1115. Print FooBar Alternately

Medium Concurrency Leetcode Link

Problem Description

In this concurrency problem, we have a code snippet defining a class `FooBar` with two methods: `foo()` and `bar()`. These methods print the strings "foo" and "bar", respectively, in a loop up to `n` times. The challenge is to ensure that when these methods are called by two separate threads, the output is "foobar" repeated `n` times, with "foo" and "bar" alternating properly.

To achieve this, we need to implement a mechanism that enforces the order of execution, such that the `foo()` method must print "foo" before the `bar()` method prints "bar," and this pattern is repeated for the entire loop.

Intuition

Concurrency problems often require synchronization techniques to ensure that multiple threads can work together without conflicts or race conditions. In this case, we need to make sure that "foo" is always printed before "bar."

The intuition behind the solution is to use semaphores—a classic synchronization primitive—to coordinate the actions of the threads. In the context of this problem, we use two semaphores: one that controls the printing of "foo" (`self.f`) and one that controls the printing of "bar" (`self.b`).

The `self.f` semaphore is initially set to 1, allowing the `foo()` method to print immediately. After printing, it releases the `self.b` semaphore, which is initially set to 0, thus preventing `bar()` from printing until `foo()` is printed first. Once `self.b` is released by `foo()`, the `bar()` method can print "bar" and then release the `self.f` semaphore to allow the next "foo" to be printed. This alternating process continues until the loop completes `n` times.

By carefully managing the states of the semaphores and ensuring that each method changes the semaphore state only after printing, we can achieve the desired ordering of prints, resulting in the correct output "foobar" repeated `n` times.

Solution Approach

The solution utilizes the `Semaphore` class from Python's `threading` module as the primary synchronization mechanism, allowing us to enforce the strict alternation between `foo` and `bar`.

Here's a step-by-step explanation of the code implementation:

1. **Initialization:** The `FooBar` class is initialized with an integer `n`, which represents the number of times "foobar" should be printed. Two `Semaphore` objects are created: `self.f` for "foo" with an initial value of 1, and `self.b` for "bar" with an initial value of 0. The initial values are critical: `self.f` is set to 1 to allow `foo()` to proceed immediately, and `self.b` is set to 0 to block `bar()` until `foo()` signals it by releasing `self.b`.

2. **The `foo` Method:**

   - It contains a loop that runs `n` times.
   - Each iteration begins with `self.f.acquire()`, which blocks if the semaphore's value is 0. Since `self.f` is initialized to 1, `foo()` can start immediately on the first iteration.
   - The `printFoo()` function is executed, printing "foo".
   - After printing "foo", the `self.b.release()` is called. This increments the count of the `self.b` semaphore, signaling the `bar()` method (if it is waiting) that it can proceed to print "bar".

3. **The `bar` Method:**

   - It's also a loop running `n` times.
   - Each iteration starts by calling `self.b.acquire()`, which waits until the `self.f` semaphore is released by a previous `foo()` call, ensuring that "foo" has been printed before "bar" can proceed.
   - Once the semaphore is acquired, `printBar()` is executed to print "bar".
   - After printing "bar", it invokes `self.f.release()` to increment the semaphore count for `foo`, allowing the next iteration of `foo()` to print again, hence ensuring the sequence starts with "foo".

This alternating semaphore pattern locks each method in a waiting state until the other method signals that it has finished its task by releasing the semaphore. Since `acquire()` decreases the semaphore's value by 1 and `release()` increases it by 1, this careful incrementing and decrementing of semaphore values guarantees that the print order is preserved and that the strings "foo" and "bar" are printed in the correct sequence to form "foobar" without getting mixed up or overwritten.

Example Walkthrough

Let's walk through a simple example with `n = 2`. We want our output to be "foobarfoobar" with "foo" always preceding "bar".

Here's how the synchronization using semaphores will facilitate this:

- Initialization:

  - `FooBar` object is created with `n = 2`.
  - Semaphore `self.f` is set to 1 (unlocked), allowing `foo()` to be called immediately.
  - Semaphore `self.b` is set to 0 (locked), preventing `bar()` from being called until `foo()` is done.

- First Iteration:

  1. `foo()` method is called by Thread 1.
     - `self.f.acquire()` is called, which succeeds immediately because `self.f` is 1 (unlocked).
     - `printFoo()` is executed, so "foo" is printed.
     - `self.b.release()` is called, incrementing `self.b` to 1, which unlocks `bar()`.
  2. `bar()` method is called by Thread 2.
     - `self.b.acquire()` is called, which succeeds because `self.b` was released by `foo()`.
     - `printBar()` is executed, printing "bar" after "foo".
     - `self.f.release()` is called, setting `self.f` back to 1 (unlocked) and allowing the next `foo()` to proceed.

- Second Iteration:

  1. Again, `foo()` method is called by Thread 1.
     - This time, since `self.f` was released by the previous `bar()` call, `self.f.acquire()` succeeds again.
     - `printFoo()` is executed, and another "foo" is printed.
     - `self.b.release()` is called, incrementing `self.b` and allowing `bar()` to be called.
  2. `bar()` method is again called by Thread 2.
     - With `self.b` released, `self.b.acquire()` allows the thread to proceed.
     - `printBar()` prints "bar", following the "foo" printed by the last `foo()` call.
     - Finally, `self.f.release()` is called, although in this case, it's unnecessary because we've reached our loop condition (`n` times) and no further `foo()` calls are needed.

By the end of the two iterations, we've successfully printed "foobarfoobar". Each `foo()` preceded a `bar()` thanks to our semaphore controls, and at no point could `bar()` leapfrog ahead of `foo()`. The semaphores effectively serialized access to the printing functions, ensuring the correct order despite the concurrent execution of threads.

Python Solution

```python
from threading import Semaphore
from typing import Callable


class FooBar:
    def __init__(self, n: int):
        self.n = n  # Number of times "foo" and "bar" are to be printed.
        # Semaphore for "foo" is initially unlocked.
        self.sem_foo = Semaphore(1)
        # Semaphore for "bar" is initially locked.
        self.sem_bar = Semaphore(0)

    def foo(self, print_foo: Callable[[], None]) -> None:
        """Print "foo" n times, ensuring it alternates with "bar"."""
        for _ in range(self.n):
            self.sem_foo.acquire()  # Wait for semaphore to be unlocked.
            print_foo()  # Provided print function for "foo".
            self.sem_bar.release()  # Unlock semaphore for "bar".

    def bar(self, print_bar: Callable[[], None]) -> None:
        """Print "bar" n times, ensuring it alternates with "foo"."""
        for _ in range(self.n):
            self.sem_bar.acquire()  # Wait for semaphore to be unlocked.
            print_bar()  # Provided print function for "bar".
            self.sem_foo.release()  # Unlock semaphore for "foo".

```

Java Solution

```java
class FooBar {
    private final int loopCount; // The number of times "foo" and "bar" should be printed.
    private final Semaphore fooSemaphore = new Semaphore(1); // A semaphore for "foo", allowing "foo" to print first.
    private final Semaphore barSemaphore = new Semaphore(0); // A semaphore for "bar", initially locked until "foo" is printed.

    public FooBar(int n) {
        this.loopCount = n;
    }

    // The method for printing "foo"
    public void foo(Runnable printFoo) throws InterruptedException {
        for (int i = 0; i < loopCount; i++) {
            fooSemaphore.acquire(); // Acquire a permit before printing "foo", ensuring "foo" has the turn to print
            printFoo.run();         // Output "foo"
            barSemaphore.release(); // Release a permit for "bar" after "foo" is printed, allowing "bar" to print next
        }
    }

    // The method for printing "bar"
    public void bar(Runnable printBar) throws InterruptedException {
        for (int i = 0; i < loopCount; i++) {
            barSemaphore.acquire(); // Acquire a permit before printing "bar", ensuring "bar" has the turn to print
            printBar.run();         // Output "bar"
            fooSemaphore.release(); // Release a permit for "foo" after "bar" is printed, allowing "foo" to print next
        }
    }
}

```

C++ Solution

```cpp
#include <semaphore.h>
#include <functional>

class FooBar {
private:
    int n_;                    // The number of times to print "foobar"
    sem_t sem_foo_, sem_bar_; // Semaphores used to coordinate the printing order

public:
    // Constructor that initializes the semaphores and count
    FooBar(int n) : n_(n) {
        // Initialize sem_foo_ with a count of 1 to allow "foo" to print first
        sem_init(&sem_foo_, 0, 1);
        // Initialize sem_bar_ with a count of 0 to block "bar" until "foo" is printed
        sem_init(&sem_bar_, 0, 0);
    }

    // Deconstructor that destroys the semaphores
    ~FooBar() {
        sem_destroy(&sem_foo_);
        sem_destroy(&sem_bar_);
    }

    // Method for printing "foo"
    void foo(std::function<void()> printFoo) {
        for (int i = 0; i < n_; ++i) {
            // Wait on sem_foo_ to ensure "foo" is printed first
            sem_wait(&sem_foo_);
            // printFoo() calls the provided lambda function to output "foo"
            printFoo();
            // Post (increment) sem_bar_ to allow "bar" to be printed next
            sem_post(&sem_bar_);
        }
    }

    // Method for printing "bar"
    void bar(std::function<void()> printBar) {
        for (int i = 0; i < n_; ++i) {
            // Wait on sem_bar_ to ensure "bar" is printed after "foo"
            sem_wait(&sem_bar_);
            // printBar() calls the provided lambda function to output "bar"
            printBar();
            // Post (increment) sem_foo_ to allow the next "foo" to be printed
            sem_post(&sem_foo_);
        }
    }
};

```

Typescript Solution

```typescript
// The number of times to print "foo" and "bar"
let n: number;
// Promises and callbacks for signaling
let canPrintFoo: (() => void) | null = null;
let canPrintBar: (() => void) | null = null;
// Deferred promise resolvers
let fooPromiseResolver: (() => void) | null = null;
let barPromiseResolver: (() => void) | null = null;

/**
 * Initializes the synchronization primitives.
 * @param {number} count - The number of iterations to run the sequence.
 */
function initFooBar(count: number): void {
    n = count;
    // Start with the ability to print "foo"
    canPrintFoo = (() => {
        fooPromiseResolver = null;
        if (canPrintBar) canPrintBar();
    });
    // Prevent "bar" from printing until "foo" is printed
    canPrintBar = null;
}

/**
 * Prints "foo" to the console or another output.
 * @param {() => void} printFoo - A callback function that prints "foo".
 */
async function foo(printFoo: () => void): Promise<void> {
    for (let i = 0; i < n; i++) {
        await new Promise<void>((resolve) => {
            fooPromiseResolver = resolve;
            if (canPrintFoo) canPrintFoo();
        });
        // The provided callback prints "foo"
        printFoo();
        // Allow "bar" to be printed
        canPrintBar = (() => {
            barPromiseResolver = null;
            if (fooPromiseResolver) fooPromiseResolver();
        });
        // Block until 'bar' is printed
        canPrintFoo = null;
    }
}

/**
 * Prints "bar" to the console or another output.
 * @param {() => void} printBar - A callback function that prints "bar".
 */
async function bar(printBar: () => void): Promise<void> {
    for (let i = 0; i < n; i++) {
        await new Promise<void>((resolve) => {
            barPromiseResolver = resolve;
            if (canPrintBar) canPrintBar();
        });
        // The provided callback prints "bar"
        printBar();
        // Allow "foo" to be printed again
        canPrintFoo = (() => {
            fooPromiseResolver = null;
            if (barPromiseResolver) barPromiseResolver();
        });
        // Block until 'foo' is printed again
        canPrintBar = null;
    }
}

// Example usage:
initFooBar(3); // Initialize printing "foo" and "bar" three times each
foo(() => console.log("foo"));
bar(() => console.log("bar"));

```

Time and Space Complexity

Time Complexity

The time complexity of the `FooBar` class methods `foo` and `bar` are both `O(n)`. Each method contains a loop that iterates `n` times, where `n` is the input that represents the number of times the "foo" and "bar" functions should be called, respectively. The methods invoke `acquire` and `release` on semaphores, but the acquire/release operations are constant-time `O(1)` operations, assuming that there is no contention (which should not happen here given the strict alternation). The `printFoo` and `printBar` functions are also called `n` times each, and if we consider these functions to have `O(1)` time complexity, which is a reasonable assumption for a simple print operation, then this does not change the overall time complexity of the `foo` and `bar` methods.

Space Complexity

The space complexity of the `FooBar` class is `O(1)` since the space required does not grow with `n`. The class maintains fixed resources: two semaphores and one integer variable. No additional space is allocated that would scale with the input `n`, meaning that the memory usage is constant irrespective of the size of `n`.