## Overview

In this assignment, you will be developing a concurrent web server. To simplify this project, we are providing you with the code for a non-concurrent (but working) web server. This basic web server operates with only a single thread; it will be your job to make the web server multi-threaded so that it can handle multiple requests at the same time.

The goals of this project are: - To learn the basic architecture of a simple web server - To learn how to add concurrency to a non-concurrent system - To learn how to read and modify an existing code base effectively

Useful reading from OSTEP includes:

- Intro to threads (click here)
- Using locks (click here)
- Producer-consumer relationships (click here)
- Server concurrency architecture (click here)

## HTTP Background

Before describing what you will be implementing in this project, we will provide a very brief overview of how a classic web server works, and the HTTP protocol (version 1.0) used to communicate with it; although web browsers and servers have evolved a lot over the years

the old versions still work and give you a good start in understanding how things work. Our goal in providing you with a basic web server is that you can be shielded from learning all of the details of network connections and the HTTP protocol needed to do the project; however, the network code has been greatly simplified and is fairly understandable should you choose to to study it.

link to history lesson (click here)

Classic web browsers and web servers interact using a text-based protocol called **HTTP** (**Hypertext Transfer Protocol**). A web browser opens a 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.

HTTP is built on top of the **TCP/IP** protocol suite provided by the operating system. Together, TPC and IP ensure that messages are routed to their correct destination, get from source to destination reliably in the face of failure, and do not overly congest the network by sending too many messages at once, among other features. To learn more about networks, take a networking class (or many!), or read this free book (click here).

Each piece of content on the web server is associated with a file in the server's file system. The simplest is *static* content, in which a client sends a request just

to read a specific file from the server. Slightly more complex is *dynamic* content, in which a client requests that an executable file be run on the web server and its output returned to the client. Each file has a unique name known as a **URL** (**Universal Resource Locator**).

As a simple example, let's say the client browser wants to fetch static content (i.e., just some file) from a web server running on some machine. The client might then type in the following URL to the browser: `http://www.justanexample.org/index.html`. This URL identifies that the HTTP protocol is to be used, and that an HTML file in the root directory (`/`) of the web server called `index.html` on the host machine `www.justanexample.org` should be fetched.

The web server is not just uniquely identified by which machine it is running on but also the **port** it is listening for connections upon. Ports are a communication abstraction that allow multiple (possibly independent) network communications to happen concurrently upon a machine; for example, the web server might be receiving an HTTP request upon port 80 while a mail server is sending email out using port 25. By default, web servers are expected to run on port 80 (the well-known HTTP port number), but sometimes (as in this project), a different port number will be used. To fetch a file from a web server running at a different port number (say 8000), specify the port number directly in the URL, e.g., `http://www.justanexample.org:8000/index.html`.

URLs for executable files (i.e., dynamic content) can include program arguments after the file name. For example, to just run a program (`test.cgi`) without any arguments, the client might use the URL `http://www.justanexample.org/test.cgi`. To specify more arguments, the `?` and `&` characters are used, with the `?` character to separate the file name from the arguments and the `&` character to separate each argument from the others. For example, `http://www.justanexample.org/test.cgi?x=10&y=20` can be used to send multiple arguments `x` and `y` and their respective values to the program `test.cgi`. The program being run is called a **CGI program** (short for Common Gateway Interface (click here); yes, this is a terrible name); the arguments are passed into the program as part of the `QUERY_STRING` (click here) environment variable, which the program can then parse to access these arguments.

## The HTTP Request

When a client (e.g., a browser) wants to fetch a file from a machine, the process starts by sending a machine a message. But what exactly is in the body of that message? These *request contents*, and the subsequent *reply contents*, are specified precisely by the HTTP protocol.

Let's start with the request contents, sent from the web browser to the server. This HTTP request 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`. The `method` is usually `GET`, which tells the web server that the client simply wants to read the specified file; however, other methods exist (e.g., `POST`). The `uri` is the file name, and perhaps optional arguments (in the case of dynamic content). Finally, the `version` indicates the version of the HTTP protocol that the web client is using (e.g., HTTP/1.0).

The HTTP response (from the server to the browser) 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` (i.e., the client can't access that file), and `404` for `File Not Found` (the famous error). Two important lines in the header are `Content-Type`, which tells the client the type of the content in the response body (e.g., HTML or gif or otherwise) and `Content-Length`, which indicates the file size in bytes.

For this project, you don't really 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. However, it's always good to learn more, isn't it?

## A Basic Web Server

The code for the web server is available in this repository. You can compile the files herein by simply typing `make`. Compile and run this basic web server before making any changes to it! `make clean` removes .o files and executables and lets you do a clean build.

When you run this basic web server, you need to specify the port number that it will listen on; ports below number 1024 are *reserved* (see the list here (click here)) so you should specify port numbers that are greater than 1023 to avoid this reserved range; the max is 65535. Be wary: if running on a shared machine, you could conflict with others and thus have your server fail to bind to the desired port. If this happens, try a different number!

When you then connect your web browser to this server, make sure that you specify this same port. For example, assume that you are running on `bumble21.yourmachine.org` and use port number 8003; copy your favorite HTML file to the directory that you start the web server from. Then, to view this file from a web browser (running on the same or a different machine), use the url `bumble21.yourmachine.org:8003/favorite.html`. If you run the client and web server on the same machine, you can just use the hostname `localhost` as a convenience, e.g., `localhost:8003/favorite.html`.

To make the project a bit easier, we are providing you with a minimal web

server, consisting of only a few hundred lines of C code. As a result, the server is limited in its functionality; it does not handle any HTTP requests other than `GET`, understands only a few content types, and supports only the `QUERY_STRING` environment variable for CGI programs. This web server is also not very robust; for example, if a web client closes its connection to the server, it may trip an assertion in the server causing it to exit. We do not expect you to fix these problems (though you can, if you like, you know, for fun).

Helper functions are provided to simplify error checking. A wrapper calls the desired function and immediately terminate if an error occurs. The wrappers are found in the file `io-helper.h`); more about this below. One should always check error codes, even if all you do in response is exit; dropping errors silently is **BAD C PROGRAMMING** and should be avoided at all costs.

# Finally: Some New Functionality!

In this project, you will be adding two key pieces of functionality to the basic web server. First, you make the web server multi-threaded. Second, you will implement different scheduling policies so that requests are serviced in different orders. 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).

## Part 1: Multi-threaded

The basic web server that we provided has a single thread of control. Single-threaded web servers suffer from a fundamental performance problem in that only a single HTTP request can be serviced at a time. Thus, every other client that is accessing this web server must wait until the current http request has finished; this is especially a problem if the current HTTP request is a long-running CGI program or is resident only on disk (i.e., is not in memory). Thus, the most important extension that you will be adding is to make the basic web server multi-threaded.

The simplest approach to building a multi-threaded server is to spawn a new thread for every new http request. The OS will then schedule these threads according to its own policy. The advantage of creating these threads is that now short requests will not need to wait for a long request to complete; further, 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 the descriptor for this connection into a fixed-size buffer; in your basic implementation, the master thread should not read from this connection. The number of elements in the buffer is also specified on the command line. Note that the existing web server has a single thread that accepts a connection and then immediately handles the connection; in your web server, this thread should place the connection descriptor into a fixed-size buffer and return to accepting more connections.

Each worker thread is able to handle both static and dynamic requests. 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 either reading the static file or executing the CGI process), 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. Note: if your implementation performs any busy-waiting (or spin-waiting) instead, you will be heavily penalized.

Side note: do not be confused by the fact that the basic web server we provide forks a new process for each CGI process that it runs. Although, in a very limited sense, the web server does use multiple processes, it never handles more than a single request at a time; the parent process in the web server explicitly waits for the child CGI process to complete before continuing and accepting more HTTP requests. When making your server multi-threaded, you should not modify this section of the code.

## 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; the order in which the requests complete will depend upon how the OS schedules the active threads.
- **Smallest File First (SFF)**: When a worker thread wakes, it handles the request for the smallest file. This policy approximates Shortest Job First to the extent that the size of the file is a good prediction of how long it takes to service that request. Requests for static and dynamic content may be intermixed, depending upon the sizes of those files. Note that this algorithm can lead to the starvation of requests for large files. You will also note that the SFF policy requires that something be known about each request (e.g., the size of the file) before the requests can be scheduled. Thus, to support this scheduling policy, you will need to do some initial processing of the request (hint: using `stat()` on the filename) outside of the worker threads; you will probably want the master thread to perform this work, which requires that it read from the network descriptor.

## Security

Running a networked server can be dangerous, especially if you are not careful. Thus, security is something you should consider carefully when creating a web server. One thing you should always make sure to do is not leave your server running beyond testing, thus leaving open a potential backdoor into files in your system.

Your system should also make sure to constrain file requests to stay within the sub-tree of the file system hierarchy, rooted at the base working directory that the server starts in. You must take steps to ensure that pathnames that are passed in do not refer to files outside of this sub-tree. One simple (perhaps overly conservative) way to do this is to reject any pathname with `..` in it, thus avoiding any traversals up the file system tree. More sophisticated solutions could use `chroot()` or Linux containers, but perhaps those are beyond the scope of the project.

## Command-line Parameters

Your C program must be invoked exactly as follows:

```
prompt> ./wserver [-d basedir] [-p port] [-t threads] [-b buffers] [-s schedalg]
```

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

- **basedir**: this is the root directory from which the web server should operate. The server should try to ensure that file accesses do not access

files above this directory in the file-system hierarchy. Default: current working directory (e.g., `.`).

- **port**: the port number that the web server should listen on; the basic web server already handles this argument. Default: 10000.
- **threads**: the number of worker threads that should be created within the web server. Must be a positive integer. Default: 1.
- **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. Default: 1.
- **schedalg**: the scheduling algorithm to be performed. Must be one of FIFO or SFF. Default: FIFO.

For example, you could run your program as:

```
prompt> server -d . -p 8003 -t 8 -b 16 -s SFF
```

In this case, your web server will listen to port 8003, create 8 worker threads for handling HTTP requests, allocate 16 buffers for connections that are currently in progress (or waiting), and use SFF scheduling for arriving requests.

## Source Code Overview

We recommend understanding how the code that we gave you works. We provide the following files:

- **wserver.c**: Contains `main()` for the web server and the basic serving loop.
- **request.c**: Performs most of the work for handling requests in the basic web server. Start at `request_handle()` and work through the logic from there.
- **io_helper.h** and **io_helper.c**: Contains wrapper functions for the system calls invoked by the basic web server and client. The convention is to add `_or_die` to an existing call to provide a version that either succeeds or exits. For example, the `open()` system call is used to open a file, but can fail for a number of reasons. The wrapper, `open_or_die()`, either successfully opens a file or exists upon failure.
- **wclient.c**: Contains `main()` and the support routines for the very simple web client. To test your server, you may want to change this code so that it can send simultaneous requests to your server. By launching `wclient` multiple times, you can test how your server handles concurrent requests.
- **spin.c**: A simple CGI program. Basically, it spins for a fixed amount of time, which you may useful in testing various aspects of your server.

- **Makefile**: We also provide you with a sample Makefile that creates `wserver`, `wclient`, and `spin.cgi`. You can type make to create all of these programs. 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. As you create new files, you will need to add them to the Makefile.

The best way to learn about the code is to compile it and run it. Run the server we gave you 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 that speaks HTTP. Make small changes to the server code (e.g., have it print out more debugging information) to see if you understand how it works.

## Additional Useful Reading

We anticipate that you will find the following routines useful for creating and synchronizing threads: `pthread_create()`, `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, read the man pages (RTFM).