1279. Traffic Light Controlled Intersection

Easy Concurrency [Leetcode Link]

Problem Description

The problem describes an intersection of two roads, Road A and Road B. Cars can travel on Road A from North to South (direction 1) and from South to North (direction 2), while on Road B, they can travel from West to East (direction 3) and from East to West (direction 4). There is a traffic light for each road. The key aspects are:

- A green light allows cars to cross the intersection along that road.
- A red light means cars cannot cross and must wait for a green light.
- The lights for the two roads cannot be green simultaneously. When one is green, the other must be red.
- Initially, Road A has a green light and Road B has a red light.
- The traffic lights should avoid causing a deadlock, where cars are perpetually waiting to cross.

Cars arriving at the intersection are represented with a `carId` and `roadId`. The `roadId` signifies which road the car intends to use. The direction of the car is indicated by `direction`. Two functions are provided - `turnGreen`, which turns the traffic light green on the current road, and `crossCar`, allowing the car to cross the intersection.

The core challenge is to design a system using these components to ensure cars can cross the intersection without deadlocks.

Intuition

To ensure a deadlock-free traffic system, we need to manage the traffic lights in a way that each car will eventually be able to cross the intersection. The logic should meet two main criteria:

- No two cars from different roads should cross at the same time.
- Changing the traffic light to green on a road when it is already green is unnecessary and should be avoided.

The solution uses a lock mechanism via the `Lock` class from the threading module to prevent the intersection state from being changed by multiple cars at the same time, ensuring consistency in traffic light changes and crossings. The idea is to synchronize the access to the traffic light state, so at any moment, only one car can influence it.

A `TrafficLight` object maintains the state of the traffic light for both roads with a variable, starting with road A as green (road = 1). When a car arrives at the intersection (`carArrived` method), we acquire the lock to prevent other cars from entering the intersection. Then, we check if the car's road ID is the same as the road currently allowed to cross. If not, we call `turnGreen()` to change the traffic light to green for this car's road and update the `self.road` attribute. Then, we let the car cross by calling `crossCar()` and release the lock so the next car can proceed. This mechanism ensures that the cars will cross one at a time, changing the traffic light only when necessary and thus preventing deadlocks.

Solution Approach

The solution utilizes a simple concurrency control mechanism with a `Lock` from the `threading` module in Python, ensuring that the state of the traffic light is manipulated by only one car at a time.

Here's the step-by-step implementation:

1. An instance of `TrafficLight` is initialized with:
   - A `Lock` object to control concurrent access to the traffic light state.
   - An integer field `road`, initialized to `1`, indicating that road A is green at the start.
2. When a car arrives, it invokes the `carArrived` method with its `carId`, `roadId`, the `direction` it wants to travel, and two callback functions: `turnGreen` and `crossCar`.
3. The first action within `carArrived` is to acquire the lock by calling `self.lock.acquire()`. This step ensures that only one car can execute the following code at any given time, effectively serializing cross-intersection operations.
4. Once inside the critical section, the method proceeds to check if the traffic light needs to be changed. This is determined by comparing the `roadId` of the car with the `road` attribute of the `TrafficLight` instance. If they are not equal, it means the car is on the road that currently has a red light and the light needs to be changed to green. This is performed by:
   - Updating `self.road` to `roadId` of the current car, indicating which road is now green.
   - Calling the `turnGreen()` function to simulate the traffic light changing to green for that road.
5. After ensuring the road is green for the car, the `crossCar()` function is called, allowing the car to cross the intersection.
6. Finally, before exiting, the `carArrived` method releases the acquired lock by calling `self.lock.release()`, allowing the next car to proceed with its attempt to cross the intersection.

The `Lock` effectively serves as a gatekeeper to ensure the intersection's state integrity is maintained, eliminating the chances of Deadlock by serializing the changes to the traffic lights and the crossing of cars. This design is simplistic but sufficient for the stated problem requirements, showcasing a classic use of concurrency primitives such as mutexes (locks) to ensure thread-safe operations.

Example Walkthrough

Let's illustrate the solution approach with a hypothetical situation involving four cars arriving at the intersection. Assume that initially, Road A has a green light and Road B has a red light as per the rules.

1. **Car 1** arrives, with `carId = 1`, `roadId = 1` (Road A), and `direction = 1` (North to South).
   - `carArrived` is called for **Car 1**.
   - The lock is acquired, and it's checked against the traffic light's state.
   - Since **Car 1** is on Road A, which is already green, no traffic light change is necessary.
   - `crossCar` is called for **Car 1**, allowing it to cross.
   - The lock is released.
2. **Car 2** arrives shortly after, with `carId = 2`, `roadId = 1` (Road A), and `direction = 2` (South to North).
   - `carArrived` is called for **Car 2**.
   - **Car 2** successfully acquires the lock as **Car 1** has released it.
   - The traffic light is still green for Road A.
   - `crossCar` is called for **Car 2**, allowing it to cross.
   - The lock is released.
3. **Car 3** arrives next, with `carId = 3`, `roadId = 2` (Road B), and `direction = 3` (West to East).
   - `carArrived` is called for **Car 3**.
   - The lock is acquired.
   - The traffic light check reveals a change is needed since **Car 3** is on Road B, and the current green light is for Road A.
   - The `road` attribute is updated to `2`.
   - `turnGreen()` is called, setting the traffic light green for Road B.
   - `crossCar` is called for **Car 3**, and it crosses the intersection.
   - The lock is released.
4. Finally, **Car 4** approaches, with `carId = 4`, `roadId = 2` (Road B), and `direction = 4` (East to West), while the light is still green on Road B thanks to **Car 3**.
   - The lock is acquired by the `carArrived` method for **Car 4**.
   - Since the light is already green for Road B, no light change is needed.
   - `crossCar` is called for **Car 4** to cross.
   - The lock is finally released.

This walkthrough demonstrates how the lock mechanism ensures that each car's arrival is dealt with one by one, allowing for proper management of the traffic lights and safe crossing of the intersection without deadlocks. The lights change only when necessary, and multiple cars on the same road can take advantage of the light being green without requiring the light to be repeatedly changed.

Python Solution

```python
from threading import Lock
from typing import Callable

class TrafficLight:
    def __init__(self):
        self.lock = Lock()  # lock to control concurrency and avoid race condition
        # state to keep track of which road is green
        # initially, road 1 is set to green
        self.green_road_id = 1

    def carArrived(
        self,
        carId: int,  # identifier of the car
        roadId: int,  # identifier of the road the car is on; can be 1 or 2
        direction: int,  # direction the car is traveling in
        turnGreen: Callable[[], None],  # function to call to turn the light green
        crossCar: Callable[[], None],   # function to call to let the car cross
    ) -> None:
        # acquire lock to ensure exclusive access to the light
        with self.lock:
            # If the car's road ID is different from the current green road,
            # change the traffic light to this car's road
            if self.green_road_id != roadId:
                self.green_road_id = roadId  # update the green road ID
                turnGreen()  # call the method to turn the traffic light green

            crossCar()  # allow the car to cross the intersection
            # lock is released automatically when the 'with' block ends

```

Java Solution

```java
class TrafficLight {
    private int currentRoadId = 1; // Variable to store which road has the green light

    public TrafficLight() {
        // Constructor - no initialization needed as we start with road 1 by default
    }

    // Synchronized method to allow cars to arrive at the intersection without race conditions
    public synchronized void carArrived(
        int carId,        // ID of the car arriving at the intersection
        int roadId,       // ID of the road the car is on. Can be 1 (road A) or 2 (road B)
        int direction,    // Direction of the car, not used in the current context
        Runnable turnGreen,   // Runnable to turn light to green on the car's current road
        Runnable crossCar     // Runnable to allow the car to cross the intersection
    ) {
        // Check if the road that the car wants to use does not have green light
        if (roadId != currentRoadId) {
            turnGreen.run();       // Turn the light green for the current road
            currentRoadId = roadId; // Update the current road ID to the car's road ID
        }

        // Allow the car to cross the intersection
        crossCar.run();
    }
}

```

C++ Solution

```cpp
#include <mutex>
#include <functional>

class TrafficLight {
private:
    int currentRoadId = 1; // Variable to store which road has the green light
    std::mutex mtx; // Mutex to protect shared resources and prevent race conditions

public:
    // Constructor - no initialization needed as we start with road 1 by default
    TrafficLight() {}

    // Synchronized method to allow cars to arrive at the intersection without race conditions
    void carArrived(
        int carId,             // ID of the car arriving at the intersection
        int roadId,            // ID of the road the car is on. Can be 1 (road A) or 2 (road B)
        int direction,         // Direction of the car, not used in the current context
        std::function<void()> turnGreen,    // Function to turn light to green on the car's road
        std::function<void()> crossCar      // Function to allow the car to cross the intersection
    ) {
        {
            // Locking the mutex to ensure exclusive access to the shared variable currentRoadId
            std::lock_guard<std::mutex> lock(mtx);

            // Check if the road that the car wants to use does not have green light
            if (roadId != currentRoadId) {
                turnGreen();             // Turn the light green for the current road
                currentRoadId = roadId;  // Update the current road ID to the car's road ID
            }
        } // Mutex is released automatically when lock goes out of scope

        // Allow the car to cross the intersection
        crossCar();
    }
};

```

Typescript Solution

```typescript
// Variable to store the ID of the current road with a green light
let currentRoadId = 1;

// Function to simulate the car arriving at the intersection
// Ensures synchronization to prevent race conditions
function carArrived(
  carId: number,          // ID of the car arriving at the intersection
  roadId: number,         // ID of the road the car is on. Can be 1 (for road A) or 2 (for road B)
  direction: number,      // Direction of the car, not currently used
  turnGreen: () => void,  // Function to invoke to turn the traffic light green for the current road
  crossCar: () => void    // Function to invoke to allow the car to cross the intersection
): void {
  // The synchronization mechanism provided by the `synchronized` keyword in Java is
  // not directly available in TypeScript/JavaScript. We would typically need to manage
  // concurrency with Promise chains, Async/Await, or other synchronization primitives.

  // Simulate synchronization lock to ensure one car processes at a time
  // (This is conceptual, actual implementation would require additional code.)
  synchronized(() => {
    if (roadId !== currentRoadId) {
      turnGreen(); // If the car's road ID differs from the current, turn the light to green
      currentRoadId = roadId; // Update the current road ID
    }

    crossCar(); // Once the traffic light is green, the car can cross the intersection
  });
}

// Synchronization helper function (Note: This is a placeholder as JavaScript/TypeScript
// does not directly support the synchronized concept out-of-the-box)
function synchronized(action: () => void): void {
  // Enter synchronization lock
  action();
  // Exit synchronization lock
}

```

Time and Space Complexity

Time Complexity

The time complexity of the `carArrived` method is `O(1)`. This is because there are no loops or recursive calls that depend on the size of the input. Each function call to `turnGreen()` and `crossCar()` is considered a constant time operation. Acquiring and releasing a lock is also a constant time operation.

Space Complexity

The space complexity of the `TrafficLight` class is `O(1)`. It uses a fixed amount of space: one lock and one integer variable, regardless of the number of times the `carArrived` method is called or the number of car objects interacting with the system.