Aidan Smith

CSE 130

Design for Assignment 2

1.0 Introduction

This assignment is based on creating a multi-threaded httpserver. This will allow us to serve clients faster as our throughput is increased by having more simultaneous threads. Even though we are serving more clients, our latency should be unaffected as we are not sacrificing how long it takes to do each task, but rather doing more tasks at a time by utilizing more of our computer's processing power. This assignment will be building on the previous assignment, and thus will not care to explain the processes involved in assignment 1.

1.1 Goals and objectives

We will be taking assignment 1 which was a single threaded httpserver and will be turning into a multithreaded httpserver for assignment 2. Additionally we will be adding a logging feature, which will have all requests from clients put into a log file, fail or success. This log file can be read from at any time and will be formatted as specified below. Finally, we will be adding a health checking feature. This will tell us the number of fails and the number of total requests made to the server.

1.2 Statement of scope

Httpserver is an executable that runs a server. It can be run with different options, this is expanded on in the README. Clients will be able to connect to this server through curl commands and the port of the server. The client may GET, PUT, or HEAD for different files. The client may also call GET on healthcheck, which isn't a file but a protocol. It works as mentioned above, and is also saved to the log.

2.0 New operations

This assignment saw the introduction of many new commands in order to construct the httpserver. Their use in the program will be expanded on in the functions section.

2.1 getopt()

Getopt is used to read the arguments from the command line in a clean and unordered fashion. This way the server initializer may order the port number, number of threads, and log file in any order. They may also not give the number of threads or a log file and it will still work. It searches the argument list for the flags and then can be used

to find the argument after the flag. This makes it easy to attach the expected value to the flag.

2.2 pthread_create()

This command is used to create additional threads. Normally a program would run on just one, the normal program execution itself. By creating more threads, we can do multiple things at a time, and thus we are multithreading. These threads must share CPU and in our case will be sharing variables which can cause problems.

2.3 pthread_mutex_init() & pthread_cond_init()

These commands are used to initialize the mutexes and conditionals used in this program.

2.4 pthread_mutex_lock() & pthread_mutex_unlock()

These commands are used to lock and unlock a chunk of code, so that only one thread may work on it at a time. This is highly important for multithreading, as there are some operations that cannot be atomic, and thus need protection. If they are not protected and are shared between threads, their values could be distorted and thus cause harm to the server.

2.4 pthread_cond_wait() & pthread_cond_signal()

These commands are used to tell a thread to unlock and wait for a conditional until it can lock again, and to tell a thread that it has the conditional and may now lock and complete its task. This is important because without these, a thread would hog the lock but not be able to do anything as it may not be ready to read the information yet. Thus using these commands it can unlock and wait until the proper information has been set for it to calculate.

2.5 pwrite()

Pwrite is used similarly to write, but the difference is that pwrite can write using an offset.

3.0 Structs used

I used two structs in order to send a group of information between functions so that each function could read from and write to it as needed.

3.1 struct httpObject message

This struct was used in the previous lab in order to complete single threaded server tasks. It has been left unchanged, except for an extension of sizes for the given

filename, method, and http version. These will allow the printing of incorrect size requests to the log file.

3.2 struct threadStruct threadStruct

This struct was created for this assignment, in order to pass information to our different multithreading elements. It contains the client queue, and its start index and its end index, this allows the queue to be handled as a circular queue and thus save space. The queue and its indices are protected by a mutex so that single requests aren't completed by multiple threads. The struct also contains the number of fails and requests made, this aids in health checking. It also has the global offset, which is the current offset to be used by a thread so that it doesn't tread on another thread's allocated space on the log file. This is protected by a mutex and increased every time a thread is logging. The struct also has the file descriptor of the log file if it has been supplied, otherwise it is 0. Finally the struct contains the two conditionals and two mutexes used in this program. The two conditionals and mutexes apply to the two groups of previously mentioned objects on the threadStruct.

4.0 Design

This design still utilizes only one file, built on from the previous lab, and a makefile.

4.1 httpserver.c

Httpsever.c is made up of 6 functions, 3 edited ones from the previous lab, and 3 new ones to handle the requirements of assignment 2. It also uses the int main() to read arguments from the server initialization to create the server, and then also contain the server dispatcher.

4.2 Makefile

Makefile is used to run httpserver.c cleanly, its usage is described in the file.

5.0 Functions

Httpserver for assignment 2 is built off of assignment 1 and thus utilizes the previous assignment's functions. These functions will not be mentioned as they are covered in the assignment 1 design document, but their changes will be mentioned. For the new version of httpserver, the design is now made up of a changed int main() to work as the server initializer and the dispatcher thread. It also now has a worker thread function, which is called as many times as requested by the user. A function for logging has also been created. This function does all the necessary work to get the server to write all previous requests into the log if it has been requested by the server initializer. The last new function is a substitute function to the

construct_http_response from the assignment 1. This function is solely used for writing the correct response to the user when they call a health check.

5.1 int main(int argc, char* argv)

This function begins by reading in the arguments from the httpserver initializer. It sorts them out using getopt() and saves them in the threadStruct. Inside of getopt() the server initializes the log file if it was given. It then initializes the mutexes and conditionals, the threads, and the threadStruct. It then creates the server as done before in assignment 1. After this it now hits an infinite while loop, used as the dispatcher thread. It will wait for a client to connect, and then it will add it to the queue, locking the queue when writing to it, and then signaling a worker thread that it has done so after it finishes writing and unlocks. It then loops back around and waits for another client to connect.

5.2 void* threadFunction(void* threadStruct)

This function is the thread function. It will be created by int main() and will initialize the httpObject used for each individual thread. It will then wait until a signal is received from the dispatcher that there is a client on the queue. It will then dequeue and call the read_http_request function. After it returns, it will check if the log file was initialized, and if so then it will call write_to_log.

5.3 void read_http_request(ssize_t client_sockd, struct httpObject* message, struct threadStruct* threadStruct) {

This function is from assignment 1 and has a few changes. Health check functionality has been added, in order to avoid going down the normal path of operation as if healthcheck was a file. It makes sure it was requested as per the spec, and then calls construct_health_check. It then returns, not going to other assignment 1 functions.

5.4 void process_request(ssize_t client_sockd, struct httpObject* message) {

This function is unchanged from the previous assignment.

5.5 void construct_http_response(ssize_t client_sockd, struct httpObject* message) {

This function is unchanged from the previous assignment.

5.6 void construct_health_check(ssize_t client_sockd, struct httpObject* message, struct threadStruct* threadStruct){

This function is created to act like construct_http_response, but is only called when a health check is requested. This extra function is created so that threadStruct doesn't have to be passed to the two functions above.

5.7 void write_to_log(struct httpObject* message, struct threadStruct* threadStruct){

The final function, write_to_log, is used to write the requests to the log, fail or not. It begins by writing the header to a buffer. It then calculates the length of this buffer, the expected length of the body, and the length of the footer. It then locks the global offset variable, first saving it as the local offset, and then adding the previously mentioned sum to the global variable. The new value of the global variable will now be the starting place of the next thread. The local offset will be used by the thread to write to the correct place in the log file, under its allocated space.

It then goes on to creating the body of the request and puts it into the buffer. The body requires tracking of the byte number, and converting each byte to its hex equivalent. For a PUT or a GET request, it will look at the main file supplied. If it's a HEAD or a failed request then nothing is written for the body. If it is a health check, then the contents of the health check are put into the log. If the buffer gets close to being full, then it will pwrite to the log file, using a while loop to make sure it writes everything. Whether it pwrites here or not, it keeps on going and takes the body onto its buffer. Once the body is complete, it has to add the header to the buffer. It then pwrites as there is no more information to put onto the buffer. At the end it adds to the counter for the number of requests, and the number of fails if it failed. It ends with a return.

6.0 User interface

This is described in README.md.