应用程序的线程通常在优先级1～15上运行。通过设定进程和线程的优先级，一个应用程序可以决定哪些线程得到偏爱（获得更高优先级）。ZeroPage系统线程运行在优先级0并且把所有要释放的页转化为全部包含0的页。每一个实时的处理器都有一个独立的ZeroPage线程。

每个线程都有一个基于进程优先级的基本优先级和一个线程自己的相对优先级。用于决定一个线程在32个列表中的哪一个列表进行排队的优先级取决于当前优先级，通常是得到和当前线程的基本优先级一样的优先级，但并不总是这样。在特定的情况下，非实时线程的当前优先级被内核一下子提到尽可能高的优先级（但是不会超过优先级15）。因为图11-28的排列以当前的优先级为基础，所以改变优先级可以影响调度。对于实时优先级的线程，没有任何的调整。

现在让我们看看一个线程在什么样的时机会得到提升。首先，当输入输出操作完成并且唤醒一个等待线程的时候，优先级一下子被提高，给它一个快速运行的机会，这样可以使更多的I/O可以得到处理。这里保证I/O设备处于忙碌的运行状态。提升的幅度依赖于输入输出设备，典型地磁盘片对应于1级，串行总线对应于2级，6级对应于键盘，8级对应于声卡。

其次，如果一个线程在等待信号量，互斥量同步或其他的事件，当这些条件满足线程被唤醒的时候，如果它是前台的进程（该进程控制键盘输入发送到的窗口）的话，这个线程就会得到两个优先级的提升，其他情况则只提升一个优先级。这倾向于把交互式的进程优先级提升到8级以上。最后，如果一个窗口输入就绪使得图形用户接口线程被唤醒，它的优先级同样会得到大幅提升。

提升不是永远的。优先级的提升是立刻发生作用的，并且会引起处理器的再次调度。但是如果一个线程用完它的时间分配量，它就会降低一个优先级而且排在新优先级队列的队尾。如果它两次用完一个完整的时间配额，它就会再降一个优先级，如此下去直到降到它的基本优先级，在基本优先级得到保持不会再降，直到它的优先级再次得到提升。

还有一种情况就是系统变动（fiddle）优先级。假设有二个线程正在一个生产者-消费者类型问题上一起协同工作。生产者的工作需要更多的资源，因此，它得到高的优先级，例如说12，而消费者得到的优先级为4。在特定的时刻，生产者已经把共享的缓冲区填满，信号量发生阻塞，如图11-29a所示。

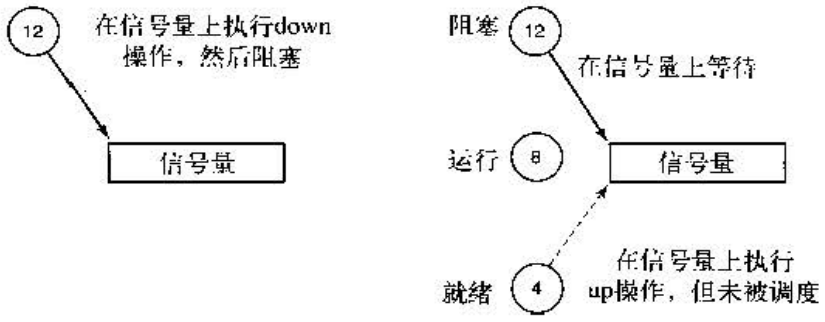

图11-29 优先级转置的示例

如图11-29b所示，在消费者得到调度再次运行之前，一个无关的线程在优先级8已就绪得到调度运行。只要这个线程想要运行，它将会一直运行，因为这个线程的优先级高于消费者的优先级，而比它优先级高的生产者由于阻塞也不能够运行。在这种情况下，直到优先级为8的线程运行完毕，生产者才有机会再次运行。

Windows通过一个称为大hack来解决此类问题的。系统记录一个已就绪的线程自从上次得到运行后距离当前的时间有多久。如果它超过一个特定的阈值，它就被提升到15级的优先级并得到两个时间配额的运转。这就可能解决生产者阻塞的情况。在两个时间配额用完之后，它的优先级一下子又回到原来的优先级而不是逐级别地缓慢下降到原来的优先级。或许较好的解决方法是把那些用完时间配额的线程的优先级不断地降低。毕竟，问题不是由饥饿的线程所引起的，而是由贪婪线程造成的。这一问题广为人知地称作*优先级倒转*（priority inversion）。

在优先为16条线程获得互斥量却长时间得不到调度的时候会发生一个类似的问题，致使更重要的系统线程由于等待互斥量而不能运行发生饥饿。这一问题在操作系统里通过在那些只需要短时间拥有互斥