OS-PROJECT-REPORT

Multi-Threaded Ludo Game

Members

Dawood Hussain(i222410)
Talha Muktiar(i222720)
Momin Nazar(i222491)

SECTION E

Contents

NAMES:

Dawood Hussain (22i2410)
Talha Mukhtiar (22i2720)
Momin Nazar (22i-2491)

System Specifications:

- **Operating System:** Linux Ubuntu 22.04 LTS (On Virtual Box)
- **Processor:** Intel Core i7-11370H
- **RAM:** 16 GB DDR4
- **Compiler:** g++ version 9.3.0
- **GPU:** RTX 3060
- **Libraries:** SFML 2.5.1 for graphics and audio

Introduction:

The purpose of this project is to implement a multithreaded Ludo game using C++ and pthreads. The project simulates the popular board game with multiple players competing on a grid to win. Using binary semaphores ensures thread-safe execution and synchronization, allowing smooth gameplay. The implementation is divided into two phases:

1. **Phase I:** Initializing the Ludo board and tokens.
2. **Phase II:** Implementing player turns, dice rolls, and the overall game logic.

Phase I: Board Initialization and Token Placement

Pseudocode: Ludo Game Implementation

Data Structures

1. **FieldType (Enum):**
   - Represents the type of each cell on the board, e.g., `SAFE_PATH`, `RHOME`, `BHOME`, etc.
2. **Point (Class):**
   - Represents a position on the board using `x` and `y` coordinates.
3. **Pawn (Class):**
   - Properties: `currentLoc`, `isSafe`, `isHome`, `color`, `HasWon`, `pawnPath`, `x`, `y`, `HomeX`, `HomeY`.

- Methods:
  - `getPath()`: Returns the pawn's path.
  - `resetToHome()`: Resets pawn to home position.
  - `setCurrentLoc(position)`: Sets the current location.
  - `getLocation()`: Returns the current `Point` on the board.
  - `moveToNextPosition()`: Moves to the next position.
  - `hasWon()`: Checks if the pawn has won.

4. **Player (Class):**
   - Properties: `ID`, `name`, `Pawns[4]`, `isKiller`, `Turns`.
   - Methods:
     - `InitializePawns()`: Sets up pawns for the player, assigns `pawnPath` based on player color.
5. **LudoBoard (Class):**
   - Properties: `cells`, `pawns`.
   - Methods:
     - `initializeBoard()`: Sets up the board with cell types and colors.
     - `toggleSafe(pawn)`: Toggles the safe state of a pawn.
     - `CheckCollisions(pawn)`: Checks for collisions between pawns.
     - `handleMouseClick(window, event)`: Handles mouse click interactions with pawns.
     - `draw()`: Draws the board and pawns.

Initialization

1. **Board Setup:**
   - Define a 15x15 `board` with `FieldType` values representing each cell type.
   - Initialize color-coded areas for each player (RHOME, BHOME, etc.).
2. **Pawn Setup:**
   - Each player has 4 pawns assigned specific starting positions (`HomeX`, `HomeY`) and paths (`redPath`, `bluePath`, etc.).
3. **Player Setup:**
   - Create 4 players with IDs (0-3) and assign pawns and paths based on color.
4. **Semaphores:**
   - Use semaphores to manage turn-based gameplay among 4 player threads.

Gameplay Mechanics

1. **Dice Rolling:**
   - Randomly generate a number between 1 and 6.
   - Add the rolled number to the active player's turn list.
   - If three 6s are rolled, clear the turn list.
2. **Pawn Movement:**
   - On a valid dice roll, move the selected pawn along its path.

    - If the pawn reaches the winning position (`currentLoc == pawnPath.size()`), mark it as won.
3. **Collision Handling:**
    - Check for pawns occupying the same cell.
    - If a collision occurs and pawns belong to different players, reset the opponent's pawn to its home position.
4. **Safe Zones:**
    - Pawns in `SAFE_PATH` or special zones like `STAR_CELL` are marked as safe and cannot be captured.

Rendering

1. **Draw the Board:**
    - Use SFML to render each cell in the grid with appropriate colors based on `FieldType`.
2. **Draw Pawns:**
    - Render pawns with circular shapes, applying effects like shadows, gradients, and reflections for aesthetics.
3. **Dice Display:**
    - Render a dice shape with dots representing the rolled number.
4. **Active Player Indicator:**
    - Highlight the current player's area with a marker (e.g., a black arrow).

Game Loop

1. **Initialization:**
    - Initialize the SFML window, board, players, dice, and threads.
2. **Event Handling:**
    - Detect mouse clicks to roll the dice or move pawns.
3. **Turn Management:**
    - Use semaphores to coordinate turns between players.
4. **Collision and Win Check:**
    - Check for collisions and determine if a player has won (all pawns reached the finish).
5. **Rendering:**
    - Clear the screen, draw the board, pawns, dice, and active player marker.
    - Display the updated state in the SFML window.
6. **Game Over:**
    - Exit the loop and clean up resources when a player wins.

Role of Binary Semaphores:

Binary semaphores are utilized to:

- **Ensure Mutual Exclusion:** Only one player can access the dice at a time.

- **Enforce Turn Order:** Player threads are activated in sequence using semaphores, ensuring fairness.
- **Simplify Synchronization:** Binary semaphores act as a lock mechanism, controlling access to critical sections such as dice rolling and token movement.

Example Scenario(For using same Technique):

Real-Life Scenario: Airport Runway Management

Overview

Consider an airport with multiple runways but limited availability. Each runway can accommodate one airplane at a time, and the airplanes need to follow a synchronized schedule to ensure safe takeoff and landing operations. The logic of **synchronization** and **binary semaphores**, as used in your Ludo implementation, can manage this scenario efficiently.

Context

1. **Actors:**
   - Multiple airplanes (threads) waiting for their turn to use a runway.
   - Air Traffic Control (ATC) acts as the central authority, signaling which airplane can proceed.
2. **Resources:**
   - Limited runways (shared resources).
3. **Synchronization Requirement:**
   - Only one airplane can access a specific runway at a time.
   - Each airplane must wait until its turn for safe operations.

How the Logic Applies

- **Binary Semaphore:**
  Each runway is associated with a binary semaphore. When an airplane needs to use the runway, it waits (`sem_wait`). After completing its takeoff or landing, it signals (`sem_post`) the semaphore to release the runway for the next airplane.
- **Threading Logic:**
  Each airplane is represented as a thread. The ATC system assigns turns to the threads using binary semaphores, ensuring fair and collision-free operations.

Justification for Binary Semaphore Usage:

Binary semaphores are preferred for player turn management due to their simplicity in handling mutual exclusion and synchronization. Unlike mutexes, which are solely for locking, binary semaphores allow signaling across threads, making them suitable for the "turn-based" logic of Ludo. Traditional semaphores might complicate this scenario with additional count management.

Code Example:

```
sem_wait(&player_semaphores[player->ID]); // Player waits for their
turn
sem_post(&player_semaphores[nextPlayer]); // Signals the next player's
turn
```

• Work Distribution:

- **Dawood Hussain (22i2410):** Implemented the Ludo grid and pawn initialization logic.
- **Talha Mukhtiar (22i2720):** Managed player turn logic using binary semaphores and threading.
- **Momin Nazar (22i-2491):** Designed the collision detection and pawn movement algorithms.

Example Scenario for Applicability:

The threading logic used in Ludo can be applied in a **multiplayer chess game**. Each player can control a separate thread for their moves, with semaphores ensuring the turn order and managing board updates safely.

Focus on Ludo Movement Logic:

Movement of pawns is controlled based on the dice rolls. The following snippet illustrates the logic for moving a pawn safely:

```
pawn.setCurrentLoc(pawn.currentLoc + roll);
toggleSafe(&pawn); // Checks if the pawn lands in a safe zone
CheckCollisions(&pawn); // Handles any collisions with opponent pawns
```

Conclusion:

The project demonstrates the practical application of operating system concepts, such as concurrency, synchronization, and thread management, through a multithreaded Ludo game. Binary semaphores ensure proper coordination between player threads, enabling a seamless gaming experience. Future enhancements may include a more sophisticated graphical interface and networked multiplayer functionality.