Lab 2 Design Document

Shridhik John

Cruz ID: shjohn

CMPS 130, Fall 2019

1) Goal

The goals for Assignment 2 are to modify the HTTP server that I already have. I will be adding two additional features: multi-threading and logging. Multi-threading means your server must be able to handle multiple requests simultaneously, each in its own thread. Logging means that your server must write out a record of each request, including both header information and data (dumped as hex). I'll need to use synchronization techniques to service multiple requests at once, and to ensure that entries in the log aren't intermixed from multiple threads.

2) Assumptions

I assume I can use my code from Assignment 1. I also assume that I have to use mutexes and semaphores to achieve multithreading. I aso assume I'll have to use pthread_create to multithread. I assume I can find code online to convert ASCII to hex as well as resources to help me implement multithreading.

3) Design

My general approach for this lab was to start with the code from Assignment 1. The first part I'll have to work on implementing is reading the new command line input and parsing through the command

line input. Then I'll have to check to see if there is a –N or a –l, and if there is, I'll take the string right after. Then I'll have to convert those values from ASCII to integer. I should be able to get the filename for the log output, and also the thread length. With the thread length, I can create multiple sockets, and multiple threads. I'll have to then work on the logging feature. For this I'll have to have the file name, and send the buffer information into this file. The information will have to be converted from ASCII to hex. Then I'm going to parse through the hex values, and go print a new line after every 20 values. Then we'll prepend the leading zeroes and increment 20 to every line.

The variables shared between the threads are those that initialize the Queue, the filename. I use the filename variable in the connection and main functions. I use the threads in both of these functions as well. The critical region can be found where the Read and Write functions are called.

4) Pseudocode

```
void header(int handler, int status)
{// Check to see if file exists/is readable
        if exists send OK message
        else send appropriate Error message
}

void *connection (void *p) {

//if response is a Get
        // Get file name
         fname = strtok(NULL, " ");
         if (fname[0] == '/') fname++;

                //Check to see if file name is exactly 27 characters, consists of a hyphen,
                underscore, or alphanumeric value values
                // Check to see if file exists and is readable
                // If fails send error message from header function
                // else continue

        send(new_socket, buf, strlen(buf), 0); // Send to client
```

```
//if response is a Put
        // Get file name
         fname = strtok(NULL, " ");
         if (fname[0] == '/') fname++;

// Open a file and read/Write with this line command from Dog.c
        fd = open(fname, O_CREAT | O_WRONLY | O_TRUNC);
        recv(new_socket, buf, 50, 0);
        write( fd, buf, 50);
close(fd); // close file
```

int Main(int argc, char const *argv[])

```
{
//Check for max number of arguments from input and store it in "max_threads"
//  loop through all arguments based off of the argument count looking for –N and -l
for (opt = 0; opt<argc; ++opt) {
        // go through arguments and find "-N"
          if (strcmp((char *) argv[opt], "-N")==0){
                // max threads is equal to the value after -N
                max_threads = (char *) argv[opt + 1];
      }

        // go through arguments and find "-l"
        if (strcmp((char *) argv[opt], "-l")==0){
                 logenb = 1;
                // the filename for the log file is the name after -l
                filename = (char *) argv[opt + 1];
      }
   }

// convert the "max_thread" from ascii to integer value
  MX_threads = atoi(max_threads);
```

insert Geek for Geeks code to set up socket

```
 //looping through all the threads
for (TH_count = 0; TH_count < MX_threads; TH_count++) {
   new_socket[TH_count] = 0; // allows for multiple sockets
//initialize buffer
// Connect socket code to connect client with server


//Go through every line in the header
```

```
while(split_response != NULL)
 {
printf("%s\n",split_response); // prints every line in header
split_response = strtok(NULL,"\r\n");
}
split_response = strtok(response, " "); // split every line in the header/response
  while(split_response != NULL)
    {
 //    printf("%s\n",split_response); // prints every line in header
       split_response = strtok(NULL,"\r\n");

    }

    split_response = strtok(response, " "); //  divde the buffer by space

// Split response will store the information from the buffer on whether the call is a GET or PUT
command
```

5) Question

Testing: I performed a whole system testing by running the shell.sh script that was provided to us on the lab. I worked on multithreading first, and then tested multithreading. Then I worked on logging, and testing logging separately. Finally, I tested them both together. I have my Get requests working completely fine for both multithreaded requests as well as with a properly functioning log. I was unable to get multithreading completely working for PUT, and for the PUT statement's log I wasn't able to split each line with 20 bytes and add the leading zeroes. I tested this with multiple lines of code.

• Using either your HTTP client or curl and your original HTTP server from Assignment 1, do the following:

• Place four different large files on the server. This can be done simply by copying the files to the server's directory, and then running the server. The files should be around 4MiB long.

• Start httpserver.

• Start four separate instances of the client at the same time, one GETting each of the files and measure (using time(1)) how long it takes to get the files. Perhaps the best way to do this is to write a simple shell script (command file) that starts four copies of the client program in the background, by using & at the end.

• Repeat the same experiment after you implement multi-threading. Is there any difference in performance?

• What is likely to be the bottleneck in your system? How much concurrency is available in various parts, such as dispatch, worker, logging? Can you increase concurrency in any of these areas and, if so, how?

After Starting four separate instances of the client at the same time and using the GET command on each of the files and then implementing multi-threading, I can tell the performance is a lot better with the multithreading. Multithreading is a more streamlined, efficient process. The bottleneck in this would be the file that I'm calling.