Multicore Programming Homework(3)

韩云飞(SA13226297)

> Please implement Peterson lock and Test-And-Set lock which you learned from the class, and test your lock implementation using the shared counter and explain the usage of the `volatile` keyword.
> Then try to compare the performance with the locking mechanisms provided by Java (eg. **synchronized** blocks and the **lock** library). Note that spin locks assume that the number of your processors must not be smaller than the number of threads. You can test your lock implementation with the number of threads that exceeds the number of processors and see what will happen.

Solve:

首先为Peterson lock和Test-And-Set lock两种实现建立共同的抽象类：

```
abstract class Lock extends Thread{
    static int counter  = 0;  //shared counter
    abstract void lock();
    abstract void unlock();
}
```

Peterson lock的实现：

```
/*
 * Peterson Algorithm that give prior to other
```

```
 */
class PetersonLock extends Lock{
    private static boolean flag[] = {false, false};
    private static volatile int victim = -1;   //Neither of both
    private final int id;
    private int round = 1;

    public PetersonLock(int id, int round) {
        this.id = id;
        this.round = round;
    }

    void lock() {
        flag[id] = true;
        victim = id;
        while(flag[(id + 1) % 2] && victim == id) {
            System.out.println("Thread " + id + " is waiting...");
        }
    }

    void unlock() {
        flag[id] = false;
    }

    @Override
    public void run() {
        for(int i = 0; i < round; i++) {
            lock();
            counter =  ( id == 0 ? counter +1 : counter - 1);
            System.out.println("Thread " + id + " modify counter: " + counter)
;
            unlock();
        }
    }
}
```

Test-And-Set lock的实现：

```
/*
 * Test-and-Set Spin Lock
 */

class TASLock  extends Lock{
    static class AtomicBoolean {
        boolean value;
        public AtomicBoolean(boolean value) {
            this.value = value;
```

```
        }
        public synchronized boolean getAndSet(boolean newValue) {
           boolean prior = value;
           value = newValue;
           return prior;
         }
        public void set(boolean value) {
            this.value = value;
        }
     }

    private  static AtomicBoolean state = new TASLock.AtomicBoolean(false);
    private final int id;
    private int round = 1;

    public TASLock(int id, int round) {
        this.id = id;
        this.round = round;
    }

     void lock() {
      while (state.getAndSet(true)) {
          System.out.println("Thread " + id + " is waiting...");
      }
     }

     void unlock() {
      state.set(false);
     }

        @Override
        public void run() {
            for(int i =0; i < round; i++) {
                lock();
                counter +=id;
                System.out.println("Thread " + id + " modify counter: " + coun
ter);
                unlock();
            }
        }
}
```

测试程序：

```
public class LockDemo {
    public static void main(String[] args) throws InterruptedException {
//      Lock  lock0 = new PetersonLock(0, 1);
//      Lock  lock1 =  new PetersonLock(1, 1);
```

```
        Lock lock0 = new PetersonLock(0, 3);
        Lock lock1 = new PetersonLock(1, 3);
        System.out.println("PeterSon Lock:");
        lock0.start();
        lock1.start();
        lock0.join();
        lock1.join();

        System.out.println("\n\nTest-and-Set Lock:");
//      lock0 = new TASLock(2, 1);
//      lock1 = new TASLock(3, 1);
        lock0 = new TASLock(2, 3);
        lock1 = new TASLock(3, 3);
        lock0.start();
        lock1.start();
    }
}
```

如果将两种lock中的run次数设置为1时的典型输出结果为：

```
PeterSon Lock:
Thread 0 is waiting...
Thread 1 modify counter: -1
Thread 0 is waiting...
Thread 0 modify counter: 0

Test-and-Set Lock:
Thread 3 is waiting...
Thread 2 modify counter: 2
Thread 3 is waiting...
Thread 3 modify counter: 5
```

而当lock次数设置为3时，Peterson lock的效果要差一些，经常出现两个线程同时阻塞的情况：

```
PeterSon Lock:
Thread 1 is waiting...
Thread 0 modify counter: 1
Thread 0 is waiting...
Thread 1 is waiting...
Thread 0 is waiting...
Thread 1 modify counter: 0
Thread 0 is waiting...
Thread 1 is waiting...
Thread 0 modify counter: 1
Thread 1 is waiting...
Thread 0 is waiting...
Thread 1 modify counter: 0
Thread 0 is waiting...
Thread 1 is waiting...
Thread 0 modify counter: 1
```

```
Thread 1 is waiting...
Thread 1 modify counter: 0

Test-and-Set Lock:
Thread 2 modify counter: 2
Thread 3 is waiting...
Thread 2 modify counter: 4
Thread 2 modify counter: 6
Thread 3 is waiting...
Thread 3 modify counter: 9
Thread 3 modify counter: 12
Thread 3 modify counter: 15
```

volatile 关键字的作用

没有`valatile`会发生什么：

- 在本次线程内，当读取一个变量时，为提高存取速度，编译器优化时有时会先把变量读取到一个寄存器中；以后再取变量值时，就直接从寄存器中取值；
- 当变量值在本线程里改变时，会同时把变量的新值copy到该寄存器中，以便保持一致;
- 当变量在因别的线程等而改变了值，该寄存器的值不会相应改变，从而造成应用程序读取的值和实际的变量值不一致;
- 当该寄存器在因别的线程等而改变了值，原变量的值不会改变，从而造成应用程序读取的值和实际的变量值不一致;

为了避免这些情况造成的数据不一致，需要使用`volatile`关键字标识变量：

- 要求使用`volatile`声明的变量的值的时候，编译器对访问该变量的代码就不再进行优化，从而可以提供对特殊地址的稳定访问。
- 系统总是重新从它所在的内存读取数据，即使它前面的指令刚刚从该处读取过数据。而且读取的数据立刻被保存。

java中Lock 与 Synchronized的区别

数据同步需要依赖锁，Java中锁机制有两种实现方式: synchronized是在软件层面依赖JVM，而Lock则是在硬件层面依赖特殊的CPU指令实现。

1. 性能区别

- synchronized
  在资源竞争不是很激烈的情况下，偶尔会有同步的情形下，synchronized是很合适的。原因在于，编译程序通常会尽可能的进行优化synchronize，另外可读性非常好，不需要显式加锁与解锁。

- ReentrantLock
  ReentrantLock提供了多样化的同步，比如有时间限制的同步，可以被Interrupt的同步（synchronized的同步是不能Interrupt的）等。在资源竞争不激烈的情形下，性能稍微比synchronized差点点。但是当同步非常激烈的时候，synchronized的性能一下子能下降好几十倍。而ReentrantLock确还能维持常态。

所以，我们写同步的时候，优先考虑synchronized，如果有特殊需要，再进一步优化。ReentrantLock，不仅不能提高性能，还可能带来灾难。总结就是：

> 在并发高时，lock性能优势很明显，在低并发时，synchronized也能取得优势。

2. 实现原理

synchronized：对象加锁

所有对象都自动含有单一的锁，JVM负责跟踪对象被加锁的次数。如果一个对象被解锁，其计数变为0。在任务（线程）第一次给对象加锁的时候， 计数变为1。每当这个相同的任务（线程）在此对象上获得锁时，计数会递增。 只有首先获得锁的任务（线程）才能继续获取该对象上的多个锁。每当任务离开时，计数递减，当计数为0的时候，锁被完全释放。 `synchronized` 就是基于这个原理,同时 `synchronized` 靠某个对象的单一锁技术的次数来判断是否被锁，所以无需（也不能）人工干预锁的获取和释放。 如果结合方法调用时的栈和框架， 不难推测出 `synchronized` 原理是基于栈中的某对象来控制一个框架，所以对于 `synchronized` 有常用的优化是锁对象不锁方法。实际上 `synchronized` 作用于方法时，锁住的是 `this` ，作用于静态方法/属性时，锁住的是存在于永久带的 `CLASS` ，相当于这个 `CLASS` 的全局锁，锁作用于一般对象时，锁住的是对应代码块。在 `HotSpot` 中JVM实现中，锁有个专门的名字： `对象监视器` 。

当多个线程同时请求某个对象监视器时，对象监视器会设置几种状态用来区分请求的线程：

- `Contention List` ：所有请求锁的线程将被首先放置到该竞争队列
- `Entry List` ：Contention List中那些有资格成为候选人的线程被移到Entry List
- `Wait Set` ：那些调用wait方法被阻塞的线程被放置到Wait Set
- `OnDeck` ：任何时刻最多只能有一个线程正在竞争锁，该线程称为OnDeck
- `Owner` ：获得锁的线程称为Owner
- `!Owner` ：释放锁的线程

Lock：基于栈中的框架而不是对象级别

> Lock不同于synchronized面向对象，它基于栈中的框架而不是某个具体对象，所 以Lock只需要在栈里设置锁的开始和结束（lock和unlock）的地方就行，不用关心框架大小对象的变化等等。这么做的好处是Lock能提供无条件的、可轮询的、定时的、可中断的锁获取操作，相对于synchronized来说，synchronized的锁的获取是释放必须在一个模块里，获取和释放的顺序必须相反，而Lock则可以在不同范围内获取释放，并且顺序无关。java.util.concurrent.locks下的锁类很类似,依赖于java.util.concurrent.AbstractQueuedSynchronizer，它们把所有的Lock接口操作都转嫁到Sync类上，这个类继承了AbstractQueuedSynchronizer，它同时还包含子2个类：NonfairSync 和FairSync 从名字上可以看的出是为了实现公平和非公平性。AbstractQueuedSynchronizer中把所有的的请求线程构成一个队列(一样也是虚拟的)。

threads数目超过processors会发生什么

由于之前写的Peterson只适用与两个线程的情况，所以yongTASLock进行测试：
修改测试代码为：

```
for(int i=0; i < 10; i++) {
    new Thread(new TASLock(i, 2), "Thread").start();
}
```

输出结果为：

```
Test-and-Set Lock:
Thread 0 modify counter: 0
Thread 0 modify counter: 0
Thread 1 is waiting...
Thread 1 modify counter: 1
Thread 1 modify counter: 2
Thread 4 modify counter: 6
Thread 4 modify counter: 10
Thread 3 is waiting...
Thread 3 modify counter: 13
Thread 2 is waiting...
Thread 2 is waiting...
Thread 2 is waiting...
Thread 2 is waiting...
Thread 2 is waiting...
........
Thread 2 is waiting...
Thread 6 is waiting...
Thread 3 modify counter: 16
Thread 5 is waiting...
Thread 5 modify counter: 21
Thread 5 modify counter: 26
Thread 6 is waiting...
```

```
Thread 6 modify counter: 32
Thread 2 is waiting...
Thread 7 is waiting...
Thread 7 is waiting...
Thread 7 is waiting...
Thread 2 is waiting...
Thread 2 is waiting...
Thread 2 is waiting...
........
Thread 2 is waiting...
Thread 2 is waiting...
Thread 2 is waiting...
Thread 2 is waiting...
Thread 2 is waiting...
Thread 9 is waiting...
Thread 9 is waiting...
.........
Thread 9 is waiting...
Thread 6 modify counter: 38
Thread 9 modify counter: 47
Thread 9 modify counter: 56
Thread 2 is waiting...
Thread 7 is waiting...
Thread 7 modify counter: 63
Thread 7 modify counter: 70
Thread 8 is waiting...
Thread 8 modify counter: 78
Thread 8 modify counter: 86
Thread 2 modify counter: 88
Thread 2 modify counter: 90
```

发现这时程序大部分时间用来阻塞线程，执行效率变得很低。说明多核程序更适合执行在多处理器环境下，如果处理器数目受限，程序执行效率不但不会提高，反而会下降。