1116. Print Zero Even Odd

Medium Concurrency

## Problem Description

even, and odd. Three different threads will call these methods. The task is to print a sequence of numbers following a pattern:

The sequence should follow the 010203040506... pattern, and the total length of the sequence must be 2n, where n is an integer

Leetcode Link

- The sequence starts with 0
- An odd number follows the 0 Then zero again
- Then an even number ... and so on.
- For example, if n = 5, the sequence to print would be "0102030405".

provided when creating an instance of the ZeroEvenOdd class.

The challenge is to ensure that: zero() function only outputs 0's.

• odd() function only outputs odd numbers. These functions must be called in the right sequence by different threads to maintain

the mentioned sequence pattern.

effectively blocking them.

in the correct sequence.

Solution Approach

- Intuition

even() function only outputs even numbers.

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

To solve this problem, we need to coordinate between threads to print the sequence in order. If the threads run without any

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,

2. Print 0.

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

As we proceed, the call to zero printing function will: 1. Acquire z.

The odd and even functions will: Wait to acquire o or e (respectively).

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

3. Release z to allow the next zero to be printed.

2. Once acquired, print the next odd or even number.

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

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).

3. Release o or e depending on whether the next number should be odd or even.

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

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.

1. self.z (semaphore for zero): Initialized with a count of 1 because we always start the sequence with 0.

2. Invoke printNumber(0) to print zero to the console. 3. Determine whether the next number should be odd or even based on 1's value. If 1 % 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,

The even function is responsible for printing even numbers only. The loop iterates through even numbers starting from 2 until n,

3. Release the self.z semaphore, allowing the zero function to print the next 0.

3. Release the self.z semaphore, allowing the zero function to print the next 0.

## 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.

odd function

odd methods.

respectively.

incrementing by 2 in each iteration:

even function

By utilizing semaphores to control the sequence of method calls across different threads, the threads are coordinated in such a way

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

2. Invoke printNumber(i) where i is the current odd number in the loop to print the number to the console.

1. Wait until the self.o semaphore is greater than 0, and then acquire it (decrement its value).

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".

7. The zero function's thread acquires self.z again (decrements self.z to 0) and prints the second "0".

9. The thread running even function was waiting for self.e to be greater than 0, so it can now proceed.

- that printNumber prints the numbers in the sequence as if it were in a single-threaded environment. Example Walkthrough
- 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.

8. Since the next number is 2 (even), the thread running zero now releases semaphore self.e (increments self.e to 1).

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

ZeroEven0dd class will utilize semaphores to synchronize the print calls across the three threads corresponding to the zero, even, and

Python Solution

class ZeroEvenOdd:

from threading import Semaphore

def \_\_init\_\_(self, n: int):

for i in range(self.n):

print\_number(0)

if i % 2 == 0:

print\_number(i)

print\_number(i)

public ZeroEvenOdd(int n) {

\* Prints the number 0.

} else {

for (int i = 1;  $i \le n$ ; ++i) {

printNumber.accept(0);

if (i % 2 == 0) {

semaphoreZero.acquire();

semaphoreEven.release();

this.n = n;

for i in range(2, self.n + 1, 2):

for i in range(1, self.n + 1, 2):

def odd(self, print\_number: Callable[[int], None]) -> None:

:param print\_number: The function to call to print the number

Prints the odd numbers starting from 1 to n

// Semaphores serve as locks to control the print order.

\* @param printNumber The consumer interface to output a number.

// Acquire the semaphore before printing 0.

public void zero(IntConsumer printNumber) throws InterruptedException {

\* @throws InterruptedException if the current thread is interrupted while waiting.

// Determine whether to release the odd or even semaphore based on the current iteration.

private Semaphore semaphoreZero = new Semaphore(1);

private Semaphore semaphoreEven = new Semaphore(0);

private Semaphore semaphoreOdd = new Semaphore(0);

else:

from typing import Callable

self.n = n

print out of turn.

6

12

13

14

15

16

17

18

19

20

21

22

23

24

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

6

8

9

10

11

12

13

14

15

16

17

18

19

20

21

23

24

25

26

27

28

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

};

};

8

12

13

14

15

16

17

18

19

29

31

35

36

37

38

39

41

46

40 };

43 zero();

44 even();

Time Complexity

Space Complexity

45 odd();

30 };

20 };

} else {

// Only outputs the even numbers.

printNumber(i);

// Only outputs the odd numbers.

printNumber(i);

sem\_destroy(&semZero);

sem\_destroy(&semEven);

sem\_destroy(&semOdd);

6 const printNumber = (num: number) => {

// For zero printing function.

for (let i = 0; i < n; ++i) {

isOddTurn = i % 2 === 0;

32 // For odd numbers printing function.

Time and Space Complexity

O(n/2), which simplifies to O(n).

// Wait for odd's turn.

for (let i = 1; i <= n; i += 2) {

// Check if it's zero's turn before printing zero.

// Toggle the turn to odd or even after printing 0.

isZeroTurn = false; // Now it's not Zero's turn anymore.

~ZeroEvenOdd() {

Typescript Solution

console.log(num);

11 const zero = async () => {

printNumber(0);

void even(std::function<void(int)> printNumber) {

void odd(std::function<void(int)> printNumber) {

for (int i = 2;  $i \le n$ ; i += 2) {

for (int i = 1;  $i \le n$ ; i += 2) {

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

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 8 self.odd\_semaphore = Semaphore(0) # A semaphore for allowing 'odd' numbers to be printed 9 10 11 def zero(self, print\_number: Callable[[int], None]) -> None:

Prints 'zero' n times and alternates the control between printing odd and even numbers

self.zero\_semaphore.acquire() # Wait for the zero semaphore to be released

self.even\_semaphore.acquire() # Wait for the even semaphore to be released

self.zero\_semaphore.release() # Allow the zero method to print zero again

self.odd\_semaphore.acquire() # Wait for the odd semaphore to be released

self.zero\_semaphore.release() # Allow the zero method to print zero again

self.odd\_semaphore.release() # Allow the odd method to print an odd number

self.even\_semaphore.release() # Allow the even method to print an even number

# Print the even number

# Print the odd number

# Print zero

# Determine if the next number after zero should be odd or even

:param print\_number: The function to call to print the number

25 26 def even(self, print\_number: Callable[[int], None]) -> None: 27 28 Prints the even numbers starting from 2 to n 29 :param print\_number: The function to call to print the number

# If 'i' is even, the next number should be odd

# If 'i' is odd, the next number should be even

```
1 import java.util.concurrent.Semaphore;
  import java.util.function.IntConsumer;
  class ZeroEvenOdd {
      private int n;
```

/\*\*

Java Solution

```
29
                   semaphoreOdd.release();
30
31
32
33
34
       /**
35
        * Prints even numbers.
36
        * @param printNumber The consumer interface to output a number.
37
        * @throws InterruptedException if the current thread is interrupted while waiting.
38
        */
       public void even(IntConsumer printNumber) throws InterruptedException {
39
           for (int i = 2; i \le n; i += 2) {
40
               // Wait for the even semaphore to be released.
               semaphoreEven.acquire();
               printNumber.accept(i);
43
               // Release the zero semaphore after printing an even number.
44
45
               semaphoreZero.release();
46
47
48
49
       /**
50
        * Prints odd numbers.
51
        * @param printNumber The consumer interface to output a number.
52
        * @throws InterruptedException if the current thread is interrupted while waiting.
53
        */
54
       public void odd(IntConsumer printNumber) throws InterruptedException {
           for (int i = 1; i \le n; i += 2) {
55
56
               // Wait for the odd semaphore to be released.
57
               semaphoreOdd.acquire();
               printNumber.accept(i);
58
               // Release the zero semaphore after printing an odd number.
59
               semaphoreZero.release();
60
61
62
63 }
64
C++ Solution
  1 #include <semaphore.h>
  2 #include <functional> // Required for std::function
    class ZeroEvenOdd {
    private:
         int n; // The maximum number we will count up to.
         sem_t semZero, semEven, semOdd; // Semaphore for zero, even, and odd.
    public:
  9
         ZeroEvenOdd(int n) : n(n) {
 10
 11
             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.
 12
 13
             sem_init(&semOdd, 0, 0); // Initialize semOdd with no permits since we start with zero.
 14
 15
 16
         // Outputs "0" and alternates between even and odd permissions.
 17
         void zero(std::function<void(int)> printNumber) {
 18
             for (int i = 0; i < n; ++i) {
 19
                 sem_wait(&semZero); // Wait for the semaphore to be available.
 20
                 printNumber(0);
                                     // Print zero.
 21
                 // Following zero, decide whether to allow even or odd number to print next.
 22
                 if (i % 2 == 0) {
 23
                     sem_post(&semOdd); // If the turn is even (starting at 0), allow odd to print next.
```

sem\_post(&semEven); // If the turn is odd (1, 3, 5,...), allow even to print next.

sem\_wait(&semEven); // Wait for the even semaphore to be available.

sem\_wait(&semOdd); // Wait for the odd semaphore to be available.

// Print the odd number.

sem\_post(&semZero); // Signal that zero should be printed next.

// Ensure proper destruction of semaphores to avoid any resource leak.

1 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.

while (!isZeroTurn) await new Promise(resolve => setTimeout(resolve, 10));

isZeroTurn = true; // After printing an even number, it's now Zero's turn.

isZeroTurn = true; // After printing an odd number, it's now Zero's turn.

// Run the functions asynchronously to simulate parallel execution.

while (isZeroTurn || !isOddTurn) await new Promise(resolve => setTimeout(resolve, 10));

// Print the even number.

sem\_post(&semZero); // Signal that zero should be printed next.

## 21 22 // For even numbers printing function. 23 const even = async () => { for (let i = 2; $i \le n$ ; i += 2) { // Wait for even's turn. while (isZeroTurn || isOddTurn) await new Promise(resolve => setTimeout(resolve, 10)); 26

printNumber(i);

33 const odd = async () => {

printNumber(i);

• 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

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

- 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 0(n), because they don't run entirely in parallel due to the synchronization constraints.
- 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 0(1) since the amount of space used does not scale with the input size n.

```
In this problem, we are working with a multi-threaded setup using a class ZeroEvenOdd. The class comes with three methods: zero,
```