并可以访问共享内存。

另一方面，对内核来说，当它在更新变量或列表的几条指令期间将中断屏蔽是很方便的。当就绪进程队列之类的数据状态不一致时发生中断，则将导致竞争条件。所以结论是：屏蔽中断对于操作系统本身而言是一项很有用的技术，但对于用户进程则不是一种合适的通用互斥机制。

由于多核芯片的数量越来越多，即使在低端PC上也是如此。因此，通过屏蔽中断来达到互斥的可能性——甚至在内核中——变得日益减少了。双核现在已经相当普遍，四核当前在高端机器中存在，而且我们离八或十六（核）也不久远了。在一个多核系统中（例如，多处理器系统），屏蔽一个CPU的中断不会阻止其他CPU干预第一个CPU所做的操作。结果是人们需要更加复杂的计划。

2. 锁变量

作为第二种尝试，可以寻找一种软件解决方案。设想有一个共享（锁）变量，其初始值为0。当一个进程想进入其临界区时，它首先测试这把锁。如果该锁的值为0，则该进程将其设置为1并进入临界区。若这把锁的值已经为1，则该进程将等待直到其值变为0。于是，0就表示临界区内没有进程，1表示已经有某个进程进入临界区。

但是，这种想法也包含了与假脱机目录一样的疏漏。假设一个进程读出锁变量的值并发现它为0，而恰好在它将其值设置为1之前，另一个进程被调度运行，将该锁变量设置为1。当第一个进程再次能运行时，它同样也将该锁设置为1，则此时同时有两个进程进入临界区中。

可能读者会想，先读出锁变量，紧接着在改变其值之前再检查一遍它的值，这样便可以解决问题。但这实际上无济于事，如果第二个进程恰好在第一个进程完成第二次检查之后修改了锁变量的值，则同样还会发生竞争条件。

3. 严格轮换法

第三种互斥的方法如图2-23所示。几乎与本书中所有其他程序一样，这里的程序段用C语言编写。之所以选择C语言是由于实际的操作系统普遍用C语言编写（或偶尔用C++），而基本上不用像Java、Modula3或Pascal这样的语言。对于编写操作系统而言，C语言是强大、有效、可预知和有特性的语言。而对于Java，它就不是可预知的，因为它在关键时刻会用完存储器，而在不合适的时候会调用垃圾收集程序回收内存。在C语言中，这种情形就不可能发生，因为C语言中不需要进行空间回收。有关C、C++、Java和其他四种语言的定量比较可参阅（Prechelt，2000）。

在图2-23中，整型变量turn，初始值为0，用于记录轮到哪个进程进入临界区，并检查或更新共享内存。开始时，进程0检查turn，发现其值为0，于是进入临界区。进程1也发现其值为0，所以在一个等待循环中不停地测试turn，看其值何时变为1。连续测试一个变量直到某个值出现为止，称为忙等待（busy waiting）。由于这种方式浪费CPU时间，所以通常应该避免。

```
while (TRUE) {
    while (turn != 0)        /* 循环 */
    critical_region();
    turn = 1;
    noncritical_region();
}
```

a)

```
while (TRUE) {
    while (turn != 1)        /* 循环 */
    critical_region();
    turn = 0;
    noncritical_region();
}
```

b)

图2-23 临界区问题的一种解法（在两种情况下请注意分号终止了while语句）：a) 进程0；b) 进程1

只有在有理由认为等待时间是非常短的情形下，才使用忙等待。用于忙等待的锁，称为自旋锁（spin lock）。

进程0离开临界区时，它将turn的值设置为1，以便允许进程1进入其临界区。假设进程1很快便离开了临界区，则此时两个进程都处于临界区之外，turn的值又被设置为0。现在进程0很快就执行完其整个循环，它退出临界区，并将turn的值设置为1。此时，turn的值为1，两个进程都在其临界区外执行。

突然，进程0结束了非临界区的操作并且返回到循环的开始。但是，这时它不能进入临界区，因为turn的当前值为1，而此时进程1还在忙于非临界区的操作，进程0只有继续while循环，直到进程1把turn