CMPE - 382 Operating Systems

Homework 2

Spring 2022

Guidelines

You need to submit this homework on 27th May at 8pm, on LMS. Late submissions are allowed until 29th May 11:59pm, which will be penalized by 20%. Your work will not be accepted once the submission is closed on LMS.

- You need to do this homework in a group of up to 4 students.
- You may form the group with other sections as well.
- Clearly report group member names in `group-information.txt` file, given along with this homework.
- You will submit your homework to LMS (only one member of the group will submit).
- Implement each deliverable in its corresponding folder, and submit all the folders in a single zip file on LMS.
- It is better to submit incomplete assignment than none at all.
- It is strongly advised that you start working on the assignment the day you get it. Assignments WILL take time.
- Every assignment you submit should be a single zipped file containing all the other files.
- DO NOT send your assignment to your instructor over email. It won't be considered for evaluation.
- You can be called in for Viva for any assignment that you submit
- This homework is 25% worth of overall grade. Each deliverable carry equal percentage of marks.

1 Overview

In this assignment, you will be developing a concurrent web server. To simplify this project, we are providing you with the code for a non-concurrent (but working) 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 can handle multiple requests at the same time.

The goals of this project are: - To learn the basic architecture of a simple web server - To learn how to add concurrency to a non-concurrent system - To learn how to read and modify an existing code base effectively

Useful reading from OSTEP[1] includes: - Intro to threads[2] - Using locks[3] - Producer-consumer relationships[4] - Server concurrency architecture[5]

2 HTTP Background

Before describing what you will be implementing in this project, we will provide a very brief overview of how a classic web server works, and the HTTP protocol (version 1.0) used to communicate with it; although web browsers and servers have evolved a lot over the years[6], the old versions still work and give you a good start in understanding how things work. Our goal in providing you with a basic web server is that you can be shielded from learning all of the details of network connections and the HTTP protocol needed to do the project; however, the network code has been greatly simplified and is fairly understandable should you choose to to study it.

Classic web browsers and web servers interact using a text-based protocol called **HTTP** (**Hypertext Transfer Protocol**). A web browser opens a 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.

HTTP is built on top of the **TCP/IP** protocol suite provided by the operating system. Together, TPC and IP ensure that messages are routed to their correct destination, get from source to destination reliably in the face of failure, and do not overly congest the network by sending too many

[1] <http://ostep.org>

[2] <http://pages.cs.wisc.edu/~remzi/OSTEP/threads-intro.pdf>

[3] <http://pages.cs.wisc.edu/~remzi/OSTEP/threads-intro.pdf>

[4] <http://pages.cs.wisc.edu/~remzi/OSTEP/threads-cv.pdf>

[5] <http://pages.cs.wisc.edu/~remzi/OSTEP/threads-events.pdf>

[6] <https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Evolution_of_HTTP>

messages at once, among other features. To learn more about networks, take a networking class (or many!), or read this free book[7].

Each piece of content on the web server is associated with a file in the server's file system. The simplest is *static* content, in which a client sends a request just to read a specific file from the server. Slightly more complex is *dynamic* content, in which a client requests that an executable file be run on the web server and its output returned to the client. Each file has a unique name known as a **URL** (**Universal Resource Locator**).

As a simple example, let's say the client browser wants to fetch static content (i.e., just some file) from a web server running on some machine. The client might then type in the following URL to the browser: `http://www.cs.wisc.edu/index.htm` This URL identifies that the HTTP protocol is to be used, and that an HTML file in the root directory (`/`) of the web server called `index.html` on the host machine `www.cs.wisc.edu` should be fetched.

The web server is not just uniquely identified by which machine it is running on but also the **port** it is listening for connections upon. Ports are a communication abstraction that allow multiple (possibly independent) network communications to happen concurrently upon a machine; for example, the web server might be receiving an HTTP request upon port 80 while a mail server is sending email out using port 25. By default, web servers are expected to run on port 80 (the well-known HTTP port number), but sometimes (as in this project), a different port number will be used. To fetch a file from a web server running at a different port number (say 8000), specify the port number directly in the URL, e.g., `http://www.cs.wisc.edu:8000/index.html`.

URLs for executable files (i.e., dynamic content) can include program arguments after the file name. For example, to just run a program (`test.cgi`) without any arguments, the client might use the URL `http://www.cs.wisc.edu/test.cgi`. To specify more arguments, the `?` and `&` characters are used, with the `?` character to separate the file name from the arguments and the `& character to separate each argument from the others. For example,`http://www.cs.wisc.edu/test.cgi`? be used to send multiple argumentsxandyand their respective values to the program` test.cgi`. The program being run is called a **CGI program** (short for [Common Gateway Interface](https://en.wikipedia.org/wiki/Common_Gatewa yes, this is a terrible name); the arguments are passed into the program as part of the [QUERY`STRING`'](https://en.wikipedia.org/wiki/Query`string) environment variable, which the program can then parse to access these arguments.

[7] <https://book.systemsapproach.org>

3 The HTTP Request

When a client (e.g., a browser) wants to fetch a file from a machine, the process starts by sending a machine a message. But what exactly is in the body of that message? These *request contents*, and the subsequent *reply contents*, are specified precisely by the HTTP protocol.

Let's start with the request contents, sent from the web browser to the server. This HTTP request 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`, which tells the web server that the client simply wants to read the specified file; however, other methods exist (e.g., `POST`). The `uri` is the file name, and perhaps optional arguments (in the case of dynamic content). Finally, the `version` indicates the version of the HTTP protocol that the web client is using (e.g., HTTP/1.0).

The 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` (i.e., the client can't access that file), and `404` for `File Not Found` (the famous error). Two important lines in the header are `Content-Type`, which tells the client the type of the content in the response body (e.g., HTML or gif or otherwise) and `Content-Length`, which indicates the file size in bytes.

For this project, you don't really 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. However, it's always good to learn more, isn't it?

4 A Basic Web Server

The code for the web server is available in this repository. You can compile the files herein 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[8])

[8] `<https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml>`

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!

When you then connect your web browser to this server, make sure that you specify this same port. For example, assume that you are running on `bumble21.cs.wisc.edu` and use port number 8003; copy your favorite HTML file to the directory that you start the web server from. Then, to view this file from a web browser (running on the same or a different machine), use the url `bumble21.cs.wisc.edu: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`.

To make the project a bit easier, we are providing you with a minimal web server, 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 `io-helper.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.

5 Finally: Some New Functionality!

In this project, you will be providing following deliverables. You'll be mainly working in wserver.c, other files are already implemented completely, you don't need to modify anything there.

5.1 Deliverable 1 : Protected Server

To begin modifying the server, you add a functionality to create a new process upon accepting a connection. For this task, you'll fork a child process, which will be responsible to serve the request. The parent process simply waits for child to finish, and then accepts new connection. This implementation

would still be able to serve just one request, however, since you create a new process on every request, hence the server process is secured from any malicious attempt caused by a request. Please refer to lecture slide 6.44, 6.45 for this implementation.

5.2 Deliverable 2 : Concurrent and Protected Server

You can make the server concurrent i.e. it can serve more than one request concurrently. For this task, you'll modify the code of Deliverabl 1, so that parent process shouldn't wait for completion of the child, rather it simply goes to accepting more connections. Please refer to lecture slide 6.47, 6.48 for this implemenation.

5.3 Deliverable 3 : Multi-threaded Server

Creating a new process on every request is too heavy, as we create a new process every time. A thread creation is a lighter operation as compared to creating a process. 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. Please refer to lecture slide 6.52 for this implementation.

5.4 Deliverable 4 : Worker Threads

so far the server is set to be multi-threaded, however, server pays the cost of creating a new thread upon 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. Please refer to lecture slide 6.53 for this idea.

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 for this connection into a fixed-size buffer; in your basic implementation, the master thread should not read from this connection. The number of elements in the buffer 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 place the connection descriptor into a fixed-size buffer and return to accepting more connections.

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, which request is handled depends upon the scheduling policy. There could be various scheduling policies at this level, as we learned for CPU scheduling, however, you can implement FCFS scheduling for your implementation. 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 buffer is full; a worker thread must wait if the buffer is empty. In this project, you are required to use condition variables. Note: if your implementation performs any busy-waiting (or spin-waiting) instead, you will be heavily penalized.

5.5 Command-line Parameters

Your C program must be invoked exactly as follows:

```
sh prompt> ./wserver [-d basedir] [-p port] [-t threads] [-b buffers]
```

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

- **basedir**: this is the root directory from which the web server should operate. The server should try to ensure that file accesses do not access files above this directory in the file-system hierarchy. Default: current working directory (e.g., `.`).
- **port**: the port number that the web server should listen on; the basic web server already handles this argument. Default: 10000.
- **threads**: the number of worker threads that should be created within the web server. Must be a positive integer. Default: 1.

- **buffers**: 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. Default: 1.

For example, you could run your program as: `prompt> ./wserver -p 8003 -t 8 -b 16`

In this case, your web server will listen to port 8003, create 8 worker threads for handling HTTP requests, allocate 16 buffers for connections that are currently in progress (or waiting).

5.6 Additional Useful Reading

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()`. To find information on these library routines, read the man pages (RTFM).

5.7 Sample Execution

You can run the server by opening folder in vscode and navigate to `Deliverable1` folder. Right now all the deliverables have same code, you'll implement each in its corresponding folder. Now compile the program by:

`make`

Run the server by:

`./wserver -p 2022 -t 2 -b 5`

Now open your web browser and go the address:

`127.0.0.1:2022`

It'll simply display a welcome message. This is a short job, doesn't require a lot of time on server to serve it. For a longer job, you can request this page:

`127.0.0.1:2022/spin.cgi?10`

It'll take 10 seconds on server to serve this job. Change the value after ? to spin the server for desired time.

5.8 Acknowledgement

This homework is adapted from **OSTEP: Operating Systems Three Easy Parts** by Remzi et al. at University of Wisconsin-Madison.