Assignment 4

March 6, 2015

Overview

In this assignment, you will write a basic web server that handles HTTP 1.0 requests.

This web server will not directly listen for network connections; that is assignment 5. Instead, it will read an HTTP request on its standard input and write the response to standard output. There are many Unix network servers that have historically operated in this fashion; a master program called `inetd` listens for network connections and uses these kinds of programs to serve them.

Introduction to HTTP

HTTP (Hyper-Text Transfer Protocol) is a network protocol that is used by web servers and browsers to transfer web pages. We will be implementing version 1.0 of the protocol; you can read its full details, if you want, in the specification.

HTTP communications are made up of *requests* and *responses*. If you point your browser to `http://google.com/`, it will send a request like the following:

```
GET / HTTP/1.0
Host: google.com
```

To which Google will reply:

```
HTTP/1.0 200 OK
Content-Type: text/html

<html content>
```

A request consists of 3 things:

1. The *request line*, that tells the server what file or path is requested and what operation to perform (typically `GET`; sometimes `HEAD` or `POST`), along with the HTTP version it speaks. This is expressed as an absolute path; a simple server like ours will interpret that as the path to some file or directory in its *document root*.

2. A set of *headers*, providing additional information about the request. These are of the form `Name: value`; the name is case-insensitive. The headers are terminated with a blank line.

3. The *request body*; `POST` requests can have additional information associated with them. For this assignment, we will only handle `GET` and `HEAD` requests, so the request body can be safely ignored.

Similarly, the response consists of 3 parts:

1. The *status line*, stating the HTTP version, a status code (indicating success or failure; 200 is the usual success code), and an explanation of the status (usually 'OK' for 200).

2. Headers providing information about the returned file (what type it is, etc.), as well as any additional information needed to describe how the response will be sent.

3. The content of the file (or error message).

Everything except for the content of both the request and response has its lines terminated with both a carriage return and a line feed (`\r\n`), rather than just a line feed (`\n`).

Formatting Errors

When you format an error, it is adequate to respond with an error description in plain text. For example, the following is an appropriate Not Found response:

```
HTTP/1.0 404 Not Found
Content-Type: text/plain

The resource '/wombat' could not be found.
```

Provided Files

I am providing several files to you:

- `Makefile` builds the program; if you type `make`, it will build the program `httpd`.

- Several C files and corresponding header files (`request`, `hdrlist`, and `mimetypes`) to implement some of the functionality for you.

- A shell script `listen.sh` that uses a program called `socat` to listen to network connections and run your web server to handle each one.
- A directory of HTML files for testing. This is a recent snapshot of the static content of the web site for LensKit, the software developed by my research group.

For this assignment, you may also use the GLib library, version 2.36.

Library Functions

I am providing 2 libraries for you, `request` and `mimetypes` (`hdrlist` is used internally by `request`).

The `request` library parses HTTP requests, so you do not need to do that yourself. It provides a type, `req_t`, and several functions:

`req_create()` Creates a new request.

`req_destroy(req_t*)` Destroys a request, freeing its memory.

`req_is_complete(req_t*)` Test whether a request is complete (either an entire request has been parsed, or an error has ben encountered that invalidates the request).

`req_is_valid(req_t*)` Test whether a request is valid. A complete but invalid request is the result of the browser sending bad data (or your server code corrupting the data somehow).

`req_method(req_t*)` Get the HTTP *method* (`GET`, `HEAD`, or `POST`) for the request.

`req_path(req_t*)` Get the path requested in the request.

`req_header(req_t*, const char *name)` Get a header from the request.

`req_host(req_t*, const char *name)` Get the `Host:` header from the request, if there is one.

`req_parse(req_t *req, const char *data, size_t len)` Parse some data into the request. This function takes data, and parses it until it has parsed a complete request or it is out of data. It returns the number of bytes from `data` that it has used.

Here is the pattern for using the request library:

```
create new request
while request is not complete:
    read data from input
    parse data into request
if request is valid:
    handle and send response
```

```
else:
    send error (400 Bad request)
destroy request
```

The `mimetypes` library has a single function, `guess_content_type`, that takes a file name and returns the type of data that is probably in that file, suitable for the `Content-Type` header in the response.

Requirements

Write the basic web server. The Makefile provided assumes that your code is in `httpd.c`; if you create additional files, modify the `HTTPD_SOURCES` variable near the top of the `Makefile` to add your new source files.

Your server will receive a single command line argument: the path to the directory from which files are to be served (the *document root*).

Sending File Content (65 points)

The basic functionality of the program is the ability to send a file in response to a request. The file requested is the *path* in the request (`req_path`).

While the path returned is an absolute path (starts with `/`), the web server interprets the path as if the root were the document root. So a request for the path `/hello.html` sent to a server with a document root of `/srv/www` should result in the contents of the file `/srv/www/hello.html`.

For the program's basic functionality, you need to do the following:

1. Parse a request from standard input
2. If the request is invalid, respond with a '400 Bad Request' error.
3. Validate the method. If it is POST, respond with '405 Method Not Allowed'.
4. Validate the request path for security. If the path contains any `..` elements, return a '400 Bad Request' error.
5. Get the content type of the file (using `guess_content_type` and giving it the path).
6. Open the file for reading.
    - if the open failed because the file does not exist (`errno` is `ENOENT`), respond with '404 Not Found'.
    - if the open failed because access was denied (`errno` is `EACCES`), respond with '403 Forbidden'.
    - if it failed for some other reason, respond with `500 Internal Server Error`, and log the error to standard error with `perror`.
7. Start a response

- send `200 OK`
- send the headers, if any; if the file type was guessable, send it as the value of a 'Content-Type' header.
- send the blank line

8. Send the content of the file
9. Close file and exit successfully

Note: Be careful to make sure you terminate the HTTP response lines properly. They should be followed with the two-character '`\r\n`' sequence instead of just '`\n`'. The content of the file should be sent as-is.

Note: Your program **must not** send any extraneous output to standard output. The only data on standard output should be the HTTP response. Use standard error (with `fprintf`, for example) for logging any diagnostic output.

Handling Directories (10 points)

If the requested path refers to a directory, some special care is needed. Between steps 4 and 5 in the basic functionality, insert the following logic:

1. Use `stat` to check the requested path. If it is not found (`errno` is `ENOENT`), then return '404 Not Found'.

2. If the requested path is a directory, then:

   - if the last character of the path is '`/`', then proceed as if the client requested the file `index.html` located in the specified directory.
   - otherwise, send a *redirect* to tell the browser to request the path with a '`/`' at the end. So, if they request `/foo`, and `foo` is a directory in the document root, send the following response:

     ```
     HTTP/1.0 301 Moved Permanently
     Location: http://localhost:4080/foo/
     ```

     To get the hostname portion, do not hardcode it; instead, look up the `Host` header from the request with `req_host`. If there is no host, then just return the path (`/foo/`) in the `Location:` header.

File Information Headers (10 points)

Use the information from `stat`-ing the file to populate the following headers in the response, in addition to the `Content-Type` header:

- `Content-Length` must contain the size of the file, in bytes, as a decimal number.

- Last-Modified must contain the modification time of the file. See the specification for the exact format expected.

In addition, your program should track the number of bytes of file content it sends in the response, and not send more than Content-Length total bytes. The only time this is a problem is if the file is modified while it is being sent. If the file size changes from Content-Length in the course of sending the file, log an error message to standard error.

Directory Indexes (15 points)

If the requested path is a directory, and the directory does not contain an index.html file, then your program should produce and HTML-formatted list of the files in that directory.

- List files in alphabetical order (using strcoll)
- Do not include hidden files (starting with .)
- Include the last modification time and size of each file
- Each file must be a link to itself (the HTML <a> tag makes links)
- Directories must include a trailing / in both their names and links

Example output, for the directory /foo/:

```
<!doctype html>
<html>
<head>
<title>Index of /foo/</title>
</head>
<body>
<h1>Index of /foo/</h1>
<ul>
<li><a href="hamster.html">hamster.html</a>
      (9308 bytes, last modified on May 27, 2014 14:42 CST)
<li><a href="subdir/">subdir/</a>
      (directory, last modified on May 4, 2014 12:22 CST)
<li><a href="wombat.html">wombat.html</a>
      (34082 bytes, last modified on May 27, 2014 15:00 CST)
</ul>
</ul>
</body>
</html>
```

When sending such a response, you **must** send the 'Content-Type' header (with the value 'text/html'), but **cannot** send the Content-Length header (unless you build up the response in memory as a string and compute its length before sending it). You **should** send the Last-Modified header.

Cache-Friendly Retrieval (5 points extra credit)

Inspect the `If-Modified-Since` header sent by the browser, if one is present. If it specifies a time no older than the most recent of the file's `mtime` and `ctime`, then return a response of '304 Not Modified' (with no content) instead of a '200 OK' response with the file's content.

Debugging

When you run `./listen.sh /path/to/html/files`, you can point your browser at http://localhost:4080 to browse the files. If you have only done part 1, then you will need to specify a file name in the URL.

You can also use the `curl` utility to request URLs on the command line:

```
$ curl http://localhost:4080/some/file.html
```

The `-v` option to `curl` will make it print additional debugging information, including the request and the response headers.

Submitting

Submit your code as a gzipped tar archive to TRACS. To make the tar file, use the `a4-submit.tgz` target of the `Makefile`:

```
$ make a4-submit.tgz
```