A Home-Brew Multi-Process Web Server

Project objectives:

- Learn how to follow a network protocol (in this case, http)
- Learn network/socket programming
- Learn some simple multithreading

For this lab we'll build a simple multi-threaded Web server.

The directory contains:

- homework5.c: Skeleton code for the server side of a TCP application. This will be the primary file for this assignment. All your code should go in this file.
- WWW/: A directory containing example files for your Web server to distribute.
- thread_example.c: Example code that illustrates a very simple threaded programming scenario. You are not required to use or make any changes to this file, but you should understand what it does.
- A Makefile.

Your server program will receive two arguments: 1) the port number it should listen on for incoming connections, and 2) the directory out of which it will serve files (often called the document root in production Web servers). For example:

```
$ ./homework5 8080 WWW
```

This command will tell your Web server to listen for connections on port 8080 and serve files out of the WWW directory. That is, the WWW directory is considered '/' when responding to requests. For example, if you're asked for /index.html, you should respond with the file that resides in WWW/index.html. If you're asked for /dir1/dir2/file.ext, you should respond with the file WWW/dir1/dir2/file.ext.

Requirements

In addition to serving requested files, your server should handle at least the following cases:

- HTML, text, and image files should all display properly. You'll need to return the proper HTTP Content-Type header in your response, based on the file ending. Also, other binary file types like PDFs should be handled correctly. You don't need to handle everything on that list, but you should at least be able to handle files with html, txt, jpeg, gif, png, and pdf extensions.
- If asked for a file that does not exist, you should respond with a 404 error code with a readable error page, just like a Web server would. It doesn't need to be fancy, but it should contain some basic HTML so that the browser renders something and makes the error clear.
- Some clients may be slow to complete a connection or send a request. Your server should be able to serve multiple clients concurrently, not just back-to-back. For this lab, use multithreading with pthreads to handle concurrent connections.
- If the path requested by the client is a directory, you should handle the request as if it was for the file "index.html" inside that directory. Hint: use the stat() system call to determine if a path is a directory or a file. The st_mode field in the stat struct has what you need.
- The Web server should respond with a list of files when the user requests a directory that does not contain an index.html file. You can read the contents of a directory using the opendir() and readdir() calls. Together they behave like an iterator, that is, you can open a (DIR *) with opendir and then continue calling readdir(), which returns info for one file, on that (DIR *) until it returns NULL. Note that there should be no additional files created on the server's disk to respond to the request. The response should mimic result of running `python -m SimpleHTTPServer`

When testing, you should be able to retrieve byte-for-byte copies of files from your server. Use wget or curl to fetch files and md5sum or diff to compare the fetched file with the original. I will do this when grading. *For full credit, the files need to be exact replicas of the original.*

Grading Rubric

This assignment is worth 100 points in total, which I will assign as follows:

- 20 points for serving both text and binary files (that can be rendered correctly – set Content-Type) to a standard Web browser (e.g., Firefox).
- 20 points for serving exact copies of text and binary files to command line clients like wget or curl. The MD5 sums should match!
- 10 points for correctly returning a 404 error code and HTML message when a request asks for a file that does not exist.
- 10 points for serving index.html, if it exists, when asked for a file that is a directory.
- 20 points for handling multiple clients concurrently. Your server should be able to have one or more open telnet connections, doing nothing, while still being able to handle browser/wget/curl requests.
- 20 points for returning a file listing when asked for a directory that does not contain an index.html file. The listing should use simple HTML to provide clickable links. You may assume that any request for a directory by name will end with a `/` - i.e. if `homeworks` is a directory, you can assume that we will make requests for URLs ending in `/homeworks/` or `/homeworks/index.html` or `/homeworks/somefile.gif` and *not* `/homeworks`.
- 10 points of extra credit is available for an especially awesome 404 page

Grading will be done automatically using a script. We will publish this script after grading has completed; you are responsible for writing your own test cases. If you wish, you can share test cases you have written with the class.

Tips

- Take compiler warnings seriously. Unless it's an unused variable, you should address the warning as soon as you see it. Dealing with a pile of warnings just makes things more difficult later.
- Test your code in small increments. It's much easier to localize a bug when you've only changed a few lines.

- If you need to copy a specific number of bytes from one buffer to another, and you're not 100% sure that the data will be entirely text, use memcpy() rather than strncpy(). The latter still terminates early if it finds a null terminator ('\0').
- If you're trying to do some sort of specific string or memory manipulation, feel free to ask if there's a better/recommended way to do it rather than brute force. Often there may be a standard library function that will make things easier.

Roughly, your server should follow this sequence:

- Read the arguments, bind to the specified port, and find your document root (you might find the chdir() system call helpful).
- Accept a connection, and hand it off to a new thread for concurrent processing.
- Receive and parse a request from the client.
- Look for the path that was requested, starting from your document root (the second argument to your program). One of four things should happen:
- If the path exists and it's a file, formulate a response (with the Content-Type header set) and send it back to the client.
- If the path exists and it's a directory that contains an index.html file, respond with that file.
- If the path exists and it's a directory that does NOT contain an index.html file, respond with a directory listing.
- If the path does not exist, respond with a 404 code with a basic error page.
- Close the connection, and continue serving other clients.

Reminders

Always, always, always **check the return value of any system calls you make!** This is especially important for `send`, `recv`, `read`, and `write` calls that tell you how many bytes were read or written.