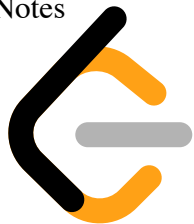


Notes



# LeetCode

[Explore Problems](#)[Interview](#)[Contest](#)[Discuss](#)[Store](#)

0

[Description](#)[Solution](#)[Discuss \(668\)](#)[Submissions](#)

1114. Print in Order

Easy

👍1109

👏180

♡Add to List

🔗Share

Suppose we have a class:

```
public class Foo {  
    public void first() { print("first"); }  
    public void second() { print("second"); }  
    public void third() { print("third"); }  
}
```

The same instance of `Foo` will be passed to three different threads. Thread A will call `first()`, thread B will call `second()`, and thread C will call `third()`. Design a mechanism and modify the program to ensure that `second()` is executed after `first()`, and `third()` is executed after `second()`.

**Note:**

We do not know how the threads will be scheduled in the operating system, even though the numbers in the input seem to imply the ordering. The input format you see is mainly to ensure our tests' comprehensiveness.

**Example 1:****Input:** `nums = [1,2,3]`**Output:** `"firstsecondthird"`**Explanation:** There are three threads being fired asynchronously. The input `[1,2,3]` means thread**Example 2:****Input:** `nums = [1,3,2]`**Output:** `"firstsecondthird"`**Explanation:** The input `[1,3,2]` means thread A calls `first()`, thread B calls `third()`, and

**Constraints:**

- `nums` is a permutation of `[1, 2, 3]`.


Accepted

117.2K

Submissions

171.9K

Seen this question in a real interview before?

Companies  *i*

v

Related Topics

v

[Concurrency](#)

Similar Questions

v

[Print FooBar Alternately](#)

Medium

Quick Navigation



★

★

★

★

★

Average Rating: 4.53 (62 votes)

Premium

## Solution

### Problems of Concurrency

The concurrency problems arise from the scenario of [concurrent computing](#), where the execution of a program is conducted in multiple processes (or threads) *simultaneously*.

By simultaneousness, the processes or threads are not necessarily running independently in different physical CPUs, but more often they interleave in the same physical CPU. *Note that, the concurrency could apply to either process or thread, we use the words of "process" and "thread" interchangeably in the following sections.*

The concurrency is designed to above all enable multitasking, yet it could easily bring some bugs into the program if not applied properly. Depending on the consequences, the problems caused by concurrency can be categorized into three types:

- **race conditions:** the program ends with an undesired output, resulting from the sequence of executing the processes.

- **deadlocks:** the concurrent processes wait for some necessary resources from each other. As a result, none of them can make progress.
- **resource starvation:** a process is perpetually denied necessary resources to progress its works.

In particular, our problem here can be attributed to the race conditions. Before diving into the solutions, we show an example of race condition.

Suppose we have a function called `withdraw(amount)` which deduces certain amount of money from the balance, if the demanding amount is less than the current balance. At the end, the function returns the remaining balance. The function is defined as follows:

Java Python

Copy

```
1 int balance = 500;
2 int withdraw(int amount) {
3     if (amount < balance) {
4         balance -= amount;
5     }
6     return balance;
7 }
```

As we can see, in the normal case, we expect that the `balance` would never become negative after the execution of the function, which is also the *desired* behavior of the function.

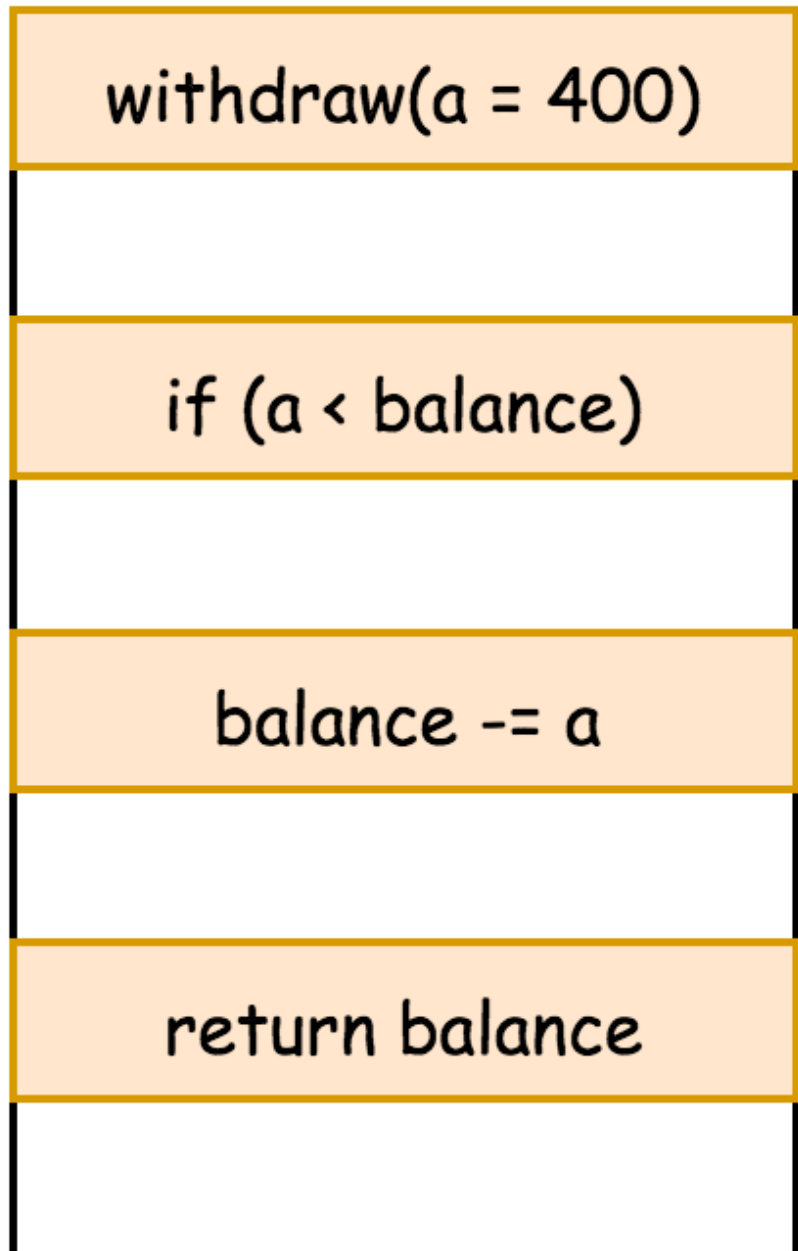
However, unfortunately we could run into a *race condition* where the `balance` becomes negative. Here is how it could happen. Imagine we have two threads invoking the function at the same time with different input parameters, *e.g.* for thread #1, `withdraw(amount=400)` and for thread #2, `withdraw(amount=200)`. The execution of the two threads is scheduled as the graph below, where at each time instance, we run exclusively only a statement from either threads.

## Race

## Timeline



## Thread #1



As one can see, at the end of the above execution flow, we would end up with a negative balance, which is desired output.

## Race-free Concurrency

The concurrency problems share one common characteristic: multiple processes/threads share some resources (*e.g.* the variable `balance`). Since we cannot eliminate the constraint of resource sharing, the key to prevent the concurrency problems boils down to *the coordination of resource sharing*.

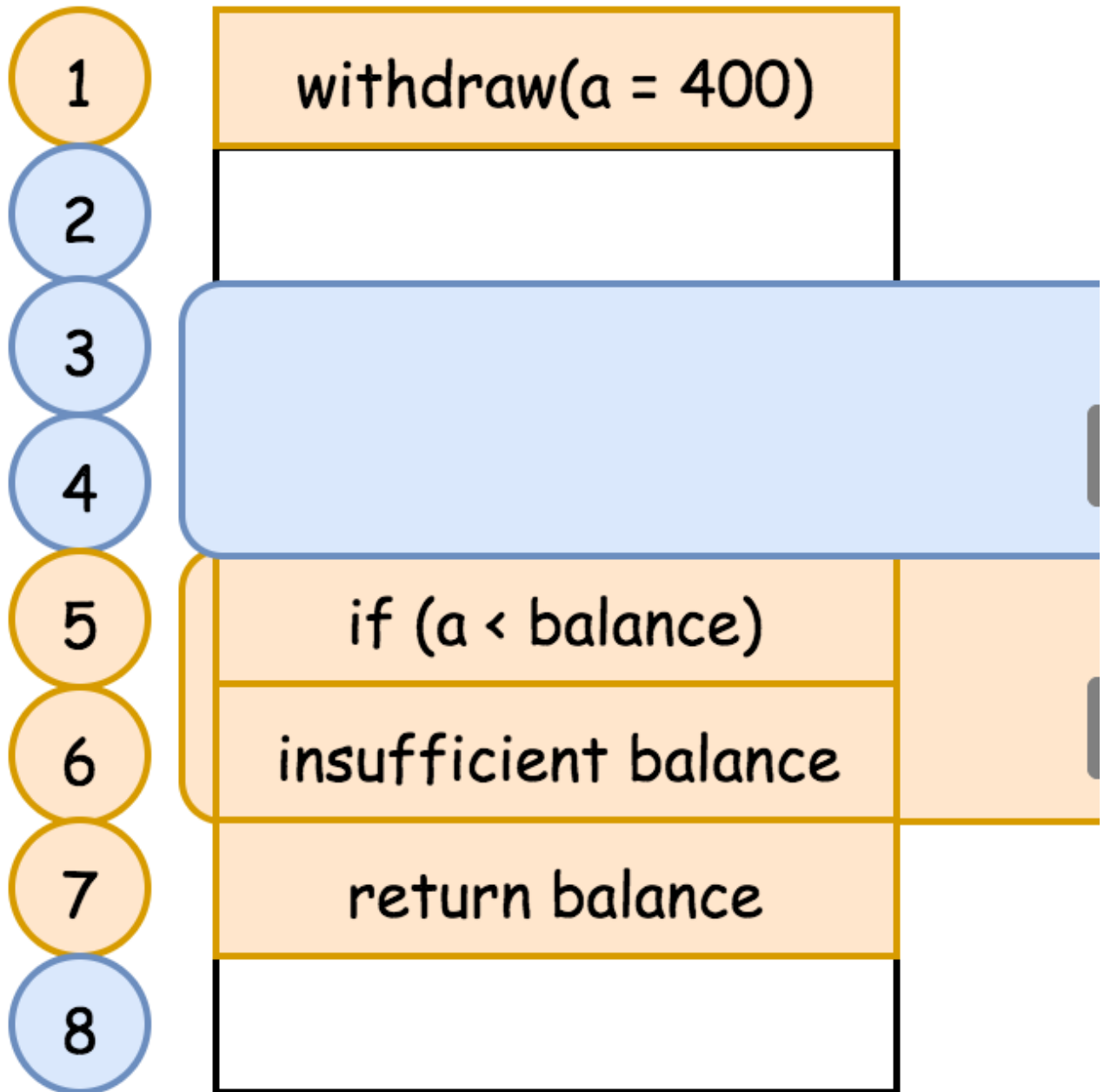
The idea is that if we could ensure *the exclusivity of certain critical code section* (*e.g.* the statements to check and deduce the balance), we could prevent the program from running into certain inconsistent states.

The solution to the race condition becomes clear: we need **certain mechanism** that could enforce the exclusivity of certain critical code section, *i.e.* at a given time, only one thread can enter the critical section.

One can consider the mechanism as a sort of *lock* that restricts the access of the critical section. Following the previous example, we apply the lock on the critical section, *i.e.* the statements of balance check and balance deduction. We then rerun the two threads, which could lead to the following flow:

## Timeline

## Thread #1



With the mechanism, once a thread enters the critical section, it would prevent other threads from entering the same critical section. For example, at the timestamp #3, the thread #2 enters the critical section. Then at the next timestamp #4, the thread #1 could have sneaked into the *dangerous* critical section if the statement was not protected by the lock. At the end, the two threads run concurrently, while the consistency of the system is maintained, *i.e.* the balance remains positive.

If the thread is not granted with the access of the critical section, we can say that the thread is *blocked* or *sleep*, *e.g.* the thread #1 is blocked at the timestamp #4. As one can imagine, once the critical section is released, *would be nice to notify the waiting threads*. For instance, as soon as the thread #2 releases the critical section at the timestamp #5, the thread #1 got notified to take over the critical section.

As a result, it is often the case that the mechanism also comes with the capability to wake up those waiting peers.

To summarize, in order to prevent the race condition in concurrency, we need a mechanism that possess two capabilities: 1). access control on critical section. 2). notification to the blocking threads.

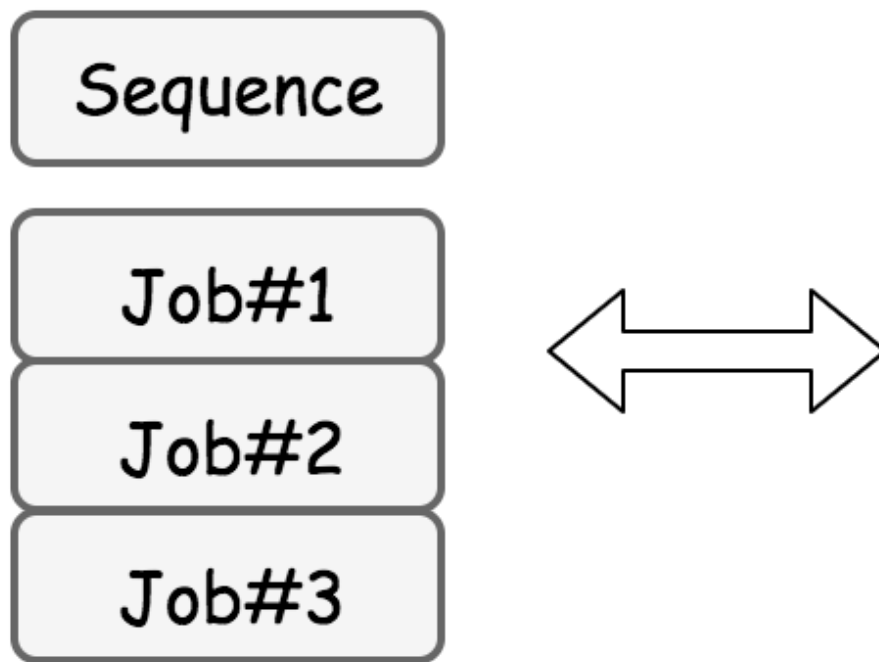
---

## Approach 1: Pair Synchronization

### Intuition

The problem asks us to complete three jobs in order, while each job is running in a separated thread. In order to enforce the execution sequence of the jobs, we could create some dependencies between pairs of jobs, *i.e.* the second job should depend on the completion of the first job and the third job should depend on the completion of the second job.

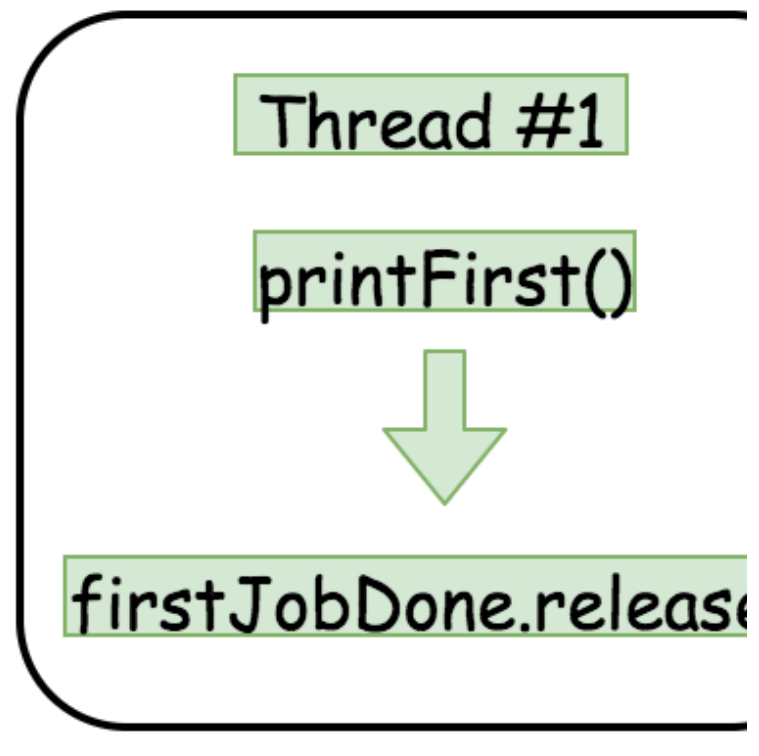
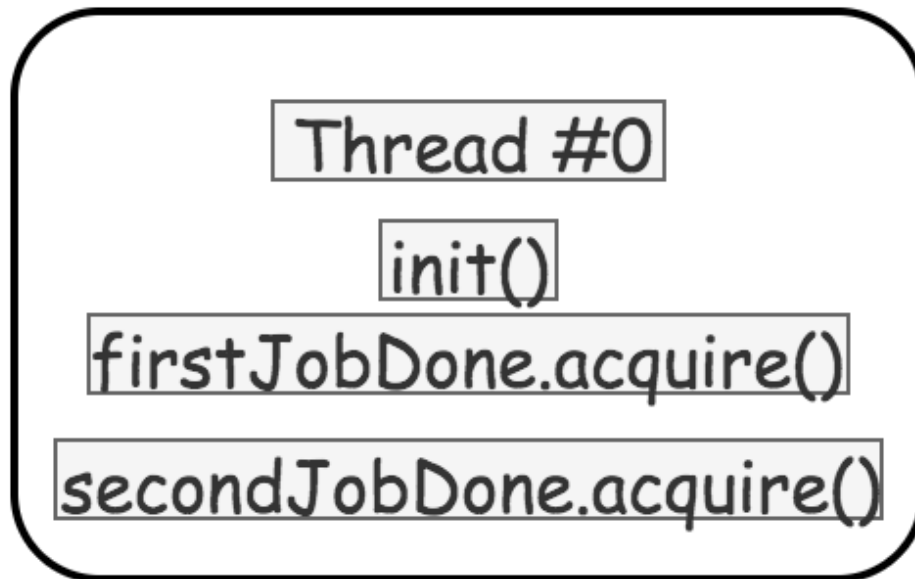
The dependency between pairs of jobs construct a *partial order* on the execution sequence of all jobs, *e.g.* with  $A < B$ ,  $B < C$ , we could obtain the sequence of  $A < B < C$ .



The dependency can be implemented by the concurrency mechanism as we discussed in the previous section. The idea is that we could use a shared variable named `firstJobDone` to coordinate the execution order between the first job and the second job. Similarly, we could use another variable `secondJobDone` to enforce the order of execution between the second and the third jobs.

### Algorithm

- First of all, we initialize the coordination variables `firstJobDone` and `secondJobDone`, to indicate that the jobs are not done yet.
- In the `first()` function, we have no dependency so that we could get straight down to the job. At the end of the function, we then update the variable `firstJobDone` to indicate that the first job is done.
- In the `second()` function, we check the status of `firstJobDone`. If it is not updated, we then wait, otherwise we proceed to the task of the second job. And at the end of function, we update the variable `secondJobDone` to mark the completion of the second job.
- In the `third()` function, we check the status of the `secondJobDone`. Similarly as the `second()` function, we wait for the signal of the `secondJobDone`, before proceeding to the task of the third job.





## Implementation

The implementation of the above algorithm heavily depends on the programming language that one chooses, since different languages provide different **constructs** for the concurrency mechanism. Though some of the constructs such as [mutex](#) and [semaphore](#) are present across several programming languages including Java, C++ and Python.

Here we provide a few examples using different constructs across the languages. In particular, one could find a nice [summary](#) in the Discussion forum about the concurrency constructs in Python.

C++JavaPython

 Copy

```
1  #include <semaphore.h>
2
3  class Foo {
4
5  protected:
6      sem_t firstJobDone;
7      sem_t secondJobDone;
8
9  public:
10
11      Foo() {
12          sem_init(&firstJobDone, 0, 0);
13          sem_init(&secondJobDone, 0, 0);
14      }
15
16      void first(function<void()> printFirst) {
17          // printFirst() outputs "first".
18          printFirst();
19          sem_post(&firstJobDone);
20      }
21
22      void second(function<void()> printSecond) {
23          sem_wait(&firstJobDone);
24          // printSecond() outputs "second".
25          printSecond();
26          sem_post(&secondJobDone);
27      }
```

Privacy - Terms

[Report Article Issue](#) Comments: 33☒ Best ☐ Most Votes ☐ Newest to Oldest ☐ Oldest to Newest

Type comment here...  
(Markdown is supported)

[Preview](#)[Post](#)[petitout](#) ★123

October 11, 2019 7:15 AM



[Read More](#)

active wait (while loops) is definitely not a good solution as [@igor84](#) was mentioning.

It is way better and simpler to use CountdownLatch or Semaphore for solving this problem.



65

 Show 3 replies Reply Share Report[nandakishorek17](#) ★47

November 24, 2020 12:20 PM

Read More

Simple semaphore based solution (accepted) -

```

class Foo {

    private Semaphore[] semaphore;

    public Foo() {
        int numOfThreads = 3;
        semaphore = new Semaphore(numOfThreads);

        try {
            for (int i = 0 ; i< numOfThreads; ++i) {
                // Binary semaphore, since there is only one resource i.e., "Print N".
                semaphore[i] = new Semaphore(1, true);

                // Don't let any thread print.
                semaphore[i].acquire();
            }
        } catch (InterruptedException ie) {
            // NO-OP.
        }
    }
}

```

Show 1 reply

Reply // Let a thread to print "first".  
 semaphore[0].release();

Share } catch (InterruptedException ie) {

Report // NO-OP.



```

    public void first(Runnable printFirst) throws InterruptedException {
        // printFirst.run() outputs "first". Do not change or remove this line.
        semaphore[0].release();
    }
}

```

[zhang-peter](#) ★71

```

    // printSecond.run() outputs "second". Do not change or remove this line.
    public void second(Runnable printSecond) throws InterruptedException {
        semaphore[1].acquire();
        printSecond.run();
        semaphore[1].release();
    }
}

```

January 11, 2020 4:18 PM

Read More

[Java] use synchronized, wait, notifyAll:

```

public void third(Runnable printThird) throws InterruptedException {
    semaphore[2].acquire();
    printThird.run();
    semaphore[2].release();
}

class Foo {
    private final AtomicInteger i = new AtomicInteger();
    private final Object lock = new Object();

    public Foo() {
        i.set(0);
    }

    public void first(Runnable printFirst) throws InterruptedException {
        synchronized (lock) {
            while (i.get() != 0) {
                lock.wait();
            }
            printFirst.run();
            i.set(1);
            lock.notifyAll();
        }
    }
}

```

Show 8 replies

Reply

Share

Report



```
public void second(Runnable printSecond) throws InterruptedException {
    synchronized (lock) {
        while (i != 1) {
            // ...
        }
        printSecond.run();
        i.set(2);
    }
}

public void third(Runnable printThird) throws InterruptedException {
    synchronized (lock) {
        while (i != 2) {
            // ...
        }
        printThird.run();
        i.set(3);
    }
}
}
```



★218

[igor84](#)

October 2, 2019 7:06 AM

Read More

Second and third threads in Java solution based on AtomicInteger will consume CPU cycles while waiting. This may be good to have faster runtime in the test, but in real life it is better to have these threads in WAITING state. For concurrency tasks it makes sense to add one more index - CPU load or something like this (in addition to runtime and memory distributions)

▲

30

▼

Show 2 replies

Reply

Share

Report



[pkonduri](#) ★752

August 6, 2020 10:51 PM

Read More

Wonderfully written. It's hard to find practical, to the point content on this topic but you've nailed it

^

6

▼

↩ Reply

🔗 Share

⚠ Report



[jkothari](#) ★4

June 22, 2020 2:47 AM

Read More

is it Necessary to use atomic integers? Mine is also working using int.

^

6

▼

💬 Show 5 replies

↩ Reply

🔗 Share

⚠ Report



[ctrl-alt-lulz](#) ★73

February 5, 2020 1:20 PM

[Read More](#)

Here's the solution with driver code, so you can run it on your own machine.

```
import threading
```

```
class Foo:
```

```
    def __init__(self):
        self.second_lock = threading.Lock()
        self.second_lock.acquire()
        self.third_lock = threading.Lock()
        self.third_lock.acquire()
```

```
    def first(self):
```

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

```
        print('first')
```

```
        self.second_lock.release()
```

```
    def second(self):
```

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

```
        with self.second_lock:
```

```
            print('second_lock')
```

```
            self.third_lock.release()
```

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

```
        with self.third_lock:
```

```
            print('third_lock')
```

```
        t1 = threading.Thread(target=a.third)
```

```
        t2 = threading.Thread(target=a.second)
```

```
        t3 = threading.Thread(target=a.first)
```

```
        t1.start()
```

```
        t2.start()
```

```
        t3.start()
```

October 2, 2019 12:20 AM

[Read More](#)

Nice article.

For C++, you can also use condition variables to achieve both the requirements mentioned above.

Link to my solution:

<https://leetcode.com/problems/print-in-order/discuss/343384/C%2B%2BWhy-most-of-the-solutions-using-mutex-are-wrong%2Bsolution>

^  
5  
v

Show 3 replies

↩ Reply

🔗 Share

⚠ Report



[vijay20](#) ★69

April 22, 2020 5:33 PM

Read More

Can't we use a single atomicinteger variable ?

^  
9  
v

↩ Reply

🔗 Share

⚠ Report



[meakub](#) ★69

October 10, 2019 3:43 AM

Read More

Can anyone please provide the calling method?

▲

4

▼

Show 2 replies

↩ Reply

🔗 Share

⚠ Report

- 
- 1
- 2
- 3
- 4
- 

You don't have any submissions yet.

☰ Problems

✂ Pick One

< Prev

1114/2444

Next >

i

C++

✓

Autocomplete

i

{ }

↺

🔄

⌂

```
xxxxxxxxxx
21
1
class Foo {
2
public:
3
    Foo() {
4
5
    }
```



6

7

```
void first(function<void()> printFirst) {
```

8

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

9

```
    printFirst();
```

10

```
}
```

11

12

```
void second(function<void()> printSecond) {
```

13

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

14

```
    printSecond();
```

15

```
}
```

16

17

```
void third(function<void()> printThird) {
```

18

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

19

```
    printThird();
```

20

```
}
```

21

```
};
```

Your previous code was restored from your local storage. [Reset to default](#)



Console ▾

Contribute



▶ Run Code ^ Submit



///

Privacy - Terms



Type here... (Markdown is enabled)

×

Saved

All Problems

✓

search problems 🔍

🔍

1

✓

<

>

✓ #1 Two Sum

Easy

✓ #2 Add Two Numbers

Medium

✓ #3 Longest Substring Without Repeating Characters

Medium

#4 Median of Two Sorted Arrays

Hard

✓ #5 Longest Palindromic Substring

Medium

✓ #6 Zigzag Conversion

Medium

✓ #7 Reverse Integer

Medium

✓ #8 String to Integer (atoi)

Medium

✓ #9 Palindrome Number

Easy

#10 Regular Expression Matching

Hard

✓ #11 Container With Most Water

Medium

✓ #12 Integer to Roman

Medium

✓ #13 Roman to Integer

Easy

✓ #14 Longest Common Prefix

Easy

✓ #15 3Sum

Medium

#16 3Sum Closest

Medium

✓ #17 Letter Combinations of a Phone Number

Medium

✓ #18 4Sum

Medium

✓ #19 Remove Nth Node From End of List

Medium

✓ #20 Valid Parentheses

Easy

✓ #21 Merge Two Sorted Lists

Easy

✓ #22 Generate Parentheses

Medium

✓ #23 Merge k Sorted Lists

Hard

#24 Swap Nodes in Pairs

Medium

✓ #25 Reverse Nodes in k-Group

Hard

✓ #26 Remove Duplicates from Sorted Array

Easy

✓ #27 Remove Element

Easy

✓ #28 Find the Index of the First Occurrence in a String

Medium

✓ #29 Divide Two Integers

Medium

#30 Substring with Concatenation of All Words

Hard

✓ #31 Next Permutation

Medium

#32 Longest Valid Parentheses

Hard

✓ #33 Search in Rotated Sorted Array

Medium

#34 Find First and Last Position of Element in Sorted Array

Medium

✓ #35 Search Insert Position

Easy

#36 Valid Sudoku

Medium

#37 Sudoku Solver

Hard

#38 Count and Say

Medium

✓ #39 Combination Sum

Medium

✓ #40 Combination Sum II

Medium

#41 First Missing Positive

Hard

#42 Trapping Rain Water

Hard

✓ #43 Multiply Strings

Medium

#44 Wildcard Matching

Hard

✓ #45 Jump Game II

Medium

✓ #46 Permutations

Medium

#47 Permutations II

Medium

✓ #48 Rotate Image

Medium

✓ #49 Group Anagrams

Medium

#50 Pow(x, n)

Medium