Assignment 2
CMPE 105: Principles of Computer System Design, Winter 2019

Due: February 15 at 5:00PM

Goals

The goals for Assignment 2 are to modify the HTTP server that you already implemented to have two additional features: multi-threading and a simple cache. Multi-threading means that your server must be able to handle multiple requests simultaneously, each in its own thread. The cache must be able to hold individual *blocks* (each 4 KiB) in memory so that it can serve them up to clients more quickly. You'll need to use synchronization techniques to service multiple requests at once, and to ensure that the cache is accessed "safely".

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

Programming assignment: multi-threaded HTTP server with in-memory caching

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 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 **must** commit your design document *before* you commit the code specified by the design document. You're welcome to do the design in pieces (*e.g.*, multi-threading first, then the design for the cache), as long as you don't write code for the parts that aren't well-specified. 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.). **If you commit code before it's designed, or you commit your design a few minutes before the working code that the design describes, you will lose points.** 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 aren't requiring you to include parts of the system that are *unchanged* from assignment 1. For example, socket setup code should be the same, and so you need not include a detailed description of how this works. 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.

Program functionality

Your code may be either C or C++, but all source files must have a `.cpp` suffix and be compiled by `clang++`. 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.

Details on the HTTP functionality your server must support are available in Assignment 1; Assignment 2 adds no more HTTP functionality, but rather improves server performance. 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.

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, 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 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 such as the cache (see below), they must use synchronization mechanisms for correctness. (If you implement a lock-free data structure that actually works correctly, you won't need to synchronize access to that data structure.)

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 (has multiple pages—see the next page button at the bottom), and this one. We will also cover some basic `pthreads` techniques in section.

In-memory block cache

To make your server faster, it will *cache* recently-used blocks in memory. Each block is 4 KiB, and contains all or part of a file that was recently received or sent by the server. The cache is implemented as a hash table whose key is the 40-character HTTP name and block number from the file, and whose content is the relevant block. If a block is only partially full—for example, the last block of a 5 KiB file—that's OK; the block only contains the valid data, but is still 4 KiB long. The cache is an LRU cache, so the blocks are maintained in a linked list (in addition to the hash table). When a block is accessed, it's put at the head of the list. If a new block is needed and the cache contains the maximum number of blocks, the block at the tail of the list is removed from the LRU list and the hash table and reinserted into both at the head under its new "name".

When a file is received, it is written to disk (as before), but a copy is kept in the cache. When a file is sent, a copy is kept in the cache. Before sending each file, however, the server checks the cache and sends a block from the cache if it finds it there rather than reading the block from the disk.

The default cache size is 40 blocks. However, the cache size may be specified using the `-c N_BLOCKS` argument to `httpserver`. You should pre-allocate the cache and insert the blocks into the cache, using a key that will never be used for a real block. A good technique would be to use `ssize_t` as the type for the block number in the file, and to use -1 to indicate that a block is free, since -1 is an invalid block number for a file.

Note that the cache is a shared data structure between the worker threads, and so it must be thread-safe. A simplistic approach is to build it like a monitor, where a single lock protects the entire read cache, only allowing one thread to access one block at a time. While this is thread-safe, it's not as concurrent as it could be, and so will earn fewer points than a *correct, thread-safe* read cache that allows multiple threads to access different blocks at once.

README and Writeup

Your repository must also include a README file (`README.md`) 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 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 4 MiB 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, cache? 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. 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, linked to the assignment page on Canvas. 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 Prof. Miller or Daniel 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 on February 5–6 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 caching separately before trying them together. You may use your own `httpclient` or an existing Web client (`curl`, `wget`) to test your server. Your writeup should say which one you used.
- Consider how to build the cache. You might want the cache to handle reading and writing the actual blocks so that the worker threads need only ask the cache to do reading and writing. This way, you leverage abstraction: worker threads just "read" and "write" data blocks, while the cache knows if it actually has to read or write, or can simply return a value from memory.
- How much concurrency can you get in the cache? Carefully think about how you need to do locking for access to the cache, and whether there are ways to have multiple threads accessing (and modifying) *different* blocks in the cache at the same time.
- Design for correctness first. Perhaps it makes sense to build your read cache with a single lock protecting the whole thing, get that working, and then try to improve concurrency.
- Aggressively check for and report errors. Transfers may be aborted on errors. However, neither the client nor the server *exits* on an error; it just goes on to the next transfer.
- 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.
- A sample command line for the server might look like this:
  ```
  ./httpserver -N 8 -c 50 localhost:8888
  ```
  This would tell the server to start 8 threads, and to have 50 blocks in the cache. 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 (20%); functionality (35%); 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.