Supplement: HTTP Daemon

Leonard König, April 12, 2021

Web servers are just file servers

The file server in the last exercise worked like this: A client connected to the server, sent the name of a file `foo.txt` and the server replied with the files contents. While rather primitive it works rather fine and is, in fact, quite close to what web servers (or http daemons) do! The Hyper-Text-Transfer-Protocol is a bit more chatty, so instead of `foo.txt` the client may send something like this (the last line is an empty line!).

```
GET /foo.txt HTTP/1.0
Host: 127.0.0.1

```

A request like this is also generated by the browser, e.g. if you enter `http://www.mywebsite.com/index.html` in the URL bar, it will connect to the server refferred to by the domain `www.mywebsite.com` (get the IP via DNS) and then send something like the following:

```
GET /index.html HTTP/1.0
Host: www.mywebsite.com

```

This instructs the server to open the file `index.html` within the servers webroot directory and send it back to the client and it notes that the server was accessed over the domain in the `Host` field. Again, it doesn't just send the file but prepends the transmission with another HTTP header:

```
HTTP/1.0 200 OK
Content-Type: text/html
Connection: close
Content-Length: 38

<html>This is the file content</html>
```

The server here tells us that the file could be retrieved correctly (`200 OK`), that it's an HTML file of 38 B length. It also notifies the client that it will close the connection after having sent the full file. Another well-known HTTP status code is `404 NOT FOUND`, often accompanied with a **cute status page**.

`scanf(3)` Functions

We now extend our file server! The first step is the *parsing* of the HTTP request, i.e. 'extracting' the information, most crucially the file path, from a string of data. Until now, we mostly did the opposite, we took information, like integers, other strings, etc. and *printed* these as a formatted string:

```
printf("Foo %d bar %d\n", 42, 100);
```

What if we have the string 'Foo 42 bar 100' and want to do the opposite: Find the value of between 'Foo' and 'bar'?

Fortunately, the C standard library provides us with functions to do just that, namely the **`scanf(3)`** family:

```
char s[] = "Foo 42 bar 100"
int d;
sscanf(s, "Foo %d", &d);
```

Here, we asked the function to 'scan' over the provided string `s`, searching for the given pattern and to 'extract' the decimal `"42"` and convert it into the number 42, writing the result into the variable `d` passed via a pointer to the function.[1] But what if the function was *unable* to match due to the string being of a different format? Or only the first few variables could be matched til the format specification and the input string disagreed? In that case we need to check the return value of `sscanf(3)` which specifies the number of successfully matched items—in the above example this can only be `0` (no match) or 1 (`%d` matched).

[1] Why do we need a pointer here, but didn't in `printf`?

I've now quietly used the `sscanf(3` function of the `scanf(3)` family and not `scanf(3)` or `fscanf(3)`. All these functions do basically the same thing, however `fscanf(3)` reads from a `FILE *` (we use `fd` so it won't work) and `scanf(3)` reads from the special standard input `FILE *stdin` which is not what we want. Hence we need to `recv(3)` the HTTP request like we read the file name in the exercise before, and then use `sscanf(3)` to extract the file name.[2]

[2] Note that `recv(3)` doesn't yield a string (NUL-terminated) but just the bytes read and returns the number of these. `sscanf(3)` needs that NUL byte, so you need to append it manually.

Gathering metadata & Sending the header

We now know which file to send and can prepare our servers answer. That answer includes a header before the actual file content as well. It contains metadata about the file we are about to send, most crucially the number of bytes the file contains.[3]

[3] Why is that needed or useful?

Unfortunately this means we need to know how big our file is before having read it—we thus need to *seek* til the end, then look at what position in the file we are, note down this number, and then seek back to the start. Depending on whether you opened the file with `open(3)` or `fopen(3)`, you need to use `lseek(3)` or `fseek(3)` and `ftell(3)` respectively.

The header also often contains the file type in form of a MIME type specification (e.g. `text/html` or `image/jpeg`), but this is not strictly required.

In order to prepare the header to be sent back to the client/browser, it makes sense to use the `printf(3)` function family. Again, we cannot use `fprintf(3)` as we are 'printing' to a socket/fd, but we can use `sprintf(3)` to write our prepared string into a buffer which we then can `send(3)`.

```
char buf[1000];
sprintf(buf, "Foo %d bar %d", 42, 100);
```