Design Document: Multi-threaded HTTP Server with Logging

Eric Tang
etang12

1 Goals

The goal of this assignment is to build upon an already functioning single-threaded HTTP server that is able to handle 3 cURL requests with methods of PUT, GET, and HEAD. It should be able to properly send an appropriate response back to the client. The main difference between this version of the HTTP server and the previous one is the implementation of multithreading which allows for concurrency. The server should be able to handle multiple requests from different clients simultaneously through the usage of several threads. These threads each function to handle the different requests from the clients and send the correct response back to each client just like in my previous single-threaded HTTP server, but synchronously. Lastly, this implementation of the server should be able to log the requests of the clients into a log file. The log file should provide information about each request that the server accepts. Logging should also be able to run in a simultaneous fashion. Overall, the main purpose of this assignment is to create a multithreaded server that can concurrently handle client requests as well as log them into file.

2 Design and Implementation

2.0 Design Flow There were many parts to this program that required intensive forethought in order to reach my end goal of having a fully functioning, multithreaded HTTP server. Since, I knew that my server functioned well as a single threaded program, I did not have to worry about dealing with bugs from the previous assignment. My first thought was to handle multithreading first. I knew it would be difficult to implement because of how hard it is to properly debug due to heisenbugs and such. Since the whole program is about concurrency and getting multiple threads to process code seemingly at once, I knew it would be better in the long run to get this down first. After getting multithreading down, I would then focus on the logging portion of the assignment since it relies on concurrency as well. Assuming my multithreading was fine, I would only need to focus on formatting issues. The health check I would save for last as I planned for it to be the simplest part of the assignment assuming I have the other two parts completed. The

plan was that I would split the assignment up into 3 parts and make sure each part worked before I moved ahead.
1. Focus on the multithreading aspect and confirm that server can handle multiple requests at once
2. Move onto the logging of such requests and ensure that the log file is properly formatted while also working with multiple threads
3. Implement the health check after the other two are complete because I assumed this to be the least time consuming part of the assignment

2.0.1 Multithreading While attempting to design how I would implement multithreading, I kept trying to figure out the potential bugs that I would face. The first and worst case I could think of was race conditions. I had to account for different ways in which multiple threads could potentially access each other's memory and give back unexpected results. I had to find some way that would prevent these threads from doing so in order to achieve concurrency. After some deliberating and guidance from the TA's, I decided to utilize a circular queue to be the backbone of my server. The idea is that the queue would store client requests which could then be grabbed by different threads. This would allow for a logical, efficient way to have multiple threads handling different connections since the queue would be a shared resource. Now, I define a dispatcher thread to be the main thread that accepts connections from clients and enqueues them into the queue and worker threads are the ones who dequeue from it so that they each have their own connections.

1. Circular queue → used to store requests from clients that dispatcher thread and worker threads can access
2. Dispatcher thread → Accepts connections from clients and enqueues them into the queue
3. Worker threads → Dequeues client connections from the queue and handle them on their own

However, there is a glaring issue with this implementation. Since the queue is shared by all these threads, there is nothing to prevent race conditions and other issues. Multiple threads could access the queue at the same time and almost definitely produce an issue in the server. Worker threads may dequeue the same client, or they may keep trying to dequeue when the dispatcher thread has not accepted a new connection yet. The dispatcher could try to enqueue, while a worker thread tries to dequeue which would cause problems in memory. The queue could be full and the dispatcher could keep trying to enqueue more connections. The list goes on and on about the potential errors that could arise. Essentially, the queue is a great tool to share client connections between all threads, but these threads are all able to stomp upon each other's memory and give undesirable results.
The solution is to use mutex locks (mutual exclusion). Using locks only allows one thread to have access to a locked section of code, critical section, at a time. This would effectively prevent the dispatcher thread and the worker threads from all accessing the queue at the same time. So, I

implemented locking of my queue functions to ensure that no threads could step on each other and manage to corrupt memory or create race conditions. It would also need to use functions like pthread_cond_wait() to make a thread sleep if there is no work to be done in order to prevent wasting CPU power through busy waiting. Since, waiting would be implemented, that means condition variables and pthread_cond_signal() would also need to be used as well. Condition variables are a type of variable that pthread_cond_wait() and pthread_cond_signal() use to notify each other that a portion of code needs to sleep or needs to be woken up. It lets the two functions communicate with each other to prevent a scenario where a thread will be waiting forever. In my implementation, I will be using 2 condition variables, one exclusively to wake up worker threads and one exclusively to wake up the dispatcher thread. A well thought out combination of these tools will allow for seamless interactions between the dispatcher thread and worker threads to further ensure that mutexes are being used correctly to prevent concurrency-related issues.

2.0.2 Logging Logging is a huge portion of the program that requires multithreading to work in order for it to be functional according to the specification. The basic idea is that for each request from the client, whether it be a valid request or not, information about the request would be written to a log file. This information includes a header that tells what method was used, the file name, and the length of the file. Following that line would an N defined amount of lines where N is dependent on the size of the file. This is because the contents of the file all need to be converted to hex, so each character from the file is converted to hex. On the 20th hex character, then it drops down to a new line and continues converting and writing hex characters. Also, in front of each body line is an 8byte decimal number (00000000, 00000020, etc) that gets incremented by 20 each line to represent how many hex characters were displayed in the previous line. The last line, or the footer, consists of ========\n, characters. That would be the proper formatting of a logged request. So, a header is given that describes the file in general. The following lines after the header are the hex conversions of the contents from the file followed by a footer line to represent the end of a log request. However, there are different kinds of requests that have different kinds of formats. The one that I just explained is used for valid PUT and GET requests. Logging still needs to account for a FAIL request where the request is either invalid due to naming conventions or not having permissions, or the file just not being found (400, 403, 404 error codes). HEAD requests also have a different kind of formatting as well, but in general they all share similarities in their formatting that can be found within the specification. The main idea is that every request needs to be logged within the log file error or not. However, there is a big issue when having to do this concurrently, because each request's log should not overwrite another since they all have to write to the same file. The implementation I chose to follow was using a function called pwrite() which allows for multiple threads to write to a file at the same time without overlapping over each other. This is possible because one of pwrite's parameters is an offset that determines where in the file it will start writing. So, I had to calculate how many bytes it would take to write a hex-converted file into the log file before I called pwrite. By

calculating the offset beforehand, I would then be able to use pwrite safely without worry of threads writing over each other. The offset would need to be a global variable that all the threads can access, since they would be using it to update where they would start writing at and incrementing it for the next thread. A mutex lock must be used here to lock the global offset as it is being incremented such that only one thread can increment the offset at a time to prevent overwriting issues.

2.0.3 Health check The health check is just a part of the assignment that when a client calls GET on it, a response will be sent back with information about the log file. It will show how many entries and errors were logged. PUT and HEAD requests to health check should be deemed as invalid and should send an error response to the client. On top of that, the health check request must also be logged with the contents of it being the amount of errors and entries. The health check request itself does not increment the number of entries or errors until the next request to it. This I decided I would implement with my logging function as it would need to use the same logic of formatting and hex conversions.

2.1 Main Function In addition to having the similar features to my assignment 1 main function where it sets up the server and accepts incoming connections, it now acts as a dispatcher thread. In here, the dispatcher delegates client requests to the worker threads. It accepts connections from the client, then it enqueues them into the shared queue. Not only that, but my main function also handles input from the command line of a client. It determines how many worker threads should be created, the port number, and the log file using the C getopt() function. The getopt() function allows for all types of combinations of inputs to work as long as the important information is all there. So, my main function acts as a parser of the inputs from the command line from the client as well as also putting client requests within the queue for worker threads to grab. It also is where the worker threads are initialized. They are created using pthread_create() and must wait before they can do anything. A mutex is used here as well in order to lock the enqueue function to prevent other threads from accessing the queue as the dispatcher is placing requests into it. Again, this is to prevent race conditions and other memory issues.

```
int main(argc, argv){
    use getopt() to parse number of threads, log file, and port number from command line;
```

create log file if logging detected from command line;
initialize circle queue struct and variables and httpObject struct;
initialize worker threads corresponding to number defined parsed at command line or default to 4 if nothing inputted;
Create a server socket and configure it so that it bounded to a server address;
Listen for incoming connections;
Create a client socket that accepts connections;

pthread_mutex_lock;
Call an enqueue function and use a mutex to lock it when enqueueing,unlock it when done and signal to a worker thread that the queue is no longer empty;
pthread_mutex_unlock;
pthread_cond_signal(worker);

2.2 cb_enqueue This function is responsible for placing a client request into the circular queue data structure. It takes a circular queue object and a client request integer, as inputs. The dispatcher thread uses this to fill the queue. It utilizes two pointers, one to the head of the queue and one to the tail of the queue to keep track of what's being put into the queue and where the pointers are (this is needed to allow for communication with dequeue function). The head should always store the oldest value put into the queue, and the tail should be updated for each new value that comes in. This way the queue can follow a first in - first out approach. Since I am using a circular queue, that means that it has a limited size. In order to prevent memory issues (out of bounds), there is a check for if the queue is full. If the queue is full, that means that the function should not stop trying to put data into it. Instead, the function will make a call to pthread_cond_wait() on a global condition variable and mutex that was passed along with the circular queue object in an infinite while loop. This way, while the queue is full a wait occurs.
Reference for circular queue data structure:
https://www.geeksforgeeks.org/circular-queue-set-1-introduction-array-implementation/
I used this to implement my circular queue and modified it to allow for waiting when the queue is full.

```
void cb_enqueue(circleBuffer *cb, int client){
    while queue is full{
        //signaling to dispatcher to stop enqueueing
        pthread_cond_wait(dispatcher, mutex)
    }
```

```
    if(head == -1){
        head = tail = 0
        queue[tail] = client;
    } else if (tail == max size - 1 && head != 0){
        tail = 0;
        queue[tail] = client;
    } else {
        tail++;
        queue[tail] = client;
    }
        size++;
}
```

2.2 cb_dequeue This function is very similar to enqueue with the main difference being that instead of filling up the queue, it removes a client request from the queue instead. It takes only the circle queue object as an input and it returns an integer that represents the client's request. So, the function dequeues a client request as a return value that can later be used by the worker thread that dequeue was called in. It still utilizes two pointers of a head and tail, to keep track of where values are within the queue. But, the main thing here is that the head will always store the next value to dequeue since our threads would want to do work on the first requests that come in so that the queue will follow a first in - first out approach. However, if the queue is empty that means the function should stop trying to dequeue. This means that the function must wait for the queue to fill up with requests before it can run again.
Reference for circular queue data structure:
https://www.geeksforgeeks.org/circular-queue-set-1-introduction-array-implementation/
I used this to implement my circular queue and modified it to allow for waiting when the queue is empty.

```
int cb_dequeue(circleBuffer *cb){
    int client_fd;
    while(queue is empty){
        //make worker threads wait while queue is empty
        wait(worker, mutex);
    }
    client_fd = queue[head];
    if(head == tail){
```

```
        head = -1;
        tail = -1;
    } else if(head == max size - 1){
        head = 0;
    }
    size--;
    return client_fd;
}
```

2.3 thread_func When initializing worker threads in the main function, it needs to use the function pthread_create(). This function, pthread_create(), has a parameter which takes in another function that the threads should start running when created. thread_func() is the function that I made to fit that parameter. This function serves to push the worker threads to do work which involves dequeuing client requests from the queue using cb_dequeue(), and handling those requests using functions from last assignment, read_http_request() and process_request(). Since, the worker threads are calling cb_dequeue() which means they need to access the circular queue, mutexes and signals are used to allow for concurrency. It also makes the worker threads log if logging is enabled within the server. To boil it down, this function keeps the worker threads running forever within a while loop where they will keep dequeuing client requests and handling them because that is their intended job.

```
void* thread_func(void* arg){
    while(true){
        pthread_mutex_lock(mutex);
        int client_fd = cb_dequeue();
        pthread_mutex_unlock(mutex);
        pthread_cond_signal(dispatcher);
        read_http_request();
        process_request();
        if(logging enabled){
                call logging function;
        }
        clear all statically allocated memory;
    }
}
```

2.4 log_func This function handles all the logging in my server. It takes an integer file descriptor of the log file and an httpObject as inputs. Now, this function will write information about the request to a log file. The thing that makes this tricky, is the fact that it must do this concurrently to the same file but not overwrite as explained earlier. So, I must use pwrite to write to the log file at a specific offset such that all threads will start writing at an offset after the others. In order to do this I need to calculate how much I would offset by for each file and use that in my pwrite() offset parameter. I explained how my logging would work earlier on in section 2.0.2 and this function is where it all happens. The hex conversions, the offset calculations, the pwriting to the log file all happen in here as well as health checks. The main thing to note is that I have 4 cases that I check for: if the request is invalid, if it is a HEAD request, if it is a PUT/GET request, and if it is a GET request to health check. There are just many cases to deal with and each case is handled differently. Each of these are dealt with individually since they all have different formatting and have different types of offset calculations. There is no one way to calculate the offsets for all these cases so each case handles offset calculations differently. The same applies to how I write to the log file. I use snprintf() to write contents of a request into a log buffer. I write the contents from that buffer into the log file. The log buffer could consist of many things depending on the type of request sent by the client. And, the offset I use to continually write into the log file would be a local one defined by the return values of snprintf() which tells me how many bytes I wrote into the log buffer. This local offset would keep incrementing depending on how many bytes are written into the log buffer, and then pwrite would be called using that local offset and write from the log buffer into the log file. For example, PUT and GET requests I handle in this way. First, I would pwrite the header into the log file at the global offset, increment the local offset, then I would move onto the body portion. For the body portion, I use a for loop to continually loop through all the bytes read from the original file, and convert each character into hex. After I have a full line of hex, I would pwrite that line into the log file. So, I would pwrite line by line into the log buffer until all the bytes have been read from the original file. I remember to increment the local offset after each snprintf() I use to fill my log buffer in order to ensure that I'm writing to the correct place in the log file. I then finish off by writing the footer line at the end. I do similar things for FAIL and HEAD requests, where the only difference is that they are normally only 2 lines long so I do not need to delve into the hex conversion. In these cases I simply use snprintf() and write a correctly formatted log request into the log buffer and pwrite that into the log file. I increment the global offset by the return value of snprintf and move onto the next request. Health checks are handled similarly to GET/PUT requests.