Assignment 4: A *Multi-Threaded* HTTP Server
CSE 130: Principles of Computer Systems Design

`asgn4` Due: March 14, 2023 at 23:59

Goals This assignment provides you with experience managing concurrency through synchronization. Nearly all modern systems are faced with managing concurrency to increase their hardware utilization. So, this project is a great way to have some *real-world* experience! We encourage you to consider the *design* of your server before you start building the code—design is extra important when dealing with multiple threads.

Overview This project will combine Assignments 2 and 3 to build a *multi-threaded* HTTP server. Namely, your server will add a **thread-safe** queue (Asgn 3) to an **HTTP server** (Asgn 2) so that the server can serve multiple clients simultaneously. *Think: what performance metric does this form of multi-threading improve, latency or throughput?.*

While your server should process multiple clients simultaneously, it must ensure that its responses conform to a coherent and atomic **linearization** of the client requests. In effect, this means that an outside observer could not differentiate the behavior of your server from a server that uses only a single thread. Your server must create an **audit log** that identifies the linearization of your server (See below for more details).

Submission As always, the code that you submit must be in your repository on `git.ucsc.edu`. In particular, your assignment must build your HTTP server, called `httpserver`, when we execute the command `make` from the `asgn4` directory. Submit a 40-character commit ID hash on Canvas to identify the commit that you want us to grade. We will grade the last hash that you submit and will use the timestamp of your last upload to determine grace days. For example, if you post a commit hash 36 hours after the deadline, we will subtract 2 grace days from your total. When you submit, you should only submit the commit ID you want us to grade.

Assignment Details

Your server must implement the functionality explained below subject to the limitations below.

Functionality

Your new `httpserver` should take two command line arguments, `port`—the port to listen on (this is the same as Assignment 2)—`threads`—the number of worker threads to use. The `port` argument is required and the `threads` argument is optional (defaulting to 4). We recommend that you use `getopt` to parse the command-line arguments.

```
./httpserver [-t threads] <port>
```

Like in Assignment 2, your server will create, listen, and accept connections on a *socket* that is listening on a *port*. However, it will also use `threads` *worker* threads to support up to `threads` multiple clients at the same time (as in Assignment 2, each client will send only one request in each connection) and will output an *audit log* to `stderr`.

Audit Log

Your server should produce an audit log that indicates the order in which it processed requests; your server should output the audit log to `stderr`. When your server processes each request, it should add an entry to the log with the following format:

```
<Oper>,<URI>,<Status-Code>,<RequestID header value>\n
```

The comma-separated single line of text starts with the type of operation that was performed, i.e., `PUT` or `GET`. Then, the line includes the `URI` (With the same limitations as in Assignment 2, i.e., the format `/%63[A-Za-z.-]s`). The line should then include the status code that your server intended to produce in the response to the request (for example, if your server intended to send a `200 Status-Code`, but the client closed before reading the entire message, the audit log entry should still contain a `200 Status-Code`). Finally, the line should include the value of an optional HTTP header, called `RequestID`, that a client will specify (or the keyword `0` if the `RequestID` header was not found in the request associated with the line).

Log entries should be **Atomic** (i.e., there should not be any partial, overwritten, or otherwise corrupted lines in your log). The log should be durable: your server should ensure that the log entries for each request processed by your server are resident in the log. If you are doing something extra ambitious (e.g., adding your own buffering for your log), then here are the durability semantics: your server must ensure that log entries are resident in the log after we send your server a `SIGTERM` signal (you can add a signal handler to add code that runs when a signal arrives to your server). Otherwise, *we encourage you to use `fprintf` to produce audit-log entries; it provides the durability and atomicity requirements that you need.*

Ordering Requirements

Your server should produce responses that are coherent and atomic with respect to the ordering specified in your audit log. That is, if an entry for a request R_1, is earlier than an entry for a request, R_2, in the log, then the server's response to R_2 must be as though the processing for R_1 occurred in its entirety before any of the processing for R_2. We call this ordering a **linearization** because it creates a single linear ordering of all client requests. We say that this ordering is a **total order** because it provides an ordering for all elements (i.e., for any two unique requests, R_1 or R_2, your audit log will identify that either R_1 happens-before R_2 or that R_2 happens-before R_1).

Your server's linearization must conform to the order of requests that it received in the following way: if two requests, R_1 and R_2, arrive such that the start time of R_2 is after the end time of R_1, then your audit log should indicate the line for R_2 after the log line for R_1. However, if R_1 and R_2 overlap (i.e., the start time of R_1 is before the start time of R_2 and the start time of R_2 is before the end time of R_1), then your server's audit log could produce audit log entries in either order (i.e., R_1 could be before R_2 or R_2 could be before R_1).

Multi-threading

Your server should use a *thread-pool* design, a popular concurrent programming construct for implementing servers. A thread pool has two types of threads, *worker threads*, which process requests, and a *dispatcher thread*, which listens for connections and dispatches them to the workers. It would probably make sense to dispatch requests by having the dispatcher push them into a threadsafe queue (Assignment 3, see Resources below) and having worker threads pop elements from that queue.

Worker Threads Your server must create exactly `threads` worker threads (Note: this means that the server must have exactly `threads` + 1 threads). A worker thread should be idle (i.e., sleeping by waiting on a lock, conditional variable, or semaphore) if there are no requests to be processed. Each worker thread should perform HTTP processing for a request (Assignment 1, see Resources below). You will need to carefully implement synchronization to ensure that your server properly maintains a state that is shared across worker threads and to ensure coherency and atomicity.

Dispatcher Thread Your server should use a single dispatcher thread. A typical design uses the main thread (i.e., the function call `main`), as the dispatcher. The dispatcher should wait for connections from a client. Once a connection is initiated, the dispatcher should alert one of the worker threads and listen for a new client. If there are no idle worker threads, then the dispatcher should wait until a worker thread finishes its current task. Your server will have to implement correct synchronization to ensure that the dispatcher thread and worker threads correctly "hand-off" client requests without dropping any client requests, corrupting any data, or crashing the server.

Additional Functionality

In addition to supporting the methods listed above, your project must do the following:

- `httpserver` should not have any memory leaks. If you are feeling ambitious: the best way to ensure this is to write a signal handler that frees all memory. Otherwise, don't worry, you can get all of your points as long as your server never uses more than 10 MBs of memory.
- Your `httpserver` should only block request processing if either (1) there are already `thread` active clients or (2) blocking request processing is necessary to ensure that coherency/atomicity of your server's linearization of requests. This means that your server must concurrently process requests when it is safe to do so. It cannot simply spawn `threads` on demand, but only perform work in a single thread.

Limitations

You must write `httpserver` using the 'C' programming language. Your program cannot use functions, like `system` or `execve`, that allow you to execute external programs. **If your submission does not meet these minimum requirements, then the maximum score that you can get is 5%.**

Examples

We include a number of examples to illustrate the intended functionality of `httpserver`.

Example 1

This example focuses on the audit log format. Suppose that the server receives the following requests, one after the other, and that `a.txt` exists initially, but `b.txt` does not:

```
GET /a.txt HTTP/1.1\r\nRequest-Id:  1\r\n\r\n
GET /b.txt HTTP/1.1\r\nRequest-Id:  2\r\n\r\n
PUT /b.txt HTTP/1.1\r\nRequest-Id:  3\r\nContent-Length:  3 \r\n\r\nbye
GET /b.txt HTTP/1.1\r\n\r\n
```

the server should produce the audit log:

```
GET,/a.txt,200,1
GET,/b.txt,404,2
PUT,/b.txt,201,3
GET,/b.txt,200,0
```

Example 2

This example focuses on times when your server should operate totally concurrently—i.e., when it should allow process multiple requests at the same time.

Suppose that we start your server with two threads in a directory containing the files `a.txt`, `b.txt`, and `c.txt`. Your server should create two worker threads and one dispatcher thread [1]. The worker threads should wait for a message from the dispatcher, while the dispatcher thread should wait for a connection on

[1] one of these should be the *main thread*, the thread that called `main()`

its listen socket. Then, suppose that the following three requests arrive at your server *concurrently* (i.e., at the same time):

```
GET /a.txt HTTP/1.1\r\nRequest-Id: 1\r\n\r\n
GET /b.txt HTTP/1.1\r\nRequest-Id: 2\r\n\r\n
GET /c.txt HTTP/1.1\r\nRequest-Id: 3\r\n\r\n
```

The dispatcher thread should wake up one of the worker threads to handle one of the requests and wake up another worker thread to handle a second of the requests. The dispatcher should track that it has seen, but not processed, the third request [2]. Request processing should synchronize *as little as possible*—in this case, since the requests *do not* access the same URI, the processing should not need to synchronize at all! As soon as either worker finishes processing their request, that worker should begin processing the third request. The other thread should wait to be alerted by the dispatcher that there are more requests. Because the requests are operated concurrently, your server can produce any of the following six audit logs:

```
GET,/a.txt,200,1    GET,/a.txt,200,1    GET,/b.txt,200,2
GET,/b.txt,200,2    GET,/c.txt,200,3    GET,/a.txt,200,1
GET,/c.txt,200,3    GET,/b.txt,200,2    GET,/c.txt,200,3

GET,/b.txt,200,2    GET,/c.txt,200,3    GET,/c.txt,200,3
GET,/c.txt,200,3    GET,/a.txt,200,1    GET,/b.txt,200,2
GET,/a.txt,200,1    GET,/b.txt,200,2    GET,/a.txt,200,1
```

After all of this processing, the server should be in the steady state of having (1) the workers waiting for alerts from the dispatcher and (2) the dispatcher thread waiting for a connection (i.e., calling `listener_accept()` on the listen socket).

Example 3

This example identifies scenarios when your server's request processing should synchronize.

Suppose that we start your server with two threads in a directory that does not contain the file, `a.txt`. Your server should create two worker threads and one dispatcher thread. The worker threads should wait for a message from the dispatcher, while the dispatcher thread should wait for a connection on its listen socket. Suppose that the following two requests arrive at your server *concurrently*:

```
GET /a.txt HTTP/1.1\r\nRequest-Id: 1\r\n\r\n
PUT /a.txt HTTP/1.1\r\nRequest-Id: 2\r\nContent-Length: 3 \r\n\r\nbye
```

Your server will need to produce an audit log and responses to requests that are consistent with each other. Namely, there are *only two* possible combinations of audit log and response:

Option 1.

Audit Log

```
GET,/goodbye.txt,404,1
PUT,/goodbye.txt,201,2
```

Request	Response
1	`HTTP/1.1 404 Not Found\r\nContent-Length: 10 \r\n\r\nNot Found\n`
2	`HTTP/1.1 201 Created\r\nContent-Length: 8 \r\n\r\nCreated\n`

Option 2.

Audit Log

```
PUT,/goodbye.txt,201,2
GET,/goodbye.txt,200,1
```

Request	Response
2	`HTTP/1.1 201 Created\r\nContent-Length: 8 \r\n\r\nCreated\n`
1	`HTTP/1.1 200 OK\r\nContent-Length: 3 \r\n\r\nOK\n`

After all of this processing, the server should be back in the steady state of having (1) the workers waiting for requests to arrive and (2) the dispatcher thread waiting for a connection (i.e., calling `accept()` on the listen socket).

[2]We suggest (but do not require) that you track unprocessed requests in the queue from assignment 3

Rubric

We will use the following rubric for this assignment:

Category	Point Value
Makefile	10
Clang-Format	5
Files	5
Functionality	80
Total	100

Makefile Your repository includes a Makefile with the rules `all` and `httpserver`, which produce the `httpserver` binary, and the rule `clean`, which removes all `.o` and binary files. Additionally, your Makefile should use clang (i.e., it should set `CC=clang`), and should use the `-Wall`, `-Wextra`, `-Werror`, and `-pedantic` flags (i.e., it should set `CFLAGS=-Wall -Wextra -Werror -pedantic`).

Clang-Format All `.c` and `.h` files in your repository are formatted in accordance with the `.clang-format` file included in your repository.

Files The following files are included in your repository: `httpserver.c`, `Makefile`, and `README.md`. Your repository should not include binary files nor any object files (i.e., `.o` files), except for the file named `asgn4_helper_funcs.a` (see Resources below). To make it easier for you to maintain tests, you can also include binary files in any directory whose name starts with the phrase `test`.

Functionality Your `httpserver` program performs the functionality described in Assignment Details.

Resources

Here are some resources to help you:

Synchronization

Your server must use the POSIX threads library (`pthreads`) to implement multithreading. The `pthread` library is MASSIVE, but you'll only need a few things for this assignment. In particular, you might find the following groups of functions to be useful (you probably won't use them all, though):

- `pthread_create`: create a thread
- `pthread_mutex_init`, `pthread_mutex_lock`, and `pthread_mutex_unlock`: the `pthread` mutex implementation
- `sem_init`, `sem_wait`, and `sem_post`: the `pthread` semaphore implementation
- `pthread_cond_init`, `pthread_cond_signal`, and `pthread_cond_wait`: the `pthread` condition variable implementation

You will also probably find the function, `flock`, to be useful for helping atomically update files.

Testing

We provided you with three resources to test your own code:

1. An autograder, which is run each time you push code to GitLab, will show you the points that you will receive for your Makefile, Clang-Format, and Files.
2. A set of test scripts in the resources repository to check your functionality. You can use the tests to see if your functionality is correct by running them on your Ubuntu 22.04 virtual machine. We provided you with a subset of the tests that we will run, but, I bet you can figure the other ones out by adapting what we have given you :-)

3. Testing a concurrent server is difficult. So, we provided you with a number of powerful test scripts to help you test your code. We named this collection of tools "`olivertwist` and his Merrie Men"; the resources directory will contain a few examples of using them:
   (a) `olivertwist` takes a sequence of `request commands` from a `TOML` file, passes these requests to a server, records the order in which it sent requests, and records the output of the requests. The TOML file allows you to specify very specific (and interesting) interleavings of requests from clients.
   (b) `sherlock` takes an audit log and request order from `oliver` and determines if the audit log is possible w.r.t the request order that was sent (in particular, it determines if the audit log is a total ordering of what was sent).
   (c) `watson` identifies if the audit log and responses are consistent with each other.

Starter Library

We provide a starter library for you to use when writing your server—see the practicum entitled "Practicum: The Assignment 4 Starter Library" for more details. We retain the header file from asgn2 (i.e., `asgn2_helper_funcs.h`), but we repackaged all of the functionality into a single library file named `asgn4_helper_funcs.a`. If you are confused about how to use the library, please consult the course recordings—we went through the practicum in detail during class. As a final note—you don't *have* to use our code, but you might find it helpful.

Useful 'C' Libraries

Here are some tips of things that 'C' gives you that you might find useful:

1. `fprintf` ensures atomicity. Consider using it to output your audit log.
2. `flock` allows you to lock a file descriptor—it even has *reader/writer* facilities so that you can have multiple threads holding a "`shared`" lock, but only a single thread holding an "`exclusive`" lock. You might find this feature useful when you try to ensure coherency/atomicity for accesses to each URI.
3. The `pthread` library has many primitives that you might find useful, including `sem_t`, `pthread_mutex_t`, and `pthread_cond_t`.

Hints

This is a large project and there are definitely multiple ways to get started. We describe an approach below based upon our experience working with students in the past, but you do not *have* to follow this section (that is why it's under Hints!)

Design Approach

You should design your server before you start writing any code. Your design should articulate exact function prototypes and struct definitions that you plan to use. You may find it helpful to visualize how your server will function by drawing a picture similar to the "box-and-arrows" pictures that we've been drawing in class. By thoroughly designing your system, you will find that you can avoid many potential bugs, thereby saving you hours!

Keep in mind that your design will probably change as you build your server; sometimes, the act of trying to implement something illuminates how bad of an idea it was. If this happens, we encourage you to "go back to the drawing board" and re-design the server. If you find yourself implementing something that you haven't "designed", then you're doing it wrong.

Step 1: Design your Scaffolding In any project, it's good to start by thinking about how to get a basic prototype working–something that doesn't actually solve the problems in the assignment but nevertheless has running code. The scaffolding in this project is the dispatcher and worker threads—How will you create them? How will they communicate? Don't think about the processing that each worker thread needs to do, yet, just focus on how these two types of threads will interact. You should probably design some structs (or use some structs from the starter library) to make this all work.

Step 2: Design Unsynchronized Connection Processing Connection processing is where things get interesting. The first thing to do is to make the processing work without synchronization. In other words, How can you use the starter libraries that you've been given to implement a system that works in the scenario where requests access independent URIs? You should figure out how to handle audit logging at this time as well. Take a look at the aforementioned practicum—it contains a lot of tips for this step.

Step 3: Adding Synchronization to your Connection Processing The requirement that you have coherent and atomic processing requires that your worker threads synchronize. How are you going to do that? One big challenge is ensuring that requests are atomic—How can you make sure that two concurrent put requests don't clobber each other when writing to the file? Another major challenge is ensuring that the audit log and responses are consistent.

You should be able to make a clear and well-thought-out argument for why this design is thread-safe. In other words–why **must** it be the case that your design ensures coherency and atomicity?

Implementation Approach

Now that you've designed everything, we suggest that you follow those design steps, one by one, when you implement your system. So, start by implementing the aforementioned scaffolding. Then, add unsynchronized connection processing. Finally, add synchronization to your connection processing.