的线程设置一个进程范围内的标志以表示它目前拥有了一个自旋锁（Zahorjan 等人，1991）。当它释放该自旋锁时，就清除这个标志。这样调度程序就不会停止持有自旋锁的线程，相反，调度程序会给予稍微多一些的时间让该线程完成临界区内的工作并释放自旋锁。

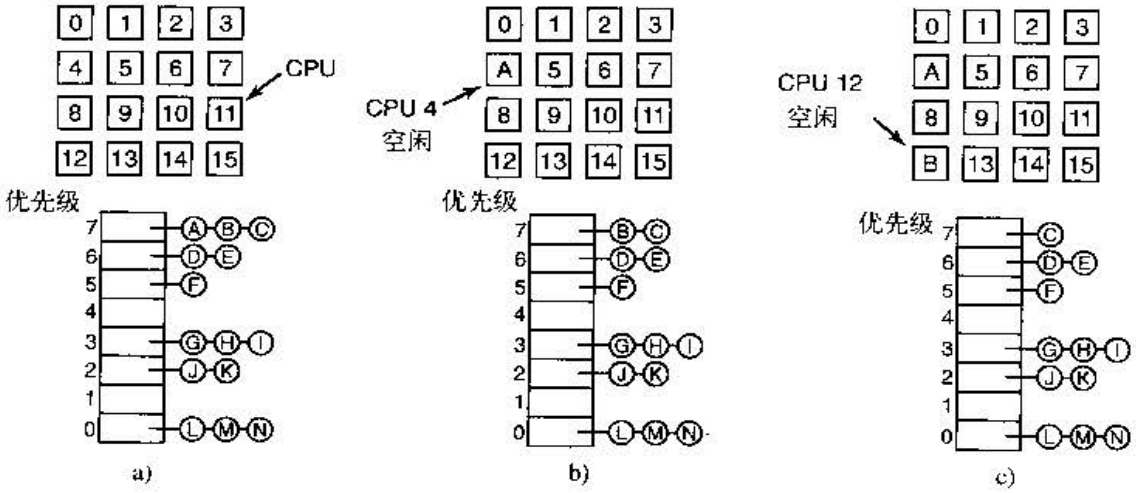

图8-12 使用单一数据结构调度一个多处理机

调度中的另一个主要问题是，当所有CPU平等时，某些CPU更平等。特别是，当线程A已经在CPU k上运行了很长一段时间时，CPU k的高速缓存装满了A的块。若A很快重新开始运行，那么如果它在CPU k上运行性能可能会更好一些，因为k的高速缓存也许还存有A的一些块。预装高速缓存块将提高高速缓存的命中率，从而提高了线程的速度。另外，TLB也可能含有正确的页面，从而减少了TLB失效。

有些多处理机考虑了这一因素，并使用了所谓*亲和调度*（affinity scheduling）（Vaswani 和Zahorjan，1991）。其基本思想是，尽量使一个线程在它前一次运行过的同一个CPU上运行。创建这种亲和力（affinity）的一种途径是采用一种*两级调度算法*（two-level scheduling algorithm）。在一个线程创建时，它被分给一个CPU，例如，可以基于哪一个CPU在此刻有最小的负载。这种把线程分给CPU的工作在算法的顶层进行，其结果是每个CPU获得了自己的线程集。

线程的实际调度工作在算法的底层进行。它由每个CPU使用优先级或其他的手段分别进行。通过试图让一个线程在其生命周期内在同一个CPU上运行的方法，高速缓存的亲和力得到了最大化。不过，如果某一个CPU没有线程运行，它便选取另一个CPU的一个线程来运行而不是空转。

两级调度算法有三个优点。第一，它把负载大致平均地分配在可用的CPU上；第二，它尽可能发挥了高速缓存亲和力的优势；第三，通过为每个CPU提供一个私有的就绪线程链表，使得对就绪线程链表的竞争减到了最小，因为试图使用另一个CPU的就绪线程链表的机会相对较小。

2. *空间共享*

当线程之间以某种方式彼此相关时，可以使用其他多处理机调度方法。前面我们叙述过的并行make就是一个例子。经常还有一个线程创建多个共同工作的线程的情况发生。例如当一个进程的多个线程间频繁地进行通信，让其在同一时间执行就显得尤为重要。在多个CPU上同时调度多个线程称为*空间共享*（space sharing）。

最简单的空间共享算法是这样工作的。假设一组相关的线程是一次性创建的。在其创建的时刻，调度程序检查是否有同线程数量一样多的空闲CPU存在。如果有，每个线程获得各自专用的CPU（非多道程序处理）并且都开始运行。如果没有足够的CPU，就没有线程开始运行，直到有足够的CPU时为止。每个线程保持其CPU直到它终止，并且该CPU被送回可用CPU池中。如果一个线程在I/O上阻塞，它继续保持其CPU，而该CPU就空闲直到该线程被唤醒。在下一批线程出现时，应用同样的算法。

在任何一个时刻，全部CPU被静态地划分成若干个分区，每个分区都运行一个进程中的线程。例如，在图8-13中，分区的大小是4、6、8和12个CPU，有两个CPU没有分配。随着时间的流逝，新的线程创建，旧的线程终止，CPU分区大小和数量都会发生变化。