CS 1550 – Project 3: Concurrent Web Server
Due: Sunday, March 24, 2024, by 11:59pm

Description

Launching a browser and visiting a site on the internet involves at least two parties: the web server and the client (you, likely with a browser). In this assignment, we will modify a single process/single-threaded web server program to use parallelism through worker processes created with fork() and compare it to one that uses pthreads.

As we described in lecture, HTTP (HyperText Transfer Protocol) is a simple, text-based communication protocol by which a web browser requests documents from a web server, and the web server replies. For instance, if you visit a web site such as http://www.example.com/test.html, your browser does the following:

1. Connects to the IP address of http://www.example.com/test.html (obtained via DNS lookup) at port 80 (standard web server port)
2. Sends the HTTP request message:

```
GET /test.html HTTP 1.0
Host: www.example.com
```

To which the server replies (assuming it finds the file):

```
HTTP/1.1 200 OK
Date: Tue, 10 Oct 2024 10:24:55 GMT
Content-Length: 4892
Connection: close
Content-Type: text/html
```

Followed by a *blank line* and the *contents of the file* (which in our example is 4892 bytes long).

If the file is not found, it returns the infamous 404 error code like so:

```
HTTP/1.1 404 Not Found
```

We have provided you with an extremely simple server that can handle GET requests with 200 and 404 responses.

What you need to do

Your task is to modify the provides server in the following ways:

1. Make a new server, server_proc.c that when a new connection is accepted, uses fork() to create a worker process that handles sending the requested file over the network connection
2. Make a new server, server_thread.c that when a new connection is accepted, uses pthread_create to make a worker thread that handles sending the requested file over the network connection.
3. Make a new server, server_cached.c based upon the server_thread.c version that holds in memory the 5 most recently requested webpages (so it does not need to access the disk to send() the contents to the client). Note that each request may alter the contents of the cache and that there may even be a worker using the now evicted page. Avoid race conditions.
4. When a worker is done, it should log the request for that particular webpage by appending it to a file named stats_proc.txt and stats_thread.txt, respectively. Note that this file needs to be exclusively accessed, so you'll need to do some sort of synchronization. Log the name of the file requested, its size in bytes, and the elapsed CPU time (in seconds) of the request, in the following tab-separated format:

   ```
   file1.html 13452 0.0123
   ```

   Obtain the elapsed time through the clock_gettime() function from <time.h>. Choose the most appropriate timers from the manpage to get the information you need. Record the time at the start of the worker and the end of the worker and subtract for the elapsed time. Display it in seconds to 4 decimal places of precision.

After you have recorded these statistics, devise an experiment to answer two questions: (1) Between threads and processes, which implementation of the server is faster? and (2) How much does a webpage cache help? Describe your experimental design, your results, and your conclusions in a 1-2 page paper

Ports and Addresses

Port 80 is the normal web server port, which can only be bound as the root user. Luckily for us, our VM allow us to be root and run our server. For an address of the machine, we will simply refer to it as *localhost*, or localhost's dedicated IP address: 127.0.0.1

Additional Setup

We will want to install a couple of programs to help us do this project under the virtual machine from project 2.

As the root user:

```
apt-get update

apt-get install tmux links
```

To get the single-threader server code from thoth:

```
scp USERNAME@thoth.cs.pitt.edu:/u/OSLab/original/server.c .
```

Testing

It can be problematic to reliably connect to a server running in the virtual machine, meaning that you will only be able to connect to your server from inside the VM itself. To do this, your best bet for testing is to use a few programs. Pick the one or ones that seem best for what you're trying to do from:

- nc localhost 80
- wget http://127.0.0.1/page.html
- lynx http://127.0.0.1/page.html

nc (netcat) is a terminal emulator. You can connect directly to your server, and anything you type will be sent when you hit enter. This means you can manually create a GET request, and you will see the reply of the server in plain text on your screen.

wget downloads files from the internet.

links is a text-mode web browser. No graphics, no tables, and minimal font support, but it's a real, working browser.

We'd like to just open a second terminal window and try your testing procedures, but under our VM, we will have to use a terminal multiplexer to do this. A terminal multiplexer will allow us to see the output of our program in one "window" while having a shell in the other. You installed one called "tmux" before.

Do this:

1. Run `tmux`
2. Execute your server
3. Hit **CTRL+B**, release, and then type " (a quotation mark -- this requires holding shift down to type)
4. You should see a window with a shell appear at the bottom
5. At any point, you can switch which shell has focus by typing **CTRL+B** and then pressing an arrow key to move up or down amongst the windows.

6. When you're done testing, type `exit` to leave the shell and the bottom window will go away. You'll probably want to use **CTRL+C** to kill your server. Then you can type `exit` again to leave `tmux`.

Hints and Requirements

- Start with the muti-process version as it isn't much to change.
    - Call `fork()` and in the child, do the response code
    - Note that when you clone a process with fork, you get all of the parent process's file descriptors. This means you'll have two copies of the socket file descriptor and two of the connection file descriptors. The parent only needs one of those and the child only needs one. Make sure to `close()` the other.
- For synchronization, use `sem_wait()` and `sem_post()` from `<semaphore.h>` in the processes, and `pthread_mutex_lock()` and `pthread_mutex_unlock()` from `<pthread.h>` for the multi-threaded version. If you need shared space to hold something between processes, use `mmap()` with the `MAP_SHARED` and `MAP_ANONYMOUS` flags. Anything allocated there will be shared between the parent and any child. Sharing between threads is just allocating things on the heap or as globals.
- When you build a threaded program or use semaprhoes, you'll need to add the `-pthread` option to gcc:

  ```
  gcc -o server_thread server_thread.c -pthread
  ```

- To run two programs named p1 and p2 simultaneously on the command line, put an & between them:

  ```
  ./p1 & ./p2
  ```

  Use this to test your parallelism.
- For testing, you may want to have the process or thread sleep for something like 1 second between sending parts of the response just so it takes longer, and you can verify the parallelism. Remember to remove this code before doing your analysis of the timings.

Submission

You need to submit:

- Your three well-commented programs' C source code (server_proc.c, server_thread.c, and server_cache.c)
- Your writeup with your analysis of the two implementations as USERNAME.pdf

Make a tar.gz file as in the first assignment, named `USERNAME-project3.tar.gz`

Upload the tar.gz file to Canvas by the deadline for credit.