基于队列的锁:mcs lock简介

今天在Quora闲逛,看到一个对于MCS Lock的问答。答题的哥们深入浅出的写了一大篇，感觉非常不错，特地翻译出来。

原文翻译

要理解MCS Locks的本质，必须先知道其产生背景(Why)，然后才是其运作原理。就像原论文提到的，我们先从spin-lock说起。spin-lock 是一种基于test-and-set操作的锁机制。

```
function Lock(lock){
    while(test_and_set(lock)==1);
}

function Unlock(lock){
    lock = 0;
}
```

test_and_set是一个原子操作，读取lock，查看lock值，如果是0，设置其为1，返回0。如果是lock值为1， 直接返回1。这里lock的值0和1分别表示无锁和有锁。由于test_and_set的原子性，不会同时有两个进程/线程同时进入该方法， 整个方法无须担心并发操作导致的数据不一致。

一切看来都很完美，但是，有两个问题：(1) test_and_set操作必须有硬件配合完成，必须在各个硬件(内存，二级缓存，寄存器)等等之间保持 数据一致性，通讯开销很大。(2) 他不保证公平性，也就是不会保证等待进程/线程按照FIFO的顺序获得锁,可能有比较倒霉的进程/线程等待很长时间 才能获得锁。

为了解决上面的问题，出现一种Ticket Lock的算法，比较类似于Lamport's backery algorithm。就像在面包店里排队买面包一样，每个人先付钱，拿 一张票，等待他手中的那张票被叫到。下面是伪代码

```
ticket_lock {
    int now_serving;
    int next_ticket;
};

function Lock(ticket_lock lock){
    //get your ticket atomically
    int my_ticket = fetch_and_increment(lock.next_ticket);
    while(my_ticket != now_serving){};
}

function Unlock(ticket_lock lock){
    lock.now_serving++;
}
```

这里用到了一个原子操作fetch_and_increment(实际上lock.now_serving++也必须保证是原子)，这样保证两个进程/线程无法得到同一个ticket。 那么上面的算法解决的是什么问题呢？只调用一次原子操作！！！最原始的那个算法可是不停的在调用。这样系统在保持一致性上的消耗就小很多。 第二，可以按照先来先得(FIFO)的规则来获得锁。没有插队，一切都很公平。

但是，这还不够好。想想多处理器的架构，每个进程/线程占用的处理器都在读同一个变量，也就是now_serving。为什么这样不好呢，从多个CPU缓存的 一致性考虑，每一个处理器都在不停的读取

本文标签

java并发 × 2

锁 × 1

推荐会员

相关标签

now_serving本身就有不少消耗。最后单个进程/线程处理器对now_serving++的操作不但要刷新到本地缓存中，而且 要与其他的CPU缓存保持一致。为了达到这个目的，系统会对所有的缓存进行一致性处理，读取新值只能串行读取，然后再做操作，整个读取时间是与处理器个数 线性相关。

说到这里，才会聊到mcs队列锁。使用mcs锁的目的是，让获得锁的时间从O(n)变为O(1)。每个处理器都会有一个本地变量不用与其他处理器同步。伪代码如下

```
mcs_node{
    mcs_node next;
    int is_locked;
}

mcs_lock{
    mcs_node queue;
}

function Lock(mcs_lock,mcs_node my_node){
    my_node.next = NULL;
    mcs_node predecessor = fetch_and_store(lock.queue,my_node);
    if(predecessor!=NULL){
        my_node.is_locked = true;
        predecessor.next = my_node;
        while(my_node.is_locked){};
    }
}

function Unlock(mcs_lock lock,mcs_node my_node){
    if(my_node.next == NULL){
        if(compare_and_swap(lock.queue,my_node,NULL){
            return ;
        }
        else{
            while(my_node.next == NULL){};
        }
    }

    my_node.next.is_locked = false;
}
```

这次代码多了不少。但是只要记住每一个处理器在队列里都代表一个node,就不难理解整个逻辑。当我们试图获得锁时，先将自己加入队列，然后看看有没有其他 人(predecessor)在等待，如果有则等待自己的is_lock节点被修改成false。当我们解锁时，如果有一个后继的处理器在等待，则设置其is_locked=false，唤醒他。

在Lock方法里，用fetch_and_store来将本地node加入队列，该操作是个原子操作。如果发现前面有人在等待,则将本节点加入等待节点的next域中,等待前面的处理器唤醒本节点。 如果前面没有人，那么直接获得该锁。

在Unlock方法中，首先查看是否有人排在自己后面。这里要注意，即使暂时发现后面没有人,也必须用原子操作compare_and_swap确认自己是最后的一个节点。如果不能确认 必须等待之后节点排(my_node.next == NULL)上来。最后设置my_node.next.is_locked = false唤醒等待者。

最后我们看一下前面的问题是否解决了。原子操作的次数已经减少到最少，大多数时候只需要本地读写my_node变量。

注释

- 1.原文来自于 http://www.quora.com/How-does-an-MCS-lock-work.
- 2.论文来自于 http://www.cs.rochester.edu/u/scott/papers/1991_TOCS_synch.pdf.
- 3.一些伪代码 http://www.cs.rochester.edu/research/synchronization/pseudocode/ss.html
- 4.关于多处理器的架构下的共享变量访问的机制可以看看Java memory model或者搜一下NUMA与SMP架构，这方面本人也不是特别懂，还请各位赐教。

- 4.熟悉Java concurrent包的同学可以自己实现一下上面几个算法，可用的类和方法有:

  AtomicReferenceFieldUpdater.compareAndSet().对应compare_and_swap
  AtomicReferenceFieldUpdater.getAndSet().对应fetch_and_store
  AtomicInteger.getAndIncrement().对应fetch_and_increment

java并发 锁

分享长微博 0

愤怒对抗嗑嗑
发表于 2013-06-03 11:13

评论

时间 推荐

本文目前还没有评论.....

我要评论

需要登录后才能发言

记住我 登录