当一个进程想进入临界区时，先检查是否允许进入，若不允许，则该进程将原地等待，直到允许为止。

这种方法不仅浪费了CPU时间，而且还可能引起预想不到的结果。考虑一台计算机有两个进程，H优先级较高，L优先级较低。调度规则规定，只要H处于就绪态它就可以运行。在某一时刻，L处于临界区中，此时H变到就绪态，准备运行（例如，一条I/O操作结束）。现在H开始忙等待，但由于当H就绪时L不会被调度，也就无法离开临界区，所以H将永远忙等待下去。这种情况有时被称作*优先级反转问题*（priority inversion problem）。

现在来考察几条进程间通信原语，它们在无法进入临界区时将阻塞，而不是忙等待。最简单的是sleep和wakeup。sleep是一个将引起调用进程阻塞的系统调用，即被挂起，直到另外一个进程将其唤醒。wakeup调用有一个参数，即要被唤醒的进程。另一种方法是让sleep和wakeup各有一个参数，即有一个用于匹配sleep和wakeup的内存地址。

生产者-消费者问题

作为使用这些原语的一个例子，我们考虑*生产者-消费者*（producer-consumer）问题，也称作*有界缓冲区*（bounded-buffer）问题。两个进程共享一个公共的固定大小的缓冲区。其中一个是生产者，将信息放入缓冲区；另一个是消费者，从缓冲区中取出信息。（也可以把这个问题一般化为m个生产者和n个消费者问题，但是我们只讨论一个生产者和一个消费者的情况，这样可以简化解决方案。）

问题在于当缓冲区已满，而此时生产者还想向其中放入一个新的数据项的情况。其解决办法是让生产者睡眠，待消费者从缓冲区中取出一个或多个数据项时再唤醒它。同样地，当消费者试图从缓冲区中取数据而发现缓冲区为空时，消费者就睡眠，直到生产者向其中放入一些数据时再将其唤醒。

这个方法听起来很简单，但它包含与前边假脱机目录问题一样的竞争条件。为了跟踪缓冲区中的数据项数，我们需要一个变量count。如果缓冲区最多存放N个数据项，则生产者代码将首先检查count是否达到N，若是，则生产者睡眠；否则生产者向缓冲区中放入一个数据项并增量count的值。

消费者的代码与此类似：首先测试count是否为0，若是，则睡眠；否则从中取走一个数据项并递减count的值。每个进程同时也检测另一个进程是否应被唤醒，若是则唤醒之。生产者和消费者的代码如图2-27所示。

```
#define N 100                                   /* 缓冲区中的槽数目 */
int count = 0;                                  /* 缓冲区中的数据项数目 */

void producer(void)
{
     int item;

     while (TRUE) {                             /* 无限循环 */
          item = produce_item();                /* 产生下一新数据项 */
          if (count == N) sleep();              /* 如果缓冲区满了，就进入休眠状态 */
          insert_item(item);                    /* 将（新）数据项放入缓冲区中 */
          count = count + 1;                    /* 将缓冲区的数据项计数器增1 */
          if (count == 1) wakeup(consumer);     /* 缓冲区空吗？ */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* 无限循环 */
          if (count == 0) sleep();              /* 如果缓冲区空，则进入休眠状态 */
          item = remove_item();                 /* 从缓冲区中取出一个数据项 */
          count = count - 1;                    /* 将缓冲区的数据项计数器减1 */
          if (count == N - 1) wakeup(producer); /* 缓冲区满吗？ */
          consume_item(item);                   /* 打印数据项 */
     }
}
```

图2-27 含有严重竞争条件的生产者-消费者问题