CAB403 ASSIGNMNET

Server/Client Minesweeper game

Jonathan Salazar – n9969071
Max Kingsman – n9432205

Contents

Statement of Completeness

	Tasks attempted	Deviations from specification	Problems in solution
Task 1	Attempted all tasks	• Server sends client a confirmation message each time a tile has been revealed or flagged	• When server exits upon receiving SIGNAL (ctrl + c), it does not cancel the threads • Leaderboard data is not sent to client to display
Task 2	Attempted all tasks	• The leader board is unable to be updated while a thread is viewing it.	• Remaining mines not synchronized. Only works correctly when using one client
Task 3	Attempted thread pool creation	• Thread is created to process connection as soon as the server accepts socket	• Threads are created after a socket is accepted

Statement of Contribution

Student name + number	Percentage Contribution (%)
Jonathan Salazar, n9969071	93
Max Kingsman, n9432205	7

Implementation of playing field

The data structure used to represent the playing field was a 2-D array. The columns and rows act as the grid and has a maximum size of 9x9. This data structure was used because it has a simple implementation process and therefore errors are less likely to occur. Two for-loops are used to create the grid and additional functions are utilised to set the state of the tiles.

Two structures GameState and Tile, are used to store the 2-D array playfield and the tile state respectively. After a user input has been validated, GameState will access the Tile structure to change the state of the chosen tile.

Another 2D array was created which will store the grid that will be sent back to the client. A set of for-loop and if-statements are then used to update the state of the playfield which is then sent to the client. When the client receives the grid, it uses two for-loops to store the data into a 2D array, which is then printed to the screen. Network byte order was used to send and receive data between the server and client to ensure the data is in the correct order to be interpreted.

Implementation of leader board

The data structure for the leader board was linked lists, linked lists were the best option due to the efficient memory utilization. The linked list made it easy for us to develop the leader board to accept new values when the game is played.

First, we create a new player object to assign values to the current player. Then we have 4 methods to update the current users record, update the game record, display the leader board and finally to erase all the values from the leader board. We also created a second struct to hold the wins.

On the client side we created a struct to hold the values, we order the list to show the fastest at the top and slowest at the bottom.

If there are no entries the leader board displays just that as shown below

Figure 1:Leaderboard with no entries

When there are entries in the leader board it looks like the below

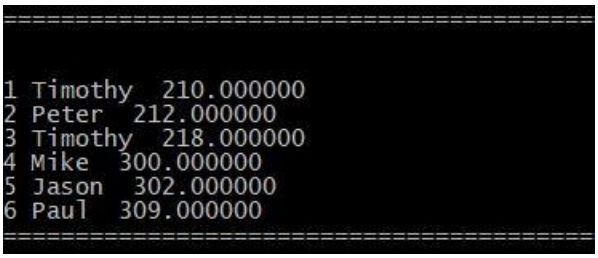

Figure 2: Leaderboard with entries

Handling critical-section problem

For multithreading programming and process synchronisation, to allow for multiple (10) users to play the game, the leader board needed to be mutex locked if a player was viewing it. We used mutexes when creating the game time for the leader board to lock the time in. A mutex was also used when creating the game record, to keep the values consistent throughout the leader board. We also didn't want to cause any possible clashes with this data so using the mutexes was the best option to achieve the results we wanted without causing race conditions and potentially corrupting the data. To synchronize the rand() function, mutex was used to lock the variable when a client was using it, then it was unlocked after the client was done with it.

Creating and managing thread pool

The thread pool works by using a producer consumer pattern. The main thread acts as a producer by accepting client sockets and placing them in a queue using the function 'listen()' function. Then a for-loop is used to extract the pending connections from the queue and assign a thread to serve the client (the consumer).

The thread pool was created by first initializing an array of type 'pthread_t', to store the threads id. The array had a maximum size of 10, which is the maximum number of clients can server at any one time. To create the threads and assign them functions to execute, a for-loop was implemented (Figure 3).

```
// if-statement to check number of clients less than or equal to max number of clients
if(number_of_connected_client >=MAX_CLIENTS){
    pthread_mutex_lock(&lock); // Lock mutex
    pthread_cond_wait(&cv, &lock); // Wait for condition signal
    pthread_mutex_unlock(&lock); // Unlock mutex
}

// for-loop to create thread pool to accept clients from listen queue
for (int i = 0; i < MAX_CLIENTS; i++){

    sin_size = sizeof(struct sockaddr_in); // Variable for accept() function
    // if-statement to check if accept() function executed correctly, prints error message if error occured
    if ((new_fd[i] = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size)) == RETURNED_ERROR) {
        perror("Main: Accepting client sockfd");
        continue;
    }

    client_position[i] = new_fd[i]; // Assign client socket descriptor to a position in 'client_position' array
    printf("server: got connection from %s\n", inet_ntoa(their_addr.sin_addr)); // Prints ip address of new client

    // if-statment to create thread to server client. If pthread_create() does not execute correctly, error message is printed,
    // else increase 'number_of_connect_clients'
    if( pthread_create(&client_thread[i], &attr, Authentication, (void*) (intptr_t)new_fd[i]) < 0){
        perror("Main: Creating thread pool");
    }
    else{
        pthread_mutex_lock(&lock); // Lock mutex
        number_of_connected_client++; // Increase number of connect clients
        pthread_mutex_unlock(&lock); // Unlock mutex
    }
}
```

Figure 3:Threadpool implementation

The for-loop is placed in 'main()', and runs for MAX_CLIENTS number of times, to serve 10 clients. The 'accept()' function is used to extract first connection from the queue and assign its socket descriptor value to a position in an array 'new_fd'. A thread is then created to process the connection with the client's socket descriptor as the argument.

An if-statement was implemented above the for-loop to ensure that no more than 10 threads can be serving a client at the same time. This is done by using 'pthread_cond_wait()' that blocks a condition variable to the calling thread. The thread can only be awakened when a client disconnects from the server and sends a signal to unblock the thread.

After a client disconnects, the thread will complete its task and can be reused to process another connection.

Instructions to compile and run program

To compile the server and client, the following command must be entered in the terminal:

Server – 'gcc -o Server Server.c -lpthread'

Client – 'gcc -o Client Client.c -lpthread'

The '-lpthread' tag at the end ensures the pthread library is linked to the program. To execute the programs, the following the commands need to be entered:

Server – './Server (Port Number)'

Client – './Client localhost (Port Number)'

The second argument for the client 'localhost' is the name of the host that the server is running on, this allows the client to be connected to the same host. The last argument for both programs 'Port Number' is the port number that the server will listen for connections on. This can be user defined, otherwise a default number of '12345' will be used. The server must be executed first then the client.

After the programs are executed, the server will not take any user input unless it's the SIGINT (ctrl + c) signal to exit the program. The client will accept user input and will display a welcome screen and login prompt for the user as shown in Figure 4.

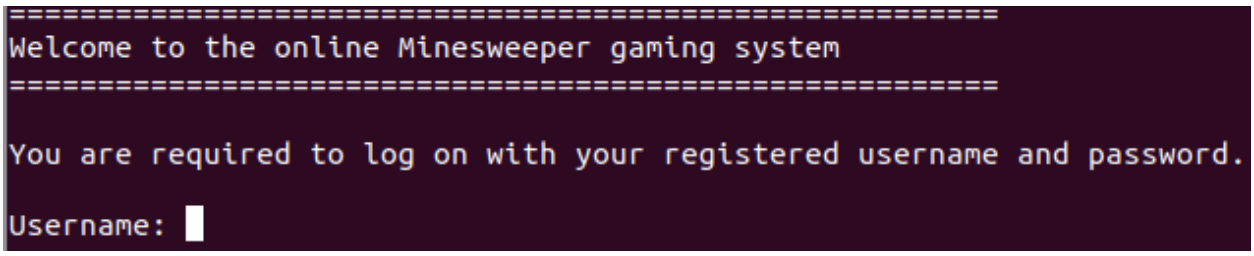

Figure 4: Login Prompt

The login prompt will require the user to enter a username and password that matches one of the credentials on the servers Authentication.txt file. If the user enters an incorrect login, the client will disconnect from the server and end the program. If a valid login is entered, the user will be brought to the Main Menu of the Minesweeper game as seen in Figure 5. The menu contains 3 selections which allow the user to play a round of the Minesweeper game, look at the leaderboard or quit the program. To select an option, enter 1, 2 or 3 in the terminal and press the ENTER key.

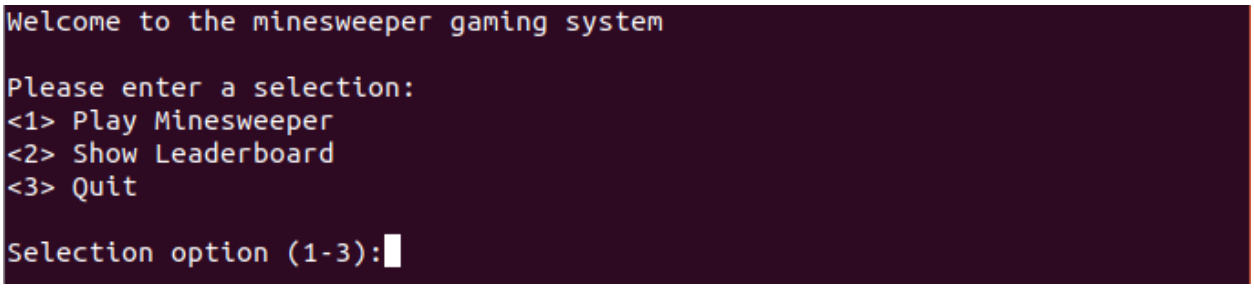

Figure 5: Main Menu

If a user chooses to play a game of Minesweeper, a blank grid will be displayed, along with the total number of remaining mines and a set of instructions to play the game (Figure 6).

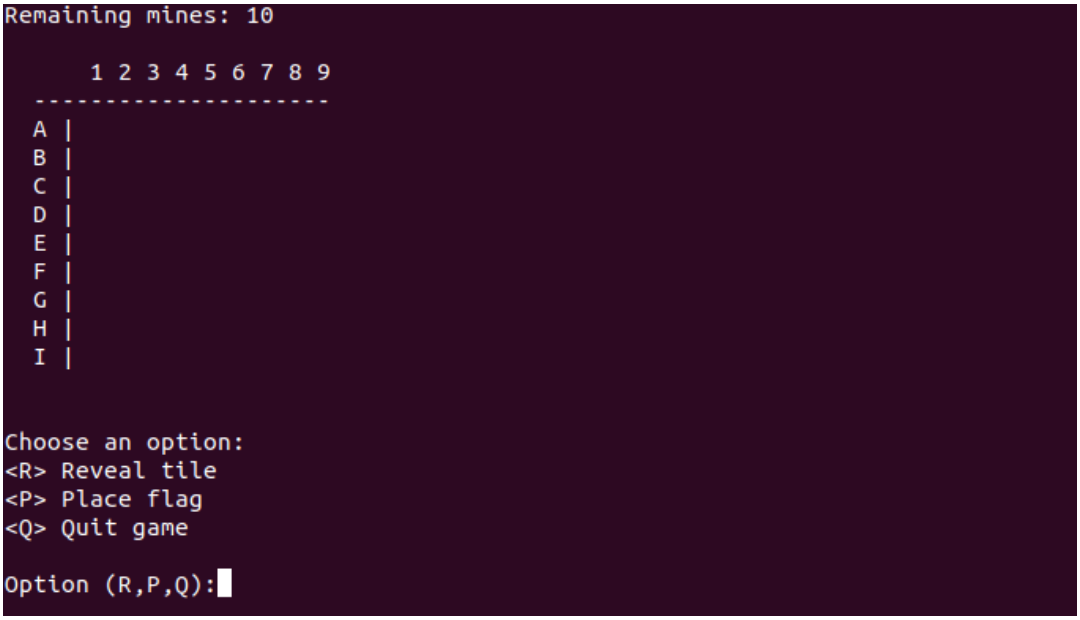

Figure 6: Blank Grid

The user is given three options to, reveal a tile, place a flag or quit the game. If the user chooses to quit the game, the client will be brought to the Main Menu. If the user opts to reveal a tile or place a flag, the client will prompt the user to enter a set of coordinates. If an invalid input is entered, a message will be printed to the screen to alert the user. On successful input of coordinates, the updated grid and a confirmation message will be displayed. This is seen in Figures 7 & 8 when a user has chosen to reveal a tile.

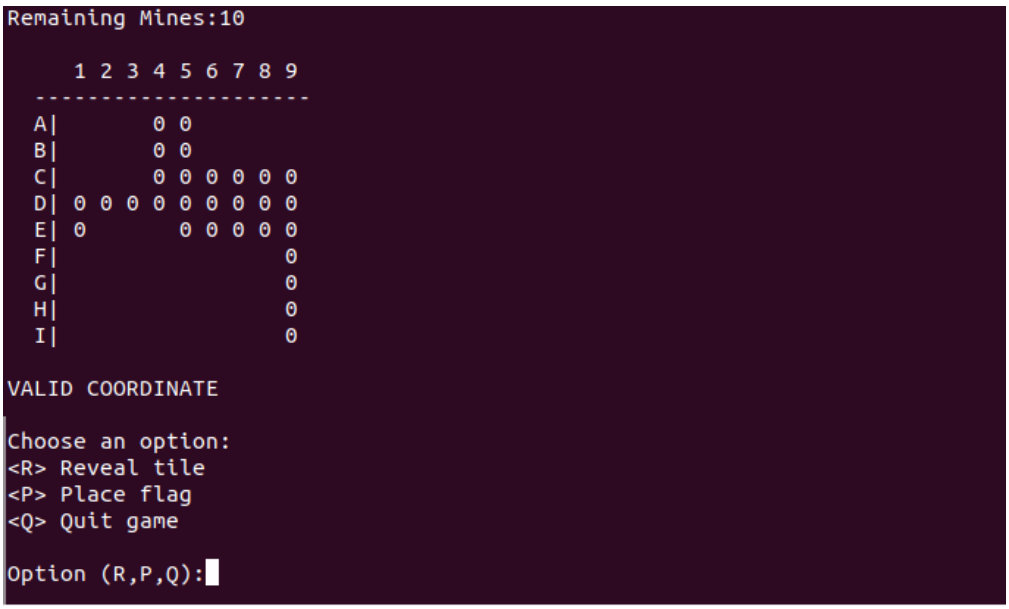

Figure 7: Tile with zero adjacent mines revealed

```
Remaining Mines:10

    1 2 3 4 5 6 7 8 9
  --------------------
  A|   2   0 0
  B|       0 0
  C|       0 0 0 0 0 0
  D| 0 0 0 0 0 0 0 0 0
  E| 0       0 0 0 0 0
  F|                 0
  G|                 0
  H|                 0
  I|                 0

VALID COORDINATE

Choose an option:
<R> Reveal tile
<P> Place flag
<Q> Quit game

Option (R,P,Q):
```

Figure 8: Tile with 2 adjacent mines revealed

On successful execution of placing a flag, the remaining mines counter is decreased by 1 (Figure 9).

```
Remaining Mines:9

    1 2 3 4 5 6 7 8 9
  --------------------
  A| + 2   0 0
  B|       0 0
  C|       0 0 0 0 0 0
  D| 0 0 0 0 0 0 0 0 0
  E| 0       0 0 0 0 0
  F|                 0
  G|                 0
  H|                 0
  I|                 0

NICE THERES A MINE THERE!

Choose an option:
<R> Reveal tile
<P> Place flag
<Q> Quit game

Option (R,P,Q):
```

Figure 9: Successful placement of flag

If a tile is revealed and happens to be a mine, the full playfield with the locations of the mines is displayed and the user is notified that they have lost the game (Figure 10). The client will wait for a user input before returning to the Main Menu.

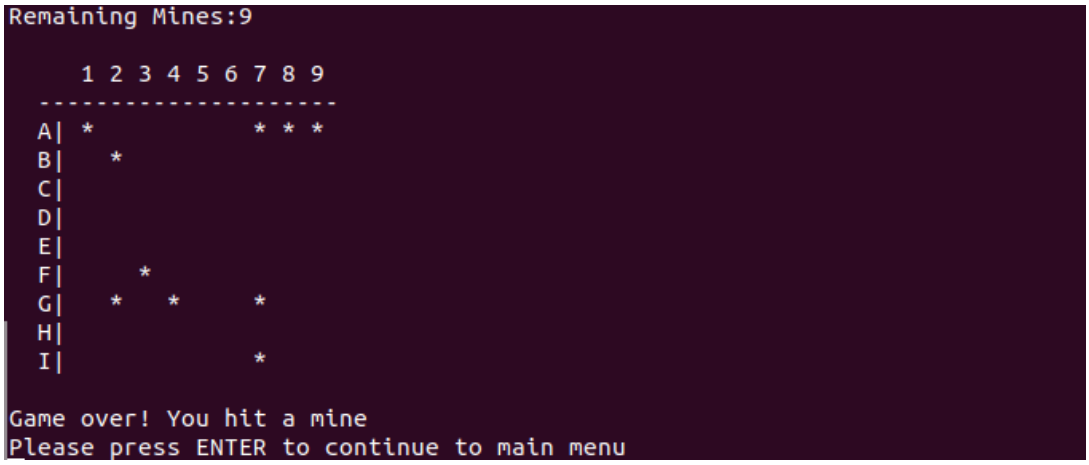

Figure 10: Revealing a mine

If the user ends places flags on all the mines a message is displayed to inform the user that the game is won, as well as the total time it took to win the game. The client then waits for an input before proceeding to the Main Menu (Figure 11).

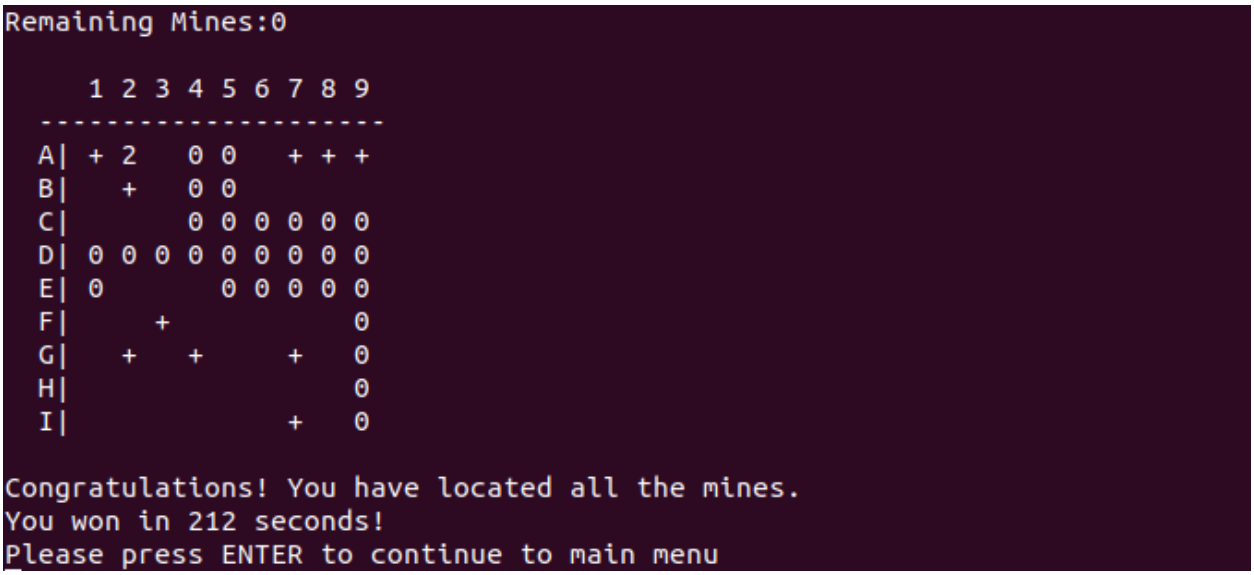

Figure 11: Winning game

To exit mid-game press ‘q’ as a game selection. (Figure 12)

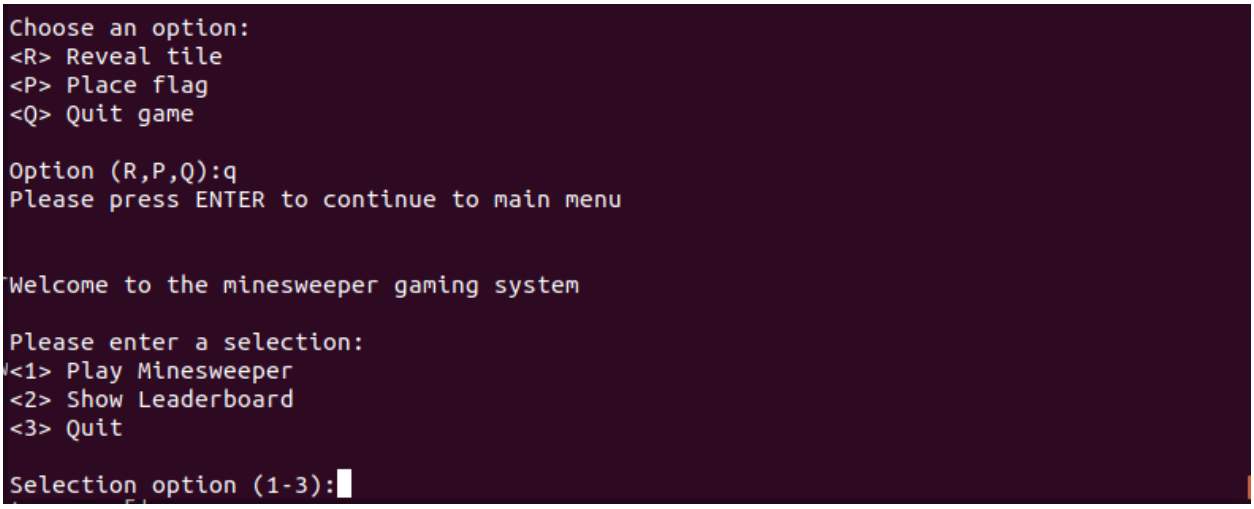

Figure 12:Exiting mid-game

To quit the game press ‘3’ at the Main Menu (Figure 13).

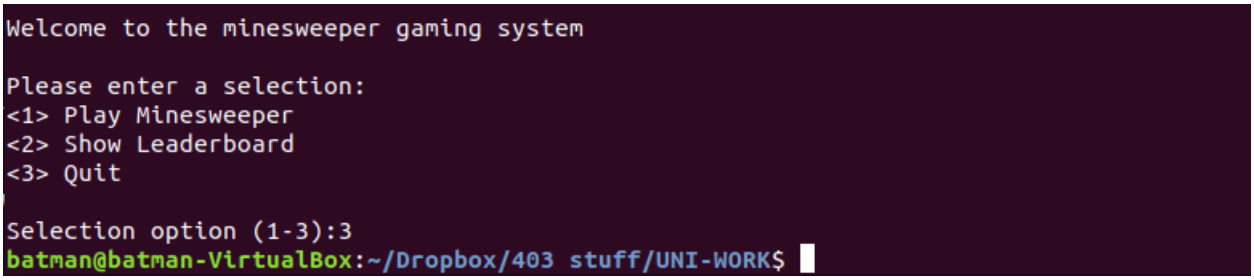

Figure 13: Quitting the game