《同步与多核》章节练习

1.请说明为什么自旋锁不适用于单核系统，但在多核系统下可以正常工作？ 对该问题有何种解决方案?

- 单核系统中，当线程拿到自旋锁并且发生抢占或中断时，它将一直持有锁，被切换到的其他线程无法拿到锁，会一直空占cpu直到他的时间片结束，等待时间过长
- 多核系统中拿到锁的线程可以尽快执行完毕，其他线程等待时间不长
- 解决方案：在单核系统中，当线程拿到自旋锁后关闭中断，且不允许抢占，直到放锁

2.在课程中，我们了解了对读者友好的读写锁定的情况。 这种锁定会导致写者饿死，其原因是？ 请提供一种对写者友好的读写锁定设计（避免写者饥饿）。

- 原因是对读者友好的读写锁会在有读者正在读时，不让写者进入临界区，而读者可以无限进入临界区，理论上读者如果一直来，写者会饿死
- 写者友好的读写锁：无论写者是正在写还是等待，后来的读者都不能进入临界区，而其他写者可以进入临界区

3.假设有一个包含四个处理器和五个相同资源的系统。 每个处理器在有限的持续时间内最多使用两个资源。 请说明为什么该系统没有死锁（所有处理器最终都将使用两个资源)

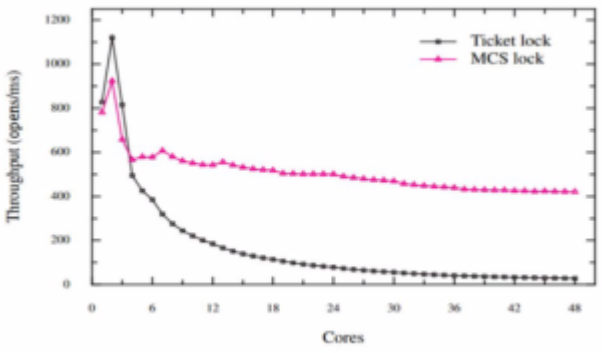
设处理器数量为x，资源（锁）数为y，每个处理器最多同时持有资源数为z，只要保证 x * (z-1) < y，就永远不会出现死锁，因为不会出现循环等待的情况，总有处理器可以拿到自己想要的资源，然后在持续时间结束后放掉

4.排号锁可能会导致在高竞争情况下性能降低。下图描述了使用排号锁和MCS锁的open()系统调用的性能。请指出使得排号锁的吞吐量下降的代码行。 另外，MCS锁是如何达到高可扩展性的呢？ （左图为排号锁代码）

```
struct lock {
        volatile unsigned owner;
        volatile unsigned next;
}

void lock_init(struct lock *lock) {
        /* Initialize ticket lock */
        lock->owner = 0;
        lock->next = 0;
}

void lock(struct lock *lock) {
        volatile unsigned my_ticket =
                atomic_FAA(&lock->next, 1);
        while (lock->owner != my_ticket) {
                /* Busy looping */
        }
        barrier();
}
```



- 使排号锁的吞吐量下降的代码行是while (lock->owner != my_ticket) { }
- MCS锁使用了一个链表来链接所有等待的线程，链表的每个节点都是一个线程，并且他们都用一个布尔值locked表示自己是否被锁定，并且在自己unlock时通知链表中的下一个节点。基于TAS的自旋锁在多核系统上不具有很好的可伸缩性。每一次TAS操作都会写入cache，那么所有线程的这条cache线都会失效，当线程数多的时候会导致大量的cache一致性流量。MCS锁的优点在于每一个线程只自旋自己的变量，因此自旋完全可以在自己所在的核心的L1中完成，只写一个私有缓存行，不再会高频竞争全局缓存行，完全没有任何cache一致性的问题，也不会产生内存访问

  参考：https://www.cnblogs.com/zhengsyao/p/mcs_lock_scalable_spinlock.html?utm_source=tuicool

5.在课程中，我们学习了NUMA-aware cohort lock。 尽管它可以提高锁在NUMA架构中的性能，但它会引入了公平性问题。请给出对此问题的问题的合理解决方案。

- 公平性问题在于全局锁在一段时间内只在一个结点内部传递
- 解决方案：持有全局锁的结点运行一段时间后主动放锁，使所有结点轮流获得全局锁

6.假设小明在ARM 处理器上实现自旋锁，并且发现关键部分没有通过锁定得到很好的保护。 在了解了弱顺序一致性后，他知道要在代码中添加内存屏障。 请在适当的位置帮助小明添加barrier()。

```
void lock(struct spinlock* lock) {
    /* locking */
    barrier();
}


void unlock(struct spinlock* lock) {
    barrier();
    /* unlocking */
}
```