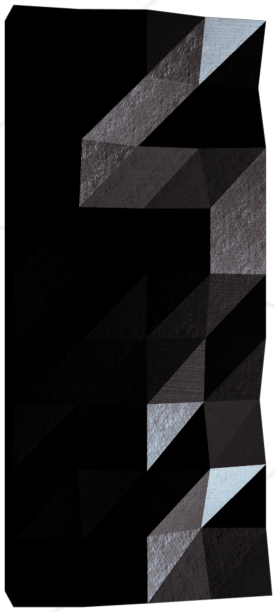

Webserv

This is when you finally understand why URLs start with HTTP

Summary:
This project is about writing your own HTTP server.
You will be able to test it with an actual browser.
HTTP is one of the most widely used protocols on the internet.
Understanding its intricacies will be useful, even if you won't be working on a website.

Version: 21.4

Contents

Chapter I

Introduction

The **Hypertext Transfer Protocol** (HTTP) is an application protocol for distributed, collaborative, hypermedia information systems.

HTTP is the foundation of data communication for the World Wide Web, where hypertext documents include hyperlinks to other resources that the user can easily access. For example, by clicking a mouse button or tapping the screen on a web browser.

HTTP was developed to support hypertext functionality and the growth of the World Wide Web.

The primary function of a web server is to store, process, and deliver web pages to clients. Client-server communication occurs through the Hypertext Transfer Protocol (HTTP).

Pages delivered are most frequently HTML documents, which may include images, style sheets, and scripts in addition to the text content.

Multiple web servers may be used for a high-traffic website.

A user agent, commonly a web browser or web crawler, initiates communication by requesting a specific resource using HTTP, and the server responds with the content of that resource or an error message if unable to do so. The resource is typically a real file on the server's secondary storage, but this is not always the case and depends on how the webserver is implemented.

Although its primary function is to serve content, HTTP also enables clients to send data. This feature is used for submitting web forms, including the uploading of files.

Chapter II

General rules

- Your program must not crash under any circumstances (even if it runs out of memory) or terminate unexpectedly.
  If this occurs, your project will be considered non-functional and your grade will be `0`.
- You must submit a `Makefile` that compiles your source files. It must not perform unnecessary relinking.
- Your `Makefile` must at least contain the rules:
  `$(NAME)`, `all`, `clean`, `fclean` and `re`.
- Compile your code with `c++` and the flags `-Wall -Wextra -Werror`
- Your code must comply with the **C++ 98 standard** and should still compile when adding the flag `-std=c++98`.
- Make sure to leverage as many `C++` features as possible (e.g., choose `<cstring>` over `<string.h>`). You are allowed to use `C` functions, but always prefer their `C++` versions if possible.
- Any external library and `Boost` libraries are forbidden.

Chapter III

Mandatory part

Program name	webserv
Turn in files	Makefile, *.{h, hpp}, *.cpp, *.tpp, *.ipp, configuration files
Makefile	NAME, all, clean, fclean, re
Arguments	[A configuration file]
External functs.	All functionality must be implemented in C++ 98. execve, pipe, strerror, gai_strerror, errno, dup, dup2, fork, socketpair, htons, htonl, ntohs, ntohl, select, poll, epoll (epoll_create, epoll_ctl, epoll_wait), kqueue (kqueue, kevent), socket, accept, listen, send, recv, chdir, bind, connect, getaddrinfo, freeaddrinfo, setsockopt, getsockname, getprotobyname, fcntl, close, read, write, waitpid, kill, signal, access, stat, open, opendir, readdir and closedir.
Libft authorized	n/a
Description	An HTTP server in C++ 98

You must write an HTTP server in C++ 98.

Your executable should be executed as follows:

```
./webserv [configuration file]
```

Even though poll() is mentioned in the subject and grading criteria, you can use any equivalent function such as select(), kqueue(), or epoll().

Please read the RFC and perform tests with telnet and NGINX before starting this project.
Although you are not required to implement the entire RFC, reading it will help you develop the required features.

III.1 Requirements

- Your program must take a configuration file as an argument, or use a default path.
- You cannot `execve` another web server.
- Your server must remain non-blocking at all times and properly handle client disconnections when necessary.
- It must be non-blocking and use only **1** `poll()` (or equivalent) for all the I/O operations between the client and the server (listen included).
- `poll()` (or equivalent) must monitor both reading and writing simultaneously.
- You must never do a read or a write operation without going through `poll()` (or equivalent).
- Checking the value of `errno` is strictly forbidden after performing a read or write operation.
- You are not required to use `poll()` (or equivalent) before reading your configuration file.

Because you have to use non-blocking file descriptors, it is possible to use read/recv or write/send functions with no poll() (or equivalent), and your server wouldn't be blocking.
But it would consume more system resources.
Thus, if you attempt to read/recv or write/send on any file descriptor without using poll() (or equivalent), your grade will be 0.

- You can use every macro and define like `FD_SET`, `FD_CLR`, `FD_ISSET` and, `FD_ZERO` (understanding what they do and how they work is very useful).
- A request to your server should never hang indefinitely.
- Your server must be compatible with standard **web browsers** of your choice.
- We will consider that NGINX is HTTP 1.1 compliant and may be used to compare headers and answer behaviors.
- Your HTTP response status codes must be accurate.
- Your server must have **default error pages** if none are provided.
- You can't use fork for anything other than CGI (like PHP, or Python, and so forth).
- You must be able to **serve a fully static website**.
- Clients must be able to **upload files**.
- You need at least the `GET`, `POST`, and `DELETE` methods.

- Stress test your server to ensure it remains available at all times.
- Your server must be able to listen to multiple ports (see *Configuration file*).

III.2 For MacOS only

Since macOS handles write() differently from other Unix-based OSes, you are allowed to use fcntl().
You must use file descriptors in non-blocking mode to achieve behavior similar to that of other Unix OSes.

However, you are allowed to use fcntl() only with the following flags:
F_SETFL, O_NONBLOCK and, FD_CLOEXEC.
Any other flag is forbidden.

III.3 Configuration file

You can take inspiration from the 'server' section of the NGINX configuration file.

In the configuration file, you should be able to:

- Choose the port and host of each 'server'.
- Set up the `server_names` or not.
- The first server for a `host:port` will be the default for this `host:port` (meaning it will respond to all requests that do not belong to another server).
- Set up default error pages.
- Set the maximum allowed size for client request bodies.
- Set up routes with one or multiple of the following rules/configurations (routes won't be using regexp):
  - Define a list of accepted HTTP methods for the route.
  - Define an HTTP redirect.
  - Define a directory or file where the requested file should be located (e.g., if url `/kapouet` is rooted to `/tmp/www`, url `/kapouet/pouic/toto/pouet` is `/tmp/www/pouic/toto/pouet`).
  - Enable or disable directory listing.

- Set a default file to serve when the request is for a directory.
- Execute CGI based on certain file extension (for example .php).
- Make it work with POST and GET methods.
- Allow the route to accept uploaded files and configure where they should be saved.
  - Do you wonder what a CGI is?
  - Because you won't call the CGI directly, use the full path as `PATH_INFO`.
  - Just remember that, for chunked requests, your server needs to unchunk them, the CGI will expect `EOF` as the end of the body.
  - The same applies to the output of the CGI. If no `content_length` is returned from the CGI, `EOF` will mark the end of the returned data.
  - Your program should call the CGI with the file requested as the first argument.
  - The CGI should be run in the correct directory for relative path file access.
  - Your server should support at least one CGI (php-CGI, Python, and so forth).

You must provide configuration files and default files to test and demonstrate that every feature works during the evaluation.

If you have a question about a specific behavior, you should compare your program's behavior with NGINX's.
For example, check how the server_name works.
We have provided a small tester. Using it is not mandatory if everything works fine with your browser and tests, but it can help you find and fix bugs.

Resilience is key. Your server must remain operational at all times.

Do not test with only one program. Write your tests in a more suitable language, such as Python or Golang, among others, even in C or C++ if you prefer.

Chapter IV

Bonus part

Here are some additional features you can implement:

- Support cookies and session management (provide simple examples).
- Handle multiple CGI.

The bonus part will only be assessed if the mandatory part is fully completed without issues. If you fail to meet all the mandatory requirements, your bonus part will not be evaluated.

Chapter V

Submission and peer-evaluation

submit your assignment in your `Git` repository as usual. Only the content of your repository will be evaluated during the defense. Be sure to double-check the names of your files to ensure they are correct.

16D85ACC441674FBA2DF65190663F42A3832CEA21E024516795E1223BBA77916734D1
26120A16827E1B16612137E59ECD492E46EAB67D109B142D49054A7C281404901890F
619D682524F5