HW3 - Web Servers and Synchronization

Due Date: 17/06/2021 23:30

TAs in charge: Bar Raveh & Mohammed Dabbah

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 "hw2-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.gle/qU7uk94EHDdbSswc7

Introduction

In this assignment, you will be developing a real, working **web server.** To greatly simplify this project, we are providing you with the code for a very basic web server. This basic web server operates with only a single thread; it will be your job to make the web server multi-threaded so that it is more efficient.

HTTP Background

Before describing what you will be implementing in this project, we will provide a very brief overview of how a web server works and the HTTP protocol. Our goal in providing you with a basic web server is that you should be shielded from all of the details of network connections and the HTTP protocol. **The code that we give you already handles everything that we describe in this section**.

Web browsers and web servers interact using a text-based protocol called HTTP (Hypertext Transfer Protocol). A web browser opens an Internet connection to a web server and requests some content with HTTP. The web server responds with the requested content and closes the connection. The browser reads the content and displays it on the screen.

Each piece of content on the server is associated with a file. If a client requests a specific disk file, then this is referred to as static content. If a client requests that a executable file be run and its output returned, then this is dynamic content. 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. The port number is optional and defaults to the well-known HTTP port of 80. URLs for executable files can include program arguments after the file name. A '?' character separates the file name from the arguments and each argument is separated by a '&' character. This string of arguments will be passed to a CGI program as part of its "QUERY_STRING" environment variable.

An HTTP request (from the web browser to the server) 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, or PUT). The URI is the file name and any optional arguments (for dynamic content). Finally, the version indicates the version of the HTTP protocol that the web client is using (e.g., HTTP/1.0 or HTTP/1.1).

An HTTP response (from the server to the browser) is similar; it consists of a response line, zero or more response headers, an empty text line, and finally the interesting part, 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.

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 HTML text for that web page!

Again, you don't need to know this information about HTTP unless you want to understand the details of the code we have given you. **You will not need to modify any of the procedures in the web server that deal with the HTTP protocol or network connections.**

Basic Web Server

The code for the web server is available in 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]
```

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 convivence we added the basic home.html file for you to experiment with.

Notes

1. On some browsers you might need to use an incognito tab.
2. If you use csl3, you 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 with localhost.

To make the project 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, 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 terminate 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

The basic web server that we provided has a single thread of control. Single-threaded web servers suffer from a fundamental performance problem in that only a single HTTP request can be serviced at a time. Thus, every other client that is accessing this web server must wait until the current http request has finished; this is especially a problem if the current http request is a long-running CGI program or is resident only on disk (i.e., is not in memory). Thus, the most important extension that you will be adding is to make the basic web server multi-threaded.

The simplest approach to building a multi-threaded server is to spawn a new thread for every new http request. The OS will then schedule these threads according to its own policy. The advantage of creating these threads is that now short requests will not need to wait for a long request to complete; further, when one thread is blocked (i.e., waiting for disk I/O to finish) the other threads can continue to handle other requests. However, the drawback of the one-thread-per-request approach is that the web server pays the overhead of creating a new thread on every request.

Therefore, the generally preferred approach for a multi-threaded server is to create a **fixed-size pool of worker threads** when the web server is first started. With the pool-of-threads approach, each thread is blocked until there is an http request for it to handle. Therefore, if there are more worker threads than active requests, then some of the threads will be blocked, waiting for new http requests to arrive; if there are more requests than worker threads, then those requests will need to be buffered until there is a ready thread.

In your implementation, you must have **a master thread that begins by creating a pool of worker threads, the number of which is specified on the command line. Your master**

thread is then responsible for accepting new http connections over the network and placing the descriptor in queue. The size of the queue is also specified on the command line.

Note that the existing web server has a single thread that accepts a connection and then immediately handles the connection; in your web server, this thread should accept the connection, place the connection descriptor into the queue, and return to accepting more connections - letting the worker threads handle the requests.

Each worker thread is able to handle both static and dynamic requests. A worker thread wakes when there is an http request in the queue; when there are multiple http requests available, it should pick the oldest request in the queue.

Once the worker thread wakes, it performs the read on the network descriptor, obtains the specified content (by either reading the static file or executing the CGI process), and then returns the content to the client by writing to the descriptor. The worker thread then waits for another http request.

Note that the master thread and the worker threads are in a producer-consumer relationship and require that their accesses to the shared buffer be synchronized. Specifically, the master thread must block and wait if the queue is full; a worker thread must wait if the queue is empty. In this project, you are required to use *condition variables*. **If your implementation performs any busy-waiting (or spin-waiting) instead, you will be heavily penalized.**

Hints

1. You might want to separate the queue into two - one queue for requests waiting to be picked up by a worker thread, and one queue (or list) for requests currently handled by some worker thread. If you do so - make sure that the sum of requests in both those queues is ≤ the size of the queue specified in the command line.
2. Make sure that the worker thread does not block the other threads while handling its request!

Notes

1. Do not be confused by the fact that the basic web server we provide forks a new process for each CGI process that it runs (in requestServeDynamic). Although, in a very limited sense, the web server does use multiple processes, it never handles more than a single request at a time; the parent process in the web server explicitly waits for the child CGI process to complete before continuing and accepting more http requests._ **For the first part, you should not modify this section of the code.**

Part 2: Overload Handling

In this project, you will implement a number of different policies for handling overload. In many web servers, the server stops serving any requests when too many requests come in because clients time out before data is returned.

You will implement several overload policies in your web server. When there are enough buffers, your code should just queue requests without using a policy and let the threads execute them in the order they arrived. When not enough buffers are available, your code should apply one of the following policies:

- **block** : your code for the listening (main) thread should block (not busy wait!) until a buffer becomes available.
- **drop tail** : your code should drop the new request immediately by closing the socket and continue listening for new requests.
- **drop head** : your code should drop the oldest request in the queue that is not currently being processed by a thread and add the new request to the end of the queue.
- **Bonus (5 pt) - random drop** : when the queue is full and a new request arrives, drop 25% of the requests in the queue (that are not handled by a thread) randomly. You can use the rand() function to choose which to drop. An example of this function is in output.c. Once old requests have been dropped, you can add the new request to the queue.

Part 3: 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 entity 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 headers that it doesn't know about; 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
- **Stat-req-dispatch**: The dispatch interval (the duration between the arrival time and when the request was picked up by worker thread)

You should also keep the following statistics for each thread:

- **Stat-thread-id**: The id of the responding thread (numbered 0 to number of threads-1)
- **Stat-thread-count**: The total number of http requests this thread has handled
- **Stat-thread-static**: The total number of static requests this thread has handled
- **Stat-thread-dynamic**: The total number of dynamic requests this thread has handled

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.

The webserver must return the statistics headers in every response, including a response to an invalid request (404,503 etc). Invalid requests should also be counted as handled requests for the request count stats.

Program Specifications

Your C program must be invoked exactly as follows:

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

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.
- **schedalg**: the scheduling algorithm to be performed. Must be one of "block", "dt", "dh", or "random".

For example, if you run your program as

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

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. All procedures in this file begin with the string, "request".
- **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.

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 can include only the headers included in segel.h.**

The best way to learn about the code is to compile it 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*.

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
```

Do not submit any .o files. You do not need to copy any files for creating clients or CGI programs

Important Note: 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