Assignment Report: CSC3150 Operating Systems - Assignment 2

Xingqiliang - 123090669

1 Introduction [2']

This assignment focuses on implementing a multi-threaded interactive game using pthread in C++. The program simulates a terminal-based game where a player navigates through a map, collecting gold pieces while avoiding moving walls. The game demonstrates practical applications of concurrent programming concepts, particularly thread synchronization and shared resource management.

The core implementation utilizes the pthread library to create multiple concurrent threads that handle different aspects of the game:

- Player Movement Thread: Captures keyboard input asynchronously and updates the player's position in real-time using WASD controls.
- Wall Movement Threads: Six independent threads control horizontal wall movements across different rows, with alternating directions (left and right) to create dynamic obstacles.
- Gold Movement Threads: Six threads manage the horizontal movement of gold pieces, which the player must collect to win the game.
- Auto-Refresh Thread: Continuously updates the terminal display to render the game state at approximately 20 frames per second.

The primary challenge in this multi-threaded architecture is ensuring thread-safe access to shared resources, particularly the game map array. This is achieved through mutex locks (`pthread_mutex_t`) that protect critical sections where threads read or modify the shared game state. The program demonstrates essential concurrent programming techniques including thread creation, synchronization, collision detection, and graceful termination of multiple threads.

This implementation serves as a practical exploration of parallel processing, race condition prevention, and the coordination of multiple asynchronous tasks within a single application.

2 Design [5']

2.1 Overall Architecture

The program employs a multi-threaded architecture with 14 concurrent threads (1 player thread, 6 wall threads, 6 gold threads, and 1 refresh thread) coordinated through a single mutex lock. The game state is maintained in a shared 2D character array (`map[ROW][COLUMN+1]`) that serves as the central data structure accessed by all threads.

2.2 Thread Design and Synchronization

2.2.1 Mutex-Based Synchronization

All threads access the shared game map through mutex-protected critical sections using `pthread_mutex_lock()` and `pthread_mutex_unlock()`. This prevents race conditions that could corrupt the game state or cause undefined behavior. The mutex is initialized in the main function via `pthread_mutex_init()` and destroyed at program termination with `pthread_mutex_destroy()`.

2.2.2 Player Movement Thread

The player thread continuously polls for keyboard input using a custom `kbhit()` function. Upon detecting input, it locks the mutex, updates the player's position based on WASD keys, performs collision detection, and unlocks the mutex. The thread terminates when the player presses 'Q' or when a collision with a wall occurs.

2.2.3 Wall Movement Threads

Six wall threads are created, each controlling a 15-character horizontal wall on different rows. Even-indexed walls move right, while odd-indexed walls move left, creating alternating patterns. Each thread updates its wall position every 100ms, implementing wrap-around behavior at map boundaries. Collision detection with the player is performed during each update, triggering game termination if contact occurs.

2.2.4 Gold Movement Threads

Six gold threads independently manage gold piece movements. Each gold piece moves horizontally with a randomly initialized direction (left or right). When a gold piece reaches the player's position, it is marked as collected by setting its coordinates to (-1, -1), and the `gold_collected` counter increments. The game ends successfully when all six gold pieces are collected.

2.2.5 Auto-Refresh Thread

A dedicated refresh thread continuously redraws the game map at approximately 20 FPS (50ms intervals). This separates rendering logic from game logic, ensuring smooth visual updates regardless of the varying update frequencies of other threads.

2.3 Terminal Input Handling - kbhit() Implementation

Critical Bug Fix: The original template's `kbhit()` function had a significant flaw where it occasionally changed terminal settings on every call, causing flickering and unintended echoing of user keystrokes to the screen. This resulted in visual artifacts and disrupted the game experience.

Solution Implemented: The modified implementation uses the `select()` system call with a zero timeout to check for keyboard input availability without altering terminal settings. Additionally, raw mode is enabled once at program start using `enable_raw_mode()`, which disables canonical mode and echo globally. This approach eliminates the need for repeated terminal reconfiguration and resolves the keystroke echo bug completely.

The key improvements include:

- Using `select()` with `fd_set` for non-blocking input detection
- Setting `O_NONBLOCK` flag on `STDIN_FILENO` for non-blocking reads
- Disabling `ICANON` and `ECHO` flags in `termios` structure once during initialization
- Restoring original terminal settings via `disable_raw_mode()` on program exit

2.4 Game Logic and Collision Detection

The program implements three types of collision detection:

1. Player-Wall Collision: Checked after player movement and during wall updates. Results in immediate game loss.
2. Player-Gold Collision: Checked after player movement. Removes the gold piece and increments the collection counter.
3. Gold-Player Collision: Checked during gold movement. Handles the case where moving gold reaches the player's position.

2.5 Thread Lifecycle Management

The main thread creates all worker threads using `pthread_create()` and waits for their completion using `pthread_join()`. The `stop_game` flag serves as a shared termination signal—when set to 1, all threads exit their main loops and call `pthread_exit()`. This ensures graceful shutdown and proper resource cleanup.

3 Environment and Execution [2']

3.1 Development and Runtime Environment

3.1.1 Operating System Configuration

The program was developed and tested on the following operating system environment:

- Operating System: Ubuntu 20.04.6 LTS (Focal Fossa)
- Kernel Version: Linux 5.15.10 (Custom compiled on October 6, 2025 for ASS1 and reused)
- System Architecture: x86_64
- Virtualization Platform: VMware® Workstation 17 Pro (Version 17.5.2 build-23775571)

3.1.2 Development Toolchain

The compilation and development toolchain consists of:

- GCC Compiler: Version 9.4.0 (Ubuntu 9.4.0-1ubuntu1 20.04.2)
- GNU Make: Version 4.2.1
- Development IDE: Visual Studio Code 1.104.3 (user setup)

3.2 Compilation Instructions

The program can be compiled using the following command:

```
gcc -o hw2 hw2.cpp -lpthread
```

3.3 Execution Instructions

To execute the compiled program, run the following command in the terminal:

```
./hw2
```

Game Controls:

- `W` - Move player up
- `A` - Move player left

- `S` - Move player down
- `D` - Move player right
- `Q` - Quit the game

4 Conclusion [2']

This assignment provided valuable hands-on experience in multi-threaded programming using POSIX threads (pthread) in a practical and interactive application. By implementing a terminal-based game with concurrent moving obstacles, collectible items, and real-time player interaction, I gained deep insights into the complexities and benefits of parallel programming.

4.1 Key Accomplishments

The successful implementation demonstrates mastery of several critical operating system concepts:

- Thread Creation and Management: Created and coordinated 14 concurrent threads with distinct responsibilities, understanding thread lifecycle from creation (`pthread_create()`) to termination (`pthread_join()`).
- Synchronization Mechanisms: Implemented mutex-based critical sections to prevent race conditions when multiple threads access shared resources, ensuring data integrity throughout the game execution.
- Non-blocking I/O: Developed a robust keyboard input handling mechanism using `select()` and terminal raw mode, resolving bugs in the original template and achieving smooth user interaction.
- Resource Management: Properly initialized and destroyed synchronization primitives, restored terminal settings, and ensured graceful program termination under all conditions.

4.2 Broader Understanding

This assignment reinforced the theoretical concepts learned in lectures by providing tangible experience with thread synchronization, mutual exclusion, and the producer-consumer paradigm (where threads both produce and consume game state updates). It also highlighted the challenges of debugging non-deterministic concurrent systems where execution order varies between runs.

Overall, this project successfully bridged the gap between theoretical operating system concepts and practical implementation, equipping me with essential skills for developing robust multi-threaded applications in real-world scenarios.