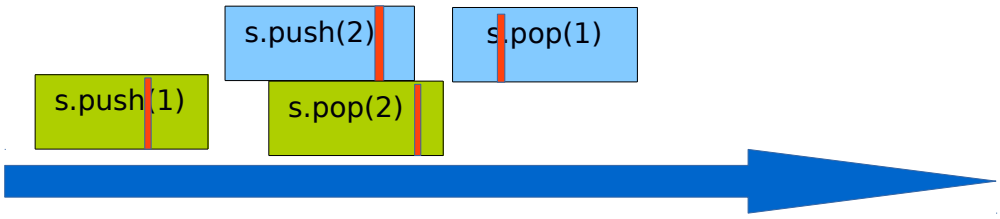
Multicore Programming Homework(6)

韩云飞(SA13226297)

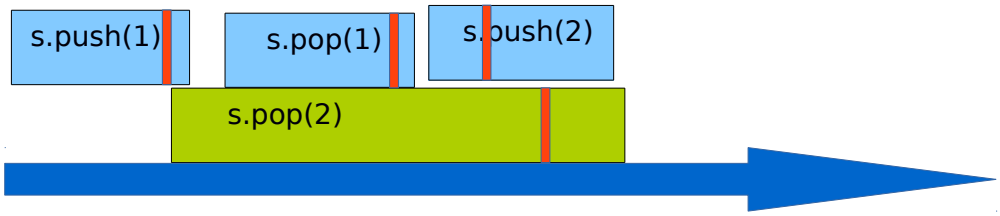
1. **Concurrent Objects:**

a) Please construct some linearizable execution histories for concurrent stacks.

example 1



example 2



b) **[optional]** Exercise 32 in Chapter 3.11

该队列使用一个items数组存放元素，为了方便起见假设数组是无界的。tail域是一个AtomicInteger，初始值为0。enq()方法通过将tail增加1来保留一个槽，然后在那个地址存入元素。注意，这两个步骤不是原子的：在tail被增加1以后和在元素被存入数组中之前存在一个时间间隔。

deq()方法读tail的值，然后以升序次序从槽0到tail遍历数组。对每一个槽，用空值与当前内容交换，并返回第一个非空元素。若所有的槽都为空，重新开始这个过程。

给出一个执行实例，说明enq()的可线性化点不可能在第15行出现。

提示：给出一个执行，其中两个enq()调用没有按照它们执行第15行代码的次序被线性化。

给出另一个例子，说明enq()的可线性化点不可能在第16行出现。

由于这是enq()中仅有的两个存储器访问操作，可以推知enq()方法没有单个可线性化点。这是否意味着enq()是不可线性化的呢？

(1) 假设 15 行为线性化点：假设执行前队列为空：

A 线程： q.enq(1)

B 线程： q.enq(2) q.deq()

A,B 线程并行执行，由于 15 行和 16 行之间执行存在时间间隔，执行的轨迹可能如下：

A 先执行至第 15 行，取得位置 0；

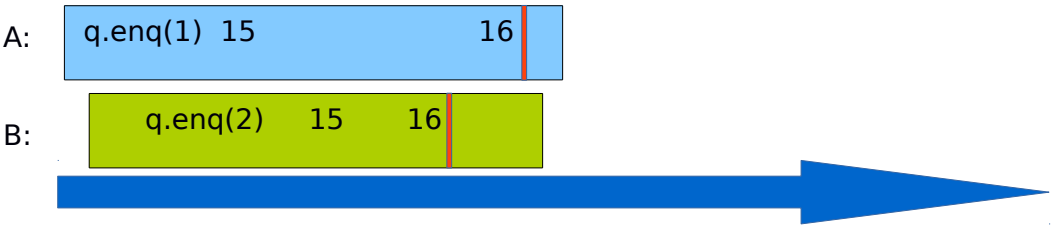
B 执行至 15 行，取得位置 1;

B 执行第 16 行，item[1] = 2;

B 执行 deq 操作，由于位置 0 现在仍为 null,导致出队元素为位置 1 的 2;

A 执行 16 行，item[0] = 1;
这时，虽然 A 先执行到线性化点，但程序 1 的执行效果却是 B 先入队，在 A 还未执行完 16 行时，出队的自然是 B 线程入队的元素 2。和在 15 行线性化结果不一致，说明 15 行不可能为线性化点。

(2) 假设 16 行为线性化点:



在上图执行历史中，虽然 A 线程的线性化点在后面到达，但由于 A 先在 15 行获得较小的 index，A 线程的元素 1 仍进入队列的前面。说明程序的执行效果和在 16 行线性化不一致，16 行不可能为线性化点。

(3)没有线性化点不代表不可以线性化，指定可线性化点只是用来说明可线性化的一种方法，二者并不等价。分析可知 enq 满足线性化的形式化定义:

定义3.6.1 如果经历H存在有一个扩展H'及一个合法的顺序经历S，并使得

L1 $complete(H')$与S等价，且

L2 若在H中方法调用m_0先于m_1，那么在S中也成立。

则称经历H是**可线性化的**。

S称作是H的一个线性化。(H可以有多个线性化。)

说明 enq 是可线性化的。

2. **Monitors and Blocking Synchronizations:** Exercise 95 : (1) in Chapter 8.7

(1)用锁和条件实现储蓄账户对象:

```
public class Account {
   private int balance;
   Lock lock;
   Condition condition;

   public Account(int balance) {
          this.balance = balance;
   }

   public void withdraw(int amount)throws InterruptedException{
                lock.lock();
                try {
                      while(balance<amount) {
                            condition.await();
                      }
                balance -= amount;
                } finally {
```

```
                    lock.unlock();
                }
        }

        public void deposit(int amount) throws InterruptedException{
                lock.lock();
                try {
                    balance +=amount;
                    condition.notifyAll();
                } finally {
                        lock.unlock();
                }
        }

}
```

2. 现在假设有两种取款方式：普通的和优先的。设计一种实现，能保证一旦有优先的取款在等待处理则普通的取款不会进行下去。

给每个 withdraw 增加一个优先权标识，在执行 withdraw 之前进行判断，如果 prior 为 HIGH 则返回；获得锁后，将 prior 置为自己的优先权，释放锁之前再将 prior 置为 LOW：

```
public class Account {
    private int balance;
    Lock lock;
    Condition condition;

    enum Prior { LOW, HIGH};
    Prior prior = Prior.LOW;;

    public Account(int balance) {
        this.balance = balance;
    }


public void withdraw(int amount, Prior prior) throws
                                    InterruptedException{
        if ( prior == Prior.LOW && this.prior == Prior.HIGH) {
            return;
        }
        lock.lock();
        try {
            this.prior = prior;
            while(balance<amount) {
                condition.await();
            }
```

```
                balance -= amount;
                } finally {
                        this.prior = Prior.LOW;
                        lock.unlock();
                }
}
                . . . . . .
}
```

3. 现在增加一个transfer()方法，它将总存款从一个账户转账到另一个账户：

```
void transfer(int k, Account reserve) {
  lock.lock();
  try {
    reserve.withdraw(k);
    deposit(k);
  } finally {lock.unlock();}
}
```

给定10个账户的一个集合，它们的结余是未知的。在1:00时，n个线程均设法把100美元从另一个账户转账到自己的账户。在2:00时，一个Boss线程给每个账户存1000美元。每个在1:00被调用的转账方法都一定会返回吗？

会返回，最极端的情况是：某个账户余额不足 100$，其他账户都要从其转帐，需要从一点等待到 2 点。但 2 点存入的 1000$可以满足其他账户的 withdraw 操作，其他账户会依次获得锁，转账成功并返回。