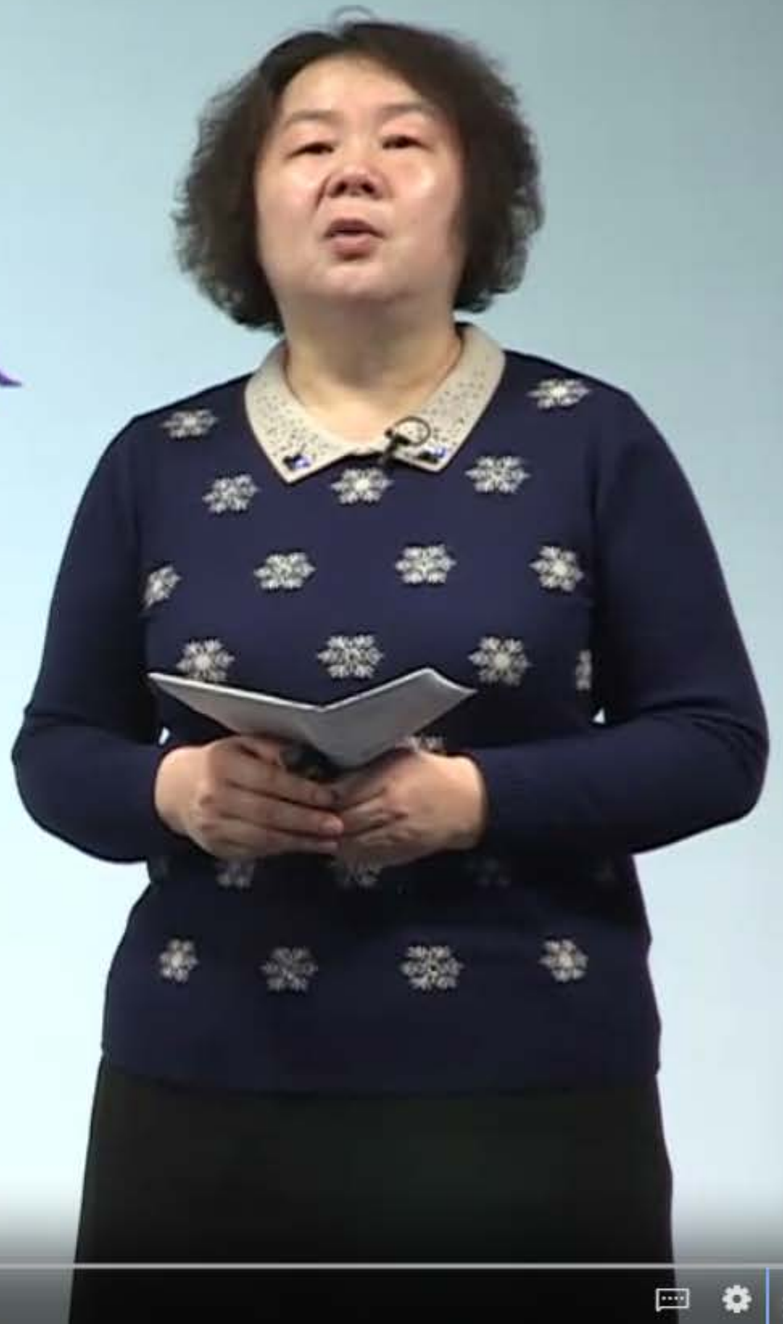
中断屏蔽、TSL指令、XCHG指令

进程互斥的硬件解决方案



硬件解法1—中断屏蔽方法

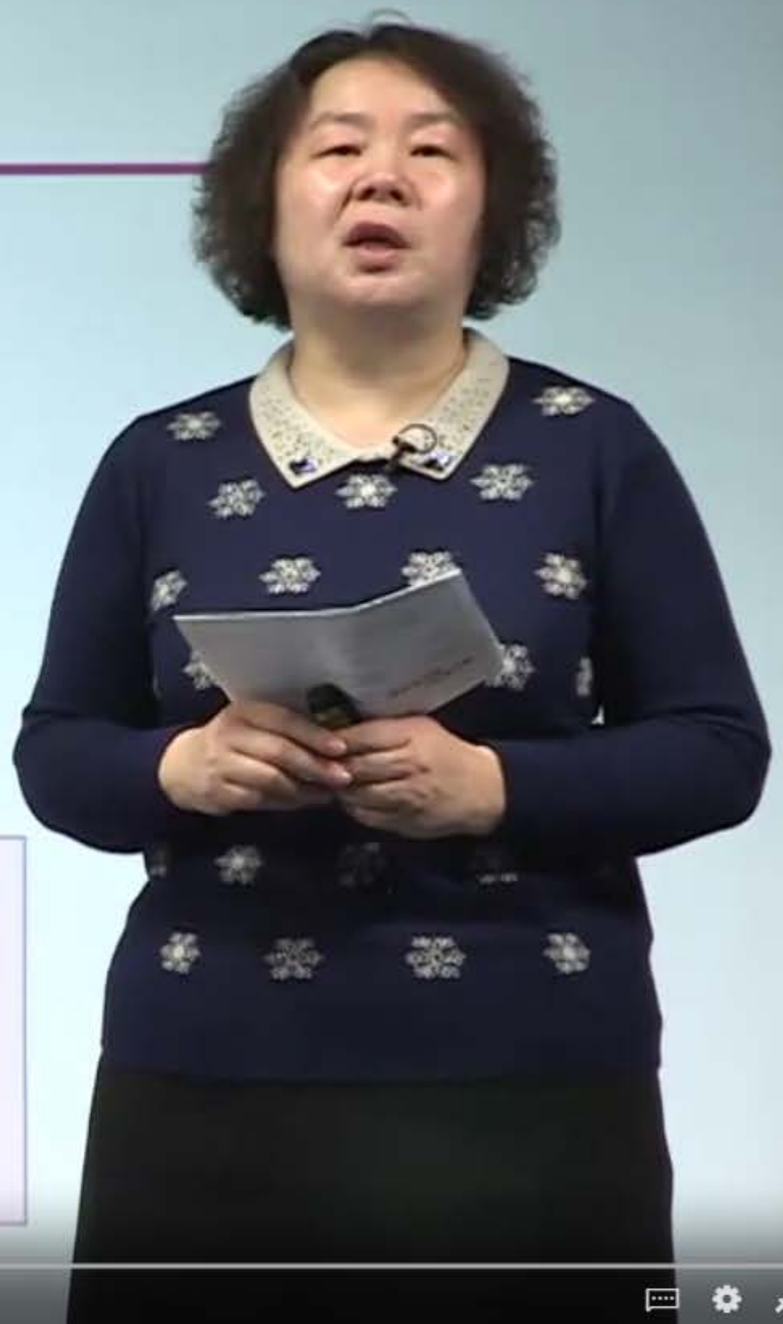
“开关中断”指令

执行“关中断”指令

临界区操作

执行“开中断”指令

- 简单，高效
- 代价高，限制CPU并发能力（临界区大小）
- 不适用于多处理器
- 适用于操作系统本身，不适于用户进程



硬件解法2

—“测试并加锁”指令

TSL指令：TEST AND SET LOCK

```
enter_region:
        TSL REGISTER,LOCK
        CMP REGISTER,#0
        JNE enter_region
        RET
```

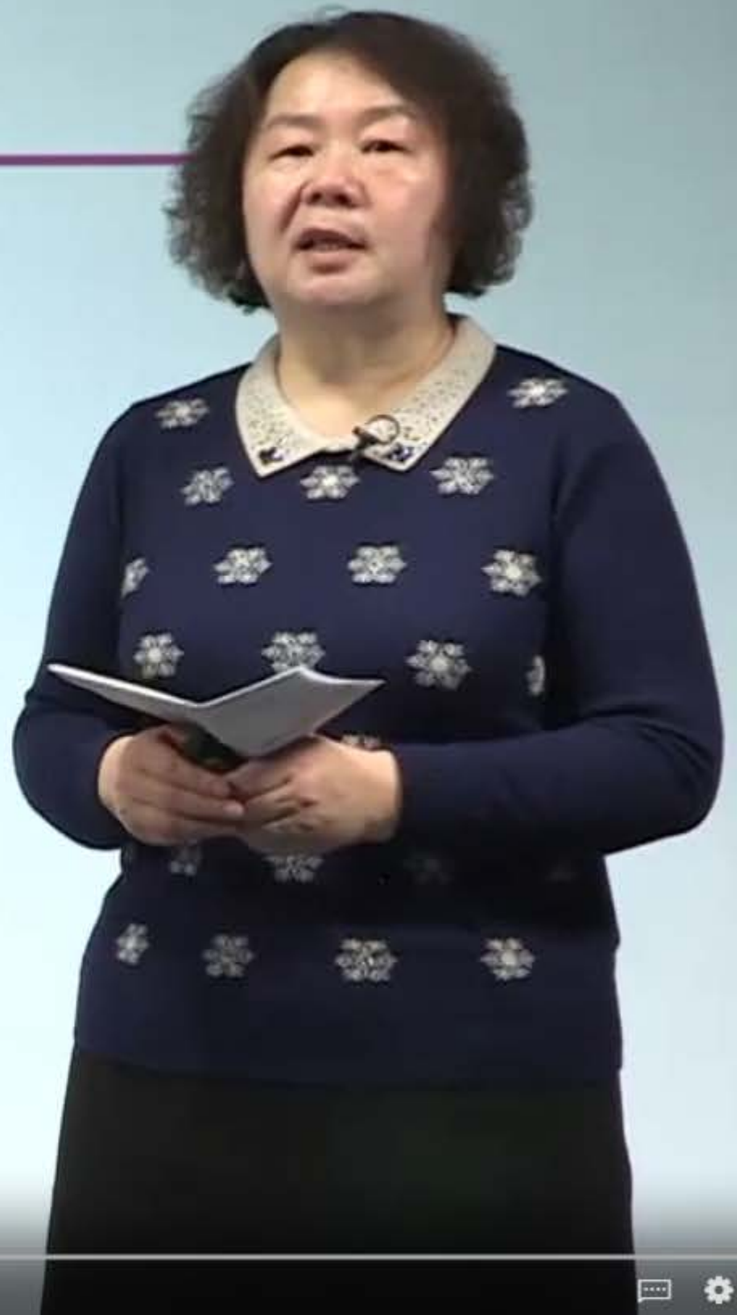
复制锁到寄存器并将锁置1
判断寄存器内容是否是零?
若不是零，跳转到enter_region
返回调用者，进入了临界区

循环

```
leave_region:
        MOVE LOCK,#0
        RET
```

在锁中置0
返回调用者

提问：对多处理器系统有效吗？为什么？



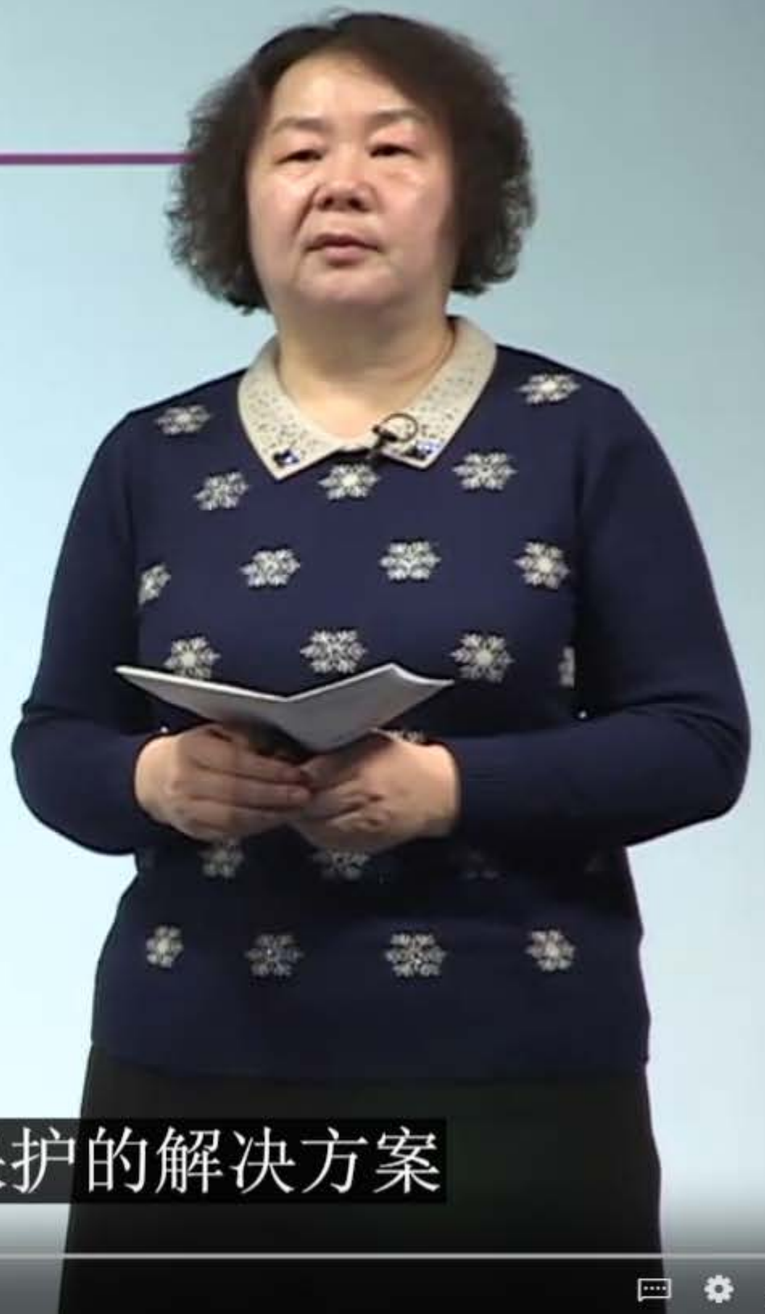
硬件解法3—“交换”指令

XCHG指令：EXCHANGE

```
enter_region:
        MOVE REGISTER,#1          | 给寄存器中置1
        XCHG REGISTER,LOCK        | 交换寄存器与锁变量的内容
        CMP REGISTER,#0           | 判断寄存器内容是否是零?
        JNE enter_region          | 若不是零，跳转到enter_region
        RET                       | 返回调用者，进入了临界区

leave_region:
        MOVE LOCK,#0              | 在锁中置0
        RET                       | 返回调用者
```

循环



下面我们小结一下 各种各样的对临界区的保护的解决方案

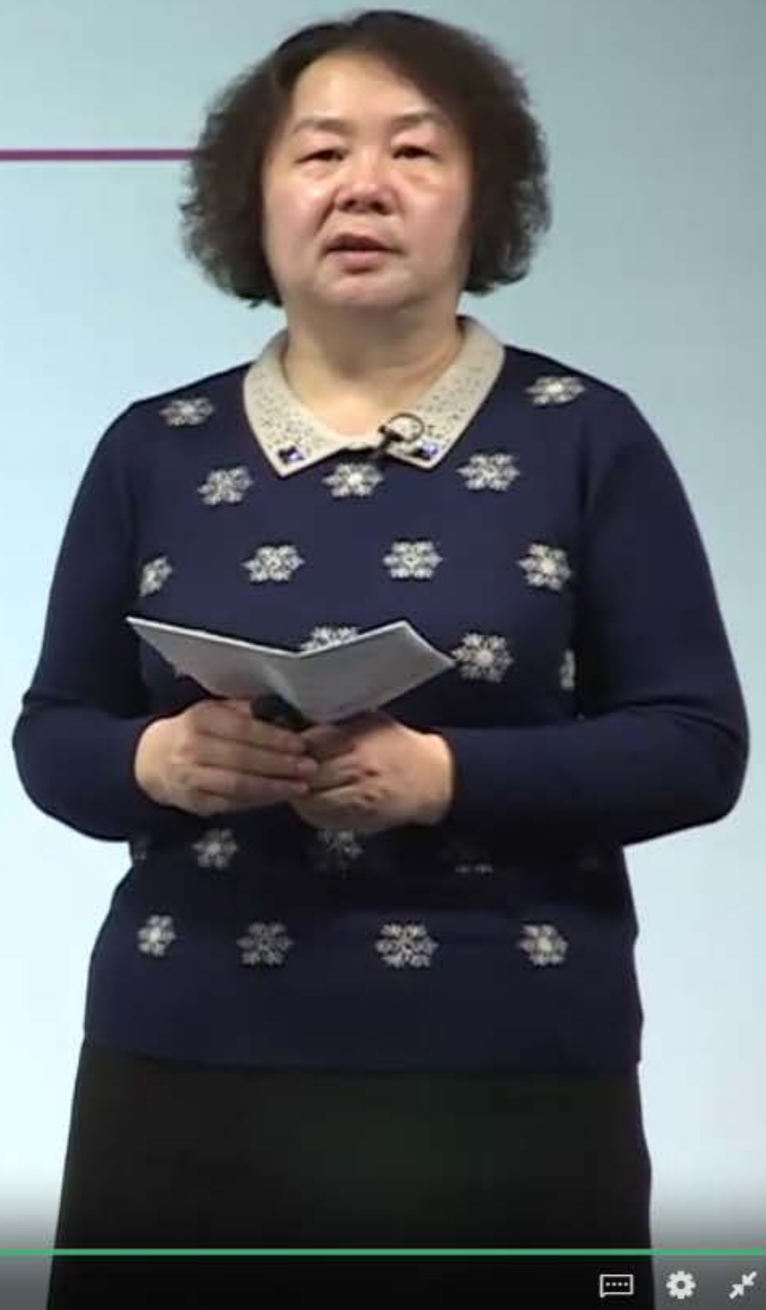
小结

- 软件方法
  - 编程技巧
- 硬件方法
- 忙等待(busy waiting)

进程在得到临界区访问权之前，持续测试而不做其他事情

自旋锁 Spin lock （多处理器 √)

- 优先级反转（倒置）



0:00 下面呢，我们介绍一下进程互斥的硬件解决方案 之所以叫硬件解决方案呢，其实 是说我们通过一些指令，来完成这项工作 首先我们来看一下第一种解法，也是非常常用的一种解法 就是利用一条指令，叫做开关中断指令，啊，这是一个俗称，实际上是 允许中断或禁止中断，啊，这样一条指令 也就是在进临界区之前，我们先把中断关闭，屏蔽掉 禁止，然后就进入临界区做相应的操作 出临界区的时候呢，再把中断打开，也就是允许中断 那么我们知道，实际上是原语操作 都是用这样一种方式来实现的，所以临界区的保护也可以用开关中断这样一个指令来实现 这种实现方法呢，非常的简单，非常的高效，啊，非常的高效 但是它的代价呢，也很高。 所谓代价高呢，就是说 你保护的这个区域，也就是临界区的区域 如果设置的比较大，范围划定的比较大 那么就会带来对 CPU 的这种并发能力的一个什么呀，限制 就是有些指令执行的时候就没有并发执行。 当然了这个是其中一个 缺陷，那么还有呢，就是说开关 中断这个指令，只适合于用于单处理器 它不适用于多处理器，因为 开关中断或者是禁止或者允许中断只针对一个处理器 它对其它处理器是没有效果的。 也就是换句话说 当一个 CPU 可以是不接收中断 那并不意味着其它 CPU 不可以接收中断，啊，这个所以在多处理器情况下 不适用这种方法。 最后呢，我们来看，其实这种方法呢 虽然很好用，也很简单，但是用户程序是不能用的，用户进程是不能使用这个特权指令的 因此，用户程序它不能够直接在代码里头是允许开中断 或者关中断，这是不允许的啊。 所以只能够操作系统来用，所以操作系统 很多地方都是采用这种方法来达到目的的 硬件的第二种解法呢 是有用一条特殊的指令叫做，测试并加锁这条指令 测试并加锁这条指令呢，实际上是这一条指令 做了两个事情，一个呢是先读，啊，一个内存单元内容 读到寄存器，然后再去写，把内存单元的内容写上某个值 啊我们来看一下，啊，这条指令的一个使用 那么 TSL 这条指令，那么使用的过程中呢，是让复制锁到 寄存器，然后呢并且把锁置成 1，这个 Lock 呢是一个内存单元 把这个内存单元的值，不管它前面是什么值，把它内容先 复制到了寄存器，然后再把这内存单元的值设置成 1 那么就是说原来这个内存单元 原来要如果是 0 了呢，那么寄存器里内容就是 0。 如果原来已经是 1 了，也就说已经上了锁了呢，那么这 1 也就在寄存器里头 那么下面呢，我们接着来看，判断一下寄存器的内容是，是不是 0，啊是不是 0 如果是 0 说明原来的是没上锁的，那你要想上锁就把它锁上就 可以了，你锁上就可以进入临界区了。 所以如果是 0 那就直接返回调用者，那么就进入临界区了。 但是如果不是 0 那就要去再次做测试，所以要反复测试，因此我们看

到如果不是 0 就跳转到这样一个标号的地方，然后呢重新再做刚才的事情 好，这就是 TSL 指令实现的一段 代码，这段代码呢可以相当于加锁这样一个功能 那么我们说就是说，这件事情从 内存单元读一个值到寄存器，再往内存单元写 那么这件事情是怎么完成的呢？实际上是通过 把总线封锁，就把总线锁住 把总线锁住之后，那么 先读后写的这个操作都完了，再把总线打开 因此，这条指令呢，是在 总线一级上做了一些工作，然后达到了这个结果 那这里头呢，也看到了一个循环，也就是说 如果不是 0，也就是原来是上锁的，那么 就跳到了这个，一个标志的地方，然后再去判断是不是 0 再去判断是不是 0，所以这也是一个 busy waiting，也是一个反复循环的过程，直到 这个单元变成 0 为止 那么这种解法能不能用于多处理器呢？ 对多处理器是不是有效呢？ 那么大家呢去查有关资料呢，给出你的答案 第三种解法 实际上是用了另外一条指令，交换指令 交换指令的作用是把两个位置 可能是寄存器或者是内存单元，只是两个位置的内容呢 在一条指令结束的时候，把两个位置进行 一个交换，啊，内容进行一个交换。 所以我们来看一下，我们 也是实现一个加锁这么一个过程，首先呢先给寄存器当中设置成 1 然后就通过交换，交换寄存器和锁变量 通过这个交换，就把这 1 设置给锁变量了，那么原来锁变量里头是 0 和 1 的值 或者 1 的值，就送到了寄存器，然后再接着去判断，跟前面 TSL 指令是一样的 那么这是加锁 解锁呢，实际上就是一个 move 语句 把这个内容复制成 0 就可以了 啊，这就是解锁。 所以上面呢我们就介绍了三种硬件的解决方案 那么所谓硬件解决方案就是用指令 特殊的一些指令来达到保护临界区的目的 下面我们小结一下各种各样的对临界区的保护的解决方案 那么软件的解法呢，主要是它带来了一些开销 啊，因为你要想进入临界区之前，先要做一系列的判断 带来一些开销，同时呢，对编程的技巧 要求比较高，因此呢一般的开发人员呢是不愿意，啊，碰 这样的一些事情的。 不愿意做这些事情，因为要考虑各种各样的情况 如果考虑不周到，可能还会出现一些逻辑性的错误，很难查出来 这是软件解法。 当然软件解法当中 Dekker 算法和 Peterson 算法是一个正确的解法 硬件解法呢，主要是通过指令 像开关中断的指令，测试并设置加锁这样的指令，或者交换的这个指令，可以解决相应的保护问题 那么在软件的解法当中，和硬件解法当中，我们看到了，多次看到了这样一个循环的存在 而这个循环呢，是不断的来测试 测试这个锁是不是打开了，或者是测试这个条件是不是成立 因此呢，它是在 CPU 上，在那儿一直在做测试，所以呢这种

测试呢我们称之为忙等待 啊，忙等待。 所谓忙等待就是，进程 在得到临界区的访问之前，持续做测试 不做任何其他的事情，那么浪费了 CPU 的周期 那么忙等待呢，在单 CPU 的系统当中肯定不是一个好的解决方案 因为如果你占着 CPU 那么其它进程上不了 CPU，是没办法 把相应的资源还回来的，所以 在单处理器系统中，我们应该把这种方案给它 这个抛弃。 可是，啊，事情总是在发展着 到了多处理器的时候，由于 有多个处理器，所以某一个内存单元，它的值 从 1 变成 0。 那么可以由另外一个处理器 CPU 来完成这项工作 所以，这个 CPU 某一个进程 上了一个 CPU，想要进入临界区，想要加锁 在这个锁或临界区的访问权没有得到之前，它可以一直在保持测试 但是呢，其它的 CPU 会去 运行相应的进程，去把这个锁打开或者是把临界区，啊，还回来 那大家说为什么不，等，它也 不做测试了呢？呃，它去下 CPU 行不行呢？那么我们知道 切换实际上呢是会带来开销的，因为 临界区的使用应该是个很短的时间，所以很快就会有进程把临界区让过来 啊让出来，所以在这种情况下呢 一个进程在某一个 CPU 上持续做测试 这是允许的，因为切换的开销比这个还要大 啊，所以呢，在多处理器情况下呢，又把忙等待这种思路呢 又引进来，那么就形成了一种锁，特定的锁 叫做自旋锁，啊，Spin lock 这个锁就是说如果我想得到临界区使用权，那么我首先要判断是不是有进程在临界区 如果有进程在临界区，那我就在那里自旋，啊，在那儿不断地去测试，直到 这个临界区还回来，啊，这是关于一个特定的解决方案，叫自旋锁 在多处理器的情况下，这个自旋锁是常用的一种进入临界区的一个方法 那么临界区的问题呢 还会带来我们刚才已经介绍过的优先级反转 我们在讲进程调度的时候，基于优先级的抢占式的调度 可能会导致优先级反转现象的产生 而这个反转现象的产生呢，其主要的一个原因呢，可能就是临界区的使用，啊 临界区的使用，所以它也会由于临界区的这个保护，而带来的一个优先级反转问题 这和前面我们所讲的这种进程调度算法的 某种现象呢，其实是相吻合的