Steve Tolvaj

CIS-3207

Project 3: Networked Spell Checker

11/14/2021

Program Description

The goal of the Networked Spell Checker program is to act as a server for clients to connect to the program and spell check words sent by them. These connections are maintained until the client disconnects. While the client is connected, the program will check words that are entered by the client and sent to the server. The server will then compare the words that were sent to a dictionary and send back the word with either ": OK" or ": MISSPELLED" to indicate the spell checking was completed. The spell checking will be completed using multiple (worker) threads.

There will also be a log file that is generated based on the worker thread's output. The log file will contain several types of data. The first type will be the time the spell check of a word was requested in seconds and microseconds. The next set of data will be the time the spell check was completed in seconds and microseconds. The last set of data will be the word that was checked with the result (OK or MISSPELLED) and the original priority of the network connection that was accepted. The output of the log will be in the multiple worker threads which will put it into a circular array log buffer. A single logger thread will then work off this buffer and write it to a file.

The server can accept multiple different command-line arguments. The main arguments that must be supplied for the program to run are "./spell_checker <max connection buffer size> <number of threads> <0 for FIFO or 1 for random priority of connection processing>". An example of this would be "./spell_checker 12 5 1". The dictionary and network port number may also be specified in these arguments and must be specified before the previously mentioned arguments in any order. Both, none, or either dictionary path and/or port number may be specified. If they are not supplied, the default port used is 8889 and a dictionary.txt file is used.

A client was also created for the server to accept requests from. The client.c file contains a simple client program to connect to the server that defaults to IP address 127.0.0.1 and port 8889 if no command-line arguments are used or both IP address and port must be specified. i.e. either ./client or ./client <IP><Port> must be used.

Program Design

There will be three types of defined structures used. A client_socket will hold the socket descriptor and priority for the client. A socket_buffer will be used to contain the array of client sockets, along with pointers required for the circular buffer implementation and the pthread lock/condition variables. The log_buffer will contain an array of character arrays to contain any log output for another thread to work on.

The first step of the program functioning is to accept the command line arguments and store the values in memory. Then the circular array buffers, thread condition/lock, and network variables will be

initialized/declared. The default/specified dictionary will also be loaded into memory using a load_dictionary() function. Afterward, the number of worker threads specified by the command line will be generated into an array. The program will bind to the localhost address 127.0.0.1. This will allow communication with the clients and these connections will be stored in the circular array buffer along with a priority number (if FIFO priority, the priority variables will be in ascending order).The program will listen for new client connections and put them into the connection circular buffer.

The connection circular socket_buffer will be used to store client connection descriptors along with their priority as structs. If the socket_buffer is full the put_socket() function will use the pthread condition variable to wait until the get_socket() function removes a client_socket from the buffer and vice versa. Whenever these two functions are trying to access the socket_buffer, they will be pthread locked until completed to avoid any unwanted behavior. The log_buffer will be implemented in the same way except will store an array of strings to be printed to a file.

The worker threads will be using a function called worker(). Each thread running this function will continue to run until the program is completed and can wait until a client_socket is available due to using pthread condition variables within the get_socket() function. Once a client_socket is available to the worker thread it will be used to read the word that was sent to be spell-checked by the client. At this point, the time of this request is saved for the log. This word is input into a function that compares the word to the dictionary (check_dictionary(word)) and will return 1 if the word is found or 0 if not. If the word is found ": OK" is concatenated to the end or ": MISSPELLED" if it was not. Immediately after the concatenated word is sent back to the client. The completed time is saved for the log and a string of the log results is put into the log buffer (REQUEST_TIME_SECONDS, REQUEST_TIME_MICRO_SECONDS, COMPLETED_TIME_SECONDS, COMPLETED_TIME_MICRO_SECONDS, SPELLING_RESULT, PRIORITY). The word and result variables are then set to null and 0 to be ready for the next word that arrives.

The logger() function will be used by the single thread to write the log file. As log data is added into the log_buffer, the log thread will write this to the file until the program terminates.

Testing, Results, and Debugging

This program was created in multiple different parts and then incrementally combined to avoid excessive debugging. The first part of the program that was created was the circular buffer using only integers to test its functionality. This was done by using a for loop to populate the buffer and then also to remove the elements from the buffer.

The next part of the process was implementing all pthread variables and functions needed for the worker threads and logger thread to work correctly with the buffers. First locks and condition variables were added into the put and get functions of the log and connection buffers. Afterward, an array was populated with several threads. During the first run a loop was used to fill the buffer while the worker threads worked on the data and removed them, no issues came up. To test the upper wait bound of the main thread filling the buffer a thread sleep(1) was added into the worker thread function to slow how fast they consumed the data. Then the opposite was done, where the main thread producer loop had a thread sleep(1) added. This also worked well showing the pthread conditional variables were working correctly. After testing all the above we could use the same logic to implement both the socket_buffer and log_buffer and know they will function correctly and in the correct priority.

The next function tested was reading the dictionary from a default file or a path specified through the command line and the spell checking of a word to the dictionary in memory. Reading the file and storing it was fairly simple and then using a while loop to compare the word specified by the client would return a result if found or not found.

The worker thread function was then changed to implement these functions above by first making sure the client_sockets would be removed from the circular buffer and words would be read by the server and printed to the console and back to the client. Once this was successful the check_dictionary() function would be used to compare the words and return an OK result or MISSPELLED result to the client. A log string would also be added to the log buffer for the log thread to simply write to a file as the worker threads produced the data.

One last functionality that was tested was proper parsing of the command line arguments. This was done using print statements to make sure they were being checked for errors. Another function was created called is_numeric that will loop through each character and check for isdigit(). If not, a digit it returns 0 right away. This is used to be able to differentiate between the dictionary file path argument and the port number. This allows the dictionary and port number arguments to be used interchangeably. If any command-line argument does not match a prompt is printed and the program exits.

Results of Testing and Priority of Clients

The results of the testing were done using the custom client that was created, netcat using different ports, along with running the server on a cloud and local computer. All these types of testing reproduced the same results.

The clients that entered the socket_buffer were affected if the max buffer size was less than the number of connections queued. If any connections were waiting, they would be waiting until enough clients were consumed from the socket_buffer. This meant that if a client was waiting and a word was sent for spell checking there would also be no result until it was able to be consumed by a worker thread. Once it was able to be consumed it would then read all words and produce results along with printing to the log file.

If the number of threads was reduced below how many clients have connected a similar issue like the previous one would occur. When the socket_buffer is too small the main thread would be forced to wait and this would fill the backlog of the network socket bind function. If there were not enough threads in comparison to the number of clients the buffer would fill too quickly and would still cause a backlog of clients waiting to connect.

The priority of the connections only affects the output by there being enough threads to accept the client sockets. If there are enough threads the priority of the output is based on which client sends a word to the server first. If there is a backlog due to the queue not being big enough then the priority would make an impact by only allowing spell checking to occur with the clients that are assigned to a worker thread. If they are not high enough priority they will wait and still accept words to stdin but nothing will be processed.

Discussion and Analysis

Many important concepts should be kept in mind while using concurrency and using socket programming. It is very important to use multiple threads when working with multiple clients to be efficient or they will sit and wait until the first one's job is done. When using concurrency with sockets, it makes it very useful to process data from multiple sources. Using different buffer and max thread counts impacts performance differently. It may be important to find a balance when dealing with limited system resources. Such as allocating fewer threads and buffer size when system resources are limited or using more buffer size and threads if performance is important.

It was very easy to see the key differences between the producers (main) thread and the consumers (worker) threads when they were out of balance. During the testing and debugging phase I used thread sleep to either put the producer or consumer threads to sleep to test the wait conditions when the buffer was full or empty. The dramatically showed if the producer is slower than the consumer the buffer will be empty most of the time and vice versa.

When analyzing the priority, it was also interesting to notice that it only made a difference on what clients would get results back from the server. The client was able to initially connect but would then fill the buffer and any other ones would be in the actual socket backlog built into the socket programming library.