Servers

In this section, you will implement several different servers. With the conditional compilation preprocessor directives (i.e. the `#ifdef` directives), we only need to change how we call the request handler in each of these different servers.

Fork server

Implement `forkserver`. You won't be writing much new code.

- The child process should call `request_handler` with the client socket fd. After serving a response, the child process will terminate.
- The parent process will continue listening and accepting incoming connections. It will not wait for the child.
- Remember to close sockets appropriately in both the parent and child process.

Thread server

Implement `threadserver`.

- Create a new `pthread` to send the proper response to the client.
- The original thread continues listening and accepting incoming connections. It will **not** join with the new thread.

Pool server

Implement `poolserver`.

- Your thread pool should be able to concurrently serve exactly `--num-threads` clients and no more. Note that we typically use `--num-threads + 1` threads in our program. The original thread is responsible for accepting client connections in a while loop and dispatching the associated requests to be handled by the threads in the thread pool.
- Begin by looking at the functions in `wq.h`.

    - The original thread (i.e. the thread you started the `httpserver` program with) should `wq_push` the client socket file descriptors received from `accept` into the `wq_t work_queue` declared at the top of `httpserver.c` and defined in `wq.h`.
    - Then, threads in the thread pool should use `wq_pop` to get the next client socket file descriptor to handle.
- You'll need to make your server spawn `--num-threads` new threads which will spin in a loop doing the following:
    - Make blocking calls to `wq_pop` for the next client socket file descriptor.
    - After successfully popping a to-be-served client socket fd, call the appropriate request handler to handle the client request.
