Distributed Systems

CAB403 FINAL ASSIGNMENT

GRANT DARE N9476512

https://github.com/ThePapaG/CAB403MS

Statement of Completeness

In this assignment all tasks have been attempted with implementation. Task 1 has been implemented in its entirety and has been tested to work. The implementation of task 2 is too implemented using semaphores to synchronise the client connection requests to the server main connection thread. Task 3 is implemented to use create a pool of MAX_CLIENTS threads on server initialisation for client handling.

Student Contribution Statement

At the start of this assignment it was a collaborative effort between myself, Grant Dare, and Callum Scott. In the git log you can see that on the 29th of October I committed a commit titled "Restructured the Entire Assignment". In this commit I had deleted all previous contributions made by Callum and started from scratch. All work in the final solution of this assignment is my own where none remains from my previous team member.

Compilation and Run Instructions

The readme file contains this information if able to view, however it will be repeated here. The assignment follows a common file structure where the root contains a bin, src, and include directory where the executables, c source code, and c headers are located respectively. For ease of compilation a simple make file has been created in the root and so to compile the assignment, from root, simply enter the command;

make

This will run the following command line compilation command on both the server and the client;

gcc -g -std=c99 -D_POSIX_SOURCE -pthread SOURCE CODE -I./include -o bin/TARGET

Then to run the respective programs direct to the bin directory and use the command;

./server.exe PORT

./client.exe HOST PORT

Where the server port is the port you'd like the server to open the connection on (this will default to 12345 if not selected). Where the client HOST is the IP address of the server running (localhost if on the same machine), and PORT is the port that the server has open for connection (if default server port this is 12345).

Important Design Choices

- To prevent cheating, as requested by the customer, All interaction with the game is server side and the data sent to the client is char representations of the game state formulated by the server. This means that the need for byte wise network transfer is unneeded since chars are only single byte structures and numbers are double byte (they contain a sign byte).
- To improve client performance the send and receive commands are running independently of each other on different threads.
- Semaphores are used server side over pthread signals for synchronisation with one pthread keeping track of clients in the que and another semaphore keeping track of clients connected (so that no more than MAX_CLIENTS can connect).

Game Data Structures

The game implementation itself uses two structures. GameState which holds the play field as a 2D array of the next structure, the number of mines remaining, and the time the game state was initialised. The second structure is the Tile structure which contains the fields; adjacent_mines to hold the number data for the mines around it, revealed is a Boolean to determine if the tile can be drawn, is_flag is a Boolean to assist in the drawing of the playfield, is_mine is a Boolean to determine game over and win conditions.

Leader Board Structure

As each client can have multiple entries and the client entries must be sorted according to their own values, 4 structures were used here (including the client structure as it holds relevant fields). These structures are; the client for hold a linked list of their own values (this can be adapted to view only a single players successes in future), a client entry to hold the data about the clients victory, a leaderboard entry which acts as a linked list and is created when the leader board sort function is called (this is where the client entries are sorted and added to the leaderboard linked list), and the leader board.

Critical-Section Solution

There are multiple parts to the critical section solution that I have chosen to implement. There are 3 mutex locks currently implemented; one for client queue, one for clients handled, and one for the leader board. The client queue mutex ensures that only one thread can access and add a client to the que at a time so that the clients don't get lost. The clients handled mutex ensures that once a thread picks up a client, only that thread can checkout that client from the que. Finally the leader board mutex ensures that only one client (thread) can be accessing the data structures used to create the leader board at once.

Thread-Pool Implementation

The thread-pool initialises with a value of MAX_CLIENTS since this is the maximum amount of clients we can connect at a time. The threads setup their initialisation variables and then wait for the semaphore to signal there is a new client ready to be handled. This then links back to the previous section, Critical-section solution, in that what ever thread acquires the mutex lock and checks out the client first will be the thread handling the client, all others will continue to wait for a new client. Upon server shutdown, all threads will join and gracefully shut down.