说说无锁(Lock-Free)编程那些事

1. 引言

现代计算机，即使很小的智能机亦或者平板电脑，都是一个多核(多CPU)处理设备，如何充分利用多核CPU资源，以达到单机性能的极大化成为我们码农进行软件开发的痛点和难点。在多核服务器中，采用多进程或多线程来并行处理任务，俨然成为了大家性能调优的标准解决方案。多进程(多线程)的并行编程方式，必然要面对共享数据的访问问题，如何并发、高效、安全地访问共享数据资源，成为并行编程的一个重点和难点。 传统的共享数据访问方式是采用同步原语(临界区、锁、条件变量等)来达到共享数据的安全访问，然而，同步恰恰和并行编程是对立的，很容易成为并行程序中的瓶颈。一方面，有些同步原语是操作系统的内核对象，调用该原语会带来昂贵的上下文切换(用户态切换到内核态)代价，同时，内核对象是一个比较有限的资源。另一方面，同步杜绝了并行操作，一个线程在访问共享数据的时候，其他的多个线程必须在排队空闲等待，同时，同步可扩展性很弱，随着并行线程的增加，很容易成为程序的一个瓶颈，甚至出现，服务性能吞吐量并没CPU核数增加并发线程的增加呈现线性增长，相反出现下降的情况。 于是，人们开始研究对共享数据进行并发访问的数据结构和算法，通常有以下几方面：

```
1. Transactional memory --- 事务性内存
2. Fine-grained algorithms --- 细粒度(锁)算法
3. Lock-free data structures --- 无锁数据结构
```

(1) 事务内存（Transactional memory）TM是一个软件技术，简化了并发程序的编写。 TM借鉴了在数据库社区中首先建立和发展起来的概念， 基本的想法是要申明一个代码区域作为一个事务。一个事务（transaction ） 执行并原子地提交所有结果到内存（如果事务成功），或中止并取消所有的结果（如果事务失败）。 TM的关键是提供原子性（Atomicity），一致性（Consistency ）和隔离性（Isolation ）这些要素。 事务可以安全地并行执行，以取代现有的痛苦和容易犯错误(下面几点)的技术，如锁和信号量。 还有一个潜在的性能优势。 我们知道锁是悲观的（pessimistic ），并假设上锁的线程将写入数据，因此，其他线程的进展被阻塞。 然而访问锁定值的两个事务可以并行地进行，且回滚只发生在当事务之一写入数据的时候。但是，目前还没有嵌入式的事务内存，比较难和传统代码集成，需要软件做出比较大的变化，同时，软件TM性能开销极大，2-10倍的速度下降是常见的，这也限制了软件TM的广泛使用

```
1. 因为忘记使用锁而导致条件竞争(race condition)
2. 因为不正确的加锁顺序而导致死锁(deadlock)
3. 因为未被捕捉的异常而造成程序崩溃(corruption)
4. 因为错误地忽略了通知，造成线程无法正常唤醒(lost wakeup)
```

(2) 细粒度(锁)算法是一种基于另类的同步方法的算法，它通常基于“轻量级的”原子性原语(比如自旋锁)，而不是基于系统提供的昂贵消耗的同步原语。细粒度(锁)算法适用于任何锁持有时间少于将一个线程阻塞和唤醒所需要的时间的场合，由于锁粒度极小，在此类原语之上构建的数据结构，可以并行读取，甚至并发写入。Linux 4.4以前的内核就是采用_spin_lock自旋锁这种细粒度锁算法来安全访问共享的listen socket，在并发连接相对轻量的情况下，其性能和无锁性能相媲美。然而，在高并发连接的场景下，细粒度(锁)算法就会成为并发程序的瓶颈所在。

(3) 无锁数据结构，为解决在高并发场景下，细粒度锁无法避免的性能瓶颈，将共享数据放入无锁的数据结构中，采用原子修改的方式来访问共享数据。

目前，常见的无锁数据结构主要有：无锁队列(lock free queue)、无锁容器(b+tree、list、hashmap等)。 本文以一个无锁队列实现片段为蓝本，来谈谈无锁编程中的那些事。下面是一个开源C++并发数据结构lib中的无锁队列的实现片段(https://github.com/khizmax/libcds)

```
371         bool do_dequeue( dequeue_result& res )
372         {
373             node_type * pNext;
374             back_off bkoff;
375
376             node_type * h;
377             while ( true ) {
378                 h = res.guards.protect( 0, m_pHead, []( node_type * p ) -> value_type * { return node_traits::to_value_ptr( p );});
379                 pNext = res.guards.protect( 1, h->m_pNext, []( node_type * p ) -> value_type * { return node_traits::to_value_ptr( p );});
380                 if ( m_pHead.load(memory_model::memory_order_acquire) != h )
381                     continue;
382
383                 if ( pNext == nullptr ) {
384                     m_Stat.onEmptyDequeue();
385                     return false;    // empty queue
386                 }
387
388                 node_type * t = m_pTail.load(memory_model::memory_order_acquire);
389                 if ( h == t ) {
390                     // It is needed to help enqueue
391                     m_pTail.compare_exchange_strong( t, pNext, memory_model::memory_order_release, atomics::memory_order_relaxed );
392                     m_Stat.onBadTail();
393                     continue;
394                 }
395
396                 if ( m_pHead.compare_exchange_strong( h, pNext, memory_model::memory_order_acquire, atomics::memory_order_relaxed ))
397                     break;
398
399                 m_Stat.onDequeueRace();
400                 bkoff();
401             }
402
403             --m_ItemCounter;
404             m_Stat.onDequeue();
405
406             res.pHead = h;
407             res.pNext = pNext;
408             return true;
409         }
```

```
475         bool enqueue( value_type& val )
476         {
477             node_type * pNew = node_traits::to_node_ptr( val );
478             link_checker::is_empty( pNew );
479
480             typename gc::Guard guard;
481             back_off bkoff;
482
483             node_type * t;
484             while ( true ) {
485                 t = guard.protect( m_pTail, []( node_type * p ) -> value_type * { return node_traits::to_value_ptr( p );});
486
487                 node_type * pNext = t->m_pNext.load(memory_model::memory_order_acquire);
488                 if ( pNext != nullptr ) {
489                     // Tail is misplaced, advance it
490                     m_pTail.compare_exchange_weak( t, pNext, memory_model::memory_order_release, atomics::memory_order_relaxed );
491                     m_Stat.onBadTail();
492                     continue;
493                 }
494
495                 node_type * tmp = nullptr;
496                 if ( t->m_pNext.compare_exchange_strong( tmp, pNew, memory_model::memory_order_release, atomics::memory_order_relaxed ))
497                     break;
498
499                 m_Stat.onEnqueueRace();
500                 bkoff();
501             }
502             ++m_ItemCounter;
503             m_Stat.onEnqueue();
504
505             if ( !m_pTail.compare_exchange_strong( t, pNew, memory_model::memory_order_release, atomics::memory_order_relaxed ))
506                 m_Stat.onAdvanceTailFailed();
507             return true;
508         }
```

上面是一个普通单向链表队列的无锁实现，对比普通的链表队列实现，无锁实现复杂了很多，多出了很多独有的特征操作：

1. C++11 标准的原子性操作： load、store、compare_exchange_weak、compare_exchange_strong
2. 一个无限循环： while (true) { ... }
3. 局部变量的安全性(guards)：t = guard.protect(m_pTail, node_to_value());
4. 补偿策略(functor bkoff)：这不是必须的，但可以在连接很多的情况下缓解处理器的压力，尤其是多个线程逐个地调用队列时。
5. helping方法：本例中，dequeue中帮助enqueue将m_pTail设置正确。

```
// It is needed to help enqueue
m_pTail.compare_exchange_strong( t, pNext, memory_model::memory_order_release,
                memory_model::memory_order_relaxed );
```

6. 标准原子操作中使用的内存模型(memory model)，也就是内存栅栏(屏障)：memory_order_release、memory_order_acquire等

下面分别讲一下上面提到无锁队列实现中的6个特征。

2. 原子性、原子性原语

我们知道无论是何种情况，只要有共享的地方，就离不开同步，也就是concurrency。对共享资源的安全访问，在不使用锁、同步原语的情况下，只能依赖于硬件支持的原子性操作，离开原子操作的保证，无锁编程(lock-free programming)将变得不可能。 留意本例的无锁队列的实现例子，我们发现原子性操作可以简单划分为两部分：

1. 原子性读写(atomic read and write)：本例中的原子load(读)、原子store(写)
2. 原子性交换(Atomic Read-Modify-Write -- RMW)：本例中的compare_exchange_weak、compare_exchange_strong

原子操作可认为是一个不可分的操作；要么发生，要么没发生，我们看不到任何执行的中间过程，不存在部分结果(partial effects)。可以想象的到，原子操作要保证要么全部发生，要么全部没发生，这样原子操作绝对不是一个廉价的消耗低的指令，相反，原子操作是一个较为昂贵的指令。那么在无锁编程中，我们要避免滥用原子操作，那么什么情况下，我们需要对共享变量的操作采用原子操作呢？对变量的普通的读取赋值操作是原子的吗？ 通常情况下，我们有一个对共享变量必须使用原子操作的规则：

> 任何时刻，只要存在两个或多个线程并发地对同一个共享变量进行操作，并且这些操作中的其中一个是执行了写操作，那么所有的线程都必须使用原子操作。

如果违反上面的规则，即存在某个线程使用了非原子操作，那么你将会陷入一个在C++11标准中称之为数据竞争(data race)（这里的数据竞争和Java中的data race概念，以及更通用的race condition是不一样的）的情形。如果你引发了数据竞争，那么就会得到一个"未定义行为（undefined behavior）"的结果，它们会导致torn reads(撕裂读)和torn writes(撕裂写)，也就是一个非完整的读写。 什么样的内存操作是原子的呢？通常情况下，如果一个内存操作使用了多条CPU指令，那么这个内存操作是非原子的。那么只使用一条CPU指令的内存操作是不是就一定是原子的呢？答案是不一定，某些仅仅使用一条CPU的内存操作，在绝大多数CPU架构上是原子，但是，在个别CPU架构上是非原子的。如果，我们想写出可移植的代码，就不能做出使用一条CPU指令的内存操作一定是原子的假设。在C/C++中，所有的内存操作都被假定为非原子性的，即使是普通的32位整形赋值，除非编译器或硬件厂商有特殊说明这个赋值操作是原子的。在所有的现代x86，x64，Itanium，SPARC，ARM和PowerPC处理器中，普通的32位整形，只要内存地址是对齐的，那么赋值操作就是原子操作，这个保证是特定平台下编译器和处理器做出的保证。由于C/C++语言标准并没对整型赋值是原子操作做出保证，于是，要想写出真正可移植的C和C++代码时，我们只能使用C++11提供的原子库(C++11 atomic library)来保证对变量的load(读)和store(写)是原子的。

2.1 不能不说的关键字：volatile

通过上面我们知道，在现代处理器中，对于一个对齐的整形类型(整形或指针)，其读写操作是原子的，而对于现代编译器，用volatile修饰的基本类型正确对齐的保障，并且限制了编译器对其优化。这样通过对int变量加上volatile修饰，我们就能对该变量进行原子性读写。

```
volatile int i=10;//用volatile修饰变量i
......//something happened
int b = i;//atomic read
```

由于volatile 在某种程度上限制了编译器的优化，而很多时候，对于同一个变量，我们在某些地方有原子性读写的需求，在某些地方我们又不需要原子性读写，这个时候希望编译器该优化的时候就优化。然而，不加volatile修饰，那么就做不到前面一点。加了volatile，后面这一方面就无从谈起，怎么办？其实，这里有个小技巧可以达到这个目的：

```
int i = 2; //变量i还是不用加volatile修饰
#define ACCESS_ONCE(x) (*(volatile typeof(x) *)&(x))
#define READ_ONCE(x) ACCESS_ONCE(x)
#define WRITE_ONCE(x, val) ({ ACCESS_ONCE(x) = (val); })
a = READ_ONCE(i);
WRITE_ONCE(i, 2);
```

通过上面我们知道，用volatile修饰的int在现代处理器中，能够做到原子性的读写，并且限制编译器的优化，每次都是从内存中读取最新的值，很多同学就误以为volatile能够保证原子性并且具有Memery Barrier的作用。其实vloatile既不能保证原子性，也不会有任何的Memery Barrier(内存栅栏)的保证。上面例子中，volatile仅仅是保证int的地址对齐，而对齐后的整形在现代处理器中，是能够做到原子性读写的。在C++中volatile具有以下特性：

```
1. 易变性：所谓的易变性，在汇编层面反映出来，就是两条语句，下一条语句不会直接使用上一条语句对应的volatile变量的寄存器内容，而是重新从内存中读取。
2. "不可优化"性：volatile告诉编译器，不要对我这个变量进行各种激进的优化，甚至将变量直接消除，保证程序员写在代码中的指令，一定会被执行。
3. "顺序性"：能够保证Volatile变量间的顺序性，编译器不会进行乱序优化。Volatile变量与非Volatile变量的顺序，编译器不保证顺序，可能会进行乱序优化。
```

2.2 Compare-And-Swap(CAS)

对于CAS相信大家都不陌生，在学术圈，compare-and-swap (CAS) 被认为是最基础的一种原子性RMW操作，其伪代码如下：

```
bool CAS( int * pAddr, int nExpected, int nNew )
atomically {
    if ( *pAddr == nExpected ) {
         *pAddr = nNew ;
         return true ;
    }
    else
        return false ;
}
```

上面的CAS返回bool告知原子性交换是否成功，然而在有些应用场景中，我们希望CAS 失败后，能够返回内存单元中的当前值，于是就有一个称为 valued CAS的变种，伪代码如下：

```
int CAS( int * pAddr, int nExpected, int nNew )
atomically {
      if ( *pAddr == nExpected ) {
           *pAddr = nNew ;
           return nExpected ;
       }
       else
            return *pAddr;
}
```

CAS作为最基础的RMW操作，其他所有RMW操作都可以通过CAS来实现，例如 fetch-and-add(FAA)，伪代码如下：

```
int FAA( int * pAddr, int nIncr )
{
     int ncur = *pAddr;
     do {} while ( !compare_exchange( pAddr, ncur, ncur + nIncr ) ;//compare_exchange失败会返回当前值于ncur
     return ncur ;
}
```

在C++11的原子lib中，主要有以下RMW操作：

```
std::atomic<>::fetch_add()
std::atomic<>::fetch_sub()
std::atomic<>::fetch_and()
std::atomic<>::fetch_or()
std::atomic<>::fetch_xor()
std::atomic<>::exchange()
std::atomic<>::compare_exchange_strong()
std::atomic<>::compare_exchange_weak()
```

其中compare_exchange_weak()就是最基础的CAS，使用compare_exchange_weak()我们可以实现其他所有的RMW操作，C++11 atomic library中的原子RMW操作有点少，不能满足我们实际需求，我们可以自己动手实现自己需要的原子RMW操作。 例如：我们需要一个原子对内存中值执行乘法，也就是 atomic fetch_multiply，实现伪代码如下：

```
uint32_t fetch_multiply(std::atomic<uint32_t>& shared, uint32_t multiplier)
{
    uint32_t oldValue = shared.load();
    while (!shared.compare_exchange_weak(oldValue, oldValue * multiplier))
    {
    }
    return oldValue;
}
```

以上的原子RMW操作都是只能对一个integer变量进行原子修改操作，如果我们想同时对两个integer变量进行原子操作，怎么实现呢？我们知道C++11的原子库std::atomic<>是一个模版，这样我们可以用一个结构体来包含两个integer变量，来对结构体进行原子修改，实现如下：

```
struct Terms
{
    uint32_t x;
    uint32_t y;
};
std::atomic<Terms> terms;

void atomicFibonacciStep()
{
    Terms oldTerms = terms.load();
    Terms newTerms;
    do
    {
        newTerms.x = oldTerms.y;
        newTerms.y = oldTerms.x + oldTerms.y;
    }
    while (!terms.compare_exchange_weak(oldTerms, newTerms));
}
```

到这里，可能大家会有疑问了，是不是terms.compare_exchange_weak(oldTerms, newTerms)在内部加了锁，要不怎么能够原子修改呢？ C++11的原子库std::atomic<> template可以是任何类型(int、bool等buil-in type，或user-defined type)，但并不是所有的类型的原子操作是lock-free的。C++11 标准库 std::atomic 提供了针对整形(integral)和指针类型的特化实现，其中 integal 代表了如下类型char, signed char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long, long long, unsigned long long, char16_t, char32_t, wchar_t，这些特化实现，都包含了一个is_lock_free()成员来用于判断该原子类型是原子操作是否是lock-free的。上面的例子中，在X64平台下，用GCC4.9.2编译出来的代码terms.compare_exchange_weak(oldTerms, newTerms)是lock-free的，在其他平台下就不能保证了。在实际应用中，通常情况下，同时满足以下条件的原子类的原子操作才能做出是lock-free的保证：

```
1. The compiler is a recent version MSVC, GCC or Clang.
2. The target processor is x86, x64 or ARMv7 (and possibly others).
3. The atomic type is std::atomic<uint32_t>, std::atomic<uint64_t> or std::atomic<T*>
for some type T.
```

2.3 Weak and Strong CAS

相信大家看到C++11的CAS操作有两个compare_exchange_weak和compare_exchange_strong，CAS怎么还有强弱之分呢？现代处理器架构对CAS的实现分成两大阵营：(1)实现了原子性CAS原语 -- X86、Intel Itanium、Sparc等处理器架构，最早实现于IBM System 370。(2)实现LL/SC对(load-linked/store-conditional) -- PowerPC, MIPS, Alpha, ARM 等处理器架构，最早实现于DEC，通过LL/SC对可以实现原子性CAS，但在一些情况下它并不具有原子性。为什么会存在LL/SC对的使用，而不直接实现CAS原语呢？要说明LL/SC对存在的原因，不得不说一下无锁编程中的一个棘手问题：ABA问题。

2.3.1 ABA问题

下面无锁堆栈的实现片段：

```
// Shared variables
static NodeType * Top = NULL; // Initially null

Push(NodeType * node) {
```

```
            do {
/*Push1*/       NodeType * t = Top;
/*Push2*/       node->Next = t;
/*Push3*/    } while ( !CAS(&Top,t,node) );
}

NodeType * Pop() {
          Node * next ;
          do {
/*Pop1*/         NodeType * t = Top;
/*Pop2*/         if ( t == null )
/*Pop3*/              return null;
/*Pop4*/         next = t->Next;
/*Pop5*/    } while ( !CAS(&Top,t,next) );
/*Pop6*/    return t;
}
```

假设当前堆栈有4个成员是：A-->B-->C-->D，A位于栈顶。下面的一个执行时序会导致一个栈被破坏的ABA问题：

1. Thread X执行Pop()操作，并在执行完/*Pop4*/这行代码后Thread X被切出去，这个时候对于Thread X来说， t == A； next == A->next == B；Top == A；当前栈：A-->B-->C-->D
2. Thread Y 执行NodeType * pTop=Pop()操作，接着又执行Pop(),最后执行Push(pTop)。这个时候当前栈变成了：A-->C-->D。
3. 这个时候Thread X被调度执行/*Pop5*/这行代码，由于栈顶元素依然是A，于是CAS(&Top,t,next)执行成功，Top变成指向了B，栈顶指针指向了一个已经不再栈中的元素B，整个栈被破坏了。

通过这个例子，我们知道ABA问题是所有基于CAS的无锁容器的一个灾难问题，要解决ABA问题有两个思路：

1. 不要重用容器中的元素，本例中，Pop出来的A不要直接Push进容器，应该new一个新的元素A_n出来然后在push进容器中。当然new一个新的元素也不绝对安全，如果是A先被delete了，接着调用new来new一个新的元素有可能会返回A的地址，这样还是存在ABA的风险。一般对于无锁编程中的内存回收采用延迟回收的方式，在确保被回收内存没有被其他线程使用的情况下安全回收内存。
2. 允许内存重用，对指向的内存采用标签指针(Tagged Pointers)的方式，标签作为一个版本号，随着标签指针上的每一次CAS运算而增加，并且只增不减。本例中，如果采用标签指针方式，Tread X的t指向Top的时候Top的标签为T1，这个时候t == A并且标签是T1。随后Tread Y执行Pop()，Pop()，Push(),Top至少进过了3次CAS，标签变成了T1+3，于是Top == A并且标签是T1+3，这样在Thread X被调度执行/*Pop5*/这行代码的时候，虽然t == Top == A，但是标签不一样，于是CAS会失败，这样栈就不会被破坏了。

2.3.2 Load-Linked / Store-Conditional -- LL/SC对

通过上面我们知道，ABA问题的本质在于，CAS进行比较的是指针指向的内存地址，虽然在/*Pop1*/行读取Top指向的内存地址，到/*Pop5*/行的CAS，t和Top都是指向A的内存地址，但是A内存里面的内容已经发生过变化了(A的next变成了C)。如果处理器能够感知得到在进行CAS的内存地址的内如发生了变化，让CAS失败的话，那么就能从源头上解决ABA问题。于是PowerPC, MIPS, Alpha, ARM 等处理器架构的开发人员找到了load-linked、store-conditional (LL/SC) 这样的操作对来彻底解决ABA问题，伪代码如下：

```
word LL( word * pAddr ) {
      return *pAddr ;
}

bool SC( word * pAddr, word New ) {
      if ( data in pAddr has not been changed since the LL call) {
           *pAddr = New ;
           return true ;
      }
      else
         return false ;
}
```

LL/SC对以括号运算符的形式运行，Load-linked（LL）运算仅仅返回 pAddr 地址的当前变量值。如果 pAddr 中的内存数据在读取之后没有变化，那么 Store-conditional（SC）操作将会成功，它将LL读取 pAddr 地址的存储新的值，否则，SC将执行失败。这里的pAddr中的内存数据是否变化指的是pAddr地址所在的Cache Line是否发生变化。在实现上，处理器开发者给每个Cahce Line添加额外的比特状态值（status bit）。一旦LL执行读运算，就会关联此比特值。任何的缓存行一旦有写入，此比特值就会被重置；在存储之前，SC操作会检查此比特值是否针对特定的缓存行。如果比特值为1，意味着缓存行没有任何改变，pAddr 地址中的值会变更为新值，SC操作成功。否则本操作就会失败，pAddr 地址中的值不会变更为新值。 CAS通过LL/SC对得以实现，伪代码如下：

```
bool CAS( word * pAddr, word nExpected, word nNew ) {
   if ( LL( pAddr ) == nExpected )
      return SC( pAddr, nNew ) ;
   return false ;
}
```

可以看到通过LL/SC对实现的CAS并不是一个原子性操作，但是它确实执行了原子性的CAS，目标内存单元内容要么不变，要么发生原子性变化。由于通过LL/SC对实现的CAS并不是一个原子性操作，于是，该CAS在执行过程中，可能会被中断，例如：线程X在执行LL行后，OS决定将X调度出去，等OS重新调度恢复X之后，SC将不再响应，这时CAS将返回false，CAS失败的原因不在数据本身(数据没变化)，而是其他外部事件(线程被中断了)。 正是因为如此，C++11标准中添入两个compare_exchange原语-弱的和强的。也因此这两原语分别被命名为compare_exchange_weak和compare_exchange_strong。即使当前的变量值等于预期值，这个弱的版本也可能失败，比如返回false。可见任何weak CAS都能破坏CAS语义，并返回false，而它本应返回true。而Strong CAS会严格遵循CAS语义。 那么，何种情形下使用Weak CAS，何种情形下使用Strong CAS呢？通常执行以下原则：

> 倘若CAS在循环中（这是一种基本的CAS应用模式），循环中不存在成千上万的运算（循环体是轻量级和简单的,本例的无锁堆栈），使用compare_exchange_weak。否则，采用强类型的compare_exchange_strong。

2.3.3 False sharing(伪共享)

现代处理器中，cache是以cache line为单位的，一个cache line长度L为64-128字节，并且cache line呈现长度进一步增加的趋势。主存储和cache数据交换在 L 字节大小的 L 块中进行，即使缓存行中的一个字节发生变化，所有行都被视为无效，必需和主存进行同步。存在这么一个场景，有两个变量share_1和share_2，两个变量内存地址比较相近被加载到同一cahe line中，cpu core1 对变量share_1进行操作，cpu core2对变量share_2进行操作，从cpu core2的角度看，cpu core1对share_1的修改，会使得cpu core2的cahe line中的share_2无效，这种场景叫做False sharing(伪共享)。 由于LL/SC对比较依赖于cache line，当出现False sharing的时候可能会造成比较大的性能损失。加载连接（LL）操作连接缓存行，而存储状态（SC)）操作在写之前，会检查本行中的连接标志是否被重置。如果标志被重置，写就无法执行，SC返回 false。考虑到cache line比较长，在多核cpu中，cpu core1在一个

while循环中变量share_1执行CAS修改，而其他cpu corei在对同一cache line中的变量share_i进行修改。在极端情况下会出现这样的一个livelock(活锁)现象：每次cpu core1在LL(share_1)后，在准备进行SC的时候，其他cpu core修改了同一cache line的其他变量share_i，这样使得cache line发生了改变，SC返回false，于是cpu core1又进入下一个CAS循环，考虑到cache line比较长，cache line的任何变更都会导致SC返回false，这样使得cup core1在一段时间内一直在进行一个CAS循环，cpu core1都跑到100%了，但是实际上没做什么有用功。 为了杜绝这样的False sharing情况，我们应该使得不同的共享变量处于不同cache line中，一般情况下，如果变量的内存地址相差住够远，那么就会处于不同的cache line，于是我们可以采用填充（padding）来隔离不同共享变量，如下：

```
struct Foo {
       int volatile nShared1;
       char   _padding1[64];     // padding for cache line=64 byte
       int volatile nShared2;
       char   _padding2[64];     // padding for cache line=64 byte
};
```

上面，nShared1和nShared2就会处于不同的cache line，cpu core1对nShared1的CAS操作就不会被其他core对nShared2的修改所影响了。 上面提到的cpu core1对share_1的修改会使得cpu core2的share_2变量的cache line失效，造成cpu core2需重新加载同步share_2；同样，cpu core2对share_2变量的修改，也会使得cpu core1所在的cache line实现，造成cpu core1需要重新加载同步share_1。这样cpu core1的一个修改造成cpu core2的一个cache miss，cpu core2的一个修改造成cpu core1的一个cache miss的反复现象就是所谓的Cache ping-pong问题，出现大量Cache ping-pong意味着大量的cache miss，会造成巨大的性能损失。我们同样可以采用填充（padding）来隔离不同共享变量来解决cache ping-pong。

3 局部变量的安全性

通过上面，我们知道实现无锁数据结构在内存使用上存在两个棘手的问题：一是ABA问题，二是内存安全回收问题。这两个问题之间联系比较密切，但是鲜有两全其美的办法，同时解决这两大难题，通常采用各个击破，分别予以解决。 有种从根源上解决这个问题的方法，那就是不产生这两个问题，对于无锁队列来说，我们可以实现一个定长无锁队列，队列在初始化的时候确定好队列的大小n，这样一次性分配好所需的内存(n * sizeof(node))。 定长无锁队列将一块连续的内存分割成n个小内存块block，每个内存块block可以存储一个队列node(当然在队列node过大的情况下，可以用连续的几个内存块来存储一个队列node)。通过head和tail两个指针来进行队列node的入队和出队，从head到tail是已经被使用的内存block(被已入队的队列node占用)，从tail到head之间是空闲内存block。入队的时候，首先原子修改tail指针(tail指针向后移动若干block)，占据需要使用的block，然后往blcok中写入队列node。出队的时候，首先原子修改head指针(head指针向后移动若干block)，占据需要读取的block，然后从block中读取队列node。 定长无锁队列不存在内存的分配和回收问题，同时内存block的位置固定，像一个环形buf一直在循环读写使用，不存在ABA问题。定长无锁队列存在一个队列元素读写完整性问题，由于入队采用的是先入队在写入内容的方式，于是存在队列node内容还没写入完毕就会被出队读取了，读取到一个不完整的node。同样，出队采用先出队，在读取队列node内容，于是也存在内容还没读取的时候，被新的队列node入队的内容给覆盖了。要解决这个问题不复杂，只要给每个队列node加上一个tag标记是否已经写入完毕、是否已经读取完毕即可。 定长无锁队列虽然不存在ABA问题和内存安全回收问题，但是由于其队列是定长的，扩展性比较差。对于ABA问题的解决方案，前面已经介绍了标签指针（Tagged pointers）和LL/SC对两种解决方案。下面着重介绍以下内存安全回收的解决方案。 内存安全回收问题根源上是待回收的内存还被其他线程引用中，此时如果delete该内存，那么引用该内存的线程就会出现使用非法内存的问题，那么我们只能延迟回收该内存，即在安全时刻再delete。目前用于lock free代码的内存回收的经典方法有：Lock Free Reference Counting、Hazard Pointer、Epoch Based Reclamation、Quiescent State Based Reclamation等。

3.1 Epoch Based Reclamation(基于周期的内存回收)

Epoch Based方法采用递增的方式来维护记录当前正在被引用的内存版本ver_i，如果能知道当前被引用的的内存的最小版本ver_min，那么我们就可以安全回收所有内存版本小于ver_min的内存了。通常不会给一个内存对象一个版本，这样版本太多，难以管理，一个折中方案是一个周期内的内存对象都是分配同一个版本ver_p。那么，最少需要几个不同版本呢？一个版本是肯定不可以的，这样就无法区分哪些内存对象是可以安全回收的，哪些是暂时不能回收的。两个版本ver_0、ver_1是否OK呢？假设当前的内存对象都被分配版本号ver_0，在某一个时刻t1，我们决定变更版本号为ver_1，这样新的内存对象就被分配版本ver_1。这样才t1后，在我们再次变更版本号为ver_0前，版本号为ver_0的内存对象就不再增加了。那么，在所有使用版本号为ver_0的内存对象的线程都不再使用这些内存对象后，假设这个时候是t2，这时我们就可以开始回收版本号ver_0的内存对象，回收耗时k*n(n是待回收的内存对象)。很明显，我们再次变更版本号为ver_0的时刻t3是一定要大于等于t2+k*n时刻的，因为，如果t3<t2+k*n，那么在t2+k*n至t3间产生的版本号为的对象就会存在非安全回收的风险。 可以看出采用两个版本是ok的，但是细心的同学会发现，这样的回收粒度有点粗，版本号为ver_1的内存对象在t1至t2+k*n这段时间内一直在增长，整个时间长度依赖于内存对象被引用的时间和ver_0的内存对象被回收的时间，这样可能会引起滚雪球效应，越往后面回收时间会越长。 通过上面的分析，我们知道，如果想版本号变更的时间点不依赖ver_0的内存对象被回收的时间，我们需要增加一个版本号ver_2，那么在t2时刻，我们就可以切换版本号为ver_2，同时可以启动回收ver_0的内存对象。 通过上面的分析，Epoch Based算法维护了一个全局的epoch(取值为0、1、2)和三个全局的retire_list（每个全局的epoch对应一个retire list, retire list 存放逻辑删除后待回收的节点指针）。除此之外我们为每个线程维护一个局部的thread_active flag(这个用来标识thread时候已经不再引用该epoch值的内存对象)和thread_epoch(取值自然也为0、1、2)。算法如下：

```
#define N_THREADS 4 //假设一共4个线程
const EPOCH_COUNT = 3 ;
bool active[N_THREADS] = {false};
int epoches[N_THREADS] = {0};
int global_epoch = 0;
vector<int*> retire_list[3];
void read(int thread_id)
{
  active[thread_id] = true;
  epoches[thread_id] = global_epoch;
  //进入临界区了。可以安全的读取
  //......
  //读取完毕，离开临界区
  active[thread_id] = false;
}
void logical_deletion(int thread_id)
{
  active[thread_id] = true;
  epoches[thread_id] = global_epoch;
  //进入临界区了，这里，我们可以安全的读取
  //好了，假如说我们现在要删除它了。先逻辑删除。
  //而被逻辑删除的tmp指向的节点还不能马上被回收，因此把它加入到对应的retire list
  retire_list[global_epoch].push_back(tmp);
  //离开临界区
  active[thread_id] = false;
  //看看能不能物理删除
  try_gc();
}
bool try_gc()
{
  int &e = global_epoch;
  for (int i = 0; i < N_THREADS; i++) {
```

```
    if (active[i] && epoches[i] != e) {
        //还有部分线程没有更新到最新的全局的epoch值
        //这时候可以回收(e + 1) % EPOCH_COUNT对应的retire list。
        free((e + 1) % EPOCH_COUNT);//不是free(e)，也不是free(e-1)。
        return false;
    }
  }
  //更新global epoch
  e = (e + 1) % EPOCH_COUNT;
  //更新之后，那些active线程中，部分线程的epoch值可能还是e - 1（模EPOCH_COUNT）
  //那些inactive的线程，之后将读到最新的值，也就是e。
  //不管如何，(e + 1) % EPOCH_COUNT对应的retire list的那些内存，不会有人再访问到了，可以回收它们
了
  //因此epoch的取值需要有三种，仅仅两种是不够的。
  free((e + 1) % EPOCH_COUNT);//不是free(e)，也不是free(e-1)。
}
bool free(int epoch)
{
  for each pointer in retire_list[epoch]
    if (pointer is not NULL)
      delete pointer;
}
```

Epoch Based Reclamation算法规则比较简单明了，该算法规则有个重要的缺陷是，它依赖于所有使用ver_0的内存对象的线程都进入到下个周期ver_1后，ver_0的内存对象才能被回收。只要有一个线程未能进入到下个周期ver_1，那么那些大多数已经没有引用的ver_0内存对象就不能被删除回收。这个在线程存在不同的优先级时候，优先级低的线程会导致优先级高的线程延迟待删除元素增长变得不可控，一旦某个线程一直无法进入下一个周期，会导致无限的内存消耗。

3.2 险象指针（Hazard pointer）

Hazard Pointer由Maged M. Michael在论文"Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects"中提出，基本思路是将可能要被访问到的共享对象指针（成为hazard pointer）先保存到线程局部，然后再访问，访问完成后从线程局部移除。而要释放一个共享对象时，则要先遍历查询所有线程的局部信息，如果尚有线程局部保存有这个共享对象的指针，说明这个线程有可能将要访问这个对象，因此不能释放，只有所有线程的局部信息中都没有保存这个共享对象的指针情况下，才能将其释放。 我们知道Hazard Pointer封装了原始指针，那么Hazard Pointer的内存和生命周期本身如何管理呢？以下是常见的策略：

1，Hazard Pointer本身的内存只分配，不释放。在stack、queue等数据结构里，需要的Hazard Pointer数量一般为1或者2，所以不释放问题不大。对于skip list这种数据结构又有遍历需求的，那么Hazard Pointer可能就不是非常适用了，可以考虑使用Epoch Based Reclamation技术。据我所知，这也是memsql使用的内存回收策略。

2，每个线程拥有、管理自己的retire list和hazard pointer list ，而不是所有线程共享一个retire list，这样可以避免维护retire list和hazard pointer list的开销，否则我们可能又得想尽脑汁去设计另外一套lock free的策略来管理这些list，先有鸡先有蛋，无穷无尽。所谓retire list就是指逻辑删除后待物理回收的指针列表。

3，每个线程负责回收自己的retire list中记录维护的内存。这样，retire list是一个线程局部的数据结构，自己写，自己读，吃自己的狗粮。

4，只有当retire list的大小（数量）达到一定的阈值时，才进行GC。这样，可以把GC的开销进行分摊，同时，应该尽可能使用Jemalloc或者TCmalloc这些高效的、带线程局部缓存的内存分配器。

3.3 Hazard Version

HazardPointer的实现简单，但是其有个不足：需要为每个共享变量维护一个线程的hazard pointer，这样使用者需要仔细分析算法以尽量减少同时存在的hazard pointer，Hazard Pointer机制也与具体数据结构的实现比较紧的耦合在一起，对于skip list这样的有遍历需求的数据结构同时存在的hazard pointer很容易膨胀比较多，内存使用是个问题。 因此在Hazard Pointer基础上发展出了被称为Hazard Version技术，它提供类似lock一样的acquire/release接口，支持无限制个数共享对象的管理。 与Hazard Pointer的实现不同：首先全局要维护一个int64_t类型的GlobalVersion；要访问共享对象前，先将当时的GlobalVersion保存到线程局部，称为hazard version；而每次要释放共享对象的时候先将当前GlobalVersion保存在共享对象，然后将GlobalVersion原子的加1，然后遍历所有线程的局部信息，找到最小的version称为reclaim version，判断如果待释放的对象中保存的version小于reclaim version则可以释放。hazard version就类似于给每个内存对象分配一个单调递增的version的Epoch Based方法，是更细粒度的内存回收。

4 补偿策略(functor bkoff)

补偿策略通常作为避免大量CAS竞争的一种退避策略，在大并发修改同一变量的情况下，能有效缓解CPU压力。考虑这么一个场景，N个线程同时对一个无锁队列进行入队操作，于是同时进行的N个CAS操作，最终只有一个线程返回成功。于是，失败的N-1个线程在下一个循环中继续重试同时进行N-1个CAS操作，最终还是只有一个线程返回成功。这样一直下去，大量的CPU被消耗在无用的CAS操作上，我们知道CAS操作是一个很重的一个操作，服务器性能会急剧下降。这好比来自四面八方的车辆汇聚到一个出口，大家都比较自私的想要最快通过的话，那么这个路口会被堵的水泄不通，理想的情况是将车辆流水线化，这样大家都能较快通过出口。然而实际情况是比较难流水线化的，于是，我们采用礼让的方式，在尝试通过路口发现堵塞的时候，就delay一小会在尝试通过。一个简单的实现如下：

```
  bkoof()
  {
    static const int64_t INIT_LOOP = 1000000;
    static const int64_t MAX_LOOP = 8000000;
    static __thread int64_t delay = 0;
    if (delay <= 0) {
      delay = INIT_LOOP;
    }
    for (int64_t i = 0; i < delay; i++) {
      CPU_RELAX();
    }
    int64_t new_delay = delay << 1LL;
    if (new_delay <= 0 || new_delay >= MAX_LOOP) {
      new_delay = INIT_LOOP;
    }
    delay = new_delay;
  }
}
```

5 helping方法

Helping方法是一种广泛存在于无锁算法中的方法，特别是在一个线程帮助其它线程去执行任务场景中。本例中的无锁队列实现，会出现m_pTail的暂时错位不正确，主要的原因是m_pTail的修改(如下)并不能保证一定成功。

```
m_pTail.compare_exchange_strong( t, pNew, memory_model::memory_order_acq_rel,
        memory_model::memory_order_relaxed );
```

这里为什么不重试让m_pTail指向正确的位置呢？这里主要是实现策略和成本开销的问题，考虑这么一个场景：

```
1. 当前时刻队列有3个节点(A-->B-->C)，队列状态：m_pHead->m_pNext == A，m_pTail == C
2. 这时线程1执行入队enqueue(D)，线程2执行入队enqueue(E)。
3. 线程1执行enqueue(D)进行到最后一步，这时队列状态：(A-->B-->C-->D)，m_pHead->m_pNext == A，
m_pTail == C
4. 线程2执行enqueue(E)，这时它发现m_pTail->m_pNext != NULL，m_pTail位置不正确了。
```

这个时候，线程2有两个选择：(1) 不断重试等待线程1将m_pTail设置正确后，自己在进行下面的操作步骤。(2) 顺路帮线程1一把，自己将m_pTail调整到正确位置，然后在进行下面的操作步骤。如果采用(1)，线程2可能会进入一个较漫长的等待来等线程1完成m_pTail 的设置。采用(2)则是一个双赢的局面，线程2不在需要等待和依赖线程1，线程1也不再需要在m_pTail设置失败的时候进行重试了。

6 内存屏障（Memory Barriers）

6.1 What Memory Barriers ?

内存屏障，也称内存栅栏，内存栅障，屏障指令等，是一类同步屏障指令，是CPU或编译器在对内存随机访问的操作中的一个同步点，使得此点之前的所有读写操作都执行后才可以开始执行此点之后的操作。大多数现代计算机为了提高性能而采取乱序执行，这使得内存屏障成为必须。语义上，内存屏障之前的所有写操作都要写入内存；内存屏障之后的读操作都可以获得同步屏障之前的写操作的结果。因此，对于敏感的程序块，写操作之后、读操作之前可以插入内存屏障。 通常情况下，我们希望我们所编写的程序代码能"所见即所得"，即程序逻辑满足程序的顺序性(满足program order)，然而，很遗憾，我们的程序逻辑("所见")和最后的执行结果("所得")隔着：

```
1. 编译器
2. CPU取指执行
```

1. 编译器将符合人类思考的逻辑（程序代码）翻译成了符合CPU运算规则的汇编指令，编译器了解底层CPU的思维模式，因此，它可以在将程序翻译成汇编的时候进行优化（例如内存访问指令的重新排序），让产出的汇编指令在CPU上运行的时候更快。然而，这种优化产出的结果未必符合程序员原始的逻辑，因此，作为程序员，必须有能力了解编译器的行为，并在通过内嵌在程序代码中的memory barrier来指导编译器的优化行为（这种memory barrier又叫做优化屏障，Optimization barrier），让编译器产出即高效，又逻辑正确的代码。
2. CPU的核心思想就是取指执行，对于in-order的单核CPU，并且没有cache，汇编指令的取指和执行是严格按照顺序进行的，也就是说，汇编指令就是所见即所得的，汇编指令的逻辑严格的被CPU执行。然而，随着计算机系统越来越复杂（多核、cache、superscalar、out-of-order），使用汇编指令这样贴近处理器的语言也无法保证其被CPU执行的结果的一致性，从而需要程序员告知CPU如何保证逻辑正确。

综上所述，memory barrier是一种保证内存访问顺序的一种方法，让系统中的HW block（各个cpu、DMA controler、device等）对内存有一致性的视角。 通过上面介绍，我们知道我们所编写的代码会根据一定规则在与内存的交互过程中发生乱序。内存执行顺序的变化在编译器(编译期间)和cpu(运行期间)中都会发生，其目的都是为了让代码运行的更快。就算是为了性能而乱序，但是乱序总有个度吧(总不能将指针的初始化的代码乱序在使用指针的代码之后吧，这样谁还敢写代码)。编译器开发者和cpu厂商都遵守着内存乱序的基本原则，简单归纳如下：

```
不能改变单线程程序的执行行为 -- 但线程程序总是满足Program Order(所见即所得)
```

在此原则指导下，写单线程代码的程序员不需要关心内存乱序的问题。在多线程编程中，由于使用互斥量，信号量和事件都在设计的时候都阻止了它们调用点中的内存乱序(已经隐式包含各种memery barrier)，内存乱序的问题同样不需要考虑了。只有当使用无锁(lock-free)技术时–内存在线程间共享而没有任何的互斥量，内存乱序的效果才会显露无疑，这样我们才需要考虑在合适的地方加入合适的memery barrier。

6.1.1 编译期乱序

考虑下面一段代码：

```
int Value = 0;
int IsPublished = 0;

void sendValue(int x)
{
    Value = x;
    IsPublished = 1;
}

int tryRecvValue()
{
    if (IsPublished)
    {
        return Value;
    }
    return -1;  // or some other value to mean not yet received
}
```

在出现编译期乱序的时候，sendValue可能变成如下：

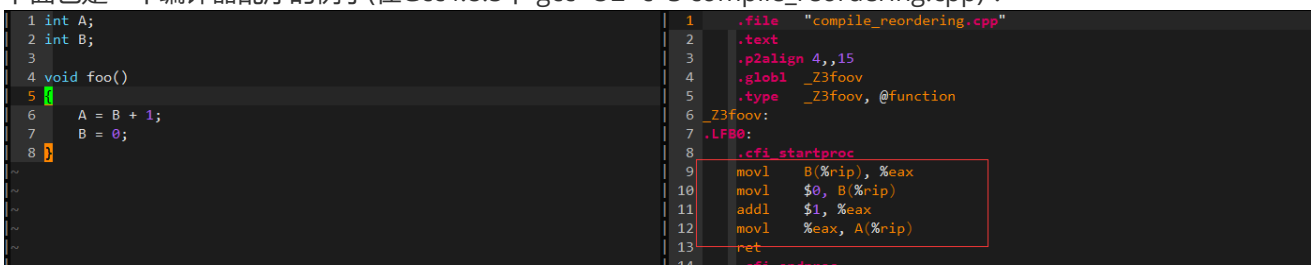
```
void sendValue(int x)
{
        IsPublished = 1;
    Value = x;
}
```

对于但线程而言，这样的乱序是不会有影响的，因为sendValue(10)调用后，IsPublished == 1; Value == 10；这时调用tryRecvValue()就会得到10和乱序前是一样的结果。但是对于多线程，线程1调用sendValue(10)， 线程2调用tryRecvValue()，当线程1执行完IsPublished = 1;的时候，线程2调用tryRecvValue()就会得到Value的初始默认值0，这和程序原本逻辑违背，于是我们必须加上编译器的barrier来防止编译器的乱序优化：

```
#define COMPILER_BARRIER() asm volatile("" ::: "memory")

int Value;
int IsPublished = 0;

void sendValue(int x)
{
    Value = x;
    COMPILER_BARRIER();          // prevent reordering of stores
    IsPublished = 1;
}
```

```
int tryRecvValue()
{
    if (IsPublished)
    {
        COMPILER_BARRIER();      // prevent reordering of loads
        return Value;
    }
    return -1;  // or some other value to mean not yet received
}
```

下面也是一个编译器乱序的例子(在Gcc4.8.5下 gcc -O2 -c -S compile_reordering.cpp)：



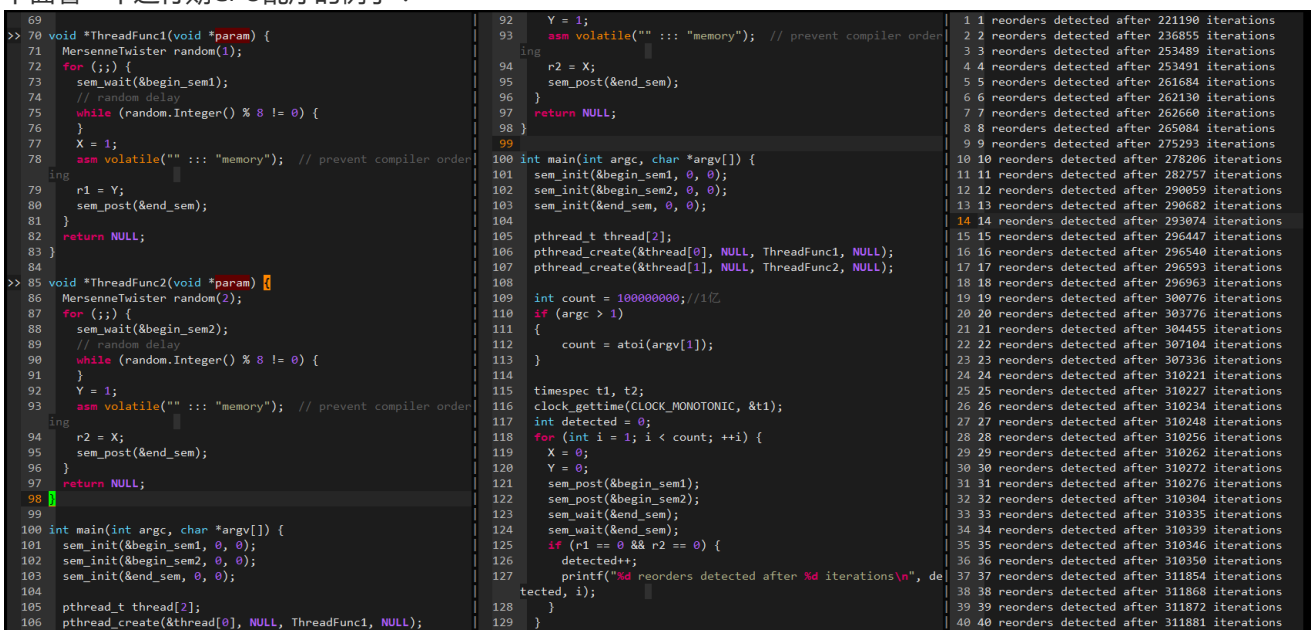
```
int A;
int B;

void foo()
{
    A = B + 1;
    B = 0;
}
```

```
    .file   "compile_reordering.cpp"
    .text
    .p2align 4,,15
    .globl  _Z3foov
    .type   _Z3foov, @function
_Z3foov:
.LFB0:
    .cfi_startproc
    movl    B(%rip), %eax
    movl    $0, B(%rip)
    addl    $1, %eax
    movl    %eax, A(%rip)
    ret
```

可以看出，在开启-o2编译器优化选项时，内存会发生乱序，在写变量A之前会先写变量B。

6.1.2 运行期乱序

下面看一个运行期CPU乱序的例子：



```
void *ThreadFunc1(void *param) {
  MersenneTwister random(1);
  for (;;) {
    sem_wait(&begin_sem1);
    // random delay
    while (random.Integer() % 8 != 0) {
    }
    X = 1;
    asm volatile("" ::: "memory");  // prevent compiler ordering
    r1 = Y;
    sem_post(&end_sem);
  }
  return NULL;
}

void *ThreadFunc2(void *param) {
  MersenneTwister random(2);
  for (;;) {
    sem_wait(&begin_sem2);
    // random delay
    while (random.Integer() % 8 != 0) {
    }
    Y = 1;
    asm volatile("" ::: "memory");  // prevent compiler ordering
    r2 = X;
    sem_post(&end_sem);
  }
  return NULL;
}

int main(int argc, char *argv[]) {
  sem_init(&begin_sem1, 0, 0);
  sem_init(&begin_sem2, 0, 0);
  sem_init(&end_sem, 0, 0);

  pthread_t thread[2];
  pthread_create(&thread[0], NULL, ThreadFunc1, NULL);
  pthread_create(&thread[1], NULL, ThreadFunc2, NULL);

  int count = 100000000;//1亿
  if (argc > 1)
  {
      count = atoi(argv[1]);
  }

  timespec t1, t2;
  clock_gettime(CLOCK_MONOTONIC, &t1);
  int detected = 0;
  for (int i = 1; i < count; ++i) {
    X = 0;
    Y = 0;
    sem_post(&begin_sem1);
    sem_post(&begin_sem2);
    sem_wait(&end_sem);
    sem_wait(&end_sem);
    if (r1 == 0 && r2 == 0) {
      detected++;
      printf("%d reorders detected after %d iterations\n", detected, i);
    }
  }
```

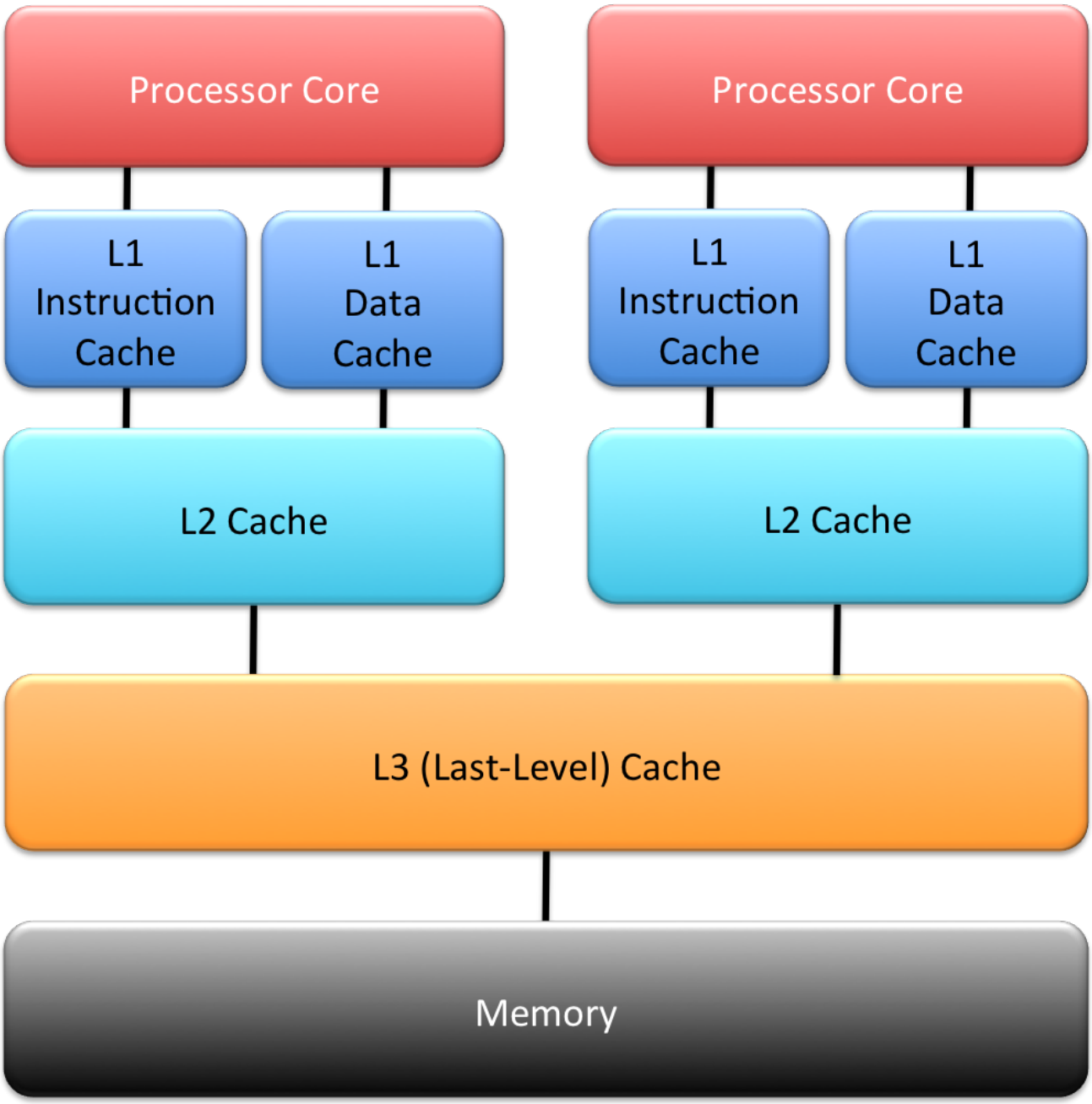
```
1 reorders detected after 221190 iterations
2 reorders detected after 236855 iterations
3 reorders detected after 253489 iterations
4 reorders detected after 253491 iterations
5 reorders detected after 261684 iterations
6 reorders detected after 262130 iterations
7 reorders detected after 262660 iterations
8 reorders detected after 265084 iterations
9 reorders detected after 275293 iterations
10 reorders detected after 278206 iterations
11 reorders detected after 282757 iterations
12 reorders detected after 290059 iterations
13 reorders detected after 290682 iterations
14 reorders detected after 293074 iterations
15 reorders detected after 296447 iterations
16 reorders detected after 296540 iterations
17 reorders detected after 296593 iterations
18 reorders detected after 296963 iterations
19 reorders detected after 300776 iterations
20 reorders detected after 303776 iterations
21 reorders detected after 304455 iterations
22 reorders detected after 307104 iterations
23 reorders detected after 307336 iterations
24 reorders detected after 310221 iterations
25 reorders detected after 310227 iterations
26 reorders detected after 310234 iterations
27 reorders detected after 310248 iterations
28 reorders detected after 310256 iterations
29 reorders detected after 310262 iterations
30 reorders detected after 310272 iterations
31 reorders detected after 310276 iterations
32 reorders detected after 310304 iterations
33 reorders detected after 310335 iterations
34 reorders detected after 310339 iterations
35 reorders detected after 310346 iterations
36 reorders detected after 310350 iterations
37 reorders detected after 311854 iterations
38 reorders detected after 311868 iterations
39 reorders detected after 311872 iterations
40 reorders detected after 311881 iterations
```

可以看出在22W多次迭代后检测到一次乱序，乱序间隔在摇摆不定。

6.2 Why Memory Barriers ?

6.2.1 现代处理器cache架构

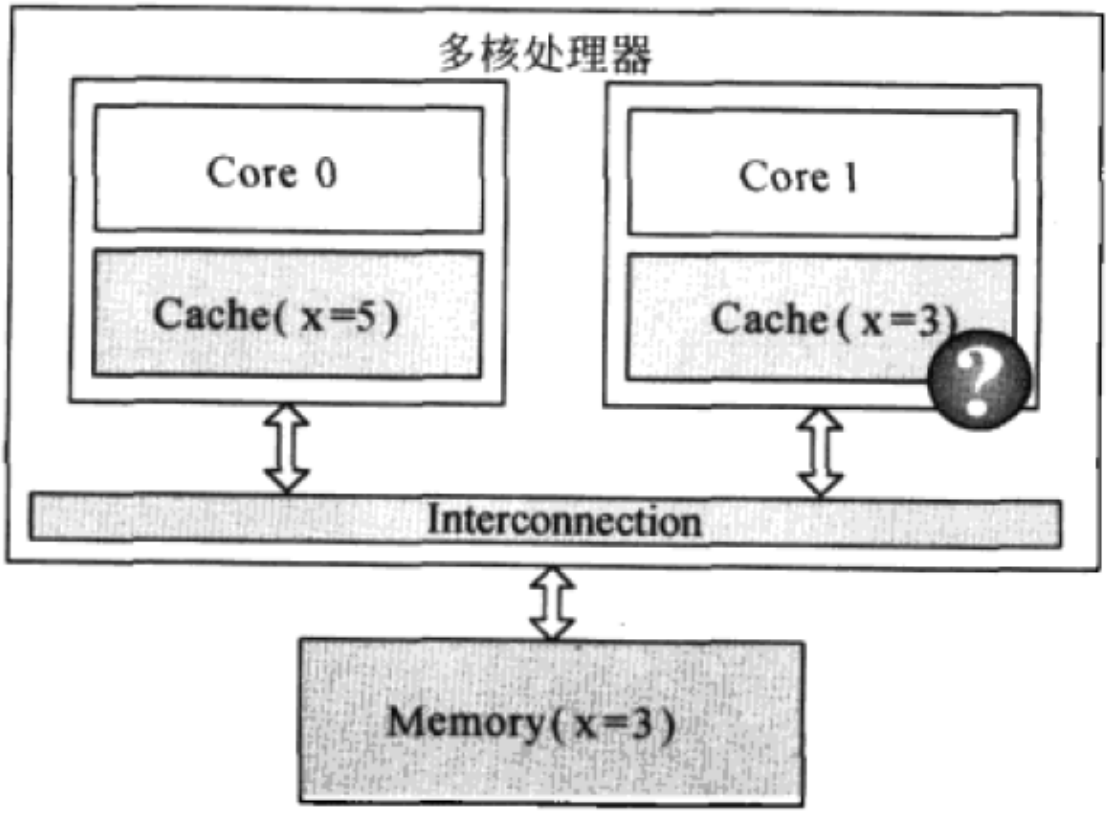
通过上面，我们知道存在两种类型的Memory Barriers：编译器的Memory Barrier、处理器的Memory Barrier。对于编译器的Memory Barrier比较好理解，就是防止编译器为了优化而将代码执行调整乱序。而处理器的Memory Barrier是防止CPU怎样的乱序呢？CPU的内存乱序是怎么来的？ 乱序会有问题本质上是读到了老的数据，或者是一部分读到新的一部分读到老的数据，例如：上面的例子中，已经读到了IsPublished的新值，却还是读到了Value老的值，从而引起问题。这种数据不一致怎么来的呢？相信这个时候大家脑海里已经浮现出一个词了：Cache。 首先我们来看看现代处理器基本的cache架构



现代处理器为了弥补内存速度低下的缺陷，引入Cache来提高处理器访问程序和数据的速度，Cache作为连接内核和内存的桥梁，极大提升了程序的运行速度。为什么处理器内部加一个速度快，容量小的cache就能提速呢？这里基于程序的两个特性：时间的局部性(Temporal locality)和空间的局部性(Spatial)

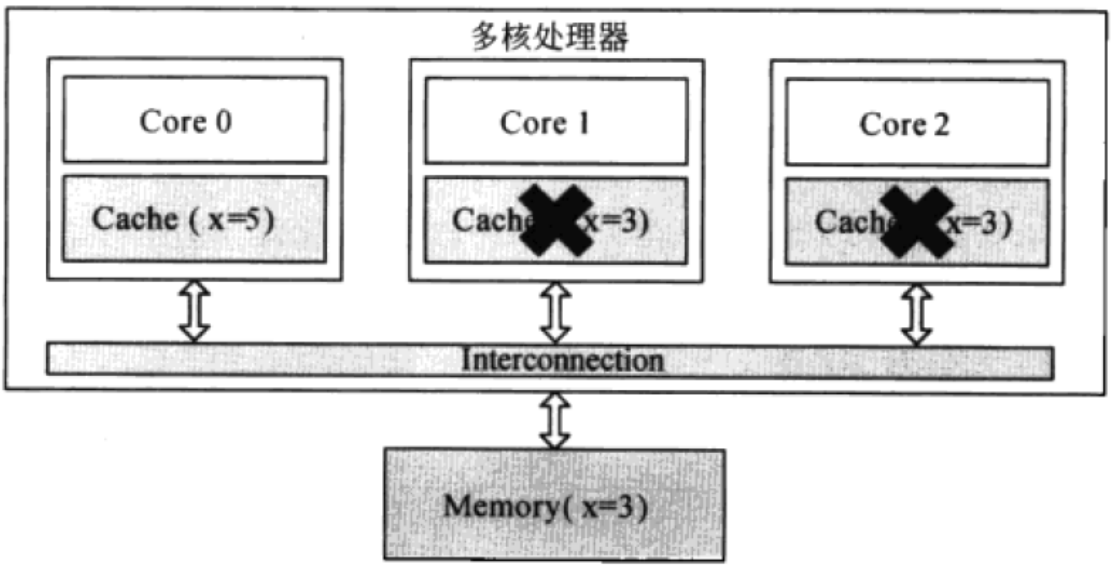
> [1] 时间的局部性(Temporal locality)：如果某个数据被访问了，那么不久的将来它很有可能被再次访问到。典型的例子就是循环，循环的代码被处理器重复执行，将循环代码放在Cache中，那么只是在第一次的时候需要耗时较长去内存取，以后这些代码都能被内核从cache中快速访问到。
> [2] 空间的局部性(Spatial)：如果某个数据被访问了，那么它相临的数据很可能很快被访问到。典型的例子就是数组，数组中的元素常常安装顺序依次被程序访问。

现代处理器一般是多个核心Core，每个Core在并发执行不同的代码和访问不同的数据，为了隔离影响，每个core都会有自己私有的cache(如图的L1和L2)，同时也在容量和存储速度上进行一个平衡(容量也大存储速度越慢，速度：L1>L2>L3， 容量：L3>L2>L1)，于是就出现图中的层次化管理。Cache的层次化必然带来一个cache一致性的问题：



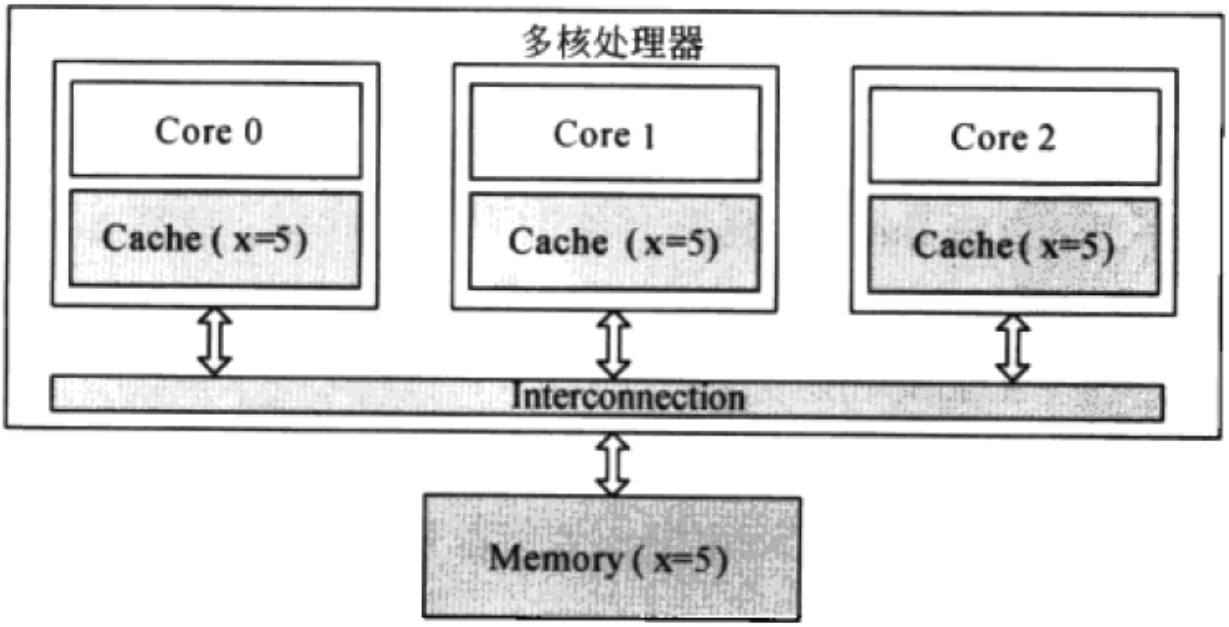
如图的例子，变量X(初始值是3)被cache在Core 0和Core 1的私有cache中，这时core 0将X修改成5，如果core 1不知道X已经被修改了，继续使用cache中的旧值，那么可能会导致严重的问题，这就是Cache的不一致导致的。为了保证Cache的一致性，处理器提供两个保证Cache一致性的底层操作：Write Invalidate和Write Update。

Write Invalidate(置无效)：当一个CPU Core修改了一份数据X，那么它需要通知其他core将他们的cache中的X设置为无效(invalid)(如果cache中有的话)，如下图



Write invalidate 示例

Write Update(写更新)：当一个CPU Core修改了一份数据X，那么它需要通知其他core将他们的cache中的X更新到最新值(如果cache中有的话)，如下图



Write update 示例

Write Invalidate和Write Update的比较：Write Invalidate是一种更为简单和轻量的实现方式，它不需要立刻将数据更新到存储中(这时一个耗时过程)，如果后续Core 0继续需要修改X而Core 1和Core 2又不再使用数据X了，那么这个Update过程就有点做了无用功，而采用write invalidate就更为轻量和有效。不过，由于valid标志是对应一个Cache line的，将valid标志设置为invalid后，这个cache line的其他本来有效的数据也不能被使用了，如果处理不好容易出现前面提到的False sharing(伪共享)和Cache pingpong问题。

6.2.2 cache一致性协议MESI

由于Write Invalidate比较简单和轻量，大多数现代处理器都采用Write Invalidate策略，基于Write Invalidate处理器会有一套完整的协议来保证Cache的一致性，比较经典的当属MESI协议，奔腾处理器采用它，很多其他处理器都是采用它的一个小变种。 每个核的Cache中的每个Cache Line都有2个标志位：dirty标志和valid标志位，两个标志位分别描述了Cache和Memory间的数据关系(数据是否有效，数据是否被修改)，而在多核处理器中，多个核会共享一些数据，MESI协议就包含了描述共享的状态。 这样在MESI协议中，每个Cache line都有4个状态，可用2个bit来表示(也就是，每个cache line除了物理地址和具体的数据之外，还有一个2-bit的tag来标识该cacheline的4种不同的状态)：

[1] M(Modified): cache line数据有效，但是数据被修改过了，本Cache中的数据是最新的，内存的数据是老的，需要在适当时候将Cache数据写回内存。因此，处于modified状态的cacheline也可以说是被该CPU独占。而又因为只有该CPU的cache保存了最新的数据（最终的memory中都没有更新），所以，该cache需要对该数据负责到底。例如根据请求，该cache将数据及其控制权传递到其他cache中，或者cache需要负责将数据写回到memory中，而这些操作都需要在reuse该cache line之前完成。

[2] E(Exclusive)：cache line数据有效，并且cache和memory中的数据是一致的，同时数据只在本cache中有效。exclusive状态和modified状态非常类似，唯一的区别是对应CPU还没有修改cacheline中的数据，也正因为还没有修改数据，因此memory中对应的data也是最新的。在exclusive状态下，cpu也可以不通知其他CPU cache而直接对cacheline进行操作，因此，exclusive状态也可以被认为是被该CPU独占。由于memory中的数据和cacheline中的数据都是最新的，因此，cpu不需对exclusive状态的cacheline执行写回的操作或者将数据以及归属权转交其他cpu cache，而直接reuse该cacheline（将cacheine中的数据丢弃，用作他用）。

[3] S(Shared)：cache line的数据有效，并且cache和memory中的数据是一致的，同时该数据在多个cpu cache中也是有效的。和exclusive状态类似，处于share状态的cacheline对应的memory中的数据也是最新的，因此，cpu也可以直接丢弃cacheline中的数据而不必将其转交给其他CPU cache或者写回到memory中。

[4] I(Invalid)：本cache line的数据已经是无效的。处于invalid状态的cacheline是空的，没有数据。当新的数据要进入cache的时候，优选状态是invalid的cacheline，之所以如此是因为如果选中其他状态的cacheline，则说明需要替换cacheline数据，而未来如果再次访问这个被替换掉的cacheline数据的时候将遇到开销非常大的cache miss。

在MESI协议中，每个CPU都会监听总线(bus)上的其他CPU对每个Cache line的所有操作，因此该协议也称为监听(snoop)协议，监听协议比较简单，被多少处理器使用，不过监听协议的沟通成本比较高。有另外一种协议叫目录协议，他采用集中管理的方式，将cache共享的信息集中在一起，类似一个目录，只有共享的Cache line才会交互数据，这种协议沟通成本就大大减少了。在基于snoop的处理器中，所有的CPU都是在一个共享的总线上，多个CPU之间需要相互通信以保证Cache line在M、E、S、I四个状态间正确的转换，从而保证数据的一致性。通常情况下，CPU需要以下几个通信message即可：

[1] Read消息：read message用来获取指定物理地址上的cacheline数据。

[2] Read Response消息：该消息携带了read message请求的数据。read response可能来自memory，也可能来自其他的cache。例如：如果一个cache有read message请求的数据并且该cacheline的状态是modified，那么该cache必须以read response回应这个read message，因为该cache中保存了最新的数据。

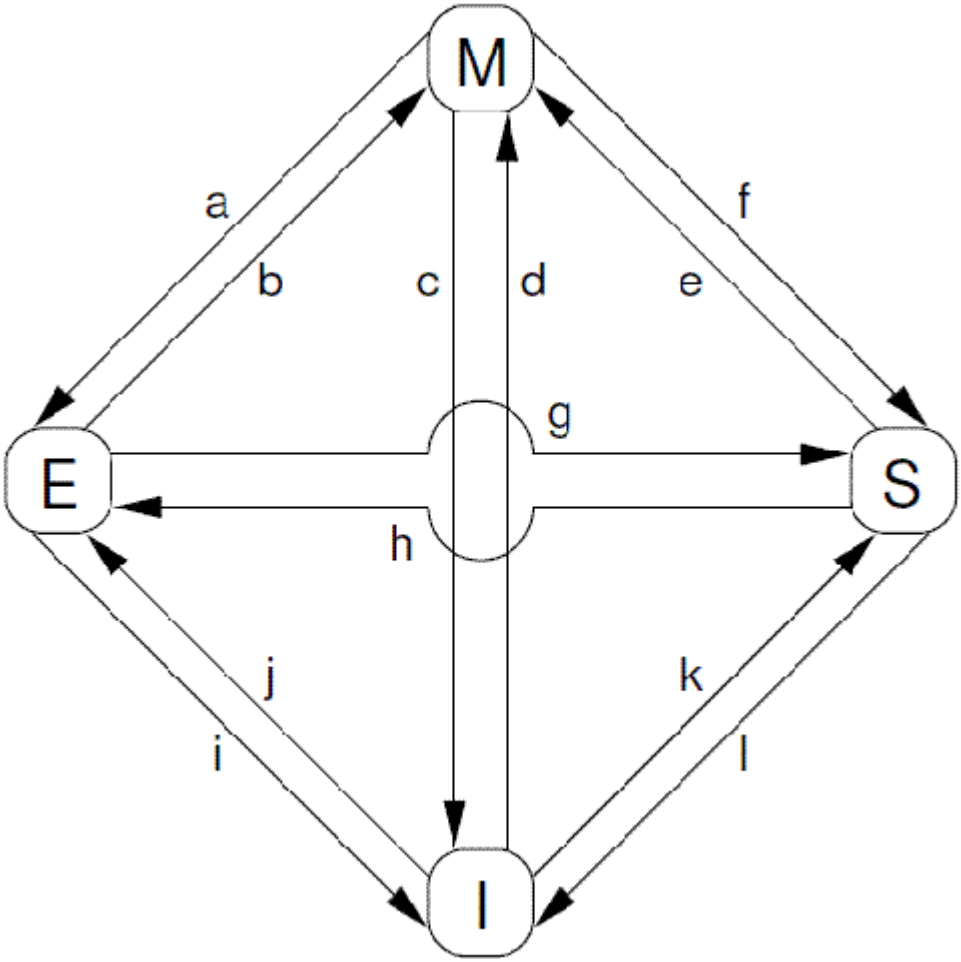
[3] Invalidate消息：该命令用来将其他cpu cache中的数据设定为无效。该命令携带物理地址的参数，其他CPU cache在收到该命令后，必须进行匹配，发现自己的cacheline中有该物理地址的数据，那么就将其移除并用Invalidate Acknowledge回应。

[4] Invalidate Acknowledge消息： 收到invalidate message的cpu cache，在移除了其cache line中的特定数据之后，必须发送invalidate acknowledge消息。

[5] Read Invalidate消息： 该message中也包括了物理地址这个参数，以便说明其想要读取哪一个cacheline数据。此外，该message还同时有invalidate message的功效，即其他的cache在收到该命令后，移除自己cacheline中的数据。因此，Read Invalidate message实际上就是read + invalidate。发送Read Invalidate之后，cache期望收到一个read response以及多个invalidate acknowledge。

[6] Writeback消息： 该message包括两个参数，一个是地址，另外一个是写回的数据。该消息用在modified状态的cacheline被驱逐出境（给其他数据腾出地方）的时候发出，该命名用来将最新的数据写回到memory（或者其他的CPU cache中）。

根据protocol message的发送和接收情况，cacheline会在“modified”, “exclusive”, “shared”, 和 “invalid”这四个状态之间迁移，具体如下图所示：



对上图中的状态迁移解释如下：

[a] Transition (a)：cache可以通过writeback transaction将一个cacheline的数据写回到memory中（或者下一级cache中），这时候，该cacheline的状态从Modified迁移到Exclusive状态。对于cpu而言，cacheline中的数据仍然是最新的，而且是该cpu独占的，因此可以不通知其他cpu cache而直接修改之。

[b] Transition (b)：在Exclusive状态下，cpu可以直接将数据写入cacheline，不需要其他操作。相应的，该cacheline状态从Exclusive状态迁移到Modified状态。这个状态迁移过程不涉及bus上的Transaction（即无需MESI Protocol Messages的交互）。

[c] Transition (c)：CPU 在总线上收到一个read invalidate的请求，同时，该请求是针对一个处于modified状态的cacheline，在这种情况下，CPU必须该cacheline状态设置为无效，并且用read response”和“invalidate acknowledge来回应收到的read invalidate的请求，完成整个bus transaction。一旦完成这个transaction，数据被送往其他cpu cache中，本地的copy已经不存在了。

[d] Transition (d)：CPU需要执行一个原子的readmodify-write操作，并且其cache中没有缓存数据，这时候，CPU就会在总线上发送一个read invalidate用来请求数据，同时想独自霸占对该数据的所有权。该CPU的cache可以通过read response获取数据并加载cacheline，同时，为了确保其独占的权利，必须收集所有其他cpu发来的invalidate acknowledge之后（其他cpu没有local copy），完成整个bus transaction。

[e] Transition (e)：CPU需要执行一个原子的readmodify-write操作，并且其local cache中有read only的缓存数据（cacheline处于shared状态），这时候，CPU就会在总线上发送一个invalidate请求其他cpu清空自己的local copy，以便完成其独自霸占对该数据的所有权的梦想。同样的，该cpu必须收集所有其他cpu发来的invalidate acknowledge之后，才算完成整个bus transaction。

[f] Transition (f)：在本cpu独自享受独占数据的时候，其他的cpu发起read请求，希望获取数据，这时候，本cpu必须以其local cacheline的数据回应，并以read response回应之前总线上的read请求。这时候，本cpu失去了独占权，该cacheline状态从Modified状态变成shared状态（有可能也会进行写回的动作）。

[g] Transition (g)：这个迁移和f类似，只不过开始cacheline的状态是exclusive，cacheline和memory的数据都是最新的，不存在写回的问题。总线上的操作也是在收到read请求之后，以read response回应。

[h] Transition (h)：如果cpu认为自己很快就会启动对处于shared状态的cacheline进行write操作，因此想提前先霸占上该数据。因此，该cpu会发送invalidate敦促其他cpu清空自己的local copy，当收到全部其他cpu的invalidate acknowledge之后，transaction完成，本cpu上对应的cacheline从shared状态切换exclusive状态。还有另外一种方法也可以完成这个状态切换：当所有其他的cpu对其local copy的cacheline进行写回操作，同时将cacheline中的数据设为无效（主要是为了为新的数据腾些地方），这时候，本cpu坐享其成，直接获得了对该数据的独占权。

[i] Transition (i)：其他的CPU进行一个原子的read-modify-write操作，但是，数据在本cpu的cacheline中，因此，其他的那个CPU会发送read invalidate，请求对该数据以及独占权。本cpu回送read response”和“invalidate acknowledge”，一方面把数据转移到其他cpu的cache中，另外一方面，清空自己的cacheline。

[j] Transition (j)：cpu想要进行write的操作但是数据不在local cache中，因此，该cpu首先发送了read invalidate启动了一次总线transaction。在收到read response回应拿到数据，并且收集所有其他cpu发来的invalidate acknowledge之后（确保其他cpu没有local copy），完成整个bus transaction。当write操作完成之后，该cacheline的状态会从Exclusive状态迁移到Modified状态。

[k] Transition (k)：本CPU执行读操作，发现local cache没有数据，因此通过read发起一次bus transaction，来自其他的cpu local cache或者memory会通过read response回应，从而将该cacheline从Invalid状态迁移到shared状态。

[l] Transition (l)：当cacheline处于shared状态的时候，说明在多个cpu的local cache中存在副本，因此，这些cacheline中的数据都是read only的，一旦其中一个cpu想要执行数据写入的动作，必须先通过invalidate获取该数据的独占权，而其他的CPU会以invalidate acknowledge回应，清空数据并将其cacheline从shared状态修改成invalid状态。

下面通过几个例子，说明一下MESI协议是怎么工作的。CPU执行序列如下：

Sequence #	CPU #	Operation	CPU Cache				Memory	
			0	1	2	3	0	8
0		Initial State	-/I	-/I	-/I	-/I	V	V
1	0	Load	0/S	-/I	-/I	-/I	V	V
2	3	Load	0/S	-/I	-/I	0/S	V	V
3	0	Invalidation	8/S	-/I	-/I	0/S	V	V
4	2	RMW	8/S	-/I	0/E	-/I	V	V
5	2	Store	8/S	-/I	0/M	-/I	I	V
6	1	Atomic Inc	8/S	0/M	-/I	-/I	I	V
7	1	Writeback	8/S	8/S	-/I	-/I	V	V

第一列是操作序列号，第二列是执行操作的CPU，第三列是具体执行哪一种操作，第四列描述了各个cpu local cache中的cacheline的状态（用meory address/状态表示），最后一列描述了内存在0地址和8地址的数据内容的状态：V表示是最新的，和cpu cache一致，I表示不是最新的内容，最新的内容保存在cpu cache中。

[1] sequence 0：初始状态下，内存地址0和8保存了最新的数据，而4个CPU的cache line都是invalid(没cache任何数据或cache的数据都是过期无效的)。

[2] sequence 1：CPU 0对内存地址0执行load操作，这样内存地址0的数据被加载到CPU 0的cache line中，CPU 0的cache line从Invalid状态切换到Share状态(这个时候，CPU 0的cache line和内存地址0都是相同的最新数据)。

[3] sequence 2：CPU 3也对内存地址0执行load操作，这样内存地址0的数据被加载到CPU 3的cache line中，CPU 3的cache line从Invalid状态切换到Share状态(这个时候，CPU 0、CPU 3的cache line和内存地址0都是相同的最新数据)。

[4] sequence 3：CPU 0执行对内存地址8的load操作，(内存地址0和8共用一个cache line set)由于cache line已经存放了内存地址0的数据，这个时候，CPU 0需要将cache line的数据清理掉(Invalidation)以便腾出空间存放内存地址8的数据。由于，当前cache line的状态是Share，CPU 0不需要通知其他CPU，CPU 0在Invalidation cache line的数据后，就加载内存地址8的数据到cache line中，并将cache line状态改成Share。

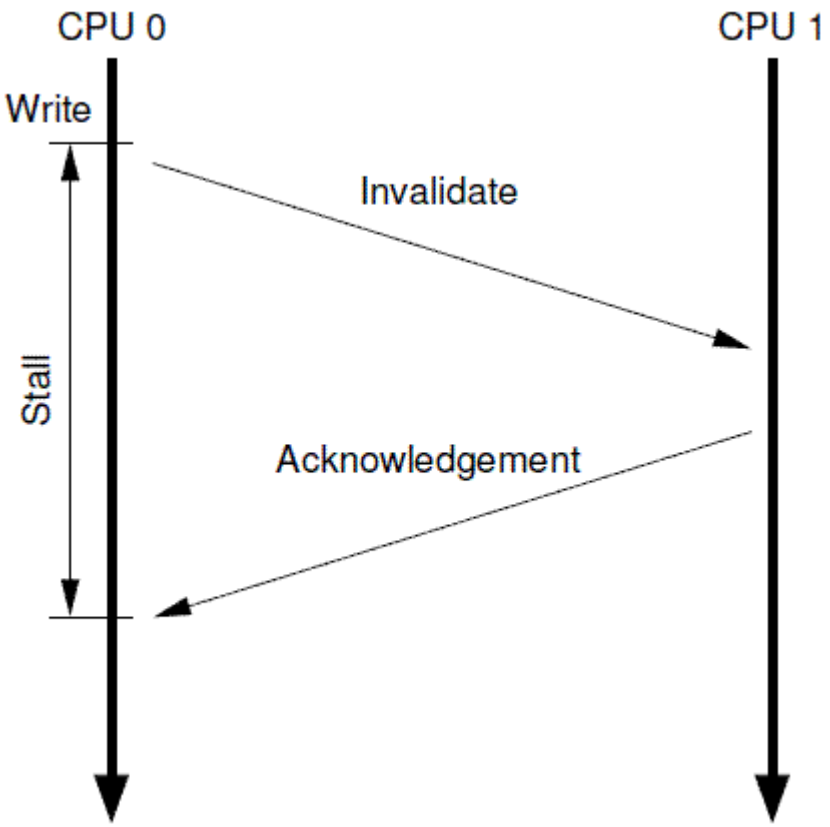
[5] sequence 4：CPU 2对内存地址0执行load操作，由于CPU 2知道程序随后会修改该值，它需要独占该数据，因此CPU 2向总线发送了read invalidate命令，一方面获取该数据(自己的local cache中没有地址0的数据)，另外，CPU 2想独占该数据（因为随后要write）。这个操作导致CPU 3的cacheline迁移到invalid状态。当然，这时候，memory仍然是最新的有效数据。

[6] sequence 5：CPU 2对内存地址0执行Store操作，由于CPU 2的cache line是Exclusive状态(对内存地址0的数据是独占状态的)，于是CPU 2可以直接将新的值写入cache line覆盖老值，cache line状态转换成Modified状态。(这个时候，内存地址0中的数据已经是Invalid的，其他CPU如果想load内存地址0的数据，不能直接从内存地址0加载数据了，需要嗅探(snoop)的方式从CPU 2的local cache中获取。

[7] sequence 6：CPU 1对内存地址0执行一个原子加操作。这时候CPU 1会发出read invalidate命令，将地址0的数据从CPU 2的cache line中嗅探得到，同时通过invalidate其他CPU local cache的内容而获得独占性的数据访问权。这时候，CPU 2中的cache line状态变成invalid状态，而CPU 1的cache line将从invalid状态迁移到modified状态。

[8] sequence 7：CPU 1对内存地址8执行load操作。由于cache line已经存放了内存地址0的数据，并且该状态是modified的，CPU 1需要将cache line的数据写回地址0，于是执行write back操作将地址0的数据写回到memory(这个时候，内存地址0中的数据从Invalid变成有效的)。接着，CPU 1发出read命令，从CPU 0中得到内存地址8的数据，并写入自己的cache line，cache line状态转换成Share。

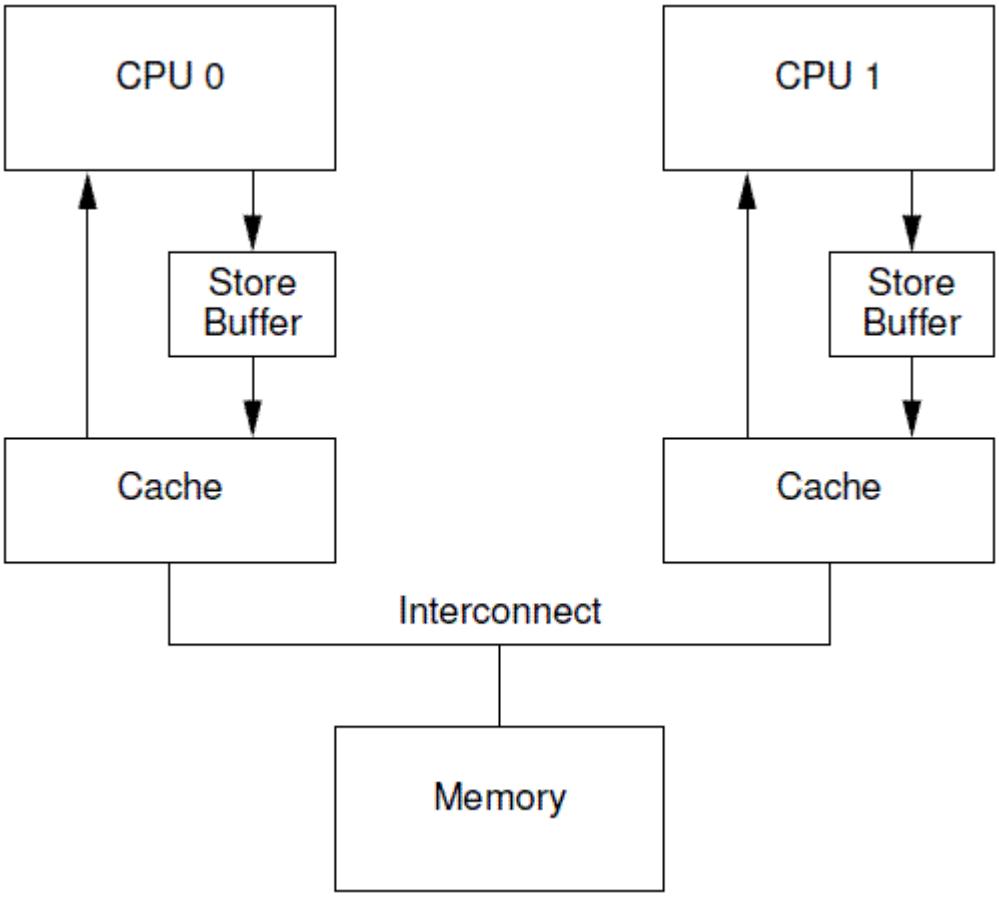
通过上面的例子，我们发现，对于某些特定地址的数据（在一个cache line中）重复的进行读写，这种结构可以获得很好的性能(例如，在sequence 5，CPU 2反复对内存地址0进行store操作将获得很好的性能，因为，每次store操作，CPU 2仅仅需要将新值写入自己的local cache即可)，不过，对于第一次写，其性能非常差，如图：



cpu 0发起一次对某个地址的写操作，但是local cache没有数据，该数据在CPU 1的local cache中，因此，为了完成写操作，CPU 0发出invalidate的命令，invalidate其他CPU的cache数据。只有完成了这些总线上的transaction之后，CPU 0才能正在发起写的操作，这是一个漫长的等待过程。

6.2.3 Store Buffer

对于CPU 0来说，这样的漫长等待显得有点没必要，因为，CPU 1中的cache line保存有什么样子的数据，其实都没有意义，这个值都会被CPU 0新写入的值覆盖的。为了给CPU 0提速，需要将这种同步阻塞等待，变成异步处理。于是，硬件工程师，修改CPU架构，在CPU和cache之间增加store buffer这个HW block，如下图所示：



一旦增加了store buffer，那么cpu 0无需等待其他CPU的相应，只需要将要修改的内容放入store buffer，然后继续执行就OK了。当cache line完成了bus transaction，并更新了cache line的状态后，要修改的数据将从store buffer进入cache line。引入了store buff，带来了一些复杂性，一不小心，会带来本地数据不一致的问题。我们先看看下面的代码：

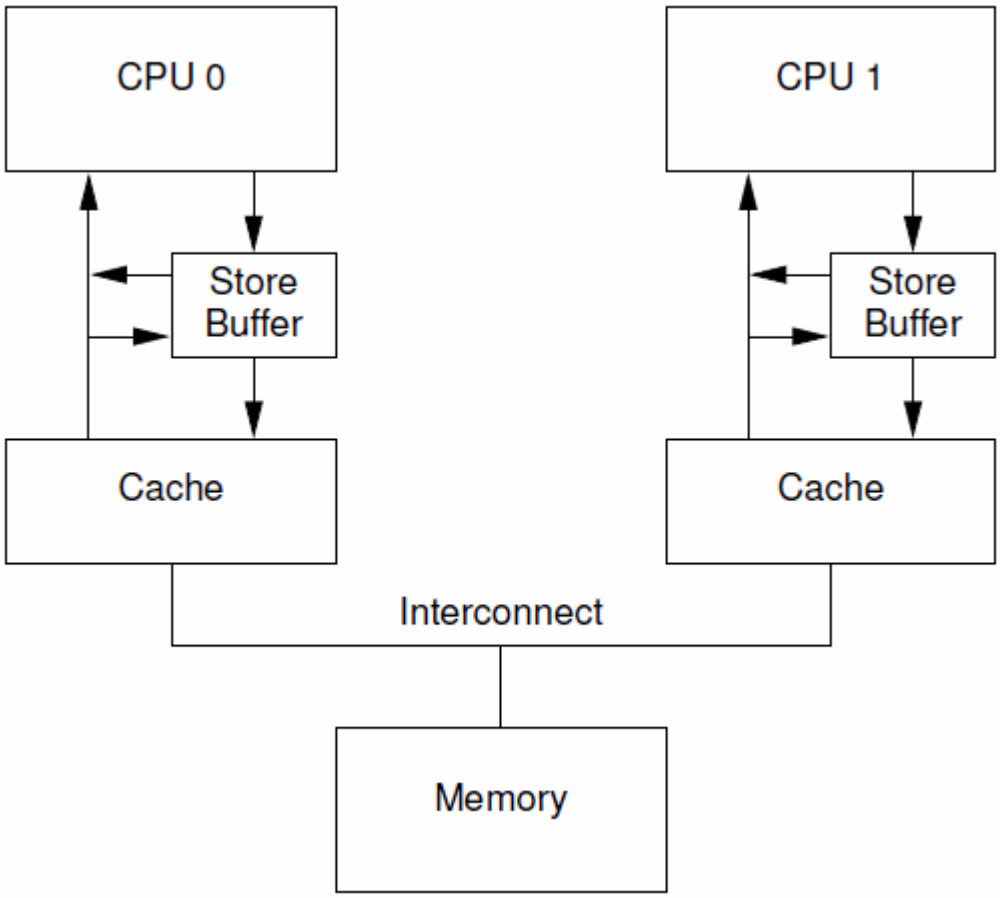
```
1 a = 1;
2 b = a + 1;
3 assert(b == 2);
```

a和b都是初始化为0，并且变量a在CPU 1的cache line中，变量b在CPU 0的cacheline中。

如果cpu执行上述代码，那么第三行的assert不应该失败，不过，如果CPU设计者使用上图中的那个非常简单的store buffer结构，那么你应该会遇到“惊喜”（assert失败了）。具体的执行序列过程如下：

```
[1] CPU 0执行a=1的赋值操作, CPU 0遇到cache miss
[2] CPU 0发送read invalidate消息以便从CPU 1那里获得数据，并invalid其他cpu保存a数据的local cache line。
[3] 由于store buff的存在，CPU 0把要写入的数据“1”放入store buffer
[4] CPU 1收到read invalidate后回应，把本地cache line的数据发送给CPU 0并清空本地cache中a的数据。
[5] CPU 0执行b = a + 1
[6] CPU 0 收到来自CPU 1的数据，该数据是“0”
[7] CPU 0从cache line中加载a，获得0值
[8] CPU 0将store buffer中的值写入cache line，这时候cache中的a值是“1”
[9] CPU 0执行a+1，得到1并将该值写入b
[10] CPU 0 executes assert(b == 2), which fails. OMG，你期望b等于2，但是实际上b等于了1
```

导致这个问题的根本原因是我们有两个a值，一个在cache line中，一个在store buffer中。store buffer的引入，违反了每个CPU按照其视角来观察自己的行为的时候必须是符合program order的原则。一旦违背这个原则，对软件工程师而言就是灾难。还好，有”好心“的硬件工程师帮助我们，修改了CPU的设计如下：



这种设计叫做store forwarding，当CPU执行load操作的时候，不但要看cache，还有看store buffer是否有内容，如果store buffer有该数据，那么就采用store buffer中的值。有了store forwarding的设计，上面的步骤[7]中就可以在store buffer获取正确的a值是”1“而不是”0“，因此计算得到的b的结果就是2，和我们预期的一致了。 store forwarding解决了CPU 0的cache line和store buffer间的数据一致性问题，但是，在CPU 1的角度来看，是否也能看到一致的数据呢？我们来看下一个例子：

```
1 void foo(void)
2 {
3       a = 1;
4       b = 1;
5 }
6
7 void bar(void)
8 {
9       while (b == 0) continue;
10  assert(a == 1);
11 }
```

同样的，a和b都是初始化成0.

我们假设CPU 0执行foo函数，CPU 1执行bar函数，a变量在CPU 1的cache中，b在CPU 0 cache中，执行的操作序列如下：

[1] CPU 0执行a=1的赋值操作，由于a不在local cache中，因此，CPU 0将a值放到store buffer中之后，发送了read invalidate命令到总线上去。
[2] CPU 1执行 while (b == 0) 循环，由于b不在CPU 1的cache中，因此，CPU发送一个read message到总线上，看看是否可以从其他cpu的local cache中或者memory中获取数据。
[3] CPU 0继续执行b=1的赋值语句，由于b就在自己的local cache中（cacheline处于modified状态或者exclusive状态），因此CPU0可以直接操作将新的值1写入cache line。
[4] CPU 0收到了read message，将最新的b值"1"回送给CPU 1，同时将b cacheline的状态设定为shared
[5] CPU 1收到了来自CPU 0的read response消息，将b变量的最新值"1"值写入自己的cacheline，状态修改为shared。
[6] 由于b值等于1了，因此CPU 1跳出while (b == 0)的循环，继续前行。
[7] CPU 1执行assert(a == 1)，这时候CPU 1的local cache中还是旧的a值，因此assert(a == 1)失败。
[8] CPU 1收到了来自CPU 0的read invalidate消息，以a变量的值进行回应，同时清空自己的cacheline，但是这已经太晚了。
[9] CPU 0收到了read response和invalidate ack的消息之后，将store buffer中的a的最新值"1"数据写入cacheline，然并卵，CPU 1已经assertion fail了。

CPU 1出现异常的assertion fail的根本原因是，CPU 0在发出read invalidate message后，并没有等待CPU 1收到，就继续执行将b改写为1，也就是store buffer的存在导致了CPU 1先看到了b修改为1，后看到a被修改为1。遇到这样的问题，CPU设计者也不能直接帮什么忙(除非去掉store buffer)，毕竟CPU并不知道哪些变量有相关性，这些变量是如何相关的。不过CPU设计者可以间接提供一些工具让软件工程师来控制这些相关性。这些工具就是memory-barrier指令。要想程序正常运行，必须增加一些memory barrier的操作，具体如下：

```
1 void foo(void)
2 {
3       a = 1;
4       smp_mb();
5       b = 1;
6 }
7
8 void bar(void)
9 {
10      while (b == 0) continue;
11      assert(a == 1);
12}
```

smp_mb() 这个内存屏障的操作会在执行后续的store操作之前，首先flush store buffer（也就是将之前的值写入到cacheline中）。达到这个目标有两种方法：

[1] CPU遇到smp_mb内存屏障后，需要等待store buffer中的数据完成transaction并将strore buffer中的数据写入cache line；

[2] CPU在遇到smp_mb内存屏障后，可以继续前行，但是需要记录一下store buffer中的数据顺序，在store buffer中的数据严格按顺序全部写回cache line之前，其他数据不能先更新cache line，需要按照顺序先写到store buffer才能继续前行。

通常采用的是方法[2]，增加了smp_mb()后，执行序列如下：

[1] CPU 0执行a=1的赋值操作，由于a不在local cache中，因此，CPU 0将a值放到store buffer中之后，发送了read invalidate命令到总线上去。
[2] CPU 1执行 while (b == 0) 循环，由于b不在CPU 1的cache中，因此，CPU发送一个read message到总线

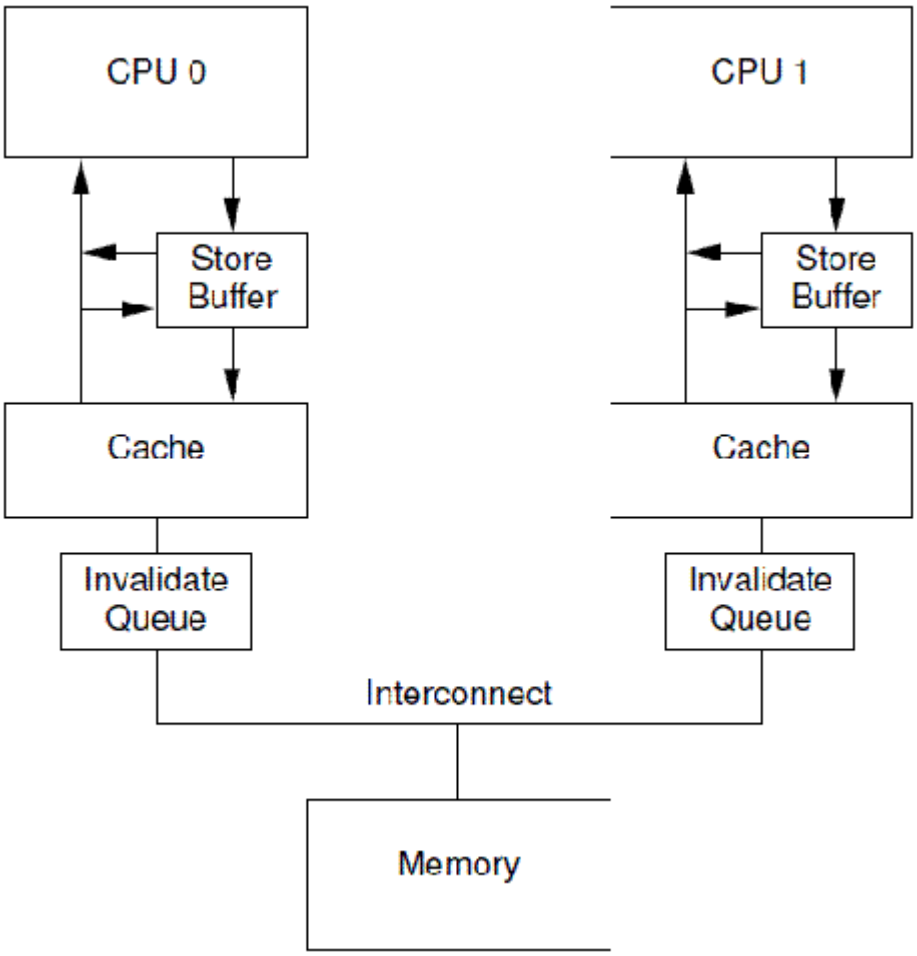
上，看看是否可以从其他cpu的local cache中或者memory中获取数据。
[3] CPU 0执行smp_mb()函数，给目前store buffer中的所有项做一个标记（后面我们称之marked entries）。当然，针对我们这个例子，store buffer中只有一个marked entry就是“a=1”。
[4] CPU 0继续执行b=1的赋值语句，虽然b就在自己的local cache中（cacheline处于modified状态或者exclusive状态），不过在store buffer中有marked entry，因此CPU 0不能直接操作将新的值1写入cache line，取而代之是b的新值'1'被写入store buffer(CPU 0也可以不执行b=1语句，等到a的transaction完成并写回cache line，在执行b=1，将b的新值'1'写入cache line)，当然是unmarked状态。
[5] CPU 0收到了read message，将b值”0“（新值”1“还在store buffer中）回送给CPU 1，同时将b cacheline的状态设定为shared。
[6] CPU 1收到了来自CPU 0的read response消息，将b变量的值（'0'）写入自己的cacheline，状态修改为shared。
[7] 由于smp_mb内存屏障的存在，b的新值'1'隐藏在CPU 0的store buffer中，CPU 1只能看到b的旧值'0'，这时CPU 1处于死循环中。
[8] CPU 1收到了来自CPU 0的read invalidate消息，以a变量的值进行回应，同时清空自己的cacheline。
[9] CPU 0收到CPU 1的响应msg，完成了a的赋值transaction，CPU 0将store buffer中的a值写入cacheline，并且将cacheline状态修改为modified状态。
[10] 由于store buffer只有一项marked entry（对应a=1），因此，完成step 9之后，store buffer的b也可以进入cacheline了。不过需要注意的是，当前b对应的cache line的状态是shared。
[11] CPU 0想将store buffer中的b的新值'1'写回cache line。由于b的cache line是share的。CPU 0需要发送invalidate消息，请求b数据的独占权。
[12] CPU 1收到invalidate消息，清空自己b的 cache line，并回送acknowledgement给CPU 0。
[13] CPU 1的某次循环执行到while (b == 0)，这时发现b的cache line是Invalid的了，于是CPU 1发送read消息，请求获取b的数据。
[14] CPU 0收到acknowledgement消息，将b对应的cache line修改成exclusive状态，这时候，CPU 0终于可以将b的新值1写入cache line了。
[15] CPU 0收到read消息，将b的新值1回送给CPU 1，同时将其local cache中b对应的cacheline状态修改为shared。
[16] CPU 1获取来自CPU 0的b的新值，将其放入cache line中。
[17] 由于b值等于1了，因此CPU 1跳出while (b == 0)的循环，继续前行。
[18] CPU 1执行assert(a == 1)，不过这时候a值没有在自己的cache line中，因此需要通过cache一致性协议从CPU 0那里获得，这时候获取的是a的最新值，也就是1值，因此assert成功。

从上面的执行序列可以看出，在调用memory barrier指令之后，使得CPU 0迟迟不能将b的新值'1'写回cache line，从而使得CPU 1一直不能观察到b的新值'1'，造成CPU 1一直不能继续前行。直观上CPU 0似乎不受什么影响，因为CPU 0可以继续前行，只是将b的新值'1'写到store buffer而不能写回cache line。不幸的是：每个cpu的store buffer不能实现的太大，其entry的数目不会太多。当cpu 0以中等的频率执行store操作的时候（假设所有的store操作导致了cache miss），store buffer会很快的被填满。在这种状况下，CPU 0只能又进入等待状态，直到cache line完成invalidation和ack的交互之后，可以将store buffer的entry写入cacheline，从而为新的store让出空间之后，CPU 0才可以继续执行。这种状况恰恰在调用了memory barrier指令之后，更容易发生，因为一旦store buffer中的某个entry被标记了，那么随后的store都必须等待invalidation完成，因此不管是否cache miss，这些store都必须进入store buffer，这样就很容易塞满store buffer。

6.2.4 Invalidate Queue

store buffer之所以很容易被填充满，主要是其他CPU回应invalidate acknowledge比较慢，如果能够加快这个过程，让store buffer尽快进入cache line，那么也就不会那么容易填满了。 invalidate acknowledge不能尽快回复的主要原因是invalidate cacheline的操作没有那么快完成，特别是cache比较繁忙的时候，这时，CPU往往进行密集的loading和storing的操作，而来自其他CPU的，对本CPU local cacheline的操作需要和本CPU的密集的cache操作进行竞争，只要完成了invalidate操作之后，本CPU才会发生invalidate acknowledge。此外，如果短时间内收到大量的invalidate消息，CPU有可能跟不上处理，从而导致其他CPU不断的等待。 要想达到快速回复acknowledgement，一个解决方法是，引入一个缓冲队列，接收到invalidate请求，可以先将请求入队缓冲队列，

就可以回复acknowledgement消息了，后面在异步完成invalidate操作。于是硬件工程师，引入一个invalidate queue，有invalidate queue的系统结构如下图所示：



异步延后处理，也需要有个度才行。一旦将一个invalidate（例如针对变量a的cacheline）消息放入CPU的Invalidate Queue，实际上该CPU就等于作出这样的承诺：在处理完该invalidate消息之前，不会发送任何相关（即针对变量a的cacheline）的MESI协议消息。为什么是在发出某个变量a的MESI协议消息的时候，需求去检查invalidate queue看是否有变量a的invalidate消息呢？而不是在对该变量的任何操作都需要检查以下invalidate queue呢？其实这样在保证MESI协议正确性的情况下，进一步保证性能的折中方案。 因为，在单纯考虑性能的情况下，少去检查invalidate queue，周期性(一定时间，cpu没那么繁忙、invalidate queue容量达到一定)批量处理invalidate queue中的消息，这样性能能够达到最佳。但是，这样在某些情况下，使得MESI协议失效。例如：在一个4核的机器上，变量a初始值是'0'，它cache在CPU 0和CPU 1的cache line中，状态都是Share。

```
[1] CPU 0需要修改变量a的值为'1'，CPU 0发送invalidate消息给其他CPU(1~3).
[2] 其他CPU(1~3)将invalidate消息放入invalidate queue，然后都回复给CPU 0.
[3] CPU 0收到响应后，将a的新值'1'写入cache line并修改状态为Modified。
[4] CPU 2需要读取a的时候遇到cache miss，于是CPU 2发送read消息给其他CPU，请求获取a的数据。
[5] CPU 1收到read请求，由于a在自己的cache line并且是share状态的，于是CPU 1将a的invalid值'0'响应给CPU 2。
[6] CPU 2通过一个read消息获取到一个过期的非法的值，这样MESI协议无法保证数据一致性了。
```

于是，为了保证MESI协议的正确性，CPU在需要发出某个变量的a的MESI协议消息的时候，需要检查invalidate queue中是否有该变量a的invalidate消息，如果有需要先出来完成这个invliadte消息后，才能发出正确的MESI协议消息。在合适的时候，发出正确的MESI协议是保证了不向其他CPU传递错误的信息，从而保证数据的一致性。但是，对于本CPU是否也可以高枕无忧呢？我们来看同上面一样的一个例子：

```
1 void foo(void)
2 {
3       a = 1;
4       smp_mb();
5       b = 1;
6 }
7
8 void bar(void)
9 {
10      while (b == 0) continue;
11      assert(a == 1);
12 }
```

在上面的代码片段中，我们假设a和b初值是0，并且a在CPU 0和CPU 1都有缓存的副本，即a变量对应的CPU0和CPU 1的cacheline都是shared状态。b处于exclusive或者modified状态，被CPU 0独占。我们假设CPU 0执行foo函数，CPU 1执行bar函数，执行序列如下：

```
[1] CPU 0执行a=1的赋值操作，由于a在CPU 0 local cache中的cacheline处于shared状态，因此，CPU 0将a的新值“1”放入store buffer，并且发送了invalidate消息去清空CPU 1对应的cacheline。
[2] CPU 1执行while (b == 0)的循环操作，但是b没有在local cache，因此发送read消息试图获取该值。
[3] CPU 1收到了CPU 0的invalidate消息，放入Invalidate Queue，并立刻回送Ack。
[4] CPU 0收到了CPU 1的invalidate ACK之后，即可以越过程序设定内存屏障（第四行代码的smp_mb() ），这样a的新值从store buffer进入cacheline，状态变成Modified。
[5]  CPU 0 越过memory barrier后继续执行b=1的赋值操作，由于b值在CPU 0的local cache中，因此store操作完成并进入cache line。
[6] CPU 0收到了read消息后将b的最新值“1”回送给CPU 1，并修正该cache line为shared状态。
[7] CPU 1收到read response，将b的最新值“1”加载到local cacheline。
[8] 对于CPU 1而言，b已经等于1了，因此跳出while (b == 0)的循环，继续执行后续代码
[9] CPU 1执行assert(a == 1)，但是由于这时候CPU 1 cache的a值仍然是旧值0，因此assertion 失败
[10] 该来总会来，Invalidate Queue中针对a cacheline的invalidate消息最终会被CPU 1执行，将a设定为无效，但，大错已经酿成。
```

CPU 1出现assert失败，是因为没有及时处理invalidate queue中的a的invalidate消息，导致使用了本cache line中的一个已经是invalid的一个旧的值，这是典型的cache带来的一致性问题。这个时候，我们也需要一个memory barrier指令来告诉CPU，这个时候应该需要处理invalidate queue中的消息了，否则可能会读到一个invalid的旧值。

```
当CPU执行memory barrier指令的时候，对当前Invalidate Queue中的所有的entry进行标注，这些被标注的项被称为marked entries，而随后CPU执行的任何的load操作都需要等到Invalidate Queue中所有marked entries完成对cacheline的操作之后才能进行
```

因此，要想保证程序逻辑正确，我们需要给bar函数增加内存屏障的操作，具体如下：

```
1 void foo(void)
2 {
3       a = 1;
4       smp_mb();
5       b = 1;
6 }
7
8 void bar(void)
9 {
10      while (b == 0) continue;
11      smp_mb();
12      assert(a == 1);
13 }
```

bar()函数添加smp_mb内存屏障后，执行序列如下：

```
[1] ~ [8] 同上
[9] CPU 1遇到smp_mb内存屏障，发现下一条语句是load a，这个时候CPU 1不能继续执行代码，只能等待，直到Invalidate Queue中的message被处理完成
[10] CPU 1处理Invalidate Queue中缓存的Invalidate消息，将a对应的cacheline设置为无效。
[11] 由于a变量在local cache中无效，因此CPU 1在执行assert(a == 1)的时候需要发送一个read消息去获取a值。
[12] CPU 0用a的新值1回应来自CPU 1的请求。
[13] CPU 1获得了a的新值，并放入cacheline，这时候assert(a == 1)不会失败了。
```

在我们上面的例子中，memory barrier指令对store buffer和invalidate queue都进行了标注，不过，在实际的代码片段中，foo函数不需要mark invalidate queue，bar函数不需要mark store buffer。因此，许多CPU architecture提供了弱一点的memory barrier指令只mark其中之一。如果只mark invalidate queue，那么这种memory barrier被称为read memory barrier。相应的，write memory barrier只mark store buffer。一个全功能的memory barrier会同时mark store buffer和invalidate queue。 我们一起来看看读写内存屏障的执行效果：对于read memory barrier指令，它只是约束执行CPU上的load操作的顺序，具体的效果就是CPU一定是完成read memory barrier之前的load操作之后，才开始执行read memory barrier之后的load操作。read memory barrier指令象一道栅栏，严格区分了之前和之后的load操作。同样的，write memory barrier指令，它只是约束执行CPU上的store操作的顺序，具体的效果就是CPU一定是完成write memory barrier之前的store操作之后，才开始执行write memory barrier之后的store操作。全功能的memory barrier会同时约束load和store操作，当然只是对执行memory barrier的CPU有效。 现在，我们可以改一个用读写内存屏障的版本了，具体如下：

```
1 void foo(void)
2 {
3       a = 1;
4       smp_wmb();
5       b = 1;
6 }
7
8 void bar(void)
9 {
10      while (b == 0) continue;
11      smp_rmb();
12      assert(a == 1);
13 }
```

可见，memory barrier需要成对使用才能保证程序的正确性。什么情况下使用memory barrier，使用怎样的memory barrier，和CPU架构有那些相关性呢？

6.3 How Memory Barriers ?

memory barrier的语义在不同CPU上是不同的，因此，想要实现一个可移植的memory barrier的代码需要对形形色色的CPU上的memory barrier进行总结。幸运的是，无论哪一种cpu都遵守下面的规则：

```
[1]、从CPU自己的视角看，它自己的memory order是服从program order的
[2]、从包含所有cpu的sharebility domain的角度看，所有cpu对一个共享变量的访问应该服从若干个全局存储顺序
[3]、memory barrier需要成对使用
[4]、memory barrier的操作是构建互斥锁原语的基石
```

6.3.1 有条件的顺序保证

要保证程序在多核CPU中执行服从program order，那么我们需要成对使用的memory barrier，然而成对的memory barrier并不能提供绝对的顺序保证，只能提供有条件的顺序保证。那么什么是有条件的顺序保证？考虑下面一个访问例子(这里的access可以是读或写)：

CPU1	CPU2
access(A);	access(B);
smp_mb();	smp_mb();
access(B);	access(A);

从CPU1角度来看，对A的访问总是先于对B的访问。但是，关键的是从CPU2的角度来看，CPU1对A、B的访问顺序是否就一定是A优先于B呢？假如在CPU2感知CPU1对A的访问结果的情况下，是否可以保证CPU2也能感知CPU1对B的访问结果呢？这是不一定的，例如执行时序如下，那么显然，在CPU2感知CPU1对A的访问结果的情况下，是并不能感知CPU1对B的访问结果(CPU2对A的访问要早于CPU1对B的访问)。

时序	CPU1	CPU2
1	access(A);	
2	smp_mb();	
3		access(B);
4		smp_mb();
5		access(A);
6	access(B);	

另外，如果CPU1对B的访问结果已经被CPU2感知到了，那么，在这个条件下，CPU1对A的访问结果就一定能够被CPU2感知到。这就是观察者(CPU2)在满足一定条件下才能保证这个memory的访问顺序。

对于上面例子中的access操作，在内存上包括load和store两种不同操作，下面列出了CPU1和CPU2不同的操作组合共16个，下面来详细描述一下，在不同的操作组合下memory barrier可以做出怎样的保证。

	CPU 1		CPU 2		Description
0	load(A)	load(B)	load(B)	load(A)	Ears to ears.
1	load(A)	load(B)	load(B)	store(A)	Only one store.
2	load(A)	load(B)	store(B)	load(A)	Only one store.
3	load(A)	load(B)	store(B)	store(A)	Pairing 1.
4	load(A)	store(B)	load(B)	load(A)	Only one store.
5	load(A)	store(B)	load(B)	store(A)	Pairing 2.
6	load(A)	store(B)	store(B)	load(A)	Mouth to mouth, ear to ear.
7	load(A)	store(B)	store(B)	store(A)	Pairing 3.
8	store(A)	load(B)	load(B)	load(A)	Only one store.
9	store(A)	load(B)	load(B)	store(A)	Mouth to mouth, ear to ear.
A	store(A)	load(B)	store(B)	load(A)	Ears to mouths.
B	store(A)	load(B)	store(B)	store(A)	Stores “pass in the night”.
C	store(A)	store(B)	load(B)	load(A)	Pairing 1.
D	store(A)	store(B)	load(B)	store(A)	Pairing 3.
E	store(A)	store(B)	store(B)	load(A)	Stores “pass in the night”.
F	store(A)	store(B)	store(B)	store(A)	Stores “pass in the night”.

由于CPU架构千差万别，上面的16种组合可以分成3类

```
[1] Portable Combinations -- 通杀所有CPU
[2] Semi-Portable Combinations -- 现代CPU可以work，但是不适应在比较旧的那些CPU
[3] Dubious Combinations -- 基本是不可移植的
```

6.3.1.1 通杀所有CPU

(1) Pairing 1 情况3，CPU执行代码如下：（A和B的初值都是0）

CPU1	CPU2
X = A;	B = 1;
smp_mb();	smp_mb();
Y = B;	A = 1;

对于这种情况，两个CPU都执行完上面的代码后，如果X的值是1，那么我们可以断定Y也是等于1的。也就是如果CPU1感知到了CPU2对A的访问结果，那么可以断定CPU1也必能感知CPU2对B的访问结果。但是，如果X的值是0，那么memory barrier的条件不存在，于是Y的值可能是0也可能是1。 对于情况C，它是和情况1是对称，于是结论也是类似的：（A和B的初值都是0）

CPU1	CPU2
A = 1;	Y = B;
smp_mb();	smp_mb();
B = 1;	X = A;

同样，两个CPU都执行完上面的代码后，如果Y的值是1，那么可以断定X的值也是1。 (2) Pairing 2 情况5，CPU执行代码如下：（A和B的初值都是0）

CPU1	CPU2
X = A;	Y = B;
smp_mb();	smp_mb();
B = 1;	(Z = X;)A = 1;

两个CPU都执行完上面的代码后，在不影响逻辑的情况下，在CPU2的A=1;前面插入代码Z=X，根据情况C，如果Y的值是1，那么Z的值就一定是A，由于Z=X执行在A=1前面，那么Z的值是A的初始值0，于是X的值一定是0。同样，如果X等于1，那么我们一定可以得到Y等于0; (3) Pairing 3 情况7，CPU执行代码如下：（A和B的初值都是0）

CPU1	CPU2
X = A;	B = 2;
smp_mb();	smp_mb();
(Z = B;) B = 1;	A = 1;

两个CPU都执行完上面的代码后，在不影响逻辑的情况下，在CPU1的B=1;前面插入代码Z=B，根据情况3，如果X等于1，那么可以断定Z等于2，也就是在CPU1执行完毕Z=B代码前，B的值是2，由于CPU1在执行完Z=B后会执行B=1，于是对CPU1而已，最后B的值是1。 通过上面(1)，如果CPU1执行的全是store操作，而CPU2执行的全是load操作(对称下，CPU2执行的全是store操作，而CPU1执行的全是load操作)，那么会有一个memory barrier条件使得执行得到一个确定的顺序，并且是通吃所有CPU的。而，(2)和(3)经过插入代码也可以转换成(1)的情况。 情况D，CPU执行代码如下：（A和B的初值都是0）

CPU1	CPU2
A = 1;	Y = B;
smp_mb();	smp_mb();
B = 1;	(Z = A;)A = 2;

该情况是情况7是类似的。在Y等1的时候，最终A等于2.

6.3.1.2 现代CPU可以work，但是不适应在比较旧的那些CPU

(1) Ears to Mouths 情况A，CPU执行代码如下：（A和B初值都是0，其他变量初始值是-1）

CPU1	CPU2
A = 1;	B = 1; (Z = X;)
smp_mb();	smp_mb();
X = B;	Y = A;

这种情况下，比较容易推算出X等1的时候，Y可能为0也可能为1，当X等于0的时候，也比较容易推算出Y值可以为1。但是，X等0的时候，Y有没可能也是0呢？ 我们通过插入代码(Z=X)，这样就转换成情况C，在X等于0，Z等于0的时候，那么memory barrier条件成立，于是Y必然等1。然而，如果X等于0的时候，Z不等于0，这个时候memory barrier条件就不能成立了，这个时候Y就可能为0。 下面我们来讲下，上面情况下会出现X和Y同时为0。在一个有Invalidate queue和store buffer的系统中，B和X在CPU1的local cache中并且是独占的，A和Y在CPU2的local cache中并且也是独占的。CPUs的执行序列如下：

```
[1] CPU1对A发起store操作，由于A不在CPU1的cache中，CPU1发起invalidate message，当然，CPU1不会停下它的脚步，将A的新值'1'放入store buffer，它就继续往下执行
[2] smp_mb使得CPU1对store buffer中的entry进行标注（当然也对Invalidate Queue进行标注，不过和本场景无关），store A的操作变成marked状态
[3] CPU2对B发起store操作，由于B不在CPU2的cache中，CPU2发起invalidate message，当然，CPU2不会停下它的脚步，将B的新值'1'放入store buffer，它就继续往下执行
[4] CPU2收到CPU1的invalidate message将该message放入Invalidate Queue后继续前行。
[5] smp_mb使得CPU2对store buffer中的entry进行标注（当然也对Invalidate Queue进行标注），store B的操作变成marked状态
[6] CPU1收到CPU2的invalidate message将该message放入Invalidate Queue后继续前行。
[7] CPU1前行执行load B，由于B在CPU1的local cache独占的(CPU1并不需要发送任何MESI协议消息，它并不需要立即处理Invalidate Queue里面的消息)，于是CPU1从local cache中得到B的值'0'，接着CPU1继续执行store X，由于X也在CPU1的local cache独占的，于是，CPU1将X的新值修改为B的值'0'并将其放入store buffer中。
[8] CPU2前行执行load A，由于A在CPU2的local cache独占的(CPU2并不需要发送任何MESI协议消息，它并不需要立即处理Invalidate Queue里面的消息)，于是CPU2从local cache中得到A的值'0'，接着CPU2继续执行store Y，由于Y也在CPU2的local cache独占的，于是，CPU2将Y的新值修改为B的值'0'并将其放入store buffer中。
[9] CPU1开始处理Invalidate Queue里面的消息，将本local cache中的B置为Invalide，同时响应Invalidate response message给CPU2
[10] CPU2收到Invalidate response message后，这个时候可以将store buffer里面的B和Y写回cache line，最后B为1，Y为0。
[11] CPU1和CPU2类似，最终A为1，X为0.
```

(2) Pass in the Night 情况F，CPU执行代码如下：（A和B初值都是0，其他变量初始值是-1）

CPU1	CPU2
A = 1;	B = 2;
smp_mb();	smp_mb();
B = 1;	A = 2;

情况F，正常情况下，无论如何，但是无论如何，在两个CPU都执行完上面的代码之后{A==1,B==2} 这种情况不可能发生。不幸的是，在一些老的CPU架构上，是可能出现{A==1,B==2} 的，出现这种情况和上面的原因有点类似，下面也简单描述一下，在一个有Invalidate queue和store buffer的系统中，B在CPU1的local cache中并且是独占的，A在CPU2的local cache中并且也是独占的。CPUs的执行序列如下：

```
[1]~[6]和 (1) Ears to Mouths中的基本一样
[7] CPU1继续前行，由于B在CPU1的local cache独占的(CPU1并不需要发送任何MESI协议消息，它并不需要立即处理Invalidate Queue里面的消息)，于是，CPU1将B的新值'1'放入store buffer中。
[8] CPU2继续前行，由于A在CPU2的local cache独占的(CPU1并不需要发送任何MESI协议消息，它并不需要立即处理Invalidate Queue里面的消息)，于是，CPU2将A的新值'2'放入store buffer中。
[9] CPU1开始处理Invalidate Queue里面的消息，将本local cache中的B置为Invalide(这个时候store buffer里面B的新值'1'也被invalidate了)，同时响应Invalidate response message给CPU2
[9] CPU2开始处理Invalidate Queue里面的消息，将本local cache中的置为Invalide(这个时候store buffer里面A的新值'2'也被invalidate了)，同时响应Invalidate response message给CPU1
[10] CPU1收到Invalidate response message，这个时候可以将store buffer中的A=1刷到cache line，最终A的值为1
[11] CPU2收到Invalidate response message，这个时候可以将store buffer中的B=2刷到cache line，最终B的值为2
```

到这来，大家应该会发现，第一个赋值（对于CPU1而言是A = 1，对于CPU2而言是B = 2）其实是pass in the night，静悄悄的走过，而第二个赋值（对于CPU1而言是B = 1，对于CPU2而言是A = 2）则会后发先至，最终导致第一个赋值先发而后至覆盖第二个赋值。 其实，只要符合下面的使用模式，上面描述的操作顺序（第二个store的结果被第一个store覆盖）都是有可能发生的：

CPU1	**CPU2**
A = 1;	B = 2;
smp_mb();	smp_mb();
xxxx;	xxxx;

前面说的'ears to mouths'也是这种模式，不过，对于21世纪的硬件系统而言，硬件工程师已经帮忙解决了上面的问题，因此，软件工程师可以安全的使用Stores “Pass in the Night”。

####6.3.1.3 基本不可移植 剩下的情况0、1、2、4、6、8、9这7种情况的组合，即使是在21世纪的那些新的CPU硬件平台上，也是不能够保证是可移植的。当然，在一些硬件平台上，我们还是可以得到一些确定的执行顺序的。(1) Ears to Ears 情况0，CPUs上全是load操作

CPU1	**CPU2**
load A;	load B;
smp_mb();	smp_mb();
load B;	load A;

由于load操作不能改变memory的状态，因此，一个CPU上的load是无法感知到另外一侧CPU的load操作的。不过，如果CPU2上的load B操作返回的值比CPU 1上的load B返回的值新的话（即CPU2上load B晚于CPU1的load B执行），那么可以推断CPU2的load A返回的值要么和CPU1上的load A返回值一样新，要么加载更新的值。 (2) Mouth to Mouth, Ear to Ear 这个组合的特点是一个变量只是执行store操作，而另外一个变量只是进行load操

作。执行序列如下：

CPU1	CPU2
load A;	store B;
smp_mb();	smp_mb();
store B;	load A;

这种情况下，如果CPU2上的store B最后发生(也就是，上面代码执行完毕后，在执行一次load B得到的值是CPU2 store B的值)，那么可以推断CPU2的load A返回的值要么和CPU1上的load A返回值一样新，要么加载更新的值。
(3) Only One Store

CPU1	CPU2
load A;	load B;
smp_mb();	smp_mb();
load B;	store A;

这种情况下，只有一个变量的store操作可以被另外的CPU上的load操作观察到，如果在CPU1上运行的load A感知到了在CPU2上对A的赋值，那么，CPU1上的load B必然能观察到和CPU2上load B一样的值或者更新的值。

6.3.2 memory barrier内存屏障类型

6.3.2 .1 显式内存屏障

6.3.1章节列举的16种情况的例子中内存屏障smp_mb()指的是一种全功能内存屏障(General memory barrier)，然而全功能的内存屏障对性能的杀伤较大，某些情况下我们可以使用一些弱一点的内存屏障。在有Invalidate queue和store buffer的系统中，全功能的内存屏障既会mark store buffer也会mark invalidate queue，对于情况3，CPU1全是load它只需要mark invalidate queue即可，相反CPU2全是store，它只需mark store buffer即可，于是CPU1只需要使用读内存屏障(Read memory barrier)，CPU2只需使用写内存屏障(Write memory barrier)。 情况3，修改如下

CPU1	CPU2
X = A;	B = 1;
read_mb();	write_mb();
Y = B;	A = 1;

到这来，我们知道有3种不同的内存屏障，还没有其他的呢？我们来看一个例子： 初始化 int A = 1; int B = 2; int C = 3; int *P = &A; int *Q = &B;

CPU1	CPU2
B = 4;	Q = P;
write_mb()	
P = &B;	D = *Q;

*通常情况下，Q最后要么等于&A，要么等于&B。也就是说：Q == &A， D == 1 或者 Q == &B， D == 4，绝对不会出现Q == &B， D == 2的情形。然而，让人吃惊的是，DEC Alpha 下，就可能出现Q == &B， D == 2的情形。 于是，在DEC Alpha 下，CPU2上的Q=P 下面需要插入一个memory barrier来保证程序顺序，这来用一个读内存屏障(read_mb)即可，但是我们发现CPU2上的Q = P和D = *Q是一个数据依赖关系，是否可以引入一个更为轻量的内存屏障来解决呢？ 于是这里引入一种内存屏障-数据依赖内存屏障dd_mb(data dependency memory barrier)，dd_mb是一种比read_mb要弱一些的内存屏障（这里的弱是指对性能的杀伤力要弱一些）。read_mb适用所有的load操作，而ddmb要求load之间有依赖关系，即第二个load操作依赖第一个load操作的执行结果（例如：先load地址，然后load该地址的内容）。ddmb被用来保证这样的操作顺序：在执行第一个load A操作的时候（A是一个地址变量），务必保证A指向的数据已经更新。只有保证了这样的操作顺序，在第二load操作的时候才能获取A地址上保存的新值。*

CPU1	CPU2
B = 4;	Q = P;
write_mb()	data_dependency_mb()
P = &B;	D = *Q;

在纯粹的数据依赖关系下使用数据依赖内存屏障dd_mb来保证顺序，但是如果加入了控制依赖，那么仅仅使用dd_mb是不够的，需要使用read_mb，看下面例子：

CPU1	CPU2
	Q = &A;
	if (t)
B = 4;	Q = P;
write_mb()	data_dependency_mb()
P = &B;	D = *Q;

*由于加入了条件if (t)依赖，这就不是真正的数据依赖了，在这种情况下，CPU会进行分支预测，可能会"抄近路"先去执行*Q的load操作，在这种情况下，需要将data_dependency_mb改成read_mb。 到这来，我们知道有4中不同的内存屏障种类：

```
[1] Write (or store) memory barriers -- 写内存屏障
[2] Data dependency barriers -- 数据依赖内存屏障
[3] Read (or load) memory barriers -- 读内存屏障
[4] General memory barriers -- 全功能内存屏障
```

6.3.2 .2 隐式内存屏障

有些操作可以隐含memory barrier的功能，主要有两种类型的操作：一是加锁操作，另外一个是释放锁的操作。

```
[1] LOCK operations -- 加锁操作
[2] UNLOCK operations -- 释放锁操作
```

(1) 加锁操作被认为是一种half memory barrier，加锁操作之前的内存访问可以任意渗透过加锁操作，在其他执行，但是，另外一个方向绝对是不允许的：即加锁操作之后的内存访问操作，必须在加锁操作之后完成。 (2) 和lock操作一样，unlock也是half memory barrier。它确保在unlock操作之前的内存操作先于unlock操作完成，也就是说unlock之前的操作绝对不能越过unlock这个篱笆，在其后执行。当然，另外一个方向是OK的，也就是说，unlock之后的内存操作可以在unlock操作之前完成。 我们看下面一个例子：

```
1 *A = a;
2 LOCK
3 C = 1;
4 UNLOCK
5 *B = b;
```

上面的程序有可能按照下面的顺序执行：

```
2 LOCK
3 C = 1;
5 *B = b;
1 *A = a;
4 UNLOCK
```

通过上面，我们得知，经LOCK-UNLOCK对不能实现完全的内存屏障的功能，但是，它们也的确会影响内存访问顺序，参考下面的例子： 多个CPU对一把锁操作的场景：

CPU1	CPU2
A = a; LOCK M; B = b; C = c; UNLOCK M; D = d;	E = e; LOCK M; F = f; G = g; UNLOCK M; H = h;

这种情况下，CPU1或者CPU2，只能有一个进入临界区，如果是CPU1进入临界区的话，对A B C的赋值操作，必然在对F G H变量赋值之前完成。如果CPU2进入临界区的话，对E F G的赋值操作，必然在对B C D变量赋值之前完成。

6.3.3 C++11 memory order

要编写出正确的lock free多线程程序，我们需要在正确的位置上插入合适的memory barrier代码，然而不同CPU架构对于的memory barrier指令千差万别，要写出可移植的C++程序，我们需要一个语言层面的Memory Order规范，以便编译器可以根据不同CPU架构插入不同的memory barrier指令，或者并不需要插入额外的memory barrier指令。 有了这个Memory Order规范，我们可以在high level language层面实现对在多处理器中多线程共享内存交互的次序控制，而不用考虑compiler，CPU arch的不同对多线程编程的影响了。

C++11提供6种可以应用于原子变量的内存顺序：

```
[1] memory_order_relaxed
[2] memory_order_consume
[3] memory_order_acquire
[4] memory_order_release
[5] memory_order_acq_rel
[6] memory_order_seq_cst
```

上面6种内存顺序描述了三种内存模型(memory model)：

```
[1] sequential consistent(memory_order_seq_cst)
[2] relaxed(momory_order_relaxed)
[3] acquire release(memory_order_consume, memory_order_acquire, memory_order_release,
memory_order_acq_rel)
```

6.3.3.1 C++11中的各种关系

C++11引入上面6种内存顺序本质上是为了解决"visible side-effects"的问题，也就是读操作的返回值问题，通俗来讲：

```
线程1执行写操作A之后，如何可靠并高效地保证线程2执行的读操作B，load A的结果是完整可见的？
```

为了解决"visible side-effects"这个问题，C++11引入"happens-before"关系，其定义如下：

```
Let A and B represent operations performed by a multithreaded process. If A happens-
before B, then the memory effects of A effectively become visible to the thread
performing B before B is performed.
```

OK，现在问题就转化为：如何在A、B两个操作之间建立起happens-before关系。在推导happens-before关系前，我们先描述下面几个关系：

6.3.3.1.1 Sequenced-before 关系

定义如下：

```
Sequenced before is an asymmetric, transitive, pair-wise relation between evaluations
executed by a single thread, which induces a partial order among those evaluations.
```

Sequenced-before是在同一个线程内，对求值顺序关系的描述，它是非对称的，可传递的关系。

```
[1] 如果A is sequenced-before B，代表A的求值会先完成，才进行对B的求值
[2] 如果A is not sequenced before B 而且 B is sequenced before A，代表B的求值会先完成，才开始对A的求值。
[3] 如果A is not sequenced before B 而且 B is not sequenced before A，这样求值顺序是不确定的，可能A先于B，也可能B先于A，也可能两种求值重叠。
```

6.3.3.1.2 Carries a dependency 关系

定义如下：

```
Within the same thread, evaluation A that is sequenced-before evaluation B may also carry a dependency into B (that is, B depends on A), if any of the following is true
1) The value of A is used as an operand of B, except
        a) if B is a call to std::kill_dependency
        b) if A is the left operand of the built-in &&, ||, ?:, or , operators.
2) A writes to a scalar object M, B reads from M
3) A carries dependency into another evaluation X, and X carries dependency into B
```

简单来讲，carries-a-dependency-to 严格应用于单个线程，建立了 操作间的数据依赖模型：如果操作 A 的结果被操作 B 作为操作数,那么 A carries-a-dependency-to B(一个直观的例子：B=M[A])， carries a dependency具有传递性。

6.3.3.1.3 Dependency-ordered before 关系

该关系描述的是线程间的两个操作间的关系，定义如下：

```
Between threads, evaluation A is dependency-ordered before evaluation B if any of the following is true
1) A performs a release operation on some atomic M, and, in a different thread, B performs a consume operation on the same atomic M, and B reads a value written by any part of the release sequence headed by A.
2) A is dependency-ordered before X and X carries a dependency into B.
```

case 1指的是：线程1的操作A对变量M执行“release”写，线程2的操作B对变量M执行“consume”读，并且操作B读取到的值源于操作A之后的“release”写序列中的任何一个（包括操作A本身）。 case 2描述的是一种传递性。

6.3.3.1.4 Synchronized-with 关系

定义如下：

```
An atomic operation A that performs a release operation on an atomic object M synchronizes withan atomic operation B that performs an acquire operation on M and takes its value from any side effect in the release sequence headed by A.
```

该关系描述的是，对于在变量 x 上的写操作 W(x) synchronized-with 在该变量上的读操作 R(x), 这个读操作欲读取的值是 W(x) 或同一线程随后的在 x 上的写操作 W',或任意线程一系列的在 x 上的 read-modify-write 操作（如 fetch_add()或 compare_exchange_weak()）而这一系列操作最初读到 x 的值是 W(x) 写入的值。 例如：A Write-Release Can Synchronize-With a Read-Acquire，简单来说， 线程1的A操作写了变量x，线程2的B操作读了变量x，B读到的是A写入的值或者更新的值，那么A, B 间存在 synchronized-with 关系。

6.3.3.1.5 Inter-thread happens-before 关系

定义如下：

```
Between threads, evaluation A inter-thread happens before evaluation B if any of the
following is true
1) A synchronizes-with B
2) A is dependency-ordered before B
3) A synchronizes-with some evaluation X, and X is sequenced-before B
4) A is sequenced-before some evaluation X, and X inter-thread happens-before B
5) A inter-thread happens-before some evaluation X, and X inter-thread happens-before B
```

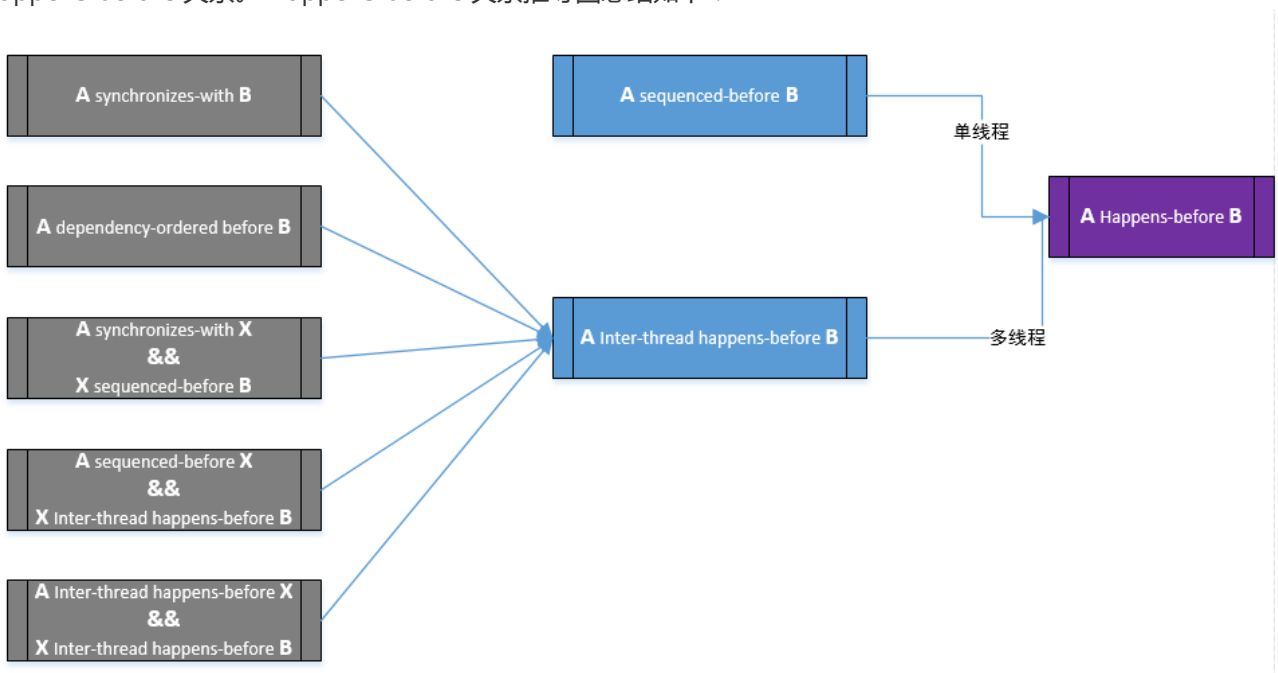
Inter-thread happens-before 关系具有传递性。该关系描述的是，如果A inter-thread happens-before B，则线程1的A操作对memory的访问结果，会在线程2的B操作执行前对线程2是可见的。

6.3.3.1.6 Happens-before 关系

定义如下：

```
Regardless of threads, evaluation A happens-before evaluation B if any of the following
is true:
1) A is sequenced-before B
2) A inter-thread happens before B
```

Happens-before 指明了哪些指令将看到哪些指令的结果。 对于单线程，sequenced-before关系即是Happens-before 关系，表明了操作 A 排列在另一个操作 B 之前。 对于多线程，则inter-thread happens before关系即是Happens-before 关系。 Happens-before 关系推导图总结如下：



6.3.3.2 6种memory order描述

下面我们分别来解析一下上面说的6种memory order的作用以及用法。 ####6 .3.3.2.1 顺序一致次序 - memory_order_seq_cst SC是C++11中原子变量的默认内存序，它意味着将程序看做是一个简单的序列。如果对于一个原子变量的操作都是顺序一致的，那么多线程程序的行为就像是这些操作都以一种特定顺序被单线程程序执行。 从同步的角度来看，一个顺序一致的 store 操作 synchroniezd-with 一个顺序一致的需要读取相同的变量的 load 操作。除此以外，顺序模型还保证了在 load 之后执行的顺序一致原子操作都得表现得在 store 之后完成。 顺序一致次序对内存序要求比较严格，对性能的损伤比较大。

6.3.3.2.2 松弛次序 - memory_order_relaxed

在原子变量上采用 relaxed ordering 的操作不参与 synchronized-with 关系。在同一线程内对同一变量的操作仍保持happens-before关系，但这与别的线程无关。在 relaxed ordering 中唯一的要求是在同一线程中，对同一原子变量的访问不可以被重排。 我们看下面的代码片段，x和y初始值都是0

```
// Thread 1:
r1 = y.load(std::memory_order_relaxed); // A
x.store(r1, std::memory_order_relaxed); // B
// Thread 2:
r2 = x.load(std::memory_order_relaxed); // C
y.store(42, std::memory_order_relaxed); // D
```

由于标记为memory_order_relaxed的atomic操作对于memory order几乎不作保证，那么最终可能输出r1 == r2 == 42，造成这种情况可能是编译器对指令的重排，导致在线程2中D操作先于C操作完成。 Relaxed ordering比较适用于"计数器"一类的原子变量，不在意memory order的场景。

6.3.3.2.3 获取-释放次序 --memory_order_release, memory_order_acquire, memory_order_acq_rel

Acquire-release 中没有全序关系，但它供了一些同步方法。在这种序列模型下，原子 load 操作是 acquire 操作(memory_order_acquire)，原子 store 操作是release操作(memory_order_release), 原子read_modify_write操作(如fetch_add()，exchange())可以是 acquire, release 或两者皆是(memory_order_acq_rel)。同步是成对出现的，它出现在一个进行 release 操作和一个进行 acquire 操作的线程间。一个 release 操作 syncrhonized-with 一个想要读取刚才被写的值的 acquire 操作。 也就是，如果在线程1中，操作A对原子M使用memory_order_release来进行atomic store，而在另外一个线程2中，操作B对同一个原子变量M使用memory_order_acquire来进行atomic load，那么线程1在操作A之前的所有写操作(包括操作A)，都会在线程2完成操作B后是可见的。 我们看下面一个例子：

```
std::atomic<std::string*> ptr;
int data;

void producer()
{
    std::string* p  = new std::string("Hello");//A
    data = 42;//B
    ptr.store(p, std::memory_order_release);//C
}

void consumer()
{
    std::string* p2;
    while (!(p2 = ptr.load(std::memory_order_acquire)))//D
        ;
```

```
    assert(*p2 == "Hello"); //E
    assert(data == 42); //F
}

int main()
{
    std::thread t1(producer);
    std::thread t2(consumer);
    t1.join(); t2.join();
}
```

首先，我们可以直观地得出如下关系：A sequenced-before B sequenced-before C、C synchronizes-with D、D sequenced-before E sequenced-before F。利用前述happens-before推导图，不难得出A happens-before E、B happens-before F，因此，这里的E、F两处的assert永远不会fail。

6.3.3.2.3 数据依赖次序 memory_order_consume

memory_order_consume是轻量级的memory_order_acquire，是 memory_order_acquire 内存序的特例:它将同步数据限定为具有直接依赖的数据。能够用memory_order_consume的场景下就一定能够使用memory_order_acquire，引入memory_order_consume的目的是为了在一些已知的PowerPC和ARM等weakly-ordered CPUs上，对于在对有数据依赖的数据进行同步的时候不要插入额外memory barrier，因为它们本身就能保证在有数据依赖的情况下机器指令的内存顺序，少了额外的memory barrier对性能提升还是比较大的。memory_order_consume描述的是dependency-ordered-before关系。我们看上面的例子，把D中memory_order_acquire改成memory_order_consume会怎样呢？ 这个时候，由于p2和ptr有数据依赖，上面例子基本的关系对是：A sequenced-before B sequenced-before C、C dependency-ordered before D、D carries a dependency into E， E sequenced-before F。 根据关系推导，由C dependency-ordered before D && D carries a dependency into E得到C dependency-ordered before E，进一步得到C Inter-thread happens-before E，继而A sequenced-before C && C Inter-thread happens-before E得到A Inter-thread happens-before E，于是得到A Happens-before E，E永远不会assert fail。对于F，由于D、F间不存在 carries a dependency关系，那么F的assert是可能fail的。 通常情况下，我们可以通过源码的小调整实现从Release-Acquire ordering到Release-Consume ordering的转换，下面是一个

例子：

Release-Acquire ordering	Release-Consume ordering

a) atomic Guard(0);
int Payload = 0;
 	
//thread 1
b) g = Guard.load(memory_order_acquire);
if (g != 0)
c) p = Payload;
 	
//thread 2
Payload = 42;
d) Guard.store(1, memory_order_release);
	atomic Guard(nullptr);
	int Payload = 0;
	
	//thread 1
	g = Guard.load(memory_order_consume);
	if (g != nullptr)
	  p = *g;
	
	//thread 2
	Payload = 42;
	Guard.store(&Payload, memory_order_release);

7. 总结

本文通过一个无锁队列为引子，介绍了无锁编程涉及的6个技术要点，其中内存屏障是最为关键，使用什么样的memory barrier，什么时候使用memory barrier又是其中的重中之重。memory barrier不容易理解，要想正确高效地使用memory barrier就更难了，通常情况下，能不直接用memory barrier原语就不用，最好使用锁(互斥量)等互斥原语这样的隐含了memory barrier功能的原语。锁在在很长一段时间都被误解了，认为锁是慢的，由于锁的引入，给性能带来巨大的瓶颈是很常见的。但这并不意味着所有的锁都是缓慢的，当我们使用轻量级锁并控制好锁竞争的时候，锁依然有非常出色的性能表现，锁不慢，锁竞争慢。

参考资料

http://chonghw.github.io/ http://chonghw.github.io/blog/2016/08/11/memoryreorder/ http://chonghw.github.io/blog/2016/09/19/sourcecontrol/ http://chonghw.github.io/blog/2016/09/28/acquireandrelease/

http://www.wowotech.net/kernel_synchronization/Why-Memory-Barriers.html http://www.wowotech.net/kernel_synchronization/why-memory-barrier-2.html http://www.wowotech.net/kernel_synchronization/memory-barrier-1.html http://www.wowotech.net/kernel_synchronization/perfbook-memory-barrier-2.html https://kukuruku.co/post/lock-free-data-structures-introduction/ https://kukuruku.co/post/lock-free-data-structures-basics-atomicity-and-atomic-primitives/ https://kukuruku.co/post/lock-free-data-structures-the-inside-memory-management-schemes/