HTTP Server Design

Jake Armendariz

May 5, 2020

Modularized

If you look at my previous design doc and code, I changed a lot! Why, you may be asking did I decide to do this? I am scared about hw2 and thought I should follow TA advice and make the reading, processing and sending responses in their own functions. I think this will make it much easier during threading. Before a lot of the code was in the server file, I think this change will make threading much easier.

The previous code did not overwrite and truncate files. So I added O_TRUNC to my open function in PUT, this fixed the problem 👍

Resources

- To help with stat https://www.includehelp.com/c-programs/find-size-of-file.aspx
- To help tokenize strings https://www.tutorialspoint.com/c*standard*library/c*function*strtok.htm
- To help breaking request into key value pair https://stackoverflow.com/questions/7887003/c-sscanf-not-working
- I talked to Edward Chan a couple times about this assignment. We did not share code, but I wanted to mention this in case of similarities. We speicifcally talked about finding the double carriage and how to structure error messages.

socket/file functions

`int socket(int domain, int type, int protocol)` Creates a server socket. It will return a file descriptor for a new socket, -1 if it fails

`int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen)` binds a name to a socket. Returns 0 on success, -1 on failure

`int listen(int sockfd, int backlog)` marks the socket referred to buy socketfd as a passive socket, will be used by accept to get requests

`int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);` accept extracts the first connection request on queue, creates a new connectd socket and returns its file descriptor. We can use this as the file descriptor for our client. On failure it returns -1.

`int open(message->filename, O_RDONLY);` or
`int open(message->filename, O_CREAT | O_WRONLY | O_TRUNC, 0644);` open opens a file. Depending on the flags determines how I will use the file, the first instance is for a get request, where we have to only read file and send contents, the second is for a put request, where I want to CREATE, or OPEN, and O_TRUNC, overwrites the file.

`int read(int sockfd, const void *buff, size_t len) ssize_t recv(int sockfd, const void *buff, size_t len, flags)` both read in a file (sockfd) into a char or 8 bit unsigned integer array of size size_t and return the size of file that was read.

`int write(int sockfd, const void *buff, size_t len) ssize_t send(int sockfd, const void *buff, size_t len, flags)` both write to a sockfd, the only difference is send has flags to add ontop if something needs to be done differently.

`int fstat(int fd, struct stat *statbuf);` fstats returns information about a file, filling *statbuf with its contents

`int sprintf(char *str, const char *format, ...)` sends a formatted string to the first argument. I think this is what I should use for http responses

server.c

Contains Micheals socket code, and then calls all the functions to read and process request. All the work is done in these three lines

```
    read_request(client_fd, message);
    process_request(message);
    send_response(client_fd, message);
```

request.c

Contains implementation of the three fuctions named above. As well as the `void create_http_response(struct httpObject *message, char* response_type, int filesize)` and `void create_error_response(struct httpObject *message, int errnum)` which create the http response. This is the meat of the actual coding done in this project. I wanted to keep only the essentials in this file, library.c has helper functions which assist these and reduce coupled and repeated code.

library.c

Contains debuggin functions. print(char *str) that will use write(2) to write text, but only when I have debug on, this way I can have test cases all over my code without printing excess code. Also contains helper functions for validating and processing the request, including finding the header, validating the form of key-value pair and determining the request method (head, get and put)

http response headers

Inside of request.c I implemented a function called `void create_http_response(struct httpObject message, char * response_type, int filesize)` this function will take in the struct, error status, and content length and using sprintf, turn that into a string, which will be saved in message->response. I decided to handle the entire response in one function in an attempt to reduce any coupling. It allocates the string for response, and frees it in one swoop;

error handling

For error handeling. Any problems in the requests will be found in `int valid_request(char req[], struct httpObject *message)` which checks for key value pairs, a valid filename and content length if provided. After this I will send a 400 error if found with my `void error_response(int client_fd, int errnum)` function. Which formats a error depending on the errno value. If it doesn't recognize the error, sending to 500. I thought this would be a nice way to keep my errors neat, I have one function that handles all mal requests, this way, if any error occurs, I can just add in this function with errno value.

Reading Requests

To read http responses I have a function `void read_request(int client_fd, struct httpObject *message)` which initializes my httpObject. I send a string copy of my request `int valid_request(char req[], struct httpObject *message)` to my request to be digested by strtok, with each section verified to be in correct structure %s: %s with sscanf. I can also get the file size of header this way. I divded my read function into here so I could modify the request string using strtok, but keep it in tact for error handeling purposes. (in case we grow the project, I want to be able to use the request more, and it not be broken after checking format)

After reading the request and classifying it's method, I send to switch case statement inside of server.c I understand having this not be in a modular function is less pretty to look at. But I wanted my server.c to be the caller of all request.c functions. In this way I can understand where the error is happening, in my understanding/processung of the request/response: (request.c) or my socket usage, reading and writing files

(server.c)

Processing Requests

For head and get, I check the file for errors and create the http response inside the httpObject. For Put requests, I also check for errors with fstat in the file, but I check to see if the file either exists (and has write rights) or does not exisit. If either of these are false, I send an error response.

Sending Response

- Head: I send the response already created
- Get: Open file, send response header, send the bytes file in filesize/buffer_size packages.
- Put: Open file, overwrite and write the extra text that exisited in the header response, and all the excess

GET

GET and HEAD requests are already validated, thus I can open the file requested. If error, I will send to error*response function which will classify, 403 or 404. After this I can get the filesize with stat and st.st*size, and send proper http response. I cover how I read in large files below.

HEAD

Uses the same procedure as get, but will break out of case block once sending 200 OK or error.

PUT

Opens file with O*CREAT | O*WRONLY | O*TRUNC flags. The content length was already obtained in the read*request function. So all I have to do is read, and write until empty, returning errors along the way.

Reading and writing large files

I didn't do this properly in dog.c, I assume I lost some points for it. In this version I did get the size of the file using stat and struct stat st for determining the file size. Then I continue to read/write until it is empty (subtracting amount read from total each time until empty). I decided to use this instead of reading of unknown size because I wanted to decouple reading and writing from the header. This way, I could send the header of file size, before having to read and write.

Most recent redesign changes

I wanted to clean up my design doc and my code before submitting. I removed some repeat code, and I changed my HEAD request to send the file size.