DESIGN DOC for httpserver.c (asgn2 CSE130)

Author: Adarsh Sekar

1. Introduction:

1.1 Goals and Objectives:

The goal of this assignment is to build on the httpserver.c program we made in asgn1 by adding 3 things: multithreading, a log and a health checker. The idea of multithreading will allow us to run our program (or operation, more specifically) on multiple threads, so that we can perform multiple tasks at once, with each task running on its own thread. The log is meant to keep track of all the operations we perform, and the status of them, i.e, whether they succeeded or not and why(status code), and if they succeeded, the data transmitted (for PUT and GET). The health check is a procedure that when called, should return the number of errors in log.

Another change in the implementation is that I will be creating the files httpfunc.c and httpfunc.h to write all the functions that I will be making. This is just to increase modularity, and overall neatness of the program. To run the program, you will need:

- httpserver.c
- httpfunc.c
- httpfunc.h

1.2 Statement of scope:

The program will be called with a port number and optional arguments of the number of threads and a log file. The syntax will be as follows:

./httpserver <port number> -N <number of threads> -l <log file>

The -N and -l are optional, and if emitted, the number of threads will be set to 4, and/or the logging being emitted. The order of the arguments can also be changed, however, <number of threads> must always follow -N, and <log file> must always follow -l. For this assignment, I will be writing a .c file httpfunc.c and a header file httpfunc.h, where I will create all the different functions I will use, simply to keep things neater. The three main ideas that I will need to implement are multithreading, logging, and the health checker.

1.2.1 Multithreading:

To implement multithreading, I will use the pthread.h library of functions. My main() function will act as the dispatcher thread. Inside main(), I will call pthread_create() to create the number of threads specified. Each of these threads will be created with the function workers(). In workers, we will take care of requests that are sent to the threads from main().

1.2.2 Logging:

To deal with logging, I will create a helper function that will print to the logfile for me, called logger() (explained later). First, I will need to print out the request to the logfile, followed by the filename and then the length, ending with a newline in the following format:

<request> /<filename> length <content-length>\n

Then, logger() will need to read in data from fd 20 bytes at a time, and print them on newlines. Finally, after we've printed all the data to our log, we end with:

========\n
to indicate the end of the log entry. We do not have to display the content on a HEAD request, only on a PUT or GET.
If there is an error, we will print out the following instead:
FAIL: <request> /<filename> <http version> --- response <status code>

Another thing we must be wary of while logging is keeping track of a logfile offset, that will let the threads know where to start writing into the logfile from. Therefore, I will need to create a variable called log_offset, that all the worker threads have access to. The threads will need to have access to this offset, and also be able to change it, but in the right order. So, I will create another mutex that all the threads have access to. The threads will lock the mutex when they are trying to access the log_offset, and will store it's value in a local variable, and increment the log_offset by the number of bytes they expect to write. Then, they will unlock the mutex again.

1.2.3 Health Check:

The health check is going to be fairly simple compared to the other 2. The first thing we will do is check in our parse_http_request() function whether the filename is 'healthcheck'. We also check if logging is enabled. If it is, then we check if it is also a GET, and if it is, we exit out of our parse_http_request() function. If either logging isn't enabled or the request isn't a GET we set the appropriate error code (400) and return.

Now, in construct_http_response, if the status_code is 200, we will check for healthcheck as the filename. If it is the filename, we will print out the health check, i.e, number of errors and number of total requests.

To actually get the number of requests and errors, each of the threads will have a pointer to num_requests and num_errors. These will hold the number of requests and errors respectively. When we get the offset, we will also store local versions of these, and increment them appropriately, i.e, increment num_requests always, and num_errors only when we must.

2. Data Design:

In this section, we will go over the design of functions that I created, and how to use them. Some functions may have already been explained in the design doc for assignment 1, and I will not go over them again. The below functions will also use many library functions, which will be briefly explained in section 3. Our functions will be split into 4 categories: Data types, Helper Functions, Main Functions and main(). All Data Types are defined in the header file httpfunc.h. Helper Functions and Main Functions will be implemented in httpfunc.c, whereas main will be implemented in httpserver.c. This is to make the code neater, and easier to understand by creating more modularity.

2.1 Data Types:

Here, I will go over structs that I created.

```
struct workerThread {
	httpObject message;
	int cfd;
	int logfd;
	int id;
	uint64_t* num_requests;
	uint64_t* num_errors;
	int* free;
	off_t* offset;
	pthread_t worker_id;
	pthread_cond_t cond_var;
	pthread_cond_t* dissignal;
	pthread_mutex_t* lock;
	pthread_mutex_t* dlock;
	pthread_mutex_t* loglock;
}
```

The above struct, I created with the help of TA Michael C. The idea behind this struct is that I want one place where I can store all my data for a given thread. The int's cfd and logfd are to store the file descriptors of the client and logfile (logfile is optional). The httpObject is so that after parsing a request, we can store the information there. The pthread_t is to store the id of our thread. Our cond_var will be used to wake up and put our thread to sleep. The dissignal will be a pointer to a pthread_cond_t variable that the dispatcher will have access to. This variable can be changed by the worker threads, to signal the dispatcher that they are free to accept a new http request. The variable int* free will be used to let the dispatcher know the thread is free. The various mutex locks are for preventing threads from interfering with each other's data.
lock: Used to make sure only one thread is waiting for an instruction at a time.
dlock: A lock used in main(). Only used for pthread_cond_wait() to prevent problems with it.
loglock: Used when getting the offset and updating and receiving the num_requests and num_errors.
The uint64_t pointers num_request and num_error are going to be used to maintain the health check values. num_error will be used to keep track of errors, and num_request will keep track of number of requests.

```
struct httpObject{
	char method[10];
	char filename[BUFFER_SIZE];
	char httpversion[9];
	ssize_t content_length;
	int status_code;
	 uint8_t buffer[BUFFER_SIZE];

}
```

This struct is the same as from asgn1, our previous single threaded httpserver.c, but with a bigger buffer size for variables like method, and filename, in cases where we need to log incorrect methods or filenames, that are bigger than normal.

2.2 Helper Functions:

void logger(int logfd, httpObject* message, ssize_t bytes_read, off_t* offset, ssize_t* bytes_written):

This function will take the file descriptor of the log file, int logfd, an httpObject* message, the number of bytes read (bytes_read), a pointer to an offset, and a pointer to the number of bytes we've written so far. The function is called in print_to_client(), inside a while loop.

This logger is used only to print the data from a PUT or GET. The data read in from the PUT or GET file is stored in message->buffer. We read this buffer and format the data into hex characters, separated by spaces, and with only 20 characters per line. Each line starts with the number of characters uptil that line. Before we do this however, we check if bytes_written is divisible by 20. If it isn't, then we still have a line to fill in, so we fill up the rest of the line, and then continue onto the newlines. To format the lines, I use sprintf() to format the hex characters into a buffer I create. I continuously overwrite the buffer after printing its contents to the log file every 20 bytes of data. I also increment the offset pointer as I go, so that the offset always points to where we want to print, and doesn't overwrite data. To print to the log file, I use pwrite().

void increment_offset(int log_offset, httpObject* message):

This function will be used to increment the offset for the log file. It takes in the current offset, int log_offset, and an httpObject* message. In this function, we first check the message->status_code. If it is 200 or 201, we enter one of 2 if statements. If it is HEAD, we enter the if statement for HEAD. Here, we add to the offset 4 bytes for HEAD, followed by the a byte for space and a byte for a slash, then the number of bytes the file name is, and the bytes for another space, then the 6 bytes for 'length' with finally the bytes required by message->content_length, plus a newline. If it is a PUT or GET, we do the same thing, with the number of bytes 4 to 3 for PUT and GET, but we also message->content_length, plus an extra (content_length/20) * 2 bytes because of the starting byte count and newline, plus an extra 2 bytes if content_length % 20 != 0.

If message->status_code is not equal to 200 or 201, then if the call is a HEAD, we add to the offset the number of bytes needed to display "FAIL: HEAD /<filename> <http version> --- response <status code>\n". For PUT and GET, it is similar, but replace "HEAD" with the respective call.

For all calls, we add another 9 bytes for the ending "========\n".

void print_to_client(int fd, int cfd, struct httpObject* message, off_t offset, int lfd):

This function is mostly the same as the function with the same name form asgn1, the single threaded httpserver. The main difference is that logging of file data is taken care of in this function. Therefore, this function is also called for successful PUTS. We call logger now in this function, but only print data to the client if the request is a GET. This allows the function to work as a logger as well as a way to print to the client with just a few extra lines of code, that would make the function more abstract, and allows me to not have to make a separate function to take care of logging data. logger() is called with the same message->buffer that would be printed to the client. Therefore for GET's, we only have to read the data once, instead of doing it twice.

2.3 Main Functions:

I will now go over the design of the main functions we will use. These main functions may use helper functions that were explained in the previous sub-section. These functions are the main functions that will take care of the three main ideas I went over in sub-section 1.2.

void* worker(void* data):

This function will be where the worker thread parses the request coming in from the client. The first thing we will do is store the data into a local variable of type workerThread*, this_thread. Then, we will enter a while(true) loop, inside of which we will enter a while loop, where we make the thread wait on the condition variable this_thread->cond_var, using pthread_cond_wait(), to be signalled to wake up and read data from a client. Before entering the second while loop, we will lock our mutex this_thread->lock, that we created in the dispatcher.

When we are signalled, our thread wakes up, and we should have the client file descriptor in this_thread->cfd. Now, we unlock our mutex this_thread->lock so that a different thread can now wait for an instruction. Then we proceed to parse the http request, like we did in asgn1. We call read_http_response(), process_request() and construct_http_response sequentially.

In between process_request() and construct_http_response(), I lock the condition mutex this_thread->loglock. Then I retrieve this_thread->offset, this_thread->num_errors and this_thread->num_requests, and store them in local variables. Before exiting the lock, I call my function increment_offset(), to increment the offset by the bytes we expect to write, and also increment this_thread->num_requests. I increment this_thread->num_errors if the request status code is greater than 201. Then I unlock this_thread->loglock. Then, I call construct_http_response with our offset, this_thread->logfd, and local variables num_errors and num_requests.

Once we are done, we will reset this_thread->cfd to -1. We will then send a signal to the dispatcher letting it know that the thread is free (This will be done by changing the dispatcher's condition variable).

void process_request(int client_fd, struct httpObject* message, int lfd):

This function is a modified version of the same function in asgn1, the single threaded httpserver. It is called inside of worker(). It has been modified to take in a log file descriptor int lfd. The main change to this function is we now have an if statement that before we check what the method is, that checks if the filename is 'healthcheck'. If it is, we check if the method is not GET. If some other method is specified other than get, we set message->status_code to 403 and return. If the message is GET, we check if lfd is greater than 0, i.e, if logging is enabled. If it is, then we return, and will continue onto construct_http_response(). Otherwise, if logging was not enabled, we will set message->status_code to 404 and return.

void construct_http_response(int cfd, struct httpObject* message, off_t offset, int lfd, uint64_t num_errors, uint64_t num_requests):

This function is a modified version of the same function in asgn1, the single threaded httpserver. It is called inside of worker(). The function has been modified to take in an off_t offset for the log offset, an int lfd, for the log file if it is present, and two uint64_t's num_errors and num_requests for health checks.

The first change you will notice is an if statement to check if method->filename is healthcheck. If this is the case, we have a log file present, and we print out the number of errors (num_errors) followed by a newline and the number of requests (num_requests). We also log this request into our logfile using the format described in 1.2.2 in Logging. The format will be the same as a successful GET request, except there is no data following.

The next change is that there is an if statement for all the different message->status_code cases. This if statement checks for a log file. If there is a log file, we log the request, otherwise we don't.

The third change is that if the status_code is 200, we share an if statement for a PUT and GET instead of a PUT and HEAD. This is because we log file data in our print_to_client() function, so that we only have to open the file once for a GET, and to make sure that we only log the data if the PUT was done with no errors. Again, before we print to the logger in print_to_client(), we print <request> /<filename> length <content-length>\n to the logger inside of this function, before the content of the file inside print_to_client().

The final change is if the message->status_code is 201, we call print_to_client() if logging is enabled, as logging of data is taken care of in print_to_client.

2.4 int main(int argc, char** argv):

Our main function is where we will accept client connections, and then send the client socket descriptors to threads so that they can process the requests. Essentially, it will be the dispatcher.

The first thing that will happen in main, is that we will check if argc is less than 2 or more than 6. If either of these is the case, we know our program was called with too few or too many arguments, so we error out.

If we have the right number of arguments, then we will call getopt() with the optstring "-:N:l:", so we know we accept the -N option (number of threads) , the -l option (logfile), and any other arg (for port number). We will set a checker to true if we got the port number so that we don't read in more than one extra parameter. If any other options are provided, we error out. Number of threads will be stored in a variable numthreads, the port number in port, and the log file in logfile. I do some checks to make sure that the data provided is valid, i.e, port greater than 1024, valid logfile and number of threads. If everything is good, we move on to the next step.

Once we get all this, we proceed to initialize our server socket. We create a pthread_mutex_t variable dlock, loglock and lock. 'Dlock' is the mutex lock for the dispatcher (main()), loglock will be used by our worker threads when getting logging information (such as offset), and lock will be a lock we use for our worker threads when receiving client connections. We also create a pthread_cond_t free, that all our worker threads have a pointer to, and an off_t offset initialised to 0 that all threads will have a pointer to. This variable will be used to signal main() that a thread is free to accept a client. We also have an int free_thread that the worker threads have a pointer to, initialized to -1. Once that's done, we create a workerThread array of size numthreads, and initialize each of the threads with our workers variables initialized to the above variables we mentioned.

Then, we enter a while true loop. In this loop, we first lock our mutex, dlock and check if there was an error locking it. If there was, we return(EXIT_FAILURE). Otherwise, we enter a while loop that loops while our variable free_thread is -1. Inside the loop, we call pthread_cond_wait(), putting our main() to sleep. When the free_thread is greater than -1, it means that one of our threads is free, so we wake up our main, and we set our respective threads cfd to the client_sockd. Then we signal our respective worker thread to wake up using pthread_cond_signal(). After this, we unlock our mutex (checking if there was an error unlocking), and loop again through the loop.

<u>**3. Libraries and Library functions:**</u>

Here, I will go over functions that were not part of the starter code. I will only mention new functions I am using, that weren't already explained in the DESIGN doc of asgn1. For all functions, please check the Linux man pages for better descriptions.

<u>Libraries:</u>

pthread.h: This is the main new library I use. It is needed for all pthread functions and variables.

<u>Functions:</u>

pthread_create(): This function is used to create a new operating thread. It is called with a pointer to a pthread variable, a pointer to a function the thread will run, and a pointer to any data that the function needs. In my case, I call it as such:
pthread_creat(&mythread, NULL, worker, &worker_data).
The function and data must be of type void*.

pthread_cond_wait(): I use this function to put a thread to sleep when it is waiting for a client socket descriptor. It is called with a pointer to a condition variable, and a locked mutex. I use this function in both my worker() and main() functions. If there is an error, it will return -1 and errno will be set appropriately.

pthread_cond_signal(): This function is used to signal a sleeping thread to wake up. It takes in only a pointer condition variable. I use this function in both my worker() and main() functions. If there is an error, it will return -1 and errno will be set appropriately.

pthread_mutex_lock(): This function takes in a pointer to a mutex, and locks the mutex. Once the mutex is locked, any other thread that tries to lock the mutex must first wait for the first thread to unlock the mutex. This allows mutual exclusion, preventing threads from accessing the same variables, and ensuring that only one thread is in its critical region at a time. If there is an error, it will return -1 and errno will be set appropriately.

pthread_mutex_unlock(): This function takes in a pointer to a mutex and unlocks the mutex, allowing the next thread in the queue to lock the mutex and enter it's critical region. If there is an error, it will return -1 and errno will be set appropriately.

getopt(): I use this function to parse in the command line arguments.

sprintf(): I use this function to read in characters from a string, and store the read characters into another string. This function is used in my logger() as well as construct_http_response. In the logger, I use it to format bytes into hex and print to the log file, whereas in construct_http_response, I use it to format strings to print to the log file.

pwrite(): This function is used to write to a file. It essentially works the same as write(), that we used in asgn1, but also takes in an offset, that tells it the offset number of bytes from the start of the file to print from. We use this function for logging, again in both logger() and construct_http_response.