Design for Asgn 1: Http Server

Part 1: Struct httpObject

A struct is used to store and maintain the data from the server and the client across all processes and functions, this will be stored in the httpserver.h file:

```
typedef struct httpObject
{
        char buffer[BUFFER_SIZE + 1];               // Header will not be larger than 4 KiB
        char method[METHOD_SIZE];           // valid method will only be PUT, or GET
        char fileName[FILENAME_SIZE];               // File name to be searched for or created
        char httpVersion[HTTPVERSION_SIZE];         // HTTP/1.1
        char statusText[STATUS_SIZE];               // "OK", "Bad Request", "Not Found", etc..
        char response[RESPONSE_SIZE];               // Contains the full response send to client
        int client_sockd;                           // socket the client is on
        long int contentLength;             // size of the client's file
        uint16_t statusCode;                        // 200, 201, 400, 403, 404, 500
} httpObject;
```

The data that is in this struct will be reused throughout this assignment to minimize repeating data, we used this struct to organize information.

Part 2: Implement Server Socket in Main

The data that needs to be added to create a successful server and client socket are these:

```
        constant char* port
        char* address
        int enable
        int server_sockd
        int ret
        httpObject msg (a struct in httpserver.h file)
        struct sockaddr_in server_addr
        struct sockaddr client_addr
        socklen_t client_addrlen
```

The port will store a specific port number depending on what the user uses as his localhost. The enable variable will help the setsockopt function. The address variable will hold the value of whatever the user wants to call the http server with.The server_sockd will hold the socket function value. The ret variable will hold the bind, setsockopt, and listen function. We will use the sockaddr_in to hold the values for the server address and the sockaddr to hold the client address that needs to be accessed. There will also need to be a socklen_t size to hold the correct address length for the client socket to be used.

A memset is used to set the server_addr to the size of the server_addr along with other calls from the struct sockaddr_in :

```
memset(&server_addr, 0, sizeof(server_addr)

server_addr.sin_family = AF_INET

server_addr.sin_port = hton(atoi(port))

server_addr.sin_addr.s_addr = getaddr(address)
```

Prior to connecting to the client server there will be checks to make sure certain system calls are used correctly, if not exit the code:

```
if((server_sockd = socket(...)) <= 0){
        print (error in socket)
        exit code
}
if((ret = setsockopt(...)) < 0){
        print (error in setsockopt)
        exit code
}

if((ret = bind(...) < 0){
        print (error in bind)
        exit code
}

if( (ret = listen(...)) < 0){
        print (error in listen)
        exit code
}
```

Once all checks are looked through the next thing is to connect to the client. A while loop will be used to run through the functions that are declared in Part 3:

```
while(true){
        AcceptConnection(...)
        ReadHTTPRequest(...)
        ProcessHTTPRequest(...)
        ConstructHTTPResponse(...)
        SendHTTPResponse(...)

        ...
}
```

What is left to add in the while loop is to memset the information and to close the connection between the server and the client:

```
memset(...)
close(client_sockd)
```

Part 3: Functions for HTTPServer

Prior to creating the process and execution of the HTTP Server, the identification by the “name” of the address is required for an IPv4 address which is represented in sockaddr_in. We will resolve this in this function:

```
unsigned long GetAddress(char* name)
```

In this function, provided by Daniel Alves’s sample code, we will use the addrinfo struct to help contain and access the desired address. The getaddrinfo and freeaddrinfo functions will be used as well:

```
if(getaddrinfo(...) != 0 || info == NULL){
        print error message
        write(...)
        exit(1)
}
```

After checking if there is an address info error, we store the address information in res:

```
unsigned long res
res = ((struct sockaddr_in*) info → ai_addr) → sin_addr.s_addr
freeaddrinfo(info)
return res
```

When the server sockets are all set up the next thing is to connect the client and server together. The entire process and execution of HTTP Server consists of these five high level steps:

1. Accept Connection – creates a connection between the client and the server
2. Read http message – parses through the clients request and checks if it is valid
3. Process request – stores the information in a struct
4. Construct response – constructs a response message based on the validity of the request
5. Send message – sends the response back to the client

In order to have these information pass accordingly between the client and the server, these 5 functions are organized to do that:

Part 3a: AcceptConnection

```
void AcceptConnection( httpObject* pMessage, struct sockaddr* client_addr, socklen_t*
client_addrlen, int server_sockd);
```

Create a client socket and check for errors.
The system call accept(...) is needed to accept the connection between the client socket and server, which is implemented in this function.

```
        client_socket = accept(...)
```

We check if there is an error with the client_socket when it’s value is less than 0.

```
        if(client_socket < 0){
                Contentlength = 0
                print error message
        }
```

Part 3b: Read HTTP Message

This function will check and store the method name, file name, and http version. If the request is a PUT then it also stores content length.

```
void ReadHTTPRequest( httpObject* pMessage);
```

The system call recv(...) is used to check if the message is received into the socket. We also make sure that the bytes read in this function is ssize_t.

```
        ssize_t bytesRead = 0

        if((bytesRead = recv(...)) < 0){
                print error (in recv(...))
        }else if (bytesRead == 0){
                print error (client is disconnected)
        }
```

To read the client that was requested, we read the information with the tool sscanf(...) and store them.

```
        sscanf(requestBuffer, “%s %s %s”, method, tempBuffer, httpversion)
```

With this we parse the information, but we need to check the buffer to see what is being read.

```
        if(buffer[0] == ‘/’){
```

```
        move information to the right location
}

if(check if tempBuffer == 10 ascii characters){
        copy filename and the buffer
}else{
        Print error with error code
}
```

Now we check to see if the put method was called:

```
if(method == put){
        store the content length
}
```

Part 3c: Process Request

This function rescans the stored content to make sure all information is stored correctly and is valid.

```
void ProcessHTTPRequest( httpObject* pMessage);
```

The information that needs to be checked is if the method name is valid, if the filename is valid, and if the HTTP version is valid.

```
if(method name is the method name){
        sscanf(error, …, messageStatus)
        print error code
        Send a response to the server
}

if(filename is the filename){
        sscanf(error, …, messageStatus)
        print error code
        Send a response to the server
}

if(httpversion is the httpversion){
        if( (compare string == httpversion)  !=  0){
                sscanf(error, …, messageStatus)
                print error code
                Send a response to the server
        }
}
```

Lastly, we look for the GET request:

```
if(method == GET) {
        if(file exist but empty){
                print success code
                sscanf(good, …, statusMessage)
        }else{
                print error code
                sscanf(Missing, …, statusMessage)
        }

        sprintf(reponse, Contentlength, httpversion, statusCode, statusMessage,
        content length)
}
```

Part 3d: Construct Response

This function will check that if no errors were thrown, a valid status code and status text are set according to the request.

```
void ConstructHTTPResponse( httpObject* pMessage);
```

The PUT and GET are checked first:

```
        if(method == PUT){
                sscanf(Created, …, statusMessage)
        }else if (method == GET){
                sscanf(OK , ..., statusMessage)
        }else{
                sscanf(Error, …, statusMessage)
                print error code
                write out the response for the error
        }
```

Now we check to see if the file exists and if it does grant access permission to read or write accordingly:

```
        if(fileExists){
                sscanf(OK, …, statusMessage)
                print success code
        }

        if(method == GET){
                ret = access(fileName, file_flag)
                if(file not in directory){
                        print error code
                        sscanf(Error, …, statusMessage)
                        send response to the server
                }else if(file access in wrong area){
                        print error code
                        sscanf(Error, …, statusMessage)
                        send response to the server
                }

                ret = access(fileName, read_flag)
        if(read is not ok){
              print error code
              sscanf(Error, …, statusMessage)
              send response to the server
        }

        ret = access(fileName, write_flag)

        if(write is not ok){
              print error code
              sscanf(Error, …, statusMessage)
              send response to the server
        }
  }
```

Part 3e: Send Response

This function handles the client requests (PUT & GET) and sends the final response as the final step.

```
void SendHTTPResponse( httpObject* pMessage);
```

To check if there are any errors or problems, we first check the string method is PUT to handle the request:

```
if(method == PUT){
      open file
      if(file doesn't exist){
            print error
            close file
      }

      if(file is empty){
            print success code
            sscanf(create, …, statusMessage)
            close file
      }
      ...
}
```

Now we read the buffer and write out what is given from the file. We check to see what is in the file or exit if there is an error:

```
bytesRead = read(...)

if(bytesRead is wrong){
      print error
}

ssize_t bytesWritten = write(...)
ssize_t totalBytes = bytesWritten

while(totalBytes < contentlength){
    if(bytesWritten is wrong){
        print error
    }else{
        bytesRead = read(...)
        bytesWritten = write(...)
        totalBytes = totalBytes + bytesWritten
    }
}

close(file);
```

We check if the request is GET and we do the same thing as PUT but we open the file and just read it:

```
if(method == GET){
    if(file exist){
        print success code
    }

    open file(READ)
    if(file has errors){
        close file
        print error code
    }else{
        ssize_t bytesRead = read(...)
        if(bytesRead has error){
            print error code
        }
        ssize_t totalBytes = 0
        ssize_t bytesWritten = 0
        while(totalBytes < contentlength){
            bytesWritten = write(...)
            bytesRead = read(...)
            totalBytes = totalBytes + bytesWritten
        }
    }
    close(file)
}
```

Part 4: Helper Functions for HTTPServer

These helper functions will be used to help organize the 5 main functions needed to run the client and server:

Handles functionality when clients request is GET:

```
void HandleGET( httpObject* pMessage);
```

This helper function is implemented in `void SendHTTPResponse( httpObject* pMessage)` to organize the read functions when GET is called in method.

Handles functionality when clients request is PUT:

```
void HandlePUT( httpObject* pMessage);
```

This helper function is implemented in `void SendHTTPResponse( httpObject* pMessage)` to organize the read and write functions when PUT is called in method.

Checks the validity of the client's file name:

```
int H_CheckFileName(char* name);
```

This helper function will loop through the given filename and check to see if it is a viable filename. This is achieved by checking each character in the filename to make sure each character is a legal character.

Checks the validity of the client's file method name. A valid method name will only be PUT or GET:

```
int H_CheckMethod( httpObject* pMessage);
```

This helper function will compare the string in the method to see if the string value is a PUT or GET. If it is PUT return 1 or if it is GET return 2.

Checks the validity of the clients http version. It should only ever be “HTTP/1.1” or else an error is thrown:

```
void H_CheckHTTPVersion( httpObject* pMessage);
```

This helper function is implemented in `void ProcessHTTPRequest( httpObject* pMessage)`. This will compare strings if HTTP/1.1 is part of the version. If it is not it will print out an error code with a bad request.

Calls either HandlePUT() or HandleGET():

```
void H_CallMethod( httpObject* pMessage);
```

This helper function is called in `void SendHTTPResponse( httpObject* pMessage)`. This just checks if the string is PUT or GET and calls the respected function. If there is an error, send an error code and a bad request.

Parses through the clients request and stores the content length:

```
void H_StoreContentLength( httpObject* pMessage);
```

This helper function is called in `void ReadHTTPRequest( httpObject* pMessage)`. This function uses the strtok() function to help store the content length.

This function is used if a file descriptor throws an error. The HTTP status code error is set automatically based on the errno:

```
void H_SetStatusOnError( httpObject* pMessage);
```

This helper function uses a switch case to organize the desired HTTP status code errors.

Different from H_StoreContentLength(),This function H_ReadContentLength() analyzes the file pointed to by stat() and returns the size by using .st_size. This is stored in the content length for future reference:

```
void H_ReadContentLength( httpObject* pMessage);
```

Simply checks if the file exists. Makes use of stdbool.h:

```
bool H_CheckFileExists( httpObject* pMessage);
```

Sets the response that will eventually be sent to the client. This is called frequently throughout the program:

```
void H_SetResponse( httpObject* pMessage);
```

Sends the response to the client:

```
void H_SendResponse( httpObject* pMessage);
```

This helper function uses the send() and write() function to send the desired response in the server. The send() value will be stored in ssize_t while the write() function will be stored in size_t.

Part 5: Testing Code

We tested our code by checking both GET and PUT input and outputs:

GET:

We tested GET with a zero sized file, a small text file, a large text file, and a small test file with ___ -- in the name. We also checked for a proper response with the 200, 400 error on a bad resource, 403 on a locked file, and a 404 error for a resource that doesn't exist.

PUT:

We tested PUT with a zero sized file, a small text file, a large text file, and a small test file with ___ -- in the name. We also checked for a proper response with the 200, 400 error on a bad resource, 403 on a locked file, and a 404 error for a resource that doesn't exist.

Both test cases went through and returned the correct content length as well as the proper server message response.

ASSIGNMENT QUESTION:

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).

If the connection is closed early during a PUT request the server will go in an infinite loop without any error checking stopping this process. The reason why it stays in an infinite loop is because the program will keep reading and writing from a socket that is closed making the while loop run forever. If there is an error checking process the files descriptor should print out "bad file descriptor" and will output HTTP/1.1 400 Bad Request. This should also stop the while loop and the content length will return 0.

There are no sockets in `dog`, so any socket error never occurred. Because we added socket connections in httpserver, a new level of complexity was created.