HW3 - Web Servers and Synchronization

Due Date: 19.06.2025

TAs in charge: Shachar Asher Cohen – cohenshachar@campus.technion.ac.il

Important: The Q&A for the exercise will take place at a public forum Piazza only. Critical updates about the HW will be published in pinned notes in the piazza forum. These notes are mandatory and it is your responsibility to be updated. A number of guidelines to use the forum:

- Read previous Q&A carefully before asking the question; repeated questions will probably go without answers.
- Be polite, remember that course staff does this as a service for the students.
- You're not allowed to post any kind of solution and/or source code in the forum as a hint for other students; In case you feel that you have to discuss such a matter, please come to the reception hour.
- When posting questions regarding this assignment, put them in the "hw3-wet" folder. Pay attention that in the dry part there are questions about the wet part, so it is recommended to review these questions before, and answer them gradually while working on the wet part.

Only the TA in charge, can authorize postponements. In case you need a postponement, please fill out the attached form: https://forms.office.com/r/LVhfmu3ZLv?origin=lprLink

Introduction

In this assignment, you will develop a **working multi-threaded web server**. The server will maintain a **log of handled jobs** ("the server's book") and allow **multiple concurrent clients** to **read** or **add entries** to this log.

To simplify the project, we provide a basic single-threaded web server that already:

- Handles both **GET** and **POST** requests
- Includes built-in response composition
- Abstracts away low-level networking and HTTP handling
- A stub log implementation and more necessary structs for quality of life.

Your task is:

1) to make this server **multi-threaded** for efficiency.
2) add statistic collection as will be described later.
3) add logging **functionality**.

HTTP Background

This section is a brief overview of the **HTTP protocol**, which is already handled for you in the starter code.

- Web browsers and servers communicate via **HTTP** (Hypertext Transfer Protocol), a text-based request/response protocol.
- The browser opens a connection, makes a request (e.g., for a file), the server responds, and the connection is closed.
- **GET** requests are typically used to retrieve static content (such as HTML files) from the server. Each file has a unique name known as a URL (Universal Resource Locator).
  For example, the URL: www.cs.technion.ac.il:80/index.html identifies an HTML file called "/index.html" on Internet host "www.cs.technion.ac.il" that is managed by a web server listening on port 80.
- URLs can also reference **dynamic content**, like CGI scripts, using the ? and & characters to pass arguments, e.g. A request like: “GET /req.cgi?user=alex&id=5” is asking to run req.cgi with the parameters user = alex and id = 5 and return the

output. (If you're interested, you can read more about CGI in the attached link- what is cgi programming - Stack Overflow).

- **POST** requests are used to submit data to the server, often resulting in a change in server state or side effects (e.g., updating a database or creating a resource).
- In this assignment:
    - **GET request** behaves similarly to a traditional GET request, with one key difference - It also **appends an entry** representing the request itself to the book **before returning it**.
    - **POST request** retrieves the current contents of the **server's book**, which is a log of all requests previously made to the server.

HTTP Request Structure:	**HTTP Response Structure:**
consists of a request line, followed by zero or more request headers, and finally an empty text line. A request line has the form: [method] [uri] [version]. The method is usually GET (but may be other things, such as POST, OPTIONS, PUT or whatever the server knows how to handle).	consists of a response line, zero or more response headers, an empty text line, and the response body. A response line has the form [version] [status] [message]. The status is a three-digit positive integer that indicates the state of the request; some common states are 200 for "OK", 403 for "Forbidden", and 404 for "Not found". Two important lines in the header are "Content-Type", which tells the client the MIME type of the content in the response body (e.g., html or gif) and "Content-Length", which indicates its size in bytes.
[method] [uri] [version] [header1]: [value1]	[version] [status code] [message] Content-Type: text/html

[header2]: [value2] [optional body for POST requests]	Content-Length: 123 [body content]

If you would like to see the HTTP protocol in action, you can connect to any web server using telnet. For example, run telnet www.cs.technion.ac.il 80 and then type:

```
GET / HTTP/1.1
host: www.cs.technion.ac.il
[empty line - simply press enter twice after previous line]
```

You will then see the raw HTML text for that web page!

You do **not need to modify** any of the code related to HTTP parsing or network connections. The starter code fully handles these aspects for you. However, understanding this background will help you reason about how requests are structured and processed internally and how to complete the assigment.

Basic Web Server

The code for the web server is available on the course website. You can compile the files by simply typing make. Compile and run this basic web server before making any changes to it! make clean removes .o files and executables and lets you do a clean build.
When you run this basic web server, you need to specify the port number that it will listen on; ports below number 1024 are *reserved* (see the list here) so you should specify port numbers that are greater than 1023 to avoid this reserved range; the max is 65535. Be wary: if running on a shared machine, you could conflict with others and thus have your server fail to bind to the desired port. If this happens, try a different number!
You can run the server by running:

```
./server [desired_port]
```

In addition, we have provided you with a very primitive HTTP client. This client can request a file from any HTTP server by running:

```
./client [hostname] [port] [filename] [method]
```

The server supports the methods GET and POST.

When you connect to the server, make sure that you specify this same port. For example, assume that you started the server on csl3.cs.technion.ac.il and used port number 8003, by running:

```
./server 8003
```

The server can respond with any html file you put in the public directory (automatically generated by make). Let's assume you copied there your favorite html file - favorite.html.

To view this file using the provided client (running on the same or a different machine), run:

```
./client csl3.cs.technion.ac.il 8003 favorite.html
```

To view this file from a web browser (running on the same or a different machine), use the url:

```
csl3.cs.technion.ac.il:8003/favorite.html
```

If you run the client and web server on the same machine, you can just use the hostname localhost as a convenience, e.g., localhost:8003/favorite.html.

For your convenience we added the basic home.html file for you to experiment with.

<u>Notes</u>

1. On some browsers you might need to use an incognito tab.
2. If you use csl3, your client/browser must be connected to the Technion's network. If you don't have access to the Technion's network (or just don't want to use csl3), you can always run the server locally on your machine and access it with localhost.

To make the homework a bit easier, the web server is very minimal, consisting of only a few hundred lines of C code. As a result, the server is limited in its functionality; it does not handle any HTTP requests other than **GET and POST,** understands only a few content types, and supports only the QUERY_STRING environment variable for CGI programs. This web server

is also not very robust; for example, if a web client closes its connection to the server, it may trip an assertion in the server causing it to exit. We do not expect you to fix these problems (though you can, if you like, you know, for fun).

Helper functions are provided to simplify error checking. A wrapper calls the desired function and immediately terminates if an error occurs. The wrappers are found in the file segel.h); more about this below. **One should always check error codes, even if all you do in response is exit;** dropping errors silently is BAD C PROGRAMMING and should be avoided at all costs.

Part 1: Multi-threaded Web Server — Detailed Explanation and Guidance

Background:

The provided starter server is single-threaded — it can only handle one HTTP request at a time. This design leads to performance bottlenecks, especially when:

- A request takes a long time to complete (e.g., running a CGI script).
- Disk I/O or network latency causes delays in responding.

Since the server blocks on one request, all other incoming clients must wait, even for trivial requests. This results in poor scalability and user experience.

The simplest multithreading strategy is to create **a new thread for every incoming request**. This ensures:

- Lightweight/fast requests don't get stuck behind slow ones.
- Threads blocked on I/O don't hold up the rest of the server.

However, this approach has downsides:

- **Thread creation overhead** on each request.
- Potential performance degradation under high load.

Task 1: Implement a Fixed-Size Thread Pool Producer-Consumer Server

Overview

You will modify the provided basic web server to handle multiple HTTP requests concurrently by implementing:

- One master thread (the producer) that accepts incoming client connections.
- A fixed-size pool of worker threads (the consumers) that process these requests in parallel.

This design is often called a thread pool and is more efficient than creating a new thread for every request because:

- Thread creation overhead is avoided on each request.
- Thread management is centralized and controlled.
- It allows better control over server resource usage and load.

How It Works

1. Master Thread (Producer):
    - Listens on the server socket for new client connections.
    - When a connection is accepted, it adds the connection to a bounded FIFO queue.
    - If the queue is full, the master thread blocks (waits) until space becomes available.
    - Signals worker threads that new work is ready.
2. Worker Threads (Consumers):
    - Created once at server startup; the pool size is fixed.
    - Each worker thread waits on a condition variable for work to appear in the queue.
    - When awakened, a worker picks the oldest request from the queue (FIFO).
    - Processes the request (handle GET or POST) – provided function see request.h/c.
    - Returns to waiting for the next request.

Synchronization Requirements

Because multiple threads (master + workers) access shared resources (the request queue, the server's log), synchronization is crucial.

- Use mutex locks to protect the shared request queue against race conditions.
- Use condition variables to avoid busy-waiting:

    - Workers wait when the queue is empty.
    - Master waits when the queue is full.
- No spin-locks or busy waiting allowed – will be penalized heavily. (i.e., no looping while checking conditions without sleeping).

Handling GET and POST Requests with Reader-Writer Synchronization:

The server's will be loging valid requests it has handled. A shared data structure called Server_Log (see log.h/c) maintains a log of all handled requests.

- **GET requests:**
    - Are considered writers to the book — they append an entry to the log.
    - Require exclusive access to the book.
- **POST requests:**
    - Are considered readers — they read and return the entire log.
    - Multiple POST requests (readers) can access the book simultaneously.

Task 2: You must implement the log (see log.c/h) with a reader-writer synchronization scheme with writer priority:

- If readers and writers are waiting simultaneously, give precedence to writers.
- This ensures writers don't starve behind a flood of readers.

Edge Cases and Important Notes

- **Queue Size Limit:** The bounded queue has a fixed capacity (queue_size).
    - When full, the master thread must block, so new connections wait until there is space.
- **Queue Management:**
    - You may implement one queue for pending requests.
    - Optionally, you can maintain a second structure for requests currently being processed to enforce queue size constraints (sum of pending + active ≤ queue_size).
- Worker threads should never block other threads while processing their assigned request. This means once a worker takes a request off the queue, it releases the lock so others can continue.
- The basic server forks a process for CGI in requestServeDynamic:
    - Your multi-threaded server must not modify this CGI forking logic.

    - The server currently waits for each CGI child process before continuing.
- FIFO order of requests must be maintained when worker threads pick requests from the queue.
- Proper error checking and robust synchronization (no race conditions, no deadlocks) are required.

Implementation Hints

- Use pthread_mutex_t for mutexes and pthread_cond_t for condition variables.
- Initialize your thread pool and synchronization primitives at server startup.
- Use functions provided in segel.c for error-checked system calls and synchronization primitives.
- Implement the reader-writer lock for the server's book log separately, ensuring:
    - Multiple concurrent POST readers.
    - Exclusive GET writers with priority.

Part 2: Usage Statistics

You will need to modify your web server to collect a variety of statistics. Some of the statistics will be gathered on a per-request basis and some on a per-thread basis. All statistics will be returned to the web client as part of each http response. Specifically, you will be embedding these statistics in the response headers; we have already made place-holders in the basic web server code for these headers (Check out how the server sends "Content-length" and "Content-type" and what the client does with this information). Note that most web browsers will ignore these additional headers; to access these statistics, you will want to run our modified client.

For each request, you will record the following times or durations at the granularity of milliseconds. You may find `gettimeofday()` useful for gathering these statistics.

- **Stat-req-arrival**: The arrival time, as first seen by the master thread

  format:

```
Stat-Req-Arrival:: %ld.%06ld
```

- **Stat-req-dispatch**: The dispatch interval (the duration between the arrival time and when the request was picked up by worker thread)

  format:

```
Stat-Req-Dispatch:: %ld.%06ld
```

You should also keep the following statistics for each thread:

- **Stat-thread-id**: The id of the responding thread (Let N be the number of worker threads: Worker_IDs numbered: 1 to N)

  format:

```
Stat-Thread-Id:: %d
```

- **Stat-thread-count**: The total number of http requests this thread has handled

  format:

```
Stat-Thread-Count:: %d
```

- **Stat-thread-static**: The total number of static requests this thread has handled

  format:

```
Stat-Thread-Static:: %d
```

- **Stat-thread-dynamic**: The total number of dynamic requests this thread has handled

  format:

```
Stat-Thread-Dynamic:: %d
```

- **Stat-thread-post**: The total number of post requests this thread has handled

  format:

```
Stat-Thread-Post:: %d
```

Thus, for a request handled by thread number i, your web server will return the statistics for that request and the statistics for thread number i.

For this we provided the function append_stats in request.c:

```
int append_stats(char* buf, threads_stats t_stats, struct timeval arrival, struct timeval dispatch){
    int offset = strlen( Str: buf);  // Start after what's already written to buf

    offset += sprintf( stream: buf + offset,  format: "Stat-Req-Arrival:: %ld.%06ld\r\n",
                       arrival.tv_sec, arrival.tv_usec);

    offset += sprintf( stream: buf + offset,  format: "Stat-Req-Dispatch:: %ld.%06ld\r\n",
                       dispatch.tv_sec, dispatch.tv_usec);

    offset += sprintf( stream: buf + offset,  format: "Stat-Thread-Id:: %d\r\n",
                       t_stats->id);

    offset += sprintf( stream: buf + offset,  format: "Stat-Thread-Count:: %d\r\n",
                       t_stats->total_req);

    offset += sprintf( stream: buf + offset,  format: "Stat-Thread-Static:: %d\r\n",
                       t_stats->stat_req);

    offset += sprintf( stream: buf + offset,  format: "Stat-Thread-Dynamic:: %d\r\n",
                       t_stats->dynm_req);
    offset += sprintf( stream: buf + offset,  format: "Stat-Thread-Post:: %d\r\n\r\n",
                       t_stats->post_req);
    return offset;
}
```

Your code should follow the following logic:

- All requests (including errors) increment the request counter.
- valid static requests increment the static counter.
- valid dynamic requests increment the dynamic counter
- valid post requests increment the post counter
- 501, 403 and 404 errors do not increment the dynamic/static/post counters

*Note that some of the implementation should happen in handle_request.

Server Log:

The log must contain a record of all valid request statistics handled by the server (the records being the statistics gathered at handling. The Log is required to be returned as a

chained string of the all the requests stats at handle. i.e. the data pushed to the log should be identical to that returned from the append_stats function output on an empty buffer.

- GET adds an entry to the book; POST reads and returns the book contents.

Program Specifications

Your C program must be invoked exactly as follows:

```
./server [portnum] [threads] [queue_size]
```

The command line arguments to your web server are to be interpreted as follows:

- **portnum**: the port number that the web server should listen on; the basic web server already handles this argument.
- **threads**: the number of worker threads that should be created within the web server. Must be a positive integer.
- **queue_size**: the number of request connections that can be accepted at one time. Must be a positive integer. Note that it is not an error for more or less threads to be created than buffers.

For example, if you run your program as

```
./server 5003 8 16
```

then your web server will listen to port 5003, create 8 worker threads for handling http requests, allocate a 16 buffers queue for connections that are currently in progress or waiting, and use drop tail scheduling for handling overload.

Hints

We recommend understanding how the code that we gave you works. We provide the following .c files:

- **server.c**: Contains main() for the basic web server.
- **request.c**: Performs most of the work for handling requests in the basic web server.
- **log.c**: A stub of a log for you to implement your real logging logic connected to request.c already so see that you work correctly with what is given (you are allowed to change almost everything if you see fit to do so, in the end your server will be compiled and executed).
- **segel.c**: Contains wrapper functions for the system calls invoked by the basic web server and client. The convention is to capitalize the first letter of each routine. Feel free to add to this file as you use new libraries or system calls. You will also find a corresponding header (.h) file that should be included by all of your C files that use the routines defined here.
- **client.c**: Contains main() and the support routines for the very simple web client. To test your server, you may want to change this code so that it can send simultaneous requests to your server. At a minimum, you will want to run multiple copies of this client.
- **output.c**: Code for a CGI program that repeatedly sleeps for a random amount of time. You may find that having a CGI program that takes awhile to complete is useful for testing your server. The documentation for how to use this program is in the source file.

- You might need to add data structures, create your own implementations and add them to your submission.

We also provide you with a sample Makefile that creates server, client, and output.cgi. You can type "make" to create all of these programs. You can type "make clean" to remove the object files and the executables. You can type "make server" to create just the server

program, etc. If you create new files, you will need to add them to the Makefile. **You are allowed to program only in C, and include only the headers included in segel.h.**

The best way to learn about the code is to compile and run it.
Run the server we gave you with your preferred web browser, run this server with the client code we gave you. You can even have the client code we gave you contact any other server. Make small changes to the server code (e.g., have it print out more debugging information) to see if you understand how it works.

We have provided a few comments, marked with "HW3", to point you to where we expect you will make changes for this project. We recommend first making the server multi-threaded, then add in the different overload handling algorithms, beginning with the easiest (drop tail), and keep the usage statistics for last.

We anticipate that you will find the following routines useful for creating and synchronizing threads: *pthread_create, pthread_mutex_init, pthread_mutex_lock, pthread_mutex_unlock, pthread_cond_init, pthread_cond_wait, pthread_cond_signal*.

Example Output

Attached are some screenshots of our basic webserver for GET(static) and POST requests for you to compare and see that you are ready to start.

```
shachar@Shachar:/mnt/c/Users/shach/Desktop/server$ ./server 8000
```

```
shachar@Shachar:/mnt/c/Users/shach/Desktop/server$ ./client localhost 8000 home.html POST
clientfd = 3
POST home.html HTTP/1.1
host: Shachar


sentHeader: HTTP/1.0 200 OK
Header: Server: OS-HW3 Web Server
Length = 24
Header: Content-Length: 24
Header: Content-Type: text/plain
Header: Stat-Req-Arrival:: 0.000000
Header: Stat-Req-Dispatch:: 0.000000
Header: Stat-Thread-Id:: 0
Header: Stat-Thread-Count:: 0
Header: Stat-Thread-Static:: 0
Header: Stat-Thread-Dynamic:: 0
Log is not implemented.
```

```
shachar@Shachar:/mnt/c/Users/shach/Desktop/server$ ./client localhost 8000 home.html GET
clientfd = 3
GET home.html HTTP/1.1
host: Shachar


sentHeader: HTTP/1.0 200 OK
Header: Server: OS-HW3 Web Server
Length = 293
Header: Content-Length: 293
Header: Content-Type: text/html
Header: Stat-Req-Arrival:: 0.000000
Header: Stat-Req-Dispatch:: 0.000000
Header: Stat-Thread-Id:: 0
Header: Stat-Thread-Count:: 0
Header: Stat-Thread-Static:: 0
Header: Stat-Thread-Dynamic:: 0
<html>

<head>
  <title>OS-HW3 Test Web Page</title>
</head>

<body>

<h2> OS-HW3 Test Web Page</h2>

<p> Test web page: simply awesome.</p>

<p> Click <a href="https://www.youtube.com/watch?v=dQw4w9WgXcQ"> here</a> for something
even more awesome.</p>

</body>
</html>
```

Submission

You should create a zip file containing

- All of your server source files (*.c and *.h)
- a Makefile
- A file named submitters.txt which includes the ID, name and email of the participating students. The following format should be used:

```
Linus Torvalds linus@gmail.com 234567890

Ken Thompson ken@belllabs.com 345678901
```

Important Note 1: Make sure your code compiles! We do not add any files when testing your code therefore run the Makefile on your submission and make sure it works. There is no need submit any .o files.

Important Note 2: when you submit, keep your confirmation code and a copy of the file(s), in case of technical failure. Your confirmation code is the only valid proof that you submitted your assignment when you did.Good luck!

-- Course staff