CAB403 Alexander Brimblecombe

(Individual)

n10009833

Major Assignment

Statement of completeness / Report

Statement of completeness

All of the features of the assignment have been implemented, except for network byte order used for transmitting multi-byte data types. This was due to the fact that the program was already largely implemented and I did not have time to make appropriate alterations as the transmitted data types vary in size by a large margin.

The signal handling for closing down the server is also something which can be improved in my implementation. I opted for cancelling the threads rather than joining them, as threads are blocked when waiting on a client with the recv command. All of the resources used by the threads are either globally accessible or defined on the stack so there should be no memory leaks in this implementation. An improvement to this situation would be creating a socket receiivng thread on the client's side dedicated to filling a buffer of communications from the server. This buffer could then be tested for a quit command which could in turn allow for the client to also send a quit command.

TEAM AND CONTRIBUTION: This assignment was completed individually.

Data Structures used for Game

For the implementation of the game, I opted for making a dedicated header file to declare the structures for the game types. These types are shared by the client and the server which means that a full Tile struct cant be sent over a socket, and interpreted on the client's end.

```
typedef struct {
        u_int8_t adjacent_mines;
        u_int8_t is_revealed;
        u_int8_t is_bomb;
        u_int8_t flagged;
        u_int8_t x;
        u_int8_t y;
}Tile;

/* Struct for game type */

typedef struct {
        bool game_over;
        Tile tiles[NUM_TILES_X][NUM_TILES_Y];
        int remaining_mines;
        int start_time;
}Game;
```

A variation from the assignment specification is the fact the the x and y position of each tile is stored in the struct. This is to allow a single Tile to contain all the information that the client needs to know when receiving from the server. (This wasn't necesarry when one tile was sent, however in

my implementation it is necessary when tiles are sent recursively as it allows the client to successively know the position of each surrounding tile when there are 0 surrounding mines). The types of data stored within the Tile struct have all been converted into unsigned integers. This is to reduce the size of the Tile packets sent to the client from the server. The variable 'flagged' has also been added for when the client is printing a representation of the playfield.

The struct for the game is used to play game instances on the server. The mines are placed randomly among the two dimensional array of tiles, with the remaining mines of each tile reflecting how many of its neighbors are mines. A client also keeps a copy of the game state, however none of the generated mines are communciated to the client, i.e. the game field is initialised to empty for the client. Tiles are then sent to the client, as he/she chooses to reveal them.

The game_over value is to control the loop of the function used to play a game instance on the server as well as the client. The remaining_mines values is used to store how many of the tiles containing mines are currently unflagged. The 'start_time' variable is used to extrapolate the time elapsed of the game when game_over becomes true.

Data Structures used for The Leaderboard

```
typedef struct user_profile{
        char username[40];
        int games_played;
        int games_won;
        struct user_profile * next;
}profile;

typedef struct leader_node{
        profile * player;
        time_t time_elapsed;
        struct leader_node * next;
}node;
```

The struct for profile is to store information about each account that signs in. This to avoid each node in the leaderboard being updated for a specific player every time they play a game, rather, their profile is updated which is contained within each node (game won). Everytime a game is played, their profile is updated. If no profile containing the username of the connected client exits, a new profile is added which will be updated upon every new game played by that user.

The struct for node is to store individual entries relating to a won game by a specific user. Pointing to the profile of the player who won the game to get their name and details, as well as storing the time elapsed for that specific game.

Data Structures used for Handling connections

```
struct client_queue{
        int fd;
        struct client_queue *next;
};

struct connection_inst{
        int thread; // allocated thread
        int fd; // descriptor of socket associated with client.
        char username[40]; // username of client currently being serviced.
        char password[40]; // password of client currently being serviced.
        bool connect; // stores whether a client is currently connected to socket.
};
```

The client_queue struct is used for handling connection requests from clients. When a client attempts to connect to the server, if there are free threads their socket is added to this queue. The threads are then notified that there is a new client to be handled. A thread can then retrieve this socket from the queue and start its handling loop. This client_queue linked list is used instead of a bounded buffer for the producer and consumer approach of handling a thread pool, this meant that I didn't have to keep track of multiple array indexes to add and remove items in the queue.

The connection_inst struct is used to keep track of information about a client connected to a specific thread. A global array of 10 'connect_inst's (thread pool size) is initialised at the beginning with each thread relating to one item. This is to allow thread information to be globally accessible by other functions. The boolean 'connect' is to keep track of whether a client is actually connected to a specific thread. The 'fd' stores the socket of the connected client, username and password are used to authenticate a client connected to a thread and 'thread' keeps track of which index each connection_inst occupies in the array.

Critical Section for Leadboard

The critical section for the leaderboard is using reader and writer locks from the reader_writer solution from topic 5 reader_writer.c. The read lock function acquires the reader mutex and the reader count mutex and increases the reader count by one, then it releases both locks. If the reader count is equal to 1, the writer mutex is locked. The read unlock function acquires the reader count mutex, decreases the reader count and then unlocks the reader count mutex. Within this section, if the reader count is equal to 0, the writer mutex is unlocked.

The writer lock function acquires both the reader and writer mutexes, preventing the reader mutex from being acquired by any other reader. The write unlock function releases both mutexes.

This behaviour allows multiple threads to pass the reader lock concurrently, with the writer lock only being acquired by threads when there are no other threads with the reader lock. When there is a writer thread acquiring the writer lock, no other threads are able to pass the reader lock.

The writer lock is used in the function *int update_leaderboard(char * username, int won, int elapsed_time)*. This functions updates the list of user profiles, inserts a new leaderboard node relating to a game that was just played, and sorts the leaderboard to account for updated profiles.

The reader lock is used in the function *int send_leaderboard(int sock_fd)* which sends leaderboard data to the clients requesting the leaderboard (readers).

Critical Section for client queue

Critical Section for Random generator

Thread Pool Synchronisation

Compile Instructions

1. Navigate to folder with make file - run ‘make’ to compile server_program and client_program

2. Run 'server_program' with either a argument for port number - or no argument (default port: 12345) ./server_program 12345

3. run instances of 'client_program' inside with IP address and port number as arguments.
   i.e. ./client_program 127.0.0.1 12345

NOTE: if the server program is run from outside the directory of the executable file the authentication.txt file will not be found