
A Simple Web Server in C

In this project, we'll finish the implementation of a web server in C.

What you need to write:

- HTTP request parser
- HTTP response builder
- LRU cache
 - Doubly linked list (some functionality provided)
 - Use existing hashtable functionality (below)
- Your code will interface with the existing code. Understanding the existing code is an expected part of this challenge.

What's already here:

- `net.h` and `net.c` contain low-level networking code
- `mime.h` and `mime.c` contains functionality for determining the MIME type of a file
- `file.h` and `file.c` contains handy file-reading code that you may want to utilize, namely the `file_load()` and `file_free()` functions for reading file data and deallocating file data, respectively (or you could just perform these operations manually as well)
- `hashtable.h` and `hashtable.c` contain an implementation of a hashtable (this one is a bit more complicated than what you built in the Hashtables sprint)
- `llist.h` and `llist.c` contain an implementation of a doubly-linked list (used solely by the hashtable—you don't need it)
- `cache.h` and `cache.c` are where you will implement the LRU cache functionality for days 3 and 4

What is a Web Server?

A web server is a piece of software that accepts HTTP requests (e.g. GET requests for HTML pages), and returns responses (e.g. HTML pages). Other common uses are GET requests for getting data from RESTful API endpoints, images within web pages, and POST requests to upload data to the server (e.g. a form submission or file upload).

Reading

- Networking Background

-
- Doubly-Linked Lists
 - LRU Caches
 - MIME types

Assignment

We will write a simple web server that returns files and some specialized data on a certain endpoint.

- `http://localhost:3490/d20` should return a random number between 1 and 20 inclusive as `text/plain` data.
- Any other URL should map to the `serverroot` directory and files that lie within. For example:

```
1 http://localhost:3490/index.html
```

should serve the file

```
1 ./serverroot/index.html
```

Examine the skeleton source code in `server.c` and `cache.c` for which pieces you'll need to implement.

IMPORTANT Spend some time inventorying the code to see what is where. Write down notes. Write an outline. Note which functions call which other functions. Time spent up front doing this will reduce overall time spent down the road.

The existing code is all one big hint on how to attack the problem.

For the portions that are already written, study the moderately-well-commented code to see how it works.

There is a `Makefile` provided. On the command line, type `make` to build the server.

Type `./server` to run the server.

Main Goals

Read through all the main and stretch goals before writing any code to get an overall view, then come back to goal #1 and dig in.

Part 1

1. Implement `send_response()`.

This function is responsible for formatting all the pieces that make up an HTTP response into the proper format that clients expect. In other words, it needs to build a complete HTTP response with the given parameters. It should write the response to the string in the `response` variable.

The total length of the header **and** body should be stored in the `response_length` variable so that the `send()` call knows how many bytes to send out over the wire.

See the HTTP section above for an example of an HTTP response and use that to build your own.

Hint: `sprintf()` for creating the HTTP response. `strlen()` for computing content length. `sprintf()` also returns the total number of bytes in the result string, which might be helpful. For getting the current time for the Date field of the response, you'll want to look at the `time()` and `localtime()` functions, both of which are already included in the `time.h` header file.

The HTTP `Content-Length` header only includes the length of the body, not the header. But the `response_length` variable used by `send()` is the total length of both header and body.

You can test whether you've gotten `send_response` working by calling the `resp_404` function from somewhere inside the `main` function and passing it the `newfd` socket (make sure you do this *after* the `newfd` socket has been initialized by the `accept` system call in the while loop). If the client receives the 404 response when you make a request to the server, then that's a pretty clear indication that your `send_response` is working.

2. Examine `handle_http_request()` in the file `server.c`.

You'll want to parse the first line of the HTTP request header to see if this is a `GET` or `POST` request, and to see what the path is. You'll use this information to decide which handler function to call.

The variable `request` in `handle_http_request()` holds the entire HTTP request once the `recv()` call returns.

Read the three components from the first line of the HTTP header. Hint: `sscanf()`.

Right after that, call the appropriate handler based on the request type (`GET`, `POST`) and the path (`/d20` or other file path.) You can start by just checking for `/d20` and then add arbitrary files later.

Hint: `strcmp()` for matching the request method and path. Another hint: `strcmp()` returns 0 if the strings are the *same*!

Note: you can't `switch()` on strings in C since it will compare the string pointer values instead of the string contents. You have to use an `if-else` block with `strcmp()` to get the job done.

If you can't find an appropriate handler, call `resp_404()` instead to give them a "404 Not Found" response.

3. Implement the `get_d20()` handler. This will call `send_response()`.

See above at the beginning of the assignment for what `get_d20()` should pass to `send_response()`.

If you need a hint as to what the `send_response()` call should look like, check out the usage of it in `resp_404()`, just above there.

Note that unlike the other responses that send back file contents, the `d20` endpoint will simply compute a random number and send it back. It does not read the number from a file.

The `fd` variable that is passed widely around to all the functions holds a *file descriptor*. It's just a number used to represent an open communications path. Usually they point to regular files on disk, but in this case it points to an open *socket* network connection. All of the code to create and use `fd` has been written already, but we still need to pass it around to the points it is used.

4. Implement arbitrary file serving.

Any other URL should map to the `serverroot` directory and files that lie within. For example:

`http://localhost:3490/index.html` serves file `./serverroot/index.html`.

`http://localhost:3490/foo/bar/baz.html` serves file `./serverroot/foo/bar/baz.html`.

You might make use of the functionality in `file.c` to make this happen.

You also need to set the `Content-Type` header depending on what data is in the file. `mime.c` has useful functionality for this.

Part 2 Implement an LRU cache. This will be used to cache files in RAM so you don't have to load them through the OS.

When a file is requested, the cache should be checked to see if it is there. If so, the file is served from the cache. If not, the file is loaded from disk, served, and saved to the cache.

The cache has a maximum number of entries. If it has more entries than the max, the least-recently used entries are discarded.

The cache consists of a doubly-linked list and a hash table.

The hashtable code is already written and can be found in `hashtable.c`.

1. Implement `cache_put()` in `cache.c`.

Algorithm:

- Allocate a new cache entry with the passed parameters.
- Insert the entry at the head of the doubly-linked list.
- Store the entry in the hashtable as well, indexed by the entry's `path`.
- Increment the current size of the cache.
- If the cache size is greater than the max size:
 - Remove the cache entry at the tail of the linked list.
 - Remove that same entry from the hashtable, using the entry's `path` and the `hashtable_delete` function.
 - Free the cache entry.
 - Ensure the size counter for the number of entries in the cache is correct.

2. Implement `cache_get()` in `cache.c`.

Algorithm:

- Attempt to find the cache entry pointer by `path` in the hash table.
- If not found, return `NULL`.
- Move the cache entry to the head of the doubly-linked list.
- Return the cache entry pointer.

3. Add caching functionality to `server.c`.

When a file is requested, first check to see if the path to the file is in the cache (use the file path as the key).

If it's there, serve it back.

If it's not there:

- Load the file from disk (see `file.c`)
- Store it in the cache
- Serve it

There's a set of unit tests included to ensure that your cache implementation is functioning correctly. From the `src` directory, run `make tests` in order to run the unit tests against your implementation.

Stretch Goals

1. Post a file

1. Implement `find_start_of_body()` to locate the start of the HTTP request body (just after the header).
2. Implement the `post_save()` handler. Modify the main loop to pass the body into it. Have this handler write the file to disk. Hint: `open()`, `write()`, `close()`. `fopen()`, `fwrite()`, and `fclose()` variants can also be used, but the former three functions will be slightly more straightforward to use in this case.

The response from `post_save()` should be of type `application/json` and should be `{"status": "ok"}`.

2. Automatic `index.html` serving We know that if the user hits `http://localhost:3490/index.html` it should return the file at `./serverroot/index.html`.

Make it so that if the user hits `http://localhost:3490/` (which is endpoint `/`, on disk `./serverroot/`), if no file is found there, try adding an `index.html` to the end of the path and trying again.

So `http://localhost:3490/` would first try:

```
1 ./serverroot/
```

fail to find a file there, then try:

```
1 ./serverroot/index.html
```

and succeed.

3. Expire cache entries It doesn't make sense to cache things forever—what if the file changes on disk?

Add a `created_at` timestamp to cache entries.

If an item is found in the cache, check to see if it is more than 1 minute old. If it is, delete it from the cache, then load the new one from disk as if it weren't found.

You'll have to add a `cache_delete` function to your cache code that does the work of actually removing entries that are too old from the cache.

4. Concurrency *Difficulty: Pretty Dang Tough*

Research the pthreads library.

When a new connection comes in, launch a thread to handle it.

Be sure to lock the cache when a thread accesses it so the threads don't step on each other's toes and corrupt the cache.

Also have thread cleanup handlers to handle threads that have died.