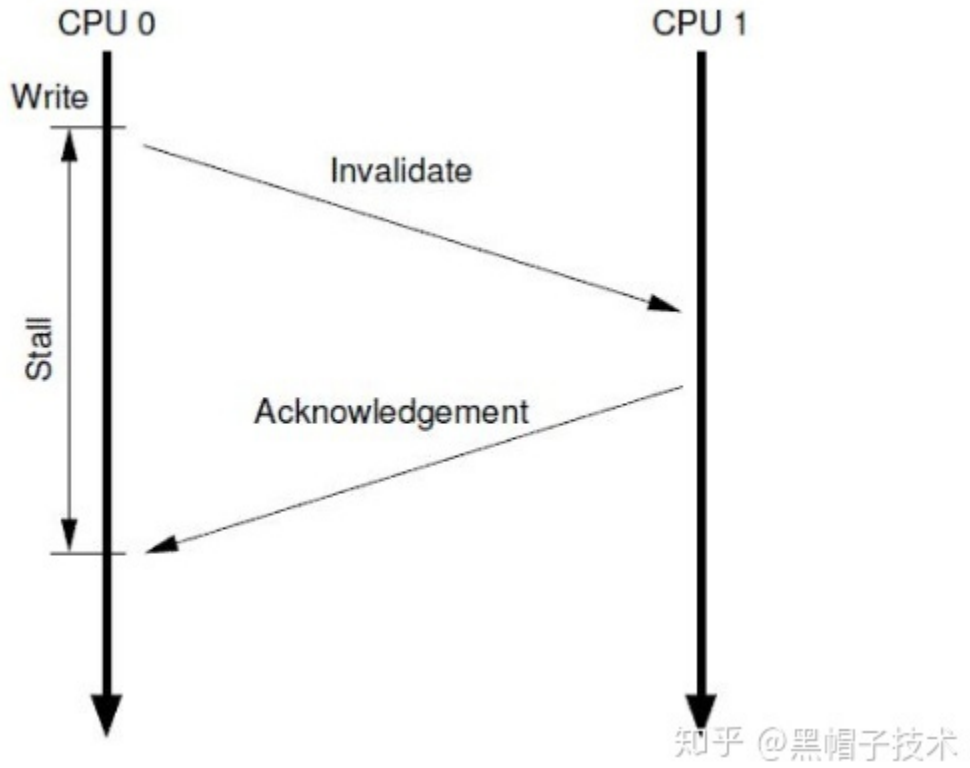
zhuanlan.zhihu.com /p/125737864

内存屏障（Memory Barrier）究竟是个什么鬼？

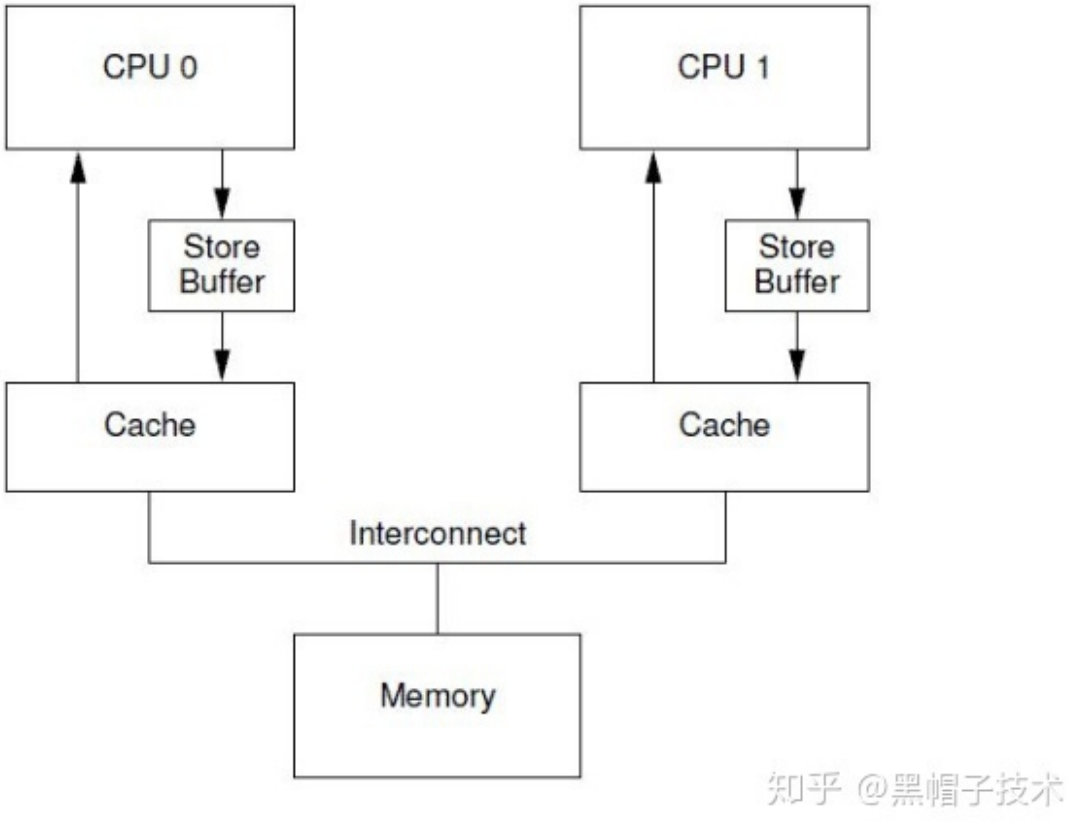
在开始阅读之前我们假设读者已经掌握了缓存一致性协议的MESI相关知识。如果没有建议阅读 带你了解缓存一致性协议 MESI

问题的产生



如上图 CPU 0 执行了一次写操作，但是此时 CPU 0 的 local cache 中没有这个数据。于是 CPU 0 发送了一个 Invalidate 消息，其他所有的 CPU 在收到这个 Invalidate 消息之后，需要将自己 CPU local cache 中的该数据从 cache 中清除，并且发送消息 acknowledge 告知 CPU 0。CPU 0 在收到所有 CPU 发送的 ack 消息后会将数据写入到自己的 local cache 中。**这里就产生了性能问题：当 CPU 0 在等待其他 CPU 的 ack 消息时是处于停滞的（stall）状态，大部分的时间都是在等待消息。**为了提高性能就引入的 Store Buffer。

Store Buffer



store buffer 的目的是让 CPU 不再操作之前进行漫长的等待时间，而是将数据先写入到 store buffer 中，CPU 无需等待可以继续执行其他指令，等到 CPU 收到了 ack 消息后，再从 store buffer 中将数据写入到 local cache 中。有了 store buffer 之后性能提高了许多，但常言道：“有一利必有一弊。”store buffer 虽然提高了性能但是却引入了新的问题。

```
a = 0 , b = 0;
a = 1;
b = a + 1;
assert(b == 2);
```

假设变量 a 在 CPU 1 的 cache line 中 ， 变量 b 在 CPU 0 的 cache line 中。上述代码的执行序列如下：

1. CPU 0 执行 a = 1。

2. CPU 0 local cache 中没有 a ，发生 cache miss 。

3. CPU 0 发送 read invalidate 消息获取 a ，同时让其他 CPU local cache 中的 a 被清除。

4. CPU 0 把需要写入 a 的值 1 放入了 store buffer 。

5. CPU 1 收到了 read invalidate 消息，回应了 read response 和 acknowledge 消息，把自己 local cache 中的 a 清除了。

6. CPU 0 执行 b = a + 1 。

7. CPU 0 收到了 read response 消息得到了 a 的值是 0 。

8. CPU 0 从 cache line 中读取了 a 值为 0 。

9. CPU 0 执行 a + 1 , 并写入 b ，b 被 CPU 0 独占所以直接写入 cache line ， 这时候 b 的值为 1。

10. CPU 0 将 store buffer 中 a 的值写入到 cache line ， a 变为 1。

11. CPU 0 执行 assert(b == 2) , 程序报错。

导致这个问题是因为 CPU 对内存进行操作的时候，顺序和程序代码指令顺序不一致。在写操作执行之前就先执行了读操作。另一个原因是在同一个 CPU 中同一个数据存在不一致的情况 ， 在 store buffer 中是最新的数据， 在 cache line 中是旧的数据。为了解决在同一个 CPU 的 store buffer 和 cache 之间数据不一致的问题，引入了 Store Forwarding。store forwarding 就是当 CPU 执行读操作时，会从 store buffer 和 cache 中读取数据， 如果 store buffer 中有数据会使用 store buffer 中的数据，这样就解决了同一个 CPU 中数据不一致的问题。但是由于 Memory Ordering 引起的问题还没有解决。

内存操作顺序

Memory Ordering

```
a = 0 , b = 0;
void fun1() {
  a = 1;
  b = 1;
}

void fun2() {
  while (b == 0) continue;
  assert(a == 1);
}
```

假设 CPU 0 执行 fun1() , CPU 1 执行 fun2() , a 变量在 CPU 1 cache 中 ， b 变量在 CPU 0 cache 中。 上述代码的执行序列如下：

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，继续执行。
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。
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。

产生问题的原因是 CPU 0 对 a 的写操作还没有执行完，但是 CPU 1 对 a 的读操作已经执行了。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。

写内存屏障

Store Memory Barrier

```
a = 0 , b = 0;
void fun1() {
  a = 1;
  smp_mb();
  b = 1;
}

void fun2() {
  while (b == 0) continue;
  assert(a == 1);
}
```

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）。smp_mb() 操作主要是为了让数据在local cache中的操作顺序是符合program order的顺序的，为了达到这个目标有两种方法：方法一就是让CPU stall，直到完成了清空了store buffer（也就是把store buffer中的数据写入cacheline了）。方法二是让CPU可以继续运行，不过需要在store buffer中做些文章，也就是要记录store buffer中数据的顺序，在将store buffer的数据更新到cacheline的操作中，严格按照顺序执行，即便是后来的store buffer数据对应的cacheline已经ready，也不能执行操作，要等前面的store buffer值写到cacheline之后才操作。增加smp_mb() 之后，操作顺序如下：

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 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CPU0并没有直接操作将新的值1写入cache line，取而代之是b的新值”1“被写入store buffer，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. 完成了bus transaction之后，CPU 1可以load b到寄存器中了（local cacheline中已经有b值了），当然，这时候b仍然等于0，因此循环不断的loop。虽然b值在CPU 0上已经赋值等于1，但是那个新值被安全的隐藏在CPU 0的store buffer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cacheline的状态是shared。

11. CPU 0发送invalidate消息，请求b数据的独占权。

12. CPU 1收到invalidate消息，清空自己的b cacheline，并回送acknowledgement给CPU 0。

13. CPU 1继续执行while (b == 0)，由于b不在自己的local cache中，因此 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cacheline修改成exclusive状态，这时候，CPU 0终于可以将b的新值1写入cacheline。

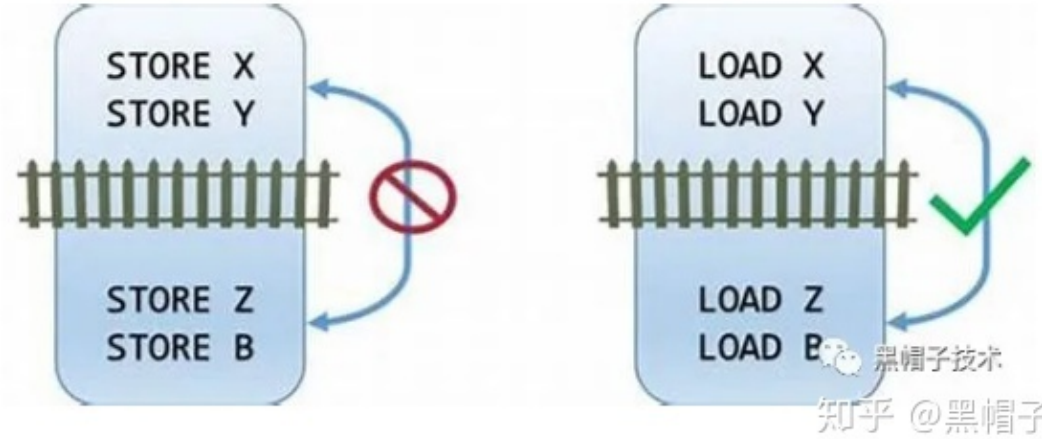
15. CPU 0收到read消息，将b的新值1回送给CPU 1，同时将其local cache中b对应的cacheline状态修改为shared。

16. CPU 1获取来自CPU 0的b的新值，将其放入cacheline中。

17. 由于b值等于1了，因此CPU 1跳出while (b == 0)的循环，继续执行。

18. CPU 1执行assert(a == 1)，不过这时候a值没有在自己的cacheline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。

通过上面的描述，我们可以看到，一个直观上很简单的给a变量赋值的操作，都需要那么长的执行过程，而且每一步都需要芯片参与，最终完成整个复杂的赋值操作过程。



上述这个例子展示了 write memory barrier，简单来说在屏障之后的写操作必须等待屏障之前的写操作完成才可以执行，读操作则不受该屏障的影响。

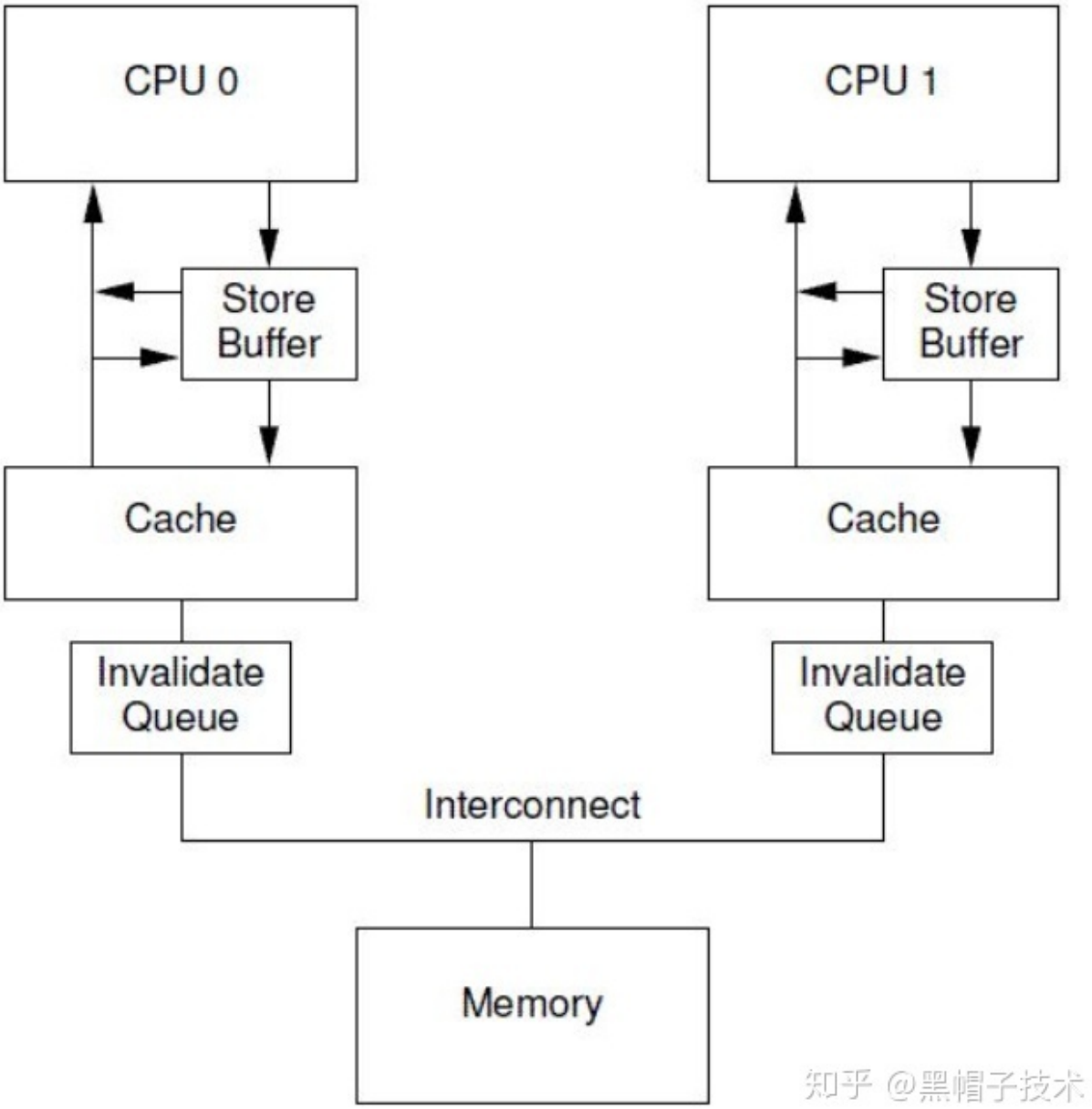
顺序写操作导致了 CPU 的停顿

Store Sequences Result in Unnecessary Stalls

按照矛盾的角度来看解决了一个问题之后伴随着又产生了一个新的问题：每个cpu的store buffer不能实现的太大，其entry的数目不会太多。当cpu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只能又进入等待状态，直到cache line完成invalidate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才可以继续执行。这种状况也可能发生在调用了memory barrier指令之后，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invalidate完成，因此不管是否cache miss，这些store都必须进入store buffer。为了解决这个问题引入了 invalidate queues 可以缓解这个状况。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cacheline，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。

然而，CPU其实不需要完成invalidate操作就可以回送acknowledge消息，这样，就不会阻止发生invalidate请求的那个CPU进入无聊的等待状态。CPU可以buffer这些invalidate message（放入Invalidate Queues），然后直接回应acknowledge，表示自己已经收到请求，随后会慢慢处理。当然，再慢也要有一个度，例如对a变量cacheline的invalidate处理必须在该CPU发送任何关于a变量对应cacheline的操作到bus之前完成。



有了Invalidate Queue的CPU，在收到invalidate消息的时候首先把它放入Invalidate Queue，同时立刻回送acknowledge 消息，无需等到该cacheline被真正invalidate之后再回应。当然，如果本CPU想要针对某个cacheline向总线发送invalidate消息的时候，那么CPU必须首先去Invalidate Queue中看看是否有相关的cacheline，如果有，那么不能立刻发送，需要等到Invalidate Queue中的cacheline被处理完之后再发送。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。

读内存屏障

Load Memory Barrier

```
a = 0 , b = 0;
void fun1() {
```

```
  a = 1;
  smp_mb();
  b = 1;
}

void fun2() {
   while (b == 0) continue;
   assert(a == 1);
}
```

假设 a 存在于 CPU 0 和 CPU 1 的 local cache 中，b 存在于 CPU 0 中。CPU 0 执行 fun1() , CPU 1 执行 fun2() 。操作序列如下：

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cacheline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assert 失败。

10. 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。

很明显，在上面场景中，加速 ack 导致fun1()中的memory barrier失效了，因此，这时候对 ack 已经没有意义了，毕竟程序逻辑都错了。怎么办？其实我们可以让memory barrier指令和Invalidate Queue进行交互来保证确定的memory order。具体做法是这样的：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，这些被标注的项次被称为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 fun2() 增加内存屏障的操作，具体如下：

```
a = 0 , b = 0;
void fun1() {
  a = 1;
  smp_mb();
  b = 1;
```

```
}

void fun2() {
  while (b == 0) continue;
  smp_rmb();
  assert(a == 1);
 }
```

当 CPU 1 执行完 while(b == 0) continue; 之后， 它必须等待 Invalidate Queues 中的 Invalidate 变量 a 的消息被处理完，将 a 从 CPU 1 local cache 中清除掉。然后才能执行 assert(a == 1)。CPU 1 在读取 a 时发生 cache miss ，然后发送一个 read 消息读取 a ，CPU 0 会回应一个 read response 将 a 的值发送给 CPU 1。

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。