CIS 3207 Assignment 3
Spell Checker

Due: Friday, April 5

10 points

description

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 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 a bi-directional one.

Much of what is considered to be "socket programming" involves the mechanics of setting up the channel (*i.e.*, the socket). 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 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 it does seem to include many of the forms of many words.)

program operation

Your 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`.

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 will also be accessible to all threads. It creates a pool of `NUM_WORKERS` worker threads, and then immediately begins to behave in the following manner:

```
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;
}
```

worker

A worker thread's main loop is as follows:

```
while (true) {
   while (the work queue isn't 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";
   }
}
```

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 sockets from the queue. Because we have concurrent access to this shared data structure, we must synchronize access to it using the techniques that we've discussed in class so that: 1) each client is serviced, and 2) the queue does 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 a mutex. Your solution should not include any attempt at synchronization through: busy waiting, thread yields, or sleeps.

efficiency

Producers 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, semaphores 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 mutexes, *etc*.

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. We've seen code like this in lecture. More code and some very good advice can be found in Bryant and O'Hallaron Chapter 12.

testing your program

As you're first developing your server, you'll probably run it as well as 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. 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'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.

Extra credit (+possible 2 points)

Develop a client process to be used to request spell checking from the server process of Project 3.

The client process should create N threads, each of which will repeatedly send a random word to the server for spelling analysis. The client process should be able to be used on multiple client machines simultaneously.

The client process should log the activities of its worker threads. For each worker thread transaction with the server, the log file should include a record: Thread number, time stamp of send request, word requested, time stamp of received response, response value.

Be sure that the resolution of time values is sufficient to determine time differences. Note that the same mechanism used in your server activity logging can be used for logging in the client development.