CIS 3207 Assignment 3
Networked Spell Checker

10 points

Project Overview

Spell checkers are useful utility programs often bundled with text editors. In this assignment, you'll create a networked spell check server. 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. Beej's Guide and BinaryTides' Socket Programming Tutorial are potentially useful online resources.

For now, the high-level view of sockets is that they are communication channels between pairs of processes, not unlike pipes. They differ from pipes in that a pair of processes communicating via a socket may reside on different machines, and that the channel 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.

Your spell check server is to be a process that will read sequences of words. If a word is in its dictionary, it's considered to be spelled properly. If not, it's considered to be misspelled. The dictionary itself is nothing but a very long word list stored in plain text form. On cis-linux2, one is available for your use in /usr/share/dict/words. (Here is a downloadable local copy.) You are not required to do any more sophisticated matching, for example, recognizing that perhaps the word "derps", which is not in the dictionary, might be the plural of "derp", which is in the dictionary. (For the record, neither "derp" nor "derps" is in cis-linux2's dictionary, although the dictionary does seem to include many of the forms of many words.)

Server Program Operation

Server Main Thread

Your server program should take as a command line argument the name of a dictionary file. If none is provided, DEFAULT_DICTIONARY is used (where DEFAULT_DICTIONARY is a named constant defined in your program). The program should also take as an argument 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).

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

When the **server** starts, the main thread opens the dictionary file and reads it into some data structure accessible by all of the threads in the program. It also creates a fixed-sized data structure which will be used to store the socket descriptors of the clients that will connect to it. This data structure will also be accessible to all threads. The main thread creates a pool of NUM_WORKERS worker threads, and then immediately begins to behave in the following manner (to accept and distribute connection requests):

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

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
    }
 }
```

Worker Thread

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

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

and the client servicing logic is:

```
while (there's a word left to read) {
  read word from the socket
  if (the word is in the dictionary) {
    echo the word back on the socket concatenated with "OK";
  } else {
    echo the word back on the socket concatenated with "MISSPELLED";
  }
  write the word and the socket response value (“OK” or “MISSPELLED”) 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 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.

Synchronization

correctness

No more than 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 attempts at 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, attempts at 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 could be used to achieve this. Again, your solution should not involve thread yields or sleeps.

the dictionary

We need to be very careful about how we access the work queue, but what about the dictionary? Is the dictionary not a shared resource that is accessed concurrently? Does it not require protection?

Once the dictionary is loaded into memory, it is only read by the worker threads, not modified, so we don't need to protect it with mutual exclusion.

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.

You may write a basic client to test your server, however, you are not required to submit one for the assignment (see extra credit for developing a client). 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 like netcat. You will need to use a client to test and demonstrate your solution.

You're also welcome to use this very basic Python client.

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 a client on other systems.

You should have many instances of clients requesting spell check services at the same time. 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.