1116. Print Zero Even Odd

Medium Concurrency

Leetcode Link

Problem Description

In this problem, we are working with a multi-threaded setup using a class `ZeroEvenOdd`. The class comes with three methods: `zero`, `even`, and `odd`. Three different threads will call these methods. The task is to print a sequence of numbers following a pattern:

- The sequence starts with 0
- An odd number follows the 0
- Then zero again
- Then an even number ... and so on.

The sequence should follow the `01020304050...` pattern, and the total length of the sequence must be `2n`, where `n` is an integer provided when creating an instance of the `ZeroEvenOdd` class.

For example, if `n = 5`, the sequence to print would be "0102030405".

The challenge is to ensure that:

- `zero()` function only outputs 0's.
- `even()` function only outputs even numbers.
- `odd()` function only outputs odd numbers. These functions must be called in the right sequence by different threads to maintain the mentioned sequence pattern.

Intuition

To solve this problem, we need to coordinate between threads to print the sequence in order. If the threads run without any coordination, there's no guarantee the sequence will maintain the required order because threading is non-deterministic in nature.

To ensure the correct order, we use `Semaphore` objects. Semaphores are synchronization primitives that can be incremented or decremented, and threads can wait for a semaphore's value to be positive before proceeding with execution.

The key idea behind this solution is to use a semaphore `z` to control the printing of zero and two other semaphores, `e` for even and `o` for odd numbers. Initially, since we always start with 0, semaphore `z` is given an initial count of 1, and the others are set to 0, effectively blocking them.

As we proceed, the call to `zero` printing function will:

1. Acquire `z`.
2. Print 0.
3. Release `o` or `e` depending on whether the next number should be odd or even.

The `odd` and `even` functions will:

1. Wait to acquire `o` or `e` (respectively).
2. Once acquired, print the next odd or even number.
3. Release `z` to allow the next zero to be printed.

This mechanism allows us to alternate between printing 0 and the next odd or even number, ensuring that the threads print numbers in the correct sequence.

Solution Approach

The solution to the problem utilizes three semaphores as a means of synchronization between threads:

1. `self.z` (semaphore for zero): Initialized with a count of 1 because we always start the sequence with 0.
2. `self.e` (semaphore for even numbers): Initialized with a count of 0, will be used to control the `even()` function.
3. `self.o` (semaphore for odd numbers): Initialized with a count of 0, will be used to control the `odd()` function.

Each function (`zero`, `even`, `odd`) runs in its thread and follows these steps:

`zero` function

The `zero` function is responsible for printing zeros. It follows this loop for `i` from 0 up to `n-1`:

1. Wait until the `self.z` semaphore is greater than 0, and then acquire it (decrement its value).
2. Invoke `printNumber(0)` to print zero to the console.
3. Determine whether the next number should be odd or even based on `i`'s value. If `i % 2` equals 0, then the next number is odd; otherwise, it's even.
4. Release the semaphore `self.o` if the next number is odd or `self.e` if it's even. This will unblock the `odd` or `even` functions, respectively.

`even` function

The `even` function is responsible for printing even numbers only. The loop iterates through even numbers starting from 2 until `n`, incrementing by 2 in each iteration:

1. Wait until the `self.e` semaphore is greater than 0, and then acquire it (decrement its value).
2. Invoke `printNumber(i)` where `i` is the current even number in the loop to print the number to the console.
3. Release the `self.z` semaphore, allowing the `zero` function to print the next 0.

`odd` function

Similarly, the `odd` function prints odd numbers. The loop iterates through odd numbers starting from 1 until `n`:

1. Wait until the `self.o` semaphore is greater than 0, and then acquire it (decrement its value).
2. Invoke `printNumber(i)` where `i` is the current odd number in the loop to print the number to the console.
3. Release the `self.z` semaphore, allowing the `zero` function to print the next 0.

By utilizing semaphores to control the sequence of method calls across different threads, the threads are coordinated in such a way that `printNumber` prints the numbers in the sequence as if it were in a single-threaded environment.

Example Walkthrough

Let's take `n = 2` as an example to illustrate how the solution approach works in practice. We want the output to be "0102." The `ZeroEvenOdd` class will utilize semaphores to synchronize the print calls across the three threads corresponding to the `zero`, `even`, and `odd` methods.

1. The `zero` function has a semaphore `self.z` initialized to 1, so it can proceed immediately.
2. Thread running `zero` function acquires semaphore `self.z` (decrements `self.z` to 0) and prints "0".
3. Since the next number is 1 (odd), the thread running `zero` releases semaphore `self.o` (increments `self.o` to 1).
4. Thread running `odd` function was waiting for `self.o` to be greater than 0, so it can proceed now.
5. The `odd` thread running the `odd` function acquires `self.o` (decrements `self.o` to 0) and prints "1".
6. The `odd` thread then releases semaphore `self.z` (increments `self.z` to 1), enabling the `zero` function to print again.
7. The `zero` function's thread acquires `self.z` again (decrements `self.z` to 0) and prints the second "0".
8. Since the next number is 2 (even), the thread running `zero` now releases semaphore `self.e` (increments `self.e` to 1).
9. The thread running `even` function was waiting for `self.e` to be greater than 0, so it can now proceed.
10. The `even` thread acquires semaphore `self.e` (decrements `self.e` to 0) and prints "2".
11. Finally, the `even` thread releases semaphore `self.z` (increments `self.z` to 1), but since we have printed all numbers up to `n`, the sequence is complete.

Throughout this process, the semaphores ensure that each number is printed in the correct order, and neither `odd` nor `even` threads print out of turn.

Python Solution

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

class ZeroEvenOdd:
    def __init__(self, n: int):
        self.n = n
        self.zero_semaphore = Semaphore(1)  # A semaphore for allowing 'zero' to be printed
        self.even_semaphore = Semaphore(0)  # A semaphore for allowing 'even' numbers to be printed
        self.odd_semaphore = Semaphore(0)   # A semaphore for allowing 'odd' numbers to be printed

    def zero(self, print_number: Callable[[int], None]) -> None:
        """
        Prints 'zero' n times and alternates the control between printing odd and even numbers
        :param print_number: The function to call to print the number
        """
        for i in range(self.n):
            self.zero_semaphore.acquire()  # Wait for the zero semaphore to be released
            print_number(0)  # Print zero

            # Determine if the next number after zero should be odd or even
            if i % 2 == 0:
                self.odd_semaphore.release()  # If 'i' is even, the next number should be odd
            else:
                self.even_semaphore.release()  # If 'i' is odd, the next number should be even

    def even(self, print_number: Callable[[int], None]) -> None:
        """
        Prints the even numbers starting from 2 to n
        :param print_number: The function to call to print the number
        """
        for i in range(2, self.n + 1, 2):
            self.even_semaphore.acquire()  # Wait for the even semaphore to be released
            print_number(i)  # Print the even number
            self.zero_semaphore.release()  # Allow the zero method to print zero again

    def odd(self, print_number: Callable[[int], None]) -> None:
        """
        Prints the odd numbers starting from 1 to n
        :param print_number: The function to call to print the number
        """
        for i in range(1, self.n + 1, 2):
            self.odd_semaphore.acquire()  # Wait for the odd semaphore to be released
            print_number(i)  # Print the odd number
            self.zero_semaphore.release()  # Allow the zero method to print zero again
```

Java Solution

```java
import java.util.concurrent.Semaphore;
import java.util.function.IntConsumer;

class ZeroEvenOdd {
  private int n;
  // Semaphores serve as locks to control the print order.
  private Semaphore semaphoreZero = new Semaphore(1);
  private Semaphore semaphoreEven = new Semaphore(0);
  private Semaphore semaphoreOdd = new Semaphore(0);

  public ZeroEvenOdd(int n) {
    this.n = n;
  }

  /**
   * Prints the number 0.
   * @param printNumber The consumer interface to output a number.
   * @throws InterruptedException if the current thread is interrupted while waiting.
   */
  public void zero(IntConsumer printNumber) throws InterruptedException {
    for (int i = 0; i < n; ++i) {
      // Acquire the semaphore before printing 0.
      semaphoreZero.acquire();
      printNumber.accept(0);
      // Determine whether to release the odd or even semaphore based on the current iteration.
      if (i % 2 == 0) {
        semaphoreEven.release();
      } else {
        semaphoreOdd.release();
      }
    }
  }

  /**
   * Prints even numbers.
   * @param printNumber The consumer interface to output a number.
   * @throws InterruptedException if the current thread is interrupted while waiting.
   */
  public void even(IntConsumer printNumber) throws InterruptedException {
    for (int i = 2; i <= n; i += 2) {
      // Wait for the even semaphore to be released.
      semaphoreEven.acquire();
      printNumber.accept(i);
      // Release the zero semaphore after printing an even number.
      semaphoreZero.release();
    }
  }

  /**
   * Prints odd numbers.
   * @param printNumber The consumer interface to output a number.
   * @throws InterruptedException if the current thread is interrupted while waiting.
   */
  public void odd(IntConsumer printNumber) throws InterruptedException {
    for (int i = 1; i <= n; i += 2) {
      // Wait for the odd semaphore to be released.
      semaphoreOdd.acquire();
      printNumber.accept(i);
      // Release the zero semaphore after printing an odd number.
      semaphoreZero.release();
    }
  }
}
```

C++ Solution

```cpp
#include <semaphore.h>
#include <functional> // Required for std::function

class ZeroEvenOdd {
private:
    int n; // The maximum number we will count up to.
    sem_t semZero, semEven, semOdd; // Semaphores for zero, even, and odd.

public:
    ZeroEvenOdd(int n) : n(n) {
        sem_init(&semZero, 0, 1); // Initialize semZero so the first number printed is zero.
        sem_init(&semEven, 0, 0); // Initialize semEven with no permits since we start with zero.
        sem_init(&semOdd, 0, 0);  // Initialize semOdd with no permits since we start with zero.
    }

    // Outputs "0" and alternates between even and odd permissions.
    void zero(std::function<void(int)> printNumber) {
        for (int i = 1; i <= n; ++i) {
            sem_wait(&semZero); // Wait for the semaphore to be available.
            printNumber(0);     // Print zero.
            // Following zero, decide whether to allow even or odd number to print next.
            if (i % 2 == 0) {
                sem_post(&semEven); // If the turn is even (starting at #2), allow odd to print next.
            } else {
                sem_post(&semOdd); // If the turn is odd (1, 3, 5,...), allow even to print next.
            }
        }
    }

    // Only outputs the even numbers.
    void even(std::function<void(int)> printNumber) {
        for (int i = 2; i <= n; i += 2) {
            sem_wait(&semEven); // Wait for the even semaphore to be available.
            printNumber(i);     // Print the even number.
            sem_post(&semZero); // Signal that zero should be printed next.
        }
    }

    // Only outputs the odd numbers.
    void odd(std::function<void(int)> printNumber) {
        for (int i = 1; i <= n; i += 2) {
            sem_wait(&semOdd); // Wait for the odd semaphore to be available.
            printNumber(i);    // Print the odd number.
            sem_post(&semZero); // Signal that zero should be printed next.
        }
    }

    // Ensure proper destruction of semaphores to avoid any resource leak.
    ~ZeroEvenOdd() {
        sem_destroy(&semZero);
        sem_destroy(&semEven);
        sem_destroy(&semOdd);
    }
};
```

Typescript Solution

```typescript
const n: number = 5; // Replace with the maximum number you wish to count up to.
let isZeroTurn = true; // Boolean to know if we should be printing 0 or switching between odd/even.
let isOddTurn = true; // Boolean to determine whose turn it is, odd or even, after zero.

// Define the printNumber function, which is a stand-in for actually printing a number.
const printNumber = (num: number) => {
  console.log(num);
};

// For zero printing function.
const zero = async () => {
  for (let i = 1; i <= n; i++) {
    // Check if it's zero's turn before printing zero.
    while (!isZeroTurn) await new Promise(resolve => setTimeout(resolve, 10));
    printNumber(0);
    // Toggle the turn to odd or even after printing 0.
    isZeroTurn = false;
    isOddTurn = false; // Now it's not Zero's turn anymore.
  }
};

// For even numbers printing function.
const even = async () => {
  for (let i = 2; i <= n; i += 2) {
    // Wait for the even's turn.
    while (isZeroTurn || !isOddTurn) await new Promise(resolve => setTimeout(resolve, 10));
    printNumber(i);
    isZeroTurn = true; // After printing an even number, it's now Zero's turn.
  }
};

// For odd numbers printing function.
const odd = async () => {
  for (let i = 1; i <= n; i += 2) {
    // Wait for odd's turn.
    while (isZeroTurn || !isOddTurn) await new Promise(resolve => setTimeout(resolve, 10));
    printNumber(i);
    isZeroTurn = true; // After printing an odd number, it's now Zero's turn.
  }
};

// Run the functions asynchronously to simulate parallel execution.
zero();
even();
odd();
```

Time and Space Complexity

Time Complexity

The time complexity of the given code is primarily dictated by the number of iterations each method will execute:

- The `zero` method is called `n` times, since it iterates from 0 to `n-1`. So, its time complexity is $O(n)$.
- The `even` method is called `n/2` times because it only deals with even numbers starting from 2 up to `n`. Thus, its time complexity is $O(n/2)$, which simplifies to $O(n)$ when analyzed for asymptotic behavior.
- Similarly, the `odd` method is called `n/2` times since it only deals with odd numbers starting from 1 to `n`. So, its complexity is also $O(n/2)$, which simplifies to $O(n)$.

All methods are executed concurrently, but since they depend on each other through the use of semaphores (synchronization mechanisms), the overall time complexity is still $O(n)$, because they don't run entirely in parallel due to the synchronization constraints.

Space Complexity

The space complexity of the code is defined by the variables and synchronization primitives used:

- Three semaphores are initiated, regardless of the value of `n`.
- The variable `self.n` represents the input parameter and does not change with respect to `n`.

Hence, the space complexity of the class and its methods is $O(1)$ since the amount of space used does not scale with the input size `n`.