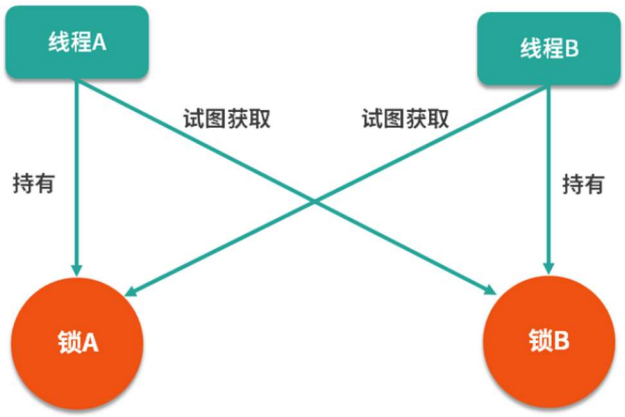
HW10: Thread Safety

Part 1: Deadlock 与 Livelock

请参考课件并查阅相关资料，分别解释 deadlock 和 livelock 并举例说明。

Deadlock：死锁是指两个或两个以上的进程（线程）在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁，这些永远在互相等待的进程（线程）称为死锁进程（线程）。



死锁的例子（图片引用自专栏 (liangliang lee.com)）：假设有线程 A 和线程 B，线程 A 持有锁 A，线程 B 持有锁 B。此时，线程 A 尝试获取锁 B，但因为线程 B 没有释放锁 B，所以线程 A 获取不到。与此同时，线程 B 也尝试获取锁 A，但因为线程 A 没有释放锁 A，所以线程 B 获取不到。这样，线程 A 和线程 B 相互持有对方需要的资源，但两者又都不释放自己手中的资源，形成了相互等待的局面，导致死锁，并且会一直等待下去。

Livelock：在活锁中，两个或多个线程不断地转移状态，而不像死锁中那样相互等待。结果是所有线程都无法执行各自的任务。一个很好的活锁例子是消息队列。当发生异常时，消息消费者会回滚事务并将消息放回队列头部，然后相同的

消息会再次从队列头部被读取，导致再次发生异常并再次放回队列头部。这种循环往复的情况会导致消费者永远无法读取到队列中的其他消息。

另一个例子是对于以下的这一份代码示例（引用自 Java 线程的死锁和活锁 - 知乎 (zhihu.com)）能看到输出结果里，两个线程都在重复的获取锁和释放锁，最终两个线程都不能完成操作。

```
public void operation1() {
    while (true) {
        tryLock(lock1, 50);
        print("lock1 acquired, trying to acquire lock2.");
        sleep(50);

        if (tryLock(lock2)) {
            print("lock2 acquired.");
        } else {
            print("cannot acquire lock2, releasing lock1.");
            lock1.unlock();
            continue;
        }

        print("executing first operation.");
        break;
    }
    lock2.unlock();
    lock1.unlock();
}

public void operation2() {
    while (true) {
        tryLock(lock2, 50);
        print("lock2 acquired, trying to acquire lock1.");
        sleep(50);

        if (tryLock(lock1)) {
            print("lock1 acquired.");
        } else {
            print("cannot acquire lock1, releasing lock2.");
            lock2.unlock();
            continue;
        }

        print("executing second operation.");
        break;
    }
    lock1.unlock();
    lock2.unlock();
}
```

```
Thread T1: lock1 acquired, trying to acquire lock2.
Thread T2: lock2 acquired, trying to acquire lock1.
Thread T1: cannot acquire lock2, releasing lock1.
Thread T2: cannot acquire lock1, releasing lock2.
Thread T2: lock2 acquired, trying to acquire lock1.
Thread T1: lock1 acquired, trying to acquire lock2.
Thread T1: cannot acquire lock2, releasing lock1.
Thread T1: lock1 acquired, trying to acquire lock2.
Thread T2: cannot acquire lock1, releasing lock2.
```

Part 2: 并发栈

并发栈，指的是支持被多个线程同时进行操作，且能保证线程安全的栈。请参考课件中的并发队列，思考如何设计并发栈（但不要求实现）。具体的，请回答以下问题：

1. 你将采用什么方式，使得多个线程可以同时进行push、pop 和 isEmpty?

通过自己的初步思考及分析、在并发栈中，push 操作相当于写者，因为它向栈中添加元素。isEmpty 操作可以看作是读者，因为它只是判断栈是否为

空，不涉及修改栈的内容。而 pop 操作不仅会修改栈中的元素，还会返回被移除的元素信息。因此，pop 操作在并发栈中既具有写者的属性（修改栈结构），也具有读者的属性（返回被移除的元素信息），需要同时考虑写者和读者的特性，确保其线程安全性。所以，在自己的初步设想中，我认为可以基于多线程中基本的读者-写者模式对并发栈进行实现。在每次进行栈的操作时、利用互斥锁在不同线程中上锁、以保证栈操作的正确性以及数据安全性。

通过进一步的查阅资料、我也认识到也可以不基于锁来实现并发栈、我了解到课本中并发队列的作者 Michael 曾设计了一个类似的堆栈，他应用了风险指针（Hazard Pointers），使得该堆栈是第一个支持内存回收的 LFDS。

同样地、我也了解到、我们也可以基于 CAS 算法（Compare and Swap，即比较再交换）进行无锁并发栈的设计与实现。（java - 深入理解 CAS 算法原理 - 个人文章 - SegmentFault 思否）

2. **你的设计中，可能存在的性能瓶颈在哪?**

在我自己的设计中、我认为可能存在的性能瓶颈在于我实际中实现的锁的粒度，如果锁粒度过大，会导致线程之间的竞争增加，从而降低性能；如果锁粒度过小，会导致线程频繁地获得和释放锁，频繁的加锁和解锁会引入一定的开销，并且可能导致性能下降。

而在 CAS 算法中、可能存在的性能瓶颈有“ABA 问题，循环时间长开销大和只能保证一个共享变量的原子操作”，但也在文章中也有写到对应的解决方案。

3. **你的设计中是否会存在 deadlock 或者 livelock?**

在我的初步设计中，使用互斥锁来保证并发栈的线程安全性。因此，在

理想情况下，我认为不应该出现死锁或活锁。然而，然使用互斥锁可以避免一些常见的死锁情况，但仍然存在一些潜在的可能性导致死锁或活锁的情况发生。比如当多个线程同时请求锁，并且它们之间存在循环依赖关系时，可能会导致死锁的发生。因此，我需要在实际的代码实践中，结合具体场景具体分析、进一步判断项目中是否会存在 deadlock 或者 livelock。