内存屏障

原文地址 作者：Martin Thompson 译者：一粟 校对：无叶，方腾飞

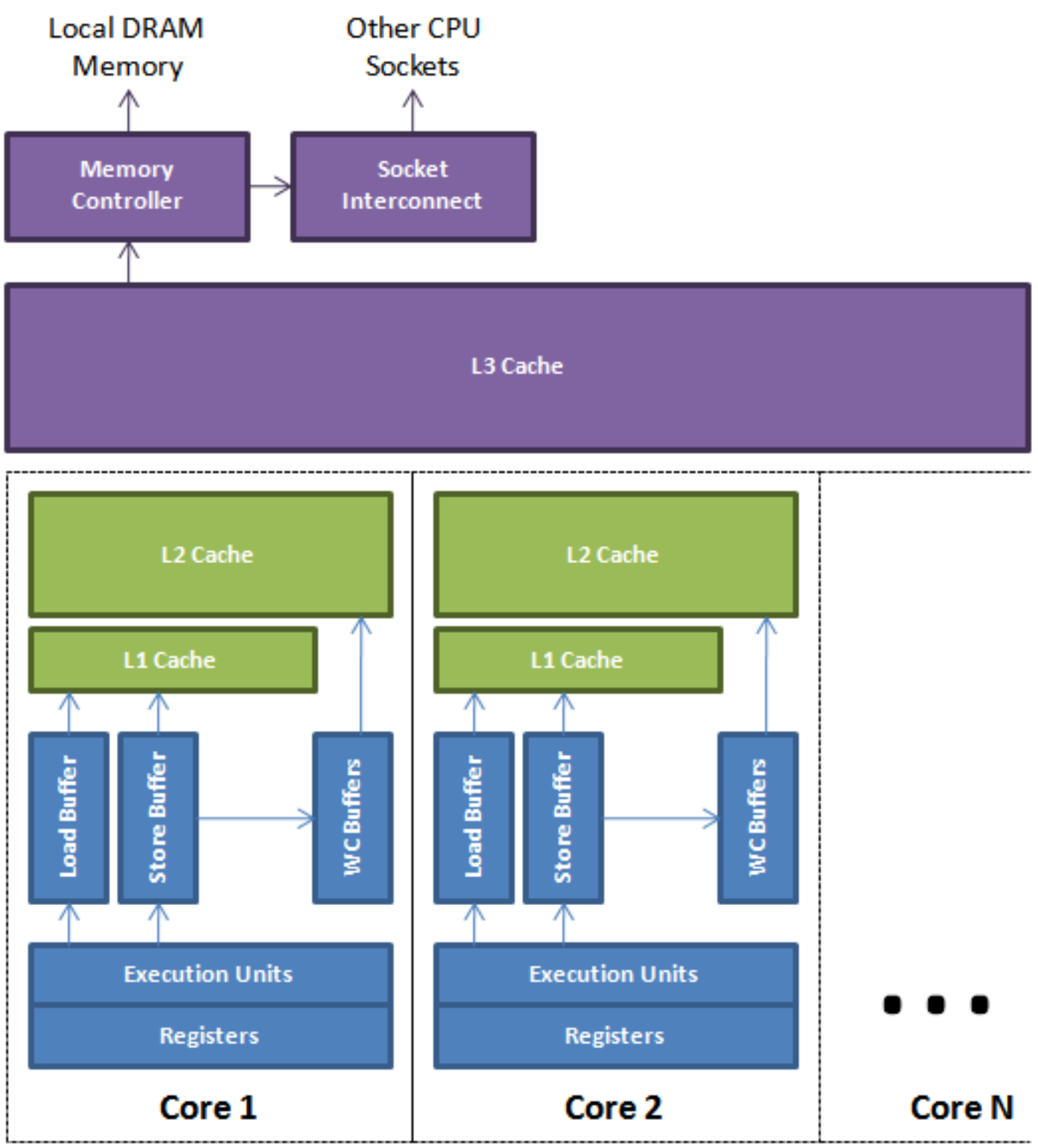
本文我将和大家讨论并发编程中最基础的一项技术：内存屏障或内存栅栏，也就是让一个CPU处理单元中的内存状态对其它处理单元可见的一项技术。

CPU使用了很多优化技术来实现一个目标：CPU执行单元的速度要远超主存访问速度。在上一篇文章“Write Combing （合并写）”中我已经介绍了其中的一项技术。CPU避免内存访问延迟最常见的技术是将指令管道化，然后尽量重排这些管道的执行以最大化利用缓存，从而把因为缓存未命中引起的延迟降到最小。

当一个程序执行时，只要最终的结果是一样的，指令是否被重排并不重要。例如，在一个循环里，如果循环体内没用到这个计数器，循环的计数器什么时候更新（在循环开始，中间还是最后）并不重要。编译器和CPU可以自由的重排指令以最佳的利用CPU，只要下一次循环前更新该计数器即可。并且在循环执行中，这个变量可能一直存在寄存器上，并没有被推到缓存或主存，这样这个变量对其他CPU来说一直都是不可见的。

CPU核内部包含了多个执行单元。例如，现代Intel CPU包含了6个执行单元，可以组合进行算术运算，逻辑条件判断及内存操作。每个执行单元可以执行上述任务的某种组合。这些执行单元是并行执行的，这样指令也就是在并行执行。但如果站在另一个CPU角度看，这也就产生了程序顺序的另一种不确定性。

最后，当一个缓存失效发生时，现代CPU可以先假设一个内存载入的值并根据这个假设值继续执行，直到内存载入返回确切的值。




```
7  执行单元 -> Load/Store缓冲区->L1 Cache --->L3 Cache-->内存控制器-->主存
9  +-> Write Combine缓冲区->L2 Cache ---+
```

代码顺序并不是真正的执行顺序，只要有空间提高性能，CPU和编译器可以进行各种优化。缓存和主存的读取会利用load, store和write-combining缓冲区来缓冲和重排。这些缓冲区是查找速度很快的关联队列，当一个后来发生的load需要读取上一个store的值，而该值还没有到达缓存，查找是必需的，上图描绘的是一个简化的现代多核CPU，从上图可以看出执行单元可以利用本地寄存器和缓冲区来管理和缓存子系统的交互。

在多线程环境里需要使用某种技术来使程序结果尽快可见。这篇文章里我不会涉及到 Cache Conherence 的概念。请先假定一个事实：一旦内存数据被推送到缓存，就会有消息协议来确保所有的缓存会对所有的共享数据同步并保持一致。这个使内存数据对CPU核可见的技术被称为内存屏障或内存栅栏。

内存屏障提供了两个功能。首先，它们通过确保从另一个CPU来看屏障的两边的所有指令都是正确的程序顺序，而保持程序顺序的外部可见性；其次它们可以实现内存数据可见性，确保内存数据会同步到CPU缓存子系统。

大多数的内存屏障都是复杂的话题。在不同的CPU架构上内存屏障的实现非常不一样。相对来说Intel CPU的强内存模型比DEC Alpha的弱复杂内存模型（缓存不仅分层了，还分区了）更简单。因为x86处理器是在多线程编程中最常见的，下面我尽量用x86的架构来阐述。

Store Barrier

Store屏障，是x86的”sfence“指令，强制所有在store屏障指令之前的store指令，都在该store屏障指令执行之前被执行，并把store缓冲区的数据都刷到CPU缓存。这会使得程序状态对其它CPU可见，这样其它CPU可以根据需要介入。一个实际的好例子是Disruptor中的BatchEventProcessor。当序列Sequence被一个消费者更新时，其它消费者(Consumers)和生产者（Producers）知道该消费者的进度，因此可以采取合适的动作。所以屏障之前发生的内存更新都可见了。

```
01  private volatile long sequence = RingBuffer.INITIAL_CURSOR_VALUE;
04  long nextSequence = sequence.get() + 1L;
09          final long availableSequence = barrier.waitFor(nextSequence);
10          while (nextSequence <= availableSequence)
12              event = ringBuffer.get(nextSequence);
13              boolean endOfBatch = nextSequence == availableSequence;
14              eventHandler.onEvent(event, nextSequence, endOfBatch);
15              nextSequence++;
17          sequence.set(nextSequence - 1L);
20      catch (final Exception ex)
22          exceptionHandler.handle(ex, nextSequence, event);
23          sequence.set(nextSequence);
25          nextSequence++;
```

Load Barrier

Load屏障，是x86上的”ifence“指令，强制所有在load屏障指令之后的load指令，都在该load屏障指令执行之后被执行，并且一直等到load缓冲区被该CPU读完才能执行之后的load指令。这使得从其它CPU暴露出来的程序状态对该CPU可见，这之后CPU可以进行后续处理。一个好例子是上面的BatchEventProcessor的sequence对象是放在屏障后被生产者或消费者使用。

Full Barrier

Full屏障，是x86上的”mfence“指令，复合了load和save屏障的功能。

Java内存模型

Java内存模型中volatile变量在写操作之后会插入一个store屏障，在读操作之前会插入一个load屏障。一个类的final字段会在初始化后插入一个store屏障，来确保final字段在构造函数初始化完成并可被使用时可见。

原子指令和Software Locks

原子指令，如x86上的”lock …” 指令是一个Full Barrier，执行时会锁住内存子系统来确保执行顺序，甚至跨多个CPU。Software Locks通常使用了内存屏障或原子指令来实现变量可见性和保持程序顺序。

内存屏障的性能影响

内存屏障阻碍了CPU采用优化技术来降低内存操作延迟，必须考虑因此带来的性能损失。为了达到最佳性能，最好是把要解决的问题模块化，这样处理器可以按单元执行任务，然后在任务单元的边界放上所有需要的内存屏障。采用这个方法可以让处理器不受限的执行一个任务单元。合理的内存屏障组合还有一个好处是：缓冲区在第一次被刷后开销会减少，因为再填充改缓冲区不需要额外工作了。

并发编程网 | 让天下没有难学的技术

长按，识别二维码，加关注

微信号: ifeves

⭐添加本文到我的收藏