的值改为0。这说明，在一个进程比另一个慢了很多的情况下，轮流进入临界区并不是一个好办法。

这种情况违反了前面叙述的条件3：进程0被一个临界区之外的进程阻塞。再回到前面假脱机目录的问题，如果我们现在将临界区与读写假脱机目录相联系，则进程0有可能因为进程1在做其他事情而被禁止打印另一个文件。

实际上，该方案要求两个进程严格地轮流进入它们的临界区，如假脱机文件等。任何一个进程都不可能在一轮中打印两个文件。尽管该算法的确避免了所有的竞争条件，但由于它违反了条件3，所以不能作为一个很好的备选方案。

4. Peterson解法

荷兰数学家T. Dekker通过将锁变量与警告变量的思想相结合，最早提出了一个不需要严格轮换的软件互斥算法。关于Dekker的算法，请参阅（Dijkstra，1965）。

1981年，G. L. Peterson 发现了一种简单得多的互斥算法，这使得Dekker的方法不再有任何新意。Peterson的算法如图2-24所示。该算法由两个用ANSI C编写的过程组成。ANSI C要求为所定义和使用的所有函数提供函数原型。不过，为了节省篇幅，在这里和后续的例子中我们将不给出函数原型。

```
#define FALSE  0
#define TRUE   1
#define N      2                                    /* 进程数量 */

int turn;                                           /* 现在轮到谁？ */
int interested[N];                                  /* 所有值初始化为0（FALSE） */

void enter_region(int process);                     /* 进程是0或1 */
{
    int other;                                      /* 其他进程号 */

    other = 1 - process;                            /* 另一方进程*/
    interested[process] = TRUE;                     /* 表明所感兴趣的 */
    turn = process;                                 /* 设置标志 */
    while (turn == process && interested[other] == TRUE); /*空语句 */
}

void leave_region(int process)                      /* 进程：谁离开？ */
{
    interested[process] = FALSE;                    /* 表示离开临界区 */
}
```

图2-24 完成互斥的Peterson解法

在使用共享变量（即进入其临界区）之前，各个进程使用其进程号0或1作为参数来调用enter_region。该调用在需要时将使进程等待，直到能安全地进入临界区。在完成对共享变量的操作之后，进程将调用leave_region，表示操作已完成，若其他的进程希望进入临界区，则现在就可以进入。

现在来看看这个方案是如何工作的。一开始，没有任何进程处于临界区中，现在进程0调用enter_region。它通过设置其数组元素和将turn置为0来标识它希望进入临界区。由于进程1并不想进入临界区，所以enter_region 很快便返回。如果进程1现在调用enter_region，进程1将在此处挂起直到 interested[0]变成FALSE，该事件只有在进程0调用 leave_region退出临界区时才会发生。

现在考虑两个进程几乎同时调用enter_region的情况。它们都将自己的进程号存入turn，但只有后被保存进去的进程号才有效，前一个因被重写而丢失。假设进程1是后存入的，则turn为1。当两个进程都运行到while语句时，进程0将循环0次并进入临界区，而进程1则将不停地循环且不能进入临界区，直到进程0退出临界区为止。

5. TSL指令

现在来看需要硬件支持的一种方案。某些计算机中，特别是那些设计为多处理器的计算机，都有下面一条指令：

```
TSL RX, LOCK
```