Assignments 2–4: A *Multi-Threaded* HTTP Server
CSE 130: Principles of Computer Systems Design

Due: May 11, 2022 at 12:01AM, May 20, 2022 at 12:01AM, and Jun 1, 2022 at 12:01 AM

Goals This assignment will provide you with experience in dealing with concurrency by using correct synchronization. Nearly all modern systems are faced with managing concurrency in order to better utilize their hardware. We'll be working on this project in the context of an HTTP server, something that you are already well acquainted with after Assignment 1. We are going to split the assignment up into three checkpoints to help you best manage your time. The checkpoints will ramp up in difficulty, so keep up with the work! You will also probably find it helpful to think about the *design* of your server before you start building the code. A well-considered design is extra important when dealing with multiple threads.

Overview You will be adding *multi-threading* and an *audit log* to your HTTP server from Assignment 1. Multi-threading will enable your server to serve multiple clients simultaneously, thus improving the *throughput* of your server. Audit logging will provide a record of the actions that your server performs. Audit logging is very common in practice; you also might find it a very useful tool to help you debug and test your software, especially since your multi-threaded server will likely be *non-deterministic*.

To help you budget your time throughout this project, you will turn in your project at three checkpoints. Each checkpoint will require partial functionality: first, you will implement the audit logging, then the work-queue, and finally the atomicity of file system operations. See the checkpoint section below for more details.

Checkpoint	Folder	Task	Points	Due
1	asgn2	Audit Logging	50	May 11 @ 12:01 AM
2	asgn3	Work Queue	50	May 24 @ 12:01 AM
3	asgn4	Atomic Requests	100	Jun 01 @ 12:01 AM

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 directory specified in the checkpoint folder above (i.e., `asgn2` for the first checkpoint, `asgn3` for the second checkpoint, and `asgn4` for the third checkpoint). In each checkpoint, you should include a `README.md` file that tracks the interesting design decisions and outlines the high-level functions, data-structures, and modules that you used in your code. For the second and third checkpoints, the `README.md` should also include a description of why your system is thread-safe (i.e., why does it work even when multiple threads are operating concurrently?). This description should talk about which variables are shared across threads, how they are modified, what critical sections you have, etc.

You must submit a 40 character commit ID hash on Canvas in order for us to identify which commit you want us to grade. We will grade the last hash that you submit to each of the checkpoints on Canvas 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. **We will use the final checkpoint to assign bonus points—i.e., you will earn bonus points by submitting to the third checkpoint early.**

Your server must implement the functionality explained below subject to the limitations below. Your code will necessarily build on Assignment 1, so there is a benefit to starting with an Assignment 1 that uses abstraction and modularity well. However, we've minimized the dependency between the assignments by limiting the Assignment 1 functionality that we will test.

Functionality

In this assignment, you'll build on your program from Assignment 1. We expect you to use your code from Assignment 1, and also to reuse code across each checkpoint. But be sure to copy the code that you want tested to the the folder for each checkpoint (`asgn2`, `asgn3`, and `asgn4`).

Your new `httpserver` should take three command line arguments, `port`—the port to listen on (this is the same as assignment 1)—`threads`—the number of worker threads to use, and `logfile`—the file in which to store your audit log. The `port` argument is required, the `threads` argument is optional (defaulting to 4), and the `logfile` is optional (defaulting to writing to `stderr`). We have provided starter code that uses `getopt` to parse the command line alongside this assignment document. The usage of the server is the following:

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

Like in Assignment 1, 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 in order to support multiple clients at the same time, and will output an *audit log* to the location specified by `logfile`.

Below, we provide details about the audit log, worker queue, and the atomicity and coherency guarantees that your server must provide. Each checkpoint in this assignment will test for these features one-by-one. We then provide some high-level guidance on what synchronization libraries you should use, and describe the assignment 1 functionality that your server must implement (this will help you ensure that you have a decent place from which to start building your server).

Audit Log

Your server must log the requests that were made to it in the order in which the server processes them. When your server processes each request, it should add an entry to the log. Each entry must have 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`, `GET`, or `APPEND`. Then, the line includes the `URI` (With the same limitations as in Assignment 1, i.e., the format `/%19[A-Z,a-z_.]s`). The line should then include the status code produced in the response to the request. Finally, the line should include the value of an HTTP header, called `RequestID`, or the keyword `0` if the `RequestID` header was not found in the request associated with the line.

There should not be any partial, overwritten, or otherwise corrupted lines in your log. The log must be consistent with the responses that your server produces: If request, R_1, is before a request, R_2, in the log, then your server must have processed R_1 and R_2 such that R_1 happens before R_2. This is especially important for operations that conflict (e.g., requests that modify the same URI).

The audit log entry should reflect the status code that your server intended to send to the client in its response. This is relevant if your server has an issue sending the right data to a client (e.g., if a server is unable to send an entire message body because the client closes a message).

Your log must be durable; the server must ensure that the log entries for each request processed by your server are resident in the log. We will send a `SIGTERM` signal to kill your server and wait for the server to terminate on shutdown, so the requirement is that the log entries are resident in the log after your serve handles the `SIGTERM`. The starter code includes a signal handler that you can modify as needed to help you with this task. Please note, if your server does not gracefully terminate after the "polite" `SIGTERM` within 5 seconds, then we will kill your process with a `SIGKILL`.

If signal handlers freak you out, you can also ensure that each log entry is flushed immediately after being written.

Example. If we send the following requests one-after-the-other, assuming that `hello.txt` exists initially, but `goodbye.txt` does not:

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

then your server should produce the log:

```
GET,/hello.txt,200,1
GET,/goodbye.txt,404,2
PUT,/goodbye.txt,201,3
GET,/goodbye.txt,200,0
```

Multi-threading

Your server must use a *thread-pool* design, a popular concurrent programming construct for implementing servers. A thread pool has two types of threads, *worker thread*, which actually process requests, and a *dispatcher thread*, which listens for connections and dispatches them to the workers.

Worker Threads

Your server must create exactly `threads` worker threads (**Note: this means that the server must have a total of `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 the HTTP processing from Assignment 1 when a request arrives.

You will have to implement correct synchronization to ensure that your server properly maintains state that is shared across worker threads. You must ensure that the output of your server is indistinguishable from the output that a single-threaded version would produce. Potential pitfalls include ensuring that the each request gets its intended response, and ensuring that all requests eventually receive a response, among others.

On a related note—your server must concurrently process requests when it is safe to do so. You cannot simply fork `threads`, but only perform work in a single thread.

Dispatcher Thread

Your server should create a single dispatcher thread. **Note: Your server must call `pthread_create` exactly `threads` times, so the dispatcher thread should probably be the main thread**. 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.

Example. Suppose that we start your server with two threads. Your server should create two worker threads and one dispatcher thread (Note: one of these should be the *main thread*, the thread that called `main()`). The worker threads should wait for requests to arrive, while the dispatcher thread should wait for a connection on its listen socket (i.e., calling `accept()` on the listen socket, as is done in the starter code for Assignment 1).

Then, suppose that we send the following three requests *concurrently* (i.e., at the same time):

```
GET /hello.txt HTTP/1.1\r\nRequest-Id: 1\r\n\r\n
GET /goodbye.txt HTTP/1.1\r\nRequest-Id: 2\r\n\r\n
GET /hellogoodbye.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 account for the third, unprocessed, request. You have a number of potential designs for this, such as (1) waiting until a worker thread is idle or (2) storing the unprocessed request somewhere and returning to listening for a connection. As soon as either worker finishes processing its request, that worker should begin processing the third and final request. The other thread should go back to a waiting state after it finishes processing its request, as should the thread that processed two requests.

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).

Atomicity and Coherency

While your server is multithreaded, it must process requests atomically and coherently. The server's request processing must follow a total ordering—i.e., for all pairs of requests, R_1 and R_2, the response to R_1 and R_2 must be equivalent to output that would be produced if R_1 *happens-before* R_2 or R_2 *happens-before* R_1, where we say that R_1 happens-before R_2 if and only if the response of R_1 is produces before the server begins processing R_2.

For example, suppose that R_1 and R_2 are `APPEND` requests that append the content, `hello 1\n`, and `hello 2\n`, respectively, to a URI that points to an empty file. The end content of the file pointed to by the URI must be either:

```
hello 1
hello 2
```

or

```
hello 2
hello 1
```

This essentially stipulates that the operations must be *atomic*.

Additionally, suppose that R_1 and R_2 are `PUT` requests that update the content of a URI that points to an empty file to be `hello 1\n`, and `hello 2\n`, respectively. Further, suppose that R_3 is a request that occurs after R_1 and R_2. Then, if R_1 happens before R_2, then R_3 must return a content body of `hello 2\n`; if R_2 happens before R_1, then R_3 must return a content body of `hello 1\n` (note: there is no other case since each request *must* have a total ordering with respect to all other requests). This requirement enforces *coherency*.

The audit log produced by your server must encode the total order of your servers processing. That is, if your server processes R_1 before R_2, then your log must indicate that R_1 happened-before R_2. **Think carefully about how the atomicity and coherency constraints interact with the audit log and thread-pool design! Getting this correct is a tough challenge :)**.

Example. Suppose that we start your server with two threads. Your server should create two worker threads and one dispatcher thread (n.b., one of these should be the *main thread*, the thread that called main). The worker threads should wait for requests to arrive, while the dispatcher thread should wait for a connection on its listen socket (i.e., calling `accept()` on the listen socket, as is done in the starter code for assignment 1).

Then, suppose that we send the following three requests *concurrently* (i.e., at the same time), assuming that `goodbye.txt` does not exist initially:

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

Your server can produce either of the following combinations or responses and audit log. It's very important, however, that the server produces a combination listed below (i.e., it cannot produce an audit log from one option but the responses from another):

Option 1.

Audit Log

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

Request-Id	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-Id	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`

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 account for the third, unprocessed, request. You have a number of potential designs for this, such as (1) waiting until a worker thread is idle or (2) storing the unprocessed request somewhere and returning to listening for a connection. As soon as either worker finishes processing its request, that worker should begin processing the third and final request. The other thread should go back to a waiting state after it finishes processing its request, as should the thread that processed two requests.

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).

Additional Functionality

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

- `httpserver` should not have any memory leaks. **Your server must cleanup any memory that it uses in its `SIG_TERM` handler; the starter code has a signal handler to help you get started on this task!**
- `httpserver` must be reasonably efficient.
- Your code should be formatted according to the clang-format provided in your repository and it should compile using `clang` with the `-Wall -Werror -Wextra -pedantic -lpthread` compiler flags. The `-lpthread` flag is what allows your program to use the `pthread` library (see Hints).

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%.**

Carry-over from Assignment 1

There is a necessary dependency between this assignment and your Assignment 1. However, we want to minimize the number of "double-jeopardy" issues that arise. Consequently, we have drastically limited the scope of requests that you will need to handle. In particular, you may assume that each request will be correctly formatted. I.e., there are no invalid requests—if your server reads some bytes, those bytes belong to some valid HTTP request. You will only need to produce the following status codes:

`Status-Code`	`Status-Phrase`	`Message-Body`	`Usage`
200	OK	`OK\n`	When a method is Successful
201	Created	`Created\n`	When a URI's file is created
404	Not Found	`Not Found\n`	When the URI's file does not
500	Internal Server Error	`Internal Server Error\n`	When an unexpected issue prevents processing

Checkpoints

There are three checkpoints for this assignment. In the first, you need to implement the audit log, but will not need to worry about multiple threads. In the second, you will need to implement the Work Queue, but do not need to worry about HTTP method atomicity. Finally, in the third, you will implement the final atomic requests and ensure that the audit log, work queue, and atomic requests all operate correctly in tandem.

Checkpoint 1. You will implement the audit log. Our tests will start your server with the default thread options, but you don't need to actually create the worker threads. This should be a fairly straightforward application of things that we've discussed in this course and is a good opportunity for you to ensure that you have all of the necessary Assignment 1 behavior ironed out. This checkpoint will be worth 50 points and should be turned in through the `asgn2` folder in your repository.

Checkpoint 2. You will implement the Work Queue. Our tests will start your server with a variety of different numbers of threads, you will need to ensure that you create the correct number. Your server will need to return valid responses when serving a large number of concurrent clients. You will need to ensure that your server does not busy wait, but also processes requests as soon as possible. However, we will ensure that the client requests are all *non-conflicting* (i.e., you will not need to worry about atomicity and coherency concerns for these tests). This will allow you to focus entirely on the Work Queue part of this assignment independently from the rest. Note–your audit log does not need to be managed atomically for Checkpoint 2 (i.e., you can have log entries that overwrite and intermix with each other). This checkpoint will be worth 50 points and should be turned in through the `asgn3` folder in your repository.

Checkpoint 3. You will implement the atomicity and coherence in the third checkpoint. This checkpoint is a superset of the past two checkpoints, i.e., you will need to perform all of the functionality from Checkpoints 1 and 2. Your server will need to start the correct number of threads, manage the work queue using synchronizaiton to ensure that all client requests are processed correctly. Your server should perform all operations atomically and coherently, ensuring that it processes each HTTP method in a total order that matches the total order in the audit log. Your audit log will need to be managed atomically for Checkpoint 3. Finally, your server should not busy loop (i.e., Worker threads should sleep when there is not work to be performed), and should also process requests *as fast as possible* (i.e., it should aim for the highest possible throughput). This checkpoint will be worth 100 points and should be turned in through the `asgn4` folder in your repository.

Testing your Code

We will be implementing a new testing approach. Our aim is to encourage you to perform your own testing in order to debug your code, rather than using the pipeline to debug your code. As such, you will be able to, on a daily basis, see how your program performs on every test, and, on a per-push basis, see how your program performs on a subset of tests:

- Each day at 9AM, I will execute the full suite of tests on the default branch of your git@ucsc repository. You will receive an email that indicates the aggregate score (e.g., xx/85 functionality points) that your current commit achieves.
- Each time you push your code to git@ucsc, the pipeline will check formatting, `make`, and execute a few of the functionality tests. We will show a small English description of these tests on the pipeline.

Hints

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.

Other Tips

- There are many functions that are not "re-entrant", which means that you cannot safely use them with multiple threads at the same time (e.g., `strtok`). These functions generally have a "re-entrant" version (e.g., `strtok_r`).
- You will likely need to lookup how some system calls (e.g., `read`) and library functions (e.g., `warn`) work. You can always Google them, but you might also find the man pages useful (e.g., try typing `man 2 read` on a terminal).
- There are a few ways to test `httpserver`. Below, we assume that you started `httpserver` on port 1234 by executing the command `./httpserver 1234`. We also assume that you are using your client on the same machine upon which the server is currently executing:
  - You can use an HTTP Client, such as a web browser. We recommend testing with `curl`, a command-line HTTP client. `curl` can produce both `GET` and `PUT` commands. For example, to execute a `GET` of the file `foo.txt` on `httpserver` and place the output into the file `download.txt`, you execute:

    ```
    curl http://localhost:1234/foo.txt -o download.txt
    ```

    `curl` can execute `PUT` and `GET` commands; use `help curl` or `man curl` on a terminal to learn more.
  - You can also use `nc` ("netcat"). To connect to your server, execute `nc localhost 1234`. Then, you can type in the text that you wish to send to your server. You can also automate this approach by piping data to `nc`. For example, to send the `PUT` command listed above to your server, execute:

    ```
    printf "PUT /foo.txt HTTP/1.1\r\nContent-Length: 12\r\n\r\nHello World!"
    |nc localhost 1234
    ```

- If you try to start your server immediately after killing a previous instance of it, you will likely see the following error:
  ```
  httpserver: bind error: Address already in use
  ```
  In this case, just restart the server with a different port number. The issue is that the operating system must ensure unique ports are used across the entire system; it often waits to gracefully close ports even after the process that was using them terminates.

Grading

Each Checkpoint will be graded with the following breakdown:

- Functionality tests: 70%
- README design doc: 15%
- Coding style: 15%

Starter Code

Your starter code does four basic things for you:

- Your starter code parses and validates command-line arguments passed to your HTTP server.
- Your starter code creates a socket, binds that socket to the local interface, and then listens for requests on the socket. The starter code calls `handle_connection()` for each request, which simply echoes bytes passed to the established connection file descriptor `connfd`.
- Your starter code includes a signal handler that ignores the `SIGPIPE` signal. This handler makes it so that socket failures will result in setting `errno` to `EPIPE` rather than throwing a signal.
- Your starter code also includes a signal handler that calls `sigterm_handler()` whenever `SIGTERM` is signaled.