### 1. 实验一

观察下面程序的运行结果，分析造成这种结果的原因，提出**一种**解决方案并写出代码清单。

public class Run {
  
 public static void main(String[] args) {
  
 try {
  
 DealThread t1 = new DealThread();
  
 t1.setFlag("a");
  
  
 Thread thread1 = new Thread(t1);
  
 thread1.start();
  
  
 Thread.sleep(100);
  
  
 t1.setFlag("b");
  
 Thread thread2 = new Thread(t1);
  
 thread2.start();
  
 } catch (InterruptedException e) {
  
 // TODO Auto-generated catch block
  
 e.printStackTrace();
  
 }
  
 }
  
}
  
  
class DealThread implements Runnable {
  
 public String username;
  
 public Object lock1 = new Object();
  
 public Object lock2 = new Object();
  
  
 public void setFlag(String username) {
  
 this.username = username;
  
 }
  
  
 @Override
  
 public void run() {
  
 if (username.equals("a")) {
  
 synchronized (lock1) {
  
 try {
  
 System.out.println("username = " + username);
  
 Thread.sleep(3000);
  
 } catch (InterruptedException e) {
  
  
 e.printStackTrace();
  
 }
  
 synchronized (lock2) {
  
 System.out.println("按lock1->lock2代码顺序执行了");
  
 }
  
 }
  
 }
  
 if (username.equals("b")) {
  
 synchronized (lock2) {
  
 try {
  
 System.out.println("username = " + username);
  
 Thread.sleep(3000);
  
 } catch (InterruptedException e) {
  
 e.printStackTrace();
  
 }
  
 synchronized (lock1) {
  
 System.out.println("按lock2->lock1代码顺序执行了");
  
 }
  
 }
  
 }
  
 }
  
}



这段程序会产生死锁的原因是两个线程在执行时，互相持有对方所需的锁资源，同时又等待对方释放自己所需的锁资源，导致彼此无法继续执行下去，从而陷入死锁状态。

具体来说，当t1的username为"a"时，线程会先获取lock1锁，然后在执行过程中休眠3秒钟。而在这段时间内，主线程将t1的username设置为"b"，创建了另一个线程thread2，并启动它。thread2在执行时，需要获取lock2锁。

由于t1的username被修改为"b"，因此在t1醒来之后，它会去获取lock2锁。但是此时，lock2已经被thread2持有，而thread2正在等待获取lock1锁。这样，t1持有lock1锁且等待lock2锁，而thread2持有lock2锁且等待lock1锁，它们相互等待对方释放锁资源，从而形成死锁。

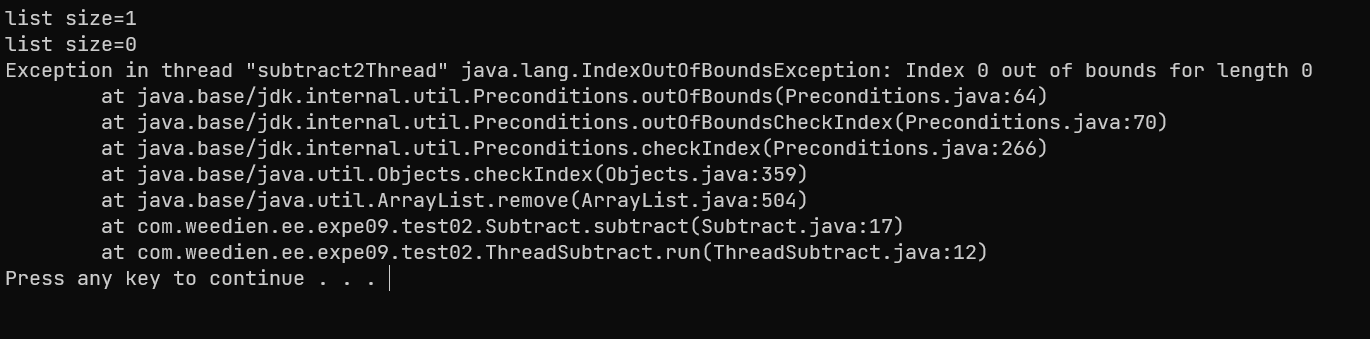
为了解决这个问题，我们可以尝试使用统一的锁顺序来避免死锁的发生。可以将lock1和lock2的获取顺序保持一致，即在两个地方都按照相同的顺序获取锁。



### 2. 实验二

请问分析出现异常的原因并用文字说明。如何解决这个问题呢？请问此时有几条线程，它们的状态分别都是什么呢？注意测试程序中启动了一个Add线程，两个sub线程。

//往list中增加一个字符串元素
  
public class Add {
  
 private String lock;
  
 public Add(String lock){
  
 this.lock=lock;
  
 }
  
 public void add(){
  
 synchronized(lock){
  
 ValueObject.list.add("anyString");
  
 lock.notifyAll();
  
 }
  
 }
  
}
  
  
// 删除list中的一个元素
  
public class Subtract {
  
 private String lock;
  
  
 public Subtract(String lock) {
  
 this.lock = lock;
  
 }
  
  
 public void subtract() {
  
 try {
  
 synchronized (lock) {
  
 if (ValueObject.list.size() == 0) {
  
 lock.wait();
  
 }
  
 ValueObject.list.remove(0);
  
 System.out.println("list size=" + ValueObject.list.size());
  
 }
  
 } catch (InterruptedException e) {
  
 e.printStackTrace();
  
 }
  
 }
  
}
  
  
// 增加List中元素的线程
  
public class ThreadAdd extends Thread {
  
 private Add p;
  
  
 public ThreadAdd(Add p) {
  
 this.p = p;
  
 }
  
  
 public void run() {
  
 p.add();
  
 }
  
}
  
  
// 删除list中的一个元素的线程
  
public class ThreadSubtract extends Thread {
  
 private Subtract c;
  
  
 public ThreadSubtract(Subtract c) {
  
 this.c = c;
  
 }
  
  
 public void run() {
  
 c.subtract();
  
 }
  
}
  
  
public class Run {
  
 public static void main(String[] args) throws InterruptedException {
  
 String lock = new String("");
  
 Add add = new Add(lock);
  
 Subtract sub = new Subtract(lock);
  
 ThreadAdd addThread = new ThreadAdd(add);
  
 ThreadSubtract subThread1 = new ThreadSubtract(sub);
  
 subThread1.setName("subtract1Thread");
  
 ThreadSubtract subThread2 = new ThreadSubtract(sub);
  
 subThread2.setName("subtract2Thread");
  
 subThread1.start();
  
 subThread2.start();
  
 Thread.sleep(1000);
  
 addThread.start();
  
 }
  
}
  
  
import java.util.ArrayList;
  
import java.util.List;
  
  
public class ValueObject {
  
 public static List list = new ArrayList();
  
}



问题：

* 数组越界访问
* 原因：add线程在添加一个元素后唤醒了两个sub线程，当第一个sub线程移除了该元素之后，第二个sub线程又执行了一次remove操作，导致抛出异常。
* 分析：sub1获得锁，但此时判断数组为空，进入等待状态；sub2无法获取锁，陷入等待；add获得锁，增加一个元素，唤醒两个sub线程；sub2获得锁，判断数组不为空，进行remove操作，释放锁；sub2获得锁，恢复到之前执行的位置，即if判断之后，再次进行remove操作；报错。（每一个线程都会记录自己的执行上下文）

解决方案：

1. 修改subtract()方法，在调用lock.wait()之前添加条件判断，判断列表大小是否为0，如果为0则等待。使用while循环替换if来确保在被唤醒后再次检查条件。

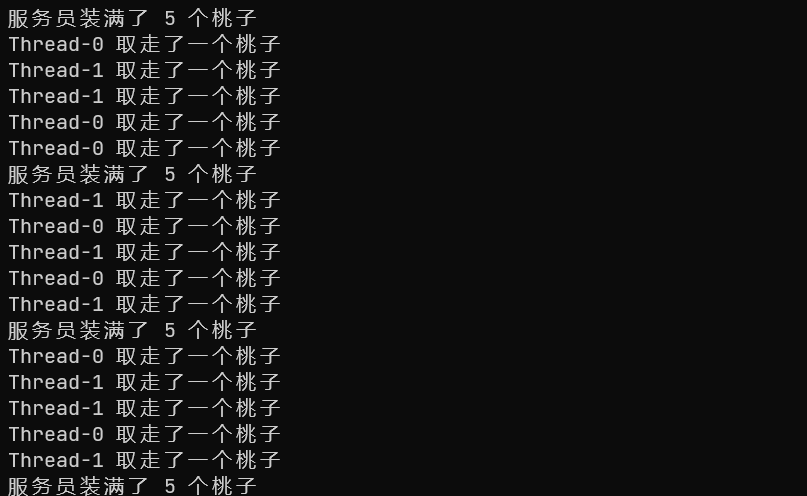
线程状态：

1. addThread线程：该线程在启动后会执行p.add()方法，然后结束。
2. subThread1线程和subThread2线程：这两个线程在启动后会执行c.subtract()方法，然后进入等待状态（等待被唤醒）。

### 3. 实验三

孙悟空在花果山举办热热闹闹的蟠桃会。桃子放在果盘Plate里，客人Guest每次要从果盘中取出一个桃子，当果盘空了以后，服务员Servant就会重新给果盘装满桃子。当果盘为空时，客人无法取出桃子，应该先通知服务员为果盘装满桃子。当果盘不空时，服务员不用为果盘装满桃子，服务员应用先通知客人从果盘中取出桃子。一个果盘最多可以盛放5个桃子。请用多线程的知识模拟这个场景。给出必要的注释。

import java.util.concurrent.locks.Condition;
  
import java.util.concurrent.locks.Lock;
  
import java.util.concurrent.locks.ReentrantLock;
  
  
public class PeachParty {
  
 private static final int MAX\_PEACHES = 5;
  
 private int peachCount = 0;
  
 private final Lock lock = new ReentrantLock();
  
 private final Condition guestCondition = lock.newCondition();
  
 private final Condition servantCondition = lock.newCondition();
  
  
 public void guestTakePeach() {
  
 lock.lock();
  
 try {
  
 // 如果果盘为空，则等待服务员装满桃子
  
 while (peachCount == 0) {
  
 try {
  
 // 通知服务员果盘为空
  
 servantCondition.signal();
  
 guestCondition.await();
  
 } catch (InterruptedException e) {
  
 e.printStackTrace();
  
 }
  
 }
  
 // 从果盘中取出一个桃子
  
 peachCount--;
  
 System.out.println(Thread.currentThread().getName() + " 取走了一个桃子");
  
 } finally {
  
 lock.unlock();
  
 }
  
 }
  
  
 public void servantRefillPeaches() {
  
 lock.lock();
  
 try {
  
 // 如果果盘未满，则等待客人取走桃子
  
 while (peachCount > 0) {
  
 try {
  
 servantCondition.await();
  
 } catch (InterruptedException e) {
  
 e.printStackTrace();
  
 }
  
 }
  
 // 装满果盘
  
 peachCount = MAX\_PEACHES;
  
 System.out.println("服务员装满了 " + peachCount + " 个桃子");
  
 // 通知客人可以继续取桃子
  
 guestCondition.signalAll();
  
 } finally {
  
 lock.unlock();
  
 }
  
 }
  
  
 public static void main(String[] args) {
  
 PeachParty peachParty = new PeachParty();
  
  
 Thread guest1 = new Thread(() -> {
  
 while (true) {
  
 peachParty.guestTakePeach();
  
 try {
  
 Thread.sleep(1000); // 模拟客人吃桃子的时间
  
 } catch (InterruptedException e) {
  
 e.printStackTrace();
  
 }
  
 }
  
 });
  
  
 Thread guest2 = new Thread(() -> {
  
 while (true) {
  
 peachParty.guestTakePeach();
  
 try {
  
 Thread.sleep(1000); // 模拟客人吃桃子的时间
  
 } catch (InterruptedException e) {
  
 e.printStackTrace();
  
 }
  
 }
  
 });
  
  
 Thread servant = new Thread(() -> {
  
 while (true) {
  
 peachParty.servantRefillPeaches();
  
 try {
  
 Thread.sleep(2000); // 模拟服务员装满桃子的时间
  
 } catch (InterruptedException e) {
  
 e.printStackTrace();
  
 }
  
 }
  
 });
  
  
 guest1.start();
  
 guest2.start();
  
 servant.start();
  
 }
  
}

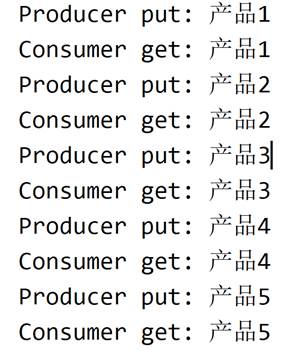


### 4. 实验四

有两条线程P和线程C。其中P线程负责向缓冲区Buff存放一个产品(String类型的数据)，C线程每次只会生成从缓冲区拿走一个产品。利用线程同步和通信机制，模拟实现多轮生产及消费。



参考如下运行界面：



import java.util.LinkedList;
  
  
class Buffer {
  
 private LinkedList<String> buffer;
  
 private int maxSize;
  
  
 public Buffer(int maxSize) {
  
 this.maxSize = maxSize;
  
 buffer = new LinkedList<>();
  
 }
  
  
 public synchronized void put(String product) throws InterruptedException {
  
 while (buffer.size() == maxSize) {
  
 wait(); // 等待缓冲区有空位置
  
 }
  
  
 buffer.add(product);
  
 System.out.println("Producer put: " + product);
  
  
 notifyAll(); // 通知等待的线程可以进行操作
  
 }
  
  
 public synchronized String get() throws InterruptedException {
  
 while (buffer.isEmpty()) {
  
 wait(); // 等待缓冲区有产品
  
 }
  
  
 String product = buffer.removeFirst();
  
 System.out.println("Consumer get: " + product);
  
  
 notifyAll(); // 通知等待的线程可以进行操作
  
  
 return product;
  
 }
  
}
  
  
class Producer implements Runnable {
  
 private Buffer buffer;
  
  
 public Producer(Buffer buffer) {
  
 this.buffer = buffer;
  
 }
  
  
 @Override
  
 public void run() {
  
 try {
  
 for (int i = 1; i <= 5; i++) {
  
 String product = "产品" + i;
  
 buffer.put(product);
  
 }
  
 } catch (InterruptedException e) {
  
 e.printStackTrace();
  
 }
  
 }
  
}
  
  
class Consumer implements Runnable {
  
 private Buffer buffer;
  
  
 public Consumer(Buffer buffer) {
  
 this.buffer = buffer;
  
 }
  
  
 @Override
  
 public void run() {
  
 try {
  
 for (int i = 1; i <= 5; i++) {
  
 buffer.get();
  
 }
  
 } catch (InterruptedException e) {
  
 e.printStackTrace();
  
 }
  
 }
  
}
  
  
public class Main {
  
 public static void main(String[] args) {
  
 Buffer buffer = new Buffer(1); // 缓冲区大小为1
  
  
 Producer producer = new Producer(buffer);
  
 Consumer consumer = new Consumer(buffer);
  
  
 Thread producerThread = new Thread(producer);
  
 Thread consumerThread = new Thread(consumer);
  
  
 producerThread.start();
  
 consumerThread.start();
  
  
 try {
  
 producerThread.join();
  
 consumerThread.join();
  
 } catch (InterruptedException e) {
  
 e.printStackTrace();
  
 }
  
 }
  
}

