Project Report: Ludo Board Game

Table of Contents

---

1. Introduction

The Ludo board game project is designed to simulate a classic strategy game using multithreading and synchronization techniques. This report details the design, implementation, and application of operating system concepts in the development of the Ludo game.

---

2. Project Overview

Ludo is a strategy board game for 2 to 4 players. The objective is to race all four tokens from start to finish according to the rolls of a single die. The project involves designing a multithreaded application where each player is represented by a thread. Synchronization techniques ensure that each thread accesses shared resources (like the dice and the board) in a controlled manner.

---

3. Phase I: Ludo Board Game

3.1. Pseudo Code

pseudo

Copy

```
function initializePlayersThread():
    for each player in players:
        create player thread
        initialize player tokens

function playTurnThread(player_id, dice_value):
    lock board_mutex
    while current_player != player_id:
        wait on turn_cond
    if dice_value == 6:
        reset turns_without_six
    else:
        increment turns_without_six
    current_player = (current_player + 1) % MAX_PLAYERS
    broadcast turn_cond
    unlock board_mutex

function hitCheckThreadFunctionThread():
    while game_over is false:
        select random row and column
        checkHitThread(row, column)
```

```
        sleep for a short duration

function masterThreadFunction():
    while game_over is false:
        checkPlayerStatus()
        sleep for a short duration

function checkPlayerStatus():
    lock board_mutex
    for each player in players:
        if player is active:
            if turns_without_six >= MAX_TURNS_WITHOUT_SIX:
                kick out player
            if all tokens are home:
                announce winner
    unlock board_mutex
```

3.2. Operating System Concepts

- **Multithreading**: Each player is represented by a thread.
- **Mutexes**: Used to protect shared resources like the board and dice.
- **Condition Variables**: Used to synchronize the order of player turns.
- **Thread Cancellation**: Used to remove players who cannot make a move for a certain number of turns.

3.3. Implemented Code

cpp

Copy

```
// Code snippet from the provided implementation
pthread_mutex_t board_mutex;
pthread_cond_t turn_cond;
int current_player;
bool game_over;
```

```
void initializePlayersThread() {
    for (int i = 0; i < MAX_PLAYERS; ++i) {
        Player player;
        player.hit_record = 0;
        player.turns_without_six = 0;
        player.is_active = true;
        for (int j = 0; j < MAX_TOKENS; ++j) {
            tokenThread token;
            token.x = -1;
            token.y = -1;
            token.inSafeZone = false;
            token.isHome = true;
            sem_init(&token.semaphore, 0, 1);
            player.tokens.push_back(token);
        }
        players.push_back(player);
    }
}

void playTurnThread(int player_id, int dice_value) {
    pthread_mutex_lock(&board_mutex);
    while (current_player != player_id) {
        pthread_cond_wait(&turn_cond, &board_mutex);
    }
    if (dice_value == 6) {
        players[player_id].turns_without_six = 0;
    } else {
        players[player_id].turns_without_six++;
    }
    current_player = (current_player + 1) % MAX_PLAYERS;
    pthread_cond_broadcast(&turn_cond);
    pthread_mutex_unlock(&board_mutex);
}
```

3.4. System Specifications

- **Operating System**: Linux
- **Compiler**: GCC
- **Libraries**: pthread, semaphore

3.5. Application in Other Scenarios

The synchronization techniques used in this project can be applied to any scenario where multiple threads need to access shared resources in a controlled manner, such as in a multi-user database system.

---

4. Phase II: Ludo Game Implementation

4.1. Pseudo Code

pseudo

Copy

```
function initializeGame():
    initialize players
    initialize board
    initialize dice

function rollDice():
    if dice is not rolling and not waiting for move:
        start dice roll animation
        generate random dice value
        handle dice roll result

function handleDiceRoll(value):
    if current player has no active tokens:
        advance turn
    if value == 6:
        increment consecutive sixes
        save current state
```

```
        if consecutive sixes == 3:
            revert moves
            display lost turn message
            advance turn
    else:
        reset consecutive sixes
    if move is possible:
        wait for move
    else:
        advance turn

function moveToken(token, steps):
    if token is in home and steps == 6:
        move token out of home
    else:
        move token along path
        check for token capture
        update player stats
```

4.2. Operating System Concepts

- **Semaphores**: Used to control access to tokens.
- **Random Number Generation**: Used to simulate dice rolls.
- **Animation**: Used to simulate token movements and dice rolls.

4.3. Implemented Code

cpp

Copy

```
// Code snippet from the provided implementation
void rollDice() {
    if (m_isRolling || waitingForMove) return;
    m_isRolling = true;
    m_diceVisible = true;
```

```
    m_diceRotation = 0;
    m_diceScale = 1.0;
    QSequentialAnimationGroup *sequence = new QSequentialAnim
ationGroup(this);
    QPropertyAnimation *emergeAnim = new QPropertyAnimation(t
his, "dicePos");
    emergeAnim->setDuration(300);
    emergeAnim->setStartValue(buttonCenter);
    emergeAnim->setEndValue(QPointF(buttonCenter.x(), buttonC
enter.y() - 50));
    emergeAnim->setEasingCurve(QEasingCurve::OutQuad);
    sequence->addAnimation(emergeAnim);
    connect(sequence, &QSequentialAnimationGroup::finished, t
his, &LudoBoard::finishDiceRoll);
    sequence->start(QAbstractAnimation::DeleteWhenStopped);
}

void handleDiceRoll(int value) {
    if (!hasActiveTokens(currentPlayer)) {
        advanceTurn();
        return;
    }
    diceValue = value;
    if (diceValue == 6) {
        consecutiveSixes++;
        if (consecutiveSixes == 3) {
            revertConsecutiveSixesMoves();
            displayLostTurnMessage();
            consecutiveSixes = 0;
            waitingForMove = false;
            advanceTurn();
            return;
        }
    } else {
        consecutiveSixes = 0;
    }
```

```
    bool canMove = false;
    for (const Token &token : playerTokens[currentPlayer]) {
        if (!isTokenInHome(token) || diceValue == 6) {
            canMove = true;
            break;
        }
    }
    if (canMove) {
        waitingForMove = true;
    } else {
        consecutiveSixes = 0;
        advanceTurn();
    }
}
```

4.4. System Specifications

- **Operating System**: Linux
- **Compiler**: GCC
- **Libraries**: pthread, semaphore, Qt (for GUI)

4.5. Application in Other Scenarios

The use of semaphores and random number generation can be applied to scenarios like resource allocation in operating systems and random event simulation in games.

---

5. Conclusion

The Ludo board game project successfully demonstrates the application of multithreading and synchronization techniques in a real-world scenario. The project not only provides an engaging game experience but also serves as a practical example of how operating system concepts can be applied in software development.

---

6. Group Details

- **Group Members**: [Hassaan (22i-2434), Awais (22i-2511), Azmar (22i-2716)]
- **Contributions**:

Hassaan (22i-2434)

- **Design and Implementation**:
  - Developed the core game logic, including player initialization and token movement.
  - Implemented the multithreading framework for player threads.
  - Created the synchronization mechanisms using mutexes and condition variables.
  - Designed the game board and token placement logic.
- **Testing and Debugging**:
  - Conducted extensive testing to ensure thread safety and synchronization.
  - Debugged issues related to thread cancellation and player turn management.
- **Documentation**:
  - Wrote the initial draft of the project report, including the introduction and project overview sections.
  - Documented the pseudo code and operating system concepts for Phase I.

Awais (22i-2511)

- **Design and Implementation**:
  - Implemented the dice rolling mechanism and its integration with player turns.
  - Developed the hit detection logic and token capture functionality.
  - Created the master thread to manage player status and game over conditions.

- **Testing and Debugging**:
  - Performed unit testing on dice rolling and token movement functions.
  - Identified and fixed bugs related to token capture and player status checks.
- **Documentation**:
  - Contributed to the pseudo code and operating system concepts for Phase II.
  - Documented the system specifications and provided code snippets for both phases.

Azmar (22i-2716)

- **Design and Implementation**:
  - Developed the GUI components using the Qt framework.
  - Implemented the animation for token movements and dice rolls.
  - Created the player selection dialog and game setup interface.
- **Testing and Debugging**:
  - Conducted integration testing to ensure seamless interaction between the game logic and GUI.
  - Debugged issues related to GUI rendering and user input handling.
- **Documentation**:
  - Wrote the group details section, including member contributions.
  - Documented the application of operating system concepts in other scenarios.