Task 2
GitHub

1 Basic Task

1.1 Description

The goal of the basic task is to implement a simple HTTP webserver. Further, the throughput of the webserver should be measured by sending increasing numbers of requests from a client located on the same host.

1.2 Setup and Methodology

For developing our server we chose C. The language is convenient for working with system calls and enables higher a throughput than many other languages.
Our server first creates a stream socket, then sets the `SO_REUSEADDR` socket option to allow reusing an address when no active listening socket uses it. Further, we set the port to 8081 and the interface index of the address to the index of the `swissknife0` interface. The newly created socket is then bound to the address and the server starts listening on it. In the loop, it waits for connections using `accept`, then reads from the socket and always replies with an `HTTP/1.1 200 OK` message if the `read` call was successful.
The repository also contains a `README.MD` that shows main steps to reproduce environment and benchmarks related to this experiment.

1.3 Benchmarking

In order to measure the throughput of our solution, we used the `wrk` HTTP benchmarking tool. It allows to specify the number of connections to the server. We measured the throughput for 10 to 100 connections with a 10 increment. Before starting the benchmarks, we resolve the IP address of the `swissknife0` interface and use `wrk` to send requests to `http://[ip]:port`.

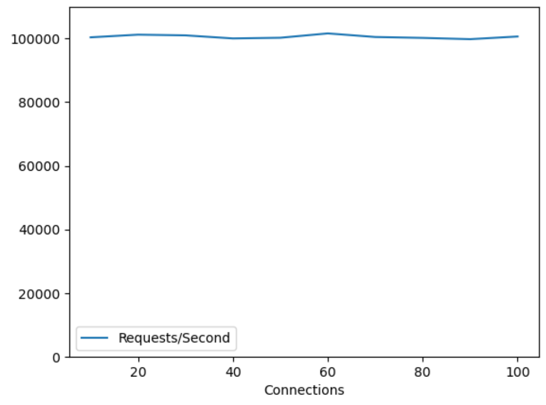

Figure 1: Basic Server Throughput

1.3.1 Result

The throughput in requests per second against the number of connections can be seen in Figure 1. Because the server only handles one connection at a time, the throughput almost does not vary with increasing the number of connections specified in `wrk`.

2 Exploration Task

2.1 Description

In the exploration task, we try to improve the performance of the webserver using various I/O methods e.g. I/O multiplexing (`select`, `epoll`), asynchronous I/O (`io_uring`). Besides, we use threads and subprocesses to implement concurrent servers. We measured and plotted their throughput.

2.2 Setup and Methodology

We started out by creating and analyzing a flame graph of our server, identifying two (expected) hotspot: `receive`, and send. We thus focused our efforts on improving the performance of these functions by employing advanced IO methods, such as select, epoll, multiprocessed epoll, a multithreaded (normal) server, and finally using io_uring. The main flow of the program with select and epoll is just like the basic task: create a socket, set the flag of SO_REUSEADDR, bind it to a local address, and mark it as a listen port. The difference lies in that the socket is now set to nonblocking I/O with ioctl or fcntl, so it will not block at accept or recv/send, which also reduces process scheduling overhead. I/O multiplexing in use here enables our single process to handle multiple connections efficiently. These two methods have a only slight improvement over the basic server with the benchmark tool wrk, since I/O multiplexing improves performance in real-world scenarios where users interact in a meaningful way with the server, upload or request files or otherwise engage in IO-heavy operations. In the tested scenario sockets are opened and kept alive for the length of the correspondence with the server, with only very short IO operations taking place (sending and receiving one message on a socket). This is a limitation imposed by wrk (or any other performance measuring tool for that matter), since it cannot reliably simulate human behavior.

select monitors passed in file descriptor sets for any changes to see when one (or more) file descriptors become ready (e.g. input becomes available), hence perform an IO operation without blocking. A downfall of select is that it can only limit a maximum of 1024 file descriptors. We thus had to copy entire file descriptor set in to the kernel every time, loop and individually check the file descriptor set for changes, and finally rewrite the fd_set for another round of polling resulting in performance loss.

epoll is an I/O event notification facility. With epoll, we can dynamically add or delete the file descriptor in the interest list managed by the kernel and use epoll_wait to get the active fd from the ready list which is populated also by the kernel. The incoming connection sockets are set to EPOLLIN|EPOLLET first, and the data in the read buffer will be read out until it returns EWOULDBLOCK.

io_uring is newly introduced, asynchronous I/O method which enables many costly IO operations to be executed (almost) simultaneously. By using shared-memory ring buffers between kernel and user space, copying data to/from the kernel is abstracted away and prevented from blocking the main calling process. To be more precise, io_uring uses two ring queues, the submission queue on which a process submits entries to be executed by the kernel. The second ring queue, the completion queue, contains the return values of submission queue entries. The user has to sufficiently describe entries on the submission queue, because it is not guaranteed that they complete in the same order as they are entered. These entries are essentially equivalents of normal system calls.

The above two synchronous IO multiplexing model provide us a possibility to handle multiple keep-alive connections and check which are ready at moment. We also try to implement a multithreaded version and a multiprocesss version. The multithreaded version will create a new thread to handle new connections returned by accept. Even though we didn't create a thread pool and still need to bear the overhead of creating threads, as threads are lighter than processes and there is no need to add any mutex locks, the server can still reach a high throughput, but its' performance drops fast when the load increases. The multiprocessed version initially forks a fixed number of subprocesses, and utilizes the mechanism of SO_REUSEPORT which is provided since Linux 3.9 to make different epoll_fds listen on the same port. The kernel handles the thundering herd problem and distribute the incoming connection to different processes to achieve load balancing.

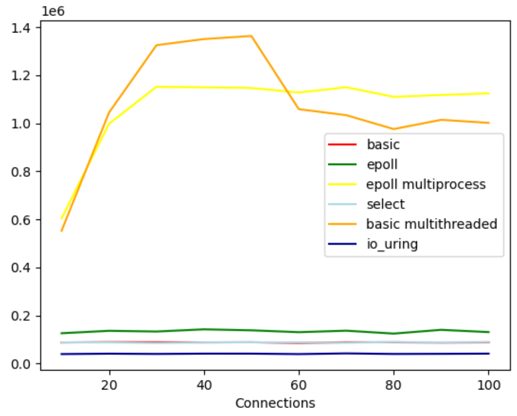

Figure 2: Throughput in requests/second in respect to connections

2.3 Result

In figure 3 a plot of all variants' measured throughput can be seen. It is evident that the basic multithreaded server variant had the highest throughput, peaking at 1.4 million requests per second, but it's not stable. Thus the fastest stable version is the multiprocess server with epoll, with 1135k req/s. Most of the variants performed similar or slightly poorer than the basic server which can be explained with the added overhead in setup, which is not balanced with a higher throughput since the benchmark profile doesn't involve a lot of IO. A more precise image of performance could be obtained in the future if the benchmarking profile would more accurately simulate IO operations such as file requests, since this is where the different IO variants shine over the basic server.

References

[1] https://unixism.net/2020/04/io-uring-by-example-part-3-a-web-server-with-io-uring/

[2] https://man7.org/linux/man-pages/man7/epoll.7.html

[3] https://man7.org/linux/man-pages/man7/socket.7.html

[4] https://lwn.net/Articles/542629/