ASSIGNMENT 5

CS 361: Systems Programming / Spring 2023

Description

In this assignment, you will add threads to make a concurrent HTTP server. You will reuse your implementation of Assignment 4 as the starting point.

Please refer to the specification document for Assignment 4 for details about the HTTP protocol.

Threading models

Your solution will implement two models for creating and managing threads.

Thread per connection

In this mode, the main thread accepts connections and creates one peer thread to handle each connection. The peer thread reads from the socket and handles the request accordingly. Once the peer thread is done, it terminates executing.

Thread pool

In this mode, your server starts with a main thread and a pool of N peer threads that will be reused. The main thread accepts connections and makes them available to the peer threads. Each peer thread executes the following loop: (1) wait for a new client, (2) handle the client, (3) close the connection to the client, and (4) loop back to wait for another client.

You can implement a simple thread pool by following the pattern producer-consumer seen in class.

Concurrency requirements

If any clients access the following URLs, they should all make progress in parallel and without interfering with each other: /ping, /echo, and file downloads. For instance, Client 1 attempts a /ping while Client 2 is downloading a file. Both should make progress in parallel.

Safety requirements

Safe read/write

Reading and writing data with URLs /read and /write, respectively, should be atomic. This means that each write either happens completely, or it does not happen at all. Consider the following scenarios:

Scenario 1 – Good write

1. Client 1 executes in parallel with Client 2
2. Client 1 is performing a /write of data aaaaaaaaaaa
3. Client 2 is performing a /write of data bbbbbbbbbb
4. Later Client 3 performs a /read
5. Client 3 either receives aaaaaaaaaa or bbbbbbbbbb

Scenario 2 – Bad write

1. Same as Scenario 1 until step 4
2. Client 3 reads a combination of a and b. E.g., aaaaabbbbb
3. Your implementation should make sure that this scenario cannot happen

Scenario 3 – Good read

1. Client 1 performs a /write of data aaaaaaaaaaa
2. Later, Clients 2 and 3 execute in parallel
3. Client 2 is performing a /write of data bbbbbbbbbb
4. Client 3 is performing a /read
5. Client 3 either receives aaaaaaaaaa or bbbbbbbbbb

Scenario 4 – Bad read

1. Client 1 performs a /write of data aaaaaaaaaaa
2. Later, Clients 2 and 3 execute in parallel
3. Client 2 is performing a /write of data bbbbbbbbbb
4. Client 3 is performing a /read
5. Client 3 receives a combination of a and b. E.g., aaaaabbbbb
6. Your implementation should make sure that this scenario cannot happen

Safe stats

You can assume that the /stats URL is only accessed by a single client without any other concurrent clients.

When accessing the /stats URL, your server should account for all previous requests accurately. Consider the following scenarios:

Scenario 1 – Good stats

1. There are 100 clients in parallel. 50 use any URL to transfer a total of 10 header bytes and 10 body bytes. 50 access URLs that lead to an error, transferring 10 bytes each.
2. After all 100 clients finish, Client 1 accesses /stat
3. The results should reflect 50 requests and 50 errors, with 500 header bytes, 500 body bytes, and 500 error bytes.

Scenario 2 – Bad stats

1. Same step as Step 1 in Scenario 1
2. Same step as Step 2 in Scenario 1
3. Any of the following happens:
    a. The results have less than 50 requests
    b. The results have less than 50 errors
    c. The results have less than 500 header bytes
    d. The results have less than 500 body bytes
    e. The results have less than 500 error bytes

Implementation guide

Doing the following will result in a simpler implementation:

- Start with a "thread per connection" model, only tests 9 and 10 are about "thread pool"
- Figure out what data is shared among all threads
    - If possible, make that data thread-private as seen in class
    - If not possible, make sure you use critical regions to access shared data
- Minimize the number of critical regions, and keep them as small as possible
- The assignment has two utility methods you should use to read/write HTTP requests/responses
    - `int recv_http_request(int sockfd, char * buffer, int max_size, int opts);`
    - `int send_fully(int sockfd, const char * data, int size, int opts);`
    - These methods are drop-in replacements for recv/send respectively
        - Replace all calls to `recv` with calls to `recv_http_request`
        - Replace all calls to `send` with calls to `send_fully`

Entry Point

You will modify file `a5.c` with your solution and implement each function as defined above.

Due Date and Resubmission Policy

This assignment is due on **April 22nd 2023** (Saturday) at **5pm CST**. There is no late policy.

The code and date used for your submission is defined by the last commit to your Git repository.

To resubmit this assignment, you are required to have attendance to Lab Sessions 10, 11, and 12. You can resubmit your assignment until **April 29th 2023** (following Saturday) at **5pm CST**. Together with your resubmission, you will have to submit a written description of what you changed from the original submission (on Gradescope).

Bonus Points

This assignment has a total of **25% bonus points**:

- 10% can be earned by using Piazza as described in the syllabus. Your posts should be public, tagged with the `assignment5` label, and non-anonymous to the instructors to count towards the bonus. You can claim bonus points through **a Gradescope quiz**.
- 15% can be earned by completing Lab Sessions 10, 11, and 12.

Submission and Grading

This assignment is submitted through Github, and has an automatic grade component of 80%. You can check your current grade at any point by submitting your code and checking the autograder. The automatic grade is determined by 8 tests, that will check if your submission outputs the expected result. Each test is worth 10%.

Questions

Tests 7 and 8 are non-deterministic, and they can pass with incorrect solutions depending on thread scheduling. You will have to answer the following questions (on Gradescope) to get full credits:

1. What measures does your solution take to ensure safe read/write while keeping the required concurrency requirements?
2. What measures does your solution take to ensure safe stats while keeping the required concurrency requirements?

Thread pool

Once you implemented all initial 8 tests, you can then pass the compilation option -DT910 by editing the provided Makefile. Now, the startup code will allow threads to be created only at the start, will only allow a certain number of threads to be created, and will terminate the server if any thread exits. This will cause all your tests to break until you fix them by implemented the thread pool model, as described above.

Tests 9 checks that the server was compiled with -DT910 and tests 1-4 are passing.

Test 10 checks that the server was compiled with -DT910 and tests 5-8 are passing.

Errors and Omissions

If you find an error or an omission, please post it on Piazza as soon as you find it.

Hardcoding and Academic Integrity

Any hardcoding will result in a 0% grade. Hardcoding is when you submit code that detects which test is being run, and simply outputs the expected result. For instance, detecting that test 22 is running, and replacing the usual execution of your submission with `printf("expected result")`.

The academic integrity policy described in the syllabus applies to this assignment. You are responsible for writing all the code that you submit. We will use an automatic tool that detects plagiarism on all submitted code, and we will investigate all instances where plagiarism is more than likely.

Please refer to the syllabus for the full academic integrity policy.