TDT4186 Operating Systems
Practical Exercise 2 - Multithreaded Server

Håvard R. Krogstie
Magnus Lykke Dahlø
Herman Håberg Seternes

March 24, 2022

(a) The logic for the single threaded web server is in `http.c` and `files.c`, handling HTTP header parsing+writing and file paths+I/O respectively. The `main.c` file does the server socket creation, binding, and accepting, both on IPv4 and IPv6, but that file is of course no longer single threaded.

(b) Counting semaphores are implemented in `sem.c`, using a counter, mutex and condition variable per semaphore.

(c) The ring buffer is implemented in `bbuffer.h`. It has an array, a head and a tail, and uses one semaphore for counting the number of items, another one counting the number of empty spaces, and a single mutex to keep repeated `bb_add` and `bb_get` calls from interleaving their critical sections.

Importantly, the semaphores cause `bb_get` to block on an empty queue, until another call to `bb_add` has fully finished inserting a new item. Same goes the other way for full queues.

(d) The `main.c` file parses the arguments, creates a global `BNDBUF` with the given amount of slots, and spawns a thread pool with the given amount of threads. Then it starts listening on a server socket, and enters an infinite loop accepting incoming connections. For each incoming client socket, it is added to the queue.

All the threads in the thread pool are in their own infinte loops, waiting for something to appear in the queue. Only one client thread will get any given client socket, thanks to the `BNDBUF`.

Once the thread has a client socket, it enables timeouts on that socket, to prevent the thread from ever getting stuck calling `recv` on a socket that never sends anything. The thread keeps calling `recv` and storing the bytes in `recv_buffer` until a newline arrives. At which point the `handle_get_request` function from `http.c` gets called, to parse and respond to the request.

The parser extracts the path part of e.g. `GET /index.html HTTP/1.1`, and calls `get_file_from_path` in `files.c`. The function also takes in the webroot that was supplied as a command line argument, to know which folder to look in.

The `get_file_from_path` function stores its result in a `malloc`ed region, and returns the size of the file. The `handle_get_request` function creates a proper HTTP

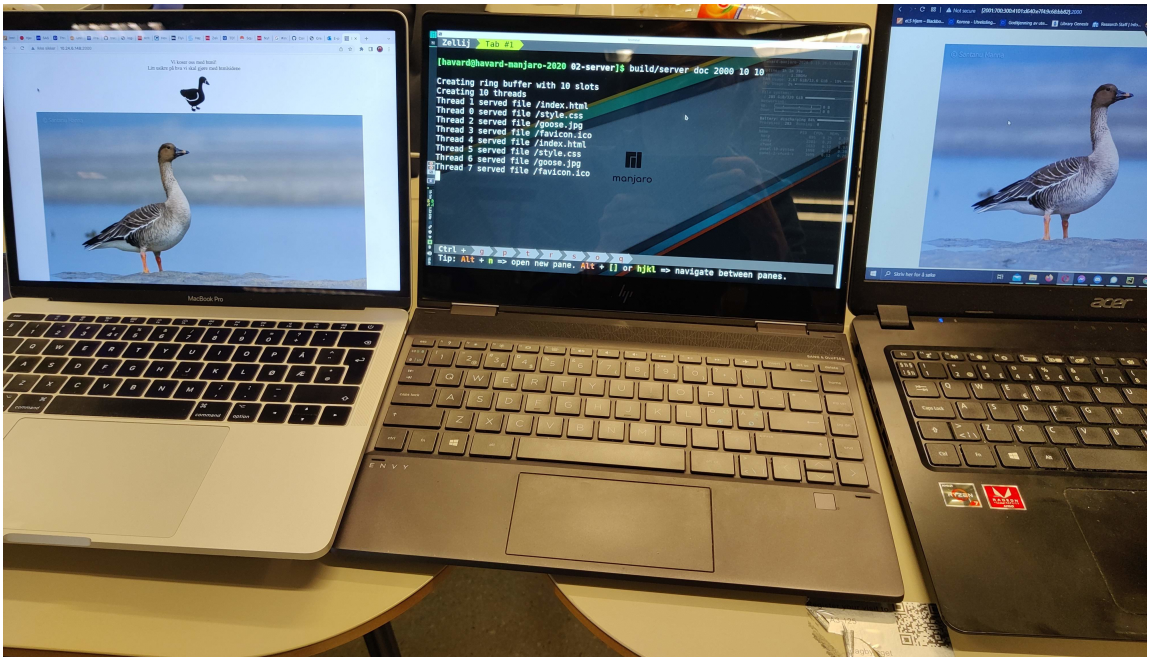

Figure 1: One machine serving on the local WiFi to two other machines. The images are also served from the server.

response containing a HTTP header with `Content-Length:` and `Content-Type:`, and the file content, and stores it in `send_buffer`. If the file didn't exist, that is handled, and a 404 HTTP header is written to `send_buffer` instead.

Note that our example website contains an image served from the server, which required handling files with null bytes, and setting correct MIME types in the response headers. We decided to use `HTTP/1.0` since Firefox failed to display the website returned with `HTTP/0.9`. See figure 1 to see two machines loading the website and its css + image from the middle machine, which prints out which thread served each file. The right machine uses IPv6.

A final important note is that `recv_buffer` and `send_buffer` are marked as `_Thread_local`, which allows all our threads to write to them at once without messing with each others' data.

(e) A naive webserver can be easily exploited if no security measures has been added. By adding a lot of `../` to the path in our request to the server, we can get access to

files outside of the webroot folder. This will let us access files like the `/etc/passwd` file, which contains the (hashed) passwords of the users on the machine.

One could solve this issue in multiple ways, but two of them are:

- Ignore all `.` chars after `/` chars. This sadly makes it impossible to host files or folders with names starting in `.`, but this is what we decided to do, since it was easy to implement. We could also try to specifically target the sequence `/../`, which would be more precise.

- Use the operating system, by running the server as a special user that only has access to the webroot folder, the OS will prevent the server from ever serving any other files. This is probably a better method, as we don't fully know what kinds of funky path parsing exists in the OS.

We implemented the first solution in our code, and only allow dots after a filename has already started. This solution is implemented in `files.c` in the function `join_path`. This function ignores all dots after slashes to escape going upward in the file structure.