Instructor: Ashvin Goel

Course Number: ECE344

Operating Systems

ECE344, FALL 2016
UNIVERSITY OF TORONTO

LAB 4: A MULTI-THREADED WEB SERVER

Due Date: Nov 7, 2016, 5 pm

In this lab, you will be developing a multi-threaded web server, and evaluating its performance. To simplify the 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

In this section, we provide a brief overview of how a simple 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. If you are really interested in the full details of the HTTP protocol, you can read the HTTP specification, but we do not recommend it for this project.

Most 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 an executable file to be run and its output returned, then this is called dynamic content. In this lab, our web server will only handle static content.

Each file requested from the server has a unique name known as a URL (Universal Resource Locator). For example, the URL `http://www.eecg.toronto.edu:80/Welcome.html` identifies an HTML file called “index.html” on Internet host “www.eecg.toronto.edu” 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.

An HTTP request (from the web browser to the server) consists of a request line, followed by zero or more request header lines, 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). The `version` indicates the version of the HTTP protocol that the web client is using (e.g., HTTP/1.0 or HTTP/1.1). The request headers define various parameters of the

request such as the type of browser (user agent) making the request. Each header is a colon-separated name-value pair in clear-text string format. The request line and other header fields must each end with <CR><LF> (that is, a carriage return character followed by a line feed character). The empty line must consist of only <CR><LF>.

An HTTP response (from the server to the browser) is similar; it consists of a response line, zero or more response header lines, 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 and the corresponding messages are 200 for **OK**, 403 for **Forbidden**, and 404 for **Not found**. Two important lines in the response 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 the size of the response body in bytes. The server can add any custom header line.

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.**

SETUP

Add the source files for this lab, available in `webserver.tar`, to your repository, and run `make` in the newly created `webserver` directory.

```
cd ~/ece344
tar -xf /cad2/ece344f/src/webserver.tar
git status # should say that "webserver/" directory is untracked
git add webserver
git commit -m "Initial code for Lab 4"
git tag Lab4-start
cd webserver
make
```

The make command will create four executables called `server`, `client_simple`, `client` and `fileset`. We will describe the `server` program here, and then describe the other three programs later.

The `server` program we have provided you is a basic, single-threaded server. When you run it, you need to specify the port number that it will listen on. You should specify port numbers that are greater than about 2000 to avoid active ports. 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 `ug205.eecg` and use port number 2003. Copy your favorite HTML file, called `favorite.html` to the `webserver` directory. Then, you can view this file from a web browser (running on the same or a different machine), by using the url: `ug205.eecg.toronto.edu:2003/favorite.html`. Note that you will need to run the web browser on one of the lab machines.

We are providing you with a bare bones, minimal web server. For example, the web server does not handle any HTTP requests other than GET. Also, it does not support running CGI programs. This web server is also not very robust; for example, if a web client closes its

connection to the server, it may crash. We do not expect you to fix these problems!

The helper functions are simply wrappers for system calls that check the error codes of the system calls and immediately terminate if an error occurs. One should **always check error codes!** However, many programmer don't like to do it because they believe that it makes their code less readable; the solution is to use these wrapper functions.

MULTI-THREADED WEB SERVER

The basic web server that we have 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 a problem for two reasons. First, if your server is running on a multi-core processor or a multi-processor machine, a single thread can only use a single processor core, under-utilizing the machine. Second, if the current HTTP request is for a file that is resident only on disk (i.e., it is not in memory), then the request will be delayed until the file can be fetched from disk, during which time all the cores will remain completely idle.

In this lab, you will address the limitations of the basic web server by making it multi-threaded. The simplest approach to building a multi-threaded server is to spawn a new thread for serving each new HTTP request. The OS will then schedule these threads according to its own policy. The advantage of creating these threads is that 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 to fetch the file) 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 designing a multi-threaded server is to create a **fixed-size pool** of worker threads when the web server is first started. With this 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 this lab, you will implement this web server design.

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 socket descriptor for this connection into a fixed-size request buffer. The number of elements in the buffer is also specified on the command line. Note that the web server we have provided uses a single thread that accepts a connection, and then immediately handles the connection by reading the request and sending the response. In your web server, this (master) thread should place the connection descriptor into the fixed-size buffer and

return to accepting more connections, i.e., it should not read the request or perform any request processing.

Each worker thread wakes up when there is an HTTP request in the queue. Once the worker thread wakes, it processes an HTTP request by performing a read on the network descriptor, obtains the specified content by reading the file that is requested, and then returns the content to the client by writing to the descriptor. The worker thread then waits for another HTTP request.

When there are multiple HTTP requests available, the requests are handled in FIFO order. Hence, when a worker thread wakes up, it handles the first request (i.e., the oldest request) in the buffer. Note that the HTTP requests will not necessarily finish in FIFO order. The order in which the requests complete will depend upon how long it takes to process the request and also on how the OS schedules the active threads.

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.

RUNNING THE WEB SERVER

The web server we provide should be invoked as follows:

```
./server port nr_threads max_requests max_cache_size
```

The command line arguments are:

- **port:** the port number that the web server should listen on.
- **nr_threads:** the number of worker threads that should be created within the web server. This number must be non-negative (>= 0). When it is zero (zero worker threads), the basic web server that we have provided is run. When it is positive, then your implementation should create worker threads.
- **max_requests:** the number of request connections that can be accepted at one time. This parameter specifies the size of the fixed-size buffer, and it applies only for the multi-threaded web server. It must be a positive integer. Note that it is not an error for more or less threads to be created than the buffer size.
- **max_cache_size:** we will use this parameter in the next lab. For the moment, you can run the server with a value of 0 for this parameter.

You can run the initial web server that we have provided on port 5003 as follows:

```
./server 5003 0 0 0
```

After you have implemented the multi-threaded web server, you could run your server as follows:

```
./server 5003 8 16 0
```

Then your server should create 8 worker threads for handling HTTP requests, and allocate 16 buffers so that up to a maximum of 16 connections can be waiting to be served. Note that a request that is being served by a worker thread should no longer be in the request queue.

RUNNING THE CLIENT

We have provided a simple browser client program called `client_simple`. Go ahead and use it as follows:

```
./client_simple www.ece.toronto.edu 80 /index.html
```

The client simple program will allow you to fetch a single file from any web server. If you wish to use it for your own web server, you can use it as follows:

```
./client_simple localhost 5003 /favorite.html
```

Here `client_simple` is run on the same machine as your server and hence it contacts the `localhost` machine. The port on which your server is assumed to be listening is 5003. The `favorite.html` file should lie in the `webserver` directory.

The `client_simple` program can be used test your web server. But it is hard to measure the performance of your web server with this simple program. Instead, we have provided you the `client` program for stress testing your web server. To run the `client` program, you will first need to generate a set of files that will be requested from the server. To do so, run the following command:

```
./fileset
```

The `fileset` program generates a set of files in a directory. By default, it generates 256 files in the `fileset_dir` directory. The mean size for each file is 3 blocks, or 12 KB. It also generates an index file called `fileset_dir.idx`. This index file lists the total number of files in the file set in the first line. After that, it lists one file per line. For each file, it lists the name of the file, a simple checksum for the file, and the size of the file. You can find out the parameters of the `fileset` program by running `fileset` with the `--help` option.

```
./fileset --help
```

The file sizes generated by the `fileset` program are distributed according to a Pareto distribution, which is a a long tail distribution. For the `fileset` program, this means that while most files are relatively small and close to the mean file size, there will be some files with very large sizes. These file size distributions are often found in practice. For example, most of the files in a system will be small but a few files are very large (e.g., video files, virtual machine images, etc.).

After generating the fileset, you can run the `client` program as follows:

```
./client [-t] host port nr_times nr_threads fileset_dir.idx
```

The `host` argument is the name of the host on which the server is running. The `port` argument should be the same as the port at which the web server is listening for connections. We have provided you a

multi-threaded `client` program, that runs with `nr_threads`. Each thread runs a loop `nr_times`, requesting a random file from the file set, once per loop. So the total number of file requests made by the client is `nr_times * nr_threads`. To request one file, make both these arguments have the value of 1. The `fileset_dir.idx` argument is the index file generated by the `fileset` program that provides the file set information.

The client requests files randomly using a self-similar process. In our case, this means that the client requests files with a lower index value more often. In particular, the top 20% of the files in file set are requested 80% of the time. Again, this behavior is typical of real-world web servers (many more people visit Shakira's facebook page than your home page).

When the `client` program is run, it will output the contents of the files in the file set to the terminal. This will be a lot of output, and the output has (you guessed that) random printable ascii characters. You can avoid seeing this output by piping the output to `/dev/null` as follows:

```
./client host port nr_times nr_threads fileset_dir.idx > /dev/null
```

Alternatively, you can use the client's `[-t]` option, which does not print the file output. Instead, it outputs in a single line, the time it took to run the client. This will allow you to time the client run.

How do we know that the server is working correctly? The server generates the file size and the file checksum in the response header lines, as shown in an example header below:

```
Header: HTTP/1.0 200 OK
Header: Server: OS Web Server
Header: Content-Type: text/plain
Header: Content-Length: 619
Header: Content-Csum: 49888
```

Recall the index file generated by the `fileset` program. This file also has the file length and checksum fields for each file. The client performs two types of checks. First, it checks that the length and the checksum fields in the index file match the values in the response header lines shown above. Second, it also checks that the file content that it receives (the response body) has the same correct length and checksum values. So if your web server is not working correctly, these checks will not pass, and your client will crash.

SOLUTION REQUIREMENTS

You are required to use **condition variables** for synchronization. **If your implementation performs any busy-waiting (or spin-waiting), you will be heavily penalized.**

The rest of the requirements are described earlier, when we described the design of the multi-threaded web server.

HINTS AND ADVICE

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

- **server.c:** Contains the main() function for the web server.
- **server_thread.c:** Currently, this file contains the main routine for serving web requests. This routine, invoked by the main server thread in the basic web server, calls various functions in `request.c`, described below. **This is the only file that you should modify for this lab.**
- **request.c:** Performs most of the work for handling HTTP requests, including reading requests from a connection, and sending the response back. All procedures in this file begin with the string “request”.
- **common.c:** Contains wrapper functions for the system calls invoked by the various programs.
- **client.c:** Contains the main() function and the support routines for our stylized web client.
- **client_simple.c:** Contains the main() function and the support routines for a simple web client that requests a single file.
- **fileset.c:** Contains the code for generating the file set.

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 or with the `client_simple` program that we have provided to you. Request a single file using the client code, and see the output of the server and the client. 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 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`, and `pthread_cond_signal`. To find information on these library routines, read their manuals on the web.

You are encouraged to reuse *your own* code that you might have developed in the previous labs or in previous courses to handle things such as queues, hashing, etc.

Feel free to consult the instructor's notes on the solutions to the producer-consumer problem.

NOTE ON SECURITY

It is worthwhile to be a little careful with security during this project. The server program that we have provided does not serve files that have absolute paths, or a **..** in the file name. Both would allow others to explore any files in your home directory to which you have access (and they normally don't). We also disallow serving “.c” and “.h” source files. Where are these checks implemented? Even so, it will be a good idea to avoid running your server for a long time.

TESTING YOUR CODE

To help you test your code better, we have provided several scripts, as described below.

run-one-experiment:

This script runs the `server` program, and then it runs the `client` program 5 times, and provides the average and the standard deviations of the run times of the client. All the client run times are recorded in the `run.out` file. Read the beginning of this file to see how it should be invoked. This script can take up to 1-2 minutes to run.

run-experiment:

This script runs the `run-one-experiment` script while varying various server parameters. Read the beginning of this file to see how it should be invoked. The script will generate two output files, `plot-threads.out` and `plot-requests.out`. These files contain the client run times when the number of threads and outstanding requests are varied for the server. This script can take 10-15 minutes to run, so use it when you are close to finishing the lab.

plot-experiment:

This script plots the output files generated by `run-experiment`. The plots are available in `plot-threads.pdf` and `plot-requests.pdf`. These plots will show you how the performance of your server changes with different number of threads and requests. Think about what you expect your server performance to be as you vary these server parameters, and whether these plots match your expectations. If you are running these scripts from a remote machine, then you will need to download the PDF files to view them on your machine.

You can test your entire code by using our auto-tester program at any time by following the testing instructions. For this assignment, the auto-tester program runs the `run-experiment` and the `plot-experiment` scripts. Then it estimates whether the performance of your server matches our expectations and provides you marks appropriately.

Sometimes when you run the server on a loaded machine, e.g., when others are also testing their programs, then your performance numbers will vary and the tester may give lower marks than you expect. It will be best to run these experiments on lightly-loaded machines. You can see the load on a machine by looking for `load average` in the output of the `top` program. A lightly-loaded machine will have a `load average` close to 0. A heavily-loaded machine will have a `load average` that close to 1 or higher. The `load average` will go up after you start running your experiment. It may also go up because others start running their experiments.

We will ensure that when we perform the marking then your code will be run on a lightly loaded machine.

USING GIT

You should only modify the following files in this lab.

```
server_thread.c
```

You can find the files you have modified by running the `git status` command.

You can commit your modified files to your local repository as follows:

```
git add server_thread.c
git commit -m "Committing changes for Lab 4"
```

We suggest committing your changes frequently by rerunning the commands above (with different meaningful messages to the commit command), so that you can go back to see the changes you have made over time, if needed.

Once you have tested your code, **and committed it** (check that by running `git status`), you can tag the assignment as done.

```
git tag Lab4-end
```

This tag names the last commit, and you can see that using the `git log` or the `git show` commands.

If you want to see all the changes you have made in this lab, you can run the following `git diff` command.

```
git diff Lab4-start Lab4-end
```

More information for using the various git commands is available in the Lab 1 instructions.

CODE SUBMISSION

Make sure to add the `Lab4-end` tag to your local repository as described above. Then run the following command to update your remote repository:

```
git push
git push --tags
```

For more details regarding code submission, please follow the lab submission instructions.

Please also make sure to test whether your submission succeeded by simulating our automated marker.
