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

Winter 2022

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 stored on the server's main memory. To simulate multiple simultaneous requests, a multithreaded client program is provided to launch read/write requests in 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 and reduce the server memory 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, and associated with a process. For example, a server process with a socket address of 127.0.0.1:3000 runs on the localhost and on port 3000. A client can establish connection to a server as long as it knows the server's socket address, i.e., the `IP:port` pair of the server.

Sockets define operations for creating connections, attaching to the network, sending/receiving data, and closing connections in client-server communication. Please see the provided code `demos/simpleServer.c` and `demos/simpleClient.c` in the development kit, which illustrate the communication between a simple echo server and a client. 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. Note that the server is a multithreaded server and can handle multiple client connections simultaneously. In other words, the provided code in `demos/simpleServer.c` and `demos/simpleClient.c` illustrates the server-client communication functions necessary for this lab.

1.1 The Case of a Single Client

We first explain what happens at both the server and client sides when there is a single client. The client does the following things:

Algorithm 1 Client actions in the sample code `simpleClient.c`

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

The server does the following things:

Algorithm 2 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 the received string data back to the client.
```

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 (on a different port of the server) identified by the integer `clientFileDescriptor` for the accepted connection, and launch a new thread to handle the connected client. In the meantime, the server still listens to and accepts incoming clients on the *original* socket `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 will be sufficient if you can 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 multi-threaded server. The server maintains an array of strings in its memory, and each client thread will perform either a write or a read to a random position (string) in the array. In the case of a read, the server sends back the corresponding requested string to the client thread, which will close the connection and terminate after receiving the returned string. In the case of a write, the server will first update the corresponding string in the array with the new text string supplied by the client and then return the updated string (from the array) to the client. Afterwards, the client thread closes the connection and terminates upon receiving the returned string. 30% of all the requests will be writes and 70% will be reads.

Note that there could be multiple concurrent read or write requests operating on the same string in the array. In this case, we need to protect the critical sections and avoid race conditions. But how can we achieve the fastest processing speed? Think about the following questions. Should we use a single mutex for the entire string array or a different mutex for each string? What is the tradeoff here? Should we use read/write locks? And should we use a read/write lock for the entire array or one for each string? What other optimization can you do to further enhance the performance?

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 increases the latency of array access, ensuring that resource contention between threads occurs (such as two threads attempting to concurrently access the same string in the array). As the purpose of this lab is to explore the various methods to protect critical sections, this resource contention is necessary. Thus, it is **mandatory** that you use `setContent` and `getContent` to write and read strings in the array.

2.1 Message Parsing

In the socket mechanism, messages are sent in terms of strings and each message is a single string. However, for a request from client, we usually need to send multiple variables at a time (in our case, we need to send the position of the string array, whether it is a read or write request, and the string to be written). Therefore we need some approach to combine these information into a single string in the client and decompose those variables in the server. In this lab, the request from the client is in the form of "XXX-Y-SSSSSS", where "XXX" is the position, "Y" is 0 or 1 indicating 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` will decompose these information into a predefined structure named `ClientRequest`.

3 Tasks and Requirements

Tasks: Implement four different multithreaded servers, each employing a different scheme to protect the critical sections for handling the read or write requests given by a multithreaded client (`client.c` in the development kit). Each server must fufill the following requirements:

1. The server should be able to handle multiple simultaneous incoming TCP connections from client requests. The server must communicate with clients via stream sockets, as shown in the sample code.
2. The server should maintain an array of n strings (`char** theArray`) in the memory of the server, where each string i ($i = 0, 1, 2, \ldots, n-1$) is filled with the initial value "String i: the initial value".
3. For a read request, the server will send the requested string from the array back to the requesting client thread. For a write request, the server will update the specified string in the array and then send the updated string (retrieved from the array) back to the client.

For the four different critical section protection schemes, the following implementations are required:

1. A single mutex protecting the entire array.
2. Multiple mutexes, each protecting a different string.
3. A single read/write lock protecting the entire array.
4. Multiple read/write locks, each protecting a different string.

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 multi-threaded server; 3) how to handle race conditions and critical sections with a basic approach, i.e., protecting the entire array with a single mutex.
2. Each 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`.

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

4. Each server should be executable with the following three command line parameters: size of the string array, server ip, server port. You must write the parameters in this order.

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

6. Measure the *average memory access latency* to process the 1000 client requests (threads). Use the provided `saveTimes` function in `common.h` to save your result. Rerun your client program for 100 times and repeat such time measurement for each run. You may use `test.sh` in the development kit to repeatedly run your client program for a number of times. **Note:** for time measurement purposes, you should disable message printing to reduce overhead.

7. The *memory access latency* for each client request is the time that the server takes to complete the read or write request, not including the socket communication time. This is measured as the time period after the server finish receiving the client request and before the server begin sending the response back to the client.

8. Minimize the memory access latency of your server program as much as possible.

Lab Report Requirements:

1. Describe your implementations clearly.

2. Using a table, compare the mean (and/or median) processing times of different server implementations (averaged over 100 runs) in terms of the *average memory access latency* of the 1000 client requests. You should repeat this experiment with different n, the size (number of strings) of `theArray`.

Specifically, n should take the values 10, 100, and 1000. However, you may experiment with other n values to support your discussion.

3. Explain your observations.

4. Please refer to the "Lab Report Guide" for the general requirements on formatting and content.

Submission Instructions:

Each team is required to submit a zip file to eClass by the submission deadline. The zip file should be named "StudentID.zip", where "StudentID" is the Student ID of **one** of your group members (it does not matter which member, though this should be consistent for all submissions throughout the course). The zip file should contain the following three folders:

1. "Code": this folder should contain all the codes necessary to compile the solution executables, including:

   (a) "Makefile": the makefile to generate the executable. By executing the "make" command, the solution executables named "main1", "main2", "main3", and "main4" are generated. Additionally, the solution executable "main$K$" should be individually generated by the command "make main$K$"

   (b) "readme": provide a short descriptor (1 sentence) identifying the critical section protection scheme implemented for each solution executable.

   (c) All the necessary source files to build the executables.

2. "Members": this folder should contain a single text file *named* "members.txt", listing the student IDs of ALL group members, with each student ID occupying one line.

3. "Report": this folder should contain the lab report, saved as a single PDF file *named* "lab_report.pdf".

Note: you must use the file names suggested above. File names are case-sensitive. Please generate the required zip file by directly compressing all of the aforementioned folders, rather than compressing a parent folder containing these folders as subfolders.

Example:

Consider that a lab group consists of the students Alice (Student ID 1234567), Bob (Student ID 7654321), and Charlie with (Student ID 4352617). The group has decided that Charlie is to handle their lab submission:

Charlie should submit a zip file named "4352617.zip", which should contain:

1. "Code" folder with the "Makefile", "readme" and all necessary source files.
2. "Members" folder containing the "members.txt" file. The "members.txt" file should have three lines: the first line being "1234567", the second line being "4352617", and the third line being "7654321". Note the Student IDs do not need to be presented in a particular order.
3. "Report" folder containing the "lab_report.pdf".

A Appendix: General Marking Requirements

Code (5 Marks):

- 1st scheme to handle critical section
- 2nd scheme to handle critical section
- 3rd scheme to handle critical section
- 4th scheme to handle critical section
- The fastest speed you have achieved

Report (5 Marks):

- Clear description of your implementations
- A table comparing the different schemes over different array sizes
- Performance interpretation