现在考虑在一个多处理机中发生的情况。在图8-10中我们看到了最坏情况的时序，其中存储器字1000，被用作一个初始化为0的锁。第1步，CPU 1读出该字得到一个0。第2步，在CPU 1有机会把该字写为1之前，CPU 2进入，并且也读出该字为0。第3步，CPU 1把1写入该字。第4步，CPU 2也把1写入该字。两个CPU都由TSL指令得到0，所以两者都对临界区进行访问，并且互斥失败。

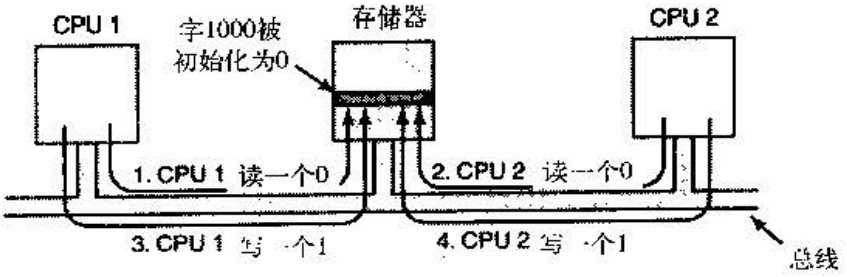

图8-10 如果不能锁住总线，TSL指令会失效。这里的四步解释了失效情况

为了阻止这种情况的发生，TSL指令必须首先锁住总线，阻止其他的CPU访问它，然后进行存储器的读写访问，再解锁总线。对总线加锁的典型做法是，先使用通常的总线协议请求总线，并申明（设置一个逻辑1）已拥有某些特定的总线线路，直到两个周期全部完成。只要始终保持拥有这一特定的总线线路，那么其他CPU就不会得到总线的访问权。这个指令只有在拥有必要的线路和和使用它们的（硬件）协议上才能实现。现代总线有这些功能，但是早期的一些总线不具备，它们不能正确地实现TSL指令。这就是Peterson 协议（完全用软件实现同步）会产生的原因（Peterson，1981）。

如果正确地实现和使用TSL，它能够保证互斥机制正常工作。但是这种互斥方法使用了自旋锁（spin lock），因为请求的CPU只是在原地尽可能快地对锁进行循环测试。这样做不仅完全浪费了提出请求的各个CPU的时间，而且还给总线或存储器增加了大量的负载，严重地降低了所有其他CPU从事正常工作的速度。

乍一看，高速缓存的实现也许能够消除总线竞争的问题，但事实并非如此。理论上，只要提出请求的CPU已经读取了锁字（lock word），它就可在其高速缓存中得到一个副本。只要没有其他CPU试图使用该锁，提出请求的CPU就能够用完其高速缓存。当拥有锁的CPU写入一个1到高速缓存并释放它时，高速缓存协议会自动地将它在远程高速缓存中的所有副本失效，要求再次读取正确的值。

问题是，高速缓存操作是在32或64字节的块中进行的。通常，拥有锁的CPU也需要这个锁周围的字。由于TSL指令是一个写指令（因为它修改了锁），所以它需要互斥地访问含有锁的高速缓存块。这样，每一个TSL都使锁持有者的高速缓存中的块失效，并且为请求的CPU取一个私有的、惟一的副本。只要锁拥有者访问到该锁的邻接字，该高速缓存块就被送进其机器。这样一来，整个包含锁的高速缓存块就会不断地在锁的拥有者和锁的请求者之间来回穿梭，导致了比单个读取一个锁字更大的总线流量。

如果能消除在请求一侧的所有由TSL引起的写操作，我们就可以明显地减少这种开销。使提出请求的CPU首先进行一个纯读操作来观察锁是否空闲，就可以实现这个目标。只有在锁看来是空闲时，TSL才真正去获取它。这种小小变化的结果是，大多数的行为变成读而不是写。如果拥有锁的CPU只是在同一个高速缓存块中读取各种变量，那么它们每个都可以以共享只读方式拥有一个高速缓存块的副本，这就消除了所有的高速缓存块传送。当锁最终被释放时，锁的所有者进行写操作，这需要排它访问，也就使远程高速缓存中的所有其他副本失效。在提出请求的CPU的下一个读请求中，高速缓存块会被重新装载。注意，如果两个或更多的CPU竞争同一个锁，那么有可能出现这样的情况，两者同时看到锁是空闲的，于是同时用TSL指令去获得它。只有其中的一个会成功，所以这里没有竞争条件，因为真正的获取是由TSL指令进行的，而且这条指令是原子性的。即使看到了锁空闲，然后立即用TSL指令试图获得它，也不能保证真正得到它。其他CPU可能会取胜，不过对于该算法的正确性来说，谁得到了锁并不重要。纯读出操作的成功只是意味着这可能是一个获得锁的好时机，但并不能确保能成功地得到锁。

另一个减少总线流量的方式是使用著名的以太网二进制指数补偿算法（binary exponential backoff algorithm）（Anderson，1990）。不是采用连续轮询，参考图2-22，而是把一个延迟循环插入轮询之间。初始的延迟是一条指令。如果锁仍然忙，延迟被加倍成为两条指令，然后，四条指令，如此这样进行，直到某个最大值。当锁释放时，较低的最大值会产生快速的响应。但是会浪费较多的总线周期在高速缓存的颠簸上。而较高的最大值可减少高速缓存的颠簸，但是其代价是不会注意到锁如此迅速地成为空闲。二进制指数补偿算法无论在有或无TSL指令前的纯读的情况下都适用。