CSCI4730 – Operating Systems

PA #2: Multi-threaded Web Server w/ Synchronization

Due date: 11:59pm, 10/26/2024

Description

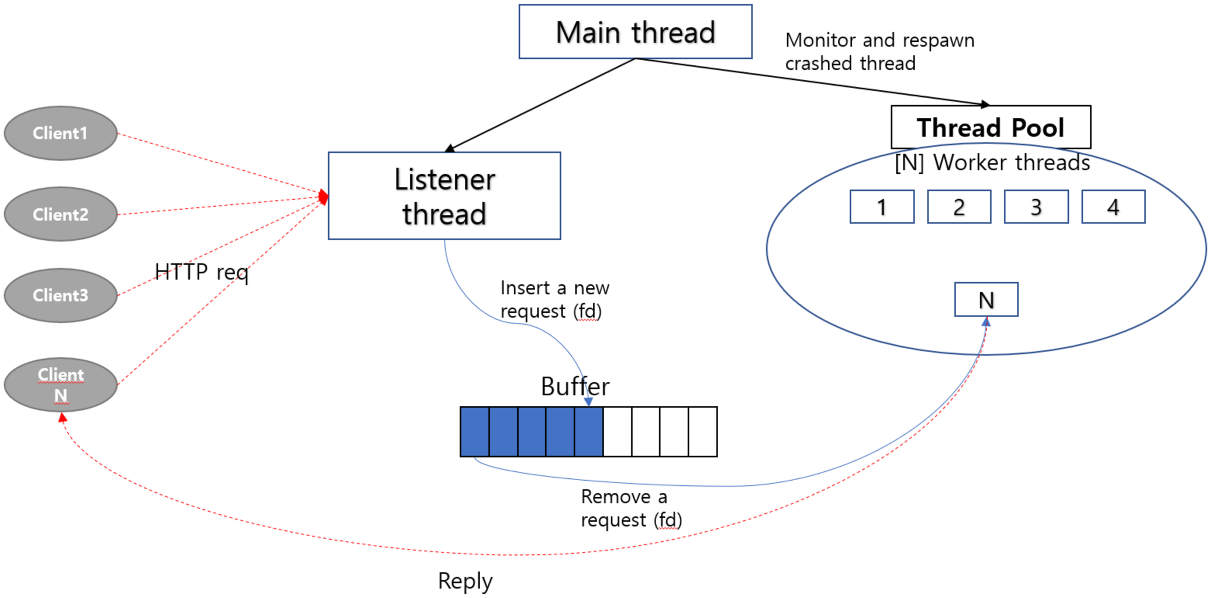
In this programming assignment, you will design and implement a multi-threaded "slow" web server using synchronization tools. The code of a single-threaded web server[1] is provided, and you will convert it into multi-threaded architectures.

Note that you **do not** need to write or modify any code that deals with sockets or the HTTP protocol. They are already implemented in "**`net.c`**" file and you don't need to modify it. Your job is to modify "**`webserver_multi.c`**" file to make it to multi-threaded architecture.

The main limitation of a single-threaded web server is its scalability just like PA #1. Such a server cannot efficiently process many requests from multiple clients simultaneously. To address this limitation, you will convert the web server to a multi-threaded model, which is better equipped to handle multiple client requests at the same time.

One straightforward way to implement a multi-threaded web server is to create a new thread for each incoming request. However, this approach can lead to unnecessary overhead due to frequent thread creations and terminations. To minimize these overheads, you will design and implement a **thread-pool model** for your web server.

[1] The code is slightly modified from http://www.jbox.dk/sanos/webserver.htm



Thread-pool: When the server starts, it creates [N+1] threads, composed of [N] worker threads and 1 listener thread. The main thread monitors the child threads.

In this assignment, you will use **producer-consumer model**. As discussed in the class, you will use two semaphores (**`sem_full`**, **`sem_empty`**), and one mutex lock (**`mutex`**).

1. **Listener (producer) thread** continually listens for client connections. When a request arrives (*i.e.*, a new item), it accepts the request and puts the associated information (file descriptor) into the shared buffer, then begins listening a new request again. If the buffer is full, the listener should wait.
2. **Each worker (consumer) thread** initially waits for the a signal from the listener thread because the buffer is initially empty. When the listener inserts a new request into the buffer, any available worker thread pulls the request information from the buffer and processes it (by calling **`process()`** function). You will need to carefully use semaphores and a mutex lock to avoid concurrency problems, such as race conditions or deadlocks.

- **`webserver.c`** is a single-thread version for your reference.
- **`webserver_multi.c`** is the template for the multi-threaded server. You will modify "**`webserver_multi.c`**" to build your multi-threaded server.
- The number of worker thread (N) is specified by the user via command-line argument. Your program will accept two command line arguments, "port number" and "number of threads".
  - **`./webserver_multi [port] <# of threads>`**
  - *e.g.,* **`./webserver_multi 5000 10`** // port 5000, create 10 worker threads
- You can use client[2] (html client) to test of your server program.
  - **`./client [host_ip or dns] [port] <# of threads>`**

[2] http://coding.debuntu.org/c-linux-socket-programming-tcp-simple-http-client

- *e.g.,* `./client 127.0.0.1 5000 10`
- **`<# of threads>`** is optional. The default is 10.

Submission

Submit a tarball file using the following command:

```
%tar czvf proj2.tar.gz README.pdf Makefile webserver_multi.c
```

1. README file containing:
   a. Your name
   b. List what you have done and how did you test them. So that you can be sure to receive credit for the parts you've done.
   c. Explain your design of shared data structure and synchronization (and show that your solution does not have any concurrency problems). Also, you need to explain how you tested and verified your program against race conditions and deadlocks.
2. All source files needed to compile, run and test your code
   a. Makefile
   b. All source files
   c. Do not submit object or executable files
3. Your code should be compiled and run correctly in odin machine.
   a. **No credit will be given if your code failed to compile with "make" in odin.cs.uga.edu** (we will use your makefile).
4. Submit a tarball through ELC.

Note

- To get full credit, you should correctly use semaphore, mutex lock, and consumer-producer model.
- The maximum number of threads of your webserver is 100.
- With a larger number of threads, the webserver should show the performance improvement (shorter end-to-end request processing time).
- With a smaller number of threads, the webserver should show the performance degradation.