Project Report

Phase I: Initial Setup and Grid Creation

Pseudocode for Phase I: Initialize Game and Grid

Initialize game board with dimensions 15x15

Initialize player list and assign player IDs, colors, and tokens

For each player:

Set tokens in yard and initialize starting positions

Assign starting positions based on player color

Place tokens on the board based on the color

Set up safe spots on the board

For each safe spot:

If the spot is empty, mark it as a safe spot ('S')

Illustration of Operating System Concepts (Phase I)

- **Atomic Variables**: Use atomic<int> for managing the current player's turn to avoid race conditions when multiple threads access it.
- **Random Number Generation**: The dice roll function mimics a random event in a gaming system, which is similar to simulating random processes in OS-level scheduling.
- **Multithreading**: The program uses threads to simulate the concurrent turns of multiple players. Each player runs in a separate thread, allowing for parallel processing of dice rolls and token movement.
- **Mutex Locking**: To prevent race conditions while updating the game state (e.g., when changing the player's turn), a mutex is used to lock the critical sections.

Phase II: Handling Token Movement, Collisions, and Dice Rolling

Pseudocode for Phase II: Token Movement and Collision Detection

Define dice roll function to generate a random number between 1 and 6

Define function to release tokens from the yard:

If token is in the yard:

Set the token to its starting position

Place the token on the board

Otherwise, skip

Define function to process dice roll:

If the roll is a 6:

Attempt to release a token from the yard

If no token can be released, grant the player an extra turn

If the player rolls three consecutive sixes, forfeit the turn

Otherwise, move the token by the rolled value

Define player routine:

For each player:

Check if it's the player's turn

Roll the dice

Process the dice roll

Move the token if necessary

Display the updated board

Pass the turn to the next player

Illustration of Operating System Concepts (Phase II)

- **Thread Synchronization**: Ensure that the game proceeds turn by turn. Use mutex to lock the section of the code that handles dice rolls and token movements, ensuring that no two players can perform these actions simultaneously.

System Specifications

1. **Grid Size**: 52x16 grid (52 rows, 16 columns)
2. **Token Types**: Four colors (Red, Yellow, Green, Blue)
3. **Dice**: A single 6-sided die, randomly rolling values from 1 to 6.
4. **Player Types**: Four players, each with 4 tokens.
5. **Synchronization**: Mutex, atomic variables, and semaphores/condition variables for managing access to shared resources.
6. **Threads**: Parent thread handles the game loop, worker threads handle token movement, dice rolling, and grid updates.
7. **Collision Handling**: Tokens will be sent back to their home if they collide with an opponent's token.

Implemented Code with Synchronization

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

#include <time.h>

```
#include <stdbool.h>
#include <string.h>

#define BOARD_DIMENSION 15
#define MAX_PLAYERS 4
#define TOKENS_PER_PLAYER 4
#define HOME_PATH_LENGTH 5

typedef struct {
    int posX, posY;
    bool isInYard;
    int startX, startY;
    bool isInHome;
    bool hasReachedHome;
} GameToken;

typedef struct {
    int playerID;
    GameToken tokens[TOKENS_PER_PLAYER];
    int killCount;
    int sixesRolledConsecutively;
    bool isActive;
    char *color;
} PlayerInfo;

typedef struct {
    int currentTurn;
    pthread_mutex_t mutexLock;
} GameStatus;

typedef struct {
```

```
    int x, y;
} BoardPosition;

GameStatus gameStatus;
PlayerInfo playersList[MAX_PLAYERS];
BoardPosition boardPath[52];
char gameBoard[BOARD_DIMENSION][BOARD_DIMENSION];

BoardPosition safeSpots[] = {
  {2, 6}, {1, 8}, {6, 12}, {8, 13},
  {6, 1}, {8, 2}, {13, 6}, {12, 8}
};

int totalSafeSpots = sizeof(safeSpots) / sizeof(safeSpots[0]);

void initializeBoard() {
  // Initialize the game board with empty spaces
  for (int i = 0; i < BOARD_DIMENSION; i++) {
    for (int j = 0; j < BOARD_DIMENSION; j++) {
      gameBoard[i][j] = ' ';
    }
  }

  // Mark safe spots
  for (int i = 0; i < totalSafeSpots; i++) {
    if (gameBoard[safeSpots[i].x][safeSpots[i].y] == ' ') {
      gameBoard[safeSpots[i].x][safeSpots[i].y] = 'S';
    }
  }
}
```

```
int diceRoll() {
    return (rand() % 6) + 1;
}

void releaseToken(PlayerInfo *player, int tokenIdx) {
    if (player->tokens[tokenIdx].isInYard) {
        int startX = initialPositions[player->playerID - 1][0];
        int startY = initialPositions[player->playerID - 1][1];

        gameBoard[player->tokens[tokenIdx].posX][player->tokens[tokenIdx].posY] = '?';
        player->tokens[tokenIdx].posX = startX;
        player->tokens[tokenIdx].posY = startY;
        player->tokens[tokenIdx].isInYard = false;

        char symbol;
        switch (player->playerID) {
            case 1: symbol = '@'; break;
            case 2: symbol = '#'; break;
            case 3: symbol = '$'; break;
            case 4: symbol = '%'; break;
            default: symbol = '?'; break;
        }

        gameBoard[startX][startY] = symbol;
        printf("Player %d released a token to position (%d, %d)\n", player->playerID, startX, startY);
    }
}

void *playerRoutine(void *arg) {
    PlayerInfo *player = (PlayerInfo *)arg;
```

```
    while (player->isActive) {
        pthread_mutex_lock(&gameStatus.mutexLock);

        if (gameStatus.currentTurn == player->playerID) {
            int roll = diceRoll();
            printf("Player %d's turn. Rolled: %d\n", player->playerID, roll);
            processDiceRoll(player, roll);
            displayBoard();

            gameStatus.currentTurn = (gameStatus.currentTurn % MAX_PLAYERS) + 1;
        }

        pthread_mutex_unlock(&gameStatus.mutexLock);
        usleep(30000);
    }

    return NULL;
}
```

Implementation of Synchronization Techniques

- **Mutex**: Used to protect shared resources like the grid and dice rolls. Each player must lock the mutex before interacting with these resources to prevent race conditions.
- **Atomic Variables**: Used for tracking the current player's turn to ensure that no two players can take their turn at the same time.
- **Condition Variables/Semaphores**: While not explicitly written in the code above, a semaphore can be implemented to ensure that players can take their turns one by one. This can be used in conjunction with the atomic variable current_turn to manage which player gets to play.

Implemention of These Concepts in Other Scenarios

The synchronization techniques used in this project can be applied to many other systems where shared resources need to be managed across multiple concurrent users or processes. For example, in an online multiplayer game, the same concepts can be applied to ensure that only one player can interact with the game environment (e.g., move their character, roll a die, etc.) at a time. Similarly, in operating systems, these concepts can be used to manage access to critical resources like files, printers, or databases, ensuring that no two processes interfere with each other while accessing these shared resources.