Web Servers and Synchronization

i. Project Goals

There are two objectives to this assignment:

- To learn how to create and synchronize cooperating threads in Unix
- To gain exposure to how a basic web server is structured

ii. Background

In this assignment, you will be developing a real, working web server. To greatly 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 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 initial 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 an 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.fudan.edu.cn:80/index.html identifies an HTML file called "/index.html" on Internet host "www.fudan.edu.cn" 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.**

iii. Basic Web Server

You should copy over all of the code files into your own working directory and 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 localhost and use port number 2016; copy your favorite html file to the directory that you start the web server from.

To view this file from a web browser (running on the same machine for example), use the url:

127.0.0.1:2016/favorite.html

To view this file using the client code we give you, use the command

client 127.0.0.1 2016 /favorite.html

To run the cgi program, output.cgi, using the client code and sending the arguments 1, 5, 1000, and 0, use the command

client 127.0.0.1 2016 "/output.cgi?1&5&1000&0"

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 very 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 (e.g., if the user presses the "stop") 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 programmers don't like to do it because they believe that it makes their code less readable. The solution is to use these wrapper functions. Note the common convention that we use of naming the wrapper function the same as the underlying system call, except capitalizing the first letter, and keeping the arguments exactly the same. **We expect that you will write wrapper functions for the new system routines that you call.**

iv. 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 two key pieces of functionality to the basic web server. First, you make the web server multi-threaded, with the appropriate synchronization. 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).

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 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.

You should investigate how to create and manage posix threads with pthread_create and pthread_detach.

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. **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:

- Any Concurrent Policy (ANY): When a worker thread wakes, it can handle any request in the buffer. The only requirement is that all threads are handling requests concurrently. (In other words, you can make ANY=FIFO if you have FIFO working.)

- 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 Static Content (HPSC): When a worker thread wakes, it handles the first request that is static content; if there are no requests for static content, it handles the first request for dynamic content. Note that this algorithm can lead to the starvation of requests for dynamic content.

- Highest Priority to Dynamic Content (HPDC): When a worker thread wakes, it handles the first request that is dynamic content; if there are no requests for dynamic content, it handles the first request for static content. Note that this algorithm can lead to the starvation of requests for static content.

You will note that the HPSC and HPDC 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: Multi-threaded Client

We provide you with a basic single-threaded client that sends a single HTTP request to the server and prints out the results. 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 request workloads (specified by a new command line argument you will add: 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.

v. Program Specifications

For this project, you will be implementing both the server and the client. Your web server must be invoked exactly as follows:

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 ANY, FIFO, HPSC, or HPDC.

For example, if you run your program as

server 2016 8 16 FIFO

then your web server will listen to port 2016, 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.

Your 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.

vi. Hints

We recommend understanding how the code that we gave you works. We provide the following .c 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".
cse.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. You will be changing this code so that it can send simultaneous requests to your server.
output.c: Code for a CGI program that is almost identical to the output program you used for testing your shell (basically, it repeatedly sleeps for a random amount of time). You may find that having a CGI program that takes a while to complete is useful for testing your server.

We have provided a few comments, marked with "CSE", to point you to where we expect you will make changes for this project.

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.

We recommend that you experiment with the existing code. 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 (e.g., www.fudan.edu.cn). Make small changes to the server code (e.g., have it print out more debugging information) to see if you understand how it works.

We 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.

You may find the following tutorials useful as well.

Linux Tutorial for Posix threads
POSIX threads programming

vii. Due Dates and Deliverables

The project is an individual project. It is due on December 31, 2016 at 5pm.

You should copy all of your source files (*.c and *.h) and a Makefile to your FTP directory. Do not submit any .o files. You do not need to copy any .html files or CGI programs.

In your README file you should have the following 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. For example, how do you implement the ANY policy?