CAB403

Student Name: Xuesen Li

Student ID: n9592059

Group/Individual: Individual\_036

# Tasks Completed

Task 1, Task 2 and Task 3 are all completed.

# Data Structure of Leader Board

The data structure used for the leader board is singly linked list. Specifically, there are two linked lists used for implementing the leader board: record list and leader board list. The record is used to track a user’s historical information of played games, such as player’s name, number of games won, and number of games played, which means each player will have only one record. The record list is composed of each player’s record. The Entry structure is used to store the information of each game won by players. As can be seen in Figure 1, there two fields in this structure. The first field “record” is a pointer pointed to the historical record of the player stored in the record list, which will be shared by all entries with the same player. The second field “duration” is the duration of game won by the player. The leader board list is composed of these entries. In addition, when putting a new entry into the leader board list, the list will make sure that the entry with the longest duration is always in the head and the entry with the shortest duration is always in the tail, which means the list is always in descending order, so the sorting of the list is eliminated, resulting in better performance.

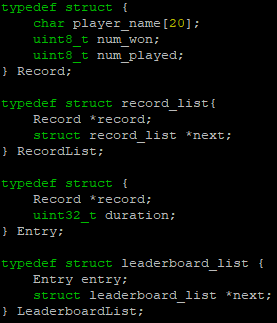


Figure 1: data structure of leader board

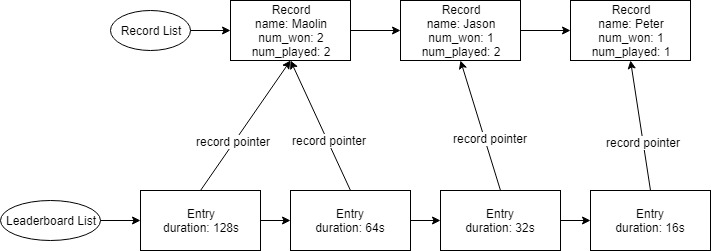


Figure 2: data structure of leader board

# Data Structure of Playfield

As can be seen in Figure 3, the playfield is composed of tiles, each tile has three fields:

* adjancent\_mines: the number of mines around the tile
* revealed: true if the tile has been revealed
* is\_mine: true if the tile contains a mine

In Game structure, NUM\_TILES\_X \* NUM\_TILES\_Y of tiles are stored as well as the duration of the game, number of mines remaining and player’s name.

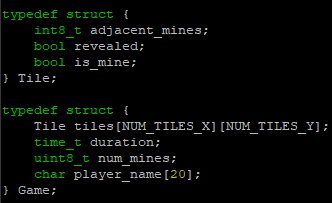


Figure 3: data structure of playfield

# Critical Sections in Task 2

The critical sections in task 2 are the operations of the leader board and the rand() function. In order to synchronize the access of the leader board and rand() function for each thread, two phtread mutex variables are used.





When a thread is trying to update or access the leader board, it needs to get the mutex first. If the operation failed, meaning the leader board cannot be changed or obtained at the moment since there might be another thread that is operating the leader board. In that case, the thread will wait until it obtains the mutex.

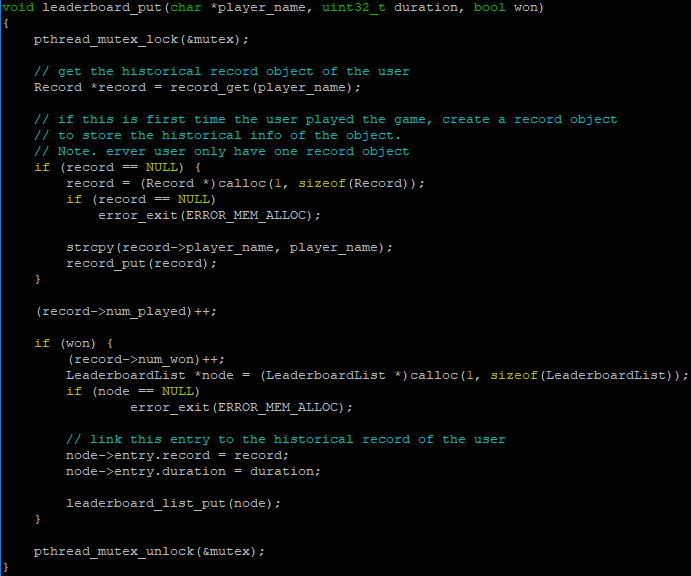


Figure 4: update of leaderboard

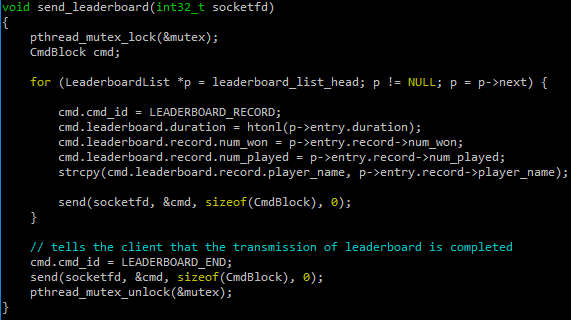


Figure 5: read of leader board

The same idea goes for handling the critical section of rand() since it is not thread safe. By using the variable mutex\_rand, the access to the function rand() is synchronized, resulting in generating correct random number by rand().

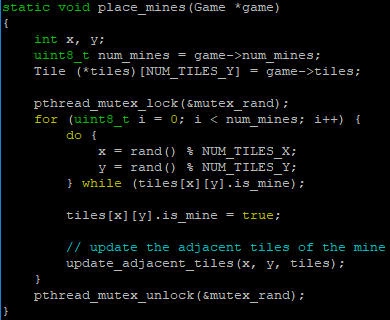


Figure 6: critical section of rand()

# Thread Pool Management

The thread pool creation can be seen in Figure 7, the function below will create an array to store thread objects and assign the task to all threads.

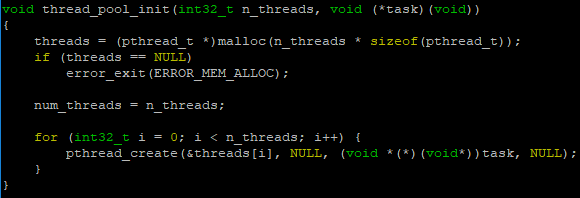


Figure 7: thread pool creation

Once a thread has started, it will try to fetch a socket connected with a client from the socket queue which is an integer array. If the queue is currently empty, it will wait until the writer, the main thread, wakes it up. After that, the thread will process the request by calling the function process\_request, which is the main loop of the gameplay. If the client has disconnected from the server, the thread will close the socket and fetch another socket from the queue.

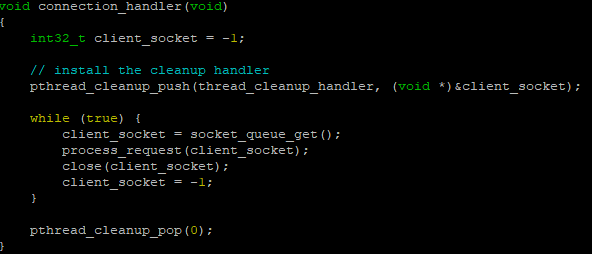


Figure 8: task that every thread will invoke

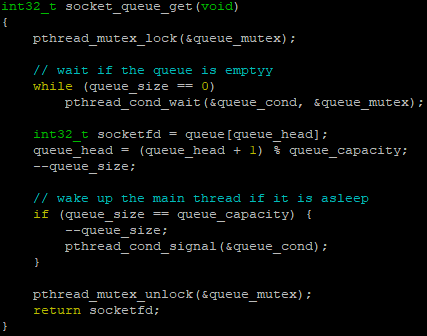


Figure 9: fetch a socket from the socket queue

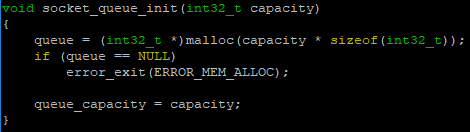


Figure 10: socket queue initialization

The main thread, acting as the socket queue writer, will put a new socket into the queue whenever a connection with a client is established, and broadcast to all threads that are currently waiting. If the queue is full, it will wait until a worker in the thread pool fetched a socket.

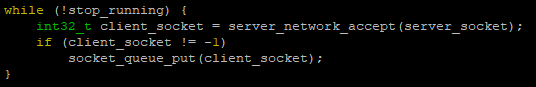


Figure 11: put a socket into the queue

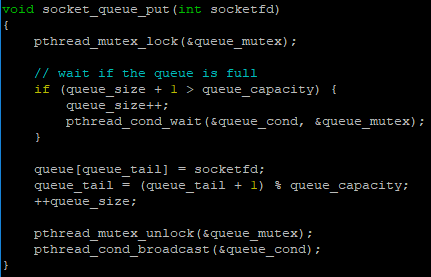


Figure 12: put a socket into the queue

When users press Ctrl-C, the function stop\_server() will be executed, which closes the socket on listening new connections and sets the stop\_running flag to true. Once completed, the main thread will send cancellation signal to every threads by calling thread\_pool\_cancel() which invokes phthread\_cancel() inside. If a thread encounters a cancellation point after the cancellation signal has been sent, it will execute thread\_cleanup\_handler() to deallocate the resources owned. In addition, if the thread is currently waiting for getting a new socket, which means the queue is empty, the thread will be waked up and relock the mutex after reaching the cancellation point, so it has to call socket\_queue\_cancellation() to unlock the mutex.

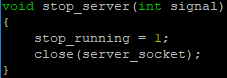


Figure 13: stop server

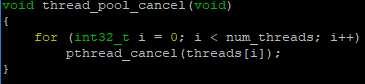


Figure 14: thread cancellation

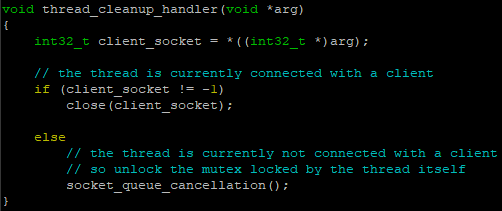


Figure 15: thread cleanup handler

Having sent the cancellation signal, the main thread will wait until all threads in the thread pool has exited successfully, and then exit after all the cleanup functions blow has been completed.

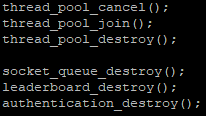


Figure 16: cleanup functions

# Compilation

The way of compiling the program is to simply execute “make” command in the shell, which will generate both server and client program. In addition, if you only want to compile either server program or client program, you can execute “make server” or “make client”.