CSCI 4061: Introduction to Operating Systems

Assignment 4: Multi-Threaded Web Server

Due: December 12 11:55pm.

1 Overview

The purpose of this lab is to construct a multithreaded web server using POSIX threads (pthreads) in C to learn about thread programming and synchronization methods. Your web server will be able to handle any type of file: HTML, GIF, JPEG, TXT, etc. It will handle a limited portion of the HTTP web protocol (namely, the GET command to fetch a web page). 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 request from the connection, and place the request in a queue for a worker thread to pick up and serve, in a loop. 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, pick up requests from it, and serve those requests back to the client. The request queue is a **bounded buffer** and will need to be properly synchronized (using CVs).You will use the following functions which have been pre-compiled 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 block, you will need to ^C it to terminate.

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); **// this is thread-safe**
int get_request (int fd, char *filename);

// 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);
(or int return_error(int fd, char *buf);) for error.

3 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 function automatically parses this for you and gives you the /dir1/subdir1/.../target file portion and stores it in filename parameter. Your web tree will be rooted at a specific location, specified by one of the arguments to your server. 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.

4 Returning Results

You will use return_result () to send the 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 (if you have implemented a cache. See section 5). 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 extensions 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”
• Files that have not been classified in the above categories may be considered, by default, to be of content-type “text/plain”.

5 Caching (Optional for Extra credit: +5)

To improve performance, you can implement caching which stores cache entries in memory. When a worker serves a request, it will look a request up from 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 and return it to the user, then put the new mapping in the cache. Since implementing caching is an optional requirement, you will get extra credit if you implement caching. **How to implement caching is totally up to you**. The cache size can be defined by an argument and you need to log information about the cache (HIT or MISS) with time (see section 6 for more detail). You only get the extra credit IF everything is working in the cache.

6 Request Logging

From the worker threads, you must carefully log each request (normal or error-related) to a file called “**web_server_log”** in the format below. **You must also protect the log from multiple**

threads attempting to access it simultaneously.
[ThreadID#][Request#][fd][Request string][bytes/error][(optional)time][(optional) Cache HIT/MISS]
Where:
• **ThreadID#** is an integer from 0 to num_workers -1 indicating the worker-thread ID of the handling thread.
• **Request#** is the total number of requests this specific worker thread has handled so far, including this 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.
• **time (Optional)** is the service time of a request (Only when you implement the cache)
• **Cache HIT/MISS (Optional)** is either "HIT" or "MISS" depending on whether this specific request was found in the cache or not. (Only when you the implement cache)

The log (on "web_server_log" file) will look like below
[6][1][5][/image/jpg/31.jpg][17772]
[7][1][6][/image/jpg/31.1jpg][File not found.]

Or if you implement caching
[8][1][5][/image/jpg/30.jpg][17772][200ms][MISS]
[9][1][5][/image/jpg/30.jpg][17772][3ms][HIT]

The server will be configurable in the followings ways:
• **port** number can be specified (you may only use ports 1025 - 65535 by default)
• **path** is the path to your web root location where all the files will be served from
• **num_dispatcher** is how many dispatcher threads to start up
• **num_workers** is how many worker threads to start up
• **qlen** is the fixed, bounded length of the request queue
• **cache_entries (Optional)** is the number of entries available in the cache (an alternative way to do this would be to specify the maximum memory used by the cache, but we are just using a limit on the number of entries)
Your server should be run as:
% **web_server port path num_dispatch num_workers qlen [cache_entries]**

7 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.tgz :** This file includes images, texts and url files to test your server. See section 8 for more detail information.

8 How to test your server

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

-> wget http://127.0.0.1:9000/image/jpg/29.jpg (**Please note that 127.0.0.1 means localhost**)
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 browsers (Internet explorer, Chrome, Firefox, and so on). We will provide “testing.tar.gz” 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 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.

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 **echo $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 **echo $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.

9 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. (Only when you implement the cache).
• 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 reasons.

10 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.
- Within your code you should use one or two sentences to describe each function that you write. You do not need to comment every line of your code. However, you might want to comment portions of your code to increase readability.
- At the top of your README file, please include the following comment:

/* CSci4061 Fall 2016 Assignment 4
* Name: <full name1>, <full name2>
* X500: <X500 for first name>, <X500 for second name> */

11 Deliverables

1. Files containing your code
2. A README file (readme and c code should indicate this is assignment 4). **README should contain information on every group member's individual contributions.** All files should be compressed into a single tar file, named assignment4_group<your group>.tar and submitted through Moodle. This is your official submission that we will grade. We will only grade the most

recent and on time submission. Failing to follow the submission instruction may result in a 5% deduction.

11 Grading

10%	README, Documentation within code, coding and style (indentation, readability of code, use of defined constants rather than numbers).
90%	Correctness, error handing, meeting the specification.

Grading will be done on the CSELabs configuration described on the web page. All project policies enforced on other projects should be assumed to apply unless stated hereafter on the moodle site.