Assignment 2: multi-threaded HTTP server with logging
CSE130-01 Fall 2019
Due: Thursday, November 21 at midnight

Goals

The goals for Assignment 2 are to modify the HTTP server that you already implemented to have two additional features: multi-threading and logging. Multi-threading means that your server must be able to handle multiple requests simultaneously, each in its own thread. Logging means that your server must write out a record of each request, including both header information and data (dumped as hex). You'll need to use synchronization techniques to service multiple requests at once, and to ensure that entries in the log aren't intermixed from multiple threads

As usual, you must have source code, a design document and writeup along with your README.md in your git repository. Your code must build httpserver using make.

Design document

Before writing code for this assignment, as with every other assignment, you must write up a design document. Your design document must be called DESIGN.pdf, and must be in PDF. You can easily convert other document formats, including plain text, to PDF. Scanned-in design documents are fine, as long as they're legible and they reflect the code you actually wrote.

Your design document should describe the design of your code in enough detail that a knowledgeable programmer could duplicate your work. This includes descriptions of the data structures you use, non-trivial algorithms and formulas, and a description of each function with its purpose, inputs, outputs, and assumptions it makes about inputs or outputs

Since a lot of the system in Assignment 2 is similar to Assignment 1, we expect you're going to "copy" a good part of your design from your Assignment 1 design. This is fine, as long as it's your Assignment 1 you're copying from. This will let you focus on the new stuff in Assignment 2.

For multithreading, getting the design of the program right is vital. We expect the design document to contain discussions of why your system is thread-safe. Which variables are shared between threads? When are they modified? Where are the critical regions? These are some of the things we want to see. Similarly, you should explain why your design for logging each request contiguously actually works.

Program functionality

Your code may be either C or C++, but all source files must have a .cpp suffix and be compiled by clang++ version 7. As before, you may not use standard libraries for HTTP, nor any FILE * or iostream calls except for printing to the screen (e.g., error messages). You may use standard networking (and file system) system calls.

We expect that you'll build on your server code from Assignment 1 for Assignment 2. Remember, however, that your code for this assignment must be developed in asgn2, so copy it there before you start, and make sure you add it to your repository using git add.

Multi-threading

Your previous Web server could only handle a single request at a time, limiting throughput. Your first goal for Assignment 2 is to use multi-threading to improve throughput. This will be done by having each request processed in its own thread. The typical way to do this is to have a “pool” of “worker” threads available for use. The server creates N threads when it starts; N = 4 by default. Your program should take an argument "-N" for the number of threads. For example, -N 6 would tell the server to use 6 worker threads. (Note this is a command-line argument so it is written when you run your server: "./httpserver -N 6" for example).

Each thread waits until there is work to do, does the work, and then goes back to waiting. Worker threads may not “busy wait” or “spin lock”; they must actually sleep by waiting on a lock, condition variable, or semaphore. Each worker thread does what your server did for Assignment 1 to handle client requests, so you can likely reuse much of the code for it. However, you’ll have to add code for the dispatcher to pass the necessary information to the worker thread. Worker threads should be independent of one another; if they ever need to access shared resources, they must use synchronization mechanisms for correctness. Each thread may allocate up to 16 KiB of buffer space for its own use; this includes space for send/receive buffers as well as log entry buffers (see below).

A single “dispatch” thread listens to the connection and, when a connection is made, alerts (using a synchronization method such as a semaphore or condition variable) one thread in the pool to handle the connection. Once it has done this, it assumes that the worker thread will handle the connection itself, and goes back to listening for the next connection. The dispatcher thread is a small loop (likely in a single function) that contacts one of the threads in the pool when a request needs to be handled. If there are no threads available, it must wait until one is available to hand off the request. Here, again, a synchronization mechanism must be used.

You will be using the POSIX threads library (libpthreads) to implement multithreading. There are a lot of calls in this library, but you only need a few for this assignment. You’ll need:

- pthread_create (...)
- Condition variables, mutexes and/or semaphores. You may use any or all, as you see fit. See pthread_cond_wait(...), pthread_cond_signal(...), pthread_cont_init(....), pthread_mutex_init(...), pthread_mutex_lock(...), pthread_mutex_unlock(...), sem_init(...), sem_overview(...), sem_wait(...), and sem_post(...) for details.

Since your server will never exit, you don’t need pthread_exit, and you likely won’t need pthread_join since your thread pool never gets smaller. You may not use any synchronization primitives other than (basic) semaphores, locks, and condition variables. You are, of course, welcome to write code for your own higher-level primitives from locks and semaphores if you want.

There are several pthread tutorials available on the internet, including this one. We will also cover some basic pthreads techniques in section.

Logging Requests

If the "-l log_file" command-line argument is provided to your httpserver program, it must log each request it gets from the client to the file log_file (obviously, log_file is whatever name you provide on the command line). If the -l option isn't provided, no logging is done.

A log record looks like this:

```
PUT abcdefghij0123456789abcdefg length 36
00000000 65 68 6c 6c 3b 6f 68 20 6c 65 6f 6c 61 20 61 67 6e 69 20 3b
00000020 65 68 6c 6c 20 6f 54 28 65 68 43 20 72 61 29 73
========
```

The first line contains the operation (PUT or GET), along with the file name and the length of the data. You'll know this before you receive all the data from the Content-Length header, or before you send the data from the information you're going to print in the response header. The first line is followed by lines that start with a zero-padded byte count, followed by up to 20 bytes printed as hex numbers. Your program can use +sprintf() to format each line into a buffer, and then write out the buffer to the file as it gets full. Lines in the log end in \n, not \r\n, so (for example) the second line (beginning with 00000000) contains 8+20×3+1 = 69 characters. The log entry is terminated by "========\n".

If the server returns an error response code, the log record looks like this:

```
FAIL: GET abcd HTTP/1.1 --- response 400\n
========
```

The record starts with "FAIL:", followed by the first line of the header (without the \r\n at the end), followed by " --- response XXX\n", where XXX is the response code (without the explanation) that your server returned.

Each request must be logged contiguously. This can be tricky, since there are multiple threads writing to the log file at the same time, which might cause the log entries to get interleaved. To avoid interleaving requests in the log file, a thread should "reserve" the space it's going to use, and use pwrite(2) to write data to a specific location in the file. Your program will need to use synchronization between threads to reserve the space, but should need no synchronization to actually write the log information, since no other thread is writing to that location. Remember, if you know the length of the data being sent or received, you can calculate exactly how much space you need for the request log entry.

README and Writeup

Your repository must also include a README file (README.md) and writeup (WRITEUP.pdf). The README may be in either plain text or have MarkDown annotations for things like bold,

italics, and section headers. The file must always be called README.md; plaintext will look "normal" if considered as a MarkDown document. You can find more information about Markdown at https://www.markdownguide.org.

The README.md file should be short, and contain any instructions necessary for running your code. You should also list limitations or issues in README.md, telling a user if there are any known issues with your code.

Your WRITEUP.pdf is where you'll describe the testing you did on your program and answer any short questions the assignment might ask. The testing can be unit testing (testing of individual functions or smaller pieces of the program) or whole-system testing, which involves running your code in particular scenarios.

For this assignment, please answer the following:

• Using either your HTTP client or curl and your original HTTP server from Assignment 1, do the following:

• Place four different large files on the server. This can be done simply by copying the files to the server's directory, and then running the server. The files should be around 4MiB long.

• Start httpserver.

• Start four separate instances of the client at the same time, one GETting each of the files and measure (using time(1)) how long it takes to get the files. Perhaps the best way to do this is to write a simple shell script (command file) that starts four copies of the client program in the background, by using & at the end.

• Repeat the same experiment after you implement multi-threading. Is there any difference in performance?

• What is likely to be the bottleneck in your system? How much concurrency is available in various parts, such as dispatch, worker, logging? Can you increase concurrency in any of these areas and, if so, how?

Submitting your assignment

All of your files for Assignment 2 must be in the asgn2 directory in your git repository. You should make sure that:

- There are no "bad" files in the asgn2 directory (i.e., object files).
- Your assignment builds in asgn2 using make to produce httpserver.
- All required files (DESIGN.pdf, README.md, WRITEUP.pdf) are present in asgn2.

You must submit the commit ID to canvas **before the deadline.**

Hints

• Start early on the design. This program builds on Assignment 1. If you didn't get Assignment 1 to work, please see the course staff ASAP for help getting it to work.

• Reuse your code from Assignment 1. No need to cite this; we expect you to do so.

• Go to section for additional help with the program. This is especially the case if you don't understand something in this assignment!

• You'll need to use (at least) the system calls from Assignment 1, as well as pthread_create and some form of mutual exclusion (semaphores and/or mutexes). You'll also need pwrite(3), which is like write, but takes a file offset.

• Test multi-threading and logging separately before trying them together.

• Aggressively check for and report errors.

• Use getopt(3) to parse options from the commandline. Read the man pages and see examples on how it's used. Ask the course staff if you have difficulty using it after reading this material.

• A sample command line for the server might look like this: ./httpserver -N 8 -l my_log.txt localhost 8888 This would tell the server to start 8 threads, and to write log records to my_log.txt. The server would listen for connections on localhost, port 8888.

Grading

As with all of the assignments in this class, we will be grading you on all of the material you turn in, with the approximate distribution of points as follows: design document (35%); coding practices (15%); functionality (40%); writeup (10%).

If the httpserver does not compile, we cannot grade your assignment and you will receive no more than 5% of the grade.

Extra Credit

We will award 15 points of extra credit to the 5 fastest servers in the class, and 10 points to the next 15 fastest servers. Please note that you must have at least a 70 on the assignment to be eligible. The extra credit may raise your grade above 100 points.

Additional Hints

- The log file after receiving four requests (two PUTs, followed by a failed GET, and a successful GET), may look like this:

```
PUT abcdefghij0123456789abcdefg length 36
00000000 65 68 6c 6c 3b 6f 68 20 6c 65 6f 6c 61 20 61 67 6e 69 20 3b
00000020 65 68 6c 6c 20 6f 54 28 65 68 43 20 72 61 29 73
========
PUT abcdefghij0123456789ab12345 length 3
00000000 65 68 6c
========
FAIL: GET abcd HTTP/1.1 --- response 400\n
========
GET abcdefghij0123456789abcdefg length 0
========
```

- The messages that are sent back to the client are the same as asgn1. The format we show for logging here in this assignment is only to be used when writing to the log file.

- To test multiple clients communicating with your server at the same time, you can do the following:

create a shell script that runs the clients back to back in the background (by putting the "&" sign at the end of the command). so that they are running together. For example, the file shell.sh can include:

```
curl -T t1 http://localhost:8080 --request-target ABCDEFabcdef012345XYZxyz-mm > cmd1.output &
curl -T t1 http://localhost:8080 --request-target ABCDEFabcdef012345XYZxyz-mm > cmd2.output &
```

and then chmod +x shell.sh and finally run it "./shell.sh" which will make the programs run at the same time.

Note that this is just a sample of the tests and we will do more test cases.