无锁队列详细分解——Lock与Cache，到底有没有锁？

原创 DPDK开源社区 2016-12-29

作者 马良

首先要感谢James Zhang 师兄多年前对我的鞭策，只是这篇文章写的晚了些，十二年已然过去了。但是他当年对我提的建议(关于Lock)一直萦绕在我心中，今天总算是有个小小的总结。

重温一下 CAS 操作的伪码。

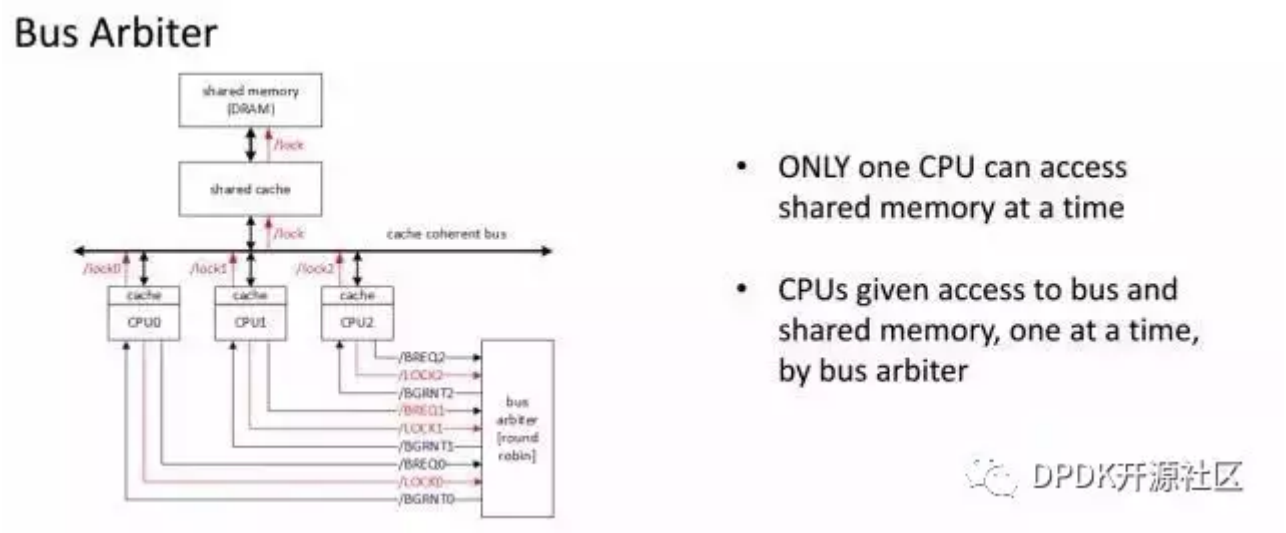
```
bool compare_and_swap (int *accum, int *dest, int newval)
{
   if ( *accum == *dest ) {
         *dest = newval;
         return true;
   }
   return false;
}
```

其对应的汇编指令是 cmpxchg， 这是一条原子操作指令，也就是说CPU 会保证这条指令执行的原子性。亦即，在特定的一个时间点，当多个cpu core 同时执行这条指令(参数相同)，只会有一个core 成功的执行swap的操作。

那么不由得激起了开发者的好奇心：CPU到底是如何保证这个原子性？以及其底层实现会给软件开发带来什么样的好处？又有什么样的缺点？本文就试图从这些角度进行尝试性的解读。

486，Pentium，时代

根据X86开发手册第三册 8.1.2 章节的说法，在486以及pentium 处理器会显式的向系统总线(也许是在3级缓存通信总线,细节没有公开)发出Lock# 信号从而获取对应Cache line的独占权。具体的实现从来没有公开过，不过下面这张图做了一个非官方的诠释。图片引自 https://www.cs.tcd.ie/jeremy.jones/CS4021/



从这张图可以看出：当core试图获取cache line的访问权(读/写)， 该core需要向bus发出一个锁信号，bus 仲裁器将会以round robin(无优先级轮流)算法选取某个core获取访问权。每次也只允许一个core获取这个cacheline，也就是独占，可以读可以写。非常明显的是这种方式非常低效，首先读写的要求没有区分开来，读的一致性要求要弱于写。一次只有一个core可以访问cacheline，而其它core则需要等待。时延高(尤其大大提高了读操作的时延)。

P6以后时代

根据开发者手册 8.1.4的描述在p6以后的x86处理器中，原子操作(例如cmpxchg)不再发出任何lock信号，一切都由 Cache 一致性协议来完成。

首先说一下我们要讨论的多核处理器的缓存结构

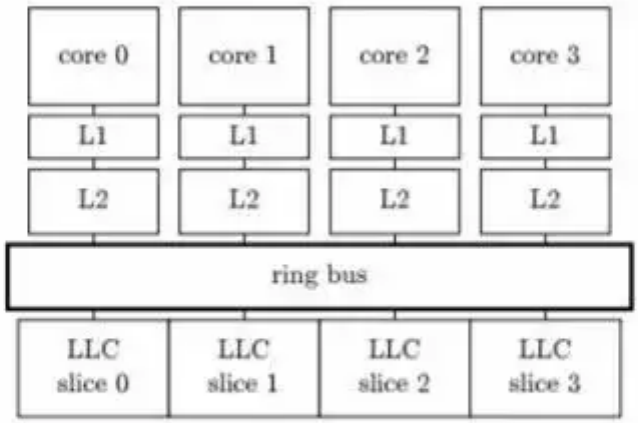


Fig. 1: Cache architecture of a quad-core Intel processor (since Sandy Bridge micro-architecture). The LLC is divided into slices, and interconnected with each core by a ring bus.

目前的结构每个core都会持有自己私有的L1/L2 Cache，但是 L3 cache是所有core共有的。那么就存在一个cache一致性的问题， 由公开资料可以知道， X86 采用的是改进型的MESI协议MESIF，主要是添加了一个 Forwarding的状态，简化有多share节点状态下 对read request的处理。每个core都挂在一个或数个cache ring bus只上。每个core都有一个cbox(bus agent)负责监听其它core对cache的操作行为， 从而根据协议采取对应的行动。

首先非常简单的说一下 MESI协议，展开讨论这个话题超出本文的范围。该协议规定：每个Cacheline有4中不同的状态 ：

Modified(M)：表示这个cacheline已经被修改，但是还未写回主存(cache中数据与主存中数据不一致。所有其它core对这个cacheline的读操作必须在该cacheline 写回主存后，写回主存后,状态变换到share。

Exclusive(E)： 表示这个cacheline 目前是独有的且与主存一致，可以转换到shared 或者M，重点是可以转换到M，也就意味着可以对其修改。

Shared(S)：表示这个cacheline 在其它core的cache中也存在， 目前也与主存一致，随时会被invalidate。

Invalid(I)：表示这个cacheline 目前不可用。

这个图的含义就是当一个core持有一个cacheline的状态为Y时，其它core对应的cacheline应该处于状态X， 比如地址 0x00010000 对应的cacheline在core0上为状态M， 则其它所有的core对应于0x00010000的cacheline都必须为I ， 0x00010000 对应的cacheline在core0上为状态S，则其它所有的core对应于0x00010000的cacheline 可以是S或者I 。

MESIF 就是对MESI做了一个优化。大家设想一下， 如果现在一个cacheline 是S状态，在多个core中有同一copy，那么现在有一个新的core需要读取该cacheline， 应该由那个core来应答。 如果每个持有该cacheline的core都来应答就会造成冗余的数据传输，所以对于在系统中有多个copy的S状态的cacheline中，必须选取一个转换为F，只有F状态的负责应答。通常是最后持有该copy的cacheline转换为F。

(以上两个状态图来自维基百科)

说了这些背景知识之后,再回到我们的CAS指令。

当两个core同时执行针对同一地址的CAS指令时，其实他们是在试图修改每个core自己持有的Cache line， 假设两个core都持有相同地址对应cacheline，且各自cacheline 状态为S，这时如果要想成功修改，就首先需要把S转为E或者M， 则需要向其它core invalidate 这个地址的cacheline，则两个core都会向ring bus 发出 invalidate这个操作，那么在ringbus上就会根据特定的设计协议仲裁是core0，还是core1能赢得这个invalidate，者完成操作，失败者需要接受结果invalidate自己对应的cacheline，再读取胜者修改后的值，回到起点。

到这里, 我们可以发现MESIF协议大大降低了读操作的时延,没有让写操作更慢，同时保持了一致性。那么对于我们的CAS操作来说，其实锁并没有消失，只是转嫁到了ring bus的总线仲裁协议中。而且大量的多核同时针对一个地址的CAS操作会引起反复的互相invalidate 同一cacheline，造成pingpong效应，同样会降低性能。只能说，基于CAS的操作仍然是不能滥用，不到万不得已不用,通常情况下还是使用数据分离模式更好。

最后更进一步 ring bus的仲裁协议又是什么？ 从公开的资料，x86这方面的公开信息非常罕见。唯一可以推断的是 这个协议会保证公平性，在invalidate的竞争中不会总是一个core赢，从而保证不会有starving core。在power4/5 的设计中有一篇论文涉及到power处理器相关的设计细节, 大家可以参考。

http://research.cs.wisc.edu/multifacet/papers/micro06_ring.pdf

---

作者简介

马良，现为Intel工程师。毕业于中国科学技术大学，主要从事高速包处理，虚拟化以及应用密码学等领域的研究。