Jordan Mai
1645861

1. **Project Description:**
    a. A multithreaded http server will be able to receive multiple connections from a client instantly and have multiple processes running simultaneously to perform the action given from the client.This program that we are building will be able to read/write files that the client has sent over and must perform that action quickly while running other requests at the same time. The program is also responsible for logging any request and response actions in a log file specified by the client.

2. **Program Logic - PSEUDOCODE**
    a. **Stack Data Structure:**
        i. **Stack *s;**
        ii. **s->array[allocated memory] = shared buffer**
        iii. **s->top = top of stack ot be pulled**
        iv. **s->size = length**
    b. **Int main()**
        i. getopt(if the flags in the argument is -N or -l):
            1. If (N):
                a. The number of threads is N numbers
            2. if(l):
                a. Open log file for logging
        ii. for(i to n):
            1. pthread_create(handle_thread)

        iii. For i in range(n):
            1. Create pthread and send to handle_thread function
        iv. while(1):
            1. Accept connection and add to stack
        v. Close(connfd)

    c. handle_thread()
        i. while(true):
            1. Continue to try to get connection from stack
            2. If (stack is not empty):
                a. Lock()
                b. Pull from stack(top++)
                c. unlock()
                d. handle_connection(connfd)

    d. Adding_to_stack():
        i. lock():

ii. Add connection to next available stack spot
iii. unlock()

e. logging():
   i. lock()
   ii. Build log request based on success or failure
   iii. unlock()

f. Handle_connection
   i. While (able to receive bytes from client curl using recv()):
      1. Receive the request line and store in a buffer
      2. Parse each word separated by space. To obtain the request, body of file, version, content length and content. Then store each in a separate buffer.
      3. IF(request is not correct):
         a. Send 400 error
         b. Close connection
      4. IF(strcmp(request is “GET”)):
         a. Send to helper function for “GET”
      5. IF(request is “PUT”):
         a. Send to helper function for “PUT”
      6. IF(request is “HEAD”):
         a. Repeat “GET” but only send header and not body for response
      7. Logging()
      8. sleep(1);
   ii. Close(connection)

g. Handle_get(connfd, object):
   i. Parse object and remove the ‘/’ using memmove()
   ii. If(object isn’t ASCII # or letters or not 15 char long):
      1. Send 400 Error
   iii. Attempt to open file with the name of body for read only
   iv. IF(file don’t exist):
      1. Send “404 Not found error”);
   v. ELSE:
      1. Check for permission and send “403” forbidden if no permission
      2. Read content and store in a separate buffer
      3. Create a response header and send required code to the client.
   vi. Close file

3. Data Structures

a. Array- An array is used to store some byte amount to be used. We will use an array buffer to store bytes taken in from the client and also use arrays to write

information into. The array needs to contain all the information and is also used to build a string of response headers.

b. **Files -** A file is used to put bytes of content into to be kept outside of a program or in this case to be sent over to users. For this program, a file is used to read and write bytes from the client server.
c. **Stack -** a stack is used to put bytes into a data structure and can also be used to take things out easily with FIFO. The stack will be used to store connections and will be pulled out from the stack when needed to send to worker thread. There is a shared memory for this.

4. **Functions**
    a. **Int main(int argc, int argv[]):**
        i. This function opens a socket and attempts to connect to the same port as a client server. This function checks for any errors when reading from argv[] and makes sure that all arguments are correct. It will persist for a while loop to keep waiting and receiving connection until closing. The error to check for is the correct amount of argument inputs and if connection is possible. This function is also responsible for making the threads to be sent to the dispatcher function. Accepted connection will be sent added to the stack
    b. **Strtouint16(char numer[]):**
        i. Converts a string to a 16 bit unsigned integer
        ii. Return 0 if string is out of range
    c. **create _listen_socket(unit16_t port):**
        i. Creates a socket for listening to the port
        ii. Closes and creates an error message on failure
    d. **Handle_connection(int connfd):**
        i. This function receives every byte that is sent from the client server. It will continue to keep receiving. This function will handle all necessary requests (“GET”, “PUT”, “HEAD”) only . This function will parse through the entire buffer and take out necessary elements to be put into different buffers. The function continues with a while loop until everything is received from the client side. This program also checks for errors such as “400 Bad Request” if the request that was sent doesn’t match the appropriate components, or “501 Not Implemented if the request is properly formatted but request is valid but not necessary for this assignment. If everything works, this function will send separate helper functions for each different request. Generate a “500 Internal Server Error” if an error on the program.

    e. **send_get(int connfd,char*body,char*version):**
        i. This function will be responsible for handling “GET” requests, in which we will read from a file and write and send bytes to the client. Firstly, the

function will parse the first line and then attempt to open the body of the first line to read(). If a file exists, the function will read every single byte from the file and generate a “200 OK” response to be sent. If the file doesn’t exist, we will generate a “404 Not Found” request. If we don’t have permission to access a file, we will generate a “403 Forbidden. Once we build the response and send() to the client, we will return back to main().

f. **send_put(int connfd, char*body, char*version, char* content_num):**
    i. This function will be responsible for handling “PUT” requests, in which we will open a file for writing and write all the bytes to this file that was sent from the client. Firstly we parse the request line and attempt to open the body for writing. If a file doesn’t exist, we will create one and write all the bytes that were sent to the newly created file. We will then generate a “201 Created”. If a file already exists, we will override everything from the existing file and send “200 OK”. If there is no permission on the file we will generate a “403 Error”. Once we created our response header and send() to client, we will go back to main()
g. **send_head(int connfd, char*version, char*body):**
    i. Same as send_get, except when we build our response header, we do not include a body.
h. **Void handle_thread(void *arg):**
    i. This function will be responsible for pulling threads from the shared buffer to be passed into the worker function. This function will be stuck in a while loop to try to find a connection to grab from. Returns NULL.
i. **add_thread():**
    i. Pushes a connection to a stack

5. **Question**

a. **Repeat the same experiment after you implement multi-threading. Is there any difference in performance? What is the observed speedup?**

    i. **There is a significant speed up in time when running with the multi-threaded server in comparison with single-thread.**

b. **What is likely to be the bottleneck in your system? How do various parts (dispatcher, worker, logging) compare with regards to concurrency? Can you increase concurrency in any of these areas and, if so, how?**

    i. **The bottleneck of my system is that I am instantly creating a thread and am continuously trying to accept a connection to be worked on. Regarding concurrency, the dispatcher is using a shared memory and buffer to store connections that were accepted. The worker thread is concurrently waiting for a connection from the dispatcher buffer and simultaneously working with a thread. Logging is shared**

concurrently and each worker function is constantly logging their request to the same log file. Each thread is working simultaneously to log something to a shared space. We can increase concurrency by adding more threads per run of the program so there is more simultaneous process happening.

c. **For this assignment you are logging only information about the request and response headers. If designing this server for production use, what other information could you log? Why?**

    i. **We can also log all information that came with the request/response such as file type, connection information, name of server/client, etc. Also possibly having information about the content that was received or sent over will be useful in logging to see if information was correct. I also believe that time of work process can be helpful for debugging purposes and to check if there were any errors in the internal program.**

d. **We explain in the Hints section that for this Assignment, your implementation does not need to handle the case where one request is writing to the same file that another request is simultaneously accessing. What kinds of changes would you have to make to your server so that it could handle cases like this?**

    i. **We would just need to have a mutex lock to lock the file if and only if a process is writing to a file with the same name. We can also use pwrite to set the specific location for writing.**