Assignment 2: multi-threaded HTTP server with redundancy
CSE130-01 Fall 2020
Due: Tuesday, November 24 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 redundancy. Multi-threading means that your server must be able to handle multiple requests simultaneously, each in its own thread. Redundancy means that each file will have three copies instead of one (if one copy is corrupted, the reader can use the other two copies.) You'll need to use synchronization techniques to service multiple requests at once, and to ensure that different copies of the same file are consistent.

As usual, you must have source code, and a design document 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 redundancy actually works.

Program functionality

Your code may be either C or C++, but all source files must have a .cpp suffix. 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 localhost:8080 -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.

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.
- Note that you cannot use the C++ threads class. You must use libpthreads.

Your httpserver must be able to process requests in parallel using multiple threads. Therefore, there cannot be a single lock for all files. Instead, you must maintain a separate lock for each file. When a file is read or written, then its corresponding lock is acquired. The only time when a global lock can be used is when a new file is added (it does not have a lock yet in your program). In this case (adding a new file), a global lock is acquired, then a lock for the file is created. After the corresponding PUT is performed, both the global and local locks are released.

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.

Redundancy

If the "-r" command-line argument is provided to your httpserver program, it must make three different copies of each file. Specifically, each copy must be in a separate folder (use three folders called "copy1", "copy2", and "copy3", and each folder will contain a copy of all files in the server. These folders must be in the same folder where the httpserver program is.) If the "-r" flag is not provided, then the files must be stored/read from the current folder where the httpserver is.

When a PUT request is received, the httpserver program would write to all three copies of the file. When a GET request is received, the httpserver first checks if the three files are identical. If they are, then the content is returned. If two of them are identical but the third is different, then one of the two identical copies is returned. If all three copies are different from each other, then an error message (error code 500) is returned.

TESTING

Your design document is also 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. In particular, we want you to describe testing that you did for:

- Checking that multiple threads can run and process requests concurrently. Specifically, when you increase the number of threads, does this lead to processing simultaneous requests faster?
- Checking that the redundancy mechanism works. What happens if one file is different from the other two? What happens when all three files are different from each other? What happens if a copy of a file does not exist in some of the "copyX" folders?

QUESTION

For this assignment, please answer the following question in the design document:

- If we do not hold a global lock when creating a new file, what kind of synchronization problem can occur? Describe a scenario of the problem.
- As you increase the number of threads, do you keep getting better performance/scalability indefinitely? Explain why or why not.

README

Your repository must also include a README file (README.md). 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.

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) are present in asgn2.
- Any scripts or files you used for testing (optional)

You must submit the commit ID **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).

• Test multi-threading and redundancy 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 localhost 8888 -N 8 -r This would tell the server to start 8 threads and to write to three copies. 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 (40%); coding practices (10%); functionality (50%).

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

Additional Hints

- The messages that are sent back to the client are the same as asgn1.
- 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 FILENAME00 > cmd1.output &
curl -T t1 http://localhost:8080 --request-target FILENAME00 > 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.