ECE 420 Parallel and Distributed Programming Lab 2: A Multithreaded Server to Handle Concurrent Read and Write Requests

Winter 2024

In this lab, you will implement a multithreaded server that can simultaneously handle multiple client requests, with each request issuing a read or write operation applied onto a random position in a global array of strings managed by the server. To simulate multiple simultaneous requests, a multithreaded client program is provided to launch read/write requests using Pthreads. The client and server should talk to each other via TCP sockets. You will be asked to optimize the processing speed of your server program by reducing the array access latency based on what you have learned in this course.

1 Background

Most clients and servers communicate by sending streams of bytes over TCP connections. This ensures that the bytes will be received error-free and in the order they were sent. For TCP connections, *sockets* refer to the endpoints of a connection between a client process and a server process.

A server or (a service) can be uniquely identified by an `IP:port` pair, where a port is a unique communication end point on a host, represented by a 16-bit integer. For example, a server process with a socket address of 127.0.0.1:3000 runs on the localhost at port 3000. A client can connect to the server provided that it knows the socket address of the server, i.e. its `IP:port`.

Sockets define the operations for connection creation, network attachment, data transmission, and connection termination in client-server communication. To demonstrate the communication between a simple echo server and a client, we have provided the code `demos/simpleServer.c` and `demos/simpleClient.c` in the development kit. Specifically, the client process records a text string from the user input and sends it to the echo server at 127.0.0.1:3000. The echo server then sends the received text string back to the client for display. In other words, the provided code in `demos/simpleServer.c` and `demos/simpleClient.c` presents the server-client communication functions necessary for this lab.

1.1 The Case of a Single Client

We present here the actions taken by both the server and client under the scenario where only a single client is interacting with the server. Specifically, the server conducts the following actions:

Algorithm 1 Server actions in `simpleServer.c` when there is a single client.

Create a socket using Socket(), which is identified by `serverFileDescriptor`;
Bind the server socket to the address 127.0.0.1:3000;
Listen on the server socket;
Block at the Accept() function until there is an incoming client connection;
Receive data from the client;
Send the received string data back to the client.

In tandem, the client conducts the following actions:

Algorithm 2 Client actions in the sample code `simpleClient.c`

Create a socket using Socket(), which is identified by `clientFileDescriptor`;
Connect to the server on 127.0.0.1:3000 using Connect();
Send data to the server using Write();
Receive the echoed data from the server using Read() and display it.

1.2 A Multithreaded Server to Handle Multiple Clients

You may have already noticed that the server in `demos/simpleServer.c` has employed multithreading to handle multiple simultaneous client requests. In this case, upon accepting an incoming client connection, the Accept() function will create a new socket, as identified by the `clientFileDescriptor` of the accepted connection, and launch a new thread to handle the connected client. In the meantime, the server still listens for and will accept future incoming clients on the original socket, as identified by the `serverFileDescriptor`.

For more background on stream socket communication, please refer to the following materials:

- Slides by Jeff Chase, Duke University
  `https://users.cs.duke.edu/~chase/tcp-chapter.pdf`
- Beej's Guide to Network Programming
  `https://beej.us/guide/bgnet/html/`

However, you do not need a deep understanding of network programming. For the purpose of completing this lab, it is sufficient for you to understand and use the sample code provided in `demos/simpleServer.c` and `demos/simpleClient.c`.

2 Concurrent Writes and Reads to a Common Data Structure

In this lab, you will launch multiple threads in your client program to connect to a multithreaded server. The server maintains an array of strings, and each client thread will issue a request to perform either a read or write operation on a random position (string) in the array. In the case of a read, the server sends to the client thread the string stored at the specified array position. In the case of a write, the server will first update the string stored at the specified array position with a new client-supplied string and then return to the client the updated string from the same array position. For both requests, the client thread terminates after receiving a response from the server. About 30% of all requests are writes and 70% are reads.

Notice that there could be multiple concurrent read or write requests operating on the same array position. To solve this problem, a synchronization mechanism (system) needs to be implemented to protect the critical sections and avoid race conditions. However, while various solutions exist, better synchronization designs can result in a noticeably shorter array access latency when compared to worse designs, which translates to faster response time from the perspective of the client. **Thus, your solution for protecting the critical sections should avoid unnecessarily impeding the actions of concurrent threads in the multithreaded server.**

2.1 Array Access Functions

We have provided the functions `setContent` and `getContent` in `common.h` for string access in the array. Note that both of these functions perform a short pause during each execution. The pause artificially simulates the latency of array access, ensuring that resource contention between threads occurs (such as when two threads attempt to concurrently access the same array position). As the purpose of this lab is to devise and optimize an implementation to protect critical sections, this resource contention is necessary. Thus, you must use these functions to realize read (`getContent`) and write (`setContent` then `getContent`) operations.

2.2 Message Parsing

A request from a client thread consists of three variables: the array position, the array access operation (read or write), and the supplied string (for write requests). To transfer these variables over a socket connection, the client thread combines these variables into a single request message and then sends the message to the server. The server, upon receiving the message, then extract these variables to obtain the client request for subsequent array access. In this lab, the client request message is formed as "XXX-Y-SSSSSS", where "XXX" is the position, "Y" indicates whether it is a read request (1 for read and 0 for write), and "SSSSSS" is the string to be written. The provided `ParseMsg` in `common.h` is provided to decompose the message into a predefined structure named `ClientRequest`.

3 Tasks and Requirements

Tasks: Implement a multithreaded server to handle concurrent array accesses from multiple client requests. The server must ensure that the critical sections are properly protected while minimizing the array access latency. Specifically, the server should implement the following.

1. The server should be able to handle multiple simultaneous incoming TCP connections generated from a multithreaded client (see `client.c` in the development kit). The server must communicate with the client via stream sockets, as shown in the sample code.

2. The server should maintain an array of n strings (`char** theArray`), where each string i ($i = 0, 1, 2, \ldots, n-1$) is filled with the initial value "String i: the initial value". (The code `demos/arrayRW.c` is provided for reference only.)

3. For a read request, the server should obtain the string from the specified array position using `getContent` and send it back to the client. For a write request, the server will update the specified array position with a supplied string `setContent`, obtain the string from the same array position using `getContent`, and then send it back to the client.

4. Critical sections caused by concurrent access on the array must be protected by a synchronization mechanism to ensure correct results for both the read and write operations. Additionally, the synchronization mechanism should be efficient by avoiding unnecessary impediments of concurrent threads and other unnecessary operations. **Note that your server will be marked on both correctness and server access latency.**

Specific Programming Requirements and Remarks:

1. Check the sample codes in `demos/` of "Development Kit Lab 2" for 1) how to perform client-server communication using stream sockets; 2) how to implement a multithreaded server; 3) how to handle race conditions and critical

sections with a basic approach, i.e., protecting the entire array with a single mutex. **Note that the synchronization design here incurs high server access latency, and is therefore not optimal.**

2. Check the `common.h` file. You must use the provided `setContent` and `getContent` to read and write your string array, and `ParseMsg` to parse the message from client requests.

3. The server should open `COM_NUM_REQUEST` = 1000 threads to handle the concurrent client requests, as defined in `common.h`. Note that the client can be compiled from the file `client.c`.

4. The server, when compiled, should be an executable named `main`. It should have the following three command line arguments: n positions in the string array, server IP, and server port. You must write the arguments in this order.

5. Make sure that your server guarantees correct read/write results. The file `attacker.c` is a client for checking correctness. Run it multiple times to test your server.

6. For each client request, you should measure the *array access latency*. This is the time that the server takes to complete a read or write request, not including the socket communication time. This latency is the period after the server finishes parsing the received client request and before the server begins sending the response back to the client.

7. Your server should record the *average* array access latency to process the 1000 client requests (threads) as the runtime. Use the provided `saveTimes` function in `common.h` to save your result. You may use `test.sh` in the development kit to repeatedly run your client several times.

8. Minimize the average array access latency of your server program as much as possible. Optimize your server design by reducing unnecessary overheads and impediments caused by your synchronization mechanism. Also, avoid running unnecessary operations in your implementation (i.e. remove all print tests before submission).

Submission:

One member in each team is required to submit a zip file to eClass before the submission deadline. While we do not enforce a naming convention for the submissions, a good template is "user**_lab2.zip" (where ** is the user number). The zip file should contain the following:

1. "Makefile": By executing the `make` command, the solution executable named "main" should be generated. Please do not use any optimization flags (i.e. no `-O3`).

2. Solution source files: You should include all source files necessary to compile your solution executable. Note that you do not need to include "demos/", "test.sh", "client.c", or "attacker.c".

4 Marking Guideline

4.1 Marking Session

Each group is required to present a short demo of their code within an appointed in-lab time-slot. Each demo consists of the components:

1. **Demo:** Upload and compile your eClass submission and verification code (to be provided during the marking session) in your VM. Demonstrate that your code works for different array sizes (n), as specified by the marker (a TA or LI). This includes verifying the correctness of your results and comparing the server access latency achieved relative to the optimal results prepared by the LI and TAs.

2. **Presentation:** Provide a short (1-minutes) verbal explanation of the server design. Focus on the strategy implemented to ensure efficient protection of critical sections.

3. **Group Response:** The marker will ask some questions to the group. Marks will be assigned to the group based on your collective response to these questions.

4. **Individual Response:** The marker will ask some questions to each group member. Marks will be assigned to each individual based on their individual response (without assistance from other group members) to these questions.

To expedite the marking process, please rehearse your demo beforehand. Be aware that:

1. You should practice the process of loading, unzipping, and compiling your eClass submission in the VM in an efficient manner.
2. The marking process is timed. While there should be ample time for the demo, the marker may cut you off if the demo runs over the time limit.
3. The questions are sampled from a pool, and may be different for different groups.
4. The contents of the questions asked by the marker may include, but are not limited to:
   (a) explaining observations from the results,
   (b) describing anticipated program performance under certain scenarios,
   (c) providing potential improvements on the existing solution to address specific scenarios,
   (d) describing certain components within the solution code,
   (e) explaining concepts and techniques learned in the lecture that are relevant to this lab.

4.2 Marking Rubric

Successful code compilation:	1
Presentation of implemented critical section protection strategy:	1
Correct server handling of read and write operations:	2
Runtime is competitive and comparable to the optimal results:	2
Group questions:	2
Individual questions:	2
Total:	**10**