https://folk.ntnu.no/michaeng/tdt4186_22/
michael.engel@ntnu.no

Practical exercises
Spring 2022

Practical Exercise 2

Multithreading and Communication

Please submit solutions on Blackboard by Tuesday, 22.3.2022 12:00h

2.1 A multithreaded web server

You all use a web server every day – though, of course, not directly, but through your web browser. In this exercise, you are going to *implement* a web server for a Unix system in several steps.

The task is to implement a multi-threaded webserver `mtwwwd` in a file `mtwwwd.c`. The server listens on a configurable TCP port `port` and delivers HTML files that are located in a directory hierarchy below the web root directory passed as `www-path`.

The web server should be invoked as follows:

```
mtwwwd www-path port #threads #bufferslots
```

Parameters three and four (#threads and #bufferslots) are used in subtask d. A reguest to the webserver using the given TCP port looks as follows:

```
GET /doc/index.html
```

The given path name must not contain whitespace characters (space, tab, etc.) and has to refer to an existing file. So with a given `www-path` of `/home/fritz/webroot` and a request path of `/doc/index.html`, the file to be delievered is `/home/fritz/webroot/doc/index.html`.

All text of the request following the path name is to be ignored. The request line is terminated by either the character sequence “\n” (linefeed, Unix line end convention) or “\r\n” (Windows line end convention). After receiving the request, the server should deliver the requested file to the client (which might be a web server or a tool such as `wget` or `curl`) using the HTTP/0.9 protocol and close the TCP connection after delivering the file contents.

a. Start by writing a single-threaded web server. The main thread should start serving a request as soon as it is received and should only accept a new connection (using `accept(2)`) after serving the previous request. Test that your web server correctly delivers documents using different paths using a web browser.

   Hint: You can optionally implement error handling by returning the infamous 404 error (see https://en.wikipedia.org/wiki/HTTP_404)

b. Implement *counting* semaphores to coordinate POSIX threads (`pthread_mutex_lock(3)`, `pthread_cond_wait(3)`) in the module sem (file `sem.c`).

(continued on the next page)

c. Implement a *ring buffer* in the module bbuffer (file `bbuffer.c`), which will be used for communicating between a *single* producer and *multiple* consumer threads.

   Use your semaphore implementation from subtask b to synchronize over- and underflow situations of the ring buffer, so that producers block when trying to insert an element into a full buffer and receivers block when trying to take an element from an empty buffer.

d. Now extend your server using POSIX threads so that multiple *worker threads* perform the processing of the requests and the main thread is only responsible for accepting connections and distributing the work. At the start of the server, create `#threads` worker threads, which should continue to exist throughout the lifetime of the server.

   Use a *ring buffer* with `#bufslot` entries to exchange data between the worker threads and the main thread. The main thread now only accepts connections and passes the respective *file descriptor* of the socket representing the connection into the ring buffer. The worker threads then take one of the socket file descriptors from the ring buffer and process the connection.

   The connection is to be processed by:

   - Receive the request on the socket file descriptor passed in the ring buffer
   - Parse the request
   - Open the file indicated in the request
   - Create and send a HTTP/0.9 response
   - Deliver the requested file
   - Close the connection
   - Start with the next request

e. *Extra credit:* As implemented, the web server has a significant *security problem*: it allows not only to access files under the `www-path` directory, but enables access to *all files* in the file system of the server that can be accessed by the user who started the webserver process.

   Find a way to exploit this security problem. Then, find *two different* ways to prevent access to files outside of the webroot directory.

Hints:

- A tutorial for using POSIX threads (pthreads) can be found at
  https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html
- A detailed description of version 0.9 of the http protocol can be found at
  http://www.w3.org/Protocols/HTTP/AsImplemented.html.
- On Unix systems, you can usually only open TCP ports < 1024 for incoming connections as root. You can use a higher port number (e.g. 8000) for testing as a regular user. The port number can be specified in the URL:
  http://localhost:8000/doc/index.html
- In the file p2.tar, you can find header files containing interface descriptions for the modules `sem` and `bbuffer`. Comments in the header files provide a description of the API. Do not change the header files of the modules.
- The server should be able to serve requests via IPv4 as well as IPv6.