Assignment 7

April 10, 2015

Overview

The Common Gateway Interface is a specification for how web servers can run other programs to produce output to be sent to the web browser. For this assignment, you will modify your forking HTTP daemon to support running CGI programs.

For full details on CGI, you can refer to the specification.

The basic idea is that the server parses the HTTP request, places relevant portions of it into environment variables, and runs the CGI script. The script outputs the HTTP response (including the status line and headers, so the script can produce non-200 responses).

Request Data

The raw path provided in an HTTP request can contain a *query string* in addition to a path. This query string provides additional information to the CGI script; it is separated from the path by a '?'. For example, the URL `http://example.com/search?q=Hello+World` would yield the rfollowing request:

```
GET /search?q=Hello+World HTTP/1.0
Host: example.com
```

In this case, the path is '/search' and the query string is 'q=Hello+World'.

You can get the query string with the `req_query` function.

In addition, `POST` requests can have a *request body*: additional data after the blank line terminating the request headers that provide a larger body of additional information. If POST data is provided, then there will be two additional headers:

Content-Type The `Content-Type` header provides the type of data being supplied. Common values are `application/x-www-form-urlencoded` (for the results of an HTML form) and `application/json`.

Content-Length The `Content-Length` header says how many bytes will be provided. Its value is a decimal number.

To generate a request, you can use `curl`:

```
$ curl -X POST -d 'user=alice&password=wombat' http://example.com/login
```

This will create the following request:

```
POST /login HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 26

user=alice&password=wombat
```

Basic CGI Support (75 points)

For this portion of the assignment, you will implement support for GET requests of CGI scripts. Use your forking HTTP server as the starting point for implementing this assignment.

A script will be considered a CGI script to be executed if it meets the following requirements:

- it is in the `cgi-bin` directory within the document root (so the URL path is `/cgi-bin/script.cgi`, for a script called `script.cgi`). To keep things simple, we will not support subdirectories within `cgi-bin`.
- it is a file (not a directory).
- it is executable (either check the `stat` mode, or use `access` to test this).

If the path begins with `/cgi-bin/`, but does not name an executable CGI script, respond with a 404 Not Found error.

For the purposes of locating the CGI script, only consider the first 2 elements of the request path. If the request's path is `/cgi-bin/frob.cgi/wombat`, then the CGI script `cgi-bin/frob.cgi` will be used. Everything else (the `/wombat`) will be saved to be passed to the CGI script in the `PATH_INFO` environment variable.

If the request is not for a CGI script, handle it as you did in Assignment 4.

If the request is for a CGI script, then execute the script (with `exec`, and appropriate calls to `fork`, `pipe`, etc. to make it work). Before executing the script, set the following environment variables:

SERVER_SOFTWARE The string `CS4350-HTTPD/0.1`

SERVER_NAME The value of the `Host` header provided by the client (`req_host`); if the request has no host, use the human-readable IP addresson which the server is listening.

SERVER_PORT The port on which the HTTP server is listening.

GATEWAY_INTERFACE The string `CGI/1.1`

SERVER_PROTOCOL The string `HTTP/1.0`

REQUEST_METHOD The HTTP method (for Part 1, this will always be `GET`)

PATH_INFO The path suffix, if one is provided. This is the `/` and everything after it at the end of the CGI script path (so `/cgi-bin/frob.cgi/wombat` has `PATH_INFO=wombat`).

QUERY_STRING The query string (see `req_query`)

REMOTE_HOST The hostname of the client that sent the request (use `getnameinfo` to get this).

REMOTE_ADDR The human-readable IP address that sent the request (use `getnameinfo` with `NI_NUMERICHOST` to get this).

PATH Reset this to just contain `/bin:/usr/bin:/usr/local/bin`.

Send everything the script produces on its standard output to the client. If the script is successfully launched, do not send any other output, as the script is responsible for producing the HTTP response.

The script's standard error should be untouched (inherited from the parent process), so that error messages can be seen.

The script needs no standard input. There are two ways this can be done:

- open `/dev/null` and wire it up to the script' standard input
- create a pipe (`pipe()`) and close the writing side of it, and wire up the reading side to standard input. This will be closer to what you need for Part 2.

If the script cannot be successfully launched, then produce a 500 Internal Server Error response. If you `fork()` + `exec()` to launch the server, then the child process can produce the error response if the setup or the `exec` call fail.

Before `exec()`-ing the script, your program should also close unnecessary file descriptors, such as the client socket. You can alternatively use `fcntl` with an operation of `F_SETFD` and arg of `FD_CLOEXEC` any time you create a file descriptor that the CGI script should not see.

When the CGI script is running, you'll have the following processes in play:

```
forking-httpd (master)
---- forking-httpd child (request handler)
     ---- CGI script (the child forks, then execs the script)
```

For this level, it is also possible to have the request handling process directly exec the CGI script without forking first. However, that will not work for the later portions of this assignment.

POST Requests (15 points)

Modify your server to support HTTP `POST` requests. These requests will have a method of `POST` (there is an example of such a request earlier in this document).

If the request is of type `POST`, and the `path` points to a CGI script (possibly with a suffix for `PATH_INFO`), then run the CGI script as in Part 1, but in addition:

1. Set the `CONTENT_TYPE` environment variable to the value of the `Content-Type` request header.
2. Set the `CONTENT_LENGTH` environment variable to the value of the `Content-Length` request header.
3. Write all additional data received from the client to the CGI script's standard input. Note that this may include data already read in the process of parsing the request; the `req_parse` function will return the bytes that it used, and the *remaining* bytes are to be sent to the CGI script, along with any other bytes that were not read into your input buffer during parsing.

To write the data, you will want to create a pipe (`pipe()`). The reading side of the pipe will be wired to the script's standard input (`dup2`), and the parent process (the request handling process in your forking HTTP server) will write the request body to the writing side of the pipe.

Process Management (10 points)

Add 2 new features to your program:

1. Wait for the CGI script process. If it exits with a non-zero exit status, print an error message to standard error indicating the script that failed and its exit code (if it exited) or signal (if it died due to a signal).

2. Impose a time limit on CGI processes. If the CGI script runs for more than 30 seconds, kill it with `SIGTERM`. If it is still running after 5 more seconds, kill it with `SIGKILL`.

Submission

Submit a .tgz file containing your code. You can create this file with `make a7-submit.tgz`.