

CSCI 4061: Introduction to Operating Systems

Project 3: Multi-Threaded Web Server

Instructor: Jon Weissman

Due: 5 pm, November 25 (Wed.), 2020

1. Overview

The purpose of this lab is to construct a multi-threaded web server using POSIX threads (pthreads) in the C language to learn about thread programming and synchronization methods. Your web server should be able to handle these file type: HTML, GIF, JPEG, TXT. and of any arbitrary size. It should handle a limited portion of the HTTP web protocol (namely, the GET command to fetch a web page / files). We will provide pieces of code (some already compiled into object files, and some source) that will help you complete the lab.

2. Description

Your server will be composed of two types of threads: dispatcher threads and worker threads. The purpose of the dispatcher threads is to repeatedly accept an incoming connection, read the client request from the connection, and place the request in a queue. We will assume that there will only be one request per incoming connection. The purpose of the worker threads is to monitor the request queue, retrieve requests and serve the request's result back to the client. The request queue is a bounded buffer and will need to be properly synchronized.

You will use the following functions which have been precompiled into object files that we have provided (full documentation of these functions has been provided in util.h. More will be said below about how to use these functions). Since the server will continuously wait for client requests, you will need to ^C it to terminate. You should make the server gracefully terminated (see section 8 for details.)

You can assume that all the below functions are thread safe.

```
void init (int port); // run ONCE in your main thread
```

Each of the next two functions will be used in order, in a loop, in the dispatch threads:

```
int accept_connection(void); // it returns fd
```

```
int get_request(int fd, char *filename); // fd is a return value  
                                         from accept_connection()
```

These functions will be used by the worker threads after handling the request

```
int return_result(int fd, char *content_type, char *buf, int  
numbytes); // for successful response
```

```
int return_error(int fd, char *buf); // for error response
```

3. Thread pool

Your server should create a fixed pool of dispatcher and worker threads when the server starts. The dispatcher thread pool size should be `num_dispatch` and the worker thread pool size should be `num_workers` (See section 8 for details). **You should make all threads detachable.**

Dynamic worker thread pool [Extra Credit A]

You can implement a dynamically varying worker thread pool size instead of having a fixed number of workers using the `dynamic_flag` option (See section 8 for details). We expect you to be creative and come up with your own policy to dynamically increase/decrease the pool size depending on the server load. But note that, there needs to be at-least one worker thread at any point of time. You must explain the policy you implemented in the ReadMe file. And there should be proper logging in terminal when the threads are created / deleted depending on your policy in the following format:

Created `<num_of_threads_created>` worker threads because `<of some reason>`.

Removed `<num_of_threads_removed>` worker threads because `<of some reason>`.

For example,

Created 10 worker threads because the server load increased by 10%

Deleted 5 worker threads because the server load decreased by 5%

Hints for implementation:-

1. You can think in terms of finding a relation between pending requests and number of worker threads. This is just one way of thinking. You can come up with your own creative policy!
2. You can keep a separate thread for implementing your policy.

4. Incoming Requests

An HTTP request has the form: `GET /dir1/subdir1/.../target_file HTTP/1.1` where `/dir1/subdir1/.../` is assumed to be located under your web tree root location. Our `get_request()` automatically parses this for you and gives you the `/dir1/subdir1/.../target` file portion and stores it in the filename parameter. Your web tree will be rooted at a specific location, specified by one of the arguments to your server (See section 8 for details). For example, if your web tree is rooted at `/home/user/joe/html`, then a request for `/index.html` would map to `/home/user/joe/html/index.html`. You can `chdir()` into the Web root to retrieve files using relative paths.

5. Returning Results

You will use `return_result()` to send the webpage data back to the web browser from the worker threads provided the file was opened and read correctly, or the data was found in the cache (see section 6). If there was any problem with accessing the file, then you should use `return_error()` instead. Our code will automatically append HTTP headers around the data before sending it back to the browser. Part of returning the result is sending back a special parameter to the browser: the content-type of the data. You may make assumptions based on the extension of the files as to which content-type they are:-

- Files ending in `.html` or `.htm` are content-type “text/html”
- Files ending in `.jpg` are content-type “image/jpeg”
- Files ending in `.gif` are content-type “image/gif”
- Any other files may be considered, by default, to be of content-type “text/plain”.

6. Caching [Extra Credit B]

To improve runtime performance, you need to implement caching which stores cache entries in memory for faster access. When a worker serves a request, it will look up the request in the cache first. If the request is in the cache (Cache HIT), it will get the result from the cache and return it to the user. If the request is not in the cache (Cache MISS), it will get the result from disk as usual, put the entry in the cache and then return result to the user. The cache size can be defined by an argument and you need to log information about the cache (HIT or MISS) (see section 7 for more details). How to implement caching is totally up to you. You can implement any cache replacement policy like random, FIFO, LRU or LFU policy to choose an entry to evict when the cache is full. You must explain the policy you implemented in the ReadMe file.

Please make sure to dynamically allocate the memory when adding entries to cache as the file sizes can be different. Also, free the memory blocks which will not be needed so as to avoid memory leaks.

7. Request Logging

From the worker threads, you must carefully log each request (normal or error-related) to a file called “web_server_log” and also to the terminal (stdout) in the format below. The logfile should be created in the same directory where the final executable “web_server” exists. You must also protect the log file from multiple threads attempting to access it simultaneously.

[threadId][reqNum][fd][Request string][bytes/error][Cache HIT/MISS]

- **threadId** is an integer from 0 to `num_workers - 1` indicating thread ID of request handling worker.

- **reqNum** is total number of requests a specific worker thread has handled so far, including current request.
- **fd** is the file descriptor given to you by `accept_connection()` for this request
- **Request string** is the filename buffer filled in by the get request function
- **bytes/error** is either the number of bytes returned by a successful request, or the error string returned by return error if an error occurred.
- **Cache HIT/MISS** is either “HIT” or “MISS” depending on whether this specific request was found in the cache or not. If you don’t attempt caching implementation (Extra Credit B), you don’t need to log it.

The log (in the “web_server_log” file and in the terminal) should look something like the example below.

```
[8][1][5][image/jpg/30.jpg][17772][MISS]
[9][1][5][image/jpg/30.jpg][17772][HIT]
[2][1][5][image/jpg/282.jpg][Requested file not found.][MISS]
```

8. Graceful Server Termination

The server should be gracefully terminated. When the server gets **SIGINT (^C) signal**, you need to **print “the number of pending requests in the request queue” to the terminal, close logfile, and remove cache if you attempt caching (extra credit B) before the main function finishes.**

9. How to run the server

Your server should be run as:

```
% ./web_server port path num_dispatcher num_workers dynamic_flag qlen cache_entries
```

The server will be configurable in the following ways:

- **port** number can be specified (you may only use ports 1025 - 65535 by default). When you send wget request (Section 11) to the server, the port number should be the same with this.
- **path** is the path to your web root location from where all the files will be served
- **num_dispatcher** is how many dispatcher threads to start up
- **num_workers** is how many worker threads to start up
- **dynamic_flag** indicates whether the worker thread pool size should be static or dynamic. (0: when you didn’t attempt extra credit A, 1: when you attempted extra credit A)
- **qlen** is the fixed, bounded length of the request queue
- **cache_entries** is the number of entries available in the cache. When you didn’t attempt extra credit B, set it to 0.

10. Provided Files and How to Use Them

We have provided many functions which you must use in order to complete this assignment. We have handled all of the networking system calls for you. We have also handled the HTTP protocol parsing for you. Some of the library function calls assumes that the program has “chdir”ed to the web tree root directory. **You need to make sure that you chdir to the web tree root somewhere in the beginning.**

We have provided a makefile you can use to compile your server. Here is a list of the files we have provided.

1. **server.c**: You only need to **modify this file to implement a server.**
2. **makefile**: You can use this to compile your program using our object files, or you can make your own. You can study this to see how it compiles our object code along with your server code to produce the correct binary executables.
3. **util.h**: This contains a very detailed documentation of each function that you must study and understand before using the functions.
4. **util.o**: This is the compiled code of the functions described in util.h to be used for the web server. Compile this into your multi-threaded server code and it will produce a fully-functioning web server.
5. **testing.zip**: This file includes images, texts and url files to test your server. See section 10 for more detailed information.
6. **web_server_sol**: This is a solution webserver executable. You can test it by sending `wget` requests to this server. This solution includes the implementation of the dynamic worker thread pool (extra credit A) and caching (extra credit B). Note that, therefore, the logs might be different from yours because of different algorithms used for extra credits. Use it after running “`chmod +x web_server_sol`”.

11. How to test your server

We will test your server with “`wget`”, the non-interactive webpage downloader. After you run the server, open a new terminal to test the server. You can try to download a file using this command:

```
-> wget http://127.0.0.1:9000/image/jpg/29.jpg
```

Please note that 127.0.0.1 means localhost, the port number used in `wget` request should match the port number specified to the `web_server`.

This command will try to download the file at `root/image/jpg/29.jpg`. **If it failed to download the file for some reason, it will show an error message.** You can also test your server with any web browser (Internet Explorer, Chrome, Firefox, and so on). We will provide “`testing.zip`” to make testing the server easier. Once you extract it, you can find an instruction file “`how_to_test`” which explains how to use and test your program with these files. The server should be able to serve requests from other machines, so you

need to run the server on a machine and run “wget” command on another machine to test your server (use the server's IP address instead of 127.0.0.1). Note, if you try to connect to the server which runs on CSELabs machine with your own machine (laptop), it may not be able to connect to the server because CSELabs machines are protected by a firewall. If you want to test it using different two machines, you can use two CSELabs machines; one for server and another one for client.

Testing Concurrency: Bash has a nifty command called xargs, which allows a set of arguments to be piped to a command such that multiple executions will be run concurrently.

The format of the command is 'xargs -n num_args -P num_procs cmd'

So, for our purposes, the command “cat urls | xargs -n 1 -P 8 wget” will run wget 8 times simultaneously (-P 8) with 1 argument each time (-n 1) from the pipe produced by cat urls.

NOTE: If -P is given an argument that is smaller than the number of args in the pipe, then multiple sets of size P will be run, with each set being concurrent.

Ex. If I had 10 arguments in the pipe, and I set -P 5, 2 sets of 5 concurrent processes will run. This can be used to test that your server really does permit concurrent execution of multiple requests.

12. Simplifying Assumptions

- The maximum number of dispatcher threads will be 100.
- The maximum number of worker threads will be 100.
- The maximum length of the request queue will be 100 requests.
- The maximum size of the cache will be 100 entries.
- The maximum length of filename will be 1024.
- Any HTTP request for a filename containing two consecutive periods or two consecutive slashes (“.” or “//”) will automatically be detected as a bad request by our compiled code for security.

13. Documentation

You must include a README containing the following information:

- Your group ID and each member’s name and X500.
- How to compile and run your program.
- A brief explanation on how your program works.
- Indicate which extra credit your group implements [Extra credit A/ Extra credit B/ Both/ None]
- Explanation of your policy to dynamically change the worker thread pool size.
- Explanation of caching mechanism used.
- Contributions of each team member towards the project development.

- Within your code you should use one or two sentences to describe each function that you wrote. You might want to comment portions of your code to increase readability.

At the top of your README file, please include the following comment:

```
/* Test machine: CSELAB_machine_name
 * Name: <full name1>, [full name2, ... ]
 * X500: <X500 for first name>, [X500 for second name, ...] */
```

14. Deliverables

One student from each group should upload to Canvas, a zip file containing the following files:

1. Source codes (all files and folders related to .c and .h files, and Makefile)
2. A README file
3. Interim report

15. Interim Evaluation

Your group need to submit 1-page interim report and your server.c source file as a zip **by 5 PM, Nov. 18.**

- server.c: We will check your implementation of **the ability to serve a single request** without synchronization, caching, and logging. The minimum scopes that you should implement are:
 - Dispatch and worker thread creation
 - Request queue handling; dispatch thread enqueues a request to the request queue, and worker thread dequeues the request from the request queue.
 - Response handling; the server sends either error response or success response. An error response can be sent by simply calling `return_error()` in worker thread. If your group has further progress in worker implementation, the server can send a success response by calling `return_result()`.
- Interim report (max 1-page): You need to include the followings:
 - Group members' name and X500
 - Screen capture of the terminal messages showing `wget` request and its response.
 - Plan for work distribution among group members
 - Plan for extra credit A and B. If you are planning to implement them, please state your idea/algorithm for each extra credit options.

16. Grading (Tentative)

5% README

10% Documentation within code, coding, and style: indentations, readability of code, use of defined constants rather than numbers

15% Interim evaluation

70% Correctness, error handling, meeting the specification. Broken down as follows:

- 20% for handling one request successfully.
- 20% for handling multiple requests.
- 20% for handling multiple concurrent requests (test with xargs).
- 10% for handling graceful server termination.

Extra 10% for implementing dynamic worker thread pool (extra credit A)

Another extra 10% for implementing caching (extra credit B).

We will use the GCC version installed on the CSELabs machines(i.e. 9.3.0) to compile your code. Make sure your code compiles and run on CSELabs.

Please make sure that your program works on the CSELabs machines e.g., KH 4-250 (csel- kh4250-xx.cselabs.umn.edu). You will be graded on one of these machines.