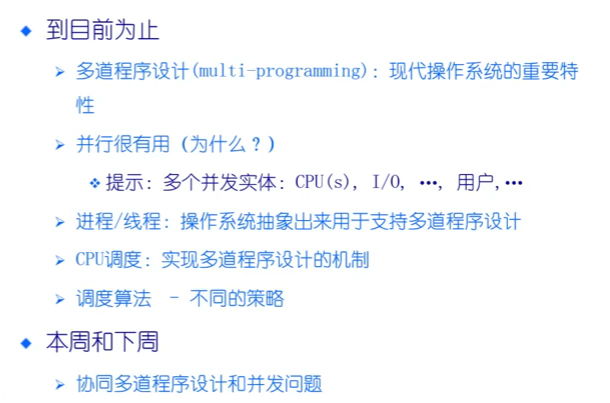
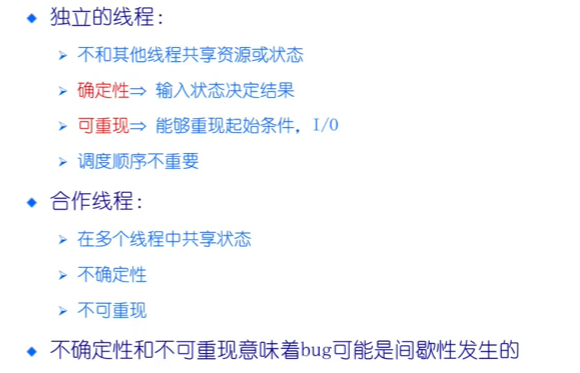
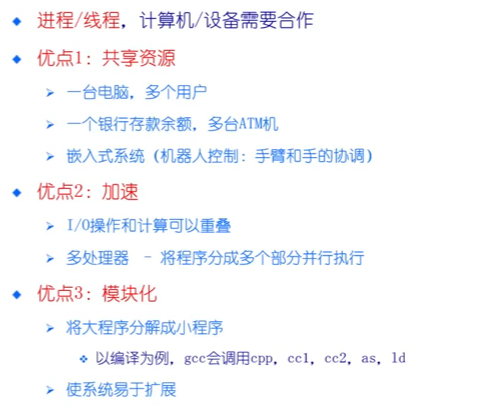
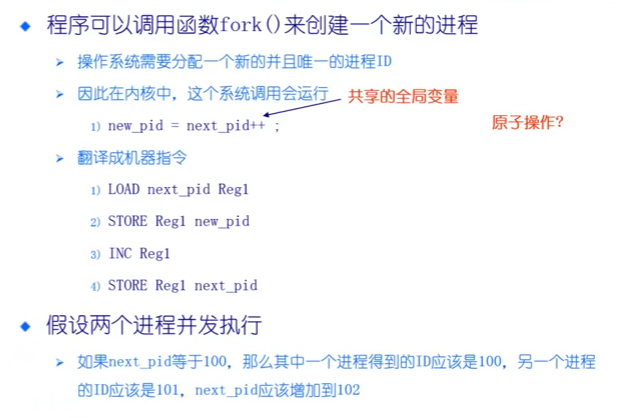
同步：

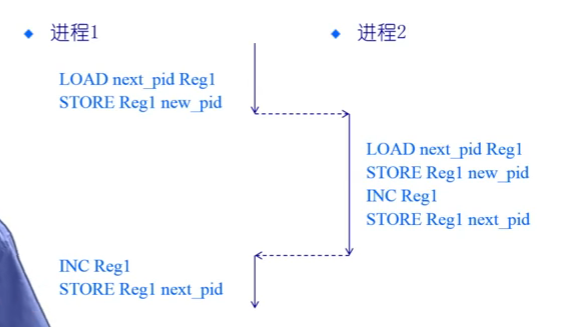








如果两个进程具有一定的并发性：



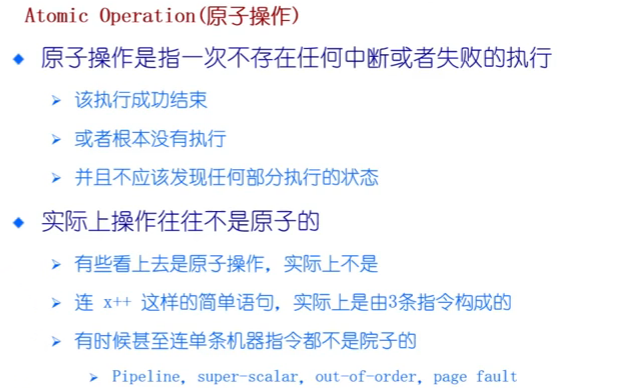
在进程1中切换到进程2，再切回来

就会出现两个进程的PID都是100，但是next\_pid为101（因为这是两个进程，其REG1不一样）（因为切换上下文，寄存器恢复的还是进程1的寄存器）

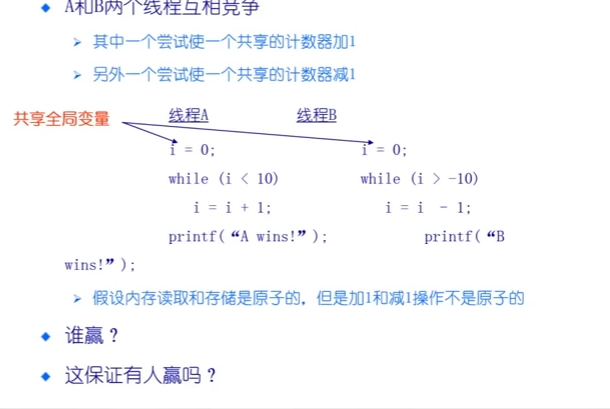
**（上述是一个典型的异常现象，汇编指令在执行时进行了上下文切换，并且调度点还可以在语句中的任何位置，最终结果具有不确定性和不可重复）**

也引入同步和互斥就是为了解决这种不确定性的问题。

保证指令的执行过程不会被打断和切换，即原子操作



在C语言的层次中：



Load是原子操作

会出现三种情况，A赢，B赢，谁都不会打印

添加锁机制

临界区：

有一段代码访问共享资源，访问共享资源的代码就是临界区

互斥：

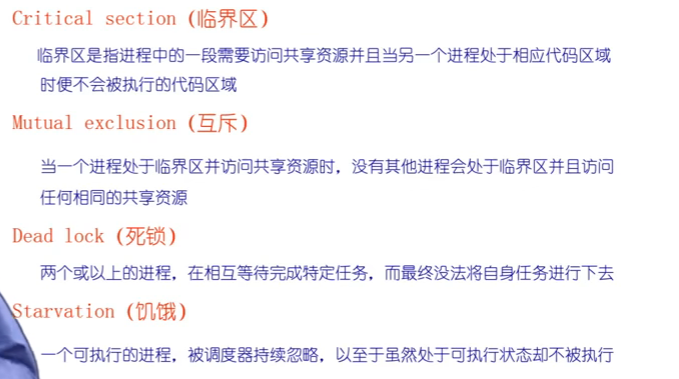
访问临界区的进程只有一个，不允许多个进程都进入临界区访问

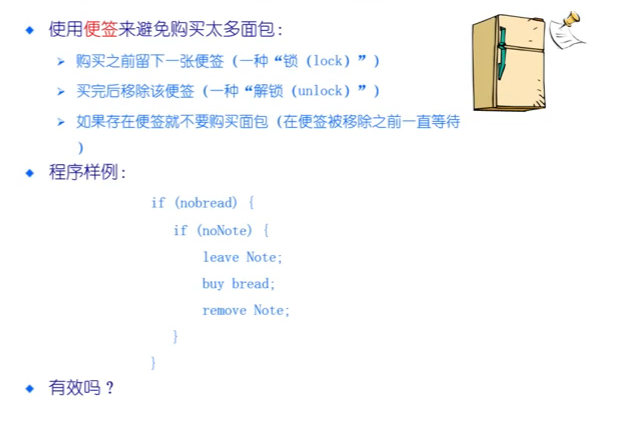
死锁：

当两个进程都拥有一定的资源，同时需要其他的资源时，可能会进入相互等待。进程a需要等待进程b的资源，而进程b也需要等待进程a。

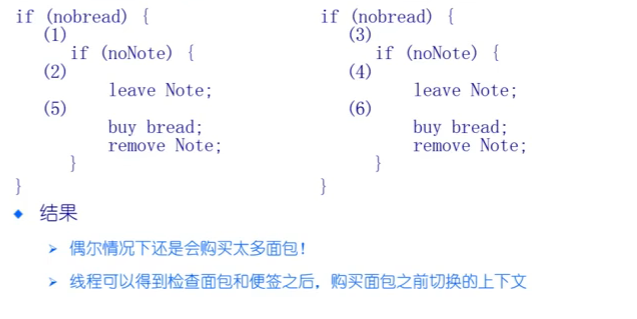
饥饿：

一个进程持续得不到执行，这个现象可能长期存在。





标签虽然实际有效，但是对于进程来说，上下文切换时



标签并有起到锁的作用。

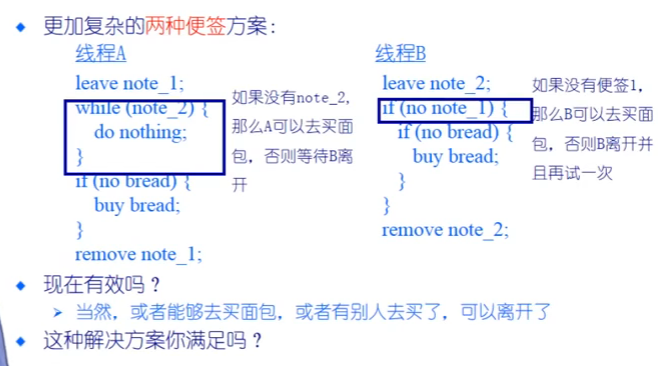
这是因为上下文切换的时机是无法确定的，很容易出现特殊情况。



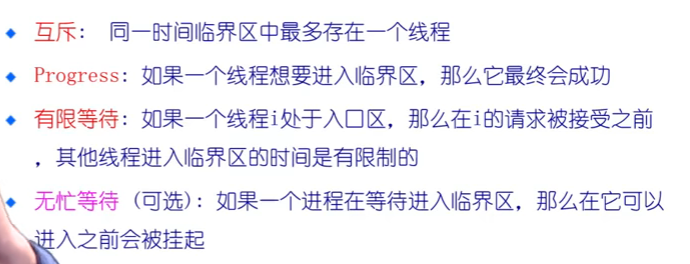
每个进程都给自己独特的标签，也不行，进程切换也有特殊情况不满足。



解决方案3：



进入临界区和离开临界区：



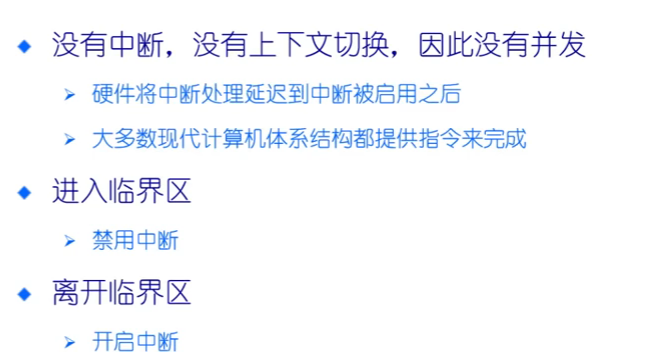
Progress：前进，进程不会一直死等，最终一定会进入。

无线等待就是饥饿状态。

实现临界区：



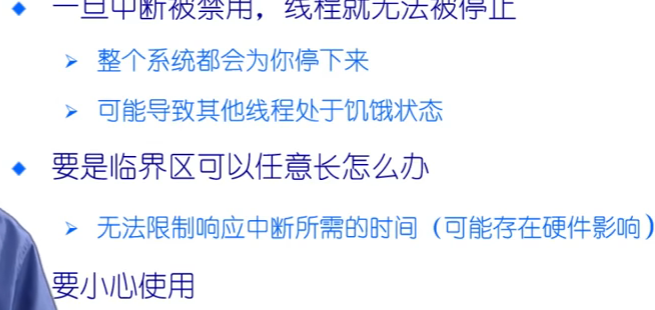
禁用硬件中断：



中断除了产生硬件事件响应之外，进程在进行切换时也需要一个时钟中断，强制打断进程，切换到操作系统完成进程切换。

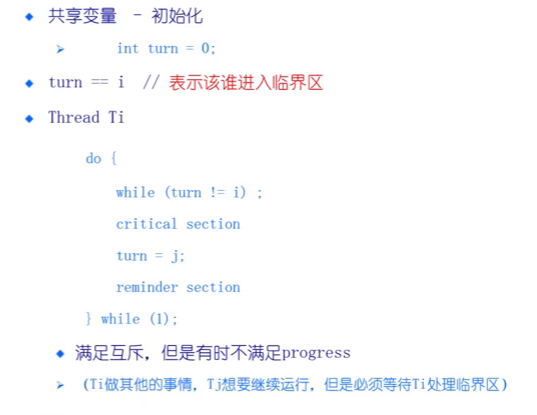
但是中断主要是用来响应外部事件，和外设交互，屏蔽中断会使硬件事件没有及时响应。

对系统效率影响很大。

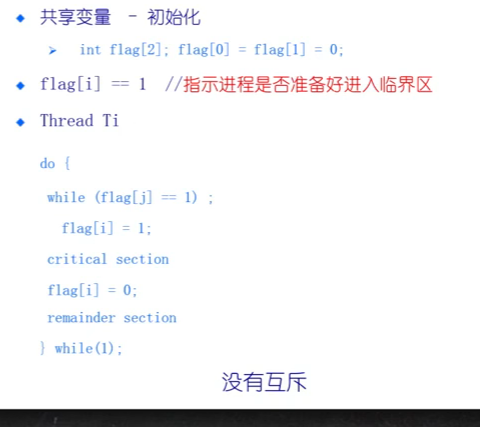


所以禁用中断要临界区很小，且多CPU下无法解决互斥。

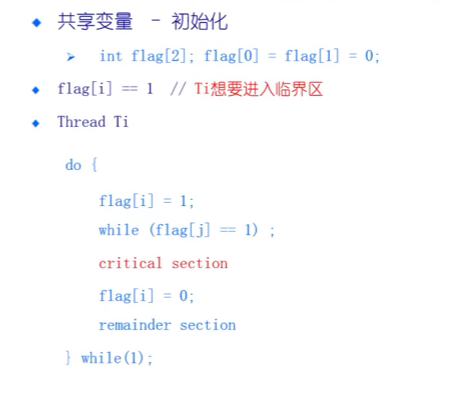
基于软件的解决办法：



满足互斥，但是不满足前进，只要有一个进程不想再访问临界区，其他进程都不能再进入临界区。



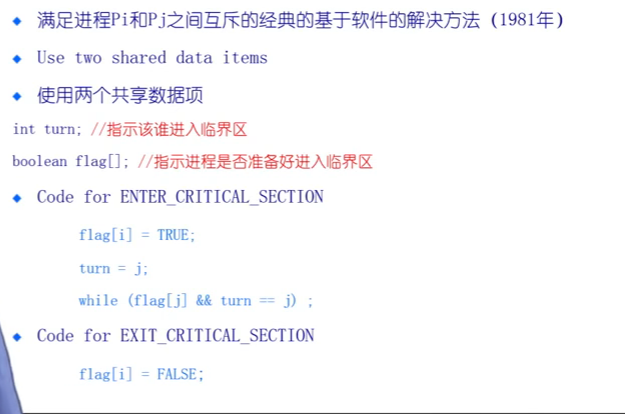
两个进程同时开始时，没有互斥



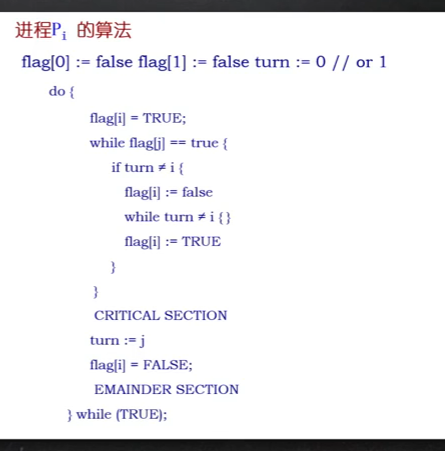
两个进程同时执行时虽然互斥，但是死锁了

正确的方法：

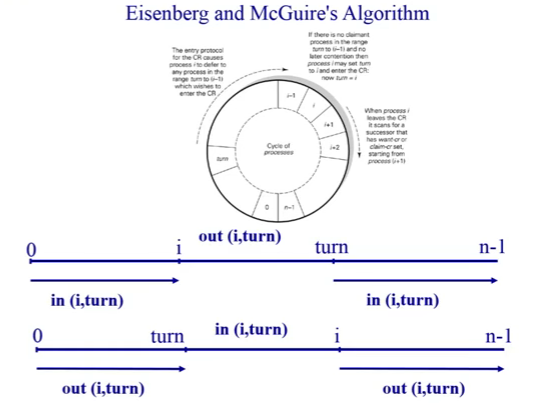
Peterson算法：



还有dekker算法：

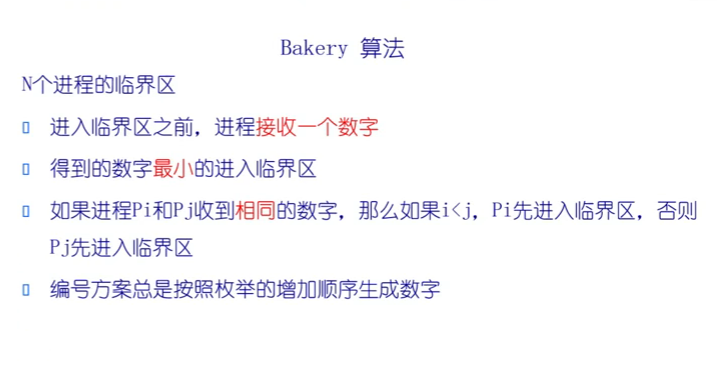


在peterson算法上实现n个进程的互斥的保护：

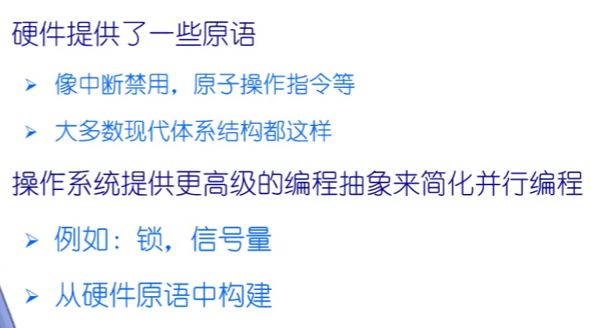


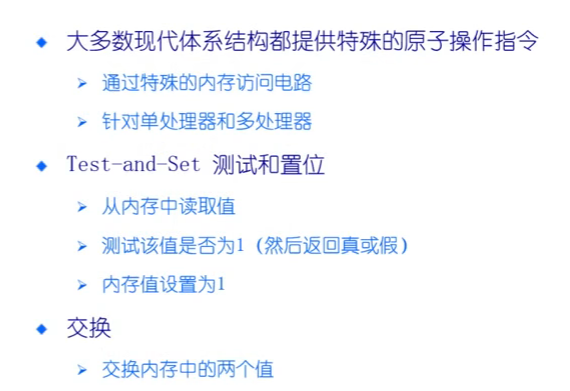
大致思路：对于进程i而言，如果前面的进程有想要进入临界区或者已经在临界区的，则进程i要等待，而对于进程i后面的进程，如果i想要进入临界区，则也需要等进程i结束之后才能再进入临界区。实现有序的循环实现n个进程进入临界区。

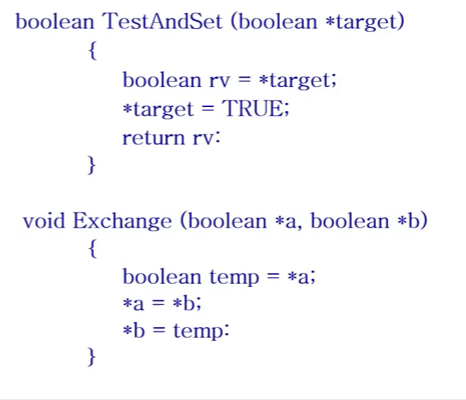
第二个方法：n个进程进入临界区



更加简介的硬件的高级抽象的实现进入临界区：

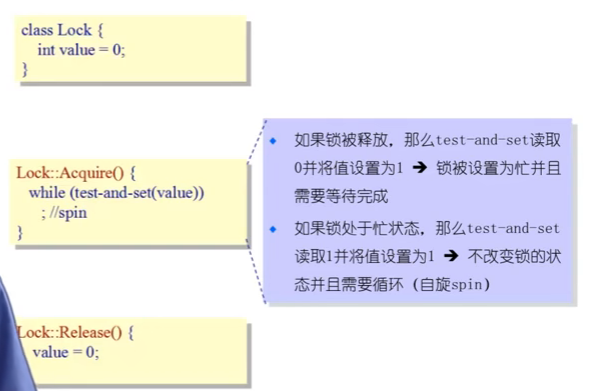


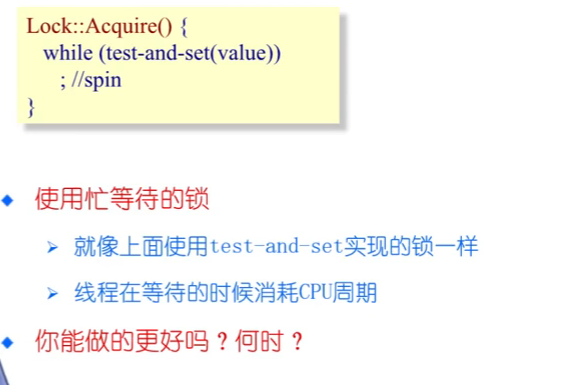




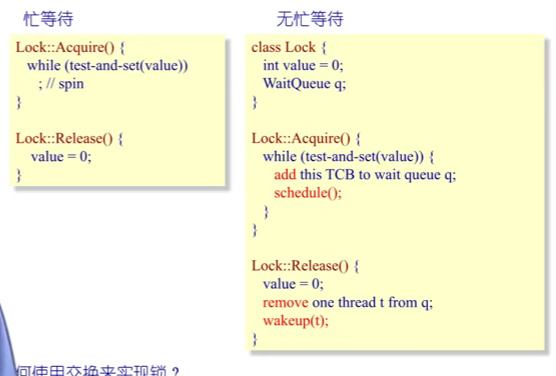
这两个指令已经被封装成了一条机器指令，即执行过程中不允许被打断

这两条指令只要有一条计算机提供了，就能实现临界区的操作：



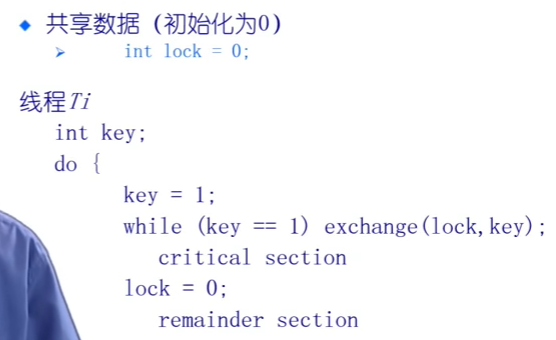


我们可以看到此时如果别的进程在临界区时，别的进程是忙等的。



可以使用进程管理，当忙等时，让其睡眠，阻塞，挂到等待队列中去。

Exchange指令进入临界区的实现：



以上基于原子操作指令的缺点：

