Assignment Report

Zheng li - 120090155

1 Introduction [2']

In this assignment, we will generate a dungeon containing some continuously moving gold pieces and walls. At the beginning, a `"0"` representing the player will appear at the center of the dungeon. The user can control the player's movement by pressing the `WASD` keys on the keyboard. If the player hits a wall, the game will end and display `"You lose the game"`. If the player collects all the gold pieces, the game will end and display `"You win the game"`. If the player presses `"Q"`, the game will exit and display `"You quit the game"`.

2 Design [5']

2.1 Map Initialization

Map Initialization: This part is given by the TA. It uses a data structure to store every coordinate's value in the entire 17x49 dungeon.

2.2 User Input Handling

User Input Handling: This part is given by the TA. The `kbhit` function is used to detect keyboard input without blocking. It checks for user input to control player movement or exit the game.

2.3 Global Variables

Global Variables: To track the positions of moving elements (player, walls, and gold), I defined global variables as shown below:

```
/* Global variables */
int player_x;         // Current x-coordinate of the player
int player_y;         // Current y-coordinate of the player
char map[ROW][COLUMN + 1]; // Map array, storing the current state of the
    game map

int wall_rows[6] = {2, 4, 6, 10, 12, 14}; // Row positions of the walls
int walls_positions[6]; // Current column positions of the walls

std::map<int, bool> gold_rows;
// Row positions and status of the gold pieces. If a certain gold piece is
    collected by the player, the corresponding bool becomes false.
int gold_positions[6]; // Current column positions of the gold pieces

int collected_gold = 0; // Number of gold pieces collected by the player

enum GameStatus {
    ONGOING,
    WIN,
    LOSE,
```

```
    QUIT
};
GameStatus game_status; // To determine the status of the game
pthread_mutex_t mtx;  // Mutex for thread synchronization
```

2.4 How to determine the game status

One of the most significant changes was replacing the `bool game_over` variable with an enum type called `GameStatus`. The new `GameStatus` enum can represent multiple game states: `ONGOING`, `WIN`, `LOSE`, and `QUIT`.

This new enum provides a clear way to represent different game endings, making the code much easier to understand and maintain.

The game can end in three ways: the player collects all gold pieces (`WIN`), hits a wall (`LOSE`), or the user quits (`QUIT`).

Using an enum allows me to clearly distinguish between these different end states.

2.5 Multithreading

I employed three threads to handle different tasks concurrently:

- **Wall Thread** (`wall_thread_f`): Manages the movement of walls. Walls continuously move in predefined directions across the map, and each wall's position is updated in a loop.
- **Gold Thread** (`gold_thread_f`): Handles the movement of gold pieces, which shift randomly in either left or right directions using the `rand` function.
- **Player Thread** (`player_thread_f`): Manages player input and movement. This thread captures the `W`, `A`, `S`, `D` keys for movement, `Q` to quit, and checks for interactions with walls and gold pieces.

These threads are synchronized using a mutex (`pthread_mutex_t mtx`) to ensure safe modifications to the shared map array, avoiding race conditions.

2.6 Thread Synchronization and Game Status Management

The game status (`game_status`) is used in all threads to determine whether the game should continue running or stop. Threads will continue their execution loop as long as `game_status` is `ONGOING`. When the game ends (either by winning, losing, or quitting), the `game_status` variable is updated, and each thread will recognize this and exit its loop accordingly.

2.7 Collision Handling

Collision with Walls: If the player hits a wall, the `game_status` becomes `LOSE`, and the game ends with a `"You lose the game"` message.

Collision with Gold: If the player moves to a position containing a gold piece, the piece disappears on the screen, and the `collected_gold` counter increases. When all gold pieces are collected, `game_status` becomes `WIN`, and the game ends with a `"You win the game"` message.

Exit: If the player presses `Q`, `game_status` becomes `QUIT`, and the game immediately exits with a `"You quit the game"` message.

2.8 How to find collisions

Wall Collision (`if_player_bump_into_wall`): The function checks if the player's position matches any of the wall positions. If a collision is detected, `game_status` is updated to `LOSE`, and the message `"You lose the game"` is printed.

Gold Collision (`if_player_meet_gold`): This function checks if the player's position matches any gold piece. If it does, the corresponding gold piece is marked as collected.

2.9 Game Speed Control

I used `usleep` to control these intervals, adding a delay between each update for smoother movement.

2.10 Main Function Workflow

The main function initializes the map, sets the positions of walls and gold pieces, and starts the three threads. It waits for the player thread to finish, and finally, it cleans up by joining all threads and destroying the mutex.

3 Environment and Execution [2']

3.1 Running Environment

The program was developed and tested in the following environment:

- **Ubuntu Version**: 16.04
- **Kernel Version**: 5.10.226
- **GCC Version**: 5.4

3.2 Execution Instructions

To compile and run the program, follow these steps:

1. **Navigate to the Source Directory**: Open a terminal and navigate to the directory containing the source code (`hw2.cpp`).
2. **Compile the Program**: Run the following command to compile the program:

   ```
   g++ hw2.cpp -lpthread
   ```

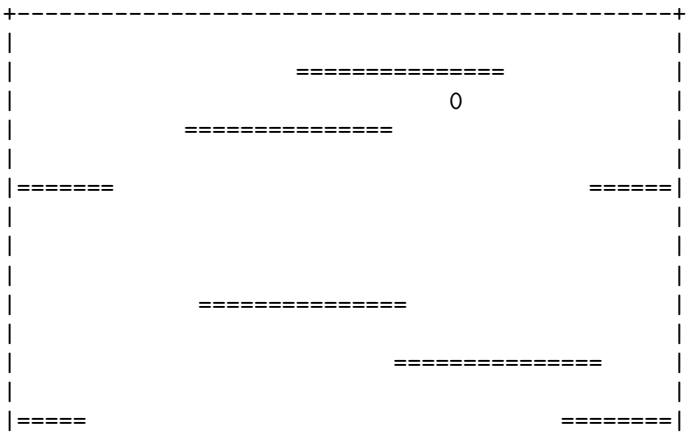
   The `-lpthread` flag is necessary to link the pthread library.
3. **Execute the Program**: After compilation, run the executable using:

   ```
   ./a.out
   ```

3.3 Program Output Examples

Here are examples of the program's output in different scenarios:

3.3.1 Example: Winning the Game



```
|                                              |
+----------------------------------------------+
You win the game
```

3.3.2 Example: Losing the Game

```
+----------------------------------------------+
|       $                                      |
|===                                 ==========|
|         $                                    |
|                          ===============     |
|                                   $          |
|                  ===============             |
|                                              |
|                                              |
|                                              |
|          =============0=                     |
|                                              |
|                        ===============       |
|    $                                         |
|=======                                ======|
|                                    $         |
+----------------------------------------------+
You lose the game
```

3.3.3 Example: Quitting the Game

```
+----------------------------------------------+
|                      $                       |
|============                                =|
|               $                              |
|                                ==============|
|                        $                     |
|=                                 ============|
|                                              |
|                         0                    |
|                                              |
|=========                                 ====|
|             $                                |
|=======                                ======|
|    $                                         |
|       ===============                        |
|                 $                            |
You quit the game
```

4 Conclusion [2']

I have learned a series of pthread operations such as `pthread_create`, `pthread_exit`, `pthread_join`, `pthread_mutex`, and `pthread_condition`. I learned what multithread programming is and how to avoid race conditions using mutexes and conditions. I learned terminal control such as how to clear the screen and how to control the cursor using `printf`. I learned how to generate a random number and use `srand` to generate different `rand` values. I learned how to suspend a thread using `usleep()` or `sleep()`.