COM3610
Programming Assignment 1
Threading and Synchronization

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 and client 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 with others. If you are 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 client.** To greatly simplify this project, code for a very basic web server and client is provided.

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 I have given you.

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. `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.

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 I give you, use the command

```
./client localhost 8099 /index.html
```

The web server provided is a real, in-use server.

In your 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

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. You need to modify the web client to send more requests with multiple threads. Specifically, your new web client must the request workload below. Your client should take a new command line argument: N, for the number of created threads.

- `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

The server code you start with is invoked as:

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

The provided 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] [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.
- `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 code (e.g., have it print out more debugging information) to see if you understand how it works.

In this and future assignments, 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, see the lecture slides, take a look at the manual pages (using the Unix command man or the websites https://linux.die.net/ or https://www.kernel.org/doc/man-pages/), 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

The server has a -logfile flag which takes as an argument the full path to a log file. You may find it useful to use and/or modify this for logging.

Most browsers have a developer mode that allows you to see the HTTP request and response sent and received by the browser. In Chrome, this is accessed with **tools->developer tools**

You should not be using WSL for development, but if you do, the server's --root flag should not point to a location in /mnt.

Here are a couple useful Clion tips:

1. If you are getting a CMake error, where CMAKE isnt recognizing the pthread_create command, add this to CMAKE right above add_executable:

```
find_package(Threads)

SET(CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} -pthread")
```

2. in your configuration, your program arguments start after ./server or ./client commands as listed in the HW2.pdf on canvas

3. follow this guide to view all threads running in parallel https://www.jetbrains.com/help/clion/parallel-stacks-view.html (the icon is on the far right of the debugger terminal).

Valgrind

Valgrind can be very useful. Here's how to install and use it:

Step 1:

Check to see if you have valgrind installed:

```
which valgrind
```

If you get a result, go to step 2. Otherwise, type:

```
sudo apt update
```

and

```
sudo apt install valgrind
```

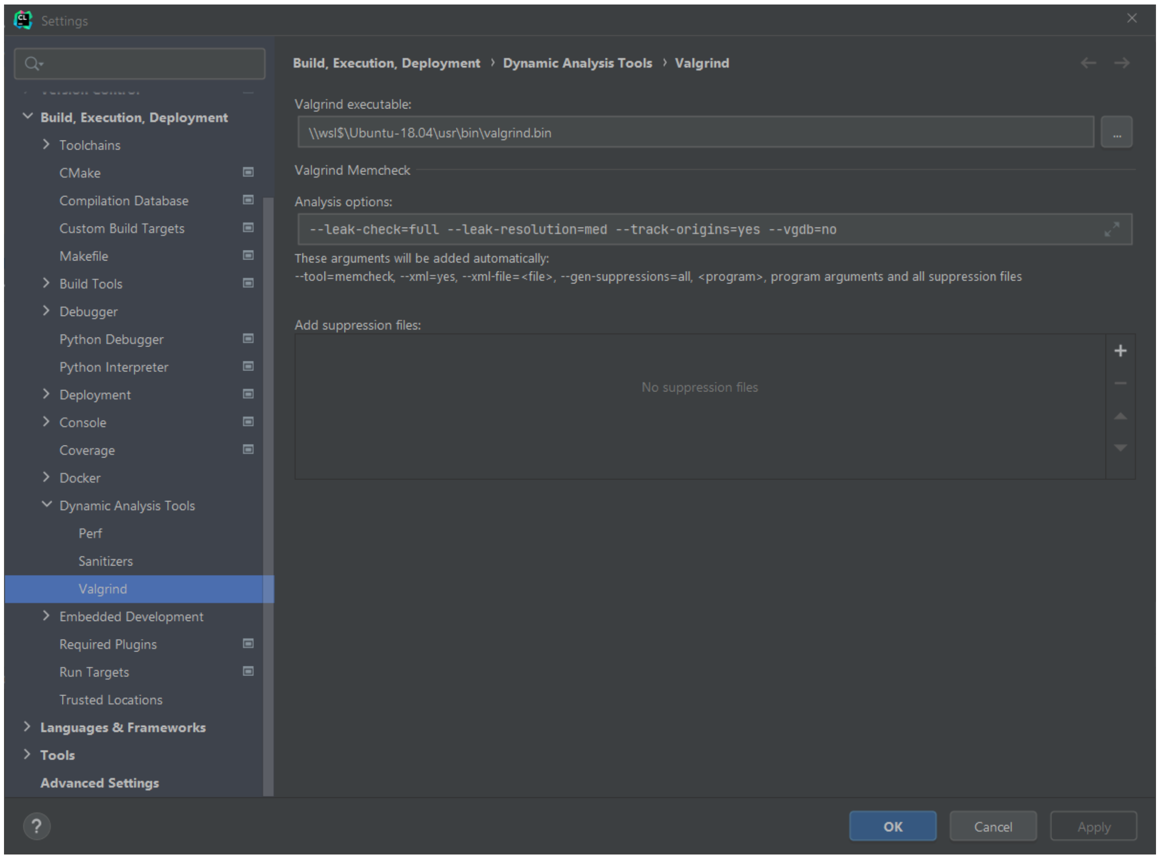
Get the location where valgrind is installed (should return /usr/bin/valgrind):

```
which valgrind
```

Step 2:

In Clion, go to the upper-right gear (or yellow circle) icon, select settings->Build, Execution, Deployment->Dynamic Analysis Tools->Valgrind

In the **Valgrind Executable** field, click the browse (...) button and browse to and click the **valgrind.bin** file located at the location from **Step 1**. In my case, Step 1 returned /usr/bin/valgrind, so browsed to /usr/bin and clicked valgrind.bin:



Build, Execution, Deployment › Dynamic Analysis Tools › Valgrind

Valgrind executable: \\wsl$\Ubuntu-18.04\usr\bin\valgrind.bin

Valgrind Memcheck

Analysis options: `--leak-check=full --leak-resolution=med --track-origins=yes --vgdb=no`

These arguments will be added automatically:
--tool=memcheck, --xml=yes, --xml-file=<file>, --gen-suppressions=all, <program>, program arguments and all suppression files

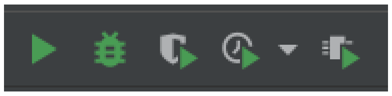
Add suppression files:

No suppression files

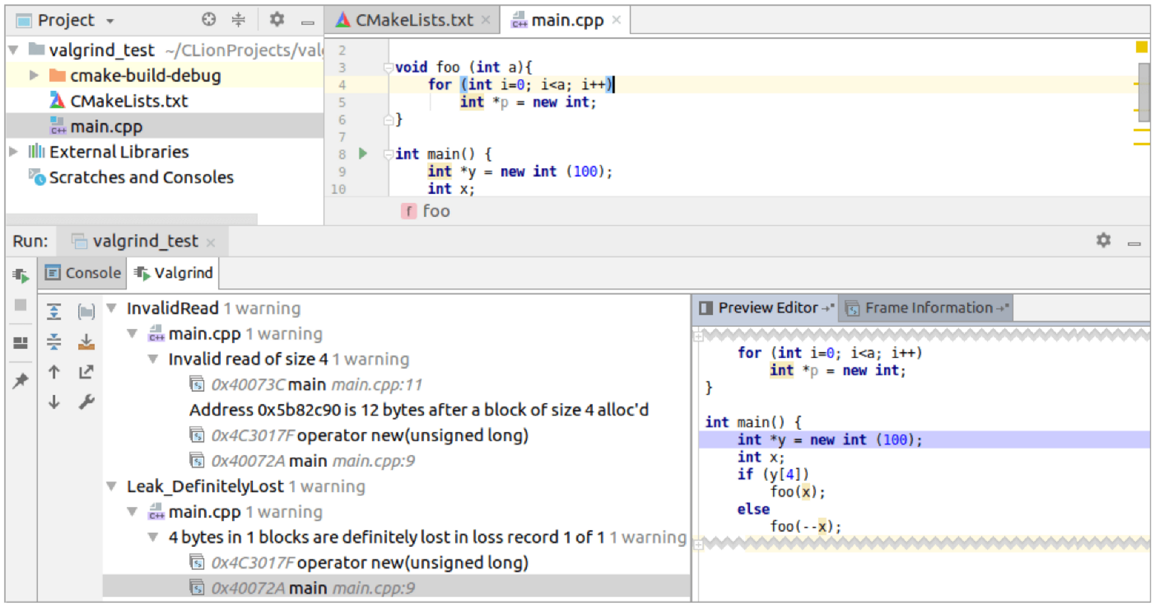
Click **OK**

Step 3:

To launch your server using valgrind, click the chip icon (right-most icon shown here):



If valgrind finds something, it will show up when you stop debugging:



```
void foo (int a){
    for (int i=0; i<a; i++)
        int *p = new int;
}

int main() {
    int *y = new int (100);
    int x;
```

```
InvalidRead 1 warning
  main.cpp 1 warning
    Invalid read of size 4 1 warning
      0x40073C main main.cpp:11
      Address 0x5b82c90 is 12 bytes after a block of size 4 alloc'd
      0x4C3017F operator new(unsigned long)
      0x40072A main main.cpp:9
Leak_DefinitelyLost 1 warning
  main.cpp 1 warning
    4 bytes in 1 blocks are definitely lost in loss record 1 of 1 1 warning
      0x4C3017F operator new(unsigned long)
      0x40072A main main.cpp:9
```

```
    for (int i=0; i<a; i++)
        int *p = new int;
}

int main() {
    int *y = new int (100);
    int x;
    if (y[4])
        foo(x);
    else
        foo(--x);
```

If you would rather use Valgrind at the command line on your client, you can do that. Here's an example of how to Valgrind the server. You can do the same thing with the client:

```
valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes --
verbose --log-file=valgrind-out.txt ./server -port 8888 --root
/home/username/COM3610/HW2/Handout/www --family ipv4 -debug
```

If Valgrind finds something, it will show up when you close your server.

Grading

Hand in a zip file of your client source code, make file, and a README file to Canvas. Do not include any .o files, html files, or graphics files. Make sure that your name is listed in the README file.

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

- 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.
- 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 client.

NOTE: Your submission will be tested and graded using the COM3610 Virtual Machine, so be sure your code compiles and runs correctly there before handing in your submission.