Webserv

This is when you finally understand why a URL starts with HTTP

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

Version: 20.8

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 a mouse click or by tapping the screen in a web browser.

HTTP was developed to facilitate hypertext and the World Wide Web.

The primary function of a web server is to store, process, and deliver web pages to clients. The communication between client and server takes place using 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 necessarily the case and depends on how the webserver is implemented.

While the primary function is to serve content, full implementation of HTTP also includes ways of receiving content from clients. This feature is used for submitting web forms, including the uploading of files.

Chapter II

General rules

- Your program should not crash in any circumstances (even when it runs out of memory), and should not quit unexpectedly.
  If it happens, your project will be considered non-functional and your grade will be `0`.
- You have to turn in a `Makefile` which will compile your source files. It must not relink.
- 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**. Then, it should still compile if you add the flag `-std=c++98`.
- Try to always develop using the most `C++` features you can (for example, 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.	Everything in C++ 98. execve, dup, dup2, 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, opendir, readdir and closedir.
Libft authorized	n/a
Description	A HTTP server in C++ 98

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

Your executable will be run as follows:

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

Even if poll() is mentionned in the subject and the evaluation scale, you can use any equivalent such as select(), kqueue(), or epoll().

Please read the RFC and do some tests with telnet and NGINX before starting this project.
Even if you don't have to implement all the RFC, reading it will help you develop the required features.

III.1 Requirements

- Your program has to take a configuration file as argument, or use a default path.
- You can't `execve` another web server.
- Your server must never block and the client can be bounced properly if 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 check read and write at the same time.
- 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 a read or a write operation.
- You don't need 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 try to read/recv or write/send in 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`, `FD_ZERO` (understanding what and how they do it is very useful).
- A request to your server should never hang forever.
- Your server must be compatible with the **web browser** 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.
- You server must have **default error pages** if none are provided.
- You can't use fork for something else 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 `GET`, `POST`, and `DELETE` methods.
- Stress tests your server. It must stay available at all cost.
- Your server must be able to listen to multiple ports (see *Configuration file*).

III.2 For MacOS only

Since MacOS doesn't implement write() the same way as other Unix OSes, you are allowed to use fcntl().
You must use file descriptors in non-blocking mode in order to get a behavior similar to the one of other Unix OSes.

However, you are allowed to use fcntl() only as follows:
fcntl(fd, F_SETFL, O_NONBLOCK, FD_CLOEXEC);
Any other flag is forbidden.

III.3 Configuration file

You can get some inspiration from the 'server' part of NGINX configuration file.

In the configuration file, you should be able to:

- Choose the port and host of each 'server'.
- Setup the `server_names` or not.
- The first server for a `host:port` will be the default for this `host:port` (that means it will answer to all the requests that don't belong to an other server).
- Setup default error pages.
- Limit client body size.
- Setup routes with one or multiple of the following rules/configuration (routes wont be using regexp):
  - Define a list of accepted HTTP methods for the route.
  - Define a HTTP redirection.
  - Define a directory or a file from where the file should be searched (for example, if url `/kapouet` is rooted to `/tmp/www`, url `/kapouet/pouic/toto/pouet` is `/tmp/www/pouic/toto/pouet`).
  - Turn on or off directory listing.

- Set a default file to answer if the request is a directory.
- Execute CGI based on certain file extension (for example .php).
- Make it work with POST and GET methods.
- Make the route able 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 request, your server needs to unchunk it, the CGI will expect `EOF` as end of the body.
  - Same things for 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 first argument.
  - The CGI should be run in the correct directory for relative path file access.
  - Your server should work with one CGI (php-CGI, Python, and so forth).

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

If you've got a question about one behavior, you should compare your program behavior with NGINX's.
For example, check how does server_name work.
We've shared with you a small tester. It's not mandatory to pass it if everything works fine with your browser and tests, but it can help you hunt some bugs.

The important thing is resilience. Your server should never die.

Do not test with only one program. Write your tests with a more convenient language such as Python or Golang, and so forth. Even in C or C++ if you want to.

Chapter IV

Bonus part

Here are the extra features you can add:

- Support cookies and session management (prepare quick examples).
- Handle multiple CGI.

The bonus part will only be assessed if the mandatory part is PERFECT. Perfect means the mandatory part has been integrally done and works without malfunctioning. If you have not passed ALL the mandatory requirements, your bonus part will not be evaluated at all.

Chapter V

Submission and peer-evaluation

Turn in your assignment in your `Git` repository as usual. Only the work inside your repository will be evaluated during the defense. Don't hesitate to double check the names of your files to ensure they are correct.