为了在C语言中表示sleep和wakeup这样的系统调用，我们将以库函数调用的形式来表示。尽管它们不是标准C库的一部分，但在实际上任何系统中都具有这些库函数。未列出的过程insert_item和remove_item用来记录将数据项放入缓冲区和从缓冲区取出数据等事项。

现在回到竞争条件的问题。这里有可能会出现竞争条件，其原因是对count的访问未加限制。有可能出现以下情况：缓冲区为空，消费者刚刚读取count的值发现它为0。此时调度程序决定暂停消费者并启动运行生产者。生产者向缓冲区中加入一个数据项，count加1。现在count的值变成了1。它推断认为由于count刚才为0，所以消费者此时一定在睡眠，于是生产者调用wakeup来唤醒消费者。

但是，消费者此时在逻辑上并未睡眠，所以wakeup信号丢失。当消费者下次运行时，它将测试先前读到的count值，发现它为0，于是睡眠。生产者迟早会填满整个缓冲区，然后睡眠。这样一来，两个进程都将永远睡眠下去。

问题的实质在于发给一个（尚）未睡眠进程的wakeup信号丢失了。如果它没有丢失，则一切都很正常。一种快速的弥补方法是修改规则，加上一个唤醒等待位。当一个wakeup信号发送给一个清醒的进程信号时，将该位置1。随后，当该进程要睡眠时，如果唤醒等待位为1，则将该位清除，而该进程仍然保持清醒。唤醒等待位实际上就是wakeup信号的一个小仓库。

尽管在这个简单例子中用唤醒等待位的方法解决了问题，但是我们很容易就可以构造出一些例子，其中有三个或更多的进程，这时一个唤醒等待位就不够使用了。于是我们可以再打一个补丁，加入第二个唤醒等待位，甚至是8个、32个等，但原则上讲，这并没有从根本上解决问题。

2.3.5 信号量

信号量是E. W. Dijkstra在1965年提出的一种方法，它使用一个整型变量来累计唤醒次数，供以后使用。在他的建议中引入了一个新的变量类型，称作信号量（semaphore）。一个信号量的取值可以为0（表示没有保存下来的唤醒操作）或者为正值（表示有一个或多个唤醒操作）。

Dijkstra建议设立两种操作：down和up（分别为一般化后的sleep和wakeup）。对一信号量执行down操作，则是检查其值是否大于0。若该值大于0，则将其值减1（即用掉一个保存的唤醒信号）并继续；若该值为0，则进程将睡眠，而且此时down操作并未结束。检查数值、修改变量值以及可能发生的睡眠操作均作为一个单一的、不可分割的原子操作完成。保证一旦一个信号量操作开始，则在该操作完成或阻塞之前，其他进程均不允许访问该信号量。这种原子性对于解决同步问题和避免竞争条件是绝对必要的。所谓原子操作，是指一组相关联的操作要么都不间断地执行，要么都不执行。原子操作在计算机科学的其他领域也是非常重要的。

up操作对信号量的值增1。如果一个或多个进程在该信号量上睡眠，无法完成一个先前的down操作，则由系统选择其中的一个（如随机挑选）并允许该进程完成它的down操作。于是，对一个有进程在其上睡眠的信号量执行一次up操作之后，该信号量的值仍旧是0，但在其上睡眠的进程却少了一个。信号量的值增1和唤醒一个进程同样也是不可分割的。不会有某个进程因执行up而阻塞，正如在前面的模型中不会有进程因执行wakeup而阻塞一样。

顺便提一下，在Dijkstra原来的论文中，他分别使用名称P和V而不是down和up，荷兰语中，Proberen的意思是尝试，Verhogen的含义是增加或升高。由于对于不讲荷兰语的读者来说采用什么记号并无大的干系，所以我们将使用down和up名称。它们在程序设计语言Algol 68中首次引入。

用信号量解决生产者-消费者问题

用信号量解决丢失的wakeup问题，如图2-28所示。为确保信号量能正确工作，最重要的是要采用一种不可分割的方式来实现它。通常是将up和down作为系统调用实现，而且操作系统只需在执行以下操作时暂时屏蔽全部中断：测试信号量、更新信号量以及在需要时使某个进程睡眠。由于这些动作只需要几条指令，所以屏蔽中断不会带来什么副作用。如果使用多个CPU，则每个信号量应由一个锁变量进行保护。通过TSL或XCHG指令来确保同一时刻只有一个CPU在对信号量进行操作。

读者必须搞清楚，使用TSL或XCHG指令来防止几个CPU同时访问一个信号量，这与生产者或消费者使用忙等待来等待对方腾出或填充缓冲区是完全不同的。信号量操作仅需几个毫秒，而生产者或消费者则可能需要任意长的时间。