Assignment 1: HTTP Server **Design Doc** **Emil Ingemansson (eingeman)**

Goals

The goal of this assignment is to create an HTTP server that will read in HTTP headers with GET and PUT commands that the server then executes. The server will then write back to the client with an HTTP header which contains a server code, and if necessary data from a file.

Design

1) The executable httpserver will take two arguments, the first being the address and the second optional argument being the port number. If no port number is specified, then the server will default to port 80.
2) Then, the program can set up a server using the system calls socket (2), bind (2), and listen (2) to create a server at the user specified address and port number if possible (This is essentially the code given on piazza).
    a. If any of these calls fail, the server can print the error and return code 500 (internal server error).

This point on, everything can be put into a loop so the server does not shut down after one request from the client.

3) Once our server is listening to client connections, the server can then call accept (2) which will return a *new_file_descriptor* which the server can use to write and read from the client.
    a. The client will have sent some sort of header (valid or invalid) and the server will need to parse through it for information it needs. Since we know that the first line must always be a GET or PUT with an address and HTTP/1.1, we can just read that in and parse through the rest looking for content-length because that's the only other header we care about.
    b. First, we will create a function to find the Content that takes an offset along with the file descriptor of where we're reading from and the buffer along with the address to variables we'll update in the function with whether its a PUT or GET and the address. This should be done in a loop until we reach the end of the http header. (Find \r\n\r\n)
        i. Once we read, we need to parse through each concurrent set of 4 characters to see if we can find a set of "\r\n\r\n". This can be done by starting our index at 3 instead of 0 and doing buffer[i-3], buffer[i-2], … buffer[i] so we do not get an out of bounds error. We can do something similar for content-length.
        ii. If we find the \r\n\r\n, we can exit the function knowing we know the offset of where it starts.
        iii. If we do not find the \r\n\r\n, then we can return an error 400 bad request from the client. (An incomplete header was sent).

    c. If the server can make the request, it will then check to see if the address is valid. The addresses are only allowed to be 27 characters long consisting of only lower case, upper case, numbers, '-', and '_'. This check can be put into a seperate function which returns a boolean true or false if the address is valid and takes in a single string argument to check if valid.
    d. If the function returns false, the server can then call an error and return a header with the code 400 (bad request).
    e. Else, the address should be valid and the server can then try to process the request at the given address.
4) If the request is a GET
    a. First, our server will try to open the file in read only with resource name that was taken from the HTTP Header.
        i. If this read returns -1, then the file does not exist and we can return our http header with status code 404 Not Found.
        ii. Else the file exists and we can perform the get for the client.
    b. If step 5a is successful, first we need to use fstat() to get the size of our file using the file descriptor we opened.
    c. Once we know the size of the file, we can send back our HTTP header to the client containing our status code and content length.
    d. After we send the header, we can then read from the file we opened and write the bytes we read back to the client.

5) Else if the client's request is a PUT
    a. First, the server will try to open the file and if that returns a nonnegative value, we know that the file already exists.
        i. If the file already exists, we will want to overwrite it so the server will just delete the file that already on the server and create a new file with the same address with what the client is sending. This can be done by specifying the O_TRUNC value in open which will truncate the value to zero when opening a file if it already exists.
    b. Then, we call open to make a new file on the server at the address (with the name) specified by the client with flag O_CREAT and O_RDWR.
        i. Here we can send back the address because we know we can read the bytes in. We will send back a 201 Created HTTP header because we will be making a new file in the server with the address specified by the user.
        ii. We can then we read in the data from the content that the client sent.

6) Then the client sent an HTTP header with a request that wasn't "get" or "put" and we then can send back an HTTP header with status code 400 bad request.

7) Once here, the server has finished the clients request and has already sent back the applicable HTTP error code. We can then close the *new_file_descriptor*.

This should be the end of the loop.

8) The server will now loop back to the stop of the while(1) loop and will process the next request sent from a client.