HTTP Server Design Document

1 Overview

This program is a multithreaded web server that logs all request/response as well as performs health checks on files. It will have a dispatcher thread that accepts requests incoming from a socket. The dispatcher will then put all requests headers with unique socket descriptors into a queue for a number of worker threads to process. The worker threads will look into the queue, pop the request data and process the request per the asgn1 spec. Threads will send the response then log it in a log file on the server. The order logged will be in the order the response was sent back. The ordering of which threads handles a dispatched request and how it logs with other concurrent active requests will be one major synchronization problem that will be handled with locks using mutexes and conditional variables.

My design is broken up into explanations of how I handled each of the challenges that came with making my http server multithreaded including: argument parsing, dispatching, logging, and a healthcheck. After explaining my design I will briefly mention all new functions used with their input and functionality. Then I will cover restrictions, limits, and constraints and then finish with a note about testing methodology.

2 Argument Parsing

The server is allowed to take in two optional flags for logging (-l) and number of threads (-N) each with arguments following the flag which are required. The logging flag specifies whether the server should log and the argument tells us what file to log. The number of threads flags and arguments specify a set number of threads. The server also takes in a required port number argument like in asgn1 .

To deal with argument parsing I will use getopt(3) to parse the command line arguments using the switch statement with one change. The server will use a - as the first character in the optstring of getopt so that the server port number will be caught under case 1 and set a port flag as well as port number. If it is caught again and the port flag is 1 we will know there are extra garbage arguments and exit. The -N and -l flags are in the switch statement and they set a flag and thread count and log file name respectively using optarg included in getopt.h.

3 Dispatching

To dispatch my threads I used a queue that holds the log file descriptor and client socket, a global mutex lock, and a global conditional variable. The program only uses one lock and conditional variable because only one thread is allowed to access the queue at a time. Therefore the access thread must have the mutex to pop information from the queue. Before the server starts waiting for requests the program first creates the input number of threads if included in command line arguments using create(3) with our and **handleRequests** function as the entry point for each thread with our request queue as input. When each thread processes through that function it will get the lock and then enter a while(true) loop from the queue which will first check to see if the queue is non empty and if so pop a request (which will also give up the lock inside the pop code so that other threads can access the queue while that thread processes) and handle it by calling the **handleRequest** function using a similar process as in asng1 with the added functionality of healthchecks and logging. Once the thread finishes handling the request it reloops and processes another request if the queue is non empty. If the queue is empty then the thread “sleeps” using the pthread_cond_wait(3) function with our conditional variable and mutex as input to first signal the thread to sleep and to second give up the lock. This method guarantees that no thread can access the queue at the same time however we are still missing one piece.

In the while loop in the main function from asgn1 the program checks for incoming requests and if one is caught it accepts it using accept(2) and pushes the file descriptor to the queue. Now when a new request gets pushed to the queue the queue will lock the code to add the request unlock it to be extra careful with our main or dispatcher thread and then signal the conditional variable to wake up a thread using pthread_cond_signal. When the thread is woken up it will be at the end of the while loop in **handleRequests** loop over again checking the queue for a request and processing it. This design guarantees that no thread can access the queue at the same time as well as allows each thread to process independent of other threads.

4 Logging

One of the major additions to the multithreaded server is logging. If enabled (via -l logname command line argument) Each request will be logged with their contents if applicable to a log file specified by the command line argument. Each request writes a header line with data about the request is it is a successful put or get it will write the contents of the file up to 20 bytes at a time in hex prepended by a decimal index string. Each logged request will end with 9 = and newline.

In the multithreaded server after the thread handles and sends a response it will go into the **writeLog** function with the message object (from asgn1) and log file descriptor opened in the main and pushed to the queue along with the client socket descriptor. One additional note, I added 3 additional variables to the message object; char bad_method[10], char bad_filename[45], and char bad_httpversion[15]. Each is assigned a value with its associated counterpart when parsing request header lines in case the server receives a request with a bad method, filename, or httpversion. They are stored to later be used when logging the erroneous request. It is important to note that we are assuming that any bad requests with a method > 10, a filename >45 and a httpversion> 15 will NOT be logged correctly.

On to the design, first let’s cover how to deal with synchronizing the threads to write to the log file without interleaving. To do this I used an atomic variable offset to keep track of where each thread should write to the file in conjunction with pwrite(2). Pwrite writes to a certain amount of bytes file at a given offset. During each thread’s execution of the **writeLog** the thread will calculate its expected which is the length of the log header line + size 69* number of full lines + the size of the remainder (content length % 20) + the length of the terminating line. Once the expected size is calculated a thread can call atomic_fetch_add which returns the old value of the offset and increases the atomic 32 bit integer by the log size. By using a shared atomic_uint_fast32_t variable for the offset we can certify that no thread will interleave and we have enough space to log very large and multiple very large files! However the speed hit to atomicly updating a memory block of that size might not be worth the ease of implementation. The write to log function will write to the log depending on three cases: if the request was a successful GET or PUT, if the request was a successful HEAD, or if the request was not successful.

If the request was an OK get or put we will use calculate the offset add it and retain our write position for the log file as talked about above. First we start an index count which we convert to a string using sprint(3) which we will copy into a 8byte char buffer We then read from the file name specified from our message object into a buffer of size 16084. 300 bytes below 16KiB so as not to have one thread allocate more than 16KiB when we use intermediate buffers for things like the log header line and such putting in our http version, file name, response code, and method. With this done we will then calculate the number of 20 byte lines to convert to text and run a function **writeLog** which starts while loop which loops as long as the total lines to read is < 0. It first reads into the buffer Inside is a loop which iterates 20 bytes at a time through the buffer up until it hits the buffers maximum full lines (buffer size /20) after the for loop we decrement the total lines by the maximum the buffer can read. In the for loop it converts each 20 bytes into hex in a char buffer[20] using snprintf with 20 “%08x” and at i, i+19. After each iteration we increment the threads offset by 69, and the for loop iterator by 20, and the number of lines by one. By the end if our remainder lines passed in to **writeLog** is > than zero then we do what the for loop does for full lines in one line and loops the snprintf the remainder bytes into our buffer. With this done we write our terminating string and add to the offset. Finally we have logged the complete file!

5 Healthcheck

When the server receives a GET request to the file healthcheck it is supposed to respond with the number of log failures in hex, a new line, total logs in hex, and then another newline. If the user sends a PUT or HEAD request for the healthcheck file the server responds with a 403 and if the user did not specify a log file with the -l flag the server returns a 404 response.

My implementation of the healthcheck is fairly straightforward. First I will create two atomic variables to track successes and failures after each log to the file. When the thread is in its **handleRequest** function if the filename is healthcheck and the request method is GET the server will respond with a response as specified in the previous paragraph. Then it will log the response using the same technique as the **writeLogBody** function with the header line, number of failures, total logs, and terminating line. It is important to note that this response type will be included as a success.

6 Components

HandleRequests function

Discussed in section three it gives threads a lock then puts them in a loop where the check to see if a request is on the queue and if it is they pop it which also gives up the lock and handle it by calling the **handleRequest** function with the file descriptor and log descriptor as input. Once it finishes it reloops and checks the queue again if there are no requests it then gets put to sleep. Keeping it verbose because it was already explained.

HandleRequest function

Takes in a _ and _ as input. This function then parses our http request as was done in asgn1. After parsing if the filename is a health check, the log fd is valid, the request method is a GET, and the response code is either 404 (incase it parsed the request as not found which we can ignore) or 400 we send the log file descriptor and message object to the **writeHealthCheck** function. If not we create the response the same as in asgn1 and after we check if the log file descriptor is valid and if so we send the http object along with the log file descriptor to the **handleLog** function.

writeHealthCheck function

This function writes a specific response back as specified in the spec and use our atomic failure and success counts as described in section 3 to do so. After writing the response this function logs the response to the log file using the same technique as **writeLog** minus the full line logging. Important to note this function assumes that the number of combined total and failures will not exceed 19 bytes (20 -1 for the space newline in between). After completion this function has successfully sent the specific healthcheck response and logged it as a success to the file.

handleLog

This function takes in a message object and log file descriptor as input and logs the response success or failure to the log file. Threads in this function open the log for writing then check the status of the message object and handle each status differently. Each starts the same however. For each case we create the log header line using sprintf(3),

If the message is a 200 or 201 Get or Put object then this function will we create the log header line using sprintf(3), calculate number of 20 byte strings to read from the file specified by the message objects filename (content-length/20), calculate the number of remainder bytes (content-length%20), and the length of the log body using the calculation from section 4. Then call **writeLogBody** with the log descriptor, file descriptor, line count, thread offset, and binary index count as input.

If the message is an OK Head request we create the log header line using sprintf(3), then append the terminating string, get the size, add that to the atomic offset using __atomic_fetch_add(3) and pwrite to the log file using the return value of __atomic_fetch_add(3) . Finally add to the atomic success variable.

If the message is none of these then it is a failed request. For this we do the exact same thing as the OK HEAD request head, only our header line is formatted differently and we add ot eh atomic failures variable as mentioned in section5

writeLogBody

Takes a threads log descriptor, file descriptor, line count, thread offset, and binary index count as input. It then logs the successful get or head request to the

requestQueue Object

The request queue object is set up in separate files, requestQueue.h and requestQueue.c. It includes a push, pop, getFD, getLD, and getCount. The queue nodes contain a log descriptor as well as a file descriptor with methods to retrieve them. Push and pop function as any queue with some important changes. The pop function unlocks the mutex so other threads can access the queue after it gets a node and the push locks and unlocks the global mutex when adding the new node just.

5 Restrictions, limitations, and constraints

5.1 No thread can use more than 16KiB of buffer space

6 Testing

6.1 Test Classes

I will first be incrementally testing my program during the writing process. This includes testing the makefile. For this assignment I will test using a small number of curl commands at first and then move to testing with a lot of curl commands using shell scripts.

After testing incrementally with a program that works for inputs I have tried I will be using the classes test cases on gitlab and test errors seen from there. More details in my Writeup.