COSC 301 Project 3, Fall 2012: Concurrent Web Server

Due Friday, 9 November 2012

In this project, you'll create a "thread pool" to handle incoming requests to a webserver. To do that, you'll need to implement a producer-consumer pattern to pass incoming requests from the main web server thread (the producer) to a consumer/worker thread to handle the request. You will use pthreads condition variables to make all this work right. In addition to handing requests off to worker threads, you'll also need to *log* the request to a file. Since multiple worker threads will need to write to this file, you'll need to protect access to it.

As with prior projects, you are welcome and encouraged to work with someone else. Please make a comment in your code to indicate who you worked with or who you got help from. Project submission is the same as with past projects: you'll need to keep your code in `git` and submit the repository name.

Overview of HTTP and Web requests

When a web browser makes a request to a web server, it uses a protocol called HTTP, which stands for HyperText Transfer Protocol. HTTP is a text-based protocol, which just means that the protocol contents are all (human-)readable strings.

A typical request sent by a browser to a server looks like:

```
GET /thefile.html HTTP/1.1\r\n
Host: example.com\r\n
User-Agent: Mozilla/5.0\r\n
\r\n
```

The first line is really the most important: it says that the request is to `GET` a file named `/thefile.html` from the server, using a specific version of HTTP (`1.1`). Every HTTP request starts with a line like this, though the specific request verb can vary (`GET` is just one of several verbs). We will only be concerned with `GET` requests in this project. The main request line is followed by a series of key/value "headers" to provide additional information about the request that the server might use. In general, we will be ignoring all these headers (but a full-fledged web server would certainly not).

Note that in the above, the `\r\n` characters indicate that there must be a carriage return (`\r`) and newline (`\n`) at the end of each line, with an empty line at the end. The blank line at the end is *required* by the protocol standard.

A response from the server to the browser for this request might look like:

```
HTTP/1.1 200 OK\r\n
Content-type: text/html\r\n
Content-length: 93\r\n
\r\n
<html><head><title>Hello, world!</title></head>
<body><p>Hello, world!</p></body></html>
```

The top line of the response indicates the HTTP version with which the server is responding, a response code (200) and a response reason (`OK`). The two lines that follow indicate that the content of the response is a text (HTML) file, and the number of bytes in the content is 93. After a blank line, the actual content is shown.

To respond to a `GET` request with *static* content (i.e., not something generated with a modern "web app"), the server must find an actual file corresponding to the name `"thefile.html"`, open it, and read and return its contents

to the browser. It's certainly possible that a file named in a request doesn't exist. In that case, a server would send a "404" response, which looks like this:

```
HTTP/1.1 404 Not Found\r\n
\r\n
```

When browsers and servers communicate using HTTP, these strings are carried over network connections. In this lab, I am giving you functions to handle any network interactions (e.g., receiving a parsing a request, and sending some data back to a browser). You'll only need to handle responding with either a 200 (OK) or 404 (Not found) response, depending whether the file named in a request exists or not.

Requirements

Handling HTTP requests

For this lab, you're given three files for the webserver (along with a `Makefile`), and two Python programs to help with testing your server.

The webserver files are `main.c`, `network.h`, and `network.c`. All this code should compile (just type `make`). When you run `./webserver`, the server will wait for a request (on port 3000), then currently do nothing with it. There are comments in `main.c` (in the function `runserver`) where you will want to add some code to create threads, and hand requests off to threads. A `-t` option can be given to the webserver to indicate the number of worker threads that should be created. When you create your thread pool, you should create exactly that many threads. This number is passed as a parameter to the `runserver` function, which is a function you will need to modify. You shouldn't change anything in the `main` function.

When an individual request arrives at the webserver, a new *socket* is created. A socket is essentially an opaque descriptor that can be used to communicate (using system calls like `read` and `write`) with another network entity. To hand off a request to a worker thread, you'll need to pass it the socket that gets created with a new connection. (This socket is called `new_sock` - see around line 64 in the `runserver` function.)

Once you have the socket, you can use that with two functions that I have written (in `network.h` and `network.c`): `getrequest` and `senddata`. You should take a look at the header and source files to see how these functions are used. In brief, the result of calling `getrequest` is either the filename that is being requested, or failure. The `senddata` call can be used to send data back to a browser; you simply need to pass in C strings (along with the number of bytes to send) in order to send data to a client. You will need to compose and return the HTTP response header, but there are templates already prepared in `network.h` to help you do that, so be sure to read that file.

The filename that you get as a result of calling `getrequest` will look something like `"/thefile.html"`, or `"/adir/thefile.html"`, or even `"thefile.html"`. It should be interpreted as follows:

- If it starts with a slash (`/`), the slash should be ignored.
- The remainder of the string should be treated as a file name relative to the path in which the webserver is running.

So, for example, if you receive the string `"/adir/thefile.html"`, and your webserver was running in the directory `"/home/jsommers"`, you would attempt to open and return the contents of the file `"/home/jsommers/adir/thefile.htr`
If the file exists, you should return an HTTP 200 response (with the correct `Content-length`). If the file does not exist, you should return an HTTP 404. You can test whether the file exists using the `stat` system call, as well as get the file size using that call. To open a file, you can use the `open` system call (`man 2 open`). To read contents of the file, you can use the `read` system call (`man 2 read`). You should always be careful to close files when you're done with them using the `close` system call.

Logging Requirements

For each request you receive, you should make a log entry (one line) in the file `weblog.txt` with the following format:

```
127.0.0.1:57881 Fri Oct 19 15:39:15 2012 "GET /testfiles/fbf4ae32.html" 200 3639
127.0.0.1:57883 Fri Oct 19 15:39:17 2012 "GET /testfiles/c8346c14.html" 200 3620
127.0.0.1:57885 Fri Oct 19 15:39:20 2012 "GET /testfiles/notafile.html" 404 26
```

Each line should start with the client's IP address and port, followed by a timestamp, followed by the HTTP request information (`GET` and the file name requested), the success code (200 or 404), and the *entire* response size (including HTTP protocol information). To see how to obtain the client's IP address and port, see line 68 in `main.c`, which has a `fprintf` statement that shows the client IP address and port in the required format. To print the date in the format above, you can use the built-in `time` and `ctime` functions. See the example also on line 68 of `main.c`.

Tips

Work out how you want to design your pool of threads, and how to hand off incoming sockets (HTTP connections) to a thread from the main server thread. You will almost certainly want to use some kind of producer/consumer model with some number of mutexes (1 or more) and condition variables (1 or more).

Once you work out the thread pool design, you can write the worker thread function, in which you'll get the file being requested (using `getrequest`), figure out whether the file exists (`stat`), then return an HTTP 200 and the contents of the file, or an HTTP 404. *After* you've processed the request, you should log the request to the logfile. Since you'll have multiple threads trying to write to the file, you'll need to make your logging function threadsafe.

Use the Python testing tools (described below) and run your server under `valgrind` to ensure that it doesn't do anything unsavory with memory, and that you don't have any leaks.

Using the Python-based testing tools

There are two Python programs in the repo that are designed to help with testing. The program `mkfiles.py` will create a specified number of dummy files that your webserver can serve, and the program `webdriver.py` can make HTTP requests to your server for the files created by `mkfiles.py`.

To use the tools, can do the following:

```
# In one terminal, start the webserver
$ ./webserver

# In another terminal, create the files and run the driver
$ python mkfiles.py -n 10   # make 10 test files
$
$ python webdriver.py -r 5 # make 5 requests to the server, randomly
                           # selecting from the generated files
```

You only need to generate the files once for testing; once they're made, you can just run `webdriver.py`. To get some help on various options that can be used with the programs, just type `python mkfiles.py -h`, or `python webdriver.py -h`.

Note that the `webdriver.py` program will randomly select files to request, and wait some number of (randomly chosen) milliseconds before requesting the next file. Each request is farmed out to a separate Python thread; take a peek at the source code if you want to see how threads are done in Python (they're much more limited than in C, but they're still certainly useful).

Turning in the project

Just post the name of your github repo to Moodle.