OSPROJECT-DOCUMENTATION-1

RULE

There are several different ways of multithreading this application. One suggested strategy is to create threads that check the following criteria:

• Four threads are created and will be assigned to each player.
• A thread to check each row and column of the grid to find out the token and player to be passed for completion. This thread should run randomly to check the hitting token and victim player.

• One thread known as “Master thread” should track for all four players of hit record as no token can enter into its home column if its hit rate is 0. The property of this thread is it can cancel all the other four threads in certain situations. For example: The stopping criteria or thread cancelation will be as follows:

o For consecutive 20 turns if any player can’t get a 6 in a die or cannot hit any opponent player, he/she will be kicked out from the game. (Cancelled by this thread).

o When all tokens of a player reach its home, it will send the signal to this master thread about its completion. The master thread will verify and then kill the respective thread after calculating its position.

Implementation:

Implemented:

1. **Four Threads for Each Player:** Each player is assigned a separate thread that handles their moves and interactions on the board. This is implemented in the `player` function, which is invoked as a thread for each player.

   ```
   pthread_t players[4]; for (int i = 0; i < 4; i++) { int *p_no = new int; p_no[0] = i; pthread_create(&players[i], NULL, &player, p_no); }
   ```

2. **Master Thread:** The "Master thread" is responsible for overseeing the game's progress, including checking the completion of players and potentially cancelling threads based on specific game conditions. This thread is implemented in the `masterthread` function.

   ```
   pthread_t master; pthread_create(&master, NULL, &masterthread, NULL); pthread_detach(master);
   ```

   Inside the `masterthread` function, there is logic to cancel player threads when a player's tokens have all reached home:

```
if(allHouses[i].tokensRemaining==0) { pthread_cancel(players[i]); }
```

Not implemented

However, the specific conditions mentioned in the task, such as cancelling a player's thread if they haven't rolled a 6 or hit an opponent in 20 consecutive turns, are not directly implemented in the provided code. The code does handle the cancellation of threads when a player's tokens have all reached home, but it does not track the number of turns without a 6 or a hit to enforce cancellation based on these criteria.

To fully meet the task's requirements, additional logic would need to be added to track the number of turns without a 6 or a hit for each player and to implement the cancellation based on these conditions. This would involve modifying the `player` function to include this tracking and adding checks in the `masterthread` function to perform the cancellation based on these additional criteria.

RULE-2

Phase-1: You are required to create the grid in this phase. Grid and dice variable should be global and considered as two shareable resources. But it is mandatory that each thread can access single resource at a time. E.g., If first player is rolling the dice, it should release the dice resource before accessing the Ludo board/Grid. Complete structure for the game will be part of this phase.

EXPECTED OUTPUT: Phase I
Main thread will display:
• Complete grid for Ludo board and token placed in their home yards.

Implementation:

Implemented:

1.
   - The dice variables (`dice`, `dice2`, `dice3`) are global, allowing shared access across different threads.
   - The grid or Ludo board is represented by the `absolutePosCoordinates` array and the `allHouses` array, which are also global.
2. **Resource Access Control:**
   - Two semaphores, `dicesem` and `tokensem`, are used to control access to these resources. The `dicesem` semaphore likely controls access to the dice, ensuring that only one player can roll the dice at a time.
   - The `tokensem` semaphore is used to manage access to the tokens on the board, ensuring that token movements by different players do not interfere with each other.
3. **Thread Functionality:**

- Each player is represented by a thread (`player` function). Within this function, the dice are rolled by first acquiring the `dicesem` semaphore, ensuring exclusive access to the dice.
- After rolling the dice and possibly updating the state based on the dice roll, the thread releases the `dicesem` semaphore before proceeding to move tokens on the board.
- To move tokens, the thread must acquire the `tokensem` semaphore, ensuring that no other thread is modifying the board at the same time.

4. **Board Drawing and Initialization:**
   - The `drawBoard` function, which is likely called from the main thread, handles the drawing of the Ludo board. This function is responsible for visually representing the grid and the initial placement of tokens in their respective starting positions.
5. **Main Thread Responsibilities:**
   - The main thread initializes the game environment, including setting up the window, initializing semaphores, creating player threads, and handling the game loop where the board is continuously drawn.

Highlighted Code Snippets:

- **Dice and Board Access Control:** `sem_t dicesem; sem_t tokensem; void* player(void *args) { sem_wait(&dicesem); rollDice(); sem_post(&dicesem); sem_wait(&tokensem); // Token movement logic here sem_post(&tokensem); }`
- **Global Variables for Dice and Board:** `int dice, dice2 = 0, dice3 = 0; int absolutePosCoordinates[52][2]; House allHouses[4];`
- **Drawing the Board:** `void drawBoard() { // Drawing logic for the board }`

This setup ensures that the game's critical sections, particularly those involving dice rolls and token movements, are accessed in a controlled manner, preventing race conditions and ensuring the game's state remains consistent across player actions.

Not Implemented:

Everything in phase-1 is implemented without any issue or problem.

RULE-2

In this phase, you have to give complete Ludo game implementation.

Conditions:

Considering the below conditions:
Token must be implemented using semaphores. They are varying when a resources relies and has been added after dice rolling.

• You have to take user input for number of tokens (maximum 4 and at least 1). This input will be take only once and assigned to each player.

• Turn will be followed randomly for four players. At each iteration, all players will try to access the dice. The one who get the dice will roll it and calculate its turn value.

• Both dice value and turn will be calculated randomly but in one iteration every player should get its turn.

• Player have their own safe squares, where no opponent can hit them. (Clearly, shown with colors in figure 1).

• When there is hit rate 0 of any player and it is near their home side, they are not allowed to enter in home, but complete another circle.

• After rolling of dice each player will release dice resource and then are able to get Ludo board resource. The tokens on the board are moved according to the random numbers of dice from (1 to 6).

Note:
Here you have to take care of mutual exclusion problem. E.g., if player one rolls the dice and get Ludo board access, at that time he has to move 4 steps onward. At the time, another player rolls the dice and get the board resource first. (as player 1 should get it first). Considering they both are having the same final place in the grid after their respective turn, there will be hit condition occurred. Player 1 will be hit by player 2 in correct sequence, but if player 2 get first access it will be hit by player 1 (which is wrong). Hint: You can use conditional variable previous turn here. So, that the player who has the last turn will get access of the Ludo board first.

IMPLEMENTATION

Implemented

The provided code snippet implements a Ludo game with several functionalities that align with the conditions you've specified. Here's a breakdown of how each functionality is implemented:

1. **Token Management with Semaphores**:
    - Semaphores are used to manage the access to the dice and the movement of tokens on the board. Two semaphores, `dicesem` and `tokensem`, are initialized and used to ensure that only one player can roll the dice or move a token at a time, addressing mutual exclusion.
    - Code for semaphore initialization: `sem_init(&dicesem, 0, 1); sem_init(&tokensem, 0, 1);`
    - Code for semaphore usage in player threads: `sem_wait(&dicesem); rollDice(); sem_post(&dicesem); sem_wait(&tokensem); // Token movement logic here`

`sem_post(&tokensem);`

2. **User Input for Number of Tokens**:
    - The number of tokens per player is taken as input at the start of the game. This value is stored in `totalTokens` and is used to initialize tokens for each house.
    - Code snippet: `cout << "Enter number of tokens:"; cin >> totalTokens; while (totalTokens < 1 || totalTokens > 4) { cout << "Tokens must be between 1 and 4, Re-enter.\n"; cin >> totalTokens; }`
3. **Random Turn Order and Dice Rolling**:
    - The turn order is managed by the `currentTurnIndex`, which is updated in each player's thread after they complete their actions.
    - Dice values are generated randomly using the `GetRandomValue` function from the raylib library, ensuring each player gets a turn to roll the dice.
    - Code snippet for dice rolling: `if (newTurn) { dice = GetRandomValue(1, 6); newTurn = false; if (dice == 6) { dice2 = GetRandomValue(1, 6); } if (dice2 == 6) { dice3 = GetRandomValue(1, 6); } }`
4. **Safe Squares and Movement Logic**:
    - Each player has safe squares that are checked during the token movement logic. The `insideSafeHouse` attribute of a token determines if it is in a safe square.
    - Movement and hit logic are implemented in the `moveToken` method of the `House` struct, where conditions check if a token lands on a square with another token and handles hits accordingly.
    - Code snippet for checking safe squares and handling hits: `if (tokens[key].iterator > 50 && this→hitrate > 0) { tokens[key].insideSafeHouse = true; } killToken(color, key);`
5. **Mutual Exclusion and Synchronization**:
    - The use of semaphores ensures that dice rolling and token movements are mutually exclusive operations. This prevents race conditions such as two players moving tokens simultaneously.
    - The `player` function ensures that each player gets a turn to roll the dice and move tokens in a synchronized manner, adhering to the game rules.

This implementation covers the core functionalities required for a basic multiplayer Ludo game, ensuring fair turn management, synchronized dice rolling, and safe token movements with proper handling of mutual exclusion scenarios.

Not Implemented:

Everything is implemented without any issue.

RULE-3

Use any appropriate and efficient synchronization technique for this project. The code must be properly commented and this carries marks too. Group details should appear on separate

page in report. You have to mention the contribution of each group member for the project.

IMPLEMENTATION

Implemented:

The provided code snippet implements a board game using multithreading and synchronization techniques, specifically using semaphores. The synchronization is crucial for managing the shared resources among multiple threads, ensuring that the game logic is correctly executed without race conditions or deadlocks.

Synchronization Technique Used:

The code utilizes POSIX semaphores (`sem_t`) to synchronize access to shared resources, such as the dice values and the current turn index. Two semaphores, `dicesem` and `tokensem`, are used to control the access:

1. `dicesem`: This semaphore controls the access to rolling the dice. It ensures that only one player can roll the dice at a time.
2. `tokensem`: This semaphore manages the turn-taking mechanism, ensuring that one player completes their move before the next player begins.

Implementation Details:

- **Initialization of Semaphores**: `sem_init(&dicesem, 0, 1); sem_init(&tokensem, 0, 1);` Here, `sem_init` is used to initialize the semaphores. The second parameter `0` indicates that the semaphore is shared between threads of a process, and `1` is the initial value of the semaphore.
- **Usage in Player Threads**: `void* player(void *args) { ... sem_wait(&dicesem); rollDice(); sem_post(&dicesem); ... sem_wait(&tokensem); // Player makes a move sem_post(&tokensem); ... }` In the `player` function, which is executed by player threads, `sem_wait` and `sem_post` are used to enter and exit critical sections, respectively. This ensures that dice rolling and turn-taking are done in an orderly manner.

Code Comments:

The code includes comments that explain the purpose of different sections and variables, which is good practice.