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 `~cs537-1/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`. Note that your client (the browser) may need to be on the CS network to connect to your server.

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 one key piece of functionality to the basic web server: you will make it multi-threaded. 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).

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. 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.