突然想聊聊这个话题，是因为知乎上的一个问题多次出现在了我的Timeline里：请问，多个线程可以读一个变量，只有一个线程可以对这个变量进行写，到底要不要加锁？可惜的是很多高票答案语焉不详，甚至有所错漏。所以我想在这篇文章里斗胆聊聊这个水挺深的问题。受限于个人水平，**文章若有错漏，还望读者不吝赐教**。

首先约定，由于CPU的架构和设计浩如烟海，本文站在工程师的角度，只谈IA32/AMD64(x86-64)架构，不讨论其他架构的细节和差异。并且文章中主要引用Intel的文档予以佐证，不关注AMD在实现细节上的差异。

众所周知，当一个执行中的程序的数据被多个执行流并发访问的时候，就会涉及到**同步（Synchronization）**的问题。同步的目的是保证不同执行流对共享数据并发操作的一致性。早在单核时代，使用锁或者原子变量就很容易达成这一目的。甚至因为CPU的一些访存特性，对某些内存对齐数据的读或写也具有原子的特性。

比如，在《Intel® 64 and IA-32 Architectures Software Developer's Manual》的第三卷System Programming Guide的Chapter 8 Multiple-Processor Management里，就给出了这样的说明：

8.1.1 Guaranteed Atomic Operations

The Intel486 processor (and newer processors since) guarantees that the following basic memory operations will always be carried out atomically:

- Reading or writing a byte
- Reading or writing a word aligned on a 16-bit boundary
- Reading or writing a doubleword aligned on a 32-bit boundary

The Pentium processor (and newer processors since) guarantees that the following additional memory operations will always be carried out atomically:

- Reading or writing a quadword aligned on a 64-bit boundary
- 16-bit accesses to uncached memory locations that fit within a 32-bit data bus

The P6 family processors (and newer processors since) guarantee that the following additional memory operation will always be carried out atomically:

- Unaligned 16-, 32-, and 64-bit accesses to cached memory that fit within a cache line

也就是说，有些内存对齐的数据的访问在CPU层面就是原子进行的（注意这里说的只是单次的读或者写，类似普通变量i的i++操作不止一次内存访问）。此时，环形队列（Ring buffer）这种数据结构在某些架构的**单核CPU**上，只有一个Reader和一个Writer的情况下是不需要额外同步措施的。原因就是`read_index`和`writer_index`的写操作在**满足对齐内存访问的情况下**是原子的，不需要额外的同步措施。注意这里我加粗了**单核CPU**这个关键字，那么到了多核心处理器的今天，该操作就不是原子了吗？不，依旧是原子的，但是出现了其他的干扰因素迫使可能需要额外的同步措施才能保证原本无锁代码的正确运行。

首先是现代编译器的代码优化和编译器指令重排可能会影响到代码的执行顺序。编译期指令重排是通过调整代码中的指令顺序，在不改变代码语义的前提下，对变量访问进行优化。从而尽可能的减少对寄存器的读取和存储，并充分复用寄存器。但是编译器对数据的依赖关系判断只能在单执行流内，无法判断其他执行流对竞争数据的依赖关系。就拿无锁环形队列来说，如果Writer做的是先放置数据，再更新索引的行为。如果索引先于数据更新，Reader就有可能会因为判断索引已更新而读到脏数据。

那禁止编译器对该类变量的优化，解决了编译期的重排序就没事了吗？不，CPU还有**乱序执行（Out-of-Order Execution）**的特性。流水线（Pipeline）和乱序执行是现代CPU基本都具有的特性。机器指令在流水线中经历取指、译码、执行、访存、写回等操作。为了CPU的执行效率，流水线都是并行处理的，在不影响语义的情况下。**处理器次序（Process Ordering，机器指令在CPU实际执行时的顺序）**和**程序次序（Program Ordering，程序代码的逻辑执行顺序）**是允许不一致的，即满足**As-if-Serial**特性。显然，这里的不影响语义依旧只能是保证指令间的**显式因果关系**，无法保证**隐式因果关系**。即无法保证语义上不相关但是在程序逻辑上相关的操作序列按序执行。从此单核时代CPU的**Self-Consistent**特性在多核时代已不存在，多核CPU作为一个整体看，不再满足**Self-Consistent**特性。

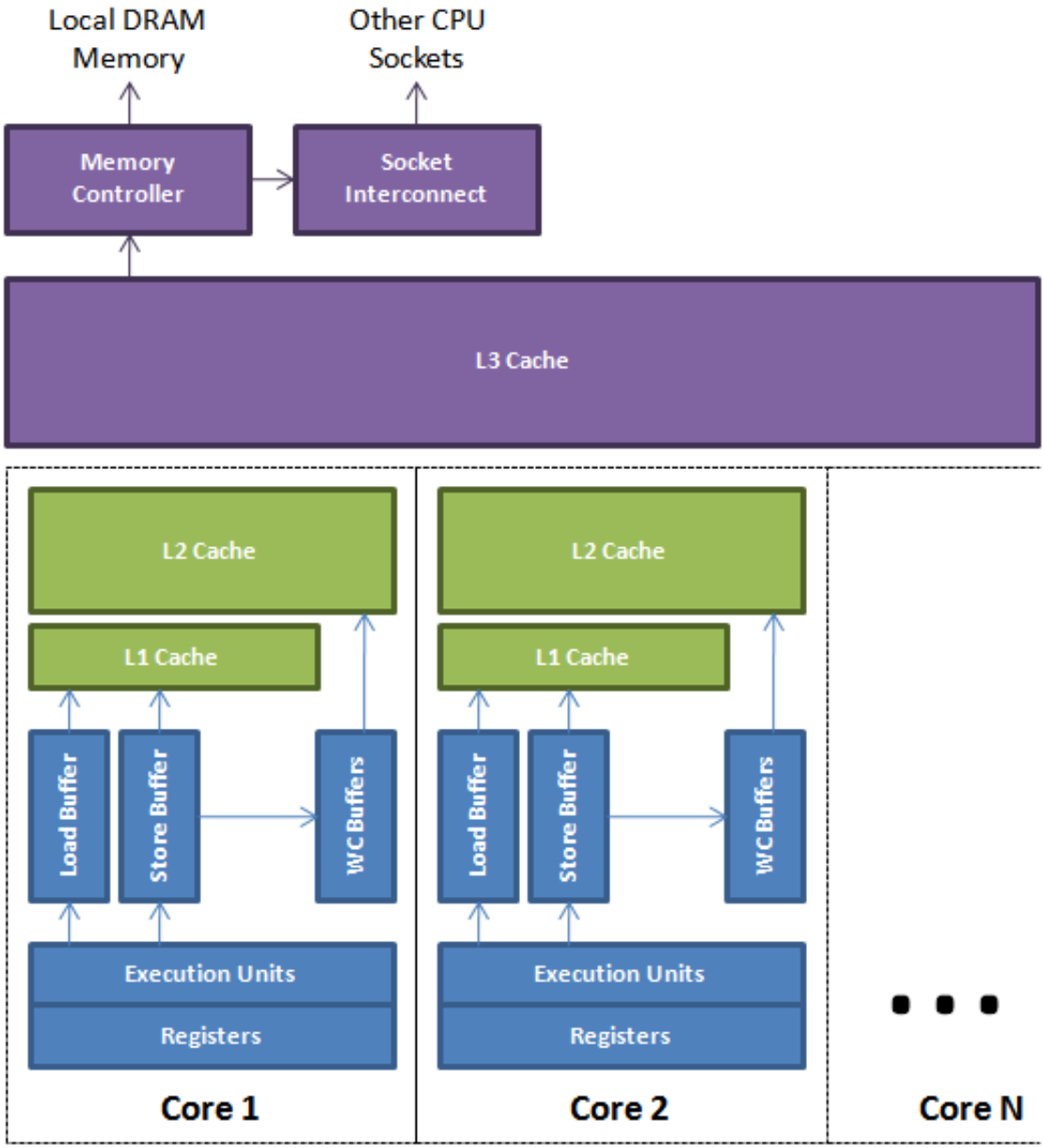
简单总结一下，如果不做多余的防护措施，单核时代的无锁环形队列在多核CPU中，一个CPU核心上的Writer写入数据，更新index后。另一个CPU核心上的Reader依靠这个index来判断数据是否写入的方式不一定可靠。index有可能先于数据被写入，从而导致Reader读到脏数据。

所有的麻烦到这里就结束了吗？当然不，还有Cache的问题。前文提到的都是**顺序一致性（Sequential Consistency）**的问题，没有涉及**Cache一致性（Cache Coherence）**的问题。虽然说一般情况下程序员只需要关注顺序一致性即可，但是区分清楚这两个概念也能更好的解释**内存屏障（Memory Barrier）**。

开始提到Cache一致性协议之前，先介绍两个名词：

- Load/Read CPU读操作，是指将内存数据加载到寄存器的过程
- Store/Write CPU写操作，是指将寄存器数据写回主存的过程

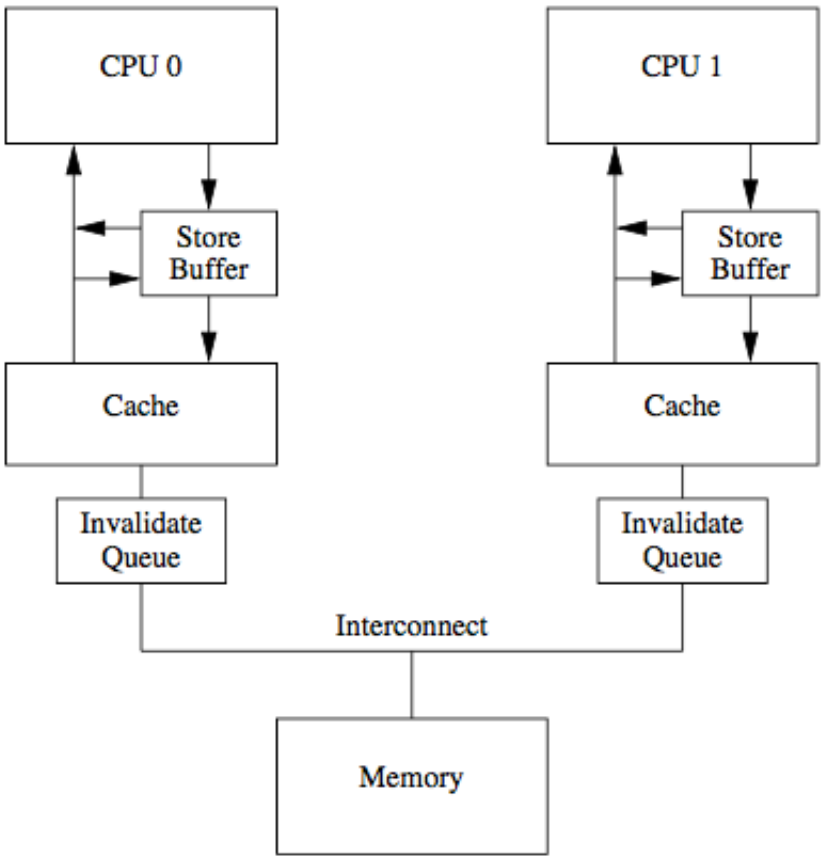
现代处理器的缓存一般分为三级，由每一个核心独享的L1、L2 Cache，以及所有的核心共享L3 Cache组成：



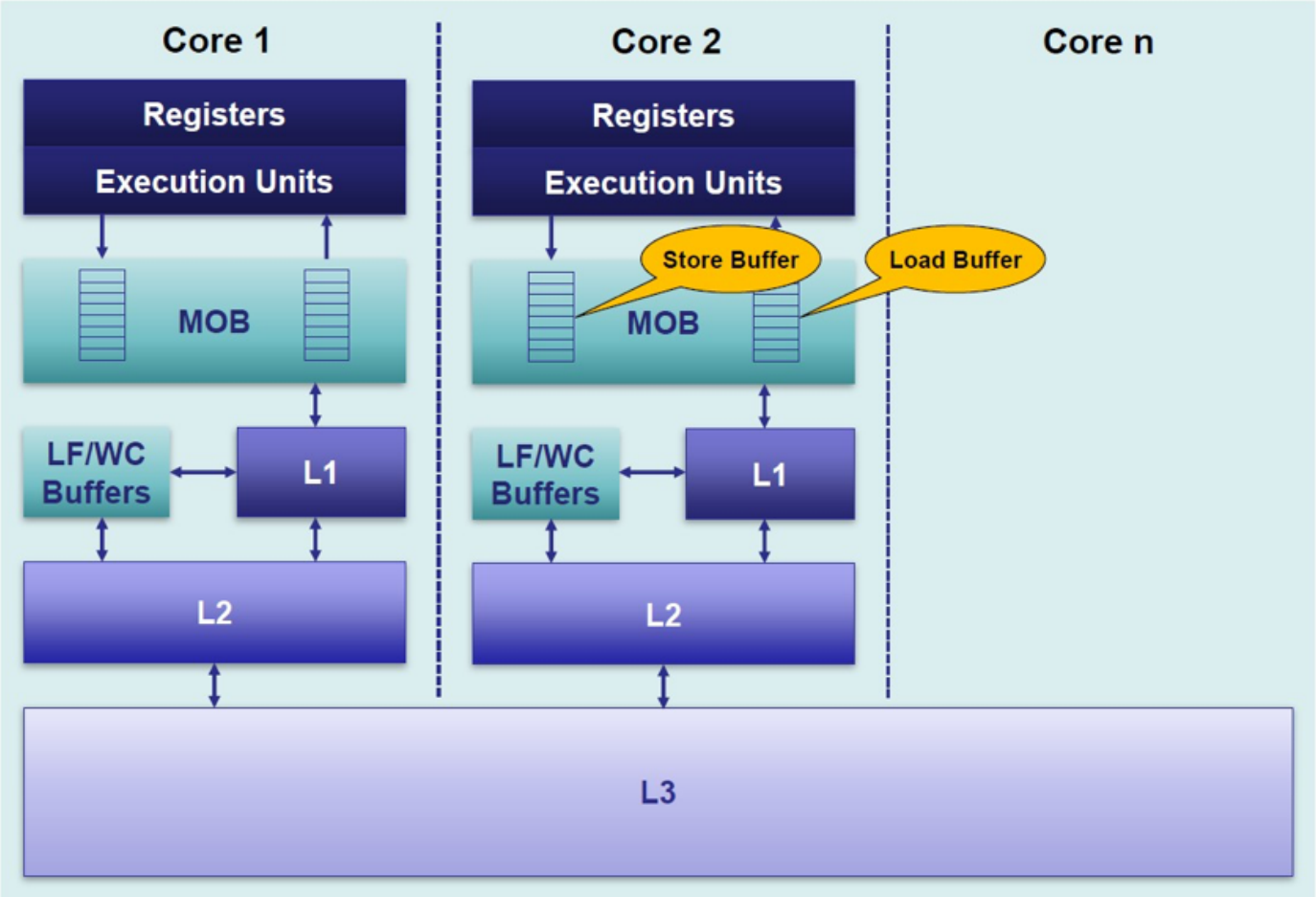
由于Cache的容量很小，一般都是充分的利用局部性原理，按行/块来和主存进行批量数据交换，以提升数据的访问效率。以前写过一篇《浅析x86架构中cache的组织结构》，这里不再赘述。既然各个核心之间有独立的Cache存储器，那么这些存储器之间的数据同步就是个比较复杂的事情。缓存数据的一致性由缓存一致性协议保证。这里比较经典的当属MESI协议。Intel的处理器使用从MESI中演化出的MESIF协议，而AMD使用MOESI协议。缓存一致

性协议的细节超出了本文的讨论范围，有兴趣的读者可以自行研究。

传统的MESI协议中有两个行为的执行成本比较大。一个是将某个Cache Line标记为Invalid状态，另一个是当某Cache Line当前状态为Invalid时写入新的数据。所以CPU通过Store Buffer和Invalidate Queue组件来降低这类操作的延时。如图：



当一个核心在Invalid状态进行写入时，首先会给其它CPU核发送Invalid消息，然后把当前写入的数据写入到Store Buffer中。然后异步在某个时刻真正的写入到Cache Line中。当前CPU核如果要读Cache Line中的数据，需要先扫描Store Buffer之后再读取Cache Line（Store-Buffer Forwarding）。但是此时其它CPU核是看不到当前核的Store Buffer中的数据的，要等到Store Buffer中的数据被刷到了Cache Line之后才会触发失效操作。而当一个CPU核收到Invalid消息时，会把消息写入自身的Invalidate Queue中，随后异步将其设为Invalid状态。和Store Buffer不同的是，当前CPU核心使用Cache时并不扫描Invalidate Queue部分，所以可能会有极短时间的脏读问题。当然这里的Store Buffer和Invalidate Queue的说法是针对一般的SMP架构来说的，不涉及具体架构。事实上除了Store Buffer和Load Buffer，流水线为了实现并行处理，还有Line Fill Buffer/Write Combining Buffer 等组件，参考文献8-10给出了相关的资料可以进一步阅读。



好了，问题背景描述的差不多了，下面该解决方案登场了。

编译器优化乱序和**CPU执行乱序**的问题可以分别使用**优化屏障 (Optimization Barrier)**和**内存屏障 (Memory Barrier)**这两个机制来解决：

优化屏障 (Optimization Barrier)：避免编译器的重排序优化操作，保证编译程序时在优化屏障之前的指令不会在优化屏障之后执行。这就保证了编译时期的优化不会影响到实际代码逻辑顺序。

IA-32/AMD64架构上，在Linux下常用的GCC编译器上，优化屏障定义为（linux kernel, include/linux/compiler-gcc.h）：

```
1
2
#define barrier() __asm__ __volatile__("": : :"memory")
```

优化屏障告知编译器：

1. 内存信息已经修改，屏障后的寄存器的值必须从内存中重新获取
2. 必须按照代码顺序产生汇编代码，不得越过屏障

> C/C++的volatile关键字也能起到优化限制的作用，但是和Java中的volatile（Java 5之后）不同，C/C++中的volatile不提供任何防止乱序的功能，也并不保证访存的原子性。

内存屏障 (Memory Barrier)分为**写屏障（Store Barrier）**、**读屏障（Load Barrier）**和**全屏障（Full Barrier）**，其作用有两个：

1. 防止指令之间的重排序

2. 保证数据的可见性

关于第一点，关于指令重排，这里不考虑架构的话，Load和Store两种操作会有Load-Store、Store-Load、Load-Load、Store-Store这四种可能的乱序结果。 上文提到的三种屏障则是限制这些不同乱序的机制。

关于第二点。写屏障会阻塞直到把Store Buffer中的数据刷到Cache中；读屏障会阻塞直到Invalid Queue中的消息执行完毕。以此来保证核间各级数据的一致性。

这里要强调，内存屏障解决的只是顺序一致性的问题，不解决Cache一致性的问题（这是Cache一致性协议的责任，也不需要程序员关注）。Store Buffer和Load Buffer等组件是属于流水线的一部分，和Cache无关。这里一定要区分清楚这两点，Cache一致性协议只是保证了**Cache一致性（Cache Coherence）**，但是不关注**顺序一致性（Sequential Consistency）**的问题。比如，一个处理器对某变量A的写入操作仅比另一个处理器对A的读取操作提前很短的一点时间，那就不一定能确保该读取操作会返回新写入的值。这个**新写入的值多久之后能确保被读取操作读取到**，这是**内存一致性模型（Memory Consistency Models）**要讨论的问题。

完全的确保顺序一致性需要很大的代价，不仅限制编译器的优化，也限制了CPU的执行效率。为了更好地挖掘硬件的并行能力，现代的CPU多半都是介于两者之间，即所谓的**宽松的内存一致性模型（Relaxed Memory Consistency Models）**。不同的架构在重排上有各自的尺度，在严格排序和自由排序之间会有各自的偏向。偏向严格排序的一边，称之为**强模型（Strong Model）**，而偏向于自由排序的一边，称之为**弱模型（Weak Model）**。AMD64架构是强模型：

Memory ordering in some architectures[7][8]

Type	Alpha	ARMv7	PA-RISC	POWER	SPARC RMO	SPARC PSO	SPARC TSO	x86	x86 oostore	AMD64	IA-64	z/Architecture
Loads reordered after loads	Y	Y	Y	Y	Y				Y		Y	
Loads reordered after stores	Y	Y	Y	Y	Y				Y		Y	
Stores reordered after stores	Y	Y	Y	Y	Y	Y			Y		Y	
Stores reordered after loads	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Atomic reordered with loads	Y	Y		Y	Y						Y	
Atomic reordered with stores	Y	Y		Y	Y	Y					Y	
Dependent loads reordered	Y											
Incoherent instruction cache pipeline	Y	Y		Y	Y	Y	Y	Y	Y		Y	

Some older x86 and AMD systems have weaker memory ordering[9]

特别地，早先时候，AMD64架构也会有Load-Load乱序发生（Memory Ordering in Modern Microprocessors, PaulE.McKenney, 2006）。

	Loads Reordered After Loads?	Loads Reordered After Stores?	Stores Reordered After Stores?	Stores Reordered After Loads?	Atomic Instructions Reordered With Loads?	Atomic Instructions Reordered With Stores?	Dependent Loads Reordered?	Incoherent Instruction Cache/Pipeline?
Alpha	Y	Y	Y	Y	Y	Y	Y	Y
AMD64	Y			Y				
IA64	Y	Y	Y	Y	Y	Y		Y
(PA-RISC)	Y	Y	Y	Y				
PA-RISC CPUs								
POWER™	Y	Y	Y	Y	Y	Y		Y
(SPARC RMO)	Y	Y	Y	Y	Y	Y		Y
(SPARC PSO)			Y	Y		Y		Y
SPARC TSO				Y				Y
x86	Y	Y		Y				Y
(x86 OOStore)	Y	Y	Y	Y				Y
zSeries®				Y				Y

注意这里的IA-64（Intanium Processor Family）是弱模型，它和Intel® 64不是一回事。后者是从AMD交叉授权来的，源头就是AMD64架构。这里不讨论历史，只需要知道平时说的x86-64/x64就是指的AMD64架构即可。

《Intel® 64 and IA-32 Architectures Software Developer's Manual》有如下的阐述：

8.2.2 Memory Ordering in P6 and More Recent Processor Families

The Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium 4, and P6 family processors also use a processor-ordered memory-ordering model that can be further defined as "write ordered with store-buffer forwarding." This model can be characterized as follows.

In a single-processor system for memory regions defined as write-back cacheable, the memory-ordering model respects the following principles (**Note** the memory-ordering principles for single-processor and multiple-processor systems are written from the perspective of software executing on the processor, where the term "processor" refers to a logical processor. For example, a physical processor supporting multiple cores and/or HyperThreading Technology is treated as a multi-processor systems.):

- Reads are not reordered with other reads.
- Writes are not reordered with older reads.
- Writes to memory are not reordered with other writes, with the following exceptions:
  - writes executed with the CLFLUSH instruction;
  - streaming stores (writes) executed with the non-temporal move instructions (MOVNTI, MOVNTQ, MOVNTDQ, MOVNTPS, and MOVNTPD); and
  - string operations (see Section 8.2.4.1).
- Reads may be reordered with older writes to different locations but not with older writes to the same location.
- Reads or writes cannot be reordered with I/O instructions, locked instructions, or serializing instructions.
- Reads cannot pass earlier LFENCE and MFENCE instructions.
- Writes cannot pass earlier LFENCE, SFENCE, and MFENCE instructions.
- LFENCE instructions cannot pass earlier reads.
- SFENCE instructions cannot pass earlier writes.
- MFENCE instructions cannot pass earlier reads or writes.

In a multiple-processor system, the following ordering principles apply:

- Individual processors use the same ordering principles as in a single-processor system.
- Writes by a single processor are observed in the same order by all processors.
- Writes from an individual processor are NOT ordered with respect to the writes from other processors.
- Memory ordering obeys causality (memory ordering respects transitive visibility).
- Any two stores are seen in a consistent order by processors other than those performing the stores

简单翻译一下：

> - 读操作之间不能重新排序
> - 写操作不能跟旧的读操作排序
> - 主存写操作不能跟其他的写操作排序，但是以下情况除外：
> - 带有CLFLUSH（失效缓存）指令的写操作
> - 带有non-temporal move指令的流存储（写入）（MOVNTI, MOVNTQ, MOVNTDQ, MOVNTPS, 和 MOVNTPD，都是SSE/SSE2扩展的指令）
> - 字符串操作（REP STOSD等）
> - 不同内存地址的读可以与较早的写排序，同一地址的情况除外
> - 对I/O指令、锁指令、序列化指令的读写不能重排序
> - 读不能越过较早的读屏障指令（LFENCE）或者全屏障指令（MFENCE）
> - 写不能越过较早的读屏障指令（LFENCE）、写屏障指令（SFENCE）和全屏障指令（MFENCE）
> - 读屏障指令（LFENCE）不能越过较早的读
> - 写屏障指令（SFENCE）不能越过较早的写
> - 全屏障指令（MFENCE）不能越过较早的读和写

> 在多处理器的情况下，单处理器内部的内存访问排序仍然依照以上的原则，并且规定处理器与处理器之间遵循如下的原则：
>
> - 某个处理器的全部写操作以同样的顺序被其它处理器观察到
> - 不同处理器之间的写操作不重排序
> - 排序遵循逻辑上的因果关系
> - 第三方总是观察到一致的写操作顺序

那么上文提到的四种可能的乱序在AMD64下明确说明不会有Load-Load乱序、Load-Store乱序，**明确会出现Store-Load乱序**，Store-Store乱序除了几种例外的情况也不会出现。参考文献5中给出了在Linux下重现出Store-Load乱序的代码，有兴趣的读者可以自行测试。

但是内存一致性模型不仅仅是没有指令重排就会保证一致的。但是如果仅仅只考虑指令重排，完全按照该规则来思考，就会遇到违反直觉的事情。特别的，在对写缓存的同步处理上，AMD64内存访问模型的 **Intra-Processor Forwarding Is Allowed**这个特性比较要命：

8.2.3.5 Intra-Processor Forwarding Is Allowed

The memory-ordering model allows concurrent stores by two processors to be seen in different orders by those two processors; specifically, each processor may perceive its own store occurring before that of the other. This is illustrated by the following example:

Example 8-5. Intra-Processor Forwarding is Allowed

Processor 0	Processor 1
mov [_x], 1 mov r1, [_x] mov r2, [_y]	mov [_y], 1 mov r3, [_y] mov r4, [_x]
Initially x = y = 0 r2 = 0 and r4 = 0 is allowed	

The memory-ordering model imposes no constraints on the order in which the two stores appear to execute by the two processors. This fact allows processor 0 to see its store before seeing processor 1's, while processor 1 sees its store before seeing processor 0's. (Each processor is self consistent.) This allows r2 = 0 and r4 = 0.

In practice, the reordering in this example can arise as a result of store-buffer forwarding. While a store is temporarily held in a processor's store buffer, it can satisfy the processor's own loads but is not visible to (and cannot satisfy) loads by other processors.

只考虑指令重排的话，AMD64架构既然不会有Load-Load重排的，r2=r4=0就不可能会出现，但是实际的结果是违反直觉的。出现这个现象的原因就是Intel对Store Buffer的处理上，Store Buffer的修改对其他CPU核心是不可见的。Processor 0对*x的修改缓存在了Processor 0的Store Buffer中，还未提交到L1 Cache，自然也不会失效掉Processor 1的L1 Cache中的相关行。Processor 1对*y的修改同理。

对于以上问题，AMD64提供了三个内存屏障指令来解决：

The SFENCE, LFENCE, and MFENCE instructions provide a performance-efficient way of ensuring load and store memory ordering between routines that produce weakly-ordered results and routines that consume that data. The functions of these instructions are as follows:

- **SFENCE** — Serializes all store (write) operations that occurred prior to the SFENCE instruction in the program instruction stream, but does not affect load operations.
- **LFENCE** — Serializes all load (read) operations that occurred prior to the LFENCE instruction in the program instruction stream, but does not affect store operations.[1]
- **MFENCE** — Serializes all store and load operations that occurred prior to the MFENCE instruction in the program instruction stream.

sfence指令为写屏障（Store Barrier），作用是：

1. 保证了sfence前后Store指令的顺序，防止Store重排序
2. 通过刷新Store Buffer保证sfence之前的Store要指令对全局可见

lfence指令读屏障（Load Barrier），作用是：

1. 保证了lfence前后的Load指令的顺序，防止Load重排序
2. 刷新Load Buffer

mfence指令全屏障（Full Barrier），作用是：

1. 保证了mfence前后的Store和Load指令的顺序，防止Store和Load重排序
2. 保证了mfence之后的Store指令全局可见之前，mfence之前的Store指令要先全局可见

如前文所说，AMD64架构上是不存在Load-Load重排的，但是当一个CPU核心收到其他CPU核心失效Cache Line的消息后，立即回复给对方一个应答信号。但是此时并没有立即失效掉Cache Line，而是将其包装成一个结构投递到自身的Load Buffer里。AMD64架构上不存在Load-Load重排并不意味着流水线真的就一条一条执行Load指令。在保证两个CPU核看到的Store顺序一致的情况下，是允许Load乱序的。比如连续的两个访存指令，指令1 Cache Miss，指令2 Cache Hit，实际上指令2是不会真的等待指令1的Load完成整个Cache替换过程后才执行的。实际流水线的实现中，Load先是乱序执行，然后有一个Load-ordering-Buffer（Load Buffer）的结构，在Load Commit之前检测冲突，Load过的地址是否又被其他CPU核心写过（没有存在失效信息）。只要没有冲突，这种乱序就是安全的。如果发生冲突，这种乱序就违反x86要求，需要被取消并Flush流水线。而上文提到的lfence指令会刷新Load Buffer，保证当前CPU核心立即读取到最新的数据。

另外， 除了显式的内存屏障指令，有些指令也会造成指令保序的效果，比如I/O操作的指令、exch等原子交换的指令，任何带有lock前缀的指令以及CPUID等指令都有内存屏障的作用。

说了这么多，环形队列（Ring buffer）在IA-32/AMD64架构上到底怎么实现才能保证安全？Linux Kernel里的KFIFO的实现可以拿来参考（include/linux/kfifo.h）：

```
unsigned int __kfifo_put(struct kfifo *fifo, const unsigned char *buffer,
unsigned int len)
{
    unsigned int l;

    len = min(len, fifo->size - fifo->in + fifo->out);









    smp_mb();


    l = min(len, fifo->size - (fifo->in & (fifo->size - 1)));
    memcpy(fifo->buffer + (fifo->in & (fifo->size - 1)), buffer, l);


```

```
    memcpy(fifo->buffer, buffer + l, len - l);









    smp_wmb();

    fifo->in += len;

    return len;
}

unsigned int __kfifo_get(struct kfifo *fifo, unsigned char *buffer,
unsigned int len)
{
    unsigned int l;

    len = min(len, fifo->in - fifo->out);









    smp_rmb();


    l = min(len, fifo->size - (fifo->out & (fifo->size - 1)));
    memcpy(buffer, fifo->buffer + (fifo->out & (fifo->size - 1)), l);


    memcpy(buffer + l, fifo->buffer, len - l);









    smp_mb();

    fifo->out += len;

    return len;
}
```

代码中的`smp_wmb()`、`smp_rmb()`和`smp_mb()`在AMD64架构上分别对应`sfence`、`lfence`、`mfence`指令。但是Linux Kernel的代码要兼容所有的SMP架构，还要考虑很多弱内存模型的架构。所以这里的内存同步操作很多，但是不一定在AMD64上是必要的。当然，如果要考虑跨平台跨架构的代码，这样做是最保险的（另外Linux Kernel 4.0上KFIFO这个数据结构变化很大，内存同步操作也仅剩下`smp_wmb()`，这个还没顾得上研究）。

如果IA-32/AMD64架构下，Ring Buffer如果要实现单Reader和单Writer不需要内存同步，需要满足哪些特性呢?

以下面的定义为例：

```
struct ring_buffer {
    uint32_t read_index;
    uint32_t write_index;
    uchar_t buffer[BUFF_LEN];
};
```

首先，`read_index`和`write_index`的写入操作必须是原子的，这就要求**这两个变量本身在P6 Family及以后的CPU上至少是不能跨Cache行的**。同时如果是32-bit的变量则P6之前的CPU还要保持32-bit字节对齐，如果是64-bit变量在IA-32上无法保障（IA-32下64bit的变量Store操作不是原子的）。另外，为了避免False Sharing，这两个变量最好按照Cache行对齐，即：

```
struct ring_buffer {
    uint32_t read_index __attribute__ ((aligned(64)));
    uint32_t write_index __attribute__ ((aligned(64)));
    uchar_t buffer[BUFF_LEN];
};
```

然后在**入队和出队的地方插入编译屏障禁止掉编译器优化**，根据Intel的文档，就能保证不会出现乱序问题：

> 主存写操作不能跟其他的写操作排序，但是以下情况除外：
>
> - 带有CLFLUSH（失效缓存）指令的写操作
> - 带有non-temporal move指令的流存储（写入）（MOVNTI, MOVNTQ, MOVNTDQ, MOVNTPS, 和 MOVNTPD，都是SSE/SSE2扩展的指令）
> - 字符串操作（REP STOSD等）
>
> 在多处理器的情况下，单处理器内部的内存访问排序仍然依照以上的原则，并且规定处理器与处理器之间遵循如下的原则：
>
> - 某个处理器的全部写操作以同样的顺序被其它处理器观察到
> - 第三方总是观察到一致的写操作顺序

至于串操作，对buffer的修改可能是memcpy之类的操作，而对index的操作是普通赋值。memcpy在某些库中的实现使用了串操作指令又会怎样？会导致Store操作乱序吗？Intel有如下的说明：

Example 8-13. String Operations Are not Reordered with later Stores

Processor 0	Processor 1
rep:stosd [_x] mov [_z], $1	mov r1, [_z] mov r2, [_y]
Initially on processor 0: EAX = 1, ECX=128, ES:EDI =_x Initially [_y] = [_z] = 0, [_x] to 511[_x]= 0, _x <= _y < _x+512, _z is a separate memory location r1 = 1 and r2 = 0 is not allowed	

Processor 1 cannot perceive the later store by processor 0 until it sees all the stores from the string operation. Example 8-13 assumes that processor 0's store to [_z] is not executed while the string operation has been interrupted. If the string operation is interrupted and the store to [_z] by processor 0 is executed by the interrupt handler, then changes to [_z] will become visible before the string operation resumes.

所以不担心index的修改出现在rep:stosd之前。但是这样做是有这样的前提的，即**Reader和Writer当前的修改不需要立即被对方知晓，即允许一段时间内的"不一致"**。否则，必然需要内存屏障来确保修改操作全局一致。

以上的结论很容易引起口水仗，所以这里再次强调该结论只是在AMD64架构下，且不考虑可移植性的情况下成立。但是，按照我个人看法，这几个屏障指令不见得在所有Intel的CPU上都是有意义的，甚至有些屏障指令在Intel某些CPU上没有该屏障本身的语义。比如lfence本意是限制Load重排，然而AMD64就没有Load-Load乱序（内存可见性另说）。这几个屏障指令更像是Intel提供给软件开发者的一个Interface，在需要加屏障的地方让开发者加吧。至于实际上需不需要，CPU本身会判断，如果不需要的话直接由CPU直接NOP掉即可。这也是一种长远的考虑，那你问我在AMD64架构的CPU上写代码的时候，需要强一致的时候加不加屏障？那当时是要加的。按照Interface写代码是最保险的，万一Intel以后出一个采用弱一致模型的CPU（替被市场淘汰的IA-64默哀三分钟），遗留代码出点问题就不好了。

下面说说锁和原子变量。对于数据竞争（Data Races）的情况，最简单和最常见的场景就是使用Mutex了，包括并不限于互斥锁、自旋锁、读写锁等。拿互斥锁来说，除了保护临界区只能有一个执行流之外，还有其他的作用。这里要引入**宽松的内存一致性模型（Relaxed Memory Consistency Models）**中的**Release Consistency**模型[6]来解释，这个模型包含了同步操作**Acquire**和**Release**：

1. Acquire：在此操作后的所有读写操作必然发生在Acquire这个动作之后
2. Release：在此操作前的所有读写操作必然发生在Release这个动作之前

要注意的是Acquire和Release都只保证了一半的顺序：

- 对于Acquire来说，并没保证Acquire前的读写操作不会发生在Acquire动作之后
- 对于Release来说，并没保证Release后的读写操作不会发生在Release动作之前

因此Acquire和Release的组合便形成了内存屏障。

Mutex的Lock操作暗含了Acquire语义，Unlock暗含了Release语义。**这里是脱离架构在讨论的，在具体的平台上如果Load和Store操作暗含Acquire和Release语义的话自然保证一致，否则可以是相关的内存屏障指令。**所以Mutex不仅会保证执行的序列化，同时也保证了访存的一致性。与之类似，**平台提供的原子变量除了保证内存操作原子之外，也会保证访存的一致性。**

GCC提供了Built-in的原子操作函数可以使用，GCC 4以后的版本也提供了Built-in的屏障函数`__sync_synchronize()`，这个屏障函数既是编译屏障又是内存屏障，代码插入这个函数的地方会被安插一条`mfence`指令。不过GCC 4.4以上才支持`mfence`，这个问题的讨论（bug？）在这里，Patch在这里。

实际上无锁的代码仅仅是不需要显式的Mutex来完成，但是存在数据竞争（Data Races）的情况下也会涉及到**同步（Synchronization）**的问题。从某种意义上来讲，所谓的无锁，仅仅只是颗粒度特别小的"锁"罢了，从代码层面上逐渐降低级别到CPU的指令级别而已，总会在某个层级上付出等待的代价，除非逻辑上彼此完全无关。另外，`Lockfree`和`Lockless`是两个概念，但这个话题太大，我个人尚且拿捏不住，就此打住。至于工程上，普通的程序员老老实实的用Mutex就好了，普通的计数类场景用原子变量也无可厚非。诸如无锁队列这种能明确证明其正确性的数据结构在一些场合也是很有价值的，用用无妨（但是多说一句，CAS这种乐观锁在核数很多的时候不见得高效，竞争太厉害的时候总体消耗很可能超出普通的锁）。但是如果不能做到在任何时候都能想明白顺序一致性的话，还是老老实实的用Mutex吧，否则造成的麻烦可比提升的这一点点效率折腾多了。

最后，讨论这些问题的文章太多了，各路说法到处飞，我也不敢保证这篇文章的说法全部正确，但至少我觉得是可以自圆其说的。如果你觉得哪里的描述有问题，不妨一起讨论，我们一起纠正这些错误的观点。

文章的撰写过程中参考了若干资料，下面列出的参考的资料和文章中，个别文章我只是"部分同意"原作者的观点，因为引用了作者部分说法，所以一并列出。这不代表我完全同意原作者观点，具体细节请读者自行判断（有了冲突，自然是以Intel最新文档的说法为准）。

参考文献

[1] Intel® 64 and IA-32 Architectures Software Developer's Manual, https://software.intel.com/en-us/articles/intel-sdm

[2] Memory Barriers/Fences, https://mechanical-sympathy.blogspot.jp/2011/07/memory-barriersfences.html

[3] Memory Barriers: a Hardware View for Software Hackers, Paul E. McKenney, Linux Technology Center, IBM Beaverton, https://www.researchgate.net/publication/228824849_Memory_Barriers_a_Hardware_View_for_Software_Hackers

[4] 为什么程序员需要关心顺序一致性（Sequential Consistency）而不是Cache一致性（Cache Coherence）?, http://www.parallellabs.com/2010/03/06/why-should-programmer-care-about-sequential-consistency-rather-than-cache-coherence/

[5] 一个关于Memory Reordering的实验, http://blog.csdn.net/yxc135/article/details/11747995

[6] 计算机体系结构：量化研究方法, （美）亨尼西 等著, 机械工业出版社, 2012-1-1

[7] http://stackoverflow.com/questions/23603304/java-8-unsafe-xxxfence-instructions

[8] Intel Sandy Bridge Configuration, http://www.7-cpu.com/cpu/SandyBridge.html

[9] Intel's Haswell CPU Microarchitecture, http://www.realworldtech.com/haswell-cpu/5/

[10] Write Combining, http://mechanical-sympathy.blogspot.com/2011/07/write-combining.html

[11] Memory ordering, https://en.wikipedia.org/wiki/Memory_ordering