1114. Print in Order

Easy Concurrency

Problem Description In this problem, we are given a class Foo with three methods: first(), second(), and third(). These methods are intended to be

regardless of how the threads are scheduled by the operating system. This means that if Thread B tries to call second() before Thread A calls first(), Thread B should wait until first() has been called. Similarly, Thread C must wait for second() to complete before calling third().

called by three different threads, let's call them Thread A, Thread B, and Thread C respectively. The challenge is to ensure that

these methods are called in the strict sequence where first() is called before second() and second() is called before third(),

Intuition

The solution to this problem involves synchronization mechanisms that allow threads to communicate with each other about the state of the execution. One common synchronization mechanism provided by many programming languages, including Python, is

the Semaphore. A semaphore manages an internal counter which is decremented by each acquire() call and incremented by each release() call. The counter can never go below zero; when acquire() finds that it is zero, it blocks, waiting until some other thread calls

release(). In the given solution:

• self.b and self.c are initialized with a count of 0 to block the second() and third() operations respectively until they are explicitly released.

• We have three semaphores: self.a, self.b, and self.c. self.a is initialized with a count of 1 to allow the first() operation to proceed immediately.

- The first() method:
- Acquires self.a, ensuring no other operation is currently in progress. • Once the first() operation is complete, it releases self.b to allow second() to proceed.

The second() method: • Acquires self.b, which will only be available once self.a has been released by the first() method.

- After completing its operation, it releases self.c to allow third() method execution. The third() method:
- Acquires self.c, which will only be available after self.b is released by the second() method. • After completing its operation, it releases self.a. This is optional in the context where only one cycle of operations (first(), second(), third()) is being performed but might be included for scenarios where the sequence may restart.

The method performs its intended operation printFirst().

1, third() can proceed (the acquire operation decrements it to 0).

ensures these functions are called in the strict sequence required.

- Solution Approach
- The solution approach effectively utilizes semaphores, which are synchronization primitives that control access to a common

• Inside the Foo class constructor, three semaphores are initialized. Semaphores self.b and self.c are initialized with a count of 0 to ensure that second() and third() methods are blocked initially. Semaphore self. a is initialized with a count of 1 to allow first() to proceed

Class Initialization:

without blocking.

second() operation to proceed.

Executing second():

Executing first(): • The first method starts by calling self.a.acquire(). Since self.a was initialized to 1, first() is allowed to proceed as acquire() will decrement the counter to 0, and no blocking occurs.

• After completion, it calls self.b.release(). This increments the counter of semaphore self.b from 0 to 1, thereby allowing a blocked

resource by multiple threads in a concurrent system. Here's a step-by-step breakdown of how the solution is implemented:

• The second method calls self.b.acquire(). If first() has already completed and called self.b.release(), the semaphore self.b counter would be 1, and second() can proceed (the acquire operation decrements it back to 0). If first() has not completed, second() will be

blocked.

 It then executes printSecond(). • Upon successful completion, it calls self.c.release(), increasing the counter of the semaphore self.c from 0 to 1, and thus unblocking the third() method.

• The third method begins with self.c.acquire(). Up until second() calls self.c.release(), third() will be blocked. Once self.c counter is

Executing third():

It executes printThird().

o Optionally, it can call self.a.release() which is omitted in the given code snippet because it's not necessary unless the sequence of operations is intended to repeat. Each semaphore acts as a turnstile, controlling the flow of operations. The use of semaphores ensures that no matter the order in

which threads arrive, they will be forced to execute in the necessary order: first() then second() then third().

This implementation exemplifies a classic synchronization pattern where the completion of one action triggers the availability of the next action in a sequence. The semaphores act like gates that open once the previous operation has signaled that it's safe for

Example Walkthrough

the next operation to start.

Imagine we have three simple functions that need to be executed in order: printFirst(), printSecond(), and printThird(). They simply print "first", "second", and "third" respectively to the console. Now, let's see how the solution approach described earlier

Let's assume the three threads are started almost simultaneously by the operating system, but due to some randomness in

thread scheduling, the actual order of invocation is Thread B (second), Thread C (third), and finally, Thread A (first).

Thread B (second) arrives first: • Calls second() method and tries to acquire semaphore self.b with an acquire() call. Since self.b was initialized with 0, it is blocked as the counter is already at 0, and acquire() cannot decrement it further. Thread B will now

As with self.b, self.c was initialized with 0, so Thread C is blocked because the semaphore's counter is not greater than 0. It must wait for self.c to be released by the second() operation.

Thread C (third) arrives next:

Thread A (first) arrives last:

Thread B (second) resumes:

• It proceeds to call the first() method which attempts to acquire semaphore self.a with an acquire() call. • Since self.a was initialized with 1, the acquire() will succeed, the counter will decrement to 0, and the first() method will proceed. The printFirst() function is executed, outputting "first".

• Upon finishing its operation, Thread B calls self.c.release(), incrementing self.c's counter to 1, allowing the third operation to proceed.

Similar to the previous steps, with the self.c counter now at 1, the third() method proceeds as acquire() brings the counter down to 0.

Upon completion, the first() method calls self.b.release(), which increments semaphore self.b's counter to 1.

• With self.b's counter now at 1, Thread B can proceed as acquire() successfully decrements it back to 0.

Thread C (third) resumes:

from threading import Semaphore

self.first_done = Semaphore(1)

self.first_done.acquire()

self.second_done.release()

self.second_done.acquire()

self.first_done.release()

void first(std::function<void()> printFirst) {

void second(std::function<void()> printSecond) {

void third(std::function<void()> printThird) {

printFirst();

printSecond();

printThird();

// Counter to keep track of the order

// Promises to control the execution order

self.third done = Semaphore(0)

self.first_done.acquire()

self.second_done.release()

self.second_done.acquire()

self.third_done.release()

Time and Space Complexity

print_first()

print_second()

def first(self, print_first: Callable[[], None]) -> None:

Execute the print_first function to output "first".

Release semaphore to allow 'second' method to run.

def second(self, print_second: Callable[[], None]) -> None:

Release semaphore to allow 'third' method to run.

Execute the print_second function to output "second".

Acquire semaphore to enter 'first' method.

Wait for the completion of 'first' method.

};

TypeScript

let count = 1;

std::unique_lock<std::mutex> lock(mtx); // Acquire the lock

std::unique_lock<std::mutex> lock(mtx); // Acquire the lock

std::unique_lock<std::mutex> lock(mtx); // Acquire the lock

count = 2; // Update the count to allow second to run

count = 3; // Update the count to allow third to run

let firstSecondControl: (value: void | PromiseLike<void>) => void;

let secondThirdControl: (value: void | PromiseLike<void>) => void;

// Promise that will resolve when it's okay to run `second`

cv.notify_all(); // Notify all waiting threads

cv.notify_all(); // Notify all waiting threads

// printFirst() outputs "first". Do not change or remove this line.

cv.wait(lock, [this] { return count == 2; }); // Wait until first is done

cv.wait(lock, [this] { return count == 3; }); // Wait until second is done

// No need to update count or notify, as no further actions are dependent on third

// printThird() outputs "third". Do not change or remove this line.

// printSecond() outputs "second". Do not change or remove this line.

Acquire semaphore to enter 'first' method.

Wait for the completion of 'first' method.

Execute the print_first function to output "first".

Release semaphore to allow 'second' method to run.

def second(self, print_second: Callable[[], None]) -> None:

Release semaphore to allow 'third' method to run.

Execute the print_third function to output "third".

This line could re-enable the flow for 'first', in case of repeated calls.

Execute the print_second function to output "second".

from typing import Callable

def __init__(self):

print_first()

print_second()

Python

class Foo:

 printThird() is executed, and "third" is printed to the console. In this example, even though the threads arrived out of order, the use of semaphores forced them to wait for their turn, ensuring

• The printSecond() function is called, printing "second" to the console.

Initialize semaphores to control the order of execution.

Semaphore 'first_done' allows 'first' method to run immediately.

Semaphore 'second_done' starts locked, preventing 'second' method from running.

wait for self.b to be released by another operation (specifically, the first() operation).

• It calls the third() method which tries to acquire semaphore self.c with acquire().

- the desired sequence of "first", "second", "third" in the console output. Solution Implementation
 - self.second_done = Semaphore(0) # Semaphore 'third_done' starts locked, preventing 'third' method from running. self.third_done = Semaphore(0) def first(self, print_first: Callable[[], None]) -> None:

self.third_done.release() def third(self, print_third: Callable[[], None]) -> None: # Wait for the completion of 'second' method. self.third_done.acquire()

Java

print_third()

```
import java.util.concurrent.Semaphore;
public class Foo {
    private Semaphore firstJobDone = new Semaphore(1); // Semaphore for first job, initially available
   private Semaphore secondJobDone = new Semaphore(0); // Semaphore for second job, initially unavailable
    private Semaphore thirdJobDone = new Semaphore(0); // Semaphore for third job, initially unavailable
    public Foo() {
       // Constructor
    // Method for the first job; prints "first"
    public void first(Runnable printFirst) throws InterruptedException {
        firstJobDone.acquire(); // Wait for the first job's semaphore to be available
       printFirst.run(); // Run the printFirst task; this should print "first"
        secondJobDone.release(); // Release the second job semaphore, allowing the second job to run
    // Method for the second job; prints "second"
    public void second(Runnable printSecond) throws InterruptedException {
        secondJobDone.acquire(); // Wait for the second job's semaphore to be available
       printSecond.run(); // Run the printSecond task; this should print "second"
        thirdJobDone.release(); // Release the third job semaphore, allowing the third job to run
    // Method for the third job; prints "third"
    public void third(Runnable printThird) throws InterruptedException {
        thirdJobDone.acquire(); // Wait for the third job's semaphore to be available
       printThird.run(); // Run the printThird task; this should print "third"
       firstJobDone.release(); // Release the first job semaphore, allowing the cycle of jobs to be restarted (if necessary)
C++
#include <mutex>
#include <condition_variable>
#include <functional>
class Foo {
private:
    std::mutex mtx; // Mutex to protect condition variable
    std::condition_variable cv; // Condition variable for synchronization
    int count; // Counter to keep track of the order
public:
    Foo() {
        count = 1; // Initialize count to 1 to ensure first is executed first
```

```
const canRunSecond = new Promise<void>((resolve) => {
 firstSecondControl = resolve;
});
// Promise that will resolve when it's okay to run `third`
const canRunThird = new Promise<void>((resolve) => {
  secondThirdControl = resolve;
});
async function first(printFirst: () => void): Promise<void> {
 // printFirst() outputs "first".
  printFirst();
 // Update the count to allow second to run
  count = 2;
 // Resolve the promise to unblock the second function
  firstSecondControl();
```

```
async function second(printSecond: () => void): Promise<void> {
    // Wait until the first function has completed
    if (count !== 2) {
      await canRunSecond;
    // printSecond() outputs "second".
    printSecond();
    // Update the count to allow third to run
    count = 3;
    // Resolve the promise to unblock the third function
    secondThirdControl();
  async function third(printThird: () => void): Promise<void> {
    // Wait until the second function has completed
    if (count !== 3) {
      await canRunThird;
    // printThird() outputs "third".
    printThird();
    // After third, there are no more actions, so nothing more to do
from threading import Semaphore
from typing import Callable
class Foo:
   def __init__(self):
       # Initialize semaphores to control the order of execution.
       # Semaphore 'first_done' allows 'first' method to run immediately.
        self.first_done = Semaphore(1)
       # Semaphore 'second_done' starts locked, preventing 'second' method from running.
        self.second_done = Semaphore(0)
```

```
def third(self, print_third: Callable[[], None]) -> None:
    # Wait for the completion of 'second' method.
    self.third_done.acquire()
    # Execute the print_third function to output "third".
    print_third()
    # This line could re-enable the flow for 'first', in case of repeated calls.
    self.first_done.release()
```

Semaphore 'third_done' starts locked, preventing 'third' method from running.

performs a constant amount of work: acquiring and releasing a semaphore. The use of semaphores is to control the order of execution but does not add any significant computational overhead. The space complexity of the Foo class is 0(1) as well. The class has three semaphores as its member variables, and the number

The time complexity of the Foo class methods first, second, and third is 0(1) for each call. This is because each method

of semaphores does not increase with the input size. Hence, the memory used by an instance of the class is constant.