1226. The Dining Philosophers

Medium Concurrency Leetcode Link

Problem Description

The Dining Philosophers problem is a classic synchronization problem involving five philosophers who do two things: think and eat. However, to eat, each philosopher needs to have both the fork to their left and the fork to their right. Since each philosopher shares a fork with their neighbor, they must coordinate fork usage to prevent deadlock, where everyone holds one fork and is waiting for the other, or starvation, where a philosopher is perpetually denied the opportunity to eat.

The philosophers are sitting at a round table, which creates a circular dependency in resource (fork) acquisition. The challenge is to implement a system that allows philosophers to pick up and put down forks in such a way that they can all continue eating and thinking indefinitely without any of them starving due to being unable to acquire the necessary forks.

The problem is to design a `wantsToEat` function conforming to the given method signature, which allows a philosopher to:

- Pick up the left fork.
- Pick up the right fork.
- Eat spaghetti.
- Put down the left fork.
- Put down the right fork.

This function will be called multiple times in parallel, simulating each philosopher's attempt to eat.

Intuition

To tackle this problem, the key is to implement a protocol that ensures philosophers can alternately think and eat without causing a deadlock or starvation. By using mutual exclusion (mutex), we can avoid the situation where multiple philosophers can hold the same fork simultaneously, leading to a deadlock. The `scoped_lock` used in the solution automatically acquires locks for both the philosopher's left and right forks upon entering the `wantsToEat` method and releases them when the method completes. This ensures that the philosopher holds both forks while they eat and that the resources are properly released afterward.

In the given C++ solution, a `std::scoped_lock` is introduced for the `mutex` objects that represent the forks. A `scoped_lock` is a lock that manages multiple mutexes while maintaining a simplified lock interface. The implementation uses the scope of the lock to handle the acquisition and release of the mutexes. This lock ensures that both or neither of the forks are acquired, preventing deadlock since a philosopher will only begin eating when both forks are available.

To prevent neighboring philosophers from picking up the same fork at the same time, an array of five mutexes is used (`mutexes_`). Each index in the `mutexes_` array corresponds to a philosopher and their right fork. The `philosopher` parameter determines which mutexes to lock. Since the philosophers are in a circle, we need to handle the case where the last philosopher (index 4) must lock the mutex at index 0 and index 4.

The solution elegantly ensures that each `wantsToEat` function call locks and releases the correct pair of forks atomically. The `eat` function will only be called when the philosopher has successfully picked up both the left and the right forks. The `std::scoped_lock` will automatically release the locks when it goes out of scope, which occurs when the philosopher finishes eating and the `wantsToEat` function exits. This ensures the forks are released in all scenarios, preventing a situation where a fork is left locked indefinitely, which would lead to starvation of a philosopher.

Solution Approach

The implementation of the solution primarily revolves around the concept of mutual exclusion, ensuring that only one thread (philosopher) can access a particular resource (fork) at any given time. The algorithm relies on the following elements:

- **Mutexes**: Mutexes are used to represent the forks. A mutex is a synchronization primitive that can be used to protect shared data from being simultaneously accessed by multiple threads.
- **Scoped Locks**: `std::scoped_lock` is a C++17 feature that simplifies the management of locking multiple mutexes. It locks the provided mutexes at the start of a block and automatically unlocks them when the block is exited. This is particularly useful to avoid common problems with locking such as deadlocks.
- **Vector of Mutexes**: A `vector` with five mutexes represents the five forks on the table, corresponding to the five philosophers. Each index in this vector represents a philosopher's right fork (and the left fork of the philosopher to their right).
- **Locking Logic**: For any given philosopher attempting to eat, they need to lock the mutexes corresponding to the forks on their left and right. To do this, we pass the current philosopher's mutex and the next philosopher's mutex (wrapping around using the conditional operator for the last philosopher) to the `scoped_lock` constructor. This ensures that both forks are locked simultaneously in a deadlock-free manner because `scoped_lock` uses a deadlock avoidance algorithm when acquiring multiple mutexes.

Here is a step-by-step breakdown of how the `wantsToEat` function works:

1. When a philosopher wants to eat, they call the `wantsToEat` function with their ID.
2. The function creates a `scoped_lock`, locking the mutexes corresponding to the philosopher's left and right forks. For the philosopher with ID 4, the locked mutexes are at index 4 and 0, as they will wrap around the table.
3. Once the locks are acquired, the philosopher can pick up both forks by calling the `pickLeftFork` and `pickRightFork` actions.
4. With both forks in hand, the philosopher then calls the `eat` action to simulate eating.
5. After eating, the philosopher puts down the forks by invoking `putLeftFork` and `putRightFork` actions.
6. The `scoped_lock` automatically releases the mutexes when the `wantsToEat` function exits the scope (at the end of the function), which allows other philosophers to then acquire these forks.

With this design, each philosopher can eat without causing deadlock or starvation, while also ensuring the concurrent access to the shared forks is properly managed.

Example Walkthrough

Let's consider a small example to illustrate the solution approach with just two philosophers for simplicity, even though the Dining Philosophers problem typically involves five. Imagine a table with two philosophers and two forks, one for each philosopher. When a philosopher wants to eat, they must follow the steps outlined in the solution approach.

1. Philosopher 0 decides they want to eat and calls `wantsToEat(0)`.
2. The `wantsToEat` function for Philosopher 0 attempts to create a `scoped_lock` for the mutexes at indices 0 (Philosopher 0's right fork) and 1 (using modulus arithmetic, it's the left fork which is also Philosopher 1's right fork).
3. Because we're using the `scoped_lock`, it attempts to lock both mutexes at the same time. If successful, Philosopher 0 now has both forks, avoiding deadlock.
4. Philosopher 0 proceeds with the `pickLeftFork` and `pickRightFork` actions, simulating the action of picking up the forks.
5. After picking up the forks, Philosopher 0 calls the `eat` action to simulate eating the spaghetti.
6. Once done eating, in most implementations, Philosopher 0 would set down the forks with `putLeftFork` and `putRightFork`. However, in our automatic scope-based system, the forks are implicitly put down when the `scoped_lock` is released.
7. The `scoped_lock` reaches the end of its scope when the `wantsToEat` function for Philosopher 0 finishes. At this point, the lock is automatically released, and the forks are available again.

Now, let's say Philosopher 1 concurrently calls `wantsToEat(1)` while Philosopher 0 has not yet finished eating.

1. Philosopher 1 also tries to create a `scoped_lock` for their forks, which correspond to mutexes at indices 1 and 0 (indicating Philosopher 1's right and Philosopher 0's left fork, respectively).
2. However, since Philosopher 0 is already holding both forks, the `scoped_lock` for Philosopher 1 cannot immediately acquire the mutex at index 1; thus, Philosopher 1 must wait.
3. Once Philosopher 0's `scoped_lock` goes out of scope, the mutexes are released, and Philosopher 1 can now acquire the locks on both forks.
4. Philosopher 1 can now pick up both forks, eat, and once they're done, the `scoped_lock` will ensure that the forks are released.

This system ensures that no deadlock occurs because the forks are always picked up and put down in a controlled manner, and no philosopher can start eating without holding both forks. Moreover, starvation is avoided since each philosopher will get a chance to acquire both forks and eat as the `scoped_lock` ensures mutexes are eventually released for others to use.

Python Solution

```python
from threading import Lock

class DiningPhilosophers:
    """
    Class representing the Dining Philosophers problem.
    """

    def __init__(self):
        """
        Initialize the DiningPhilosophers class with necessary locks for the forks.
        """
        # Initialize a list of 5 Locks, representing the 5 forks.
        self.forks = [Lock() for _ in range(5)]

    def wants_to_eat(self, philosopher, pick_left_fork, pick_right_fork, eat, put_left_fork, put_right_fork):
        """
        Method called when a philosopher wants to eat.

        It ensures that no two philosophers can hold the same fork at the same time.
        For the fifth philosopher, the right fork is considered to be the fork at index 0.

        :param philosopher: The index of the philosopher who wants to eat [0-4].
        :param pick_left_fork: Function to simulate picking up the left fork.
        :param pick_right_fork: Function to simulate picking up the right fork.
        :param eat: Function to simulate eating.
        :param put_left_fork: Function to simulate putting down the left fork.
        :param put_right_fork: Function to simulate putting down the right fork.
        """

        # Philosopher 4 (indexing from 0) has a different right fork compared to others.
        right_fork_index = 0 if philosopher == 4 else philosopher + 1

        # Acquire both forks using context management to ensure exception safety.
        with self.forks[philosopher], self.forks[right_fork_index]:
            # Simulate picking up left and right forks, eating, and putting down the forks.
            pick_left_fork()
            pick_right_fork()
            eat()
            put_left_fork()
            put_right_fork()

```

Java Solution

```java
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/**
 * Class representing the Dining Philosophers problem.
 */
public class DiningPhilosophers {

    // Alias for the action functions using Runnable, since Java does not have a std::function equivalent.
    // Runnable is chosen as it represents an action that takes no arguments and returns no result.
    public interface Action extends Runnable { }

    // Array of ReentrantLocks representing the forks.
    private final Lock[] forks = new ReentrantLock[5];

    // Constructor initializes each lock representing a fork.
    public DiningPhilosophers() {
        for (int i = 0; i < forks.length; i++) {
            forks[i] = new ReentrantLock();
        }
    }

    /**
     * Method called when a philosopher wants to eat.
     *
     * @param philosopher The index of the philosopher [0-4].
     * @param pickLeftFork Runnable action to pick up the left fork.
     * @param pickRightFork Runnable action to pick up the right fork.
     * @param eat Runnable action to perform the eating action.
     * @param putLeftFork Runnable action to put down the left fork.
     * @param putRightFork Runnable action to put down the right fork.
     */
    public void wantsToEat(int philosopher,
                           Action pickLeftFork,
                           Action pickRightFork,
                           Action eat,
                           Action putLeftFork,
                           Action putRightFork) throws InterruptedException {
        // The id of the left and right fork, taking into account the special case of the last philosopher
        int leftFork = philosopher;
        int rightFork = (philosopher + 1) % 5;

        // Lock the forks to ensure that no two philosophers can hold the same fork at the same time.
        // Locking is arranged to prevent deadlock.
        forks[leftFork].lock();
        forks[rightFork].lock();

        try {
            // Perform actions with the forks and eating in a critical section.
            pickLeftFork.run(); // Pick up left fork
            pickRightFork.run(); // Pick up right fork
            eat.run(); // Eat
            putLeftFork.run(); // Put down left fork
            putRightFork.run(); // Put down right fork
        } finally {
            // Ensure that forks are always released to avoid deadlock.
            forks[leftFork].unlock();
            forks[rightFork].unlock();
        }
    }
}

```

C++ Solution

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

// Class representing the Dining Philosophers problem
class DiningPhilosophers {
public:
    // Alias for the action functions
    using Action = std::function<void()>;

    // Method called when a philosopher wants to eat
    void wantsToEat(int philosopher,
                    Action pickLeftFork,
                    Action pickRightFork,
                    Action eat,
                    Action putLeftFork,
                    Action putRightFork) {
        // Ensure no two philosophers hold the same fork at the same time
        // For the fifth philosopher, we consider the fork to the right as the fork at position 0
        std::scoped_lock lock(forks_[philosopher], forks_[philosopher == 4 ? 0 : philosopher + 1]);

        // Pick up left fork
        pickLeftFork();

        // Pick up right fork
        pickRightFork();

        // Eat
        eat();

        // Put down left fork
        putLeftFork();

        // Put down right fork
        putRightFork();
    }

private:
    // Mutexes representing the forks
    std::vector<std::mutex> forks_ = std::vector<std::mutex>(5);
};

```

Typescript Solution

```typescript
// TypeScript doesn't have mutexes, but we can simulate a lock using Promises and async/await.
// Here's a simple lock implementation in TypeScript for educational purposes.
class Lock {
  private _isLocked: boolean = false;
  private _waitingResolvers: Array<() => void> = [];

  async acquire(): Promise<void> {
    while (this._isLocked) {
      // Wait until the lock becomes free
      await new Promise(resolve => this._waitingResolvers.push(resolve));
    }
    this._isLocked = true;
  }

  release(): void {
    if (!this._isLocked) {
      throw new Error('Lock is already released');
    }
    this._isLocked = false;
    const resolve = this._waitingResolvers.shift();
    if (resolve) {
      resolve();
    }
  }
}

// Alias for the action functions
type Action = () => void;

// Array representing the locks for the forks
const forks = Array.from({ length: 5 }, () => new Lock());

// Method called when a philosopher wants to eat
async function wantsToEat(
  philosopher: number,
  pickLeftFork: Action,
  pickRightFork: Action,
  eat: Action,
  putLeftFork: Action,
  putRightFork: Action
): Promise<void> {
  // Calculate fork indices, ensuring the right fork index wraps for the fifth philosopher
  const leftForkIndex = philosopher;
  const rightForkIndex = (philosopher + 1) % 5;

  // Acquire locks for the two forks asynchronously to simulate locking
  await Promise.all([forks[leftForkIndex].acquire(), forks[rightForkIndex].acquire()]);

  try {
    // Once both forks are acquired, the philosopher can follow the eating procedure
    pickLeftFork();   // Pick up left fork
    pickRightFork();  // Pick up right fork
    eat();            // Eat
    putLeftFork();    // Put down left fork
    putRightFork();   // Put down right fork
  } finally {
    // Always release the locks in the end, regardless of whether the actions succeeded or not
    forks[leftForkIndex].release();
    forks[rightForkIndex].release();
  }
}

// Sample usage:
// A philosopher would call the `wantsToEat` function with appropriate actions
// In practice, proper synchronization primitives would be required to prevent race conditions

```

Time and Space Complexity

Time Complexity

The time complexity of the `wantsToEat` method in the `DiningPhilosophers` class isn't determined by a simple algorithmic analysis, because it's primarily dependent on the concurrency and synchronization primitives used (mutexes and locks). Each philosopher (in this case, a thread) attempts to pick up two forks (acquiring two mutexes) before eating. The `std::scoped_lock` is used to acquire both mutexes atomically, which prevents deadlock. The actual time complexity for each philosopher to eat depends on the order and time at which each thread is scheduled as well as contention for the mutexes.

However, assuming there is no contention and each operation (`pickLeftFork`, `pickRightFork`, `eat`, `putLeftFork`, and `putRightFork`) has a constant time complexity `O(1)`, the `wantsToEat` function would have a time complexity of `O(1)` for each call in an ideal scenario. Acquiring and releasing a mutex can also be considered to have a time complexity of `O(1)`.

Space Complexity

The space complexity of the `DiningPhilosophers` class is `O(N)` where `N` is the number of philosophers (which is 5 in this case). This is due to the `vector<mutex> mutexes_` which contains a mutex for each philosopher's left fork. There are no additional data structures that scale with the number of operations or philosophers, so the space complexity is proportional to the number of philosophers.

In this code, since `N` is fixed at 5, you could argue that the space complexity can be considered as `O(1)` since it doesn't scale with input size and is fixed.