

关于编程路上的一些杂谈 多线程中锁的秘密 (二)

作者: whxiaobu

原文链接: https://ld246.com/article/1484365809330

来源网站:链滴

许可协议: 署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

来源:极乐科技知乎专栏

作者: 知秋

博客:一叶知秋

接上篇关于编程路上的一些杂谈 由线程的通信原理想到的(一), 其实已经讨论一些锁的实现了, 这里深入一下, 把问题讲明白。

底层实现原理

有volatile变量修饰的共享变量进行写操作的时候会多出第二行汇编代码,通过查IA-32架构软件开发手册可知,Lock前缀的指令在多核处理器下会引发了两件事情。

- 1. 将当前处理器缓存行的数据写回到系统内存。
- 2. 这个写回内存的操作会使在其他CPU里缓存了该内存地址的数据无效。

为了提高处理速度,处理器不直接和内存进行通信,而是先将系统内存的数据读到内部缓存(L1,L2 其他)后再进行操作,但操作完不知道何时会写到内存。如果对声明了volatile的变量进行写操作,J M就会向处理器发送一条Lock前缀的指令,将这个变量所在缓存行的数据写回到系统内存。但是,就写回到内存,如果其他处理器缓存的值还是旧的,再执行计算操作就会有问题。所以,在多处理器下为了保证各个处理器的缓存是一致的,就会实现缓存一致性协议,每个处理器通过嗅探在总线上传播数据来检查自己缓存的值是不是过期了,当处理器发现自己缓存行对应的内存地址被修改,就会将当处理器的缓存行设置成无效状态,当处理器对这个数据进行修改操作的时候,会重新从系统内存中把据读到处理器缓存里。

同样,参照上面所说的,对于volatile来说,它的实现也不外乎需要达到以下两种实现效果:

- 1) Lock前缀指令会引起处理器缓存回写到内存Lock前缀指令会引起处理器缓存回写到内存
- 2) 一个处理器的缓存回写到内存会导致其他处理器的缓存无效

对象头

对象头:包括两部分信息。第一部分用于存储对象自身的运行时数据,如哈希码,GC分代年龄、锁态、线程持有锁、等等。这部分数据的长度在32为或64位,官方称之为"MarkWord"。对象头的一部分是类型指针,即对象指向它的类元素的指针,通过这个指针来确定这个对象时那个类的实例。如果Java对象时一个数组,则对象头还必须有一块用于记录数组长度的数据。因为Java数组元数据中有数组大小的记录)

偏向锁的概念

HotSpot的作者经过研究发现,大多数情况下,锁不仅不存在多线程竞争,而且总是由同一线程多次得,为了让线程获得锁的代价更低而引入了偏向锁。当一个线程访问同步块并获取锁时,会在对象头栈帧中的锁记录里存储锁偏向的线程ID,以后该线程在进入和退出同步块时不需要进行CAS操作来加和解锁,只需简单地测试一下对象头的Mark Word里是否存储着指向当前线程的偏向锁。如果测试功,表示线程已经获得了锁。如果测试失败,则需要再测试一下Mark Word中偏向锁的标识是否设成1(表示当前是偏向锁):如果没有设置,则使用CAS竞争锁;如果设置了,则尝试使用CAS将对头的偏向锁指向当前线程。

进入正题

在关于编程路上的一些杂谈 由线程的通信原理想到的(一)其实已经有讲到volatile 的实现方式的,通上面的深入想必已经有更细致的了解,然后也相信大家对于像i++ 这种复合操作不具有原子性(i是volat le变量)很是疑惑,这里要说一个概念:

程序计数器PC

程序计数器即指令地址寄存器。在某些计算机中用来存放当前正在执行的指令地址;而在另一些计算中则用来存放即将要执行的下一条指令地址;而在有指令预取功能的计算机中,一般还要增加一个程计数器用来存放下一条要取出的指令地址。程序计数器用以指出下条指令在主存中的存放地址,CPU据PC的内容去主存取得指令。因程序中指令是顺序执行的,所以PC有自增功能。

也就是说其实i++可以理解成一条指令,而i=i+1便是两条指令了包括i+1和将结果赋给i,应该不需要再深入了,已经很明了了。

锁的语义

这里在关于编程路上的一些杂谈 由线程的通信原理想到的(一)已经有说其底层还是依靠volatile来实 ,接下来就通过ReentrantLock源码来具体对其进行分析:

```
ntrantLock implements Lock, java.io.Serializable {
ic final long serialVersionUID = 7373984872572414699L;
                                                                                                                                                                        # - II 0
          private final Sync sync;
             into fair and nonfair versions below. Uses AQS state to represent the number of holds on the lock.
             stract static class Sync extends AbstractQueuedSynchronizer {
    private static final long serialVersionUID = -5179523762034025860L;
Performs non-fair tryLock. tryAcquire is implemented to a Gparam update the new value subclasses, but both need nonfair try for trylock method to actual to the expected value.
                                                                          protected final boolean compareAndSetState(int expect, int update) {
    // See below for intrinsics setup to support this
     c = getState();
    (c == 0) {
                                                                                        unsafe.compareAndSwapInt( this, stateOffset, expect, update);
      if (compareAndSetState( ), acquires)) { }
          setExclusiveOwnerThread(current
                                                                                 * @param thread the owner thread
 else if (current == getExclusiveOwnerThread()) {
                                                                                     tected final void setExclusiveOwnerThread(Thread thread) {
  exclusiveOwnerThread = thread;
                    new Error("Maximum lock count exceeded"):
                                                                                        merThread: 1
```

```
ected final boolean compareAndSetState(int expect, int update) {
    return unsafe.compareAndSwapInt( this, stateOffset, expect, update);
private static final Unsafe unsafe = Unsafe.getUnsafe();
private static final long headOffset;
private static final long tailOffset;
private static final long waitStatusOffset;
private static final long nextOffset;
static {
    try {
         stateOffset = unsafe.objectFieldOffset
             (AbstractQueuedSynchronizer.class.getDeclaredField( "state"))
         headOffset = unsafe.objectFieldOffset
             (AbstractQueuedSynchronizer.class.getDeclaredField(
                                                                          "head"));
         * The synchronization state.
        private volatile int state;
         * This operation has memory semantics of a {@code volatile} read
           @return current state value
                                                   返回当前锁的状态值
        protected final int getState() {
            return state;
```

对于compareAndSetState来说:

CAS, CPU指令,在大多数处理器架构,包括IA32、Space中采用的都是CAS指令,CAS的语义是"认为V的值应该为A,如果是,那么将V的值更新为B,否则不修改并告诉V的值实际为多少",CAS是乐观锁 技术,当多个线程尝试使用CAS同时更新同一个变量时,只有其中一个线程能更新变量的值而其它线程都失败,失败的线程并不会被挂起,而是被告知这次竞争中失败,并可以再次尝试。

CAS有3个操作数,内存值V,旧的预期值A,要修改的新值B。当且仅当预期值A和内存值V相同时,内存值V修改为B,否则什么都不做。

对于compareAndSetState来说:它是个原子方法,原理就是是CAS.这个是高效,而且是原子的,不用加锁也会因为其他值改了而产生误操作,应为会先判断当前值,符合期望才去改变,而我们所要操作的值无非是state而已。

对于上面截图的代码说的直白点就是对于一个线程如果当前没有竞争,则直接拿到或者上锁,否则,试获取即acquire(1)方法:

```
/**

* Sync object for non-fair locks

*/
static final class NonfairSync extends Sync {
   private static final long serialVersionUID = 7316153563782823691L;
   /**

* Performs lock. Try immediate barge, backing up to normal
```

```
Reelici aliceock Molifa il Sylic Lock()
FairSync
                            195
196
                             198
                                           static final class NonfairSync extends Sync {
   🐌 🛭 serialVersionUID:long = 7316153 199
                                               private static final long serialVersionUID = 73161535637
                           200
                          202
                                                * Performs lock. Try immediate barge, backing up to no
                            203
                            204
                            205 0
                                               final void lock() {
   (ii) • lock():void
iii) • newCondition():ConditionObject
iii) • nonfairTryAcquire(int):boolean
                                                   setExclusiveOwnerThread(Thread.currentThread())
    @ # readObject(ObjectInputStream):vi 209
                                                        acquire( );
```

```
* Acquires in exclusive mode, ignoring interrupts. Implemented
* by invoking at least once {@link #tryAcquire},
* returning on success. Otherwise the thread is queued, possibly
* repeatedly blocking and unblocking, invoking {@link
* #tryAcquire} until success. This method can be used
* to implement method {@link Lock#lock}.

* @param arg the acquire argument. This value is conveyed to
* {@link #tryAcquire} but is otherwise uninterpreted and
* can represent anything you like.

*/
public final void acquire(int arg) {
   if (!tryAcquire(arg) &&
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        selfInterrupt();
}
```



首先通过tryAcquire()方法尝试获取,如果不能的话,则通过AddWaiter()方法,用当前线程生成一个ode放入队尾,而acquireQueued()则是一种自旋锁的实现方式。最后把当前线程interrupt。这里可发现,java的 AQS的实现很巧妙的一个地方就是把tryAcquire延迟到子类去实现。公平锁和非公平锁实现方式是不一样的。非公平锁的tryAcquire()的是通过nonfairTryAcquire()。

然后看acquireQueued(),其实就是一个无限循环,直到获得锁为止。通过上图源码可以看到在should arkAfterFailedAcquire()方法中,通过前一个Node的waitStatus来判断是否应该把当前线程阻塞(所用了双&&开关语义),阻塞是通过parkAndCheckInterrupt()中的**LockSupport**.park()实现。

再看一下释放锁:

```
* Attempts to release this lock.

* If the current thread is the holder of this lock then the hold

* count is decremented. If the hold count is now zero then the lock

* is released. If the current thread is not the holder of this

* lock then {@link IllegalMonitorStateException} is thrown.

* @throws IllegalMonitorStateException if the current thread does not

* hold this lock

*/

public void unlock() {

sync.release(1);
}
```

release:

* Releases in exclusive mode. Implemented by unblocking one or
* more threads if {@link #tryRelease} returns true.
* This method can be used to implement method {@link Lock#unlock}.

* @param arg the release argument. This value is conveyed to
* {@link #tryRelease} but is otherwise uninterpreted and
* can represent anything you like.

```
* @return the value returned from {@link #tryRelease}
  public final boolean release(int arg) {
    if (tryRelease(arg)) {
       Node h = head;
       if (h!= null && h.waitStatus!= 0)
          unparkSuccessor(h);
       return true;
     return false;
  }
protected final boolean tryRelease(int releases) {
      int c = getState() - releases;
      if (Thread.currentThread() != getExclusiveOwnerThread())
         throw new IllegalMonitorStateException();
      boolean free = false:
      if (c == 0) {
         free = true;
         setExclusiveOwnerThread(null);
      }
      setState(c);
      return free:
```

可以看出tryRelease和tryAcquire一样,也是延迟到子类(Sync)实现的。c==0的时候,才能成功释放,所以多次锁定(看源码就可以知道lock—次c就+1,第一张截图的第二个判断,假如是当前线程的话再+一次1)就需要多次释放才能解锁。释放锁之后,就会唤醒队列的一个node中的线程

这段代码目的在于找出第一个可以unpark的线程,一般说来head.next == head,Head就是第一个程,但Head.next可能会被置为null(参考acquireQueued()源码),因此比较稳妥的办法是从后往前找一个可用线程。

```
if (s == null || s.waitStatus > 0) {
    s = null;
    for (Node t = tail; t != null && t != node; t = t.prev)
        if (t.waitStatus <= 0)
        s = t;
}
if (s != null)
    LockSupport.unpark(s.thread);
}</pre>
```

```
public class ReentrantLock implements Lock, java.io.Serializable {
    private static final long serialVersionUID = 7373984872572414699L;
    /** Synchronizer providing all implementation mechanics */
    private final Sync sync;

/**

* Base of synchronization control for this lock. Subclassed
    * into fair and nonfair versions below. Uses AQS state to
    * represent the number of holds on the lock.

*/
abstract static class Sync extends AbstractQueuedSynchronizer {
    private static final long serialVersionUID = -5179523762034025860L;

/**
```

其实我们在设计代码的时候也是可以通过静态内部类的方式来实现一些自己想要的功能,不过我们经会用Spring框架,其通过动态代理已经实现了这个按需的延迟加载这些特性,也无须去头疼这些那些

其实关键点也就这些,绕来绕去其实就一句话,假如有A和B两个线程,A符合期望的话,那么A就可入主东宫了,B还老老实实的做它的嫔妃就是。

通过以上这些解释,其实我们发现,锁的底层其实也是在反复操作一个volatile 变量,而多线程的其操作也是基于volatile 的特性来实现的,包括计数器,barrier,各种安全工具类,理解这个其他自然不是什么问题,包括很多并发框架的和事务等的设计,先就扯到这里吧。

参考文献:

Java并发编程的艺术

往期回顾:

关于编程路上的一些杂谈 由线程的通信原理想到的(一)