时间片轮转、最高优先级、……

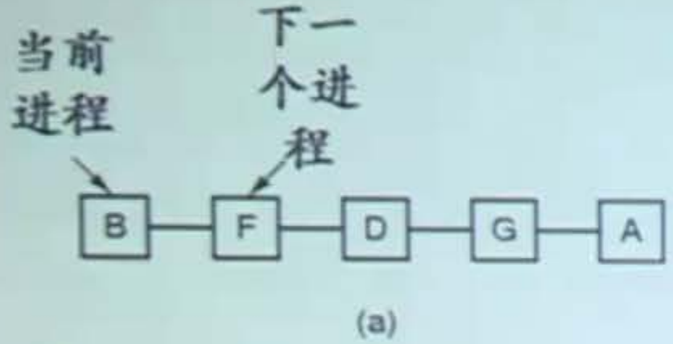
交互式系统的调度算法

交互式系统中采用的调度算法

- 轮转调度（RR-Round Robin）
- 最高优先级调度（HPF—Highest Priority First）
- 多级反馈队列（Multiple feedback queue）
- 最短进程优先（Shortest Process Next）

那么最短进程优先调度算法呢和短作业优先调度算法差不多，我们就不再介绍了

时间片轮转调度算法(1/4)



(a)

(b)

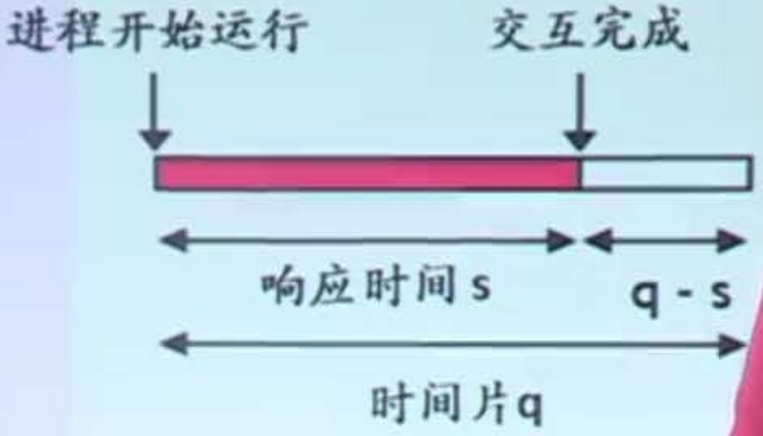
进程B用完自己的时间片后

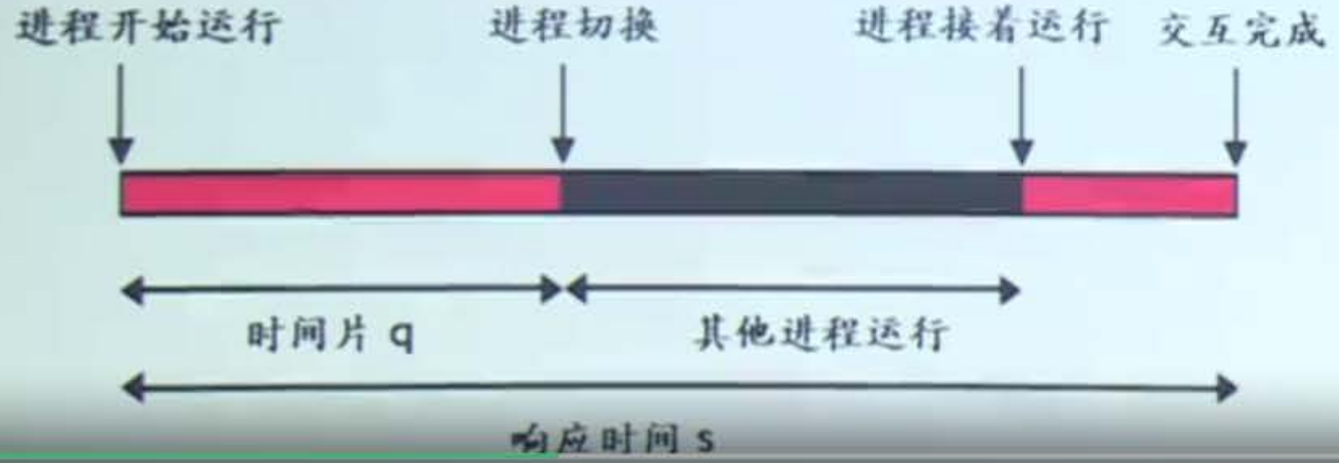
- 目标
  - 为短任务改善平均响应时间
- 解决问题的思路
  - 周期性切换
  - 每个进程分配一个时间片
  - 时钟中断一轮换

改善短作业的这个平均响应时间的这个目标

时间片轮转调度算法(2/4)

- 如何选择合适的时间片?
  - 太长--大于典型的交互时间
    - 降级为先来先服务算法
    - 延长短进程的响应时间
  - 太短--小于典型的交互时间
    - 进程切换浪费CPU时间





时间片轮转调度算法(3/4)

- 优缺点
  - 公平
  - 有利于交互式计算，响应时间快
  - 由于进程切换，时间片轮转算法要花费较高的开销

假设时间片 10ms，如果进程切换花费0.1ms，CPU开销约占1%

进程运行时间	时间片	上下文切换次数
10	12	0
10	6	1
10	1	9

这是关于时间片轮转算法当中的这个时间片 的大小会带来切换的开销的讨论。

时间片轮转调度算法(4/4)

- 优缺点（续）
  - RR对不同大小的进程是有利的
    但是对于相同大小的进程呢？

- 两个进程A、B，运行时间均为100ms
- 时间片大小为1ms
- 上下文切换不耗时

假设

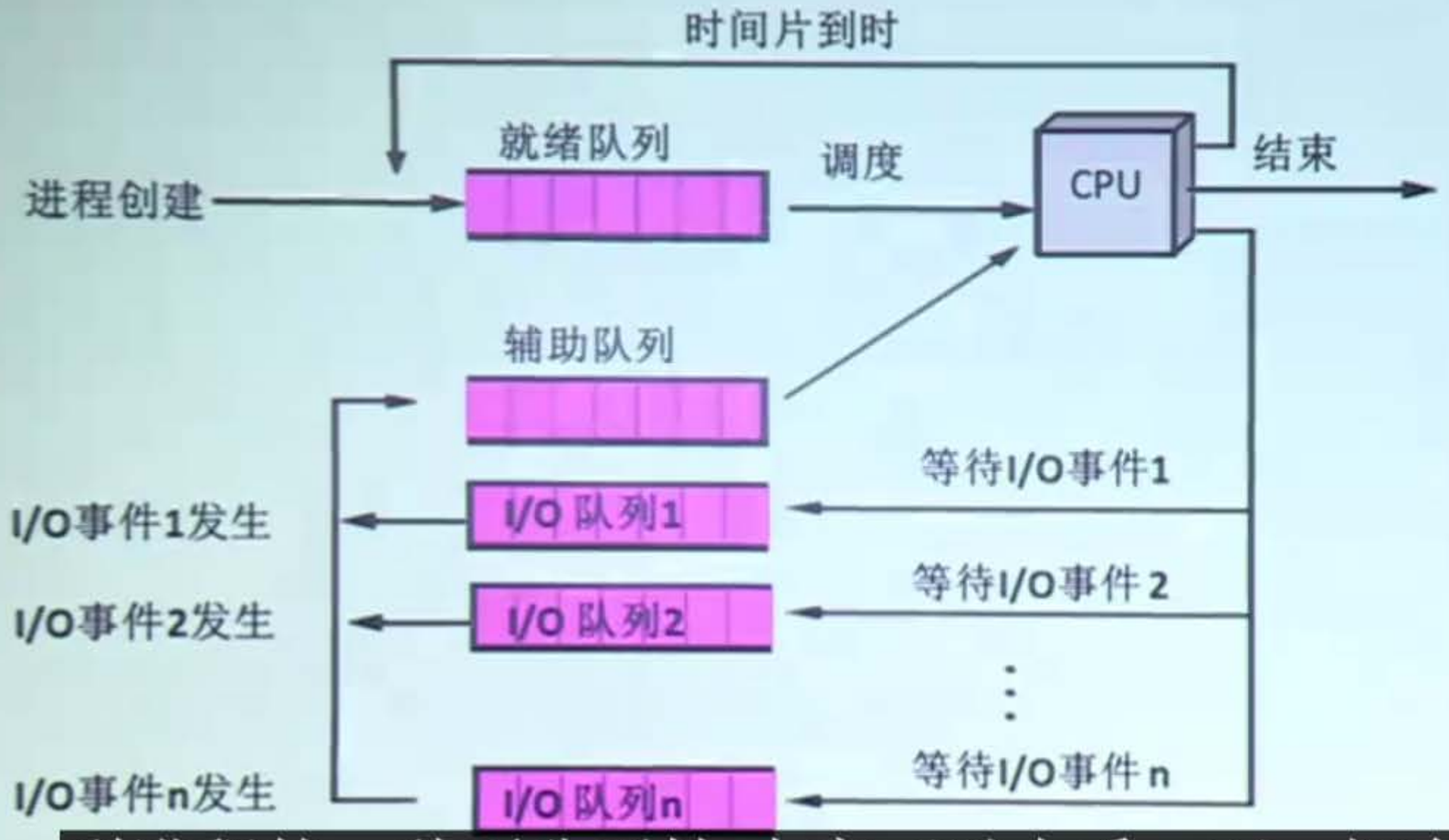
➢ 使用时间片轮转（RR）算法的平均完成时间？

199.5ms

ABABABAB…… …… …… ……A(199)B(200)

➢ 使用先来先服务（FCFS）算法呢？ 150ms

虚拟轮转法(VIRTUAL RR)



型进程的一种不公平性 大家可以去看一下，挺有意思的一个解决方案

虚拟轮转调度的队列图

Halder,S. etc. 1991

最高优先级调度算法

- 选择优先级最高的进程投入运行

- 通常：系统进程优先级 高于 用户进程
  前台进程优先级 高于 后台进程
  操作系统更偏好 I/O型进程
- 优先级可以是静态不变的，也可以动态调整
  - 优先数可以决定优先级
- 就绪队列可以按照优先级组织
- 实现简单，不公平

饥饿

因为它会导致优先级低的进程产生饥饿现象 下面我们来介绍一下

优先级反转问题(1/2)

基于优先级的抢占式

- Priority Inversion
- 又称：优先级反置、翻转、倒挂
- 现象

一个低优先级进程持有一个高优先级进程所需要的资源，使得高优先级进程等待低优先级进程运行

设H是高优先级进程，L是低优先级进程，M是中优先级进程（CPU型）

场景：L进入临界区执行，之后被抢占；

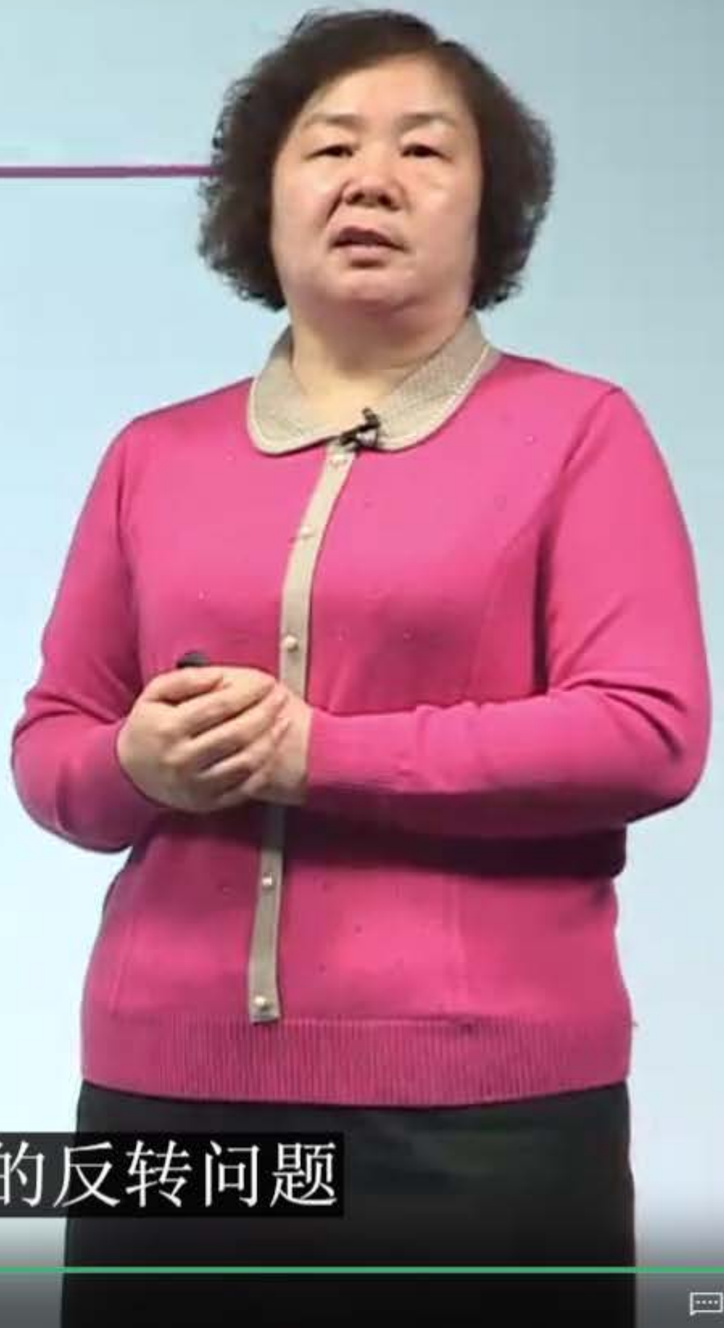
H也要进入临界区，失败，被阻塞；

M上CPU执行，L无法执行所以H也无法执行

优先级反转问题(2/2)

- 影响
  - 系统错误
  - 高优先级进程停滞不前，导致系统性能降低

- 解决方案
  - 设置优先级上限
  - 优先级继承
  - 使用中断禁止



所以呢这三种方案都可以解决优先级的反转问题

0:00 下面呢我们来介绍一下交互式系统当中所采用的一些调度算法 那么主要包括了时间片轮转 最高优先级调度，多级反馈队列 那么最短进程优先调度算法呢和短作业优先调度算法差不多，我们就不再介绍了 在交互式系统当中呢，追求的指标主要是响应时间 还有一些像公平啊，资源的平衡使用啊这样一些指标 首先我们先介绍一下时间片轮转调度算法 那么我们现在看在当前正在运行的呢是 B 进程 排在它后面的呢是一个 F 进程 当 B 进程用完它的时间片之后，它就回到了 队列的末位，那么调度呢就选择 刚才的下一个进程，现在呢是当前运行进程 F 上 CPU 这就是一个时间片轮转，然后 F 运行完它的时间片它就继续去排队 那么这个队列里面的每一个进程都有机会轮流上 CPU，然后每次呢 上一个时间片。 因此 时间片轮转调度算法，它的主要目标就是改善了 短作业或者短任务它的平均响应时间 具体的做法呢就是通过周期性地切换 然后每个进程分配一个时间片，通过了时钟中断 然后引发了这个轮换，来达到 改善短作业的这个平均响应时间的这个目标 这是时间片轮转算法。 那么我们 回答前面已经提出的问题，如何选择一个合适的时间片？ 时间片如果太长，大于 典型的交互时间，我们来看一下这张图 那么这个进程开始运行了，给了它这么长的一个时间片 它运行没有用完时间片，还差一部分 就已经完成了它的交互，已经完成了一个响应 如果系统中所有的进程或者绝大部分进程 都不到一个时间片就可以完成响应的话，那么 时间片轮转算法实际上已经退化成了一个 先来先服务的这种算法 另外如果时间片过长 如果系统中有比如说 50 个进程，那么每个进程我时间片比如说给了是 100 毫秒 那么最后一个进程要等 5 秒钟才能轮到它上 CPU 因此呢会延长某些进程的响应时间 响应时间变长了，这样不太好 好，这是时间片如果太大会带来的问题 那么时间片如果太短呢，如果短得小于 典型的一个交互时间，就是大部分进程的交互时间都，一个时间片都完成不了 我们来看这样一个图，那么 某个进程给的时间片很短，那么一个进程呢它需要这么长的时间来响应 那么在中间会切换其他进程上 CPU 所以它的响应时间就会变长，变长 这个呢也是，如果时间片太短，响应 时间也会变长，因为它会频繁地去切换 那么还有一个呢就是切换本身会带来开销 刚才前面我们已经讲过了，上下文切换是有开销的，是要花时间的 所以，切换带来的开销呢也就浪费了 CPU 的时间 这是时间片轮转算法当中如何来选择时间片 要考虑的问题。 那么典型的时间片的大小呢，这有一些经验值，那么大概在 10 毫秒到 100 毫秒之间，通常可能是五六十毫秒这个样子 我们来讨论一下 时间片轮转算法的优缺点，当然了，首先它是公平的，因为大家都轮流 上 CPU，有机会上 CPU，都用，都分配给同

样的时间片 这样的话呢，有利于交互式的计算，让它响应时间加快 但是呢它又会带来其他的问题 由于有进程的切换，所以呢会浪费一些 时间用于切换，所以带来一些开销 我们来看一个小例子啊。 如果一个进程 10，10 个单位运行完 10 个单位运行完。 那如果时间片给的是 12 个单位 那么不需要发生切换，这个进程就运行完了 如果时间片给的是 6 个单位，那么在中间要切换一次 如果时间片很小， 1 个单位 那么运行完这个进程中间要有 9 次切换。 那么当然了这个开销就可能会很大 那么到底时间片多大为好呢？那么看 这样一个例子，就是说如果时间片呢是 10 毫秒，如果切换的时间 只花 0.1 毫秒，仅占这个时间片的 大概 1% 左右，那么这样的话呢，我们觉得这种代价相对来讲还比较 合算，也就是 CPU 的这个时间呢，主要还是花在执行进程上，在切换上花的时间并不多 这是关于时间片轮转算法当中的这个时间片 的大小会带来切换的开销的讨论。 那么下面我们再来看一下 时间片轮转算法，对于不用大小的进程呢是有利的 有长有短，那么短的就可以更多地、 更快地执行完 因为它这是轮流执行，但是对于那些 大小相同的进程，就是大家执行时间差不多的进程 如果都是出现这样的进程，那可能就不利了。 我们来看一下 首先我们先假设啊，有两个进程 A 和 B，它们都分别要运行 100 毫秒，那么如果时间片大小给的是 1 毫秒 那么上下文切换呢，假设我们 没有耗费时间，就它的这个开销忽略不计，假设 如果采用时间片轮转算法 那我们算一下它的平均完成时间。 好，我们来看一下 A 运行 1 毫秒， B 运行 1 毫秒 A 运行 1 毫秒， B 运行 1 毫秒，不断地切换。 到最后 A 进程运行完已经是 199 毫秒的时候了 B 进程运行完呢已经是 200 毫秒的时候了 所以它们的平均的完成时间呢 是在 199.5 毫秒，这个数量级 如果我采用先来先服务调度算法呢？ 大家可以看到， A 执行完 100 毫秒 B呢进程执行，它呢先等了 A 执行 100 毫秒 等了 100 毫秒，然后再执行自己的 100 毫秒，所以花 300 毫秒 就两个进程都执行完了，平均一下就是 150 毫秒来执行，平均完成时间 150 毫秒 所以呢对于 A 和 B 两个进程来讲呢 得到是这样一个结果。 所以我们要了解一下时间片轮转算法，它有这样一个特点 那么时间片轮转算法呢，往往啊 对于不区分你是 I/O 型进程呢，还是 CPU 型进程 但是呢这样呢会给 I/O 型进程带来一定的不公平，我们来看一下 那么我们来看这张图 当一个 CPU 型进程被调度上 CPU 之后 它用完了它的时间片，然后重新排队 下一次再调度上 CPU 又用完一个完整时间片。 那我们来看看 I/O 型进程， I/O 型进程被调度上 CPU 之后 它没有运行完它的

时间片，可能运行的很短，然后就去等待 I/O 去了 因此它就放弃了 CPU ，进入了等待队列 那么一旦等待的结果到来，所以它就又变成一个就绪 当它上了 CPU，当它进了就绪队列，再次上 CPU 那么它又没有用完分配给它的时间片 那么这样的话，感觉 CPU 型的进程 总是用完给它的时间片。 所以它占用了更多的 CPU 时间 而 I/O 型进程总是用不完它的时间片，所以呢它有些 对这个调度算法对它有些不公平，它用的总是很少，然后又去重新排队 那么怎么样能解决这个问题呢？ 啊，有一个研究结果。 我们来看一下。 它是这么来设计的 当一个 I/O 型进程，让出 CPU 进到等待队列 从等待队列又重新回到就绪状态的时候 不去进入原来的就绪队列，单独为它设置一个队列，叫做辅助队列 也就是所有 I/O 型的进程呢，从等待变成就绪的时候呢，进到这个队列 那么调度算法在选择进程的时候呢，首先先从 这个辅助队列里去选择进程，选择了 I/O 型进程 那么 I/O 型进程上 CPU 之后，又很快地放弃了 CPU 进入等待 直到辅助队列为空 那么才去从就绪队列里头去选进程 那么试验的结果呢，其实表明这种 设计方案呢，改善了这个对 I/O 型进程的一种不公平性 大家可以去看一下，挺有意思的一个解决方案 下面我们讨论一下最高优先级调度算法。 它的思想比较简单 总是选择优先级最高就，呃，去执行 那我们之前 也讨论了进程的各种分类。 我们来看一下，啊 系统进程和用户进程，系统进程的优先级往往高于用户进程 那么前台进程和后台进程，前台进程的优先级往往高于后台进程 I/O 型进程和 CPU 型进程，那么作为操作 系统的调度算法来讲，更偏好 I/O 型进程 那么基于优先级的调度算法 当中的优先级可以是静态不变的，也可以是动态改变的 而优先级呢可以用一个优先数来决定 有了优先级之后，我们也可以重新组织就绪队列。 可以按照 优先级来组织。 呃，之前呢，我们都讨论过，所以我们简单回顾一下 那么最高优先级调度算法呢，肯定非常简单 实现起来简单，但是呢它也是一个不太公平的调度算法 因为它会导致优先级低的进程产生饥饿现象 下面我们来介绍一下 再基于优先级的调度算法当中的，产生的一个新的问题 优先级反转问题。 而这个问题的产生呢主要是 抢占式的优先级调度算法 如果是基于优先级的抢占式调度算法，就会出现一个优先级反转问题 优先级翻反转问题有的时候也称之为优先级的反置，或者是翻转，或者是倒挂 它的现象呢是这样的。 有一个低优先级的进程 它呢占有了一个高优先级进程所需要的资源 那么这样的话呢，高优先级的进程就不能运行。 因为它受制于这个低优先级的进程 那么这样的话呢，就产生了优先级反转 我们举

一个场景。 如果一个系统中，有高优先级的进程 H ，有低优先级进程 L 还有一个比它们，中间，一个中间的优先级进程，叫中优先级进程 比高的低一点、 比低的高一些的中优先级进程。 这个进程呢 恰好是一个 CPU 型进程。 它要用很长的 CPU 时间 那么如果在某种情况下 低优先级的进程被调度上 CPU 了 那么它呢，在执行过程中进入了临界区执行 进入临界区执行的过程当中，没有出临界区 就被抢占了。 那么，它就变成就绪 这个时候呢，高优先级的进程 上 CPU 运行。 由于高优先级的进程也要进临界区，又进不去 所以呢，只能被阻塞，啊，等待低优先级的进程出临界区之后，把资源还给 还给它。 可这时候呢，在低优先级进程的 这个上面呢又有一个中优先级的进程 这个中优先级的又是一个 CPU 型进程，所以它呢老占用 CPU 使得低优先级进程呢，得不到机会去运行 那么低优先级进程不能运行，那么也就是说，高优先级进程也无法运行 这就产生了一个叫做优先级反转，啊，这么一个现象 优先级反转呢，实际上是系统产生的一个错误 曾经在 1997 年，啊，在美国的这个，应该是火星探测 器上面，曾经出现过这样一个优先级反转问题 那么它是一个系统错误。 因为它会使得高优先级的进程 停止不前，导致整个系统的这个性能的降低 怎么解决这个问题呢？有三种解决方案 首先呢，是设置优先级的上限 那么这种方法实际上是说，凡是进入临界区的进程 我给它的优先级都是最高的。 你不在临界区的进程 优先级都会比这个进入临界区的这个，进程优先级要低 这样的话呢，它就可以执行完成，然后把临界区还回去 啊，那么第二种方案呢，就是优先级的继承 如果一个低优先级的进程，阻碍了一个高优先级进程执行，那么 它可以临时地继承这个高优先级的这个进程的优先级 它可以一下子把自己优先级继承到这个高优先级的这个程度 那么它就可以去运行，然后呢把临界区还回去 第三种方案呢，就叫禁止中断 凡是进入临界区的进程，那么就不再响应中断的。 直到它 出临界区才响应中断，这样的话呢就保护了这个进程，让它 继续去执行。 所以呢这三种方案都可以解决优先级的反转问题