Design Document for Assignment 0

Matthew Klein

Goal

The goal of this program is to build upon the functionality of assignment 1. We must use the simple httpserver we created in the last asgn to process multiple requests by creating a pool of threads to work on individual socket requests simultaneously. The program will also be able to handle logging the requests and getting the state of the server by calling healthcheck.

Design Specifics

The code I used for the first assignment must be modularized in order for it to work with a multithreaded server. Thus, the functionality will be similar, while more things will be stored inside a HttpObject struct. This allows us to pass around data between functions much more easily.

The function `getopt()` will be used to parse the program arguments to obtain the port number, the log file, if specified, and the number of threads the program will run on.

To parse the header a combination of `strtok_r()` and `strstr()` will be used. This will allow us to both loop through the tokens within a full request header, and split up each of the individual tokens into separate parts. Thus they can be easily manipulated and used later in the implementation of the requests. `Strtok_r()` is used as it is a re-entrant version of the `strtok()` function. This makes the function thread-safe and prevents conflicts between threads.

PUT, *HEAD*, and *GET* will all use the `fstat()` function, created using a stat struct, to obtain information about the file. This info will be used to check if the file exists, the access permissions of the server to the file, and the size of the file.

Both `open()` and `recv()` will be used in *PUT* in order to read the data part of the request from the client and write it to the file opened with `open()`. We will loop `recv()` for large files in order to fully read the data within.

GET will use `open()`, but will use `dprintf()` to send its response to the client, instead of directly writing or sending the response. This allows us to format the string being sent to the client, thus giving us the proper output. A loop `read()/write()` will also happen in *GET*. This ensures al lthe data is written to the client socket.

HEAD will use only the `fstat()` and `open()`, but will not use `read()`. This is due to the fact that *HEAD* only returns the header of the file, and not the file data of the file specified.

Before starting this functionality, we must first create a pool of threads from which requests can be taken. Each thread will run its own isntance of the functional `httpserver` from asgn1. Client sockets will be enqueued and dequeued from a circular buffer, all within a Mutex to prevent race conditions on the shared CB, as it is one of the only things shared between threads.

Once the thread has its client socket to accept from, it will read the request, process the request accordingly, log the header and/or data correctly, including errors, and will send the correct resposne back to the client socket.

Again, a majority of the program will be error checking and cathing all of the edge or corner cases.

Algorithms/Logic

We start by calling `getopt()` to parse the program arguments given to us. Inside a while loop, for every argument/option given on the command line, we will use a switch case. As the only options we are looking for are "N" & "l", we specify them in the parameters for `getopt()`. -N will bind to the number of threads requested to run the program with, while "l" will specify the log file to use for logging requests. We then open the log file with flags `O_RDWR|O_CREAT|O_TRUNC` and set the logging variable to 0 to signify that we will be logging. After we have finished parsing all of the arguments, we exit out of the loop. After this we can set the port number to `argv[optind]` as optind gives the last element of the array. The port number will get pushed to the back and be in `argv[optind]`, allowing us to access it. We terminate if no port number is specified.

```
while(getpot() != -1){
    switch(getopt()){
        case 'N':
        case l:
        case '?':
    }
}
port = argv[optind];
```

After parsing args we will initialize the necessary data structures and threads. To create the thread pool we loop the `pthread_create()` function for the number of threads given by the program args, or the default of 4 threads. Then we must initilize the circular buffer in order to be able to store our list of client sockets waiting for their request to be processed. We must also create the thread struct for each thread, which holds the circular buffer within it. After this we will loop and create the number of threads requested.

```
   for(ssize_t idx = 0; idx < num_threads; ++idx){
      pthread_create(&thread_ids[idx], NULL, thread_func, &args);
   }
```

Once the threads are runnign on their Thread functions, they will complete tasks, but I will address that later. After we have created these threads, we then must be able to add client sockets to the queue without any threads accessing the queue while in this process. Thus we will run inside a loop to accept each client socket that sends a request to the server. In order to make this critical region safe, we must have all parts of the queuing be completed within a mutex. Done through the use of mutex locks, this prvents other threads from modifiying the data structure before the thread that currently has the mutex lock is finished processing the data and completing the necessary tasks it has. We are only acquiring the lock before we go into the function

`enequeue()`, otherwise the thread will wait until it can enqueue again. This is done in the main thread.

```
      pthread_mutex_lock(&cb->mut);
      CBuff_SeqEnqueue(cb, client_sockd);
      pthread_cond_signal(&cb->cond);
      pthread_mutex_unlock(&cb->mut);
```

While the main thread is enqueuing client sockets to the buffer, all of the created threads are waiting for a signal from the main thread that an item has been enqueued to the CB, and it is possible for a thread to acquire the lock and start to process the client's request. This is done by locking and then

```
while(pcb->pq_size == 0){
      pthread_cond_wait(&pcb->cond, &pcb->mut);
   }
```

Where pcb is the CB p=assed in with the Thhread arg obj passed in. This sleeps the current thread while there is nothing inside the CB. Once the thread receives the signal to awaken, it does so and continues to work. After the signal is received, the thread tries to acquire the shared CB lock in order to dequeue an element and start processing the request from that client.

Here we can take a look at enqueuing and dequeuing from the circular buffer. We have two functions associated with the CB. First off is `enqueue()`. Since we are not returning anything, enqueue will be void, and will take in a CB pointer and a client socket to add.

```
void CBEnqueue(CB *cb, int client socket){...}
```

Enqueue will first check to see if the CB is full, if it is, it will unlock and sleep in order for a dequeue to occur and open a space to enqueue. After it relocks, it will continue. It then sets the current head pointer to the client socket value, and increments the current size of the queue. This adds the client socket to the queue, to be taken out later by dequeue.

```
CB->queue[head] = client_socket;
```

We then increment the tail pointer if we have enqueued into an empty queue instead of incrementing first inside the dequeue. Head is also incremented by one, unless it has reached the max size of the buffer. Then it will be reset to the the beginning of the buffer.

Dequeue has almost identical functionality to enqueue. However, we check for an empty queue before continuing. This is to prevent dequeue trying to remove an element the does not exist in the buffer. The value pointed to by tail is then dequeued from the buffer and placed into a variable to return after incrementing the tail pointer. Once we obtain a value, we set the tail to 0 and move it forward by one, unless it reaches the end in which it will be reset to the beginning like with the head ptr. Return the value.

After the signal is sent from enqueue, a thread will acquire the CB lock, and will dequeue an element which will become the client socket on which the functionality of asgn1 is completed. First, we will check that the parsing is correct and the

request passes the first checks like http version, file name length, and the file has only valid chars in its name. If any of these fail, this is the first time we will log. Logging will be discussed later, but if any of these checks fail, then we will send the error resposne and continue to the next iteration. We then decide which request we have obtained, and subsequently call the correct fucnction to process the request. `Strcmp()` is used on the message object's `method` field. Depending on the request type, we will perform slightly different actions on the message we received. The functionality of each process will be described below. While there are some changes to the code, a majority of it has stayed the same from the previous assignment.

The next step will be to parse the header for the information we require to fufill the request specified by the client. To do this we will need to use a combination of `strtok_r()` and `sscanf()`. Because of the way `sscanf()` works, we must first allocate static arrays to pass in the tokens of `sscanf()`, and also create an integer to store the content-length specified by the header. Then the first token is parsed from the header, delimited on "\r\n". After this, we enter a loop in which we check to see if the token, parsed by `strtok_r()` is not null. Within the loop, the first token is parsed, which will always contain the request type (e.g "PUT", "GET", or "HEAD"), the filename prefixed by a '/', and the http version. If any of these are not present or invalid, the error checking will occur after the loop. Once we have the first part of the header we continue to the next token, if the request type is of "GET" or "HEAD", we continue to the implementation of these requests. If the request type is of "PUT" we loop through the remaining tokens and check for a specified `content-length:`. If none is found, then the program responds with a 400 error. If content length is specified, then it is put into the variable already allocated for it. Onnce the header parse is done, we move on to the first error checking.

```
token = strtok_r(buffer, "\r\n");
while(token != NULL){
    parse first token using sscanf();
    if(request_type != PUT){
        break;
    }
    else{
        //check token for "Content-Length:"
        if(Content-Length: is in token){
            sscanf() for actual content length;
        }
        else{
            continue with token loop;
        }
    }
}
```

To validate the header and its information we must pass that data that we parsed from the header into multiple checks. The first check will be to check the length of the file name given. If this name is `> 27 chars`, then it is invalid. To do this, allocate a buffer of 29. If the last space in the buffer contains something, then the filename is invalid and the subsequent request cannot be completed.

```
if(strlen(filename) > 28){
    send err response 400;
```

```
}
```

After this we remove the beginning '/' from the file name using `memmove()`. Once we have the parsed file name, we then `strspn()` to compare the file name to an array of valid chars. If the return of `strspn()` is not the length of the file name, then it is a 400 error.

```
if(strspn(filename,valid_chars) != strlen(filename)){
    return a 400 response. "Bad Request";
}
```

We can then move on to the implementations of the 3 request types we are handling. Each of which will be contained in a separate 'if' statement.

Our first check if it is to *PUT*. First, we must allocate the necessary variables and buffers to be used within the implementation. We then open the file using `open()`, and create a stat object that contains all the information about the opened file.

```
open(parsed filename, OCREAT|O_WRONLY|OTRUNC, 0644);
struct stat fileinfo;
```

The flags OCREAT|O_WRONLY|OTRUNC, are used to allows the server to either create or overwrite a file, depending on the existence of the file. Then, using the stat object, we can pass in a FD which will allow us to access file information. Using this info, we first check if a file exists `fstat(FD,&fileinfo)==0`, and then check if the server has the correct permissions, in this case, write permissions. If the server doesn't have the coreect permissions we return a 403 and close the socket, wait for new request. A *PUT* request will never return a 404.

```
if(fstat(FD,&fileinfo)==0){
    if(server has correct perms){
        implement PUT;
    }
    else{
        return 403 error response, "Forbidden";
    }
}
else{
    continue;
}
```

First we call recv() again to get the data from the client, which we mostly be separate from the header, but will need to be separated in some cases. Once this recv happens again, we go into the separate cases. First if read returns 0, we PUT a file with 0 bytes onto the server. This is done with

```
write(FD, buffer, content_length);
```

If read is -1, the server sends a 400 response to the client. The next case is if the recv receives less bytes than the buffer size. If this is the case, we perfom write once to the opened file and then set the `message->status code` to 201 . We also close the FD before continuing with the listening loop to avoid memory leaks

```
if(bytes_read < buff length){
    write(FD,buff,content_length);
    drpintf(201 resposne, "Created");
    close(fd)
    close(client socket);
    continue to listen;
}
```

Lastly, if the buffer is full from the first `recv()` we must loop through the data sent by the client by calling `recv()` multiple times. It must be noted that the flag `MSG_DONTWAIT` must be specified, or otherwise the socket will be blocked and the prgram will hang. We do this, and subsequently write the file until the buffer is empty, or we have written the amount of bytes specified by the Content-Length provided in the header. We write first in the loop due to the original `recv()` call earlier in the function.

```
while(buff != 0 && total_written <= content_length){
    write to file;
    increment bytes_written;
    call recv() again;
}
```

Once this loop has finished, we have checked every case for *PUT*, and the server has sent the correct response. The server then closes the FD and the client socket, and then contiues to listen for more requests. We have completed *PUT*.

"GET" will complete similar tasks as "PUT". Instead of writing to a file, it will read a file, send a response, and write the corresponding data to the client socket. First will be the same permission checks as with *PUT*, only we will check if the file exists or not. We can do this using the same `fstat()` function as in *PUT*, but we will care about a -1 return for the first check, and throw a 404 error response if the file does not exist.

```
if(fstat(FD,&fileinfo)==0){
    if(server has correct perms){
        implement GET;
    }
    else{
        return 403 error response, "Forbidden";
    }
}
else{
    return 404 response, "Not Found";
}
```

The same checks from reading the file specified from the header will occur with `read()`. Only now we will read the data from the file and write to the client. If the client has data, it is checked with the stat struct done earlier. When the client has data, we set the status code as the `200 "Ok"` response before writing the data of the file to the client using `write()`. Again, if the amount of data in the file is larger than our buffer, we will loop the `read()` and `write()` for each instance of the loop. This will allow us to read the entire file without error. Once this porcess is done, we close the FD and the client socket, and continue to listen for more requests.

Head will constitute the same checks as with put. However, no data will be written to the client. Only a response will be returned. The same responses will be returned as *GET*, only without any data attached. We will ignore the rest of the requests as given per the specifications.

We will look at logging next. Logging is only done if there is a specified log file in the program args. A simple integer check is done to see if we complete logging or not for the specific request. I split up logging into two separate parts. Both parts will first acquire a global lock to determine and increment the global offset. This will allow the thread to complete its logging asynchronously as it only needs access to the shared resource while it is obtaining the inital offset. It will have calculated the amount to increment the offset before acquiring, as to limit the amount of time the thread has the lock, improving latency. To calculate I had to calculate the amount of bytes each file would take when converted to hex chars. This was done with,

```
    fullLine = (ssize_t) (floor(message.data_len / 20));
    ssize_t totalBytes = fullLine * 69;
    ssize_t extraLine = (message.data_len % 20);
    if (extraLine > 0) {
       extraLine *= 3;
       extraLine += 10;
    }
    totalBytes += extraLine;
    totalBytes += 9;
    snprintf(&response_buf[0], 49, "%s %s length %ld\n", message.method,
message.fname_parse, message.data_len);
    totalBytes += strlen(response_buf);
```

We calculate the total full ine bytes, then add the extra line bytes to that. We add 9 because of the termination string, and then finally add the `strlen()` of the buffer we snprintf to. As this is the most accurate way to obtain the length of the header string. One part is the logging function which takes in a log FD, a message obj, an offset, and a FD for the file being accessed. This will be done in two parts as well, the first case is if the data of the file is < 1024 (our set buffer length). In this case we do not have to worry about reading in more data from the FD of the message obj. We then loop for every character in the buffer containing the data of either the file or from the client. We convert each char into a hex char using `snprintf()`. After this, we write to the log file FD at the given offset, or incremented offset from earlier writes. This is done thourgh the use of `pwrite()`, the thread safe version of `write()`. If at any point the `index % 20 == 0`, then we know we must write a '\n' followed by an eight bit number representation of the amount of characters written before the line on which the number resides. We do this, incrementing the fileoffset every time until we have reached then end of the buffer in which we are reading from. We then pwrite the terminating string "========\n" using `pwrite()`, and return out of the func.

```
for(idx = 0; idx < readBuf; ++idx){
    if(idx % 20 == 0){
        snpritnf(logging buf, 10, "%08ld ", idx);
        fileOff += pwrite(log FD, logging buf, strlen(loging buf), fileOff);
        //put in first char here
    }
```

```
    else{
        snpritnf(logging buf, 4, "%02x ", msg->logBuf[idx]);
        fileOff += pwrite(log FD, logging buf, strlen(loging buf), fileOff);
    }
    //edge cases for partial lines and last cahracters before a newline
}
pwrite(terminator);
```

The sceond part is nearly identical. Only this contains a conditional loop. We complete the same for loop as the first case, only it is done while we still have data to read from the file. Once we have written everything from the first read, then we call read on the input FD again and complete this same loop again until we read 0 bytes from the file. We then finish by writing the terminator after the while loop, as with the first case. The logging is then complete.

The second logging case is that of a failure. The *GET* function processing handles its logging and repsonse separately than *PUT* and *HEAD* do. For the other two, we make a check after we check that logging is being implemented. If the message code is not of either a success or created, then we will log a failed request. I chose to implement this separately from the logging function as there is less overhead and complexity with implementing errors this way. To calculate the offset, we only need the length of

```
snprintf(&response_buf[0], 100, "FAIL: %s %s %s --- response %d\n", message.method,
message.fname_parse, message.httpversion, message.status_code);
```

This is the entire string we print to the logfile, terminated by the termin str. After this calulation, we can simply pwrite the `response_buf` to the logfile, increment the offset and pwrite the termin, and thus the error logging is complete. The error logging is also done in the same way when checking the original parse as stated above. Instead of logging and responding outside of the `get_process() function`, *GET* requests are processed and logged within its processing function as it was easier for myself to keep track of the data.

After we log, for *PUT* and *HEAD*, we take the status code of the message and the client socket and pass that into the function `send_response()`. This formulates the correct response for us, and then sends it to the client via `dprintf()`. Again *GET* does its own response formatting and sending, all within its processeing fucntion. The thread then continues with its loop in order for it to be able to accept more things from the queue.

Healthchcking was not implemented with this porgram.

Testing

Testing is to be done manually untill it is sufficient to test with the test script. To test manually, we must have a file that contains data first. We then generate a curl request, which will send a request to the server, on a given port. We first must test *PUT*, otherwise all other tests will fail. To test *PUT* we send the request

```
curl -s -T filename hostname:port filename
```

This sends the *PUT* request to the server and the server will respond with the appropriate message. Optionally, we can specify a '-v' flag at the end to print out the response message on the client stdout.

To test *GET*, we must either still have the server open from a previous *PUT* request, or we must send a *PUT* request first. Once the file is on the server, we can send the *GET* request.

```
curl hostname:port/filename
```

With optional '-v' flag for a print out of the server response. The *GET* returns the response, followed by the data, which is written toe the client socket.

Testing for *HEAD* and file permissions will occur manually as well.

In addition to all of the testing above, multiple manual curl requests will be sent using the command line. E.g. (curl ...& curl ... etc.). This will test my programs ability to handle concurrent requests and its ability to properly multithread. I will also have to test runnign my program with different program args. This will test the `getopt()` function, and the logging capabilities of my program.