的结果。例如，假定还是1ms切换时间，线程自旋时间最长为2ms，但是要观察实际上自旋了多长时间。如果线程未能获取锁，并且发现在之前的三轮中，平均等待时间为200μs，那么，在切换之前就应该先自旋2ms。但是，如果发现在先前的每次尝试中，线程都自旋了整整2ms，则应该立即切换而不再自旋。更多的细节可以在（Karlin 等人，1991）中找到。

8.1.4 多处理机调度

在探讨多处理机调度之前，需要确定调度的对象是什么。过去，当所有进程都是单个线程的时候，调度的单位是进程，因为没有其他什么可以调度的。所有的现代操作系统都支持多线程进程，这让调度变得更加复杂。

线程是内核线程还是用户线程至关重要。如果线程是由用户空间库维护的，而对内核不可见，那么调度一如既往的基于单个进程。如果内核并不知道线程的存在，它就不能调度线程。

对内核线程来说，情况有所不同。在这种情况下所有线程均是内核可见的，内核可以选择一个进程的任一线程。在这样的系统中，发展趋势是内核选择线程作为调度单位，线程从属的那个进程对于调度算法只有很少的（乃至没有）影响。下面我们将探讨线程调度，当然，对于一个单线程进程（single-threaded process）系统或者用户空间线程，调度单位依然是进程。

进程和线程的选择并不是调度中的惟一问题。在单处理机中，调度是一维的。惟一必须（不断重复地）回答的问题是："接下来运行的线程应该是哪一个？"而在多处理机中，调度是二维的。调度程序必须决定哪一个进程运行以及在哪一个CPU上运行。这个在多处理机中增加的维数大大增加了调度的复杂性。

另一个造成复杂性的因素是，在有些系统中所有的线程是不相关的，而在另外一些系统中它们是成组的，同属于同一个应用并且协同工作。前一种情形的例子是分时系统，其中独立的用户运行相互独立的进程。这些不同进程的线程之间没有关系，因此其中的每一个都可以独立调度而不用考虑其他的线程。

后一种情形的例子通常发生在程序开发环境中。大型系统中通常有一些供实际代码使用的包含宏、类型定义以及变量声明等内容的头文件。当一个头文件改变时，所有包含它的代码文件必须被重新编译。通常make程序用于管理并发工作。调用make程序时，在考虑了头文件或代码文件的修改之后，它仅编译那些必须重新编译的代码文件。仍然有效的目标文件不再重新生成。

make的原始版本是顺序工作的，不过为多处理机设计的新版本可以一次启动所有的编译。如果需要10个编译，那么迅速对9个进行调度而让最后一个在很长的时间之后才进行的做法没有多大意义，因为直到最后一个线程完毕之后用户才感觉到工作完成了。在这种情况下，将进行编译的线程看作一组，并在对其调度时考虑到这一点是有意义的。

1. 分时

让我们首先讨论调度独立线程的情况。稍后，我们将考虑如何调度相关的线程。处理独立线程的最简单算法是，为就绪线程维护一个系统级的数据结构，它可能只是一个链表，但更多的情况下可能是对应不同优先级一个链表集合，如图8-12a所示。这里16个CPU正在忙碌，有不同优先级的14个线程在等待运行。第一个将要完成其当前工作（或其线程将被阻塞）的CPU是CPU 4，然后CPU 4锁住调度队列（scheduling queue）并选择优先级最高的线程A，如图8-12b所示。接着，CPU 12 空闲并选择线程 B，参见图8-12c。只要线程完全无关，以这种方式调度是明智的选择并且其很容易高效地实现。

由所有CPU使用的单个调度数据结构分时共享这些CPU，正如它们在一个单处理机系统中那样。它还支持自动负载平衡，因为决不会出现一个CPU空闲而其他CPU过载的情况。不过这一方法有两个缺点，一个是随着CPU数量增加所引起的对调度数据结构的潜在竞争，二是当线程由于I/O 阻塞时所引起上下文切换的开销（overhead）。

在线程的时间片用完时，也可能发生上下文切换。在多处理机中它有一些在单处理机中不存在的属性。假设某个线程在其时间片用完时持有一把自旋锁。在该线程被再次调度并且释放该锁之前，其他等待该自旋锁的CPU只是把时间浪费在自旋上。在单处理机中，极少采用自旋锁，因此，如果持有互斥信号量的一个线程被挂起，而另一个线程启动并试图获取该互斥信号量，则该线程会立即被阻塞，这样只浪费了少量时间。

为了避免这种异常情况，一些系统采用智能调度（smart scheduling）的方法，其中，获得了自旋锁