CSE 344 FINAL PROJECT REPORT

EMIRE KORKMAZ

July 29, 2020

In this project, we need to develop two programs which are a threaded server and a client. The server loads a graph from a text file, and uses a dynamic pool of threads to handle incoming connections. The client connects to the server and request a path between two arbitrary nodes from the server program and the server provides this to the client.

1 SERVER SIDE

In this side, firstly I needed to make it a deamon program which ensures that there will be only one instance of this program. In order to make this, I created a file in /tmp directory and only removed this file at the end of the program execution. In every execution of this program, I check if this file exists. If it does, then I exit the program. Otherwise, this means the only program working is the one we just have run. Secondly, I read the file in order to find the number of vertices. With this number, I allocated some memory that Ill be needing later. These are a data structure hold the graph, the cache data structure, an array for visited vertices (for BFS), a data structure for passing to the threads, mutexes and condition variables that I use for synchronization.

I read the given file into the data structure (vertices). After, I created a socket with the given arguments from the user and waited for the clients. If a client connects, it sends its request in a message and main thread reads this message, extract the tokens and assigns a thread for this client. In assignment, main thread looks for available threads. Each thread has a field named as busy and when working they set this field to 1. When they finished their job, they set it back to 0. With this field, the main thread can tell which threads in the pool are working or waiting. If all threads are busy, then it waits with a condition variable named mainThreadCond. When a thread finishes it job, it checks if main thread is waiting or not, if it is, then the thread wakes the main thread with pthread_cond_signal. And main thread continues to wait for client connections.

All mutexes are used to solve the synchronization issues. Cache mutex is used to prevent multiple writer threads accessing the cache at the same time.

CapacitySignalMutex is used to prevent threads work and change the ratio while capacity threads work. Main thread mutex is used to prevent accepting all connection while there is no available thread. In reader writer prioritization, I had to use mutexes in order make the other mutexes wait. To pause the threads, I needed to use condition variables. With this, threads would be waiting until the main thread wants them to wake up.

At the beginning, a thread pool is created with the given number of threads from user. And waits for an assignment from the main thread. In waiting, another condition variable is used to be sure that there would be no busy waiting. If main thread decides to give an assignment to a thread, firstly it prepares the needed arguments (source, destination, socket etc.) and it wakes the thread with pthread_cond_signal.

After getting assigned by the main thread, the thread calls BFS function with the given parameters from the client. There are three possibilities. First, there is no path between source and destination. Second, there is a path between source and destination but has not been written into the cache data structure. Third, the path had already been saved by another thread into the cache. The thread sends the message back to the client. This message could be the path or a message indicates that there is no path between source and destination.

In BFS function, firstly cache data structure is checked in order to not to calculate a path again. This gives the program efficiency and speed. If the path is not found in the cache, then it will be searched in the vertices using breadth-first-search method.

Cache data structure, contains 4 fields. These are source, destination, path and the number of vertices in the path. Whenever a path is searched in this, the program looks for source and the destination. If there is a match, the index is returned. Since the cache is checked before writing, there would be no duplicated entries. In order to make an operation safe, a mutex is used before writing to prevent unwanted situations.

Another mutex is used to do bonus part. This mutex is called priority mutex. If there is any priority, lets say writers have priority, if a thread wants to read the cache firstly it checks if there is any priority for the writers, if it is, it checks if any writers work. If there are writer threads, reader thread wait using priorityCond. After the last writer thread has done it job, it checks writers if there are no more writers, then it calls unlocking function which lets the waiting threads know that there are no more writer threads. This operation is done in the reader priority as well. If there are no priority, none of these happens. Threads work normally.

When I tested this part, they were no waiting threads and server freezed. I looked at the code and realized that a wrong mutex was used in some parts. I fixed it and it worked. With the correct mutex, I made the threads that have less priority wait and also synchronized all the threads.

There is another thread for checking the servers capacity, it loops endlessly and checks for working threads. If the ratio of the working threads over the number of threads is more than 75. More threads are created. All numbers are updated. Before extending the threads, it checks if extension will make the number of threads more than the maximum number of threads which is given from the user. If this happens, thread exits. Since there are the maximum number of threads available, the capacity threads job has done. Because of this, capacity thread breaks in the loop and exits.

Additionally, another condition variable is used. This is called capacitySignalCond. What it does is making the threads wait when capacity thread extending the threads and all additional data structures. These data structure were mentioned in the reading of the file side.

Every operation is logged by the threads. The log file is given by the user in the program arguments. Another mutex is used in the writing to the log file. Mutex is locked before the preparation of the log message. After writing into the file, mutex is unlocked. A timestamp is included at the beginning of every line. In each logging, this process happens.

SIGINT is handled in this program. All endless loops are conditioned with this condition; if SIGINT is caught or not. If it is, all threads are informed using a variable. Since all loops use this condition, after they finish their jobs they go back to the loop condition, and exit.

2 CLIENT SIDE

It takes the servers address, port number, source and destination. It simply request a path from node i1 to i2 and wait for a response, while printing its output on the terminal.

It connects to the server and sends what it requires. Waits for the servers response. After getting the response, prints out the path on the terminal. It also calculates how long did it take to get a response from the server. It also prints out this information on the terminal as well.

3 NOTES

I implemented bonus part, too. You can test for both cases.