



第3章 调度

3.4 调度实例



3.4 调度实例

1. UNIX
2. Solaris
3. OpenEuler
4. Windows



3.4 调度实例

3.4.1 UNIX的进程调度



3.4.1 UNIX的进程调度

- UNIX的进程调度采用**多级反馈队列**调度算法，系统设置了多个就绪队列。进程调度程序执行时：
 - 核心首先从处于“内存就绪”或“被剥夺”状态的进程中选择一个优先级最高的进程；
 - 若系统中**同时有多个进程都具有最高优先级**，则将选择其中处于**就绪状态最久**的进程，将它从所在队列移出，恢复其上下文，使之执行；
 - 仅当最高优先级队列中没有进程时，才从次高优先级队列中找出其队首进程，令它执行一个时间片后，又剥夺该进程的执行；
 - 然后，再从优先级最高的队列中取出下一个就绪进程投入运行。
 - 同时UNIX SVR4添加了静态优先级的抢占式调度。



进程优先级的分类

- 在UNIX系统中，进程的优先级分为三类：
 - 实时优先级层次（优先数159-100）：这一层次的进程先于其他层次进程运行，能利用抢占点抢占内核进程和用户进程。
 - 内核优先级层次（优先数99-60）：这一层次的进程先于分时优先级层次进程但迟于实时优先级层次进程运行。
 - 分时优先级层次（优先数59-0）：最低优先级层次，用于非实时的用户应用。



3.4 调度实例

3.4.2 Solaris的进程调度



3.4.2 Solaris

- 基于优先级的线程调度
- 六个类别优先级
 - 实时(RT)
 - 系统 (SYS)
 - 分时(默认) (TS)
 - 交互 (IA)
 - 公平分享 (FSS)
 - 固定优先级 (FP)
- 一次只给一个线程分配某一个类别
- 每个类别都有自己的调度算法
- 默认分时采用多级反馈队列
 - 优先级和时间片成反比：优先级越高，时间片越短
 - 交互进程有更好的优先级和响应时间
 - CPU约束程序有更好的吞吐量

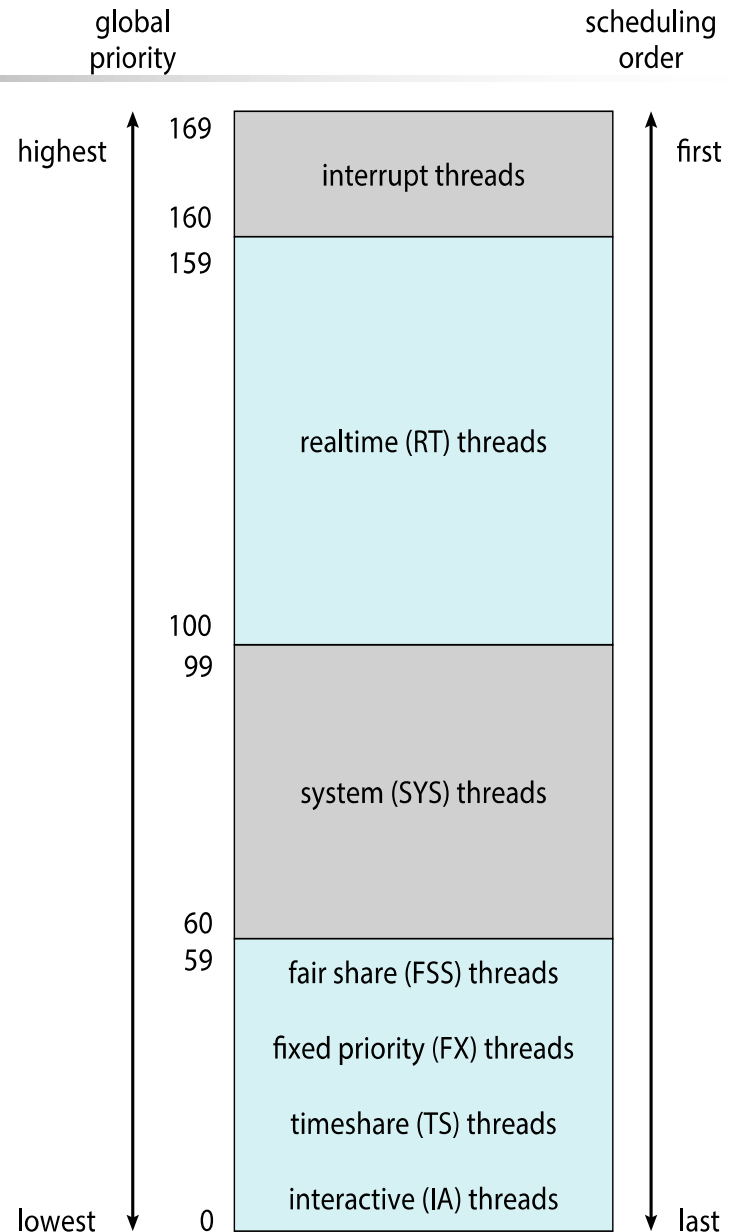


调度交互和分时线程分配表

priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

Solaris调度原则

- 调度器将面向类别的优先级转换为全局线程的优先级
 - 从中选取全局最高优先级的线程进行执行
 - 线程运行，直到(1)阻塞，(2)时间片用尽，(3)高优先级线程强占
 - 如果有多个线程具有相同优先级，则采用RR算法调度





3.4 调度实例

3.4.3 OpenEuler的进程调度

本小节内容由华为公司素材编写





3.4.3 OpenEuler调度算法

- 采用了类似Linux的算法
 - 核心调度算法：CFS（Completely Fair Scheduler, 完全公平调度）
- 调度策略和进程类别
 - 限期进程：
 - 限期调度策略（SCHED_DEADLINE）
 - 选择进程截至时间距当前时间点最近的进程运行
 - 实时进程：
 - 先进先出调度（SCHED_FIFO）
 - 轮转调度（SCHED_RR）
 - 普通进程：
 - 标准轮流分时调度（SCHED_NORMAL）
 - CFS调度算法



3.4.3 OpenEuler调度算法

■ 调度类

- 停机调度类：调度对象为停机进程/线程，用于迁移进程 (CPU-i→CPU-j)
- 限期调度类：限期进程，选择绝对截至期限最小的进程
- 实时调度类：实时进程，选择优先级最高的第一个链表中的第一个进程。FIFO, RR。
- 公平调度类：普通进程，CFS
- 空闲调度类：空闲线程，即0号线程

■ 调度类优先级：

- 停机调度类>限期调度类>实时调度类>公平调度类>空闲调度类

3.4.3 OpenEuler调度算法

■ CFS调度策略基本思想

- 调度时延/目标延迟(target latency): 进程第一次获得CPU的时间到下一次获得CPU时间的时间间隔
- 一个调度时延内, 所有进程都有机会被调度, 但进程运行时间不同。
- 随着任务的增加, 调度时延是可增加的。
- 如何为每个进程分配CPU时间配额?
 - 依据进程优先级和当前系统负载来进行。

■ CPU时间配额的问题:

- 基于nice -20—+19, 默认为0, 值越低, 优先级越高
- 将Nice值转化为权重
- 进程分配到的处理时间比例
 - $P_i = \omega_i / \sum_{j \in cfs} \omega_j$

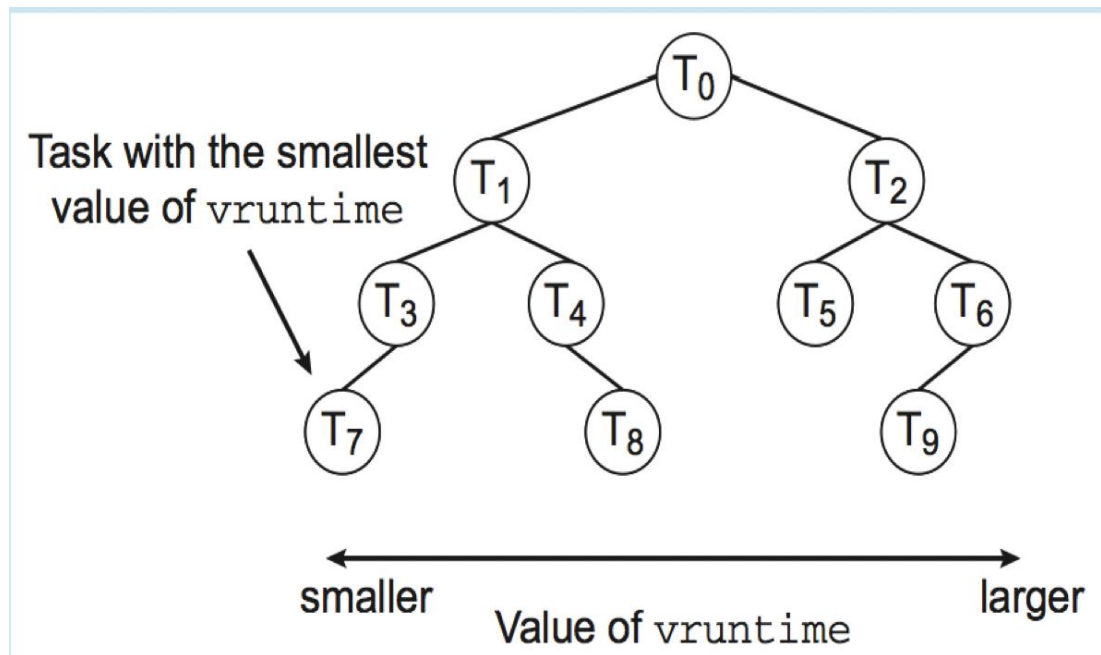
```
const int sched_prio_to_weight[40] = {  
/* -20 */ 88761, 71755, 56483, 46273, 36291,  
/* -15 */ 29154, 23254, 18705, 14949, 11916,  
/* -10 */ 9548, 7620, 6100, 4904, 3906,  
/* -5 */ 3121, 2501, 1991, 1586, 1277,  
/* 0 */ 1024, 820, 655, 526, 423,  
/* 5 */ 335, 272, 215, 172, 137,  
/* 10 */ 110, 87, 70, 56, 45,  
/* 15 */ 36, 29, 23, 18, 15,};
```

3.4.3 OpenEuler调度算法

- CPU时间配额的问题（续）
 - 调度时延sched_period
 - 依据就绪进程个数而定，若不大于sched_nr_latency(默认为8)，取sysctl_sched_latency（默认6ms）
 - 若大于sched_nr_latency，就绪进程个数*sysctl_sched_min_granuity（默认0.75ms）
 - 进程调度后分配的实际运行时间
 - $W_i = \text{sched_period} \times P_i = \text{sched_period} \times \omega_i / \sum_{j \in cfs} \omega_j$
 - 进程的虚拟运行时间
 - $V_i = W_i \times \omega_{\text{nice0}} / \omega_i$
 - 可以看出，不同进程执行完后的虚拟运行时间应该是相同的
 - 选择调度进程的依据
 - 虚拟运行时间最小的进程被选择（权重大的）

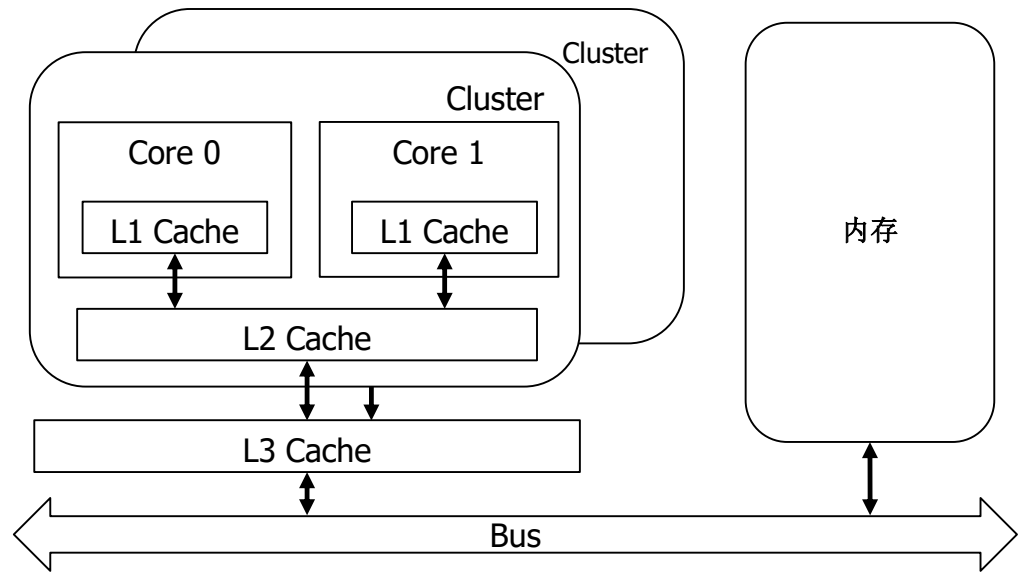
3.4.3 OpenEuler调度算法

- 采用红黑树数据结构(自平衡二叉树)
 - 当一个任务可运行, 则添加到树上, 否则删去
 - 将vruntime少的, 加到左枝, 更多的加到右枝, 则最左的叶子为最小vruntime
 - 搜索到最左边节点需要 $O(\log N)$ 操作, 但为了提高效率, 将其值缓存到rb_leftmost中, 从而加快检索
 - 有利于I/O密集型任务, 因其vruntime小于CPU密集型任务, 则可以实现抢占



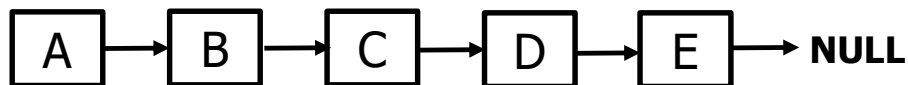
3.4.3 OpenEuler调度算法

- 多核处理器缓存和主存的关系发生变化之后，系统主要会面临如下问题：
 - 缓存一致性问题
 - 缓存亲和性问题；
 - 核间数据共享；
 - 负载均衡。

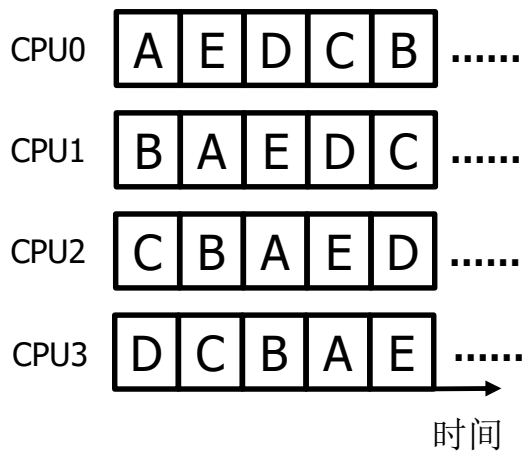


3.4.3 OpenEuler调度算法

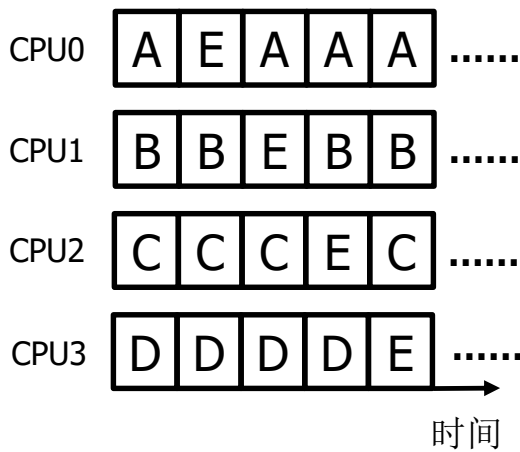
■ 单队列调度



调度队列情况：5个就绪进程



单队列调度情况(策略一)



单队列调度情况(策略二)

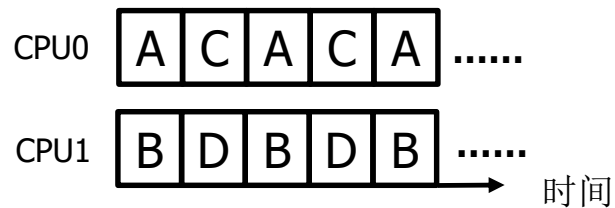
- **策略一：**每个CPU使用轮转调度算法选择下一个要执行的进程。进程会在不同CPU间转移，违背了缓存亲和性。
- **策略二：**引入亲和度机制尽可能地让进程在同一个CPU上执行。牺牲了某些进程的亲和性(如图所示中的进程E)。

3.4.3 OpenEuler调度算法

■ 多队列调度



openEuler使用多队列调度策略。图中队列Q0由CPU0维护，Q1由CPU1维护。



采用轮转调度算法，调度之初的情况。



一段时间后，假设A、C两个进程都结束，此时负载失衡。



3.4.3 OpenEuler调度算法

- 解决办法：openEuler中的迁移线程
 - 解决多CPU负载不均衡问题最直接的方法：
 - 让就绪进程跨CPU迁移
 - 如，将进程D迁移到CPU0上运行后，CPU0和CPU1则实现了负载均衡。
 - 在openEuler中，每个处理器都有一个迁移线程（称为migration/CPUID），每个迁移线程都有一个由函数组成的停机工作队列。
 - 如上例：
 - CPU0向CPU1的停机工作队列中添加一个工作函数，并唤醒CPU1上的迁移线程；
 - 该迁移线程不会被其他进程抢占，故其第一时间从停机工作队列中取出函数执行，即将进程从CPU1迁移到CPU0；
 - 此时已实现负载均衡。



3.4.4 Windows调度算法

- Windows NT5之后处理器调度的对象是线程，也称内核级线程调度。
 - 实现了基于优先级抢先式的多处理器调度系统，系统总是运行优先级最高的就绪线程。
 - 线程可在任何可用处理器上运行，但可限制某线程只能在某处理器上运行，**亲和处理器集合**允许用户线程通过Win32调度函数选择它偏好的处理器。
- Win7(NT6)引入了UMS(User Mode Scheduling)，解决了Windows NT5中Fiber(纤程)调度问题



Windows优先级类别

- 使用32个线程优先级，范围从0到31，分成三大类：
 - 实时优先级（优先数为31-16）：
 - 用于通信任务和实时任务。
 - 实时优先数线程的优先数不可变
 - 一旦就绪线程的实时优先数比运行线程高，它将抢占处理器运行。
 - 可变优先级（优先数为15-1）：
 - 用于交互式任务。
 - 可根据执行过程中的具体情况动态调整优先数，但不能突破15。
 - 1个系统线程优先级（0）
 - 仅用于对系统中空闲物理页面进行清零的零页线程。
- 调度程序会依据优先级从高到底选择线程，如果没有找到就执行idle thread



Windows 优先级类别

- Win32 API 定义的进程优先级
 - REALTIME_PRIORITY_CLASS
 - HIGH_PRIORITY_CLASS
 - ABOVE_NORMAL_PRIORITY_CLASS
 - NORMAL_PRIORITY_CLASS (默认)
 - BELOW_NORMAL_PRIORITY_CLASS
 - IDLE_PRIORITY_CLASS
 - 除了REALTIME_PRIORITY_CLASS, 其他都是可变的



Windows 优先级类别

- 具有给定优先级类的一个线程还具有一个相对优先级
 - TIME_CRITICAL, HIGHEST, ABOVE_NORMAL, NORMAL, BELOW_NORMAL, LOWEST, IDLE
- 优先级类+相对优先级→数字化的优先级
- 每个类别中，基本优先级均为NORMAL
- 如果时间片用完，且为可变优先级类型，则优先级降低，但是不会低于本类的基本值



Windows 优先级

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1



Windows 优先级类别(Cont.)

- 如果一个可变优先级的线程从等待中被释放，
 - 调度程序会提升其优先级。
- 提升数量由线程等待什么而决定：
 - 对I/O密集的保持设备忙碌，允许计算密集的使用后台空闲周求
 - 等待键盘、鼠标I/O的线程会得到较大的提升，提高响应
 - 等待磁盘的，得到中级的提升
 - 前台进程与后台进程：前台窗口能够获得3倍时间片的增加



线程优先级提升

提升线程当前优先级的情况：

- I/O操作完成

- 保证等待i/o的线程能够更多机会立即处理所得结果
- 提升幅度与I/O请求响应时间要求一致
 - 磁盘、光驱、并口和视频为1；
 - 网络、串口和命名管道为2；
 - 键盘和鼠标为6；
 - 音频为8。



线程优先级提升

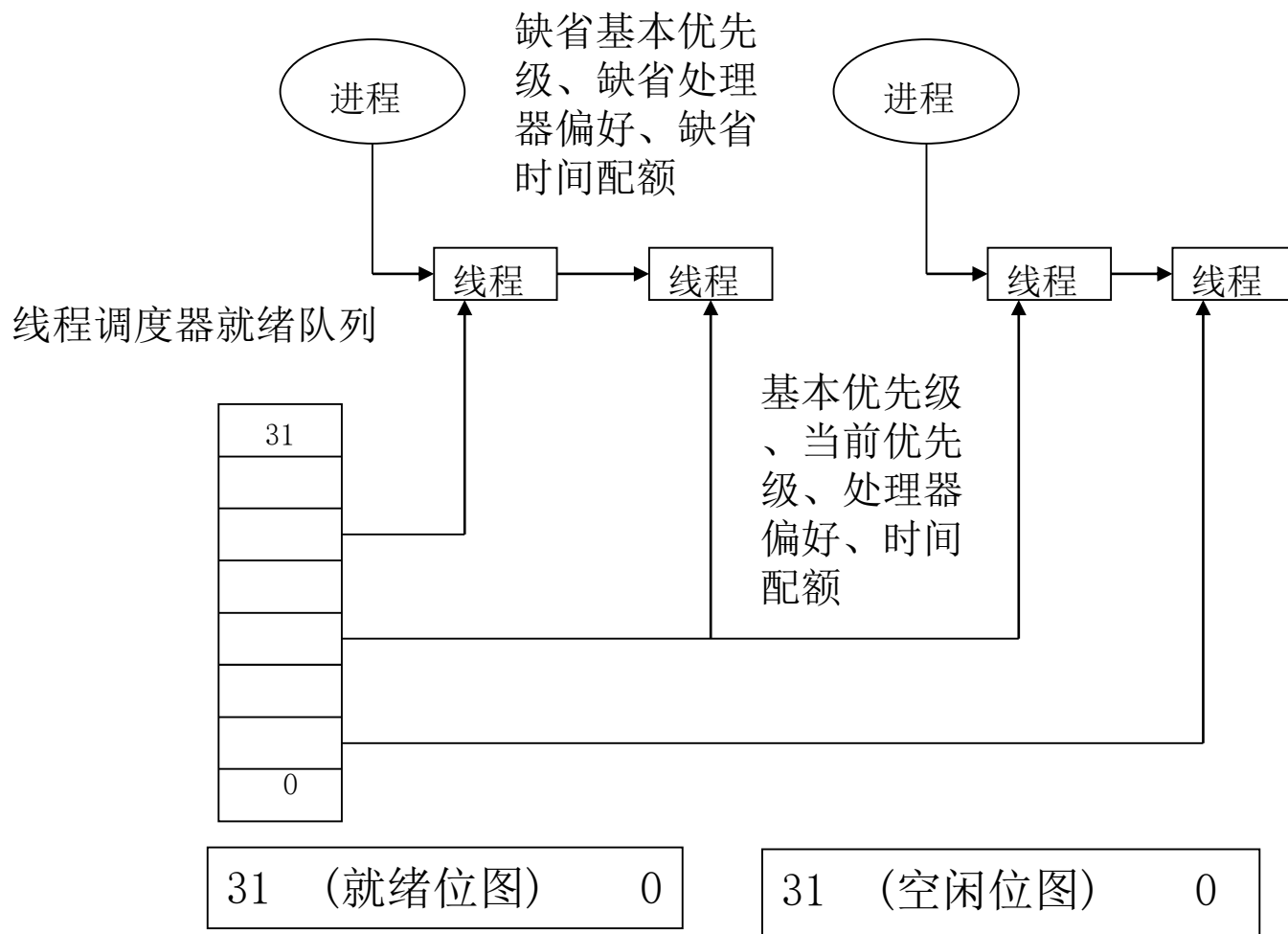
- 信号量或事件等待结束
 - 等待结束时候提升，然后随之降低
- 前台进程中的线程完成一个等待操作
 - 提高交互类型应用的响应时间
- 由于窗口活动而唤醒图形用户接口线程
 - 原因同上
- 优先级逆转问题：
 - 当一个高优先级(如11)等待低优先级(如5)的线程
 - 而低优先级线程处于就绪状态超过一定时间，但没能进入运行状态
 - 把这样的低优先级线程提高优先级，当该线程用完时间片后，立即降低优先级

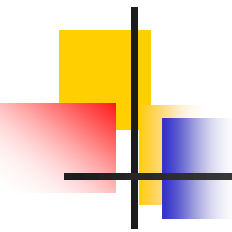


Windows 优先级类别(Cont.)

- Windows 7 user-mode scheduling (UMS)
 - 应用程序可创建和管理线程
 - 对于创建大量线程的应用程序，用户模式线程调度效率更高，可不需内核的干预
 - UMS调度程序可由编程语言库提供
 - 如C++ Concurrent Runtime (ConcRT)

线程调度数据结构





3.4 调度实例

本节结束!