Assignment Report: Game Design Using Multithreading in C

Meng Ziqi - 120020153

October 18, 2024

1 Introduction [2']

This assignment revolves around designing a text-based adventure game using C programming and multithreading to manage concurrent tasks. The game takes place on a grid map where the player controls an adventurer who needs to collect gold while avoiding obstacles like moving walls. The objective of the game is to collect all gold pieces without being hit by any moving walls. The game includes several concurrent elements such as the movement of the adventurer, walls, and gold pieces, which are handled by using threads. The challenge is to synchronize these moving parts using mutex locks and avoid potential collisions or race conditions.

The goal of the assignment is to enhance understanding of multithreading in C and how to implement real-time, dynamic interactions in a game environment. It requires knowledge of thread synchronization and non-blocking input techniques.

2 Design [5']

The design of the game is based on several core components, each contributing to the overall functionality of the game. Here is a detailed breakdown of each part:

2.1 Game Map Layout

The map is a 17-row by 49-column grid, defined as a 2D array in C. The borders of the map are surrounded by walls ('—' and '-') to prevent the adventurer from moving out of bounds. Inside the grid, there are dynamic elements like moving walls and gold pieces.

Grid Structure:

- Rows and columns form the playable area, with boundaries set at the top, bottom, left, and right edges.

- The adventurer is initialized at a fixed position in the center of the map.
- Six walls are placed horizontally in the middle rows of the map, which move continuously.
- Six pieces of gold are randomly scattered across the grid.

2.2 Multithreading Approach

Multithreading is used to handle the different elements of the game running concurrently. Each moving part (the adventurer, walls, and gold) is controlled by a separate thread. This allows real-time updating of the game state without blocking user input or causing delays in wall and gold movements.

Threads:

- **Adventurer Movement:** The adventurer's movement is controlled via keyboard input, with non-blocking input detection using 'kbhit()'. A thread is dedicated to processing the user's directional input (via 'w', 'a', 's', 'd') and updating the adventurer's position on the map.
- **Wall Movement:** Each wall moves horizontally across the map, either left or right, depending on the thread's assigned direction. Threads are used to update the wall's position at regular intervals, ensuring that they move independently.
- **Gold Movement:** Gold pieces move randomly across the map. Each gold piece has its own thread that updates its position at periodic intervals. The gold moves similarly to the walls but with a random direction assigned to each piece.
- **Map Refreshing:** A separate thread is used to continuously refresh and render the map on the terminal, ensuring that all movements and changes are visually updated in real time.

2.3 Synchronization with Mutex

Since multiple threads access and modify the shared game map, proper synchronization is crucial to prevent race conditions. A mutex lock is used to ensure that only one thread can modify the game state at any given time. Whenever a thread needs to update the map (e.g., moving the adventurer, walls, or gold), it locks the mutex before making changes and unlocks it after finishing, ensuring the game state remains consistent.

Synchronization Details:

- Mutex locks ensure that threads accessing shared resources (the game map) do not conflict with one another.
- The threads responsible for moving the adventurer, walls, and gold must wait for the lock to become available before updating the map, which prevents data corruption.

2.4 Collision Detection and Game Logic

The game ends if:

- The adventurer collides with a moving wall. This triggers a loss, and the game terminates with a message stating that the player has lost.
- The adventurer collects all the gold pieces. The game terminates with a winning message once all gold is collected.

Collision detection is implemented by checking the adventurer's position relative to the walls and gold in each game cycle. If the adventurer's coordinates match a wall or gold's position, the appropriate action is taken (end game or increment gold count).

3 Environment and Execution [2']

The program was developed and tested on a Linux system using the GCC compiler. It makes use of POSIX thread libraries (pthread) to manage concurrency. The game also utilizes ANSI escape codes to handle real-time screen updates and cursor movement within the terminal.

3.1 Development Environment:

- Operating System: Ubuntu 16.04.7 LTS
- Compiler: GCC 5.4.0
- Linux Kernel Version: 5.10.1

3.2 Compilation and Execution:

To compile the program, the following command is used:

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

To run the game, the following command is executed:

```
./a.out
```

The adventurer is controlled using the keyboard:

- **W:** Move up
- **A:** Move left
- **S:** Move down
- **D:** Move right
- **Q:** Quit the game

The game state is refreshed every 0.01 seconds, providing real-time feedback as walls and gold move. The game will automatically terminate when either all gold is collected (player wins) or a wall collision occurs (player loses).

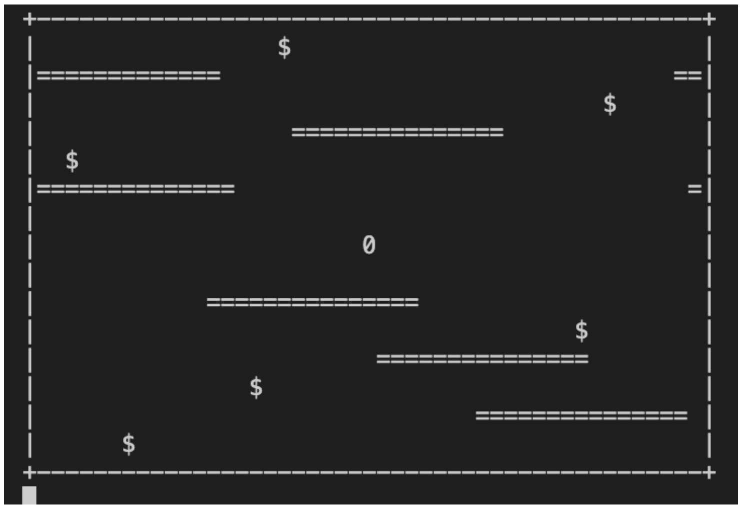
4 Conclusion [2']

This assignment provided valuable insights into the use of multithreading and synchronization in C programming. The main challenge was ensuring thread safety when multiple threads were simultaneously accessing and modifying the shared game state. By implementing mutex locks, the program successfully avoided race conditions and maintained a consistent game state.

Furthermore, managing non-blocking input and rendering real-time updates in a terminal-based environment introduced additional complexities. Through this project, I gained a deeper understanding of thread management, synchronization mechanisms, and the challenges of real-time application development. Overall, this assignment was instrumental in reinforcing the principles of concurrency and multithreading, as well as enhancing my problem-solving skills in game design and interactive programming.

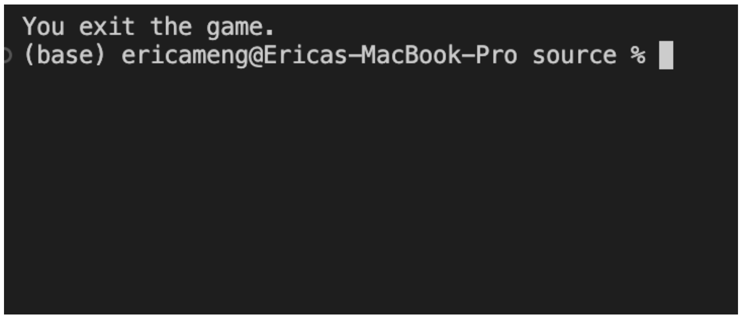
4.1 Results

The following screenshots demonstrate various states of the game:



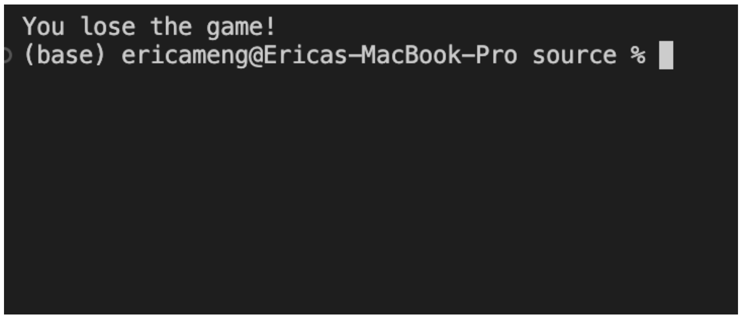
```
+-----------------------------------------------+
|                 $                             |
|=============                                ==|
|                                        $      |
|                  ===============              |
|  $                                            |
|==============                                =|
|                                               |
|                       0                       |
|                                               |
|            ===============                    |
|                                      $        |
|                        ===============        |
|               $                               |
|                               =============== |
|      $                                        |
+-----------------------------------------------+
```

Figure 1: Game Start Screen



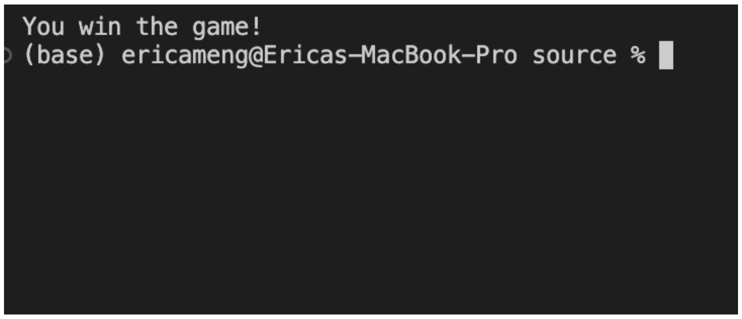
```
You exit the game.
(base) ericameng@Ericas-MacBook-Pro source %
```

Figure 2: Exit Screen



```
You lose the game!
(base) ericameng@Ericas-MacBook-Pro source %
```

Figure 3: Lose Screen



```
You win the game!
(base) ericameng@Ericas-MacBook-Pro source %
```

Figure 4: Win Screen