CAB403 Minesweeper

# Statement of Completeness

|  |  |
| --- | --- |
| Tasks | Comment |
| Place 10 mines randomly | Done |
| Player unaware of initial game state | Done |
| Player able to reveal a tile | Done |
| Player able to place a flag on a tile | Done |
| Reveal tile, no mine, return number of adjacent mines | Done |
| Reveal tile, no mine, return recursive number of adjacent mines | Done |
| Reveal tile, mine, end game | Done |
| Place all flags, win condition | Done |
| Server/Client system | Done |
| Server determines game state | Done |
| Clang using BSD sockets | Done |
| Authentication handling | Done |
| Fixed random seed | Done |
| Proper handling of SIGINT | Done |
| Track time taken | Done |
| Leaderboard | Done |
| Leaderboard in descending order | Done |
| Leaderboard in descending order, then by number of games won | Done |
| Multiple connections | Done |
| Critical section problem | TO DO |
| Solve the access to rand() issue | TO DO |
| Thread pool | TO DO |

# Team Information

Brandon Janson, N9494006

Mathew Field, N9458051

# Statement of Contribution

All members of the group contributed equally.

# Data Structures

## Playfield

The state of the game was stored in a *GameState* struct. This contained information about the playfield itself (implemented as a multi-dimensional array of *Tile* structs), and the number of remaining mines to discover.

These *Tiles* contained three fields, the number of adjacent mines, *revealed* (whether it had been revealed by the user), and *isMine* (clearly, whether this particular tile was where a mine was located).

Each time a new game was started on the server, it would initialise this array of *Tiles* (with the default state of no adjacent mines, not revealed and not a mine), and then would randomly pick *x* and *y* positions to set as mines. Then, it would go through every *tile* and determine the number of adjacent mines for each *tile*. Then, when the client queried a particular *tile*, it would not need to be constantly recalculating, and instead can focus on recursively traversing the playing field to respond with what was required.

The client and server (thread servicing the client) both had instances of their own ‘version’ of *GameState*, where the server had the official one, and the client was only able to query the server and update accordingly.

## Leader board

The leader board was implemented as a struct, containing a combination of a linked list of *Score* structs and an array of *Players*.

The *Score* struct had an associated string for the name, an integer for the score (time taken), and a pointer to the next *Score* in the list. Upon adding a new Score to the leader board, the linked list was traversed until the appropriate location was found, and then inserted into this correct location.

The *Player* struct maintained a name, a password, a number of games played, and a number of games won. This array of *Players* is initialised as soon as the server is started, as it reads in the username/password combination from *Authentication.txt*, and creates these *Players* as needed. When a user logs in to the gaming system, it searches for a *Player* with the given name and password, and then the (thread servicing the) server, maintains this *Player* struct until they close the connection.

# Critical Section Problem

The concurrent connections feature was implemented to an extent. Each new connection that was established was given a new socket number to communicate over, and this was sent to a thread to handle that connection separately, whilst the main process returned to listening for new connections. This allows any number of users to connect simultaneously. The specifications required a maximum of 10 concurrent connections, and this feature was not implemented due to a lack of time, and an incomplete solution on how to free up threads as they were finished. As well, the critical section problem was not ‘solved’, and all access to the leaderboard is not thread-safe. As such, there will be problems that occur when attempting to read at the same time as another connection is attempting to write. The access to *rand*() has also not been handled with a mutex and so is not thread-safe.

# Thread Pool

# Instructions on how to compile and run

As a part of the submission, there is a Makefile which should just run, as is. If, however, it does not compile, the simple command given below should compile the server:

*gcc -pthread -o server server.c*

The client can be compiled similarly by:

*gcc -o client client.c*

## Running the client

*./client {server\_address} {port\_number}*

You are required to specify the address and port number of the server you wish to connect to. If you are hosting locally, localhost is fine.

## Running the server

*./server {port\_number}*

You are not required to specify the port number that you want to listen on, but if you do not the default port number of 12345 will be used.

Other than that, all other interaction with the program should be in accordance with the specifications.