HTTPd-2 — Subject

version **#84fae3c4944d6c55849e55a61bd313ee04600cde**

Assistants C/Unix 2024 <assistants@tickets.assistants.epita.fr>

Copyright

This document is for internal use at EPITA (website) only.

Contents

*https://intra.forge.epita.fr

Obligations

*Obligations are **fundamental** rules shared by all subjects. They are non-negotiable and to not apply them means to face sanctions. Therefore, do not hesitate to ask for explanations if you do not understand one of these rules.*

Obligation #0: Cheating, as well as sharing source code, tests, test tools or coding-style correction tools is **strictly forbidden** and penalized by not being graded, being flagged as a cheater and reported to the academic staff.

Obligation #1: If you do not submit your work before the deadline, it will not be graded.

Obligation #2: Your submission repository must be **clean**. Except for special cases, which (if any) are **explicitely** mentioned in this document, an *unclean* repository may contain:

- binary files;[1]
- files with inappropriate privileges;
- forbidden files: `*~`, `*.swp`, `*.o`, `*.a`, `*.so`, `*.class`, `*.log`, `*.core`, etc.;
- a file tree that does not follow our specifications.

Obligation #3: All your files must be encoded in ASCII or UTF-8 without BOM.

Obligation #4: When examples demonstrate the use of an output format, you must follow it scrupulously.

Obligation #5: The coding-style needs to be respected at all times.

Obligation #6: Global variables are forbidden, unless they are **explicitly** authorized.

Obligation #7: Anything that is not **explicitly** allowed is **disallowed**.

Advice

- ▷ Read the *whole* subject.
- ▷ If the slightest project-related problem arise, you can get in touch with the assistants.

  Post to the dedicated **Discourse category** (with the appropriate **tag**) for questions about this document, or send a **ticket** to **<assistants@tickets.assistants.epita.fr>** otherwise.
- ▷ In examples, `42sh$` is our prompt: use it as a reference point.
- ▷ Do **not** wait for the last minute to start your project!

[1] If an executable file is required, please provide its sources **only**. We will compile it ourselves.

1 Introduction

1.1 Before you start

At this point, it is assumed that you have a relatively functional server that is able to handle a single request at a time. It is **strongly suggested** that you work out any remaining bugs before moving any further. Your server must be able to handle basic GET or HEAD requests flawlessly.

1.2 The Goal

The goal of this step is to make your server handle multiple request at the same time. This is an essential feature for a server because you might have multiple clients connecting to your server simultaneously.

Until now, you have been using blocking IO operations to handle your connections. This means that when your server is waiting for a connection to send a request, it is not able to handle other connections.

In order to solve this problem, you will have to use non-blocking IO operations along with `epoll(7)` to avoid delays that might happen for several reasons, like perhaps a slow connection. Once this is done, you will be able to take it even further and multithread the server to handle a significant load.

2 HTTPd

2.1 Architecture

2.1.1 Exercise

File Tree

```
httpd/
├─ Makefile  (to submit)
├─ src/
│  ├─ Makefile  (to submit)
│  ├─ config/
│  │  ├─ *  (to submit)
│  │  ├─ Makefile  (to submit)
│  │  └─ tests/
│  │     ├─ *  (to submit)
│  │     └─ config_test.c  (to submit)
│  ├─ flags.mk  (to submit)
│  ├─ libs.mk  (to submit)
│  ├─ main.c  (to submit)
│  ├─ utils/
│  │  ├─ *  (to submit)
│  │  ├─ Makefile  (to submit)
│  │  └─ tests/
│  │     └─ *  (to submit)
│  └─ {module_name}/
│     ├─ *  (to submit)
│     ├─ Makefile  (to submit)
│     └─ tests/
│        └─ *  (to submit)
└─ tests/
   └─ *  (to submit)
```

Compilation : Your code must compile with the following flags

- -std=c99 -Werror -Wall -Wextra -Wvla

Authorized functions : You are only allowed to use the following functions

- getopt
- getaddrinfo
- freeaddrinfo

- gai_strerror
- fcntl
- open
- close
- dup
- dup2
- fork
- getpid
- waitpid
- _exit
- read
- write
- lseek
- sendfile
- isatty
- inet_ntop

Authorized headers : You are only allowed to use the functions defined in the following headers

- assert.h
- ctype.h
- errno.h
- fnmatch.h
- limits.h
- signal.h
- stdarg.h
- stdbool.h
- stddef.h
- stdio.h
- stdlib.h
- string.h
- time.h
- pthread.h
- err.h
- sys/epoll.h

- sys/eventfd.h
- sys/socket.h
- sys/types.h
- sys/stat.h

2.2 Suggested Modules

During this step of HTTPd, we will not provide you with any mandatory modules. To help you out however, we will suggest a module that could help you this step.

2.2.1 Connections

Since in this step the I/O is non-blocking, this implies that a client may stop and resume sending data to your server. Until now since you handled each request at a time, clients needed to wait before having their requests processed and reads would block until everything was received. Now however, if a client stops sending data, this should not block your server and resume processing another clients' request. You must therefore create a structure that keeps track of the state of the request.

Each client should hold buffers for incoming and outgoing data. Using non-blocking IO means that you will have to handle partial reads and writes. You will better understand as you read the following sections.

3 Core Features

3.1 Multiple Clients

As you can guess, a server able to handle only one client at a time is very inefficient. For this step you will have to handle multiple clients using `epoll`.

At the end of the project, your server **has** to be able to handle at least 10 000 simultaneous connections. You can find some useful advice on the C10k website[1].

Tips

The 10 000 simultaneous connections will be tested by requesting one single resource, and with the **Connection** header set to `close`.

Be careful!

You are not allowed to use `fork(2)` for this feature.

[1] http://www.kegel.com/c10k.html

3.1.1 Non-blocking I/O

A big bottleneck that can be encountered when trying to scale a server is the use of blocking I/O. Blocking I/O is the default behavior of functions such as `read(2)` or `write(2)`. This means that when you call one of these functions and there is no data available to read or write, the function will block the program until data is available. So two clients won't be able to communicate with the server simultaneously.

Be careful!

Non-blocking I/O means that the function will return if incoming data is delayed. Make sure to check the possible return values of `read(2)` and `write(2)` in the `Error` section of the manpage to properly handle this case.

3.1.2 Epoll

To handle multiple clients, we ask of you to use `epoll(7)`. `epoll` is a family of system calls present on Linux that allow efficient I/O operations through an event notification mechanism.

System calls such as `read(2)` or `write(2)` are *blocking*: they stop the current thread while waiting for data to be available for reading or writing.

This is inefficient when handling multiple clients, as it's possible to be blocked trying to communicate with a busy client, while another is ready and waiting for our server.

`epoll` solves this problem for us. It will take the role of monitoring communication with the client and notifying us when one is ready to communicate.

For this purpose, `epoll` must be told which clients (fds) to monitor and on which type of communication (read/write).

The role of `epoll` is to tell your program when data is available for reading and writing, and when to use the file descriptor of your clients. Remember to use non-blocking file descriptors when you use `epoll`.

This is because `epoll` is *edge-triggered*: it triggers events only when changes occur on the monitored file descriptors. As a consequence, `epoll` system calls are able to reach a time complexity that is constant, and is the key to reaching thousands of simultaneous connections.

Tips

We advise you to read **very** carefully the following man pages: - `epoll(7)` - `fcntl(2)` with **O_NONBLOCK**

Tips

We also encourage you to look at the *Example for suggested usage* in the `epoll(7)` manpage.

3.1.3 Simultaneous Connections

You need to keep in mind that using non-blocking I/O and `epoll` means that you might need to handle different bits of data coming from different clients at the same time.

Let's take an example. Imagine that you have `Client A` and `Client B` connected to your server. `Client A` has a very slow connection.

```
Server <-- Client A -- "Hello, OP"                          // read exits since Client A is slow
Server <-- Client A -- "IChat is"                           // read exits since Client A is slow
Server <-- Client B -- "HTTPd is the year's best project" // end of the message
Server <-- Client A -- " awesome!"                          // end of the message
```

At the end the server should have received the following messages from each of the clients:

- Client A: "Hello, OPIChat is awesome!"
- Client B: "HTTPd is the year's best project"

Tips

Make sure to check errno when using non-blocking I/O. It will help you know when to close a connection and when to keep waiting for incoming data. Make sure to read the man pages for the non-blocking I/O functions you might use.

4 Bonus Features

4.1 Threads

Epoll is really efficient to handle multiple clients on a single thread by taking away the hassle of blocking I/O, but these operations are not the only bottleneck we face while scaling our server.

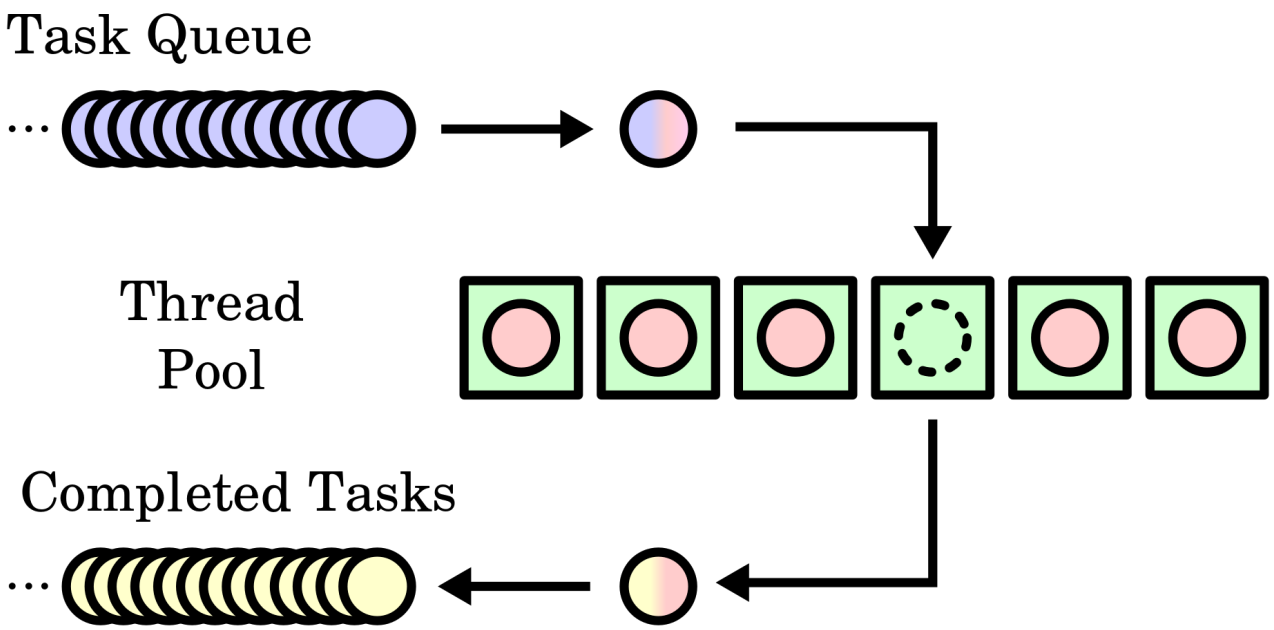
The problem is that the processing of the requests is done synchronously. In the case of a very high load, the server will now be blocked by the processing time of the requests.

To solve this problem, we will have to multithread our server in order to process multiple requests at the same time.

4.1.1 Thread Pool Pattern

The solution of our problem is not that obvious at first glance. To properly solve it, we will use a very common concurrency pattern called a thread pool.

The idea is to create a pool of worker threads that will be able to process requests in parallel. The main thread, also called the master thread, will be responsible for accepting new connections **using epoll**, and will then dispatch the requests to an **available** worker thread in the pool.



Tips

In C you will use `pthread(7)` to create your thread pool.

A queue **should** be used to store the requests that are waiting to be processed. When the main thread receives a new request, it will push it to the queue. Each thread in the pool will be responsible for popping a request from the queue and processing it. Since multiple threads will be accessing the same queue at a given time, we will have to implement a thread-safe queue to not run into race conditions.

Tips

You might want to check `pthread_mutex_init(3)` and `pthread_mutex_lock(3)` when using shared data.

Be careful!

When doing multithreaded programming, you **must** avoid race conditions at all costs. Thus, you will need to update any non-readonly data structure and other tools such as your logger to make them thread safe.

Be careful!

You are not allowed to use `fork(2)` for this feature.

4.1.2 Suggested modules

Thread Safe Queues

You may to implement a thread safe queue. This queue will be used to dispatch incoming epoll events to the right thread. This will be better illustrated below.

5 Testing

Step 1 was not about testing concurrency. It was about testing the functionality of the server. We will now test new aspects of the server.

5.1 siege

`siege(1)` is a load tester for HTTP and FTP. It can be used to benchmark an application. Options like `-c` will let you configure the number of users trying to access a resource at the same time. Then `-r` will let you specify the number of times it will try to access it. Again, you can find this information and even more on the manpage of `siege(1)`. You will need to use it to test the number of simultaneous connections your server is able to handle.

Be careful!

We will not use `siege(1)` to test the performance of the server. Do not panic if your server crashes when you use it.

5.2 Python Socket

socket API is a Python module that provides a lower level interface in order to communicate with the network. It is a wrapper around the C socket library. You should have already used it for step 1 to mainly test error cases. You can use it to simulate more advanced scenarios like a slow client or a client that disconnects before the server has finished sending the response.

I must not fear. Fear is the mind-killer.