Assignment: Process Management and Distributed Computing

Due Date: **26th October 2018**
Weighting: **40%**
Group or Individual: **Individual/Group of Two**

1 Assignment Description

Task 1: Client-Server Computing

<u>You only need to attempt this task if you are aiming to achieve a mark up to 64% for this assignment.</u>

You have been commissioned to develop a client/server system for an online games provider to expand their current offerings to registered clients. The company wishes to offer to their current clients the game of Minesweeper.

Minesweeper is a single-player game, with the objective of placing flags on a grid of either empty tiles, or tiles containing "mines". The player does not know the location of these mines at the beginning of a round, yet must place the flags in the correct locations, which is the main challenge of the game. A tile can be revealed to discover how many mines are adjacent to it, as a hint to where the mines lie. However, if a tile containing a mine is revealed, the game is over.

The following outlines the basic premise of the game of Minesweeper:

1. 10 mines are randomly placed on a 9x9 grid, with the locations unknown to the player.
2. From the player's perspective, all the tiles start in the "unknown" state.
3. The player can either select a tile to "reveal" what is underneath or place a flag on the tile.
4. When a player reveals a tile, which does not contain a mine, if the tile is adjacent to a mine (meaning a mine is in one of the surrounding 8 tiles), the tile is filled in with the number of adjacent mines. Otherwise, the tile is replaced with a zero, and the process repeats recursively for each of its neighbours, stopping once a tile with a nonzero number of adjacent mines is encountered.
5. If a player reveals a tile which contains a mine, the game is over, and the player loses.
6. Once a player has placed flags on all the tiles which contain mines, the player wins. This is different to the classic Minesweeper game, where all non-mine tiles must be revealed, to simplify game logic.

This game will be simulated using a distributed system architecture. The server determines the placement of the mines and maintains the state of the game grid. The client should therefore receive input from the player, and relay this to the server. The server can then simulate the player's move and transmit the new state of the grid to the client. **Only the visible tiles should be sent to the client**.

The client and server will be implemented in the C programming language using BSD sockets on the Linux operating system. This is the same environment used during the weekly practicals. The programs (clients and server) are to run in a terminal reading input from the keyboard and writing output to the screen.

You will have acquired all the necessary knowledge and skills to complete the assignment by attending all the lectures and practicals, and completing the associated reading from the text book.

Server

The server will take only one command line parameter that indicates which port the server is to listen on. If no port number is supplied the default port of 12345 is to be used by the server. The following command will run the server program on port 12345.

```
./server 12345
```

The server is responsible for ensuring only registered clients of the system can play Minesweeper. A file named *Authentication.txt* contains the names and the passwords of all registered clients. This file should be located in the same directory as the server binary file. This file is *not* to be changed.

The company has requested that the state of the game be deterministic, with a **fixed random seed** of **42**. This can be done using the **srand()** function at the beginning of the program, as follows:

```c
#define RANDOM_NUMBER_SEED 42

// [...]

int main(int argc, char* argv[])
{
  // [...]
  srand(RANDOM_NUMBER_SEED);    // Seed the random number generator.
  // [...]
```

Figure 1: Seeding the random number generator

What the use of a fixed random seed means is that the first game played on the server after it has started will have the same mines in the same locations each time. The second game will have a different set of mines, but they will be the same as the second game played on a different invocation of the server, and so forth.

The server should not be able to take any input from the terminal once it is running. The only way that the server is to exit is upon receiving a SIGINT from the operating system (an interrupt signal). SIGINT is a type of signal found on POSIX systems and in the Linux environment this means ctrl+c has been pressed at the server terminal. You need to implement a signal handler

and ensure the server exits cleanly. This means the server needs to deal elegantly with any threads that have been created as well as any open sockets, dynamically allocated memory and/or open files. When the server receives a SIGINT, it is to immediately commence the shutdown procedure even if there are currently connected clients.

In the 9th edition of the textbook, Section 4.6.2 Signal Handling on page 183 has an introduction to signal handlers and Section 18.9.1 Synchronisation and Signals on page 818 has a detailed description of signals on Linux.

Client

The client will take two (2) command line parameters: hostname and port number. The following command will run the client program connecting to the server on port 12345.

```
./client server_IP_address 12345
```

The game of Minesweeper is only available to registered clients of the company's online gaming system. Once the client program starts the client must authenticate by entering a registered name and password which appear in the *Authentication.txt* file located on the server. After the ***server*** validates the client's credentials the client program can proceed to the main menu. If the client does not authenticate correctly the socket is to be immediately closed. (see Figure 2 & Figure 3)

```
============================================================
Welcome to the online Minesweeper gaming system
============================================================

You are required to log on with your registered user name and password.

Username: student
Password: CAB403
```

Figure 2: Client logon screen

```
============================================================
Welcome to the online Minesweeper gaming system
============================================================

You are required to log on with your registered user name and password.

Username: student
Password: CXB403

You entered either an incorrect username or password. Disconnecting.
```

Figure 3: Unsuccessful client logon

The client program should be a menu driven application. The client program will have three options: *Play Minesweeper, Show Leaderboard, and Exit.* (Figure 4)

```
Welcome to the Minesweeper gaming system.

Please enter a selection:
<1> Play Minesweeper
<2> Show Leaderboard
<3> Quit

Selection option (1-3):
```

Figure 4: Client main menu

If the client elects to play a game of Minesweeper, the ***server*** is to allocate storage for the state of the client's game, including the grid of 9x9 tiles. All tiles should start in the unrevealed state. One possible method for storing the game state could be with a C **struct**, covered earlier in the semester.

```c
#define NUM_TILES_X 9
#define NUM_TILES_Y 9
#define NUM_MINES 10

typedef struct
{
  int adjacent_mines;
  bool revealed;
  bool is_mine;
} Tile;

typedef struct GameState
{
  // ... additional fields ...
  Tile tiles[NUM_TILES_X][NUM_TILES_Y];
};
```

Figure 5: Example struct for tiles

After allocating and initializing the storage for the game state, the **server** should randomly place **10** mines on the playfield. To ensure that the game operates deterministically, the following algorithm should be used to place the mines:

```c
void place_mines()
{
  for (int i = 0; i < NUM_MINES; i++)
  {
    int x, y;
    do
    {
      x = rand() % NUM_TILES_X;
      y = rand() % NUM_TILES_Y;
    } while (tile_contains_mine(x, y));
    // place mine at (x, y)
  }
}
```

Figure 6: Algorithm for placing mines

The do..while loop is necessary, as it is possible for the random number generator to choose coordinates where a mine has already been laid.

At this point, the **server** should wait for the **client** to send it a set of coordinates to reveal a tile or place a flag.

It is not a requirement to draw a complex graphical representation of the Minesweeper playfield. Instead, a simple textual representation should be used by the client program to represent:

- The number of remaining mines in the playfield.
- The state of each tile on the grid.

Figure 7 contains a mockup of what your interface should resemble after a new game has been started.

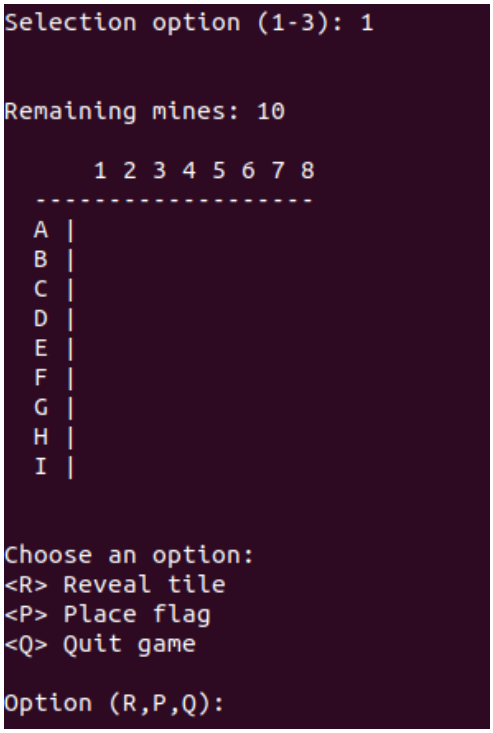

Figure 7: Empty Minesweeper grid

If the player chooses one of the first two options (revealing a tile, or placing a flag), the **client** should prompt for two coordinates, representing the row, then the column. The **client** should then send these coordinates to the server, which will update the internal state accordingly, and send the new game state to the client. If the third option is chosen, the **client** should exit from the game loop, and return to the **main menu**.

If the **client** requests that a tile be revealed, the **server** should first check to see if that tile has already been revealed. If that is the case, an error should be sent to the client. If the tile has not been revealed, one of three paths should be taken:

- If the tile **does not** contain a mine, and has a **non-zero** number of adjacent tiles, the tile should be set to **revealed**, and the number of adjacent mines should be sent to the **client.**

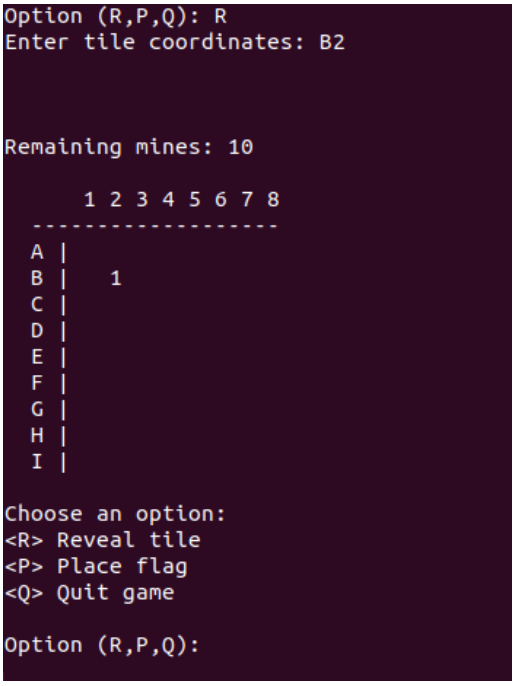

Figure 8: Revealing tile with one adjacent mine

- If the tile contains **zero** adjacent mines, this process should be repeated for all surrounding tiles, stopping when a tile with a non-zero number of adjacent tiles is found. This can be achieved using recursion and checking each of the 8 neighbouring tiles. All these revealed tiles should be sent to the **client.**

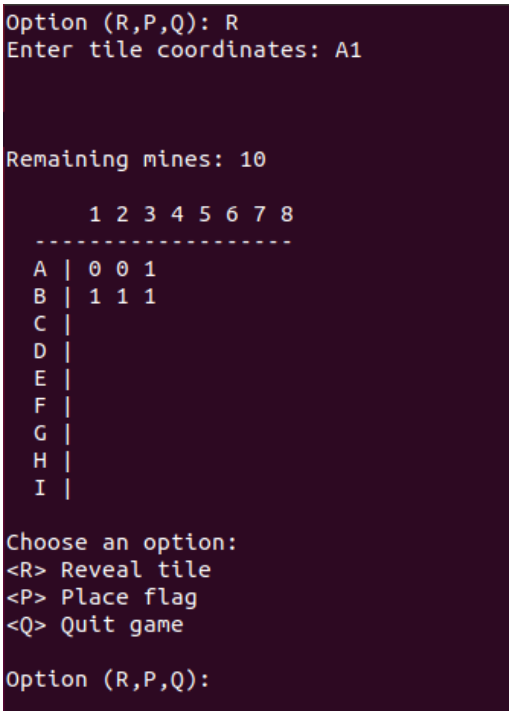

Figure 9: Revealing tile with zero adjacent mines

- If the tile contains a mine, the game is over. The **server** should send the locations of all the remaining mines to the **client**, and the **client** should display these to the player.

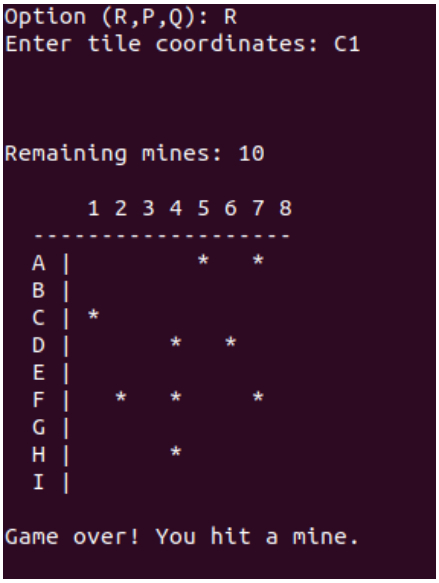

Figure 10: Player reveals a tile with a mine

If the player opts to place a flag, the **client** should again prompt for a pair of coordinates and send this to the **server**. If a mine exists at that location, the **server** should decrement the number of remaining mines, and check for the win condition (explained below). If no mine exists at the location, the **client** should be informed, and display an appropriate message. This is another difference from the classic Minesweeper game.

Figure 11: Placing a flag on a mine

After a flag has been successfully played, the win condition should be checked. This condition is true when there are **zero** remaining mines. If true, the game should finish, report the number of seconds it took to win the game, and the **client** should return to the **main menu**.

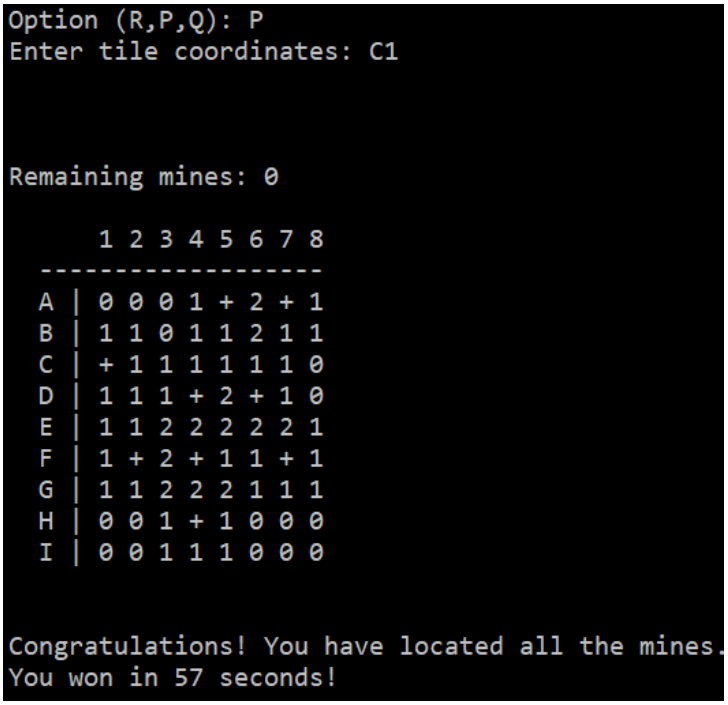

Figure 12: Flags placed on all mines

The amount of time the game has been played for needs to be tracked by the server, to prevent players from cheating. Make use of time functions in the C standard library (such as time()) to store when a player begins their game and when they finish.

A leaderboard needs to be maintained for all games of Minesweeper played. A registered user can request an up-to-date report of the fastest games of Minesweeper that have been played. The output for the leaderboard can be seen in Figure 13, and should be displayed in *descending order* of the number of seconds each successful game took to complete. This is to ensure the best-played games (that is, those games that were played in the least time) appear at the bottom and the player does not need to scroll up to see these.

If two or more games have the same number of seconds then the game played by the player with the highest number of games won should be displayed last. If two or more games were won in the same number of seconds by players with the same number of games won then display those games by the names of their players in alphabetical order.

If no games have been won, an empty leaderboard should be displayed for as shown in Figure 14. The lifetime of the leaderboard is only to be maintained for the duration of the server runtime. This information does not need to be persistent between server instances (so if the server is shut down and restarted, the leaderboard will once more be blank until some games have been won.)

```
Please enter a selection:
<1> Play Minesweeper
<2> Show Leaderboard
<3> Quit

Selection option (1-3): 2

================================================================================

Jason               1421 seconds        3 games won, 5 games played
Peter                152 seconds        2 games won, 2 games played
Timothy              104 seconds        2 games won, 3 games played
Richie                60 seconds        2 games won, 2 games played
Anthony               51 seconds        2 games won, 2 games played
Peter                 40 seconds        2 games won, 2 games played
Justin                38 seconds        2 games won, 4 games played
Maolin                36 seconds        2 games won, 2 games played
Anna                  34 seconds        2 games won, 3 games played
Justin                33 seconds        2 games won, 4 games played
Paul                  33 seconds        2 games won, 5 games played
Jason                 32 seconds        3 games won, 5 games played
Timothy               30 seconds        2 games won, 3 games played
Mike                  29 seconds        2 games won, 2 games played
Paul                  29 seconds        2 games won, 5 games played
Jason                 29 seconds        3 games won, 5 games played
Richie                28 seconds        2 games won, 2 games played
Anna                  26 seconds        2 games won, 3 games played
Anthony               24 seconds        2 games won, 2 games played
Mike                  19 seconds        2 games won, 2 games played
Maolin                 7 seconds        2 games won, 2 games played

================================================================================

Please enter a selection:
<1> Play Minesweeper
<2> Show Leaderboard
<3> Quit

Selection option (1-3):
```

Figure 13: Example output of the leaderboard

```
Please enter a selection:
<1> Play Minesweeper
<2> Show Leaderboard
<3> Quit

Selection option (1-3): 2

================================================================================

There is no information currently stored in the leaderboard. Try again later.

================================================================================


Please enter a selection:
<1> Play Minesweeper
<2> Show Leaderboard
<3> Quit

Selection option (1-3):
```

Figure 14: No statistics for any players – no game has been played since server started.

IMPORTANT NOTE: The company has specifically requested that location of the mines must **never** be sent to the client, to prevent unscrupulous users of the Minesweeper game system from cheating. Therefore, the state of all tiles should remain unknown until either a flag is placed on the mine tile, or the tile is revealed. After each move is played, the coordinates must be sent to the server for processing. This means most of the game logic must be

implemented on the server side. How this is accomplished is an implementation decision for you, as the programmer, to decide.

The client/server system is all console based. No bonuses will be given for complex user interfaces. The implementation for Task 1 does not need to handle concurrent connections and therefore does not need to be multithreaded.

Task 2: Multithreaded Programming & Process Synchronisation

You only need to attempt this task if you are aiming to achieve a mark up to 84% for this assignment.

The server you have implemented in Task 1 can only handle a single request at a time. You are required to extend the server to accept multiple concurrent connections. As each new connection is made, a new thread is created to handle the client request.

The server will allow multiple clients to use the system at the same time. You will need to implement a system whereby the server can accept multiple concurrent connections. When a client exits the thread allocated to the client is destroyed.

One potential problem that you must handle in this multithreaded programming task is a so-called Critical-Section Problem because the leaderboard is a database that may be shared among multiple concurrent connections, each of which may read or update the database. You must make sure that there will no connection that is updating the leaderboard while other connections are reading it. However, it is fine to allow multiple connections to read the leaderboard concurrently.

This critical-section problem is similar to the Readers-Writers Problem discussed in Lecture 5. Thus, you may reuse the solution to the Readers-Writers Problem. In this critical-section problem, each client (connection) may play the role of Reader or Writer at a time. While the client (connection) plays the Minesweeper game, its role is Writer as at the end of the game, it will update the leaderboard; while the client (connection) displays the leaderboard, its role is Reader as it only read data from the leaderboard, but never update the leaderboard.

A second issue to note is the random number generator. As C's rand() function is not thread-safe, accesses to this function **must** be synchronised (e.g. with a mutex).

Task 3: Thread Pool

You only need to attempt this task if you are aiming to achieve a mark up to 100% for this assignment.

The server will allow up to 10 clients to use the system at the same time. After the maximum of 10 clients have connected, if a new client tries to connect either the new client will have to wait for another user to exit or the connection may be dropped.

You will need to implement a thread pool where each incoming connection is handled in a separate thread. A thread pool reduces the cost of constantly creating and destroying threads to handle requests. Instead, threads are reused to handle connections. It also limits the number of incoming connections to guarantee the performance of the server.

You can implement your own Thread Pool reusing the producer consumer pattern from the weekly practicals, where the main thread accepts incoming sockets and places them in a queue (the producer) and 10 threads remove sockets from the queue and process the connection (the

consumer). You may also use or adapt the complete Thread Pool implementation provided in the weekly practicals.

2 Submission

Your assignment solution will be submitted electronically via Blackboard before 11:59pm on 26th October 2018. If you work in a group, you only need to submit one copy of your group assignment. Your submission should be a zip file and the file name should contain the student number(s) and include the following items:

• All source code of your solution.

• The ***make*** file that you used for generating the executable files, if any.

• A report in Word Document Format or PDF which includes:

a) A statement of completeness indicating the tasks you attempted, any deviations from the assignment specification, and problems and deficiencies in your solution.
b) Information about your team, including student names and student numbers, if applicable
c) Statement of each student's contributions to the assignment. If all members of the group contributed equally, then it is sufficient to state this explicitly.
d) Description of the data structure(s) used for representing the playfield.
e) Description of the data structure that is used for the leaderboard in your implementation.
f) Description of how the critical-section problem is handled in Task 2, if applicable.
g) Description of how the thread pool is created and managed in Task 3, if applicable.
h) Instructions on how to compile and run your program.

Note: If the program does not compile on the command line in the Linux environment used during each weekly practical, your submission will be heavily penalised. Implementations written in Visual Studio or other IDEs that do not compile on the Linux command line will receive a mark of zero (0).