Project 4a: Scalable Web Server

Note

Security: It might be worthwhile to be a little careful with security during this project. Probably not a big deal, but a good way to help with this is to make sure to run the web server out of a special directory with only a few files in it (e.g., a subdirectory of your build, or something you create specially in /tmp), and further to disallow any path names that have a **..** in them (which would allow people to go up a level or more in the directory hierarchy and thus explore any files you have access to). Minimally, don't leave your web server running for a long time.

Background

In this assignment, you will be developing a real, working **web server.** To simplify this project, we are providing you with the code for a very basic 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 is more efficient.

HTTP Background

Before describing what you will be implementing in this project, we will provide a very brief overview of how a simple web server works and the HTTP protocol. Our goal in providing you with a basic web server is that you should be shielded from all of the details of network connections and the HTTP protocol. The code that we give you already handles everything that we describe in this section. If you are really interested in the full details of the HTTP protocol, you can read the specification, but we do not recommend this for this project.

Most 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 `www.cs.wisc.edu:80/index.html`identifies an HTML file called "index.html" on Internet host "www.cs.wisc.edu" that is managed by a web server listening on port 80. The port number is optional and defaults to the well-known HTTP port of 80. URLs for executable files can include program arguments after the file name. A "?" character separates the file name from the arguments and each argument is separated by a "&" character. This string of arguments will be passed to a CGI program as part of its "QUERY_STRING" environment variable.

An HTTP request (from the web browser 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` . The `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).

An 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** , and 404 for **Not found** . Two important lines in the header are**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.

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 `~cs537-2/public/p4.` You should copy over all of the files there into your own working directory. You should compile the files by simply typing **make** . Compile and run this basic web server before making any changes to it! **make clean** removes .o files 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; 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. For example, assume that you are running on mumble21.cs and use port number 2003; 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:`mumble21.cs.wisc.edu:2003/favorite.html`

The web server that we are providing you is only about 200 lines of C code, plus some helper functions. To keep the code short and understandable, we are providing you with the absolute minimum for a web server. For example, the web server 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 crash. We do not expect you to fix these problems!

The helper functions are simply wrappers for system calls that check the error codes of those system codes and immediately terminate if an error occurs. One should **always check error codes!** However, many programmer don't like to do it because they believe that it makes their code less readable; the solution, as you know, is to use these wrapper functions. We expect that you will write wrapper functions for the new system routines that you call.

Overview: New Functionality

In this project, you will be adding ~~three~~ 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. ~~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).

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.

blocked => wait

# of worker threads is the # from the command line

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.

doesn't include master

master

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.

producer

consumer

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:

Have to implement all of theses

Handles oldest request first (queue style)

- **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 Filename First (SFNF):** When a worker thread wakes, it handles the request for the file with the smallest name. This policy is kind of silly, but that is life.
- **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.

Program Specifications

Your C program must be invoked exactly as follows:

Invoke program with the following command line:

```
prompt> server [portnum] [threads] [buffers] [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.
- **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, SFNF, or SFF.

For example, if you run your program as:

```
server 5003 8 16 SFF
```

then your web server will listen to port 5003, 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.

Hints

We recommend understanding how the code that we gave you works. All of the code is available from `~cs537-2/public/p4` . We provide the following files:

- **server.c:** Contains main() for the basic web server.
- **request.c:** Performs most of the work for handling requests in the basic web server. All procedures in this file begin with the string "request".
- **cs537.c:** Contains wrapper functions for the system calls invoked by the basic web server and client. The convention is to capitalize the first letter of each routine. Feel free to add to this file as you use new libraries or system calls. You will also find a corresponding header (.h) file that should be included by all of your C files that use the routines defined here.
- **client.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. At a minimum, you will want to run multiple copies of this client.
- **output.c:** Code for a CGI program. Basically, it spins for a fixed amount of time, which you may useful in testing various aspects of your server.

We also provide you with a sample Makefile that creates server, client, and output.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.

Useful routines

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: **RTFM** , which stands for **Read The Manual** ; you should also feel free to read the OS book (**RTFB** ?), which contains a great amount of detail on how to build producer-consumer relationships between threads.

Look in the book for