一个更好的思想是，让每个打算获得互斥信号量的CPU都拥有各自用于测试的私有锁变量，如图8-11所示（Mellor-Crummey和 Scott，1991）。有关的变量应该存放在未使用的高速缓存块中以避免冲突。对这种算法的描述如下：给一个未能获得锁的CPU分配一个锁变量并且把它附在等待该锁的CPU链表的末端。在当前锁的持有者退出临界区时，它释放链表中的首个CPU正在测试的私有锁（在自己的高速缓存中）。然后该CPU进入临界区。操作完成之后，该CPU释放锁。其后继者接着使用，以此类推。尽管这个协议有些复杂（为了避免两个CPU同时把它们自己加在链表的末端），但它能够有效工作，而且消除了饥饿问题。具体细节，读者可以参考有关论文。

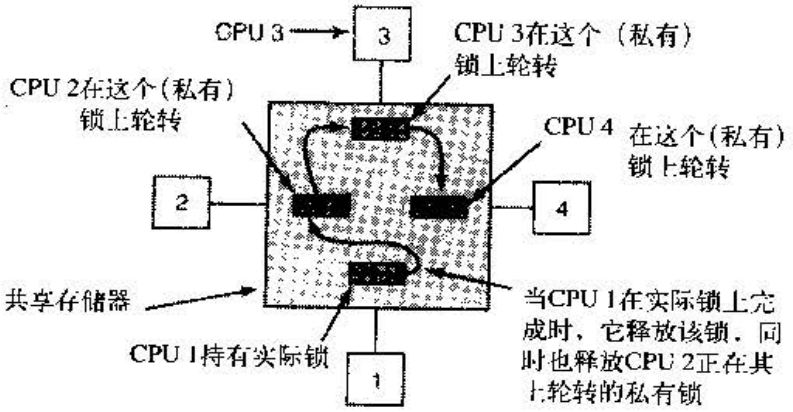

图8-11 使用多个锁以防止高速缓存颠簸

自旋与切换

到目前为止，不论是连续轮询方式、间歇轮询方式，还是把自己附在进行等候CPU链表中的方式，我们都假定需要加锁的互斥信号量的CPU只是保持等待。有时对于提出请求的CPU而言，只有等待，不存在其他替代的办法。例如，假设一些CPU是空闲的，需要访问共享的就绪链表（ready list）以便选择一个进程运行。如果就绪链表被锁住了，那么CPU就不能够只是决定暂停其正在进行的工作，而去运行另一个进程，因为这样做需要访问就绪链表。CPU必须保持等待直到能够访问该就绪链表。

然而，在另外一些情形中，却存在着别的选择。例如，如果在一个CPU中的某些线程需要访问文件系统缓冲区高速缓存，而该文件系统缓冲区高速缓存正好锁住了，那么CPU可以决定切换至另外一个线程而不是等待。有关是进行自旋还是进行线程切换的问题则是许多研究课题的内容，下面会讨论其中的一部分。请注意，这类问题在单处理机中是不存在的，因为没有另一个CPU释放锁，那么自旋就没有任何意义。如果一个线程试图取得锁并且失败，那么它总是被阻塞，这样锁的所有者有机会运行和释放该锁。

假设自旋和进行线程切换都是可行的选择，则可进行如下的权衡。自旋直接浪费了CPU周期。重复地测试锁并不是高效的工作。不过，切换也浪费了CPU周期，因为必须保存当前线程的状态，必须获得保护就绪链表的锁，还必须选择一个线程，必须装入其状态，并且使其开始运行。更进一步来说，该CPU高速缓存还将包含所有不合适的高速缓存块，因此在线程开始运行的时候会发生很多代价昂贵的高速缓存未命中。TLB的失效也是可能的。最后，会发生返回至原来线程的切换，随之而来的是更多的高速缓存未命中。花费在这两个线程间来回切换和所有高速缓存未命中的周期时间都浪费了。

如果预先知道互斥信号量通常被持有的时间，比如是50μs，而从当前线程切换需要1ms，稍后切换返回还需1ms，那么在互斥信号量上自旋则更为有效。另一方面，如果互斥信号量的平均保持时间是10ms，那就值得忍受线程切换的麻烦。问题在于，临界区在这个期间会发生相当大的变化，所以，哪一种方法更好些呢？

有一种设计是总是进行自旋。第二种设计方案则总是进行切换。而第三种设计方案是每当遇到一个锁住的互斥信号量时，就单独做出决定。在必须做出决定的时刻，并不知道自旋和切换哪一种方案更好，但是对于任何给定的系统，有可能对其所有的有关活动进行跟踪，并且随后进行离线分析。然后就可以确定哪个决定最好及在最好情形下所浪费的时间。这种事后算法（hindsight algorithm）成为对可行算法进行测量的基准评测标准。

已有研究人员对上述这一问题进行了研究（Karlin 等人，1989；Karlin等人，1991；Ousterhout，1982）。多数的研究工作使用了这样一个模型：一个未能获得互斥信号量的线程自旋一段时间。如果时间超过某个阈值，则进行切换。在某些情形下，该阈值是一个定值，典型值是切换至另一个线程再切换回来的开销。在另一些情形下，该阈值是动态变化的，它取决于所观察到的等待互斥信号量的历史信息。

在系统跟踪若干最新的自旋时间并且假定当前的情形可能会同先前的情形类似时，就可以得到最好