Upload your project to GitHub and send me the URL in an email with the subject "PR lab 1" (artiom.balan@isa.utm.md). If you will be presenting later than the deadline, include the last commit hash so I can verify the time of the commit.

Include a short report in the repository (markdown or a link to a Google doc). The report will be like a demo but with screenshots, so just include screenshots of everything you would need to show in the demo, each described with a short sentence: the commands you run, the outputs you see (browser, terminal). The report should prove that you satisfied all the requirements of the lab.

You will present to me during seminars, defend your code, and answer a few theoretical questions. The answers to the questions are either true/false or a few words. You need to be able to explain your answer.

Lab grading formula:

1. Working program - 7 points
2. Answered 2 questions (out of 3) - 2 points
3. Code understanding - 1 point

Docker

You must use Docker Compose for all your laboratory works so that I can run them on my computer.

The official Docker 101 tutorial is a good resource.

Lab 2: Concurrent HTTP server

The questions will be based on:

- A good introduction to concurrency: https://web.mit.edu/6.102/www/sp25/classes/14-concurrency/
- Section 4.3 from this article
- The definitions from this glossary
- (Optional) *The Art of Multiprocessor Programming* (you can find a copy in the Drive)

A bit of theory

When learning about concurrency, beware that there are many conflicting definitions, examples and analogies floating around, some of which are misleading and might cause you unnecessary suffering. I recommend starting with the resources linked above.

Note that high-level programmers define concurrency differently from low-level programmers. Therefore, there are two "correct" definitions of concurrency:

- In the **OS** (low-level) tradition:
    - Concurrency = tasks overlap in time (including by interleaving)
    - Parallelism = tasks run simultaneously (on multiple processors)

        - Parallel **implies** Concurrent
            - All parallel tasks are also concurrent
            - Not all concurrent tasks are parallel
    - In the PLT (high-level) tradition:
        - Concurrency is a language concept: constructing a program as independent parts
        - Parallelism is a hardware concept: executing computations on multiple processors simultaneously
        - Parallelism and Concurrency are **orthogonal**
            - A concurrent program may or may not execute in parallel
            - A parallel computation may or may not have the notion of concurrency in its structure.

As a consequence, the answer to "does parallel imply concurrent?" depends on the school of thought. As you will see, all the linked resources abide by the second school of thought. In general, the high-level view of concurrency is becoming more predominant, so it is important that you are aware of it and prove your understanding when answering the questions.

To make it clear, if your answer is along the lines of "concurrency is a more general form of parallelism" or "concurrency is when tasks *seemingly* execute at the same time", I will consider your answer **wrong**. You must at least mention the second definition.

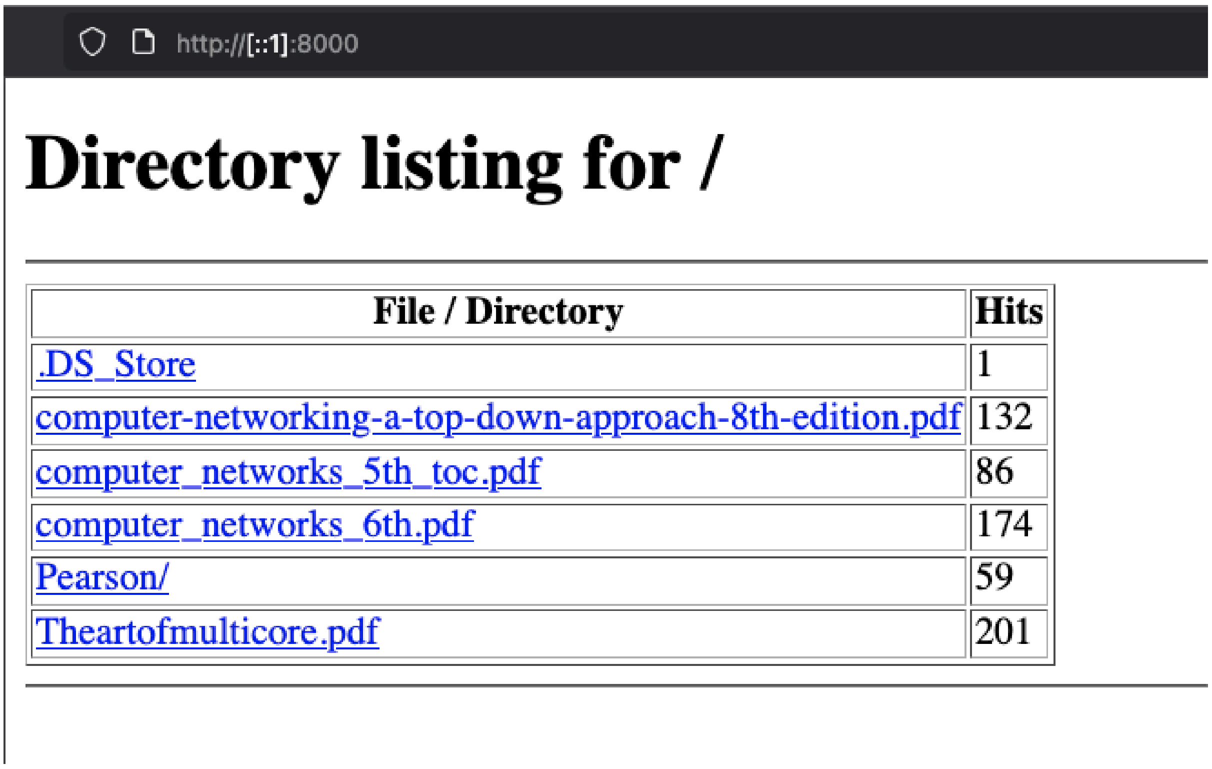
The task

In this lab, you will make your HTTP server multithreaded, so that it can handle multiple connections concurrently. You can either create a thread per request, or use a thread pool.

To test it, write a script that makes multiple concurrent requests to your server. Add a delay to the request handler to simulate work (~1s), make 10 concurrent requests and measure the amount of time in which they are handled. Do the same with the single-threaded server from the previous lab. Compare the two.

Counter: 2 points

Add a counter feature. Record the number of requests made to each file and show it in the directory listing. For example:


http://[::1]:8000

Directory listing for /

File / Directory	Hits
.DS_Store	1
computer-networking-a-top-down-approach-8th-edition.pdf	132
computer_networks_5th_toc.pdf	86
computer_networks_6th.pdf	174
Pearson/	59
Theartofmulticore.pdf	201

First, implement it in a naive way and show that there is a race condition (you can rewrite the code and add delays to force interlacing of threads). Then, use a synchronization mechanism (e.g. a lock) and show that the race condition is gone.

Rate limiting: 2 points

Implement rate limiting by client IP (~5 requests/second) in a thread-safe way. Have one friend spam you with requests and another send requests just below the rate limit. Compare the throughput for both (successful requests/second).