CSCI4730/6730 – Operating Systems

Project #2: Multi-threaded Web Server

Due date: 11:59pm, 3/3/2022

Description

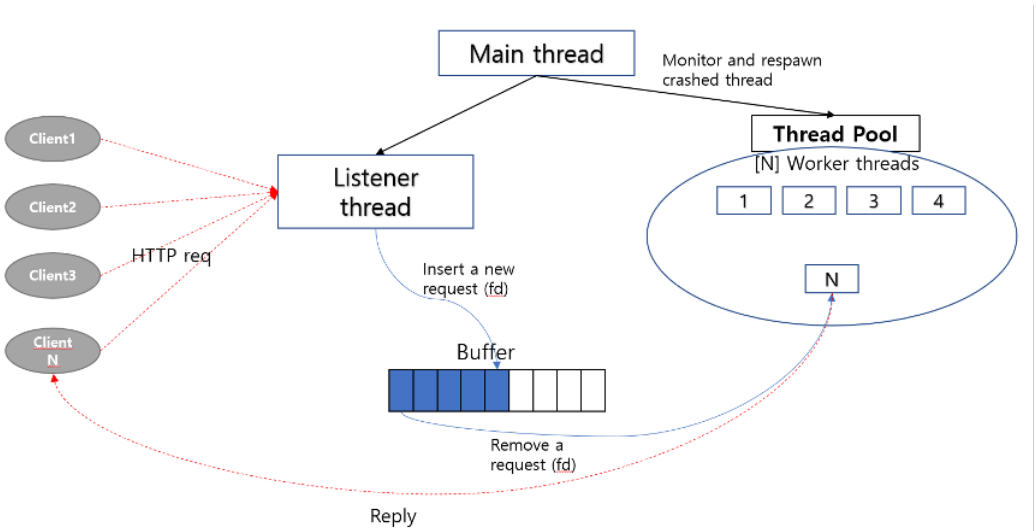
In this project, you will design and implement a multi-threaded web server. The code of the single-threaded version[1] is provided, and you will convert it into a multi-threaded architecture.

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

Part 1. Thread Pool – 60%

The main problem of a single-threaded web server is the poor scalability. It cannot scale up to large numbers of clients. To address the problem, you will convert the web server into the multi-threaded model. The multi-threaded model has a much better performance to handle multiple clients simultaneously.

One way to implement this is to create a child thread for each request. However, it causes unnecessary thread creation and termination overhead. To minimize such overheads, you will design and implement a **thread-pool model**.



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

Thread-pool: When the server launches, we create [N+1] child threads: [N] worker threads, and one listener thread. The main thread only monitors the child threads.

We will use producer-consumer solution in this problem. The synchronizations will include two semaphores (sem_full, sem_empty), and one mutex lock (mutex).

1. **Listener thread** keeps listening for client connections. When a request arrives (i.e., new item), accepts it and puts the request information (file descriptor) into the shared buffer, and starts to listen for a new request again. If the buffer is full, the listener should wait.
2. **Each worker thread** initially waits for the signal from the listener thread because the buffer is initially empty. When the listener inserts a new request into the buffer, any available child thread pulls a request information from the buffer and handles it (call process() function). You will need to carefully use semaphores and lock methods to avoid concurrency problems such as race conditions or deadlocks.

- You will modify "webserver.c" to build a multi-threaded server.
- The number of worker thread(N) is given by the user via command-line argument. Your program will accept 2 command line arguments, "port number" and "number of threads".
  - ./webserver 4000 10 // port 4000, create 10 child threads
- You can use client[2] (html client) to test of your server program.
  - ./client [host_ip or dns] [port] <# of threads>
  - ex) ./client 127.0.0.1 4001 10
  - <# of threads> is optional. The default is 10.
- Partial credit will be given to correct but not efficient solutions (e.g., busy-waiting).

Part 2. Crash Handling – 40%

If one or more child threads crash, we need to create new child threads to keep the same number of threads in the thread pool. Like the first programming project, you will use the command-line argument to simulate the thread crash. However, you do not need to repeat the failed request in the server side. The client re-sends the request when it detects the failure.

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

- You can use 3rd command-line arguments (integer between 1 and 50) to trigger a crash. 50 means each child thread has 50% chance to be killed abnormally by calling "pthread_exit(NULL)". 1 means 1% chance to crash.

  - ./webserver 4000 10 50 // port 4000, create 10 child threads, each thread has 50% chance to crash

- *Hint:* You can use "pthread_tryjoin_np()" to perform a non-blocking join.

  - Details can be found in the man page: "$man pthread_tryjoin_np"

Submission

Submit a tarball file that includes the following files through eLC

1. README file with:

   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

Note: 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** (we will use your makefile).