Server Documentation

The server is written in C and can be compiled with the specified Makefile by simply running `make` in the root directory, and then running the `webd` binary which is produced. The default port is 80 so you may need to run as root, for example `sudo ./webd`. You may pass the `-p` (or `--port`) flag if you wish to use a different port. Additionally the server may be ran as a daemon if the `-D` (or `--daemon`) flag is passed, in which case the server automatically runs in the background. For a full list of available flags, run with the flag `--help`.

Structure

The implementation is split into three C modules: one which deals with initialising the program and managing the listening socket, along with setting up a thread for each client (webd.c), one which manages individual threads/clients and handles the HTTP protocol management (client.c) and one which deals with initialising threads and tidying up/closing open sockets upon shutdown (threadlist.c).

webd.c

Contains a simple parser for dealing with options passed in as command line flags. It also sets up the signal handler so that sockets are correctly closed down once an interrupting signal is received. One thing introduced here is the idea of using goto statements for freeing resources. Resource acquisition is typically done as a stack; ie. acquire resource 1, acquire resource 2, free resource 2, free resource 1. This permits a structure like this:

```
int calculate() {
      int return_value = -1;
      if(acquire_resource_1() == ERROR)
            goto free_0;
      use_resource_1();
      if(acquire_resource_2() == ERROR)
            goto free_1;
      use_resource_2();
      if(acquire_resource_3() == ERROR)
            goto free_2;
      return_value = use_resource_3();

      free_resource_3();
free_2:
      free_resource_2();
free_1:
      free_resource_1();
free_0:
      return return_value;
}
```

Which makes the correct freeing of allocated resources easier as you can use labels to jump to the correct “point” in the freeing sequence.

This module also handles the splitting of the server process into a daemon, should the `-D` (or `--daemon`) flag be passed on the terminal.

threadlist.c

This keeps track of each thread, and the file descriptor of the websocket associated to the thread, as a linked list. This is mainly so that all open sockets can be closed when the server is shut down. The pthread library is used for threading.

client.c

Inbound requests are read into an expandable buffer. The request is processed a line at a time (ie. the buffer is cleared and read from the beginning once \r\n is read); each time a full line is received it is processed by the rq_line function. The first line read is used to determine the path received, HTTP version and method (GET/OPTIONS/etc), and successive lines read in header values. For the sake of a simple implementation, rather than storing all headers and then checking each one to read in required info, each header line is simply checked at read time whether it is a header with useful info in (such as Accept-Encoding) and if so the required info is extracted. Otherwise, the info is discarded.

Additionally, while POST requests are superficially supported by the server, the body of the request is not read at all; instead the request body is silently swallowed and is treated as an identical GET request.

Path Parsing

The path is parsed in two stages. The first stage is syntactic, and strips any web-specific parts of the path such as the query string and the HTML anchor string, which is done in the `rq_parse_path` function. This occurs when the path is read in. The second stage is semantic and prepends the path with the root directory of the HTTP server (by default this is `.`, the current working directory), and also appends `index.html` (or whichever index filename is configured) to the path if it is a directory or ends with a slash.

Response Formation

Once an empty line is received the `rq_handle` function is called which looks at the data received about the request and writes the relevant response to the stream. Again, for simplicity of this particular implementation, the HTTP responses are pre-formed as format strings, with the relevant values interpolated in; the POSIX function `dprintf` is used to write this directly to the socket. Identification of the MIME type of the requested resource is done by looking at the file extension. As there is no standardised way of doing this, for now a simple collection of some image types and HTML-relevant extensions such as CSS, JavaScript and plaintext are supported; everything else will be presented with the MIME type `application/octet-stream`.

Compression

DEFLATE compression is supported by the server when the client accepts it as an encoding. This uses the zlib implementation of DEFLATE. It is implemented on the server with a simple check before the 200 OK response is sent, in the `rq_compress` function. This is passed a pointer to the `FILE *` pointer. It checks if DEFLATE is an accepted encoding. If so, it creates a temporary file using `tmpfile`, compresses the contents of the requested file and writes the compressed data to it. It then closes the original file and sets the passed pointer to the original `FILE *` to the new temporary file. This allows the rest of the server code to remain unchanged; files created with `tmpfile` are deleted when they are closed and so this simply plugs in to the existing code with no additional work. The method used to compress with DEFLATE is based on the reference provided by zlib on the project website.