Assignment 2
CSE 130: Principles of Computer System Design, Spring 2020

Due: Monday, May 18 at 9:00PM

Goals

The goals for Assignment 2 are to modify the HTTP server that you already implemented to have three additional features: multi-threading, logging and a health check. Multi-threading will allow your server to handle requests from multiple clients simultaneously, each in its own thread. Logging means that your server must write out a record of each request, including both response information and data (dumped as hex). The health check means that a client should be able to request a special resource and the server will respond with information that can be used to analyze its working performance. For this assignment the information will be derived from the contents of the log. 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 a design document and writeup along with your `README.md` in your `git` repository. Your code must build `httpserver` using `make`.

Programming assignment: multi-threaded HTTP server with logging

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 format. You can easily convert other document formats, including plain text, to PDF.

Your design 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.

Write your design document *before* you start writing code. It'll make writing code a lot easier. Also, if you want help with your code, the first thing we're going to ask for is your design document. We're happy to help you with the design, but we can't debug code without a design any more than you can.

You **should** commit your design document *before* you commit the code specified by the design document. You're encouraged to do the design in pieces (*e.g.*, overall design and detailed design of HTTP handling functions but not of the full server), leaving the code for the parts that aren't well-specified for later, after designing them. We **expect** you to commit multiple versions of the design document; your commit should specify *why* you changed the design document if you do this (*e.g.*, "original approach had flaw X", "detailed design for module Y", etc.). We want you to get in the habit of designing components before you build them.

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.

> 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.

Program functionality

You may not use standard libraries for HTTP; you have to implement this yourself. You have already been provided with a sample starter code that contains the networking system calls you need to create a server-side socket, which you used to build Assignment 1. You may use standard file system calls, but not any `FILE *` calls except for printing to the screen (*e.g.*, error messages). Note that string functions like `sprintf()` and `sscanf()` aren't `FILE *` calls.

Your code must be C, all source files must have a `.c` suffix and be compiled by `gcc` with no errors or warnings using the following flags: `-Wall -Wextra -Wpedantic -Wshadow`. Details on the HTTP functionality your server must support are available in Assignment 1.

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`.

Once again your program will take a port number as a parameter, but this time it will also have two optional parameters defining the number N of threads and the name of the `log_file` to hold the contents of logging. Either of these parameters can be omitted, which would result, respectively, on the server running with four threads, or without logging. Those are some examples of how to execute the server, notice that optional parameters can appear in any order:

- `./httpserver 8080 -N 4 -l log_file`
- `./httpserver -N 6 1234`
- `./httpserver 8010 -l other_log`
- `./httpserver 3030`

Multi-threading

Your previous Web server could only handle a single request at a time, increasing latency under concurrent access. 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, but the argument `-N nthreads` tells the server to use $N = nthreads$. For example, `-N 6` would tell the server to use 6 worker threads.

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 thread 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 in total; 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, here are two examples. We will also cover some basic `pthreads` techniques in section.

- `http://www.cs.kent.edu/~ruttan/sysprog/lectures/multi-thread/multi-thread.html`
- `https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html`

Logging requests

If the `-l log_file` option is provided to your program, it must log each request to the file `log_file` (obviously, `log_file` is whatever name you provide on the command line). For the purposes of this assignment, if the `log_file` already exists it should be truncated, that is, all content should be erased before the server starts adding records. If the `-l` option isn't provided, no logging is done.

A log record entry for a successful PUT request that creates a file with content "`Hello; this is a small test file`" looks like this:

```
PUT /abcdefghij0123456789abcdefg length 32\n
00000000 48 65 6c 6c 6f 3b 20 74 68 69 73 20 69 73 20 61 20 73 6d 61\n
00000020 6c 6c 20 74 65 73 74 20 66 69 6c 65\n
========\n
```

The first line contains the operation (PUT, HEAD 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. If the operation is **not** HEAD the first line is followed by lines representing the content of the data transferred, that is, the data that was received from the PUT request, or the data sent in response to a GET request. The representation used will be similar, but not equal, to the output of `hexdump -C <file>` without the ASCII section. The contents of the data should be displayed at 20 bytes per line, according to this format: each line starts with a zero-padded byte decimal number representing how many bytes have been shown *before* this line, followed by up to 20 bytes of the content data 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 the second line contains $8 + 20 \times 3 + 1 = 69$ characters. The log entry is terminated by `========\n`.

A log entry to GET the file created by that previous put would look like this:

```
GET /abcdefghij0123456789abcdefg length 32\n
00000000 48 65 6c 6c 6f 3b 20 74 68 69 73 20 69 73 20 61 20 73 6d 61\n
00000020 6c 6c 20 74 65 73 74 20 66 69 6c 65\n
========\n
```

A log entry for the same resource name but with the operation HEAD would look like this:


```
HEAD /abcdefghij0123456789abcdefg length 32\n
========\n
```

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

```
FAIL: GET /abcd HTTP/1.1 --- response 404\n
========\n
```

The record starts with `FAIL:`, followed by the first line of the response (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. The error log entry is also terminated by `========\n`.

Each request must be logged *contiguously*, that is, a request entry must not be interrupted by another request entry. 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. Alternatively, you can also create a separate thread to handle writing to log, which will receive requests from the threads handling requests.

Health check

If the client wants to monitor the performance of `httpserver`, they should be able to do so by performing a health check. For the purpose of this assignment, the performance will be measured through the number of errors registered in the `log_file`. The client should then be able to GET a special resource named `healthcheck`, and the body of the response should have the number of errors registered in the `log_file` as well as the total number of entries. For example, if the server is running on the same computer on port 8080 and has 54 errors in the log with 193 entries, then the client should be able to send a GET request with `curl localhost:8080/healthcheck`, and will receive a response that looks as follows:

```
HTTP/1.1 200 OK\r\nContent-Length: 6\r\n\r\n54\n193
```

In other words, this would give the same result as if the client were the request a file named `healthcheck` that contains only `54\n193`, however no such file should be created or read from, the server will build the response based on the contents of `log_file`. This, of course, is only possible if the server is using the option `log_file`. If that is not the case the server should respond with a 404 error response. Requests to HEAD or PUT the `healthcheck` resource should get a 403 error response, even if there is no `log_file` in use.

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`**; plain text 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 Assignment 2, please answer the following question:

- Using either your HTTP client or `curl` and your *original* HTTP server from Assignment 1, do the following:
    - Place eight 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 400 MiB long.
    - Start `httpserver`.
    - Start eight *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. The best way to do this is to write a simple shell script (command file) that starts eight 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 the observed speedup?
- 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?
- For this assignment you are logging the entire contents of files. In real life, we would not do that. Why?

Submitting your assignment

All of your files for Assignment 2 must be in the `asgn2` directory in your `git` repository. When you push your repository to GITLAB@UCSC, the server will run a program to check the following:

- 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`.

If the repository meets these minimum requirements for Assignment 2, there will be a green check next to your commit ID in the GITLAB@UCSC Web GUI. If it doesn't, there will be a red X. **It's OK to commit and push a repository that doesn't meet minimum requirements for grading.** However, we will only *grade* a commit that meets these minimum requirements.

Note that the *minimum* requirements say nothing about correct functionality—the green check only means that the system successfully ran `make` and that all of the required documents were present, with the correct names. **You must submit the commit ID you want us to grade via Google Form (`https://forms.gle/sa3buopV6Z2AeX7W6`). This must be done before the assignment 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. Transfers may be aborted on errors. However, the server doesn't *exit* on an error; it deals with the error appropriately (sending the corresponding error code for the client if possible) and ends the connection in that thread, leaving the thread free to handle another connection.
- Use `getopt(3)` to parse options from the command line. 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.

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 (20%); functionality (70%); writeup (10%).

If you submit a commit ID without a green checkmark next to it or modify `.gitlab-ci.yml` in any way, your maximum grade is 5%. Make sure you submit a commit ID with a green checkmark.

Frequently Asked Questions

- Will there be multiple GET/HEAD requests to the same file?

  These are concurrent reads, which means that you should be able to handle this without locking the file at all (regardless, you should not be locking the file ever!)

- Will there be interleaved writes between GET/HEAD to the same file?

  No, we will not test this.

- Will there be concurrent PUT requests to different files?

  Yes, we will test for this.

- Will there be concurrent PUT requests to the same file?

  No, we will not test for this.

