Asgn1 Design Document
Prepared by: Shirley Phuong & Miranda Eng
CSE130 Principles of Computer Systems Design
November 2, 2020

Program Description:
This program implements a single-threaded HTTP server that responds to GET and PUT commands to read and write files named by 10-characters in ASCII. In response to a GET request from a client, `httpserver` will detect a file name, read the file contents, write the associated HTTP response header, and, if appropriate, the associated content to stdout on the client side. In response to a PUT request from a client, `httpserver` will also take data from the request then create or overwrite the client specified file to put into the server directory and write the associated HTTP response header on the client side.

Design:
Our program does not require any data structures, however we utilize non-trivial formulaic-like structures in our code. We will mention these formulaic-structures. But first, we layout our basic program logic:

Program logic:
1. Get command arguments
    a. Check number of arguments
2. Setup TCP connection
3. Get HTTP request
4. Parse HTTP request header
    a. Check syntactic requirements (e.g. 10 character file name)
5. Handle GET request
6. Handle PUT request

Additionals:
- Handle basic errors to reduce noise and produce more elegant warnings and error messages
- Locate conditions requiring associated HTTP response and handle accordingly
- Add error state variable to prevent duplicate errors from producing excess in pipeline

Functions: `getaddr()`, `getStatus()`, `sendheader()`

Program Logic:
1. Get Command Arguments
Simple if statements. If 2 arguments, set default port number (i.e. 80). If 3 arguments, set specified port number. If not 2 or 3 arguments, ask the user to re-enter with proper usage syntax.

2. Setup TCP Connection
We follow the same familiar TCP client-side logic of `socket()`, `bind()`, initializing `servaddr`, `bind()`, `listen()`, `accept()`, `close()`, then looping back to `accept()`. Check for return values from these functions and handle them accordingly.

3. Get HTTP Request
To get the header of a HTTP request file, we use `recv()` to get one char at a time from the `commfd` socket. We receive in units of one character to prevent from reading past the message length. As we loop the `recv()`, we concatenate each character to a bigger string holding all of these characters. Once this larger string contains the carriage return `\r\n\r\n` (we detect using `strstr()`), that means we have reached the end of the header and must break to stop `recv()`.

4. Parse HTTP Request Header
In order to obtain parts of the header such as the request type and the file name, we used the `strtok()` function. This grabs the first two tokens in the header.

```
request = strtok(header2, " ");
fileName = strtok(NULL, " ");
```

To catch errors made by the client side, we use a multitude of if statements. In order to check if the file name is alphanumeric, a for loop is used to go through the file name and `isalnum()`, defined in the ctype.h header file, checks if the char is either an alphabet or a digit. To make sure the request type is HTTP/1.1, `strstr()` is used. The first parameter takes in the header and the second parameter takes in the string "HTTP/1.1" and looks for it in the header.

```
for (loop through file name) {
      if (!isalnum()) {
            // 400 Bad Request
      }
}

char* ptrhttp = strstr(header, "HTTP/1.1");
```

Other errors are also checked in the beginning such as the correct request type as well as the length of the file name provided. These are just checked with the `strlen()` and `strcmp()` functions. If we get any of these errors, the response is created and sent with our sendheader() function.
In addition to this, we include an integer called "error" to keep track of any errors we run into. This is to make sure multiple responses are not sent if there are multiple errors in the request. Two defined macros are created:

```
#define ERROR 1
#define NO_ERROR_YET 0
```

Once the server runs into an error we set the integer to 1 so that it cannot go into another if statement if the request contains more than one error.

5. Handle GET Request
Our code structure for handling GET requests is not very noteworthy, however it is a very important part of the program. We utilize `stat()` to get file size, `open()` the file, then append one character at a time to a larger string for file content, much like a variation of the structure of getting the HTTP request header. Then we call `sendheader()` to write a 201 Created HTTP response. We also need to check if the requested file does not exist (404 File Not Found) or if we don't have reading permissions to the file (403 Forbidden) and call `sendheader()` accordingly to write the appropriate HTTP response.

6. Handle PUT Request
If the client sends a PUT request, the first thing done is to check for the content length if it is provided. This tells the server how much of the body we want to write to a file. If the content length isn't provided, the entire body is written to the file. In order to implement this, an if-else statement is used. If the length is provided, a for loop is used to `recv()` the body and store it in a string array until that content length is hit. Otherwise, we `recv()` until the client closes the connection to signal that they have nothing else to send.
Once the body is obtained, we use `open(filename, flags, permissions)`. The flags we include are `O_CREAT, O_WRONLY,` and `O_TRUNC` and they are bitwise ORed. The `O_CREAT` flag creates a new

file if the file name doesn't exist yet. `O_WRONLY` provides write access and `O_TRUNC` ensures that the file is overwritten if it exists by truncating the length to 0.
The permissions for the file are `S_IRUSR, S_IWUSR, S_IRGRP,` and `S_IROTH. S_IRUSR` and `S_IWUSR` give the user read and write permissions. `S_IRGP` and `S_IROTH` gives the group and others read permissions.

```
putfd = open(filename, O_CREAT | O_WRONLY | O_TRUNC, S_IRUSR | S_IWUSR |
      S_IRGRP | S_IROTH);
```

Once the file is opened, we can write to the file with the content in the char array and then close the file. At the end we call `sendheader()` with the appropriate parameters.

At the end of our infinite while loop, we just `memset()` all of our char arrays to 0 and close the connection.

Functions:

- Converting hostname and IP address: `getaddr()`
  We also use a function our TA Daniel Santos Ferreira Alves provided:

  ```
  /*
  getaddr returns the numerical representation of the address identified
  by *name* as required for an IPv4 address represented in a struct
  sockaddr_in
  Input: string of hostname or ip address
  Output: unsigned long of address
  */
  unsigned long getaddr(char *name);
  ```

  This function encapsulates `getaddrinfo()` which converts text strings representing hostnames or IP addresses into a dynamically allocated linked list of struct addrinfo structures (a data structure used to represent addresses and hostnames within the networking API).

- HTTP Status Codes: `getStatus()`
  Two functions are created in order to handle the HTTP status codes and to send the response back to the client. The first function `const char* getStatus(int code)` is used to enumerate the status codes to string literals. This is implemented using a switch statement with several cases that signify the `200 OK`, `201 Created`, `400 Bad Request`, `403 Forbidden`, `404 Not Found`, and `500 Internal Service Error` status codes. The function takes an integer status code number and returns the corresponding status string. These status codes are used when we send the response back through the `sendheader()` function.

- Sending the Response: `sendheader()`
  The `sendheader()` function takes in the connection file descriptor, the char array storing the response, the status code, the content length, and the body of the response if there is one.

  ```
  /*
  This function formats a HTTP response using sprintf() then send()
  response to client
  Inputs:
  - commfd <- port # for accept() connection
  - response <- pointer to store HTTP response string
  - code <- status code #
  - length <- length of body of response (not the header)
  ```

```
- content <- the actual content; a pointer to string
Output: void
*/
sendheader(int commfd, char* response, int code, int length, char*
content)
```

Testing:
We used a combination of unit testing and whole system testing. To check for errors and proper http status code response handling, we used unit testing because we are more concerned with functionality with errors here. As for checking to see if our server replicates the basic behaviors of a http server with a curl client, we used whole system testing because we want to see if our program just works. These are the testing steps we took:

Unit Testing:

- check for correct number of arguments
- check if port number is set properly
- print full request header
- print parsed request type
- print parsed file name
- check for 400 Bad Request
    - check if request is HTTP/1.1
    - check if file is 10 characters
    - check if file is alphanumeric
- check for 500 Internal Service
    - check if using only GET and PUT HTTP methods (e.g. not HEAD, POST, etc)
- check for 404 File Not Found
    - If GET, check if file exists
- check for 403 Forbidden
    - If GET, check if file is readable
- If GET, print file content
- If PUT, print content length if provided & file content
- In terminal, `ls -l` to check file permissions: `rw-r--r--`

Whole System Testing:

- `curl http://localhost:8080/file -v`
- `curl localhost:8080/file[0-2] -v`
- `curl -T input http://localhost:8080/output -v`
- `curl -T input http://127.0.01:8080/output -v`

Assignment Question:
Q: What happens in your implementation if, during a PUT with a Content-Length, the connection was closed, ending the communication early? This extra concern was not present in your implementation of dog. Why not? Hint: this is an example of complexity being added by an extension of requirements (in this case, data transfer over a network).
A: When the connection is closed on the client side, the file to write to is truncated to a length of 0 and all of the data on it disappears. This is because the file is still opened after it leaves the while loop and the `O_TRUNC` flag in the `open()` function truncates all existing files to a length of 0. Since it has not yet fully received the body of the request, there is nothing to write into the file. The same thing happens if the file did not exist before. Our implementation of dog did not have this concern because `dog` did not require a client side to communicate with. It did not require us to code a TCP connection or establish a handshake so there was no stalling to listen.