Assignment 2 Report: The Greatest Adventurer

Shuhua Zhang - 122090759

No page limitation

1 Introduction [2']

The project implements a multi-threaded game called The Greatest Adventurer, where the player navigates a dungeon to collect six gold shards while avoiding moving walls. The game involves real-time input handling, object movement, and synchronization between multiple threads. The program used the pthread library to control the movements of the player, walls, and gold shards in separate threads, ensuring proper synchronization through mutex locks.

The main elements in the game include:

Player: Moves in response to user input to collect gold shards.

Walls: Move horizontally across specific rows.

Gold Shards: Move horizontally across specific rows.

Game Mechanics: The game ends if the player collects all gold shards (win), collides with a wall (lose), or presses 'Q' to quit.

2 Design [5']

We mainly need to tackle 3 aspects: handle user commands of the player's motion, automatic movements of the walls and gold shards, and dynamically displaying the game interface. The primary challenge is to synchronize the movements and interactions between these components using threads and mutexes.

There are 14 threads in total:

6 threads for tracing the movements of each wall (execute the function `move_wall()`).

6 threads for tracing the movements of each gold shard (execute the function `move_gold()`).

1 thread for tracing the movement of the player (execute the function `move_player()`).

1 thread for displaying the screen (execute the function `screen_print()`).

A mutex lock `pthread_mutex_t` is created and used in functions `move_wall()`, `move_gold()`), `move_player()` and `screen_print()` to ensure safe access to shared resources. This is particularly important because multiple threads (for the player, walls, and gold shards) are updating the shared resources concurrently. By locking the mutex before modifying and unlocking it afterward, the program avoids race conditions and ensures data consistency.

Function `move_player()` traces the movement of the player according to the keyboard input of users.

1. The `kbhit()` function checks for keyboard input without blocking the thread.

2. When a key ('W', 'A', 'S', or 'D') is pressed, the player's position (player_x, player_y) is updated based on the direction of movement. For example, moving upwards means the "row" coordinate should be decreased by 1.

```
player_x--;
map[player_x][player_y] = PLAYER;
map[player_x + 1][player_y] = ' ';
```

When the player hit the frontier of the game board, the movement is considered invalid. For example, if player is at (1, 1), it can not move leftwards or upwards.

3. Handle collision. When the player crush on walls, variable `isover` will be wet to 1 and the game will end with the state 3 (lose). When the player collects the gold shards, the corresponding element in the map `gold_positions` will be set to -1 to indicate that this shard is consumed and the global variable `count` will be increased by 1. When the player collects all the gold shards, i.e. `count == 6`, `isover` will be wet to 1 and the game will end with the state 2 (win).

4. The termination of the thread is organized by variable `isover` in the loop `while(!isover)`, when the player collects all the 6 gold shards (win) or crush in walls (lose), `isover` will be set to 1 to end the thread.

Function `move_wall()` is designed to handle the movement of the walls. Walls in row 2, 6, 12 move from left to right, while in row 4, 10, 14 walls move from right to left, they wrap around boundaries when they reach the edge.

1. Specify the start position of each wall. Use `srand()` function to set random seeds using the thread info, and `rand()` function to determine the starting position.

```
srand(time(0) + (unsigned long)pthread_self());
int start = rand() % 47 + 1;
```

2. The movement of the walls is controlled by the loop `while(isover)`. Every iteration corresponds to a single static position of the wall. The mutex lock is placed within the iteration body.

3. Refresh the whole row by setting the positions (except where player is at) to empty elements " " at the start of each iteration.

```
for (i = 1; i < 48; i++){
    if (map[row_index][i]!=PLAYER){
        map[row_index][i] = ' ';
    }
}
```

4. When the player crush on walls, handle collision. This is the same case as in `move_player()`.

5. In order to make the movements look smooth and not too fast, the `usleep()` function is adopted to suspend execution of the calling thread, the sleeping time interval is set to 1 second.

Function `move_gold()` is designed to handle the movement of the gold shards. Shards are in in row 1, 3, 5, 11, 13, 15, and their moving directions are randomly set.

1. Specify the position of each gold shard. This part is same as in the `move_wall()`.

2. The movement of the shards is controlled by the loop `while(isover)`. Every iteration corresponds to a single static position of the shard. The mutex lock is placed within the iteration body.

3. Refresh the whole row by setting the positions (except where player is at) to empty elements " " at the start of each iteration.

4. When the player collects the gold shards, handle collision. This is the same case as in `move_player()`.

5. In order to make the movements look smooth and not too fast, the `usleep()` function is adopted to suspend execution of the calling thread, the sleeping time interval is set to 1 second.

Please notice that, **the check for the collision with gold shards should be carried simultaneously in `move_player()` and `move_gold()`**. If only carried in `move_gold()`, the check would fail when the player is moving at a high speed. The reason is that in `move_gold()`, the collision condition is checked every 1 second. However, during the 1 second period, the player would have already went through the shard. The same for collision with walls.

3 Environment and Execution [2']

3.1 Environment

1. Linux Distribution: Ubuntu 16.04 (checked using `lsb_release -a`)
2. Kernel Version: 5.10.10 (checked using `uname -r`)
3. GCC Version: 5.4.0 (checked using `gcc -v`)

3.2 How to run my program

1. Move your directory to `/csc3150/Assignment_2_122090759/source/` (**you may change this according to your directory**)
2. Compile the program using: `g++ hw2.cpp -lpthread`
3. Run the program using: `./a.out`

3.3 Demo output

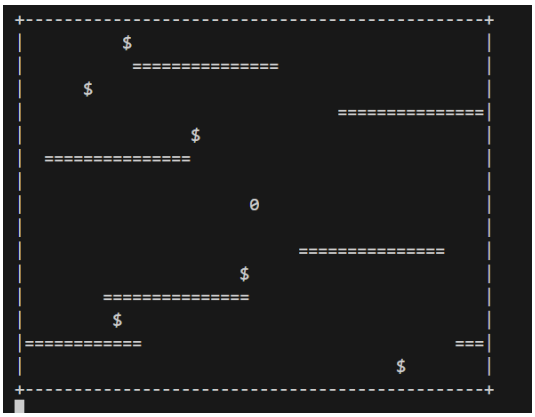

Figure1. Game map

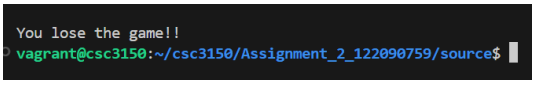

Figure2. Lose

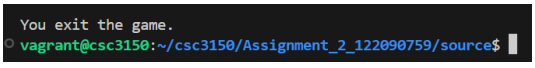

Figure3. Exit

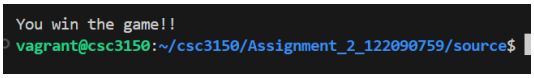

Figure4. Win

4 Conclusion [2']

The assignment provided a valuable hands-on experience in using `pthreads` for a real-time game application.

I learned how to create threads, join threads and use lock to avoid dead lock. I have a deeper understanding on the mutex mechanism and its protection on reading and writing data. I learned how to manage keyboard hit. I learned how to output at an appropriate frequency by adjusting the time interval in the `usleep()` function.