必须进行周期性的调度决策。在单处理机系统中，最短作业优先是批处理调度中知名的算法。在多处理机系统中类似的算法是，选择需要最少的CPU周期数的线程，也就是其CPU周期数×运行时间最小的线程为候选线程。然而，在实际中，这一信息很难得到，因此该算法难以实现。事实上，研究表明，要胜过先来先服务算法是非常困难的（Krueger等人，1994）。

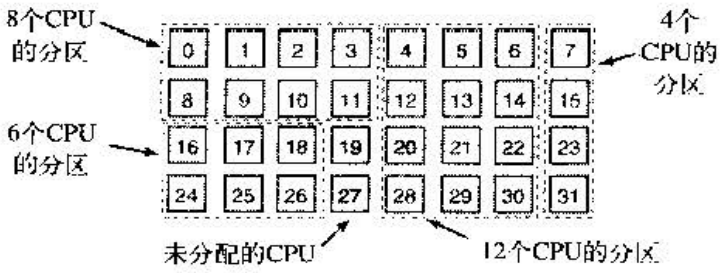

图8-13 一个32个CPU的集合被分成4个分区，两个CPU可用

在这个简单的分区模型中，一个线程请求一定数量的CPU，然后或者全部得到它们或者一直等到有足够数量的CPU可用为止。另一种处理方式是主动地管理线程的并行度。管理并行度的一种途径是使用一个中心服务器，用它跟踪哪些线程正在运行，哪些线程希望运行以及所需CPU的最小和最大数量（Tucker和Gupta，1989）。每个应用程序周期性地询问中心服务器有多少个CPU可用。然后它调整线程的数量以符合可用的数量。例如，一台Web服务器可以5、10、20或任何其他数量的线程并行运行。如果它当前有10个线程，突然，系统对CPU的需求增加了，于是它被通知可用的CPU数量减到了5个，那么在接下来的5个线程完成其当前工作之后，它们就被通知退出而不是给予新的工作。这种机制允许分区大小动态地变化，以便与当前负载相匹配，这种方法优于图8-13中的固定系统。

3. 群调度（Gang Scheduling）

空间共享的一个明显优点是消除了多道程序设计，从而消除了上下文切换的开销。但是，一个同样明显的缺点是当CPU被阻塞或根本无事可做时时间被浪费了，只有等到其再次就绪。于是，人们寻找既可以调度时间又可以调度空间的算法，特别是对于要创建多个线程而这些线程通常需要彼此通信的线程。

为了考察一个进程的多个线程被独立调度时会出现的问题，设想一个系统中有线程A_0和A_1属于进程A，而线程B_0和B_1属于进程B。线程A_0和B_0在CPU 0上分时；而线程A_1和B_1在CPU 1上分时。线程A_0和A_1需要经常通信。其通信模式是，A_0送给A_1一个消息，然后A_1回送给A_0一个应答，紧跟的是另一个这样的序列。假设正好是A_0和B_1首先开始，如图8-14所示。

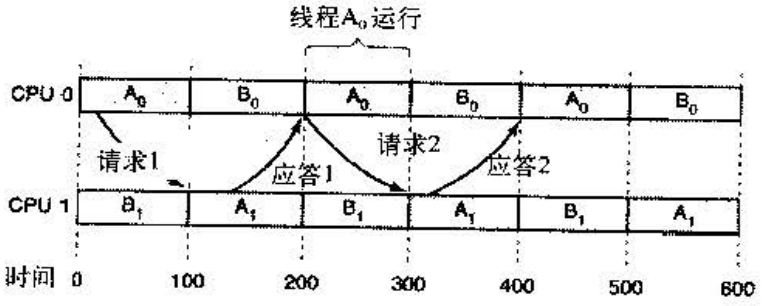

图8-14 进程A的两个异步运行的线程间的通信

在时间片0，A_0发给A_1一个请求，但是直到A_1在开始于100ms的时间片1中开始运行时它才得到该消息。它立即发送一个应答，但是直到A_0在200ms再次运行时它才得到该应答。最终结果是每200ms一个请求-应答序列。这个结果并不好。

这一问题的解决方案是群调度（gang scheduling），它是协同调度（co-scheduling）（Outsterhout，1982）的发展产物。群调度由三个部分组成：

1) 把一组相关线程作为一个单位，即一个群（gang），一起调度。

2) 一个群中的所有成员在不同的分时CPU上同时运行。

3) 群中的所有成员共同开始和结束其时间片。

使群调度正确工作的关键是，同步调度所有的CPU。这意味着把时间划分为离散的时间片，如图8-14中所示。在每一个新的时间片开始时，所有的CPU都重新调度，在每个CPU上都开始一个新的线程。在后续的时间片开始时，另一个调度事件发生。在这之间，没有调度行为。如果某个线程被阻塞，它的