在等待的条件变量，那么这个信号就会丢失。程序员必须小心使用避免丢失信号。

作为如何使用一个互斥量与条件变量的例子，图2-32展示了一个非常简单只有一个缓冲区的生产者-消费者问题。当生产者填满缓冲区时，它在生产下一个数据项之前必须等待，直到消费者清空了它。类似地，当消费者移走一个数据项时，它必须等待，直到生产者生产了另外一个数据项。尽管很简单，这个例子却说明了基本的机制。使一个线程睡眠的语句应该总是要检查这个条件，以保证线程在继续执行前满足条件，因为线程可能已经因为一个UNIX信号或其他原因而被唤醒。

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000                          /* 需要生产的数量 */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;
int buffer = 0;                                 /* 生产者消费者使用的缓冲区 */

void *producer(void *ptr)                       /* 生产数据 */
{     int i;

      for (i= 1; i <= MAX; i++) {
            pthread_mutex_lock(&the_mutex);  /* 互斥使用缓冲区 */
            while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
            buffer = i;                         /* 将数据放入缓冲区 */
            pthread_cond_signal(&condc);        /* 唤醒消费者 */
            pthread_mutex_unlock(&the_mutex); /*释放缓冲区 */
      }
      pthread_exit(0);
}

void *consumer(void *ptr)                       /* 消费数据 */
{     int i;

      for (i = 1; i <= MAX; i++) {
            pthread_mutex_lock(&the_mutex);   /* 互斥使用缓冲区 */
            while (buffer ==0 ) pthread_cond_wait(&condc, &the_mutex);
            buffer = 0;                         /* 从缓冲区中取出数据 */
            pthread_cond_signal(&condp);        /* 唤醒生产者 */
            pthread_mutex_unlock(&the_mutex);  /* 释放缓冲区 */
      }
      pthread_exit(0);
}

int main(int argc, char **argv)
{
      pthread_t pro, con;
      pthread_mutex_init(&the_mutex, 0);
      pthread_cond_init(&condc, 0);
      pthread_cond_init(&condp, 0);
      pthread_create(&con, 0, consumer, 0);
      pthread_create(&pro, 0, producer, 0);
      pthread_join(pro, 0);
      pthread_join(con, 0);
      pthread_cond_destroy(&condc);
      pthread_cond_destroy(&condp);
      pthread_mutex_destroy(&the_mutex);
}
```

图2-32 利用线程解决生产者-消费者问题

2.3.7 管程

有了信号量和互斥量之后，进程间通信看来就很容易了，实际是这样的吗？答案是否定的。请仔细考察图2-28中向缓冲区放入数据项以及从中删除数据项之前的down操作。假设将生产者代码中的两个down操作交换一下次序，将使得mutex的值在empty之前而不是在其之后被减1。如果缓冲区完全满了，生产者将阻塞，mutex值为0。这样一来，当消费者下次试图访问缓冲区时，它将对mutex执行一个down操作，由于mutex值为0，则消费者也将阻塞。两个进程都将永远地阻塞下去，无法再进行有效的工作，这种不幸的状况称作死锁（dead lock）。我们将在第6章中详细讨论死锁问题。