Introduction

In this assignment, you will develop three versions of a Web Server, relying on a custom-made networking library called `libWildcatNetworking.so`.

When you type in your browser `http://davidson.edu/about`, your browser is performing the following steps:

1. Open a connection to the machine that the name `davidson.edu` resolves to an IP address (use `dig davidson.edu` in your terminal to figure out which address).
2. Send a request for a file in this (minimal) form:

   ```
   GET /monsters_inc.jpeg HTTP/1.1
   Host: localhost:4000
   ```

3. The server waits for these requests, parses the request, extracts the filename (in this case, "about"), and sends a response in this (minimal) form:

   ```
   HTTP/1.1 200 OK
   Filename: monsters_inc.jpeg
   Content-Length: 414372

   xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx (file contents)
   xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
   ```

Our networking library, `libWildcatNetworking.so`, will provide us functions to create connections via the Internet, and functions to parse HTTP headers. You are also given a skeleton containing a bare-bones structure of a naive webserver, which you will extend and transform into a capable server. You will see that *all* improvements result from (i) enforcing good programming practices, such as error handling; and, most importantly (ii) using OS syscalls efficiently. Here is an overview (read the step-by-step instructions before implementing!):

- Part 1: Implement I/O. Your bare-bones skeleton code accepts connections from clients with the `accept_client` function in `libWildcatNetworking.so` [`server_fork.c:54`]. What you get in return is a UNIX file descriptor, which you can read from and write to, just like a file. UNIX I/O is *uniform* in the sense that performing I/O in a file descriptor referencing a file on disk is not different from doing the same over an Internet connection. A unix descriptor to a network peer is also called a *socket*.

  In this part, you will implement the read/write procedures that receive the request and transmit the response (including the file contents). All those changes are done in `clients_common.c`.

- Part 2: Multi-client support with `fork()`. After implementing the read/write functions, your webserver still has a fatal flaw. Once a client connects, no other client can be handled! Could you imagine if we had to wait another user downloading a 1TB file while we are in the queue to obtain `http://davidson.edu/about`?

  Your first version of multi-client support will fork a child process for every client that connects to your server. That way, multiple clients can download files simultaneously.

- Part 3: Multi-client support with `select()`. You are going to implement an alternative *design* for multi-client support. The problems with the previous design based on `fork()` are: (i) forking is expensive, as it involves copying a large memory footprint to another process, which is potentially too expensive just to handle a small file transfer (most file transfers are small); and (ii) one could launch a denial of service attack on your server: connect many, many times until your server cannot create child processes anymore (there's an OS limit for how many). Now, legitimate users will have their connections dropped.

  The idea with the `select()` approach is to handle multiple file descriptors, for multiple clients, in a coordinate way within *a single process*. The design is more complicated, but much more effective.

- Part 4: Multi-client support with threads. The approach based on `select()` still has some performance problems that a large-scale, production server could not tolerate. The issue we will be addressing happens because every time we read from a file on disk (in order to send that data to the requesting client, so it's an inevitable operation), our *whole process blocks*. The server loses its quantum with the OS, which will schedule a different application instead.

  Our design approach for this version is to have a *thread pool* that will serve client requests. A server launches a predetermined number of threads, and every time a client connects, you will put the request to the thread pool. Clients will be handled one by one, and blocking calls to read files from disk will *not* block the whole server, but a single thread handling the client.

Setup

You can implement this homework in any UNIX-like system (including FreeBSD, OpenBSD, Linux, macOS). The easiest programming/debugging setup is using VSCode. You have been provided with a skeleton directory with VSCode build scripts pre-set.

If you are not using the Watson 132 Linux setup:

- You may need to modify the makefile to call the appropriate compiler if you don't have `clang` installed.
- You may need to specify the path for dynamic libraries differently in `launch.json` (for example, on MacOS you would need to replace `LD_LIBRARY_PATH` with `DYLD_LIBRARY_PATH`).

Note that there is a `Makefile` that takes care of compiling the code for you. If you run

```
make webserver-clean
```

then a cleanup followed by a build will be performed. Please verify the following:

- `-I. -Inet` means that when we perform an `#include` in C, the compiler should search for the current (".") directory and the "net" directory.
- `-Lnet` means that when the compiler is linking all the separately-compiled files together into a single executable, it should search for library references in the "net" subdirectory.
- `-lWildcatNetworking` means that this specific library will be referenced in the linking of the final executable and should be used.

The VSCode build scripts simply call the `Makefile` rules, so you can use Ctrl-Shift-B to build your executable. Please verify this by observing the `tasks.json` file.

Part 1: I/O implementation

In this part, you will implement the basic I/O calls so that we have an operational webserver.

Step 1 (difficulty = M) : Complete the `read_request` function.

This function reads the HTTP request from the client, and if the `header_complete` function returns true, then it calls the `switch_state` function. Before calling the latter, you have to parse the HTTP request for the filename and protocol used by the client. You can do that using the function `get_filename`. A naive implementation of this method is provided, but this implementation is not prepared for situations such as reading only half of the header in the first call to `read()`, and it does not handle errors in the header parsing done by `get_filename`.

You should amend this function, and read up to `(BUFFER_SIZE - 1)` characters from `client->socket`. Every time you read a chunk of bytes, increment the `client->nread` variable with the total number of bytes read. While the client is still sending data (see below how you can test this), you should make subsequent calls to `read()`. Make sure to not overwrite the current buffer, but append to the string read so far.

Every time you read a chunk of data from the client, call `header_complete` to test whether the full request header has been received. If true, switch state as specified in the skeleton code, but make sure that you perform error checking in `get_filename`. If the latter function fails, it means that the request is not appropriate; in this case, set `client->status` to `STATUS_BAD`, call `finish_client`, print an error message to `stderror`, and return 0. Otherwise, if `get_filename` succeeds, call `switch_state` with the filename and protocol that you have just obtained.

More details in the comments of the `read_request` function in `clients_common.c`.

Step 2 (M) : Complete `flush_buffer` function.

This function is called when the buffer contains x meaningful bytes, `client->ntowrite` contains a value ≥ 0, and `client->nwritten` is 0. Every time we write y bytes, we add y to `client->nwritten`, and subtract y from `client->ntowrite`. Return 1 if all the writes complete successfully; otherwise print a message to `stderror` and return 0.

More details in the comments of the `flush_buffer` function in `clients_common.c`.

Step 3 (M) : Complete the `write_reply` function.

This function writes the HTTP reply to the client, and then, if the `client->file` pointer is non-null, it iteratively reads chunks of the file into `client->buffer` and uses `flush_buffer` to send it to the client. Before calling `flush_buffer`, make sure to set `client->nwritten` to 0, and `client->ntowrite` to however many bytes you read.

More details in comments of the `write_reply` function in `clients_common.c`.

Step 4 (E) : Complete the `obtain_file_size` function.

You should use the `stat(2)` syscall to obtain the size of the file passed as parameter and return it. If `stat(2)` fails, print an error and return -1.

More details in the comments of the `obtain_file_size` function in `clients_common.c`.

Step 5 (E) : Fix the `switch_state` function to handle 403 and 404 HTTP messages.

Right now, the function `switch_state`, which is called when the header was received completely, simply assumes that the file exists and it's accessible, and opens it using `fopen(3)`. You should actually use the suggested code in the comments in the `switch_state` function in `clients_common.c`, but replacing the pseudocode statements inside the `if` clause tests with the appropriate calls to `access(2)` and `fopen(3)`.

More details in the comments of the `switch_state` function in `clients_common.c`.

I/O Testing

Test your program. Run your program in a terminal with the command

```
LD_LIBRARY_PATH=net ./webserver 4000
```

If you are using macOS, remember you have to use `DYLD_LIBRARY_PATH` in order to tell the loader where to look for shared libraries.

Now, open your browser and type:

```
http://localhost:4000/monsters_inc.jpeg
```

You should see the picture of the movie Monsters Inc.

If you need to debug your program, you can launch your program using the VSCode debugger. Note that you will need to ensure that the debugger is configured to do the same things as our command line execution above: adding the `net` folder to the library path, and running the `webserver` program with the argument `4000`.

Part 2: Multi-client Support with `fork()`

In this part, you will implement multi-client server support by creating child processes. Here are the steps.

Step 6 (E) : Create a function to register signal handlers, and write appropriate signal handler functions. We'll practice this with an in-class activity.

The `SIGCHLD` handler should increase `operations_completed` only when the *exit status* of the child was `STATUS_OK`. Look at the `sigaction(2)` manual page in order to figure out how to obtain the status of a collected child.

More details in the comments of `server_fork.c`.

Step 7 (E) : Call your function to set up signal handlers for `SIGPIPE`, `SIGCHLD`, and `SIGTERM`. Look at the `sigaction(2)` manual page in order to figure out how to ignore a signal.

Step 8 (E) : Fork a different process to handle each client. Simply use the `fork()` call to create a child for every process. The piece of code below:

```
struct client *client = make_client(client_socket);
if(read_request(client)) {
   write_reply(client);
}
```

should be executed by the child process only. The parent process can safely close `client_socket` after calling `fork()`: the connection will only be closed when the last file descriptor representing the connection is closed. The child should exit with code `client->status` after executing `write_reply`.

Now that child processes are handling requests, the following piece of code:

```
if(client->status == STATUS_OK) {
   operations_completed++;
}
```

no longer makes sense, because we don't want to increment the copy of that variable in the child process. Instead, we should remove it and rely on the signal handler to increment the variable in the parent process.

Step 9 (E) : Wait for new connections, but ignore interrupted accept_client calls because of `SIGCHLD`. Many blocking routines return -1 and set the `errno` variable to `EINTR` if they are interrupted in a benign way – say, by a signal handler. You have to test the return of the `accept_client` and, if it is -1, and `errno` is set to `EINTR`, restart the call.

Fork Testing

Test your program. You have at least two options:

1. Put a large file (around 1GB) in your project's directory, and request that file in a browser tab. Open another tab and request `monsters_in.jpeg`. You should be able to receive the latter file *before* than the former.

2. Instead of relying on a large file, *simulate* a slow transfer by introducing a delay in the

`write_response` function. You can use `sleep` or `usleep` (for microsecond resolution) in order to introduce that delay.

Part 3: Multi-client Support with `select()`

You are now ready to implement multi-client support by managing multiple file descriptors at once. The `select()` system call receives a list of file descriptors for reading, a list of file descriptors for writing, and blocks until one of those descriptors could be used once (for reading or writing) without blocking. In other words, there's data available to be read, and there's some space in kernel buffers to accommodate writes by the application, which would not block by performing those operations.

Read the documentation of `select(2)` in Sec. 63.2.1 in our extra reading "The Linux Programming Interface" (available electronically from our library).

You will be adding not only the client descriptors, but also the accept descriptor into the sets passed to `select()`. If a new connection is made, the accept socket is marked as readable when `accept()` returns. If new data comes from an accepted client, its socket is marked as readable. If new data can be written to an accepted client without blocking, its socket is marked as writeable.

Despite the fact that a client can be activated multiple times for reading (request header comes in chunks) and writing (reply transfer goes in chunks), your read/write functions now only perform *one* operation, because that's the amount of operations that's guaranteed to execute without blocking (or giving an error). Therefore, the read/write functions are actually `state machines`: they use the `client->state` (**not** `client->status`) in order to register their current state between activations. The state-machine design is implemented in the `server_statemachine.c`. **This is a completely different design philosophy compared to the fork-based server.**

Add `clients_statemachine.o` to your `OBJECTS` variable in the `Makefile` so you can compile the project. Change the call from `server_fork()` to `server_statemachine()` in your main method.

Step 10 (E) : Treat signals the same way that you did in your `server_fork.c` (but only `SIGPIPE` and `SIGTERM`). That's probably a simple copy/paste from your `server_fork.c`.

Step 11 (E) : Declare (before the loop) and initialize (inside the loop) the read and write sets, according to the documentation for `select()`. You will be using the macro

FD_ZERO for that purpose.

Step 12 (E) : Add the accept socket into the read set, using FD_SET.

Step 13 (M) : Iterate over all currently accepted clients, and prepare the call to `select(2)`.

If a client has state E_RECV_REQUEST, add the client's socket into the read set. Otherwise, if the client has state E_SEND_REPLY, add the client to the write set. While you iterate, keep track of the maximum descriptor found among all client descriptors and the accept descriptor.

Step 14 (E) : Call `select()`, blocking until anything can be read from or written to. Please refer to the manual page, and the documentation pointed above for details on how to setup the syscall.

Step 15 (M) : Process `select()` when it unblocks:

Step 15.1 : Test if the accept socket has been flagged ready for reading. Use the appropriate macro described in `select(2)`.

Step 15.2 : Iterate over all currently accepted clients, and if the client is ready for reading or writing, call `handle_client`.

Part 4: Multi-client Support with Multithreading

For this version of multi-client support, you will implement the producer/consumer interface descried in `thread_pool.h`.

Step 16 (H) : Implement a producer/consumer interface where the producer calls `put_request` to insert clients into the consumption list. These clients go inside a linked list of `struct requests`, implemented internally in the module. The consumer are the NUM_THREADS threads that have been started by `start_threads`. The consumers, upon receiving a `struct request` representing a client, perform `read_request`, and, if that call executed successfully, `write_reply`. The overall structure of the code run by the threads is the following:

```
struct request *request;

while((request = get_request())) {
   struct client *client = request->client;

   if(read_request(client)) {
```

```
            write_reply(client);
        }

        if(client->status == STATUS_OK) {
            // Increment the number of operations
        }

        // make sure to free() allocated memory to avoid leaks
    }

    return NULL;
```

Finally, after executing `write_reply`, the variable `operations_completed` is incremented using an atomic operation documented here: `http://en.cppreference.com/w/c/atomic`. Atomic operations are necessary because values are being communicated between threads. You should rely either on a mutex, or on atomic operations to guarantee data propagation between threads, as discussed in class.

You should employ mutexes and condition variables *internally* in the module to implement the randezvous of producers and consumers.

The `finish_threads` function sets an internal flag within the module (using an atomic operation!), which will indicate for all the threads to stop waiting forever for upcoming requests. You have to implement a *clean finish* for the threads: so, if there are threads sleeping, they should be notified to wake up, so that they "see" the updated flag and finish the application cleanly.

Step 17 (E) : Modify your `server_fork.c` so that it calls `put_request` for a client instead of spawning a child process. Also, call `start_threads` before the server's main loop, and call `finish_threads` after you print how many oprerations have succeded.