Multicore Programming Homework(5)

韩云飞(SA13226297)

1. **Spin Locks and Contentions** : Please finish the following exercise in Chapter 7.11 of the text book “The art of multiprocessor programming” which can be downloaded from the teaching assistant system:
   1. Exercise 85

在线程释放锁时， 重用自己的结点而非该线程前驱节点，但由于前驱结点此刻不再使用其QNode，而后继结点仍然保持引用，就会导致前驱结点无法得到释放，线程数目过多时，就会造成内存的极大浪费，甚至会影响程序的正确执行。

* 1. Exercise 91

**TASLock:**

public Boolean isLocked() {

return state.get();

}

**CLHLock:**

public Boolean isLocked(){

if(myNode.locked==true&&myPred.locked==false)

return true;

return false;

}

**MCSLock:**

public Boolean isLocked(){

if(myNode.next!=NULL&&myNode.locked==false)

return true;

return false;

}

* 1. Please give the basic idea of the spin locks you learned from the class

spin lock的基本思想是在无法获得锁时，为了减少锁的延迟时间，不去阻塞自己，而是对锁的状态进行反复测试。

**(1)Test-And-Set**

- Lock

Lock is free: value is false

Lock is taken: value is true

Acquire lock by calling **TAS**(Swap true with current value, Return value tells if prior value was true or false)

If result is false, you win; If result is true, you lose

- Unlock

Release lock by writing false

**(2) Test-And-Test-And-Set**

- Lurking stage

Wait until lock “looks” free

Spin while read returns true (lock taken)

- Pouncing state

As soon as lock “looks” available

Read returns false (lock free)

Call TAS to acquire lock

If TAS loses, back to lurking

**(3)Exponential Backoff**

If I fail to get lock

- wait random duration before retry

- Each subsequent failure doubles expected wait

**Queue Lock的basic idea**

- Avoid useless invalidations

- By keeping a queue of threads

- Each thread notifies next in line, without bothering the others

**(4) Alock**

n个线程共享一个大小为n的flag数组(标识锁的状态)，但在任意给定时间内，由于每个线程都是都是在一个数组存储单元的本地cache副本上旋转，大大降低了无效流量; 第i+1个线程的锁状态由第i个线程来通知唤醒。

**(5) CLHLock**

- 每个线程通过一个线程局部变量pred指向其前驱，公共的tail域指向最近加入 到队列中的结点;

- Lock： 线程对tail域调用getAndset(),使自己的结点成为队列的尾部，同时获 得一个指向其前驱QNode的引用，最后线程在其前驱结点的locked上旋转，直 到前驱释放该锁;

- Unlock：将其locked域设为false，然后重新使其前驱结点的QNode作为新结 点以便将来的线程访问。

**(6) MCSLock**

- 与CLHLock类不同，锁链表是显式的而不是虚拟的，整个链表通过QNode对象 里的next域体现;

- Lock：要获得锁，线程把自己的QNode添加到链表的尾部，如果队列原先不为 空，则将前驱QNode的next域设置为指向它自己的QNode。然后在它自己的 QNode的locked域上自旋，直到前驱将该域设为false;

- Unlock： 检查结点的next域是否为空，如果是，则要么不存在其他线程正在争 用这个锁，要么存在一个正在争用但运行很慢的线程。在任一情况下，一旦出现 了后继，unlock方法将它后继的locked域设置为false，表明锁是空闲的。

**(7)Abortable CLH Lock**

When a thread gives up, removing node in a wait-free way is hard.

let successor deal with it.

**(8)Time-out Lock**

若一个线程超时，则该线程将它的结点标记为已放弃。这样该线程在队列中的后继将会注意到它正在自旋的结点已经被放弃，于是开始在被放弃结点的前驱上自旋。

1. **Concurrent Objects:**

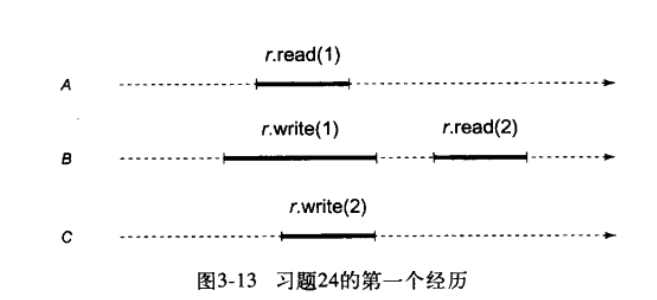
静态一致性：任一时刻，若对象变为静态的，则到此刻为止的执行，等价于目前已完成的所有方法调用的某种顺序执行。

顺序一致性：方法调用应该呈现出按照程序次序调用的执行效果。

线性一致性(Linearizability)，或称原子一致性或严格一致性指的是程序在执行的历史中在存在可线性化点P的执行模型，这意味着一个操作将在程序的调用和返回之间的某个点P起作用。这里“起作用”的意思是被系统中并发运行的所有其他线程所感知。

可线性化性所隐含的基本思想就是每一个并发经历都等价于一个顺序经历

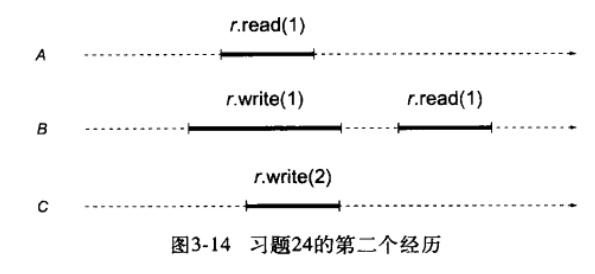
* 1. Are the histories in Figures 3.13 and 3.14 of Chapter 3.11 linearizable or not? Please give your answers and justify them.



可线性化，存在一个合法的顺序经历S：

<r.write(1), B> → <r.read(1), A> → <r.write(2), C> → <r.read(2),B>,

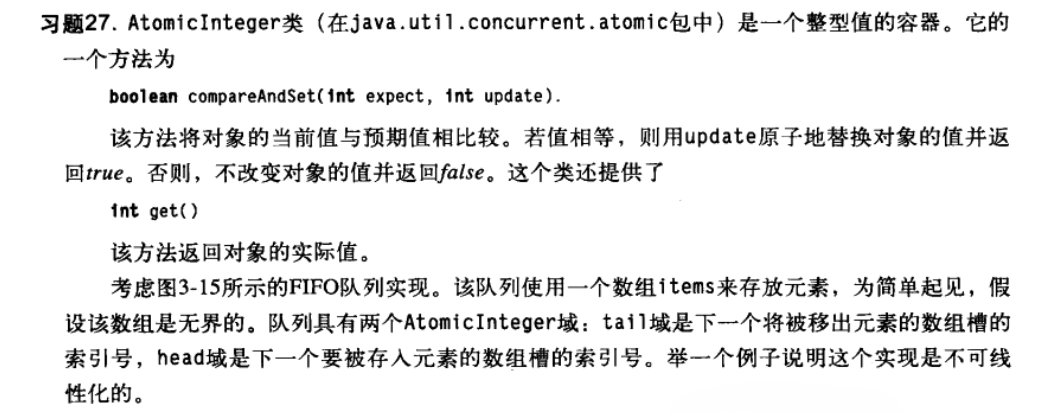
根据可线性化的形式化定义，可知该经历是可线性化的。



可线性化，各线程的可线性化点如上图中的竖线所示，这时的一个线性化经历为：

<r.write(2), C> → <r.write(1), B> → <r.read(1), A> → <r.read(1),B>

* 1. Exercise 27 in Chapter 3.11



eg: A线程执行enq至准备赋items[slot] = x，但还未完成时， B线程执行deq，value = items[slot]，这是线程B会使用错误的head完成出队操作。在这种情况下,运行结果并未按照队列数据结构的线性变化进行, 因而未能满足可线性化的要求,因此这个实现是不可线性化的。