Linux内核同步机制之（三）：memory barrier

作者：linuxer 发布于：2014-11-14 19:20 分类：内核同步机制

一、前言

我记得以前上学的时候大家经常说的一个词汇叫做所见即所得，有些编程工具是所见即所得的，给程序员带来极大的方便。对于一个c程序员，我们的编写的代码能所见即所得吗？我们看到的c程序的逻辑是否就是最后CPU运行的结果呢？很遗憾，不是，我们的“所见”和最后的执行结果隔着：

1、编译器

2、CPU取指执行

编译器将符合人类思考的逻辑（c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c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，作为c程序员，必须有能力了解编译器的行为，并在通过内嵌在c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。

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（这种CPU在现实世界中还存在吗？），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，汇编指令的逻辑被严格的被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（看，人还是最不可以替代的）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。

二、不使用memory barrier会导致问题的场景

1、编译器的优化

我们先看下面的一个例子：

```
preempt_disable（）

临界区

preempt_enable
```

有些共享资源可以通过禁止任务抢占来进行保护，因此临界区代码被preempt_disable和preempt_enable给保护起来。其实，我们知道所谓的preempt enable和disable其实就是对当前进程的struct thread_info中的preempt_count进行加一和减一的操作。具体的代码如下：

```
#define preempt_disable() \
do { \
    preempt_count_inc(); \
    barrier(); \
} while (0)
```

linux kernel中的定义和我们的想像一样，除了barrier这个优化屏障。barrier就象是c代码中的一个栅栏，将代码逻辑分成两段，barrier之前的代码和barrier之后的代码在经过编译器编译后顺序不能乱掉。也就是说，barrier之后的c代码对应的汇编，不能跑到barrier之前去，反之亦然。之所以这么做是因为在我们这个场景中，如果编译为了榨取CPU的performace而对汇编指令进行重排，那么临界区的代码就有可能位于preempt_count_inc之外，从而起不到保护作用。

现在，我们知道了增加barrier的作用，问题来了，barrier是否够呢？对于multi-core的系统，只有当该task被调度到该CPU上执行的时候，该CPU才会访问该task的preempt count，因此对于preempt enable和disable而言，不存在多个CPU同时访问的场景。但是，即便这样，如果CPU是乱序执行（out-of-order excution）的呢？其实，我们也不用担心，正如前面叙述的，preempt count这个memory实际上是不存在多个cpu同时访问的情况，因此，它实际上会本cpu的进程上下文和中断上下文访问。能终止当前thread执行preempt_disable的只有中断。为了方便描述，我们给代码编址，如下：

地址	该地址的汇编指令	CPU的执行顺序
a	preempt_disable（）	临界区指令1
a+4	临界区指令1	preempt_disable（）
a+8	临界区指令2	临界区指令2
a+12	preempt_enable	preempt_enable

当发生中断的时候，硬件会获取当前PC值，并精确的得到了发生指令的地址。有两种情况：

（1）在地址a发生中断。对于out-of-order的CPU，临界区指令1已经执行完毕，preempt_disable正在pipeline中等待执行。由于是在a地址发生中断，也就是preempt_disable地址上发生中断，对于硬件而言，它会保证a地址之前（包括a地址）的指令都被执行完毕，并且a地址之后的指令都没有执行。因此，在这种情况下，临界区指令1的执行结果被抛弃掉，因此，实际临界区指令不会先于preempt_disable执行

（2）在地址a + 4发生中断。这时候，虽然发生中断的那一刻的地址上的指令（临界区指令1）已经执行完毕了，但是硬件会保证地址a + 4之前的所有的指令都执行完毕，因此，实际上CPU会执行完preempt_disable，然后跳转的中断异常向量执行。

上面描述的是优化屏障在内存中的变量的应用，下面我们看看硬件寄存器的场景。一般而言，串口的驱动都会包括控制台部分的代码，例如：

```
static struct console xx_serial_console = {
……
    .write        = xx_serial_console_write,
……
};
```

如果系统enable了串口控制台，那么当你的驱动调用printk的时候，实际上最终是通过console的write函数输出到了串口控制台。而这个console write的函数可能会包含下面的代码：

```
do {
    获取TX FIFO状态寄存器
    barrier();
} while (TX FIFO没有ready);
写TX FIFO寄存器;
```

对于某些CPU archtecture而言（至少ARM是这样的），外设硬件的IO地址也被映射到了一段内存地址空间，对编译器而言，它并不知道这些地址空间是属于外设的。因此，对于上面的代码，如果没有barrier的话，获取TX FIFO状态寄存器的指令可能和写TX FIFO寄存器指令进行重新排序，在这种情况下，程序逻辑就不对了，因为我们必须要保证TX FIFO ready的情况下才能写TX FIFO寄存器。

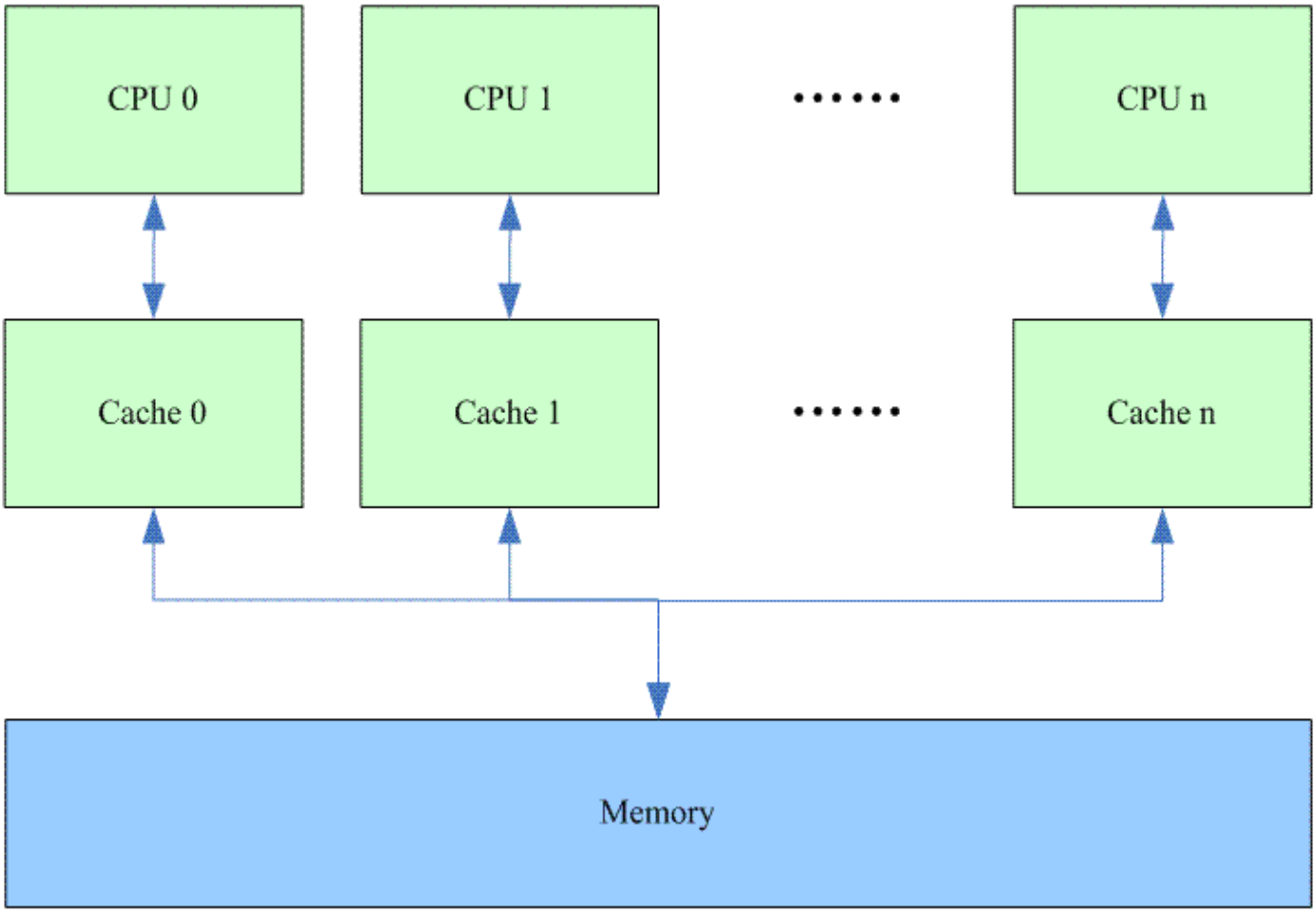
对于multi core的情况，上面的代码逻辑也是OK的，因为在调用console write函数的时候，要获取一个console semaphore，确保了只有一个thread进入，因此，console write的代码不会在多个CPU上并发。和preempt count的例子一样，我们可以问同样的问题，如果CPU是乱序执行（out-of-order excution）的呢？barrier只是保证compiler输出的汇编指令的顺序是OK的，不能确保CPU执行时候的乱序。 对这个问题的回答来自ARM architecture的内存访问模型：对于program order是A1-->A2的情况（A1和A2都是对Device或是Strongly-ordered的memory进行访问的指令），ARM保证A1也是先于A2执行的。因此，在这样的场景下，使用barrier足够了。 对于X86也是类似的，虽然它没有对IO space采样memory mapping的方式，但是，X86的所有操作IO端口的指令都是被顺执行的，不需要考虑memory access order。

2、cpu architecture和cache的组织

注：本章节的内容来自对Paul E. McKenney的Why memory barriers文档理解，更细致的内容可以参考该文档。这个章节有些晦涩，需要一些耐心。作为一个c程序员，你可能会抱怨，为何设计CPU的硬件工程师不能屏蔽掉memory barrier的内容，让c程序员关注在自己需要关注的程序逻辑上呢？本章可以展开叙述，或许能解决一些疑问。

（1）基本概念

在The Memory Hierarchy文档中，我们已经了解了关于cache一些基础的知识，一些基础的内容，这里就不再重复了。我们假设一个多核系统中的cache如下：



我们先了解一下各个cpu cache line状态的迁移过程：

（a）我们假设在有一个memory中的变量为多个CPU共享，那么刚开始的时候，所有的CPU的本地cache中都没有该变量的副本，所有的cacheline都是invalid状态。

（b）因此当cpu 0 读取该变量的时候发生cache miss（更具体的说叫做cold miss或者warmup miss）。当该值从memory中加载到chache 0中的cache line之后，该cache line的状态被设定为shared，而其他的cache都是Invalid。

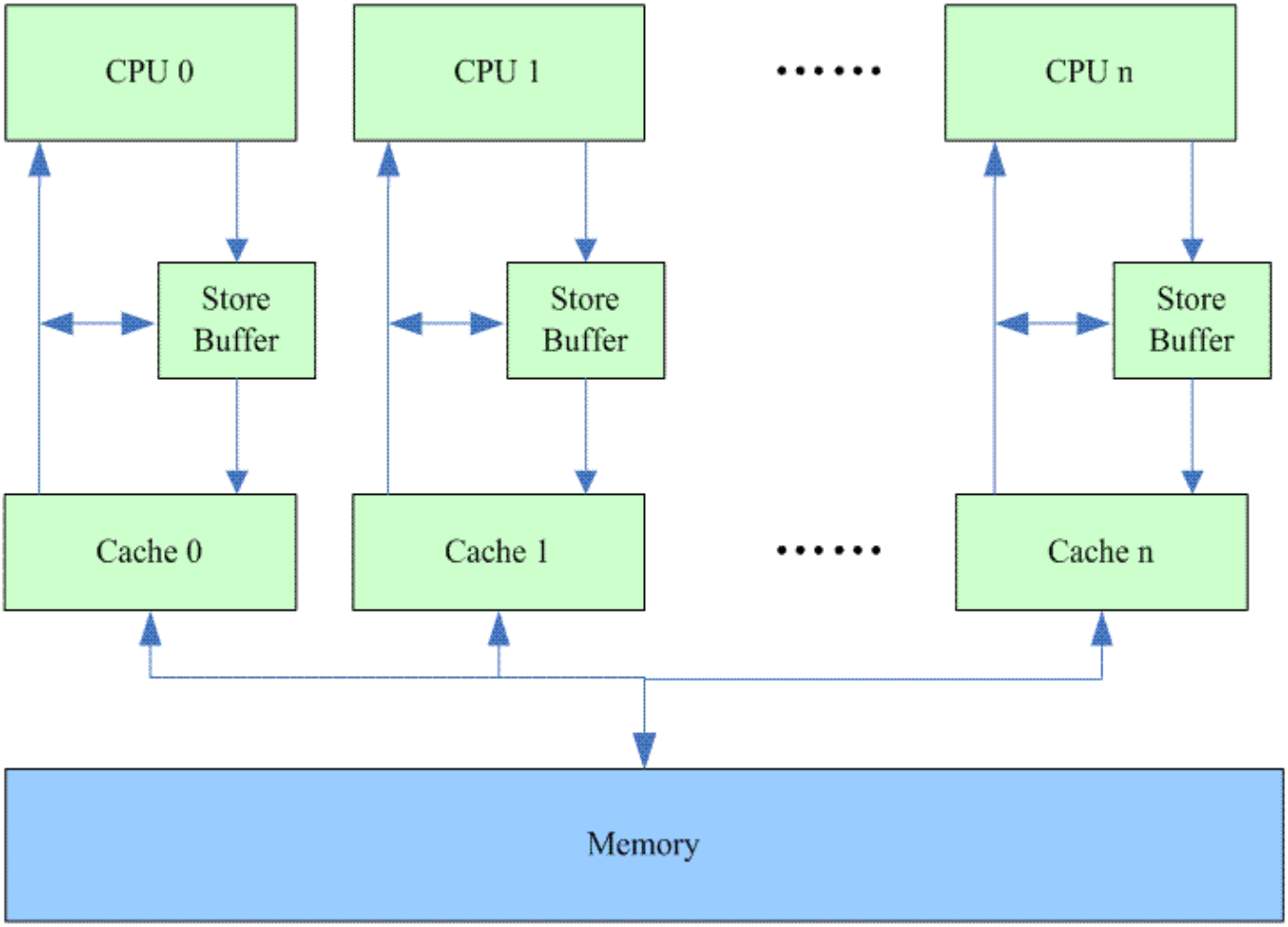
（c）当cpu 1 读取该变量的时候，chache 1中的对应的cache line也变成shared状态。其实shared状态就是表示共享变量在一个或者多个cpu的cache中有副本存在。既然是被多个cache所共享，那么其中一个CPU就不能武断修改自己的cache而不通知其他CPU的cache，否则会有一致性问题。

（d）总是read多没劲，我们让CPU n对共享变量来一个load and store的操作。这时候，CPU n发送一个read invalidate命令，加载了Cache n的cache line，并将状态设定为exclusive，同时将所有其他CPU的cache对应的该共享变量的cacheline设定为invalid状态。正因为如此，CPU n实际上是独占了变量对应的cacheline（其他CPU的cacheline都是invalid了，系统中就这么一个副本），就算是写该变量，也不需要通知其他的CPU。CPU随后的写操作将cacheline设定为modified状态，表示cache中的数据已经dirty，和memory中的不一致了。modified状态和exclusive状态都是独占该cacheline，但是modified状态下，cacheline的数据是dirty的，而exclusive状态下，cacheline中的数据和memory中的数据是一致的。当该cacheline被替换出cache的时候，modified状态的cacheline需要write back到memory中，而exclusive状态不需要。

（e）在cacheline没有被替换出CPU n的cache之前，CPU 0再次读该共享变量，这时候会怎么样呢？当然是cache miss了（因为之前由于CPU n写的动作而导致其他cpu的cache line变成了invalid，这种cache miss叫做communiction miss）。此外，由于CPU n的cache line是modified状态，它必须响应这个读得操作（memory中是dirty的）。因此，CPU 0的cacheline变成share状态（在此之前，CPU n的cache line应该会发生write back动作，从而导致其cacheline也是shared状态）。当然，也可能是CPU n的cache line不发生write back动作而是变成invalid状态，CPU 0的cacheline变成modified状态，这和具体的硬件设计相关。

（2）Store buffer

我们考虑另外一个场景：在上一节中step e中的操作变成CPU 0对共享变量进行写的操作。这时候，写的性能变得非常的差，因为CPU 0必须要等到CPU n上的cacheline 数据传递到其cacheline之后，才能进行写的操作（CPU n上的cacheline 变成invalid状态，CPU 0则切换成exclusive状态，为后续的写动作做准备）。而从一个CPU的cacheline传递数据到另外一个CPU的cacheline是非常消耗时间的，而这时候，CPU 0的写的动作只是hold住，直到cacheline的数据完成传递。而实际上，这样的等待是没有意义的，因此，这时候cacheline的数据仍然会被覆盖掉。为了解决这个问题，多核系统中的cache修改如下：



这样，问题解决了，写操作不必等到cacheline被加载，而是直接写到store buffer中然后欢快的去干其他的活。在CPU n的cacheline把数据传递到其cache 0的cacheline之后，硬件将store buffer中的内容写入cacheline。

虽然性能问题解决了，但是逻辑错误也随之引入，我们可以看下面的例子：

我们假设a和b是共享变量，初始值都是0，可以被cpu0和cpu1访问。cpu 0的cache中保存了b的值（exclusive状态），没有a的值，而cpu 1的cache中保存了a的值，没有b的值，cpu 0执行的汇编代码是（用的是ARM汇编，没有办法，其他的都不是那么熟悉）：

```
ldr     r2, [pc, #28]   ------------------------- 取变量a的地址
ldr     r4, [pc, #20]   ------------------------- 取变量b的地址
mov     r3, #1
```

```
str     r3, [r2]          -------------------------a=1
str     r3, [r4]          -------------------------b=1
```

CPU 1执行的代码是：

```
            ldr     r2, [pc, #28]   -------------------------- 取变量a的地址

            ldr     r3, [pc, #20]  -------------------------- 取变量b的地址
start:      ldr     r3, [r3]          -------------------------- 取变量b的值
            cmp     r3, #0          ------------------------ b的值是否等于0?
            beq     start           ------------------------ 等于0的话跳转到start

            ldr     r2, [r2]          -------------------------- 取变量a的值
```

当cpu 1执行到--取变量a的值--这条指令的时候，b已经是被cpu0修改为1了，这也就是说a = 1这个代码已经执行了，因此，从汇编代码的逻辑来看，这时候a值应该是确定的1。然而并非如此，cpu 0和cpu 1执行的指令和动作描述如下：

cpu 0执行的指令	cpu 0动作描述	cpu 1执行的指令	cpu 1动作描述
str r3, [r2] （a=1）	1、发生cache miss 2、将1保存在store buffer中 3、发送read invalidate命令，试图从cpu 1的cacheline中获取数据，并invalidate其cache line 注：这里无需等待response，立刻执行下一条指令	ldr r3, [r3] （获取b的值）	1、发生cache miss 2、发送read命令，试图加载b对应的cacheline 注：这里cpu必须等待read response，下面的指令依赖于这个读取的结果
str r3, [r4] （b=1）	1、cache hit 2、cacheline中的值被修改为1，状态变成modified		
	响应cpu 1的read命令，发送read response（b = 1）给CPU 0。write back，将状态设定为shared		
		cmp r3, #0	1、cpu 1收到来自cpu 0的read response，加载b对应的cacheline，状态为shared 2、b等于1，因此不必跳转到start执行
		ldr r2, [r2] （获取a的值）	1、cache hit 2、获取了a的旧值，也就是0
			响应CPU 0的read invalid命令，将a对应的cacheline设为invalid状态，发送read response和invalidate ack。但是已经酿成大错了。
	收到来自cpu 1的响应，将store buffer中的1写入cache line。		

对于硬件，CPU不清楚具体的代码逻辑，它不可能直接帮助软件工程师，只是提供一些memory barrier的指令，让软件工程师告诉CPU他想要的内存访问逻辑顺序。这时候，cpu 0的代码修改如下：

```
ldr     r2, [pc, #28]   ------------------------- 取变量a的地址
ldr     r4, [pc, #20]   ------------------------- 取变量b的地址
mov     r3, #1
str     r3, [r2]          -------------------------a=1

确保清空store buffer的memory barrier instruction
str     r3, [r4]          -------------------------b=1
```

这种情况下，cpu 0和cpu 1执行的指令和动作描述如下：

cpu 0执行的指令	cpu 0动作描述	cpu 1执行的指令	cpu 1动作描述
str r3, [r2] （a=1）	1、发生cache miss 2、将1保存在store buffer中 3、发送read invalidate命令，试图从cpu 1的cacheline中获取数据，并invalidate其cache line 注：这里无需等待response，立刻执行下一条指令	ldr r3, [r3] （获取b的值）	1、发生cache miss 2、发送read命令，试图加载b对应的cacheline 注：这里cpu必须等待read response，下面的指令依赖于这个读取的结果
memory barrier instruction	CPU收到memory barrier指令，知道软件要控制访问顺序，因此不会执行下一条str指令，要等到收到read response和invalidate ack后，将store buffer中所有数据写到cacheline之后才会执行后续的store指令		
		cmp r3, #0 beq start	1、cpu 1收到来自cpu 0的read response，加载b对应的cacheline，状态为shared 2、b等于0，跳转到start执行
			响应CPU 0的read invalid命令，将a对应的cacheline设为invalid状态，发送read response和invalidate ack。
	收到来自cpu 1的响应，将store buffer中的1写入cache line。		
str r3, [r4] （b=1）	1、cache hit，但是cacheline状态是shared，需要发送invalidate到cpu 1 2、将1保存在store buffer中 注：这里无需等待invalidate ack，立刻执行下一条指令		
…	…	…	…

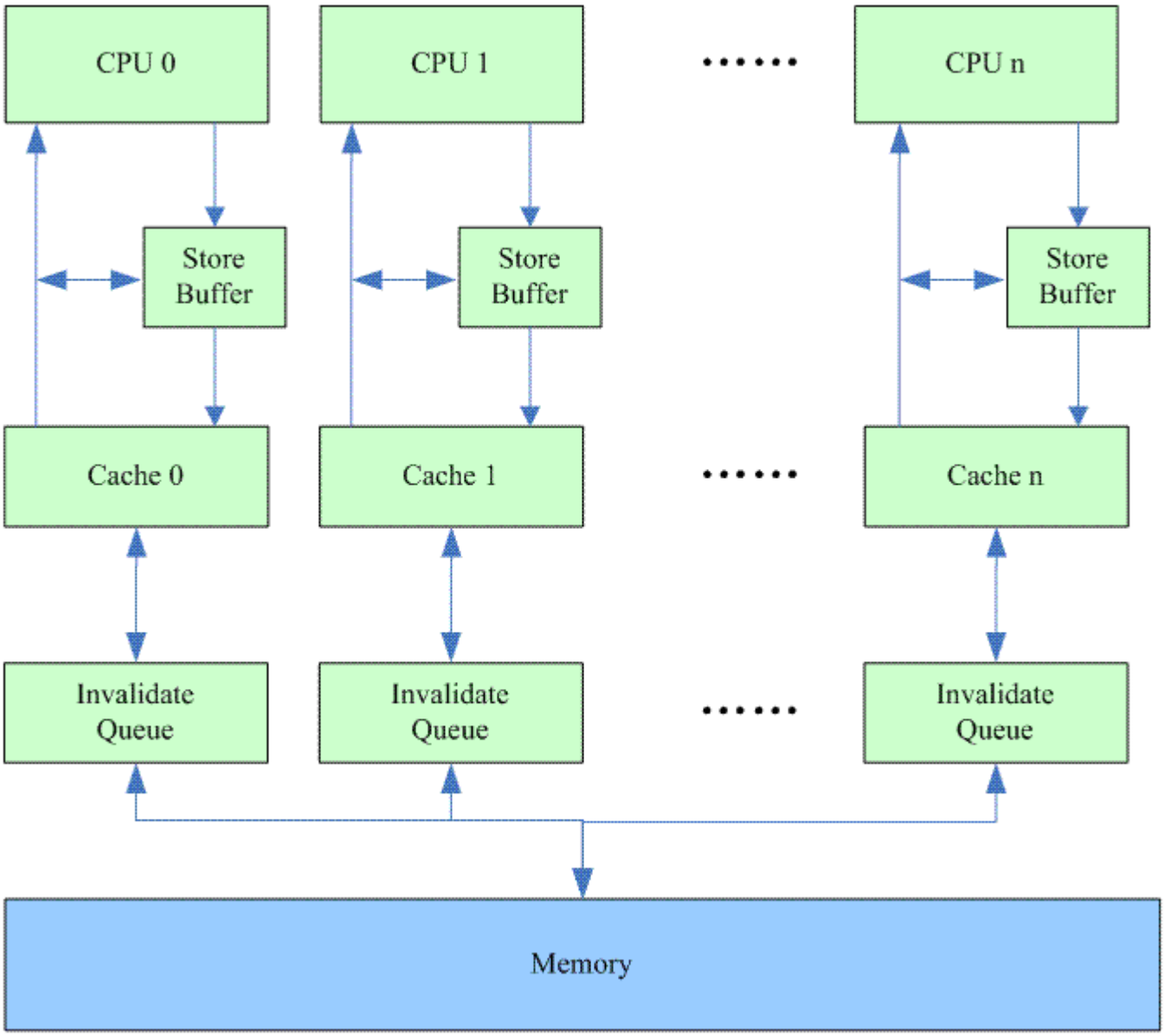
由于增加了memory barrier，保证了a、b这两个变量的访问顺序，从而保证了程序逻辑。

（3）Invalidate Queue

我们先回忆一下为何出现了stroe buffer：为了加快cache miss状态下写的性能，硬件提供了store buffer，以便让CPU先写入，从而不必等待invalidate ack（这些交互是为了保证各个cpu的cache的一致性）。然而，store buffer的size比较小，不需要特别多的store命令（假设每次都是cache miss）就可以将store buffer填满，这时候，没有空间写了，因此CPU也只能是等待invalidate ack了，这个状态和memory barrier指令的效果是一样的。

怎么解决这个问题？CPU设计的硬件工程师对性能的追求是不会停歇的。我们首先看看invalidate ack为何如此之慢呢？这主要是因为cpu在收到invalidate命令后，要对cacheline执行invalidate命令，确保该cacheline的确是invalid状态后，才会发送ack。如果cache正忙于其他工作，当然不能立刻执行invalidate命令，也就无法会ack。

怎么破？CPU设计的硬件工程师提供了下面的方法：



Invalidate Queue这个HW block从名字就可以看出来是保存invalidate请求的队列。其他CPU发送到本CPU的invalidate命令会保存于此，这时候，并不需要等到实际对cacheline的invalidate操作完成，CPU就可以回invalidate ack了。

同store buffer一样，虽然性能问题解决了，但是对memory的访问顺序导致的逻辑错误也随之引入，我们可以看下面的例子（和store buffer中的例子类似）：

我们假设a和b是共享变量，初始值都是0，可以被cpu0和cpu1访问。cpu 0的cache中保存了b的值（exclusive状态），而CPU 1和CPU 0的cache中都保存了a的值，状态是shared。cpu 0执行的汇编代码是：

```
ldr     r2, [pc, #28]   ------------------------- 取变量a的地址
ldr     r4, [pc, #20]   ------------------------- 取变量b的地址
mov     r3, #1
str     r3, [r2]          -------------------------a=1

确保清空store buffer的memory barrier instruction
str     r3, [r4]          -------------------------b=1
```

CPU 1执行的代码是：

```
            ldr     r2, [pc, #28]   ------------------------- 取变量a的地址

            ldr     r3, [pc, #20]  ------------------------- 取变量b的地址
start:      ldr     r3, [r3]          ------------------------- 取变量b的值
            cmp     r3, #0          ----------------------- b的值是否等于0?
            beq     start           ----------------------- 等于0的话跳转到start

            ldr     r2, [r2]          ------------------------- 取变量a的值
```

这种情况下，cpu 0和cpu 1执行的指令和动作描述如下：

cpu 0执行的指令	cpu 0动作描述	cpu 1执行的指令	cpu 1动作描述
str r3, [r2] （a=1）	1、a值在CPU 0的cache中状态是shared，是read only的，因此，需要通知其他的CPU 2、将1保存在store buffer中 3、发送invalidate命令，试图invalidate CPU 1中a对应的cache line 注：这里无需等待response，立刻执行下一条指令	ldr r3, [r3] （获取b的值）	1、发生cache miss 2、发送read命令，试图加载b对应的cacheline 注：这里cpu必须等待read response，下面的指令依赖于这个读取的结果
			收到来自CPU 0的invalidate命令，放入invalidate queue，立刻回ack。
memory barrier instruction	CPU收到memory barrier指令，知道软件要控制访问顺序，因此不会执行下一条str指令，要等到收到invalidate ack后，将store buffer中所有数据写到cacheline之后才会执行后续的store指令		
	收到invalidate ack后，将store buffer中的1写入cache line。OK，可以继续执行下一条指令了		
str r3, [r4] （b=1）	1、cache hit 2、cacheline中的值被修改为1，状态变成modified		
	收到CPU 1发送来的read命令，将b值		

	（等于1）放入read response中，回送给CPU 1，write back并将状态修改为shared。		
			收到response（b = 1），并加载cacheline，状态是shared
		cmp r3, #0	b等于1，不会执行beq指令，而是执行下一条指令
		ldr r2, [r2] （获取a的值）	1、cache hit （还没有执行invalidate动作，命令还在invalidate queue中呢） 2、获取了a的旧值，也就是0
			对a对应的cacheline执行invalidate 命令，但是，已经晚了

可怕的memory misorder问题又来了，都是由于引入了invalidate queue引起，看来我们还需要一个memory barrier的指令，我们将程序修改如下：

```
        ldr     r2, [pc, #28]   -------------------------- 取变量a的地址

        ldr     r3, [pc, #20]  -------------------------- 取变量b的地址
start:  ldr     r3, [r3]        -------------------------- 取变量b的值
        cmp     r3, #0          ----------------------- b的值是否等于0?
        beq     start           ----------------------- 等于0的话跳转到start

确保清空invalidate queue的memory barrier instruction

        ldr     r2, [r2]        -------------------------- 取变量a的值
```

这种情况下，cpu 0和cpu 1执行的指令和动作描述如下：

cpu 0执行的指令	cpu 0动作描述	cpu 1执行的指令	cpu 1动作描述
str r3, [r2] （a=1）	1、a值在CPU 0的cache中状态是shared，是read only的，因此，需要通知其他的CPU 2、将1保存在store buffer中 3、发送invalidate命令，试图invalidate CPU 1中a对应的cache line 注：这里无需等待response，立刻执行下一条指令	ldr r3, [r3] （获取b的值）	1、发生cache miss 2、发送read命令，试图加载b对应的cacheline 注：这里cpu必须等待read response，下面的指令依赖于这个读取的结果
			收到来自CPU 0的invalidate命令，放入invalidate queue，立刻回ack。
memory barrier instruction	CPU收到memory barrier指令，知道软件要控制访问顺序，因此不会执行下一条str指令，要等到收到invalidate ack后，将store buffer中所有数据写到cacheline之后才会执行后续的store指令		
	收到invalidate ack后，将store buffer中的1写入cache line。OK，可以继续执行下一条指令了		

str r3, [r4] （b=1）	1、cache hit 2、cacheline中的值被修改为1，状态变成modified		
	收到CPU 1发送来的read命令，将b值（等于1）放入read response中，回送给CPU 1，write back并将状态修改为shared。		
			收到response（b = 1），并加载cacheline，状态是shared
		cmp r3, #0	b等于1，不会执行beq指令，而是执行下一条指令
		memory barrier instruction	CPU收到memory barrier指令，知道软件要控制访问顺序，因此不会执行下一条ldr指令，要等到执行完invalidate queue中的所有的invalidate命令之后才会执行下一个ldr指令
		ldr r2, [r2] （获取a的值）	1、cache miss 2、发送read命令，从CPU 0那里加载新的a值

由于增加了memory barrier，保证了a、b这两个变量的访问顺序，从而保证了程序逻辑。

三、linux kernel的API

linux kernel的memory barrier相关的API列表如下：

接口名称	作用
barrier()	优化屏障，阻止编译器为了进行性能优化而进行的memory access reorder
mb()	内存屏障（包括读和写），用于SMP和UP
rmb()	读内存屏障，用于SMP和UP
wmb()	写内存屏障，用于SMP和UP
smp_mb()	用于SMP场合的内存屏障，对于UP不存在memory order的问题（对汇编指令），因此，在UP上就是一个优化屏障，确保汇编和c代码的memory order是一致的
smp_rmb()	用于SMP场合的读内存屏障
smp_wmb()	用于SMP场合的写内存屏障

barrier()这个接口和编译器有关，对于gcc而言，其代码如下：

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

这里的__volatile__主要是用来防止编译器优化的。而这里的优化是针对代码块而言的，使用嵌入式汇编的代码分成三块：

1、嵌入式汇编之前的c代码块

2、嵌入式汇编代码块

3、嵌入式汇编之后的c代码块

这里__volatile__就是告诉编译器：不要因为性能优化而将这些代码重排，我需要清清爽爽的保持这三块代码块的顺序（代码块内部是否重排不是这里的__volatile__管辖范围了）。

barrier中的嵌入式汇编中的clobber list没有描述汇编代码对寄存器的修改情况，只是有一个memory的标记。我们知道，clober list是gcc和gas的接口，用于gas通知gcc它对寄存器和memory的修改情况。因此，这里的memory就是告知gcc，在汇编代码中，我修改了memory中的内容，嵌入式汇编之前的c代码块和嵌入式汇编之后的c代码块看到的memory是不一样的，对memory的访问不能依赖于嵌入式汇编之前的c代码块中寄存器的内容，需要重新加载。

优化屏障是和编译器相关的，而内存屏障是和CPU architecture相关的，当然，我们选择ARM为例来描述内存屏障。

四、内存屏障在ARM中的实现

TODO