COM3610 Programming Assignment 2
Threading and Synchronization

You are to do this project with one partner. It is too big to tackle alone.
This project must be implemented in C

There are four objectives to this assignment:

- To modify an existing code base.
- To learn how to create and synchronize cooperating threads in Unix.
- To gain exposure to how a basic web server is structured.
- To think more about how to test the functionality of a code base.

Notes

- When compiling and linking, you should use the argument **-pthread** to the compiler. This takes care of adding in the right libraries, etc., for using pthreads. This is built into the Makefile I provide with your project.
- It is **NOT OK** to share code outside your team. If your team is having trouble, **come talk to me.**
- In this document, code and things you are expected to type are presented in `Courier fixed width font`.

Background

In this assignment, you will be developing a real, working **web server.** To greatly simplify this project, code for a very basic web server is provided. It will be your job to make the web server multi-threaded so that it is more efficient.

HTTP Background

The code for a working basic web server is provided as a starting point in order to shield you from all of the details of network connections and the HTTP protocol. The code provided already handles everything described in this section. If you are really interested in the full details of the HTTP protocol, you can read the specification, but this is not required for this project.

Web browsers and web servers interact using a text-based protocol called HTTP (Hypertext Transfer Protocol). A web browser opens an Internet connection to a web server and requests some content with HTTP. The web server responds with the requested content and closes the connection. The browser reads the content and displays it on the screen.

Each piece of content on the server is associated with a file. If a client requests a specific disk file, then this is referred to as static content. If a client requests that a executable file be run and its output returned, then this is dynamic content. Each file has a unique name known as a URL (Universal Resource Locator). For example, the URL http://www.yu.edu:80/index.html identifies an HTML file called "/index.html" on Internet host "www.yu.edu" that is managed by a web server listening on port 80. The port number in a URL is optional and defaults to the well-known HTTP port of 80. (Modern secure web servers use the protocol HTTPS instead of HTTP and listen by default on port 443)

An HTTP request (from the web browser or other client to the server) consists of a request line, followed by zero or more request headers, and finally an empty text line. A request line has the form:

```
[method] [uri] [version]
```

`[method]` is usually `GET` (but may be other things, such as `POST`, `OPTIONS`, or `PUT`). The `URI` is the file name and any optional arguments (for dynamic content). Finally, the `version` indicates the version of the HTTP protocol that the web client is using (e.g., `HTTP/1.0` or `HTTP/1.1`).

Following the request line are one or more *request header* lines, each consisting of name-value pair with the name separated from the value by a colon. These lines tell the server more details about what methods of responding to the request are acceptable to the browser. The only required one is the `host`, which tells web site host the client is trying to talk to, as each server may serve several web sites.

An HTTP response (from the server to the browser/client) is similar; it consists of a response line, zero or more response headers, an empty text line, and finally the interesting part, the response body. A response line has the form

```
[version] [status] [message]
```

The `status` is a three-digit positive integer that indicates the state of the request; some common states are 200 for "OK", 403 for "Forbidden", and 404 for "Not found". Two important response header lines `Content-Type`, which tells the client the MIME type of the content in the response body (e.g., html or gif) and `Content-Length`, which indicates its size in bytes.

If you would like to see the HTTP protocol in action, you can connect to any web server using `telnet.` For example, run `telnet www.google.com 80` and then type (note that there is an empty line at the end):

```
GET / HTTP/1.1
host: www.google.com
```

You will then see the HTML text for that web page! (Note: telnet will not work for modern *secure* web pages whose URLs start with https:// instead of http://)

Important: When running a web server and browser on the same virtual machine, the host name for your connection will be `localhost`. A typical URL might be `http://localhost/index.html`

Again, you don't need to know this information about HTTP unless you want to understand the details of the code we have given you. **You will not need to modify any of the procedures in the web server that deal with the HTTP protocol or network connections.**

Basic Web Server

The code for the web server is available from `Canvas.` You should copy over all of the files there into your own working directory. You should compile the files by typing `make`. Your shell needs to be in the same directory as your files when you type this. Compile, run, and test this basic web server before making any changes to it! `make clean` removes .o files and lets you do a clean build.

To run the server, use the following command line:

```
./server -port 8099 --root /home/yourname/COM3610/HW2/www
```

In the example above, the server will run on port 8099 and look in the folder `/home/yourname/COM3610/HW2/www` for the files to serve to web clients.

You should specify port numbers that are greater than about 2000 to avoid active ports. **If working on the CompOrg server, be sure to pick a random port so your server doesn't try to use the same port as another team's server.**

When you then connect your web browser to this server, make sure that you specify this same port.

Assuming the server command line above, to view this file from a web browser (running on the same or a different machine), use either of these URLs:
http://127.0.0.1:8099/index.html
http://localhost:8099/index.html

To view this file using the client code we give you, use the command
```
./client localhost 8099 /index.html
```

The web server provided is a real, in-use server. Most of the ~2700 lines of code can be left as-is.

In the server code, it should be the case that all error codes returned by library functions are being checked, with orderly termination of the server if a problem is found. One should **always check error codes!** However, many programmers don't like to do it because they believe that it makes their code less readable. You may either code your error checking inline or user a wrapper library (like csapp.c from COM2113). Note the common convention of naming a wrapper function the same as the underlying system call, except capitalizing the first letter, and keeping the arguments exactly the same. **In no case may you make a library call without some kind of check of the return code. All library calls you add must check error codes.**

Overview: New Functionality

In this project, you will be adding functionality to both the web server code and the web client code.

You will be adding three key pieces of functionality to the basic web server. First, the web server, as written, forks a separate process for each new connection. You must make the web server multi-threaded using a pool of ready and waiting worker threads, with the appropriate synchronization. Second, you will implement different scheduling policies so that requests are serviced in different orders. Third, you will add statistics to measure how the web server is performing. You will also be modifying how the web server is invoked so that it can handle new input parameters (e.g., the number of threads to create).

You will also be adding functionality to the web client for testing. You should think about how this new functionality will help you test that the web server is implemented correctly. You will modify the web client so that it is also multi-threaded and can initiate requests to the server in different, well-controlled groups.

Part 1: Multi-threaded Server

The basic web server that we start with is a modified version of Althttpd, the webserver that runs the sqllite.org website. The COM3610 version has had TLS support removed, which reduced the size of the source file by ~500 lines. Documentation for the design is found at https://sqlite.org/althttpd/doc/trunk/althttpd.md.

When the server launches, the function http_server() (Line 2419) runs in a loop (2488) which forks a separate copy of itself (line 2507) on demand for every incoming connection, sort of a "pop-up process." Your main task is to replace this design with one using a pool of threads to service incoming requests – no new processes will be created via fork() in your design. (Note that the name of our server code has been changed from `althttpd.c` to `server.c`. When you read the documentation info, make that change in your minds)

The simplest approach to building a multi-threaded server is to spawn a new thread for every new http request, instead of a new process. The OS will then schedule these threads according to its own policy. When one thread is blocked (i.e., waiting for disk I/O to finish) the other threads can continue to handle other requests. However, the drawback of the one-thread-per-request approach is that the web server pays the overhead of creating a new thread on every request.

Therefore, the generally preferred approach for a multi-threaded server is to create a **fixed-size pool of worker threads** when the web server is first started. With the pool-of-threads approach, each thread is blocked until there is an http request for it to handle. Therefore, if there are more worker threads than active requests, then some of the threads will be blocked, waiting for new http requests to arrive; if there are more requests than worker threads, then those requests will need to be buffered until there is a ready thread.

In your implementation, you must have a master thread that begins by creating a pool of worker threads, the number of which is specified on the command line. Your master thread is then responsible for accepting new http connections over the network and placing a descriptor for this connection into a fixed-size buffer; in your basic implementation, the master thread should not read from the connection. The number of elements in the buffer is also specified on the command line. Note that the existing web server spawns a process as soon as a connection arrives. Your design needs to have the main thread save the connection info in a buffer before handing it off to a worker thread. (It is up to you to determine what info should be saved, and in what format.) It then returns to its task of accepting new connections. You should investigate how to create and manage posix threads with `pthread_create` and `pthread_detach.`

Each worker thread is only able to handle requests for static web pages (files). A worker thread wakes when there is an http request in the queue; when there are multiple http requests available, which request is handled depends upon the scheduling policy, described below. Once the worker thread wakes, it performs the read on the network descriptor, obtains the specified content (by reading the specified static file), and then returns the content to the client by writing to the descriptor. The worker thread then waits for another http request.

Note that the master thread and the worker threads are in a producer-consumer relationship and require that their accesses to the shared buffer be synchronized. Specifically, the master thread must block and wait if the buffer is full; a worker thread must wait if the buffer is empty. In this project, you are required to use `condition variables.` **If your implementation performs any busy-waiting (or spin-waiting) instead, you will be heavily penalized.**

Part 2: Scheduling Policies

In this project, you will implement a number of different scheduling policies. Note that when your web server has multiple worker threads running (the number of which is specified on the command line), you will not have any control over which thread is actually scheduled at any given time by the OS. Your role in scheduling is to determine which http request should be handled by each of the waiting worker threads in your web server.

The scheduling policy is determined by a command line argument when the web server is started and are as follows:

- `First-in-First-out (FIFO)`: When a worker thread wakes, it handles the first request (i.e., the oldest request) in the buffer. Note that the http requests will not necessarily finish in FIFO order since multiple threads are running concurrently; the order in which the requests complete will depend upon how the OS schedules the active threads.
- `Highest Priority to Image Content (HPIC)`: When a worker thread wakes, it handles the first request that is for an image file; these are files whose name ends in `.jpg`, `.png`, *etc*. If there are no requests for image content, it handles the first request for HTML content. Note that this algorithm can lead to the starvation of requests for HTML content.
  - This is essentially FIFO for image content, then FIFO for HTML content
- `Highest Priority to HTML Content (HPHC)`: This is the opposite of HPRC. Note that this algorithm can lead to the starvation of requests for non-HTML content.
  - This is essentially FIFO for HTML content, then FIFO for image content

You will note that the HPIC and HPHC policies require that something be known about each request before the requests can be scheduled. Thus, to support this scheduling policy, you will need to do some initial processing of the request outside of the worker threads; you will want the master thread to perform this work, which requires that it read from the network descriptor.

Part 3: Usage Statistics

You will need to modify your web server to collect a variety of statistics. Some of the statistics will be gathered on a per-request basis and some on a per-thread basis. All statistics will be returned to the web client as HTTP response headers. Specifically, you will be embedding these statistics in the entity headers; we have already made a place-holder in the basic web server code for where these headers are written (line 760). You should add the additional statistics. Note that most web browsers will ignore these headers that it doesn't know about; to access these statistics, you will want to run the modified client.

For each request, you will record the following counts or times; all times should be recorded at the granularity of milliseconds, and expressed in milliseconds. You may find `gettimeofday()` useful for gathering these statistics.

- `X-stat-req-arrival-count:` The number of requests that arrived before this request arrived. Note that this is a shared value across all of the threads.
- `X-stat-req-arrival-time:` The arrival time of this request, as first seen by the master thread. **This time should be relative to the start time of the web server.**
- `X-stat-req-dispatch-count:` The number of requests that were dispatched before this request was dispatched (i.e., when the request was picked by a worker thread). Note that this is a shared value across all of the threads.
- `X-stat-req-dispatch-time:` The time this request was dispatched (i.e., when the request was picked by a worker thread). **This time should be relative to the start time of the web server.**
- `X-stat-req-complete-count:` The number of requests that completed before this request completed; we define "completed" as the point just before the worker thread starts writing the response on the socket. Note that this is a shared value across all of the threads.
- `X-stat-req-complete-time:` The time at which the worker thread begins writing the response on the socket. **This time should be relative to the start time of the web server.**
- `X-stat-req-age:` The number of requests that were given priority over this request (that is, the number of requests that arrived after this request arrived but were dispatched before this request was dispatched).

You should also keep the following statistics for each thread:

- `X-stat-thread-id:` The id of the responding thread (numbered 0 to number of threads-1)
- `X-stat-thread-count:` The total number of http requests this thread has handled
- `X-stat-thread-html:` The total number of HTML requests this thread has handled
- `X-stat-thread-image:` The total number of image requests this thread has handled

Thus, for a request handled by thread number i, your web server will return the statistics for that request and the statistics for thread number i as HTTP headers.

Part 4: Multi-threaded Client

You have been provided with a basic single-threaded client that sends a single HTTP request to the server and prints out the results. This basic client prints out all response headers with the statistics that you added, so that you can verify the server is ordering requests as expected. While this basic client can help you with some testing, it doesn't stress the server enough with multiple simultaneous requests to ensure that the server is correctly scheduling or synchronizing threads. Therefore, you need to modify the web client to send more requests with multiple threads. Specifically, your new web client must implement two different request workloads (specified by a command line argument you will add). All versions take a new command line argument: N, for the number of created threads.

- `Concurrent Groups (CONCUR)`: The client creates N threads and uses those threads to concurrently (i.e., simultaneously) perform N requests for the same file; this behavior repeats forever (until the client is killed). You should ensure that the N threads overlap sending and waiting for their requests with each other. After all of the N threads receive their responses, the threads should repeat the requests. You may find the routine `pthread_barrier_wait` useful for implementing this; **in no case should busy-waiting be used**.
- `First-In-First-Out Groups (FIFO)`: The client creates N threads and uses those threads to perform N requests for the same file; however, the client ensures that the requests are initiated in a specific thread-wise serial order, but that the responses can occur in any order. The threads are assigned a specific order $T_1 \ldots T_n$, and after T_1 sends its request, T_1 should signal T_2 that it can now send its request, T_2 should signal T_3 that it can now send its request, and so on for the N threads; the N threads then concurrently wait for the responses. After all of the N threads receive their responses, the threads should repeat the requests (starting again with T_1) until the client is killed. You might find semaphores useful for implementing this behavior; **in no case should busy-waiting be used**.

Program Specifications

For this project, you will be implementing both the server and the client. The server code you start with is invoked as:

```
./server [portnum] [folder] &
./server -port [portnum] --root [folder] &
```

Your web server must be invoked exactly as follows:

```
./server
-port [portnum]
--root [folder]
-threads [threads]
-buffers [buffers]
-schedalg [schedalg] &
```

The command line arguments to your web server are to be interpreted as follows.

- `portnum`: the port number that the web server should listen on; the basic web server already handles this argument.
- `folder`: the absolute path of the folder in which your files are hosted; the basic web server already handles this argument.
- `threads`: the number of worker threads that should be created within the web server. Must be a positive integer.
- `buffers`: the number of request connections that can be accepted at one time. Must be a positive integer. Note that it is not an error for more or less threads to be created than buffers.
- `schedalg`: the scheduling algorithm to be performed. Must be one of FIFO, HPIC, or HPHC.

For example, if you run your program as

```
./server -port 5003 --root /home/yourname/COM3610/HW2/www
         -threads 8 -buffers 16 -schedalg FIFO &
```

then your web server will listen to port 5003, serve from `/home/yourname/COM3610/HW2/www`, create 8 worker threads for handling http requests, allocate 16 buffers for connections that are currently in progress (or waiting), and use FIFO scheduling for arriving requests.

To stop your server from running, since it is a daemon, find its PID in the process table (using `ps aux | grep server` is one way) and then `kill` it. **Remember to do this before you start another version.**

Your starting code for the web client is invoked as
```
./client [host] [portnum] [filename]
```

For example:
```
./client localhost 8082 /index.html
```

Your finished web client must be invoked exactly as follows:

```
client [host] [portnum] [threads] [schedalg] [filename1] [filename2]
```

The command line arguments to your web server are to be interpreted as follows.

- `host`: the name of the host that the web server is running on; the basic web client already handles this argument.
- `portnum`: the port number that the web server is listening on and that the client should send to; the basic web client already handles this argument.
- `threads`: the number of threads that should be created within the web client. Must be a positive integer.
- `schedalg`: the scheduling algorithm to be performed. Must be one of CONCUR or FIFO.
- `filename1`: the name of the file that the client is requesting from the server.
- `filename2`: the name of a second file that the client is requesting from the server. This argument is optional. If it does not exist, then the client should repeatedly ask for only the first file. If it does exist, then each thread of the client should alternate which file it is requesting.

Hints

A good first step is to understand how the provided code works. All of the code is available as a zip file on Canvas. The following files are provided:

`server.c`:	Contains main() and support routines for the web server.
`client.c`:	Contains main() and the support routines for the very simple web client.
`Makefile`:	Support file for rebuilding your system
`www`:	Directory containing a sample HTML file and image file

You can type "make" to recompile the client and server executable. You can type "make clean" to remove the object files and the executables. You can type "make server" to create just the server program, etc. If you create new files, you will need to alter the Makefile. Search online for information on how to do this.

Start by experimenting with the existing code. The best way to learn about the code is to compile it and run it. Exercise the server with your preferred web browser. Run this server with the client code we gave you. You can even have the client code we gave you contact any other server (e.g., www.yu.edu). Make small changes to the server code (e.g., have it print out more debugging information) to see if you understand how it works.

I anticipate that you will find the following routines useful for creating and synchronizing threads: `pthread_create`, `pthread_detach`, `pthread_mutex_init`, `pthread_mutex_lock`, `pthread_mutex_unlock`, `pthread_cond_init`, `pthread_cond_wait`, `pthread_cond_signal`. To find information on these library routines, being with the manual pages (using the Unix command man), and read the tutorials below.

I strongly recommend that you use a good version control system for your code to prevent disastrous edits. There are many good tutorials for systems such as git and mercurial.

You may find the following tutorials useful as well.

Linux Tutorial for Posix threads
POSIX threads programming

Grading

Hand in a zip file (or a gzipped tar) of your source code, make file, the HTML and image files, and a README file to Canvas. Do not include any .o files, html files, or graphics files. Make sure that all your group's members are listed in the README file.

In your `README` file you should have the following five sections:

- The name and login information for all project partners. A brief description of how you divided the work between you.
- Design overview: A few simple paragraphs describing the overall structure of your code and any important structures.
- Complete specification: Describe how you handled any ambiguities in the specification. For example, how do you implement the ANY policy?
- Known bugs or problems: A list of any features that you did not implement or that you know are not working correctly
- **Testing:** This requirement an aspect that I am very interested in. **I will assign points for answering these questions.** Describe how you tested the functionality of your web server. Describe how can you use the various versions of your extended client to see if the server is handing requests concurrently and implementing the FIFO, HPSC, or HPDC policies.

  Specifically, what are the exact parameters one should pass to the client and the server to demonstrate that the server is handling requests concurrently? To demonstrate that the server is correctly running the FIFO policy? the HPSC policy? the HPDC policy? In each case, if your client and server are behaving correctly, what output and statistics should you see?

NOTE: Your team's submission will be tested and graded using the CompOrg server, so be sure your team's code compiles and runs correctly on the CompOrg server before handing in your submission.