Specification for Lab assignment 2

Programming in the UNIX environment

Created by Stefan Axelsson
Modified by Florian Westphal
Department of Computer Science and Engineering
Blekinge Institute of Technology

2015-09-10

1 Your task

Your task is to develop a web server in C. The program shall implement the HTTP 1.0 protocol specification with a few exceptions. Apache (`http://www.apache.org`) is currently by far the most popular1 web server and also one of the larger ones as it consists of approximately 70000 lines of code2. While Apache is rich of options, the THTTPD (`http://www.acme.com/software/thttpd/`) is developed to be "simple, small, portable, fast, and secure". Moreover, THTTPD is much smaller than Apache — it consists of approximately 7000 lines of code and has, of course, fewer features. A more modern small, fast webserver is lighttpd (`http://www.lighttpd.net`) These servers might be useful to look at when performing the lab, though of course you have to be wary of actual copying of code, that's not allowed. There are many more webservers listed at wikipedia: `http://en.wikipedia.org/wiki/Comparison_of_web_servers`.

2 Requirements

The server must be written in C, compile using GCC and run in Ubuntu Linux latest release (or the release that is installed in the security lab). There shall be no software limitation in the number of served clients or concurrent requests. Additionally, the server has to be implemented in a **secure** fashion. Especially, pay attention to secure string handling.

2.1 Project Structure

The basic folder structure for your project has to look as follows (you may add folders, if necessary):

```
project
    |----- webserver
    |           |----- include/
    |           |----- objects/
    |           |----- src/
    |           +----- Makefile
    +----- www
```

The `webserver` folder shall contain all your source files. Header files have to be placed into the `include` folder, source code files have to be in the `src` folder and object files, created during the project build, shall be placed into the `objects`

folder. If you see it fit, you can further divide your header or source code files into separate folders within their respective folders.
The `www` folder is supposed to be the root directory of your web server and shall contain at least one html file, called `index.html`. By default, your web server shall use this directory. Please keep in mind that it must be possible to execute your program on a different computer as well.
Last but not least, it must be possible to build your web server by issuing the `make` command within the `webserver` folder, which will place the final executable in the `webserver` folder.

2.2 HTTP 1.0

HTTP Methods The server shall implement both the simple and full versions of the following HTTP 1.0 methods:

- HEAD
- GET

HTTP requests for all other types of methods should result in the HTTP status code 501.

Status Codes The following status codes should be implemented. All status responses should be properly HTML 4.0 formatted.

Code	**Reason Phrase**
200	OK
400	Bad Request
403	Forbidden
404	Not Found
500	Internal Server Error
501	Not Implemented

2.3 HTML 4.0

All HTML formatted output from the server should follow the HTML 4.0 specification. Validation tools are available on the Internet, e.g. The W3C Markup Validation Service at `http://validator.w3.org/` or WDG HTML Validator at `http://www.htmlhelp.com/tools/validator/`. HTML Tidy[1] might be a useful tool to check your server output.

2.4 Concurrent Request Handling

The server should be able to efficiently handle requests in parallel, i.e., if one request takes (relatively speaking) a lot of time to process it should not block requests coming later, that could proceed, from beginning to being processed. See Section 5.1 at the end of the document for various strategies. In order to have the possibility of achieving top marks for the course you have to implement at least two, but preferably all, different strategies (cf. Section 4).

2.5 Process Management

The server may not create zombie processes, regardless of the server load (see the manual page for `ps`). See the course literature for information and tips on

[1]`http://tidy.sourceforge.net/`

how to deal with zombie processes. The benchmark tools siege[2] or ab[3] are usable when testing if the process management works correctly. Any dead processes will be visible using `ps`.

2.6 Command Line Options

The server must support at least the following command line options:

-h Print help text.
-p port Listen to port number *port*.
-d Run as a daemon instead of as a normal program. **(if implemented - cf. Section 2.10)**
-l logfile Log to logfile. If this option is not specified, logging will be output to syslog, which is the default. **(if implemented - cf. Section 2.11)**
-s [fork | thread | prefork | mux] Select request handling method. **(if implemented - cf. Section 2.4)**

Any of these options overrides the default value, which may have been specified by your implementation or the configuration file. If any of these options is not supported by your web server, the program should exit and return 3 as its return value. Additionally, it should print the usage/help message.

2.7 URL Validation

The server should validate URLs to have a rudimentary protection against hacking or DoS- attacks. Examples of URLs that should not be served are:

```
http://localhost:80/../../../etc/passwd
http://localhost:80/<very_long_string>.html
```

The suggested way of validating is by using the library function realpath (see the realpath manual, section 3).

2.8 Client Compatibility

The server should be able to serve Internet Explorer, Firefox on Windows, and Konqueror, Firefox, Chrome, and Lynx on Linux.

2.9 Configuration File

The server shall read a configuration file called .lab3-config, which shall be user-editable (e.g. in ASCII text format). The following configurations should be specified:

- full path of the server's document root directory; This should not be the same as the server's execution directory
- default listen port
- default request handling method **(if implemented - cf. Section 2.4)**

2.10 Run as Daemon

It shall be possible to run your server as daemon, as well as as normal program. After starting as daemon, the server should print its process ID.

2.11 Logging

The server should log standard activity and errors in CLF (Common Logfile Format), a log format used by several web servers and log analysers [4]. Logging

[2]`https://www.joedog.org/siege-home/`
[3]`https://httpd.apache.org/docs/2.2/programs/ab.html`
[4]`http://httpd.apache.org/docs/1.3/logs.html`

shall be possible to output both to syslog and to a file (however, you may choose to output to separate files, e.g. <filename>.log and <filename>.err). When output is made to syslog, which should be the default, the server must perform string length checks to ensure that possible overflow attacks do not succeed. See Section 2.6 for more information on how to handle logging output. IP addresses should be printed in IPv4 format.
To check that your logs conform to CLF, an analysing tool is recommended. A list of suggested tools is available in the References section. If you have no preferences, the Webalizer tool[5] is recommended.

2.12 Jail

The web server should correctly use the *chroot* system call (similar to *jail(2)* in BSD) to imprison the process and its descendants. Read the jail manual page for more information on how to implement this requirement.

3 Presentation

The assignment should be presented in person with both group members in attendance on a lab slot. You will be required to show the code, let me run it and be prepared to discuss your work.

4 Grading

You should demonstrate a clear understanding of the problem and detail your strategy of how to solve it, including drawbacks and advantages of the solution chosen. Your code should be clear, concise and to the point.

You are allowed to discuss problems and solutions with other groups, but the code that you write must be your own and cannot be copied, gleaned, downloaded from the internet etc. (You may of course copy idioms and snippets. However, you must be able to clearly describe each part of your implementation.

The following list details, which of the requirements, stated in Section 2, have to be implemented to get a certain grade.

Grade - D

- All requirements from Section 2.1 to Section 2.8 (including both sections)
- Note that only one strategy for concurrent request handling is required

Grade - C

- All requirements for Grade D
- Implement server configuration using a configuration file (cf. Section 2.9)
- Implement possibility to execute sever as daemon (cf. Section 2.10)

Grade - B

- All requirements for Grade C
- Implement logging (cf. Section 2.11)
- Implement process jail (cf. Section 2.12)

[5] `http://www.mrunix.net/webalizer/`

Grade - A

- All requirements for Grade B
- Implement at least one additional strategy for concurrent request handling (cf. Section 2.4 and 5.1)

Note: Even if your implementation fulfils the requirements for the grade you were targeting, you might get a lower grade (lower by one), if your code is of bad quality. For this reason, there are no requirements for grade E. You can only get an E, if your implementation fulfils the requirements for grade D and your code is of bad quality. In this way, it is impossible to fail the assignment just because of poor code quality.

5 Additional Information

5.1 Strategies for handling multiple requests

Your webserver has to implement at least one of these. If you aim for the highest mark on the course you are required to implement two or more strategies.

The strategies are: fork/exec (*fork*), threads (*thread*), preforking/thread pool (*prefork*), or I/O multiplexing/single process (*mux*). For a description of the various strategies, read the following historical account. In almost all of these the question of how to communicate between the controlling process and the worker processes, several ways are possible; using pipes, a FIFO, mmap, or another technique, research the available techniques and argue and document your choice.

select() / poll() / epoll() / kqueue() are Unix system calls used to multiplex between a bunch of file descriptors. To understand why this is important we have to go back through the history of web servers.

The basic operation of a web server is to accept a request and send back a response. The first web servers were probably written to do exactly that. Their users no doubt noticed very quickly that while the server was sending a response to someone else, they couldn't get their own requests serviced. There would have been long annoying pauses.

The second generation of web servers addressed this problem by forking off a child process for each request. This is very straightforward to do under Unix, only a few extra lines of code. CERN and NCSA 1.3 are examples for this type of server. Unfortunately, forking a process is a fairly expensive operation, so performance of this type of server is still pretty poor. The long random pauses are gone, but instead every request has a short constant pause at startup. Because of this, the server can't handle a high rate of connections.

A slight variant of this type of server uses lightweight processesör threads"instead of full- blown UNIX processes. This is better, but there is no standard LWP/threads interface so this approach is inherently non-portable. Examples of these servers: MDMA and phttpd, both of which run only under Solaris 2.x.

The third generation of servers is called pre-forking". Instead of starting a new subprocess for each request, they have a pool of subprocesses or threads that they keep around and re-use. NCSA 1.4, Apache, and Netscape Netsite are examples of this type. Performance of these servers is excellent; they can handle from two to ten times as many connections per second as the forking servers. One problem, however, is that implementing this simple-to-state idea turns out to be fairly complicated and non-portable. The method used by NCSA involves transferring a file descriptor from the parent process to an already-existing child process; you can hardly use the same code on any two different OS's, and some OS's (e.g. Linux) don't support it at all. Apache uses a different method, with all the child processes doing their own round-robin work queue via lock files, which brings in issues of portability/speed/deadlock. Besides, you still have multiple processes hanging around using up memory and context- switch CPU cycles. This brings us to...

The fourth generation. One process only. No non-portable threads/LWPs. Sends multiple files concurrently using non-blocking I/O, calling select()/poll()/kqueue() to tell which ones are ready for more data. Speed is excellent. Memory use is excellent. Portability is excellent. Examples of this generation: Spinner, Open Market, and thttpd. Perhaps Apache will switch to this method at some point.

5.2 Tools

Some tools that may be useful during development are:

- sendfile - Not really a tool but a Linux system call to send a file to a socket.
- tcpdump - Network sniffer without GUI.
- Ethereal - GUI network protocol analyser.
- Hammerhead – A web testing tool, installed in the security lab.
- netcat - Networking utility which reads and writes data across network connections, using the TCP/IP protocol.
- wget - Utility for non-interactive download of files from the Web.

5.3 References

- RFC 1945 — Hypertext Transfer Protocol – HTTP/1.0
- RFC 1521 — MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies (RFCs are available at `http://www.ietf.org/rfc.html` `http://www.rfc-editor.org/`)
- CLF (Common Logfile Format): `http://httpd.apache.org/docs/1.3/logs.html`
- A list of log analyzers: `http://www.mela.de/Unix/log.html`
- The Webalizer, "a fast, free web server log file analysis program": `http://www.mrunix.net/webalizer/`
- The Apache HTTP Server Project: `http://httpd.apache.org/`
- thttpd, tiny/turbo/throttling HTTP server: `http://www.acme.com/software/thttpd/`
- Hammerhead 2, a web testing tool: `http://sourceforge.net/projects/hammerhead/`
- Nessus, a remote security scanner: `http://www.nessus.org`

Good luck, and may the source be with you.

Typeset by LaTeX.