因此细粒度的共享成为了可能。任务数据结构只需要指向这些数据结构即可，所以为每一个线程创建一个新的任务数据结构变得很容易，或者使它指向旧线程的调度参数、内存映射和其他的数据结构，或者复制它们。事实上，条理分明的共享性虽然成为了可能，但并不意味着它是有益的，毕竟传统的UNIX系统都没有提供这样的功能。一个利用了这种共享性的Linux程序将不能移植到UNIX系统上。

Linux系统的线程模型带来了另一个难题。UNIX系统为每一个进程分配一个独立的PID，不论它是单线程的进程还是多线程的进程。为了能与其他的UNIX系统兼容，Linux对进程标识符（PID）和任务标识符（TID）进行了区分。这两个分量都存储在任务数据结构中。当调用clone函数创建一个新进程而不需要和旧进程共享任何信息时，PID被设置成一个新值；否则，任务得到一个新的任务标识符，但是PID不变。这样一来，一个进程中所有的线程都会拥有与该进程中第一个线程相同的PID。

### 10.3.4 Linux中的调度

现在我们来关注Linux系统的调度算法。首先要认识到，Linux系统的线程是内核线程，所以Linux系统的调度是基于线程的，而不是基于进程的。

为了进行调度，Linux系统将线程区分为三类：

1) 实时先入先出。

2) 实时轮转。

3) 分时。

实时先入先出线程具有最高优先级，它不会被其他线程抢占，除非那是一个刚刚准备好的、拥有更高优先级的实时先入先出线程。实时轮转线程与实时先入先出线程基本相同，只是每个实时轮转线程都有一个时间量，时间到了之后就可以被抢占。如果多个实时轮转线程都准备好了，每一个线程运行它的时间量所规定的时间，然后插入到实时轮转线程列表的末尾。事实上，这两类线程都不是真正的实时线程。执行的最后期限无法确定，更无法保证最后期限前线程可以执行完毕。这两类线程比起分时线程来说只是具有更高的优先级而已。Linux系统之所以称它们为“实时”是因为Linux系统遵循的P1003.4标准（UNIX系统对“实时”含义的扩展）使用了这个名称。在系统内部，实时线程的优先级从0到99，0是实时线程的最高优先级，99是实时线程的最低优先级。

传统的非实时线程按照如下的算法进行调度。在系统内部，非实时线程的优先级从100到139，也就是说，在系统内部，Linux系统区分140级的优先级（包括实时和非实时任务）。就像实时轮转线程一样，Linux系统根据非实时线程的优先级分配时间量。这个时间量是线程可以连续运行的时钟周期数。在当前的Linux版本中，时钟频率为1000赫兹，每个时钟周期为1ms，也叫做一个最小时间间隔（jiffy）。

像大多数UNIX系统一样，Linux系统给每个线程分配一个nice值（即优先级调节值）。默认值是0，但是可以通过调用系统调用nice（value）来修改，修改值的范围从−20到+19。这个值决定了线程的静态优先级。一个在后台大量计算π值的用户可以在他的程序里调用这个系统调用为其他用户让出更多计算资源。只有系统管理员可以要求比普通服务更好的服务（意味着nice函数参数值的范围从−20到−1）。推断这条规则的理由作为练习留给读者。

Linux调度算法使用一个重要的数据结构——调度队列。在系统中，一个CPU有一个调度队列，除了其他信息，调度队列中有两个数组，一个是正在活动的，一个是过期失效的。如图10-10所示，这两个分量都是指向数组的指针，每个数组都包含了140个链表头，每个链表具有不同的优先级。链表头指向给定优先级的双向进程链表。调度的基本操作如下所述。

调度器从正在活动数组中选择一个优先级最高的任务。如果这个任务的时间片（时间量）过期失效了，就把它移动到过期失效数组中（可能会插入到优先级不同的列表中）。如果这个任务阻塞了，比如说正在等待I/O事件，那么在它的时间片过期失效之前，一旦所等待的事件发生，任务就可以继续运行，它将被放回到之前正在活动的数组中，时间片根据它所消耗的CPU时间相应的减少。一旦它的时间片消耗殆尽，它也会被放到过期失效数组中。当正在活动数组中没有其他的任务了，调度器交换指针，使得正在活动数组变为过期失效数组，过期失效数组变为正在活动数组。这种方法可以保证低优先级的任务不会被饿死（除非实时先入先出线程完全占用CPU，但是这种情况是不会发生的）。

不同的优先级被赋予不同的时间片长度。Linux系统会赋予高优先级的进程较长的时间片。例如，