Design for httpserver.c

httpserver.c is a server capable of handling 3 HTTP requests; GET, HEAD, and PUT.

This program builds on my previous single threaded HTTP server. This design doc will reference specific actions specified in the design of the single threaded HTTP server without elaborating. If you would like to see the inner workings of parsing the request, parsing the content, or sending the response, click this link.

This code differs from the previous HTTP server in 3 ways: Multithreading, Logging, and Healthcheck.

Multithreading:

In my approach, there is a main dispatch thread and N worker threads.

The dispatcher thread:

The dispatch thread only has two jobs. First, it uses `pthread_create` to create N threads. Next, the dispatch thread listens on the socket and stores new connections in a queue named `work_queue`, which is implemented as a linked list. I decided to go with the linked list implementation because it is what made the most sense to me.

`work_queue` has two functions, `enqueue()` and `dequeue()`. Calling these functions is protected by a mutex lock, called `q_lock`, to keep their operations thread safe.

`enqueue()` has one more synchronization function. It calls `pthread_cond_signal()` on a condition variable named `q_cond_var`. This is to let the waiting threads know that there is work to be done in the `work_queue`, to avoid them simply busy waiting.

When creating the worker threads, the dispatch thread passes some required variables in a struct to the thread, “global variables” that are shared between the threads. The creation is as follows.

```
typedef struct {
                int total_requests;
                int total_errors;
                char* logfile; //name of logfile
                int logging;   // whether logging is active
                int num_thread; //total number of worker threads

} Global_vars;
```

Process for worker thread:
Thread is created and passed in args of type `Global_vars`.

The first thing a thread will do when created is check the `work_queue` for any connections that need to be handled. To do this, the thread acquires `q_lock`. If there is no work on the queue, the thread waits on `q_cond_var` until it is signaled by `enqueue()` that there is work to be done. If there is a connection, the thread calls `dequeue()`, receives the client socket, and releases `q_lock`.

After this step, it grabs the mutex for `total_requests` and increments it by 1. It then releases the lock.

The thread now parses the request. It parses the request type, filename, and content-length from the HTTP request headers.

Now the thread checks for errors: invalid filename, invalid request type, invlaid permissions. If the thread finds any of these errors, it grabs the lock for `total_errors`, increments it by one, and then releases the lock.

If there were no errors, the thread begins parsing the content. If `logging == 1` (aka active), the thread logs the content dynamically while it is parsing the content. See the section on logging later in this doc for more information on how that is done.

After parsing the content, the thread sends the appropriate response.

Logging:

I implemented my logging using a dedicated logging thread. All worker threads, after completing their request, enqueue their file, request, and status code to a logging queue (implemented the exact same way as the previous queue), which the logging thread reads from.

When the logging thread reads from the logging queue, it first calculates the size of the log file to provide the offset for future `pwrite()` and it checks to see if what it dequeued was an error (status code other than 200 or 201). If it was an error, the logging thread calls `log_error()` which correctly logs the error to the log file.

If it was not an error, the logging thread calls `log_write_wrapper()` which reads the contents of the specified file from the dequeued node and calls `log_write()` which converts the contents of the file to hex in 4080 byte reads and writes them to the logfile in the correct format. I chose 4080 bytes as a buffer size because it was reasonably large while still being divisible by 20.

`log_write()` works by looping through each byte of data in the `buffer` and writing the 2 character hex code to the `log_file_descriptor` using `pwrite()` with argument `my_offset` (where `my_offset` is initialized the the current length of the log file), incrementing `my_offset` by the amount written to the logfile. When the index of the `buffer` % 20 == 0, `log_write()` writes a newline followed by the index + `total_read` (total amount read from the file to be logged) to `log_file_descriptor` as a 0 padded 8 character integer, again incrementing `my_offset` by the amount written to the logfile.

After writing all the content to the logfile, the thread writes a `"========\n"` using `pwrite()` with argument `my_offset`.

The function then returns `my_offset` so `log_write_wrapper()` can know how much of the file has been logged.

I chose to have a dedicated logging thread because of performance. I wanted my worker threads to be able to just respond to clients and not have to log (which would slow them down considerably). Having a dedicated logging thread allows my server to log many requests to large files without noticeable delay to the clients.

Healthcheck:

If a thread notices the filename is “healthcheck”, the thread will simply respond to the client with a string of the format

`HTTP/1.1 200 OK\r\nContent-Length: X\r\n\r\n54\nY`

Where X = `total_errors` and Y = `total_requests`.

Proof there is no deadlock:

The order to which threads use the locks in my implementation is linear.
The 4 locks which could be used by the threads can be numbered as:

1. `q_lock`
2. `total_requests`
3. `total_errors`
4. `q_lock_log` (the lock for the logging queue, only grabbed if logging is enabled)

For each of these locks, the thread will not acquire the next lock until the previous lock is released.
Circular wait is avoided. A process holding resource *n* cannot wait for resource *m* if *m* < *n*.

Citations:

https://www.youtube.com/user/jms36086/about
https://www.geeksforgeeks.org/queue-linked-list-implementation/
https://stackoverflow.com/questions/7775991/how-to-get-hexdump-of-a-structure-data