在之前的文章《从 Java 内存模型看内部细节》中我们提到一个问题， final 字段的写入与构造方法返回之前，编译器会插入一个 StoreStore 屏障；同样，在 volatile 字段的写入之前，也会插入一个 StoreStore 屏障，但你是否有想过，这个屏障在 x86 的架构下为什么是no-op(空操作)呢?

这不得不从 x86 的 TSO(Total Store Order)[1] 模型说起。

论文[1]中，作者在非数学层面用四句话来概括什么是 x86-TSO：

- 首先 store buffers 被设计成了 FIFO 的队列，如果某个线程需要读取内存里的变量，务必优先读取本地 store buffer 中的值（如果有的话），否则去主内存里读取；
- MFENCE 指令用于清空本地 store buffer，并将数据刷到主内存；
- 某线程执行 Lock 前缀的指令集时，会去争抢全局锁，拿到锁后其他线程的读取操作会被阻塞，在释放锁之前，会清空该线程的本地的 store buffer，这里和 MFENCE 执行逻辑类似；
- store buffers 被写入变量后，除了被其他线程持有锁以外的情况，在任何时刻均有可能写回内存。

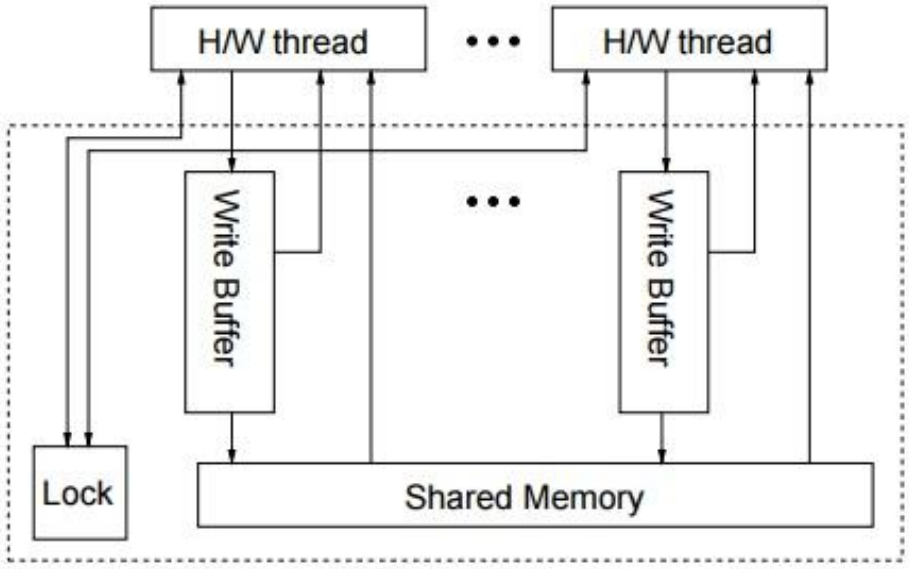


**Figure 1: x86-TSO block diagram**

上面给出的图片是否似曾相识？在我的另一篇文章里介绍过 store buffer 与其他 CPU 组件的基本结构，但是略有不同的是，这张图忽略了 CPU 缓存的存在，这是因为作者为了更好阐述 x86-TSO 模型而给我们呈现的抽象图示，不涉及到具体的 CPU 构件，这里的 thread 其实和 CPU 的 Processor 是一一对应的，这里的 write buffer，实质上就是 store buffer。

根据上面的 x86-TSO 模型，我们可以推测出 x86 架构下是不需要 StoreStore 屏障的，试想一下，x86 的 store buffer 被设计成了 FIFO，纵然在同一个线程中执行多次写入 buffer 的操作，最终依旧是严格按照 FIFO 顺序 dequeue 并写回到内存里，自然而然，对于其他任何线程而言，所『看到』的该线程的变量写回顺序是和指令序列一致的，因此不会出现重排序。

进一步思考，我们知道读屏障就是为了解决 invalidate queue 的引入导致数据不一致的问题，x86-TSO 模型下是没有 invalidate queue 的，因此也不需要读屏障（LoadLoad[2]）。

那么 LoadStore 呢？我们这里用个例子直观分析下：

```
a,b=0;
c=2;

```

```
// proc 0
void foo(){
  assert b == 0;
  c=b;
  a=2;
  b=1;
}

// proc 1
void bar(){
 if(a==2){
   assert c == 0;
 }
}
```

实际上这个断言的通过需要依赖两个屏障，LoadStore 和 StoreStore，我们上面讨论了 x86 不需要 StoreStore Barrier，因此这相当于是 no-op；我们也就只需要分析 LoadStore Barrier 也是 no-op 的情况下，上面两个断言是否一定能通过?

我的分析是这样的，foo 方法中 c、a、b 三个变量顺序写入，是不会被重排序的，这是由 store buffer FIFO 特性所决定。换句话说，b=1 不会被重排序到 c=b 之前，因此 bar 方法的断言通过；

那 b=1 是否会重排序到 assert b == 0 之前呢？我们知道 x86-TSO 要求变量读取优先检查 store buffer ，如果不存在则去主内存寻址，根据这套顺序，在执行 assert b==0 的时候 b 唯一可能就是 0，因此 foo 方法断言通过。

根据上面的分析，我们可以推论 LoadStore Barrier 也是 no-op。

最后，StoreLoad，这或许是 x86-TSO 模型下**唯一**需要考虑的重排序场景(排除编译器优化重排序)。虽然store buffer 是 FIFO，但整体架构本质依然是最终一致性而非线性一致性。这势必会出现在某个时间节点，不同处理器看到的变量不一致的情况。继续看下面的伪代码：

```
x,y=0;
// proc 0
void foo(){
  x=1;
  read y;
}

// proc 1
void bar(){
  y=1;
  read x;
}
```

如果遵循线性一致性，我们大可以枚举可能发生的情况，但无论怎么枚举，都不可能是 x=y=0，然而诡异之处在于 x86-TSO 模型下是允许 x=y=0这种情况存在的。结合上面的抽象图示分析，x86 在 StoreLoad 的场景下允许 x=y=0 发生，也就不难理解了。以 foo 方法为例，由于 y=1 的写入有可能还停留在 proc1 的 store buffer 中，foo 方法末尾读到的 y 可能是旧值，同理 bar 方法末尾也有可能读到 x 的旧值，那么读出来自然有可能是 x=y=0。

在 x86-TSO 里提到 MFENCE，这个指令用于强制清空本地 store buffer，并将数据刷到主内存，本质上就是 StoreLoad Barrier。

> This serializing operation guarantees that every load and store instruction that precedes the MFENCE instruction in program order becomes globally visible before any load or store instruction that follows the MFENCE instruction. MFENCE - Memory Fence

MFENCE 保证了在 MFENCE 指令执行前的读写操作对全局可见，可见这是一个很重的屏障，这也是 StoreLoad Barrier 所需要的。除此之外 x86-TSO 还提到 LOCK 前缀的相关指令，在释放锁的过程中有和 MFENCE 类似的过程，同样能达到 StoreLoad Barrier 的效果。

而 Hotspot VM 选择了 LOCK 指令作为 StoreLoad 屏障，这又是为何呢？Hotspot 源码中给出了这个注释：

```
enum Membar_mask_bits {
    StoreStore = 1 << 3,
    LoadStore  = 1 << 2,
    StoreLoad  = 1 << 1,
    LoadLoad   = 1 << 0
  };

  // Serializes memory and blows flags
  void membar(Membar_mask_bits order_constraint) {
    if (os::is_MP()) {
      // We only have to handle StoreLoad
      if (order_constraint & StoreLoad) {
        // All usable chips support "locked" instructions which suffice
        // as barriers, and are much faster than the alternative of
        // using cpuid instruction. We use here a locked add [esp-C],0.
        // This is conveniently otherwise a no-op except for blowing
        // flags, and introducing a false dependency on target memory
        // location. We can't do anything with flags, but we can avoid
        // memory dependencies in the current method by locked-adding
        // somewhere else on the stack. Doing [esp+C] will collide with
        // something on stack in current method, hence we go for [esp-C].
        // It is convenient since it is almost always in data cache, for
        // any small C.  We need to step back from SP to avoid data
        // dependencies with other things on below SP (callee-saves, for
        // example). Without a clear way to figure out the minimal safe
        // distance from SP, it makes sense to step back the complete
        // cache line, as this will also avoid possible second-order effects
        // with locked ops against the cache line. Our choice of offset
        // is bounded by x86 operand encoding, which should stay within
        // [-128; +127] to have the 8-byte displacement encoding.
        //
        // Any change to this code may need to revisit other places in
        // the code where this idiom is used, in particular the
        // orderAccess code.

        int offset = -VM_Version::L1_line_size();
        if (offset < -128) {
```

```
38            offset = -128;
39          }
40
41          lock();
42          addl(Address(rsp, offset), 0);// Assert the lock# signal here
43        }
44      }
45    }
```

Hotspot 采用的是 locked add [esp-C],0 ，其最主要的出发点还是性能。

# 小结

由于 x86 是遵循 TSO 的最终一致性模型，如若出现 data race 的情况还是需要考虑同步的问题，尤其是在 StoreLoad 的场景。而其余场景由于其 store buffer 的特殊性以及不存在 invalidate queue 的因素，可以不需要考虑重排序的问题，因此在 x86 平台下，除了 StoreLoad Barrier 以外，其余的 Barrier 均为空操作。

参考资料：


[1]《x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors》

[2][https://stackoverflow.com/questions/15360598/what-does-a-loadload-barrier-really-do](https://link.zhihu.com/?target=https%3A//stackoverflow.com/questions/15360598/what-does-a-loadload-barrier-really-do)

[3][https://paulcavallaro.com/blog/x86-tso-a-programmers-model-for-x86-multiprocessors/](https://link.zhihu.com/?target=https%3A//paulcavallaro.com/blog/x86-tso-a-programmers-model-for-x86-multiprocessors/)

[4][https://mp.weixin.qq.com/s?__biz=MzUzMDk3NjM3Mg==&mid=2247483755&idx=1&sn=50f80e73f46fab04d8a799e8731432c6&chksm=fa48da70cd3f5366d9658277cccd9e36fca540276f580822d41aef7d8af4dda480fc85e3bde4&token=630636109&lang=zh_CN#rd](https://link.zhihu.com/?target=https%3A//mp.weixin.qq.com/s%3F__biz%3DMzUzMDk3NjM3Mg%3D%3D%26mid%3D2247483755%26idx%3D1%26sn%3D50f80e73f46fab04d8a799e8731432c6%26chksm%3Dfa48da70cd3f5366d9658277cccd9e36fca540276f580822d41aef7d8af4dda480fc85e3bde4%26token%3D630636109%26lang%3Dzh_CN%23rd)

[5][https://www.cl.cam.ac.uk/~pes20/weakmemory/index3.html](https://link.zhihu.com/?target=https%3A//www.cl.cam.ac.uk/~pes20/weakmemory/index3.html)

[6][https://mp.weixin.qq.com/s?__biz=MzUzMDk3NjM3Mg==&mid=2247483755&idx=1&sn=50f80e73f46fab04d8a799e8731432c6&chksm=fa48da70cd3f5366d9658277cccd9e36fca540276f580822d41aef7d8af4dda480fc85e3bde4&token=1422563498&lang=zh_CN#rd](https://link.zhihu.com/?target=https%3A//mp.weixin.qq.com/s%3F__biz%3DMzUzMDk3NjM3Mg%3D%3D%26mid%3D2247483755%26idx%3D1%26sn%3D50f80e73f46fab04d8a799e8731432c6%26chksm%3Dfa48da70cd3f5366d9658277cccd9e36fca540276f580822d41aef7d8af4dda480fc85e3bde4%26token%3D1422563498%26lang%3Dzh_CN%23rd)

[7][https://software.intel.com/en-us/forums/intel-moderncode-for-parallel-architectures/topic/304284](https://link.zhihu.com/?target=https%3A//software.intel.com/en-us/forums/intel-moderncode-for-parallel-architectures/topic/304284)

[8][http://www.4e00.com/blog/java/2018/10/21/inside-java-memory-model.html](https://link.zhihu.com/?target=http%3A//www.4e00.com/blog/java/2018/10/21/inside-java-memory-model.html)

[9][https://github.com/leonlibraries/openjdk-10/blob/master/hotspot/src/cpu/x86/vm/assembler_x86.hpp#L1307](https://link.zhihu.com/?target=https%3A//github.com/leonlibraries/openjdk-10/blob/master/hotspot/src/cpu/x86/vm/assembler_x86.hpp%23L1307)