extract the files by entering the following command into a Terminal:
   $ tar -xzvf webserver.tar.gz
2. **Compilation**: Enter the 'make' command into a Terminal to compile webserver.c (this will create an executable file called 'webserver'):
   $ **make**
3. **Running the Server**: Launch the web server by executing the 'webserver' program in a Terminal with the following command:
   $ **./webserver**
4. Requesting Files with a Browser: Our web server requires port number **2222**.Using a web browser on the same machine as the web server, enter the following URL to request a file in the current directory (without the '<' or '>' characters):
   http://localhost:2222/<filename>

   (Repeat Step 4 to request other files. Our web server utilizes a persistent connection, so you can request other files without re-executing the web server program. To close the web server, enter Ctrl-C into the Terminal window used in Step 3.)

Sample Outputs of Server and Client Browser

Say I have the files SPACE.jpg, test.txt, and CS118 Ch1.ppt in the same directory as my web server program. If I run the web server and enter the url http://localhost:2222/space.jpgon my web browser, then the server writes the HTTP request to the server terminal as shown in **Figure 1**,and the SPACE.jpg file is displayed by the Chrome browser as shown in **Figure 2**.

```
GET /space.jpg HTTP/1.1
Host: localhost:2222
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/65.0.3325.181 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9
```

Figure 1: Server terminal displays all HTTP request messages

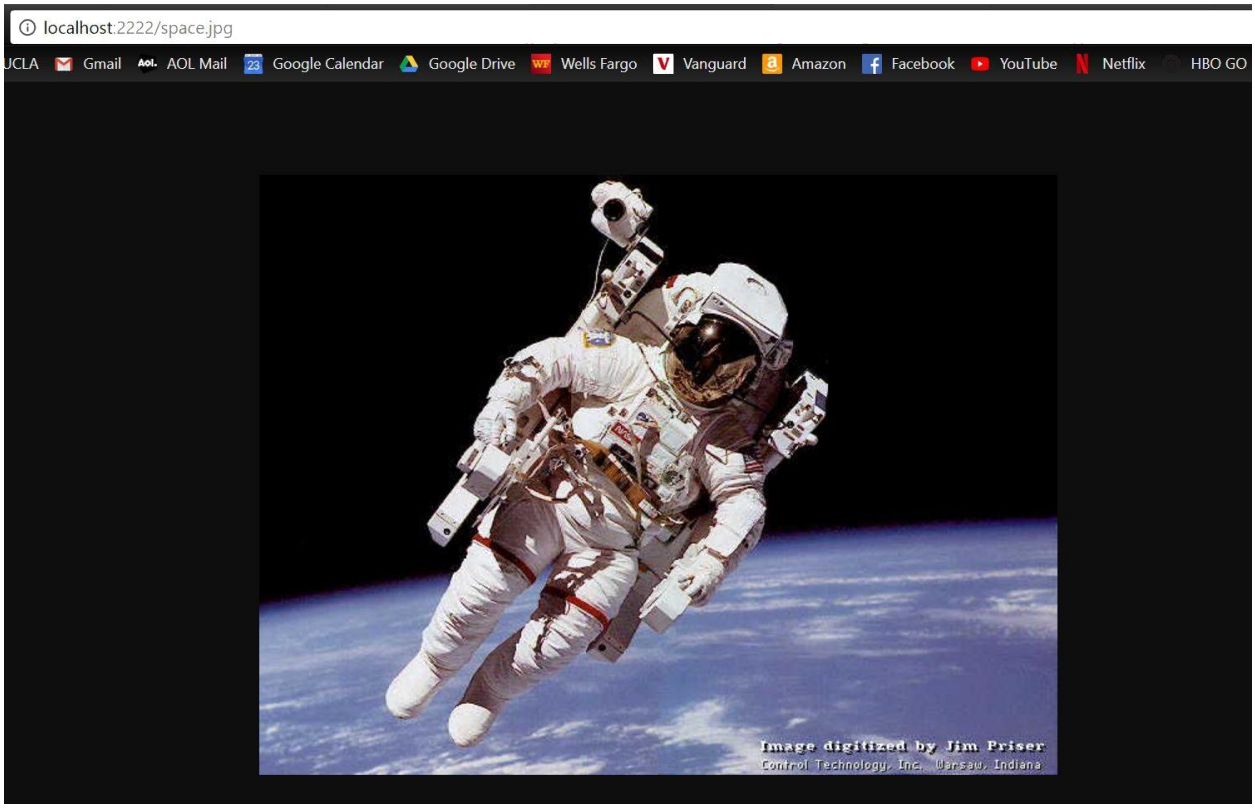

Figure 2: Image is displayed properly by Chrome browser

Similarly, if I request the file test.txt, the file is correctly displayed by the browser as shown in **Figure 3**.Requests for the file CS118 Ch1.ppt prompts a download by the browser as shown in **Figure 4**.

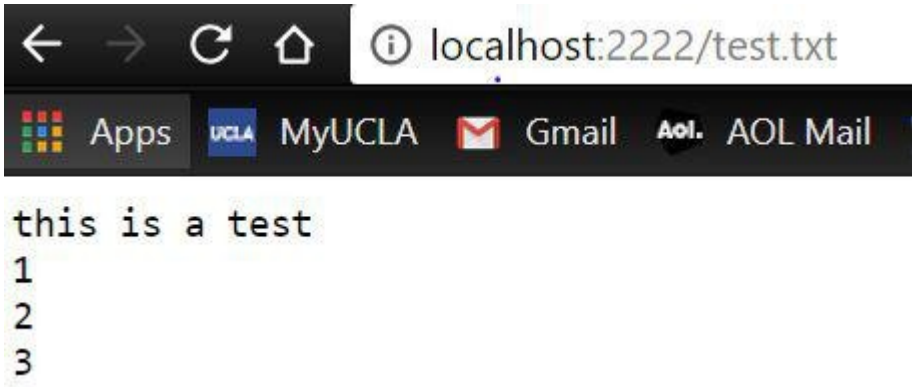

Figure 3: A .txt file is displayed properly

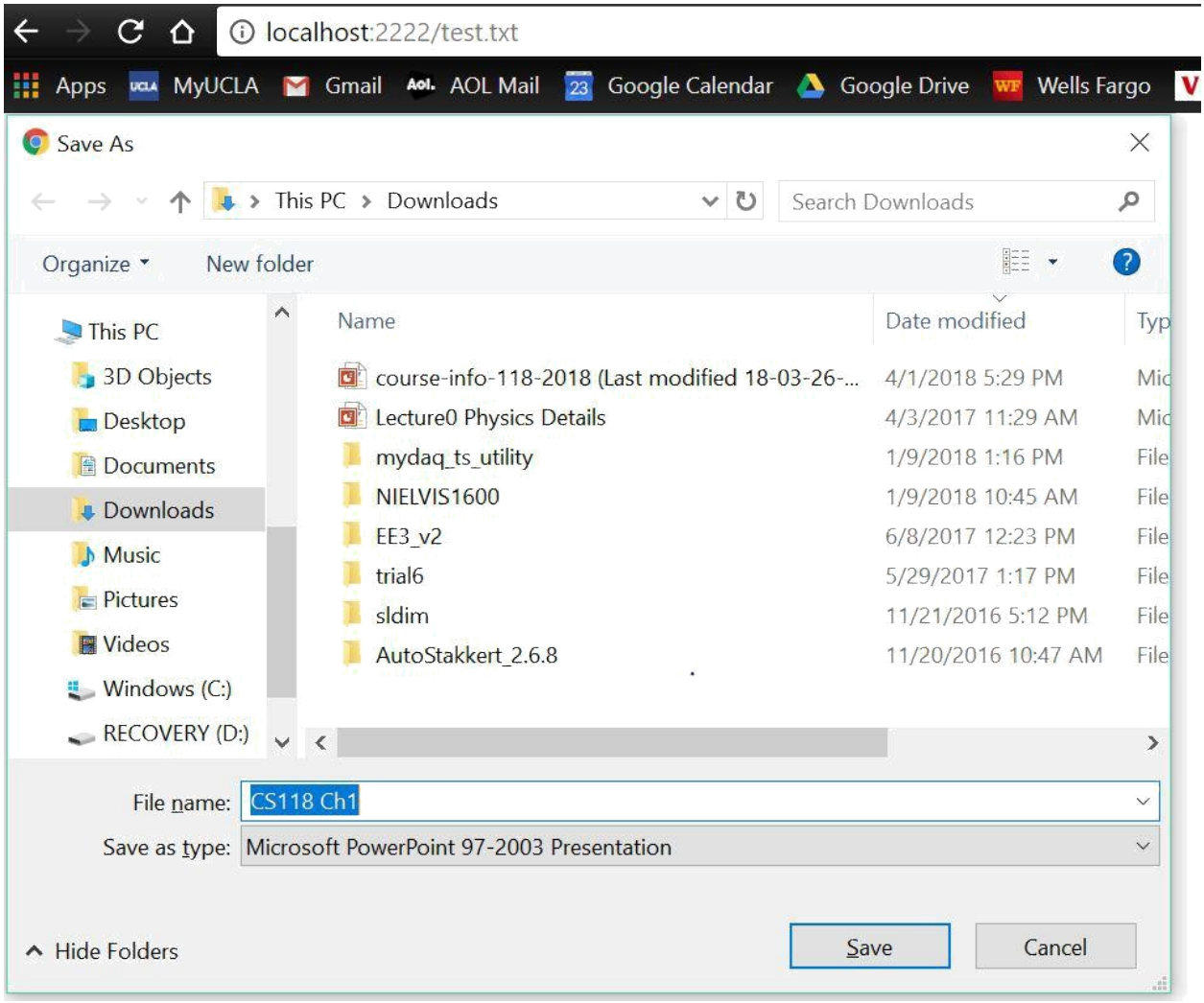

Figure 4: Binary files are downloaded

Finally, if a requested file does not exist, error 404 is returned by the server, and the message displayed in **Figure 5**is shown.

Figure 5: Error 404 is returned for files that could not be found