CIS 3207 Assignment 3
Multi-Threaded Echo Server

10 points

Project Overview

Echo servers are often used to demonstrate and test networked communication. In this assignment, you'll create an echo server and echo client and evaluate their performance. The purpose of the assignment is to gain some exposure and practical experience with multi-threaded programming and the synchronization problems that go along with it, as well as with writing programs that communicate across networks.

You'll learn a bit about network sockets in lecture and lab. Much more detailed information is available in Chapter 11 of Bryant and O'Hallaron, and Chapters 57-62 in Kerrisk (see Canvas Files: Additional Textbook References). Beej's Guide and BinaryTides' Socket Programming Tutorial are potentially useful online resources.

For now, the high-level view of network sockets is that they are communication channels between pairs of processes, somewhat like pipes. They differ from pipes in that a pair of processes communicating via a socket may reside on different machines, and that the channel of communication is bi-directional.

Much of what is considered to be "socket programming" involves the mechanics of setting up the channel (*i.e.*, the socket) for communication. Once this is done, we're left with a *socket descriptor*, which we use in much the same manner as we've used descriptors to represent files or pipes.

In this project you will develop a server program that echoes a text message on demand. Your echo server is to be a process that will read sequences of words (sentences) sent by clients. The sentences (text strings), sent by a client computer, will be sent back to the requesting client.

Server Program Operation

Server Main Thread

Your server program should take as a command line several control parameters. The first parameter is the maximum length of message that it can receive. If none is provided, DEFAULT_LENGTH is used (where DEFAULT_LENGTH is a named constant defined in your program). The program should also take as a parameter a port number on which to listen for incoming connections. Similarly, if no port number is provided, your program should listen on DEFAULT_PORT (defined in your program).

Three other parameters are the number of element cells in the ‘connection buffer’, the number of worker threads, and the terminator character to end echoing for this client. These parameters are discussed in this document.

The main server thread will have two primary functions: 1) accept and distribute connection requests from clients, and 2) construct a log file of all echo activities.

When the **server** starts, the main thread creates a fixed-sized data structure which will be used to store the socket descriptor information of the clients that will connect to it. The number of elements in this data structure (shared buffer) is specified by a program input parameter (‘size of the connection buffer’). The main thread creates a pool of worker threads (‘the number of threads’ specified as a program parameter), and then the main thread immediately begins to behave in the following manner (to accept and distribute connection requests):

```
while (true) {
  connected_socket = accept(listening_socket);
  add connected_socket information to the work buffer;
  signal any sleeping workers that there's a new socket in the buffer;
}
```

A second server thread will monitor a log queue and process entries by removing and writing them to a log file.

```
while (true) {
  while (the log queue is NOT empty) {
     remove an entry from the log
     write the entry to the log file
    }
 }
```

Connection Buffer Data

The cells in the connection buffer are filled by the main server thread. The cells are processed by the worker threads. Each cell is to contain the **connection socket, and the time at which the connection socket was received**.

Worker Thread

Each server worker thread's main loop is as follows:

```
while (true) {
  while (the work queue is NOT empty) {
    remove a socket data element from the connection buffer
    notify that there's an empty spot in the connection buffer
    service the client
    close socket
  }
}
```

and the client servicing logic is:

```
while (there's a message to read) {
  read message from the socket
  if (the message is NOT the message terminator) {
    echo the reversed message back on the socket to the client;
  } else {
    echo back on the socket "ECHO SERVICE COMPLETE";
  }
  write the received message, the socket response message or “ECHO SERVICE COMPLETE”) and
  other log information to the log queue;
  }
```

We quickly recognize this to be an instance of the Producer-Consumer Problem that we have studied in class. The work queue (connection buffer) is a shared data structure, with the main thread acting as a producer, adding socket descriptors to the queue, and the worker threads acting as consumers, removing socket descriptors from the queue. Similarly, the log queue is a shared data structure, with the worker threads acting as producers of results into the buffer and a server log thread acting as a consumer, removing results from the buffer. Because we have concurrent access to these shared data structures, we must synchronize access to them using the techniques that we've discussed in class so that: 1) each client is serviced, and 2) the queues do not become corrupted.

Echo Thread Processing

The server inserts socket descriptions into the buffer in the order received. Worker threads remove the socket descriptors in FIFO order from the buffer.

Once the message is received and the response sent to the client, the worker thread will create an entry in the log buffer. The log buffer entry is to contain the arrival time of the request, the time the reversal was completed, the message received, and the reply message sent to the client.

Synchronization

Correctness

Only a single thread at a time may manipulate the work queue. We've seen that this can be guaranteed through the proper use of mutual exclusion. Your solution should include synchronization using **locks and condition variables**.

No more than one worker thread at a time should manipulate the log queue at any one time. This can be ensured through the proper use of mutual exclusion. Again, synchronization should be using **locks and condition variables.**

Efficiency

A producer should not attempt to produce if the queue is full, nor should consumers attempt to consume when the queue is empty. When a producer attempts to add to the queue and finds it full, it should cease to execute until notified by a consumer that there is space available.

Similarly, when a consumer attempts to consume and finds the queue empty, it should cease to execute until a producer has notified it that a new item has been added to the queue. As we've seen in class, locks and condition variables can be used to achieve this. Your solution should not involve thread yields or sleeps.

Code Organization

Concurrent programming is tricky. Don't make it any trickier than it needs to be. Bundle your synchronization primitives along with the data they're protecting into a struct, define functions that operate on the data using the bundled synchronization primitives, and access the data only through these functions. In the end, we have something in C that looks very much like the Java classes you've written in 1068 and 2168 with some "private" data and "public" methods, or like monitor functions. Code and some very good advice can be found in Bryant and O'Hallaron Chapter 12.

Testing your program

At the beginning, as you are developing your server, you'll probably run the server and a client program on your own computer. When doing this, your server's network address will be the *loopback address* of 127.0.0.1. (do some research on this).

You are to write a basic client to test your server. This is part of the assignment. For initial testing of communication with the server, you could also use the Unix telnet client, which, in addition to running the telnet protocol, can be used to connect to any TCP port, or you could use a program such as netcat. You will need to use your developed client to test and demonstrate your solution.

Once you're ready to deploy your program on a real network, please restrict yourself to the nodes on cis-linux2(system list). Start an instance of your server on one of the cis-linux2 systems and run multiple simultaneous instances of your client on other systems.

You should use many instances of clients requesting echo services at the same time (for the demo, use of multiple clients is required). These clients should be run from more than 1 computer system simultaneously, i.e., each client computer system should run many client instances at the same time. Your testing and demonstration should show this.

The server program has options for buffer size and number of worker threads as parameters. **You must use variations in these parameters to demonstrate that you are able to ensure proper synchronization and performance under varying loads of requests (number of simultaneous clients and frequency of requests).**

There will be **weekly deliverables** and they are to be **submitted to Canvas** as well.

WEEK 1
Create main server thread, worker threads and log manager thread.
Create buffer and management for socket descriptor elements from main server thread
Create management of socket descriptors by worker threads
Create main thread socket for listening for incoming requests from clients
Create client send request socket for communication with main server thread
Create worker thread insertion in the log buffer

Week 2
Create worker thread communication with the client for echoing
Create log manager thread updates to log file
Create client generation and message sending
Testing with single and multiple clients and varied intervals of requests, buffer sizes and numbers of worker threads for message echo
Project completion and submission for demo