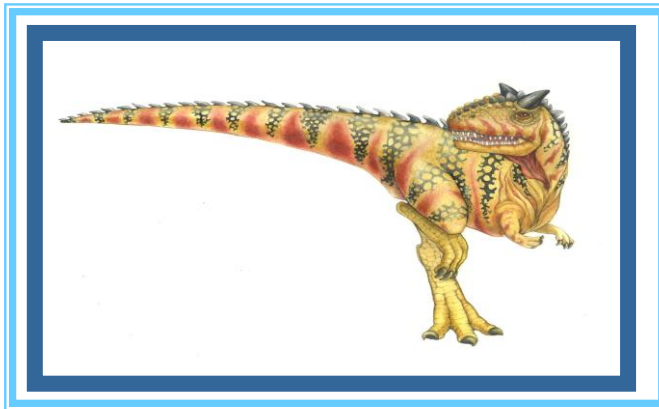




5.6 Windows Scheduling





5.6.2 Windows 2000/XP的线程调度

- Windows 2000/XP的处理机调度的调度对象是线程。
- Windows 2000/XP的线程调度并不是单纯使用某一种调度算法，而是多种算法的综合体，针对实际系统的需要进行针对性的优化和改进。





5.6.2.1 Windows 2000/XP线程状态

5.6.2.2 Windows 2000/XP的线程调度特征

5.6.2.3 Win32中与线程调度相关的应用编程接口

5.6.2.4 线程优先级

5.6.2.5 线程时间配额

5.6.2.6 调度数据结构

5.6.2.7 调度策略

5.6.2.8 线程优先级提升

5.6.2.9 对称多处理机系统上的线程调度

5.6.2.10 空闲线程



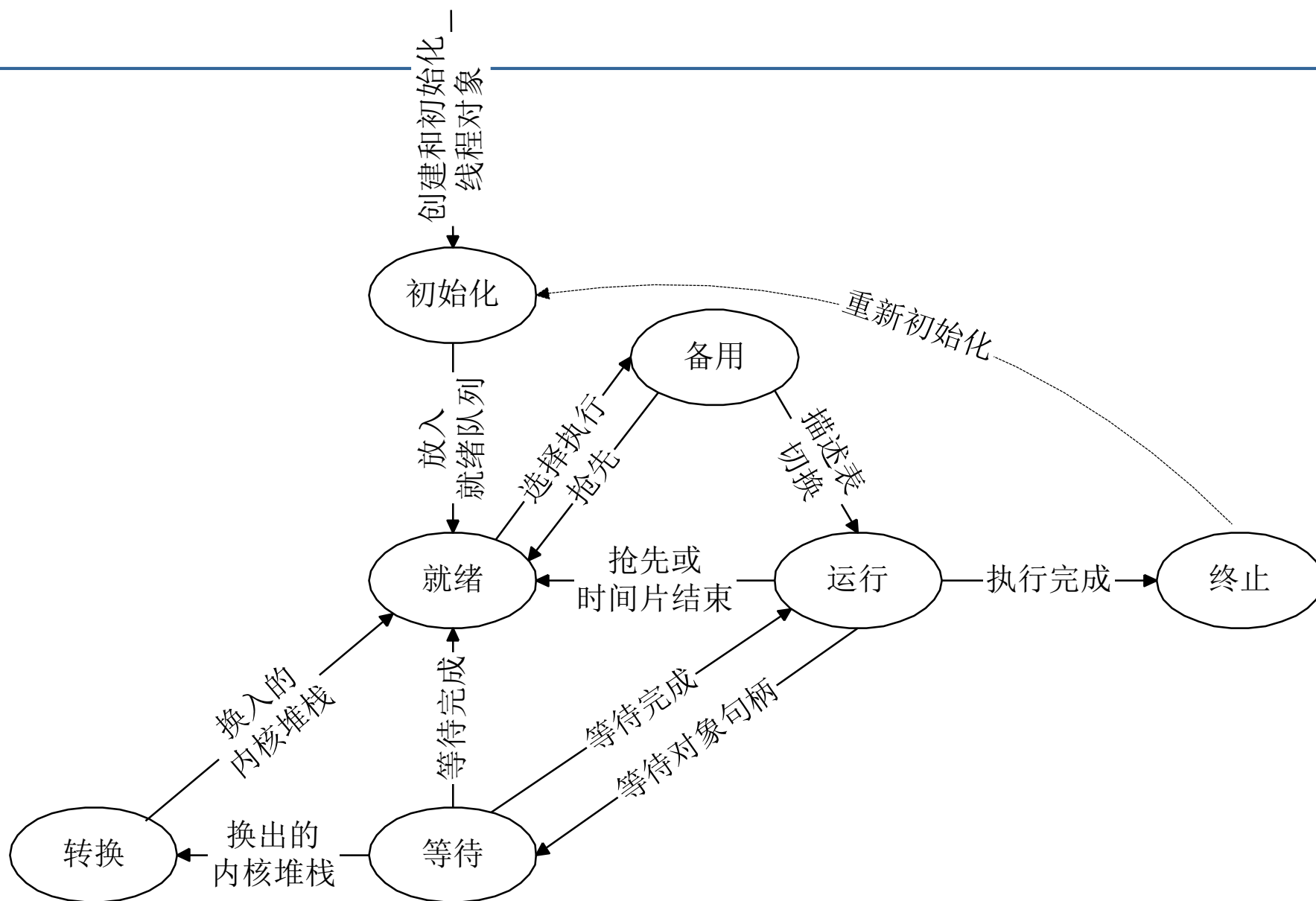


5.6.2.1 Windows 2000/XP线程状态

- Windows 2000/XP的线程是**内核线程**，系统的处理机调度对象为线程。
- Windows 2000/XP把线程状态分成下面**七种状态**，与单挂起进程模型很相似，它们的主要区别在于从就绪状态到运行状态的转换中间多上一个**备用状态**，以优化线程的抢占特征。

[返回](#)





- 就绪状态(Ready): 线程已获得除处理机外的所需资源, 正等待调度执行。
- 备用状态(Standby): 已选择好线程的执行处理机, 正等待描述表切换, 以进入运行状态。系统中每个处理机上只能有一个处于备用状态的线程。
- 运行状态(Running): 已完成描述表切换, 线程进入运行状态。线程会一直处于运行状态, 直到被抢占、时间片用完、线程终止或进入等待状态。
- 等待状态(Waiting): 线程正等待某对象, 以同步线程的执行。当等待事件出现时, 等待结束, 并根据优先级进入运行或就绪状态。
- 转换状态(Transition): 转换状态与就绪状态类似, 但线程的内核堆栈位于外存。当线程等待事件出现而它的内核堆栈处于外存时, 线程进入转换状态; 当线程内核堆栈被调回内存时, 线程进入就绪状态。
- 终止状态(Terminated): 线程执行完就进入终止状态; 如执行体有一指向线程对象的指针, 可将处于终止状态的线程对象重新初始化, 并再次使用。
- 初始化状态(Initialized): 线程创建过程中的线程状态;





Windows 2000/XP的线程控制

- **CreateThread**完成线程创建，在调用进程的地址空间上创建一个线程，以执行指定的函数；它的返回值为所创建线程的句柄。
- **ExitThread**用于结束当前线程。
- **SuspendThread**可挂起指定的线程。
- **ResumeThread**可激活指定线程，它的对应操作是递减指定线程的挂起计数，当挂起计数减为0时，线程恢复执行。





5.6.2.2 Windows 2000/XP的线程调度特征

调度单位是线程而不是进程，采用严格的抢占式动态优先级调度，依据优先级和分配时间配额来调度。

- 每个优先级的就绪线程排成一个先进先出队列；
- 当一个线程状态变成就绪时，它可能立即运行或排到相应优先级队列的尾部。
- 总运行优先级最高的就绪线程；



- 完全的事件驱动机制，在被抢占前没有保证的运行时间：
 - 进入就绪事件；
 - 时间配额用完事件；
 - 优先级改变事件；
 - 亲合处理机集合改变事件；
- 在同一优先级的各线程按时间片轮转算法进行调度；
- 在多处理机系统中多个线程并行运行；





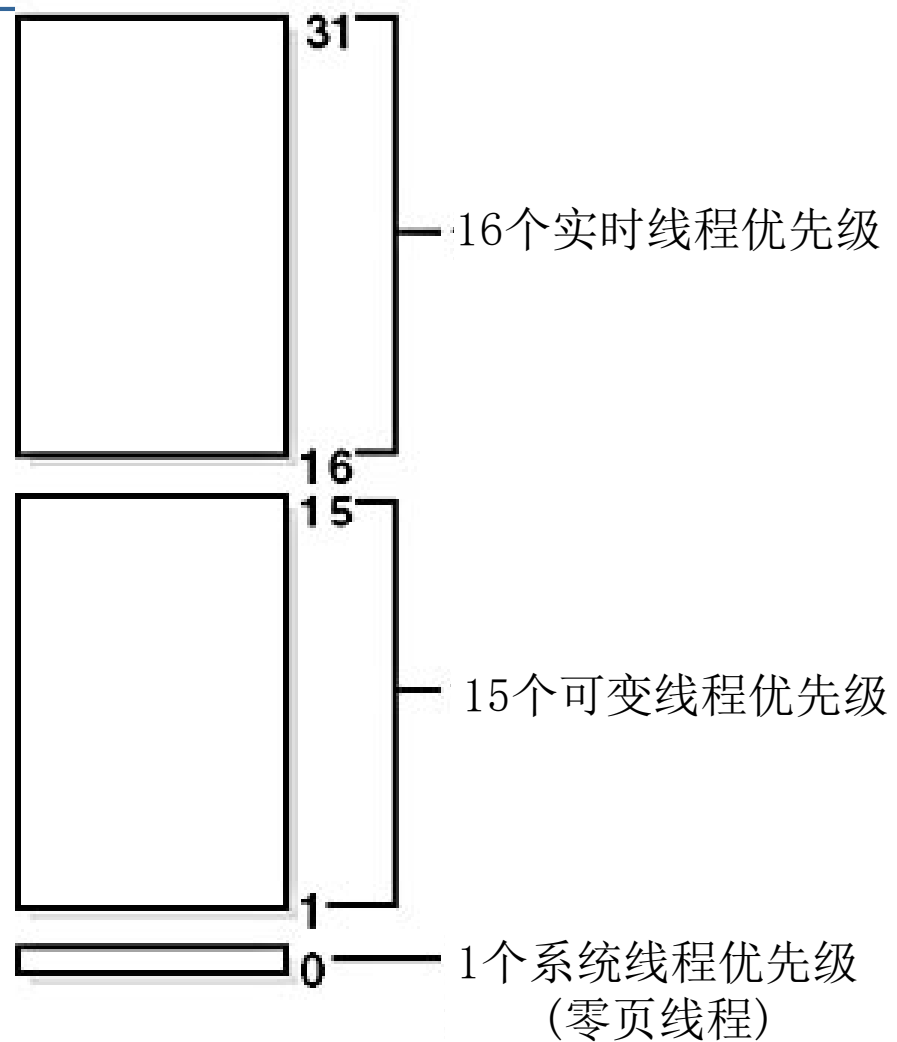
5.6.2.3 Win32中与线程调度相关的API

- Suspend/ResumeThread
- Get/SetPriorityClass
- Get/SetThreadPriority
- Get/SetProcessPriorityBoost
- Get/SetThreadPriorityBoost
- Get/SetProcessAffinityMask
- SetThreadAffinityMask
- SetThreadIdealProcessor
- SwitchToThread
- Sleep
- SleepEx





5.6.2.4 线程优先级





Process priority class

- 线程得到的优先权是相对于它们的进程优先权类而言的。开始时由 CreateProcess 设置了6个进程优先权类，并且每个类有一个基本优先权值。

- **REALTIME_PRIORITY_CLASS**，基本优先权为**24**。
- **HIGH_PRIORITY_CLASS**，基本优先权为**13**。
- **ABOVE_NORMAL_PRIORITY_CLASS**，基本优先权为**10**

//Windows NT and Windows Me/98/95: This value is not supported.

- **NORMAL_PRIORITY_CLASS**，基本优先权为**8**。
- **BELOW_NORMAL_PRIORITY_CLASS**，基本优先权为**6**

//Windows NT and Windows Me/98/95: This value is not supported.

- **IDLE_PRIORITY_CLASS**，基本优先权为**4**。



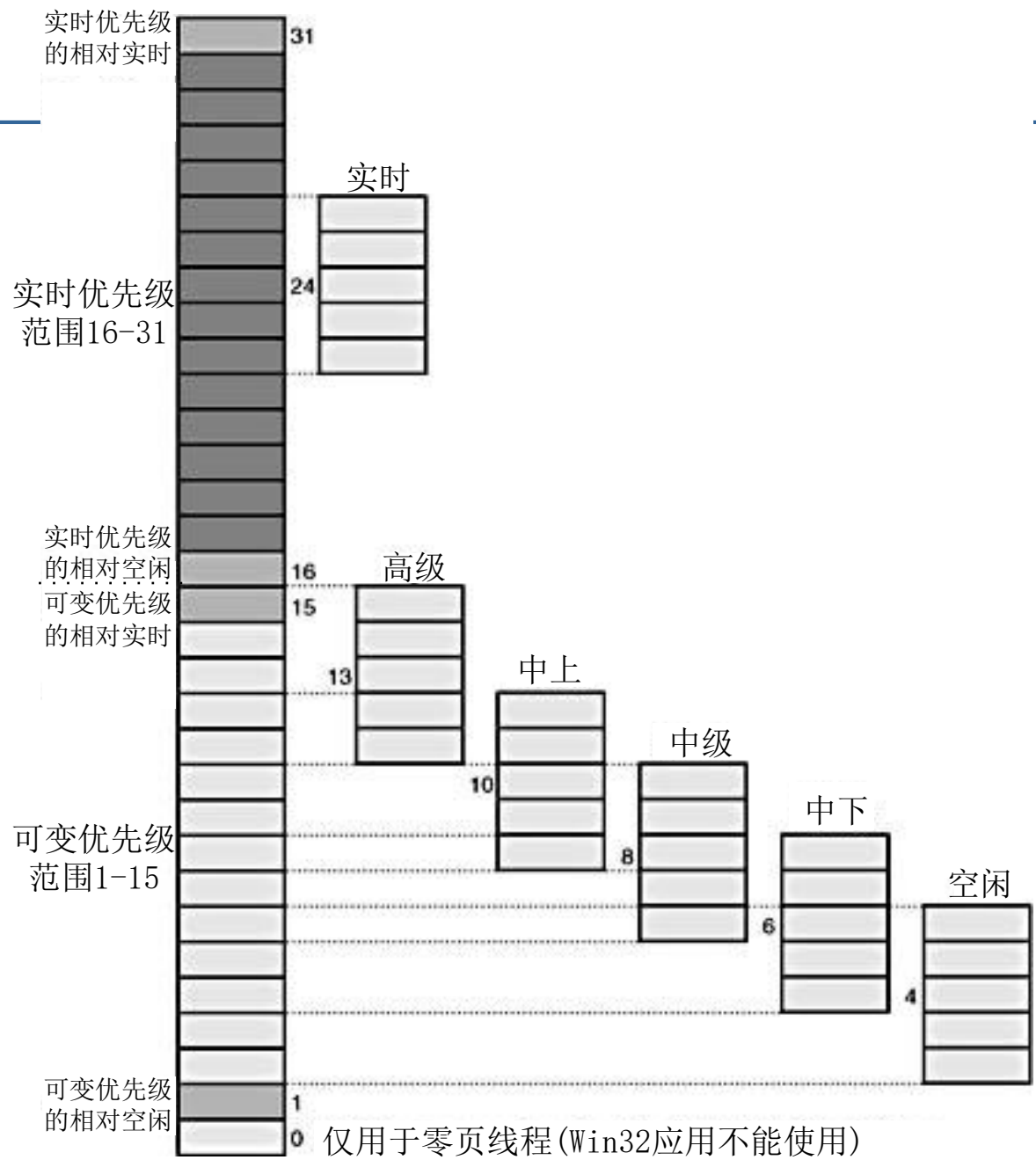


THREAD PRIORITY

■ 线程优先权的设置是相对于进程基本优先权而言的，而且在线程创建的时候，其优先权设置为进程的优先权。线程优先权在进程基本优先权的 ± 2 范围之内。作为结果的7个线程优先权的符号名称如下：

- THREAD_PRIORITY_TIME_CRITICAL
- THREAD_PRIORITY_HIGHEST +2
- THREAD_PRIORITY_ABOVE_NORMAL +1
- THREAD_PRIORITY_NORMAL 0
- THREAD_PRIORITY_BELOW_NORMAL -1
- THREAD_PRIORITY_LOWEST -2
- THREAD_PRIORITY_IDLE





仅用于零页线程 (Win32应用不能使用)





Fig . Windows 2000 /xp Priorities

	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





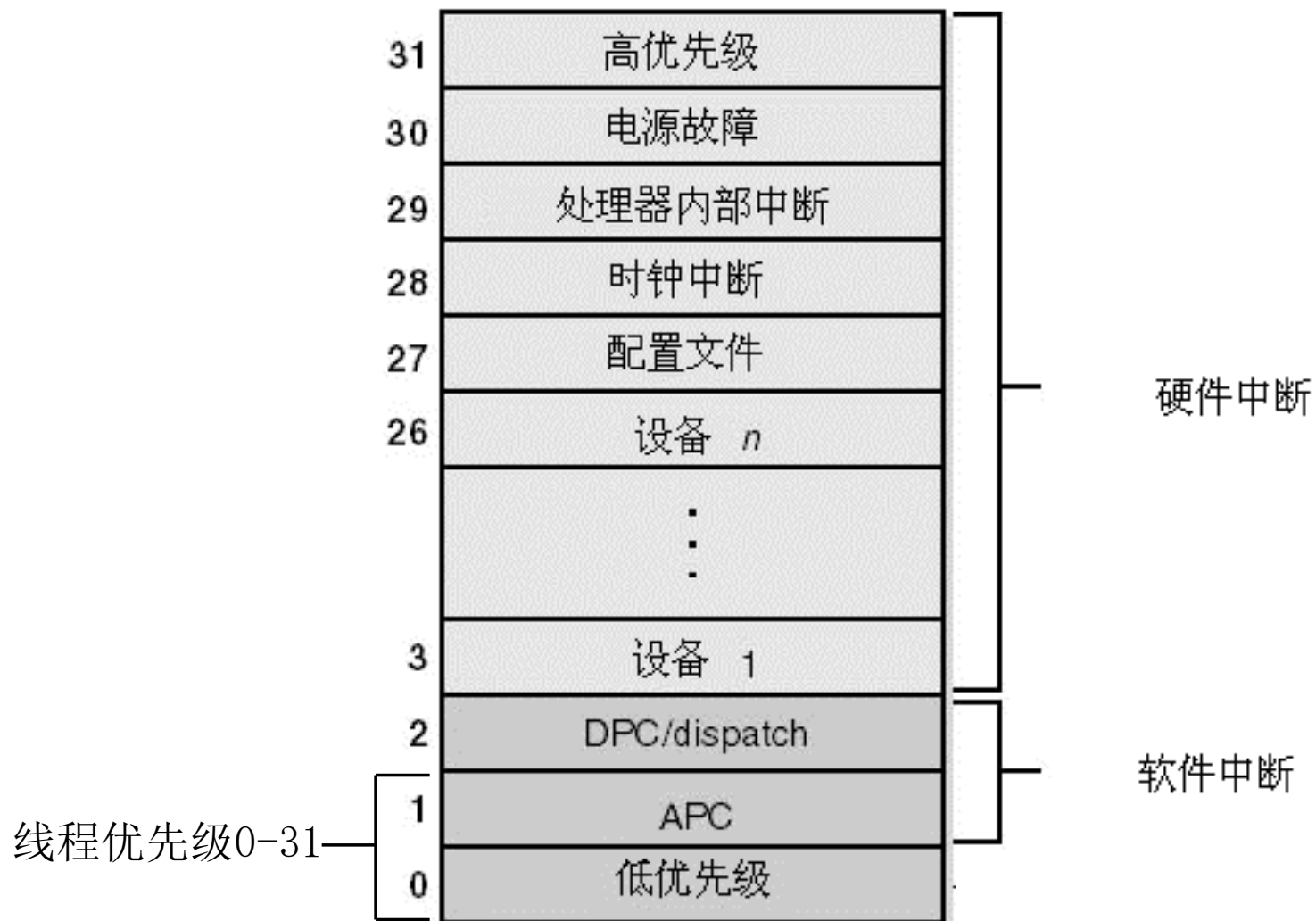
1. 实时优先级

- 在应用程序中，要把线程的优先级提升到实时优先级，用户必须有升高线程优先级的权限。
- 如果用户进程在实时优先级运行时间过多，它将可能阻塞关键系统功能的执行，阻塞系统线程的运行；但不会阻塞硬件中断处理。
- 在被其他线程抢占时，具有实时优先级线程与具有可变优先级线程的行为是不同的。
- Windows 2000/XP并不是通常意义上的实时，并不提供实时操作系统服务。





2. 中断优先级与线程优先级的关系





5.6.2.5 线程时间配额

- **时间配额**是一个线程从进入运行状态到Windows 2000/XP检查是否有其他优先级相同的线程需要开始运行之间的时间总和。
- 一个线程用完了自己的时间配额时，如果没有其它相同优先级线程，Windows 2000/XP将重新给该线程分配一个新的时间配额，并继续运行。
- 时间配额不是一个时间长度值，而是一个称为**配额单位(quantum unit)**的整数。





1. 时间配额的计算

- 缺省时，在Windows 2000专业版中线程时间配额为6；而在Windows 2000服务器中线程时间配额为36。
 - 在Windows 2000服务器中取较长缺省时间配额的原因是，保证客户请求所唤醒的服务器应用有足够的时间在它的时间配额用完前完成客户的请求并回到等待状态。
- 每次时钟中断，时钟中断服务例程从线程的时间配额中减少一个固定值3。
 - 如果没有剩余的时间配额，系统将触发时间配额用完处理，选择另外一个线程进入运行状态。
 - 在Windows 2000专业版中，由于每个时钟中断时减少的时间配额为3，一个线程的缺省运行时间为2个时钟中断间隔；在Windows 2000服务器中，一个线程的缺省运行时间为12个时钟中断间隔。





- 如果时钟中断出现时系统正在处在DPC/线程调度层次以上(如系统正在执行一个延迟过程调用或一个中断服务例程), 当前线程的时间配额仍然要减少。甚至在整个时钟中断间隔期间, 当前线程一条指令也没有执行, 它的时间配额在时钟中断中也会被减少。
- 不同硬件平台的时钟中断间隔是不同的, 时钟中断的频率是由**硬件抽象层**确定的, 而不是内核确定的。
 - 例如, 大多数x86**单处理机**系统的时钟中断间隔为**10**毫秒, 大多数x86**多处理机**系统的时钟中断间隔为**15**毫秒。



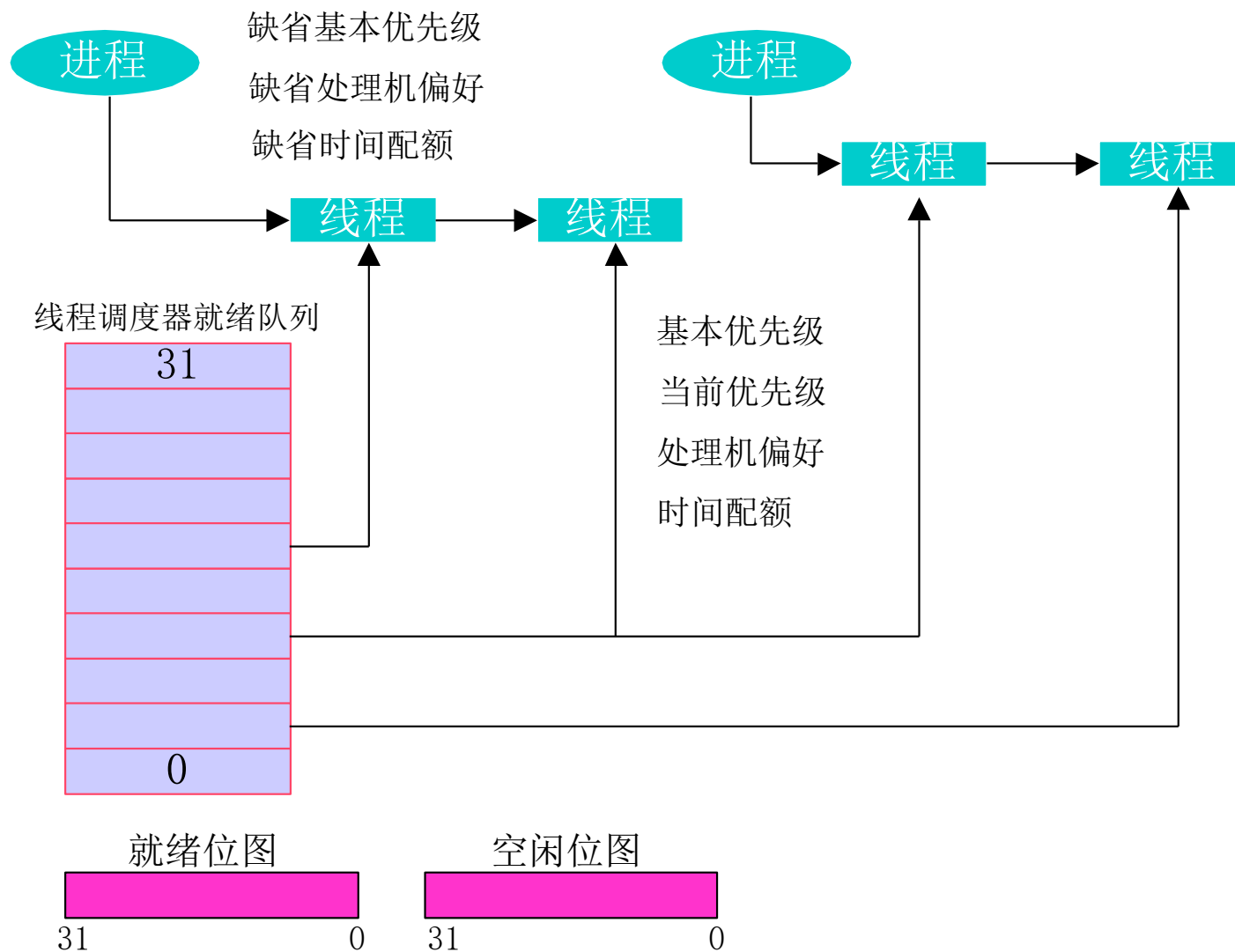


- 在等待完成时允许减少部分时间配额。
 - 当优先级小于14的线程执行一个等待函数(如WaitForSingleObject或WaitForMultipleObjects)时，它的时间配额被减少1个时间配额单位。当优先级大于等于14的线程在执行完等待函数后，它的时间配额被重置。
- 这种部分减少时间配额的做法可解决线程在时钟中断触发前进入等待状态所产生的问题。
 - 如果不进行这种部分减少时间配额操作，一个线程可能永远不减少它的时间配额。例如，一个线程运行一段时间后进入等待状态，再运行一段时间后又进入等待状态，但在时钟中断出现时它都不是当前线程，则它的时间配额永远也不会因为运行而减少。





5.6.2.6 调度数据结构



[返回](#)





■ 就绪位图(KiReadySummary)

- 为了提高调度速度，Windows 2000维护了一个称为就绪位图(KiReadySummary)的32位。就绪位图中的每一位指示一个调度优先级的就绪队列中是否有线程等待运行。B0与调度优先级0相对应，B1与调度优先级1相对应，等待。

■ 空闲位图(KildleSummary)

- Windows 2000还维护一个称为空闲位图(KildleSummary)的32位。空闲位图中的每一位指示一个处理机是否处于空闲状态。

■ 调度器自旋锁(KiDispatcherLock)

- 为了防止调度器代码与线程在访问调度器数据结构时发生冲突，处理机调度仅出现在DPC/调度层次。但在多处理机系统中，修改调度器数据结构需要额外的步骤来得到内核调度器自旋锁(KiDispatcherLock)，以协调各处理机对调度器数据结构的访问。





与线程调度相关的内核变量

变量名	变量类型	功能说明
KiDispatcherLock	自旋锁	调度器自旋锁
KeNumberProcessors	字节	系统中的可用处理机数目
KeActiveProcessors	32 位位图	描述系统中各处理机是否正处于运行状态
KiIdleSummary	32 位位图	描述系统中各处理机是否处于空闲状态
KiReadySummary	32 位位图	描述各调度优先级是否有就绪线程等待调度
KiDispatcherReadyListHead	有 32 个元素的数组	32 个元素分别指向 32 个就绪队列





5.6.2.7 调度策略

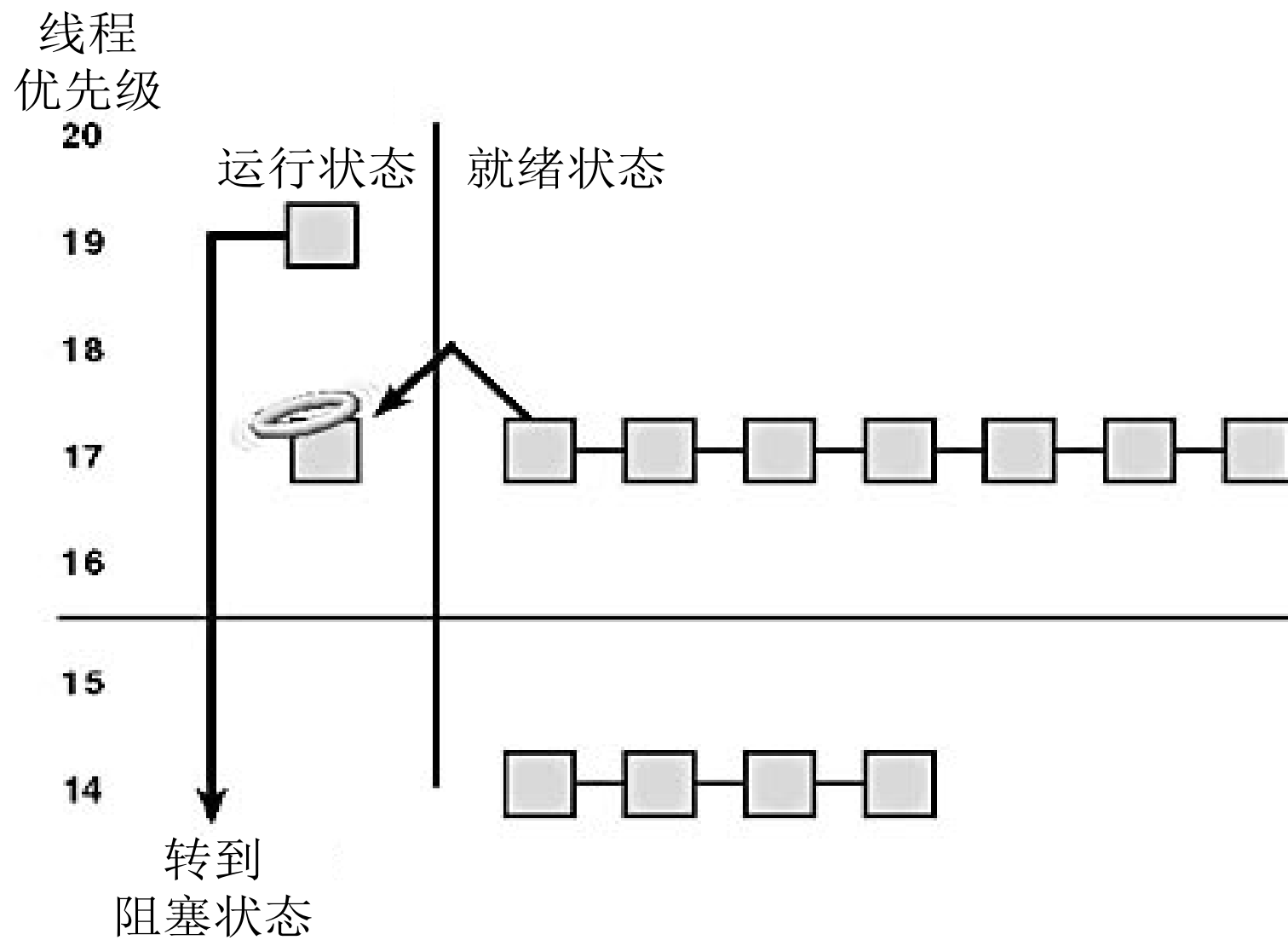
- Windows 2000/XP严格**基于线程的优先级**来确定哪一个线程将占用处理机并进入运行状态。
- Windows 2000/XP在单处理机系统和多处理机系统中的线程调度是不同的。
- 这里介绍单处理机系统中线程调度。

[返回](#)





1. 主动切换



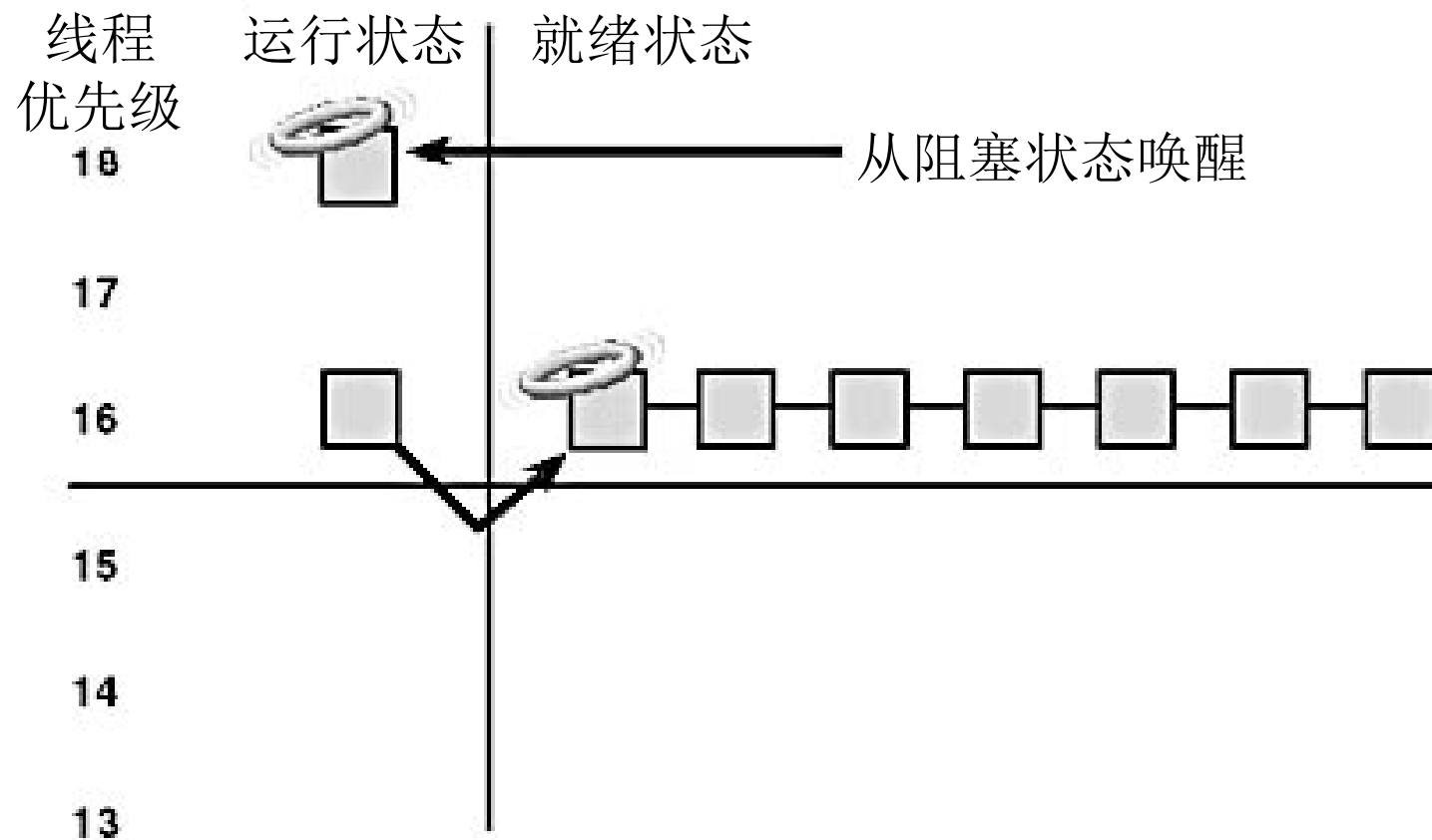


- 许多Win32等待函数调用(如WaitForSingleObject或WaitForMultipleObjects等)都使线程等待某个对象，等待的对象可能有事件、互斥信号量、资源信号量、I/O操作、进程、线程、窗口消息等。
- 通常进入等待状态线程的时间配额不会被重置，而是在等待事件出现时，线程的时间配额被减1，相当于 $1/3$ 个时钟间隔；如果线程的优先级大于等于14，在等待事件出现时，线程的时间配额被重置。





2. 抢占



当一个高优先级线程进入就绪状态时，正在处于运行状态的低优先级线程被抢占。



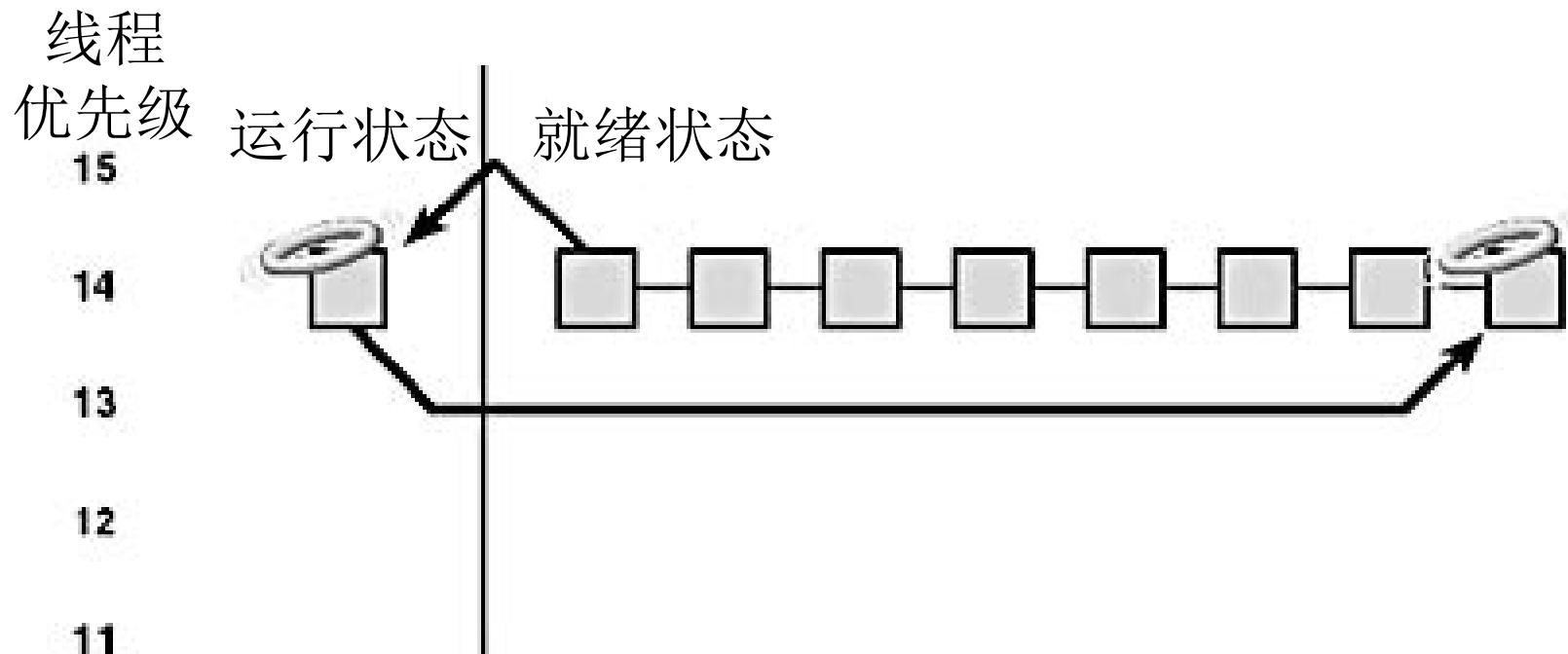


- 可能在以下两种情况下出现抢占：
 - 高优先级线程的等待完成，即一个线程等待的事件出现。
 - 一个线程的优先级被增加或减少。
- 用户态下运行的线程可以抢占内核态下运行的线程。
 - 在判断一个线程是否被抢占时，并不考虑线程处于用户态还是内核态，调度器只是依据线程优先级进行判断。
- 当线程被抢占时，它被放回相应优先级的就绪队列的队首。
 - 处于实时优先级的线程在被抢占时，时间配额被重置为一个完整的时间配额；
 - 处于动态优先级的线程在被抢占时，时间配额不变，重新得到处理机使用权后将运行到剩余的时间配额用完。





3. 时间配额用完



线程完整用完一个规定的时间片值时，重新赋予新时间片值，优先级降一级（不低于基本优先级），放在相应优先级就绪队列的尾部。





- 如果刚用完时间配额的线程优先级降低了，Windows 2000将寻找一个优先级高于刚用完时间配额线程的新设置值的就绪线程。
- 如果刚用完时间配额的线程的优先级没有降低，并且有其他优先级相同的就绪线程，Windows 2000将选择相同优先级的就绪队列中的下一个线程进入运行状态，刚用完时间配额的线程被排到就绪队列的队尾(即分配一个新的时间配额并把线程状态从运行状态改为就绪状态)。
- 如果没有优先级相同的就绪线程可运行，刚用完时间配额的线程将得到一个新的时间配额并继续运行。





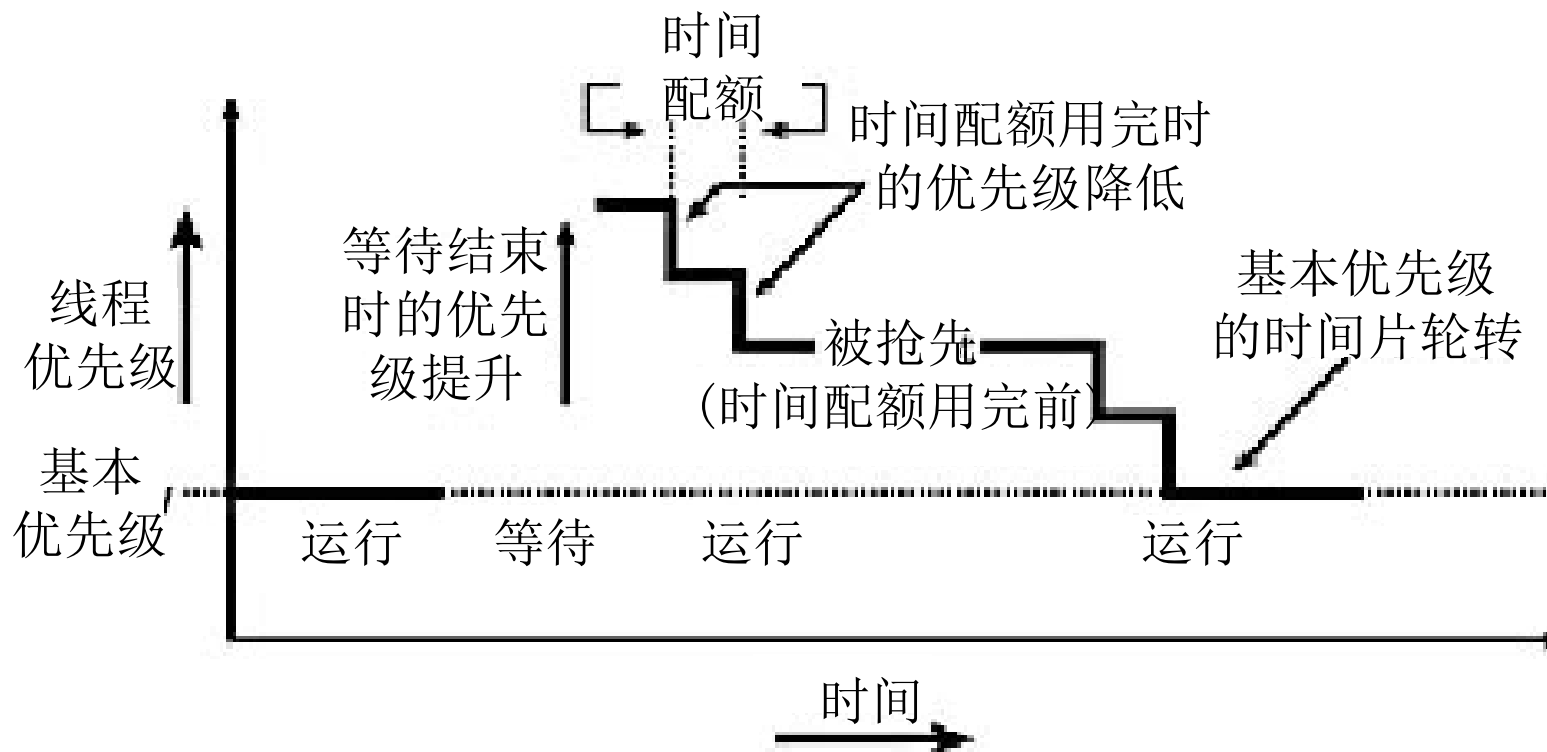
4. 线程结束

- 当线程完成运行时，它的状态从运行状态转到终止状态。
- 线程完成运行的原因可能是
 - 通过调用**ExitThread**而从主函数中返回
 - 通过被其他线程通过调用**TerminateThread**来终止。
- 如果处于终止状态的线程对象上没有未关闭的句柄，则该线程将被从进程的线程列表中删除，相关数据结构将被释放。





5.6.2.8 线程优先级提升



线程由于调用等待函数而阻塞时，减少一个时间片，并依据等待事件类型提高优先级；如等待键盘事件比等待磁盘事件的提高幅度大。

[返回](#)





- 在下列5种情况下，Windows 2000会提升线程的当前优先级：
 - I/O操作完成
 - 信号量或事件等待结束
 - 前台进程中的线程完成一个等待操作
 - 由于窗口活动而唤醒图形用户接口线程
 - 线程处于就绪状态超过一定时间，但没能进入运行状态(处理机饥饿)
- 线程优先级提升的目的是改进系统吞吐量、响应时间等整体特征，解决线程调度策略中潜在的不公正性。但它也不是完美的，它并不会使所有应用都受益。
- Windows 2000永远不会提升实时优先级范围内(16至31)的线程优先级。





1. I/O操作完成后的线程优先级提升

- 在完成I/O操作后，Windows 2000将临时提升等待该操作线程的优先级，以保证等待I/O操作的线程能有更多的机会立即开始处理得到的结果。
- 为了避免I/O操作导致对某些线程的不公平偏好，在I/O操作完成后唤醒等待线程时将把该线程的时间配额减1。
- 线程优先级的实际提升值是由设备驱动程序决定的。与I/O操作相关的线程优先级提升建议值在文件“Wdm.h”或“Ntddk.h”中。设备驱动程序在完成I/O请求时通过内核函数IoCompleteRequest来指定优先级提升的幅度。
- 线程优先级的提升幅度与I/O请求的响应时间要求是一致的，响应时间要求越高，优先级提升幅度越大。





线程优先级提升的建议值

设备	优先级提升值
磁盘、光驱、并口、视频	1
网络、邮件槽、命名管道、串口	2
键盘、鼠标	6
音频	8

- 线程优先级提升是以线程的基本优先级为基点的，不是以线程的当前优先级为基点。
- 当用完它的一个时间配额后，线程会降低一个优先级，并运行另一个时间配额。这个降低过程会一直进行下去，直到线程的优先级降低至原来的基本优先级。
- 优先级提升策略仅适用于可变优先级范围(0到15)内的线程。不管线程的优先级提升幅度有多大，提升后的优先级都不会超过15而进入实时优先级。





2. 等待事件和信号量后的线程优先级提升

- 当一个等待执行事件对象或信号量对象的线程完成等待后，它的优先级将提升一个优先级。
- 阻塞于事件或信号量的线程得到的处理机时间比处理机繁忙型线程要少，这种提升可减少这种不平衡带来的影响。
- SetEvent、PulseEvent或ReleaseSemaphore函数调用可导致事件对象或信号量对象等待的结束。
- 提升是以线程的基本优先级为基点的，而不是线程的当前优先级。提升后的优先级永远不会超过15。在等待结束时，线程的时间配额被减1，并在提升后的优先级上执行完剩余的时间配额；随后降低1个优先级，运行一个新的时间配额，直到优先级降低到初始的基本优先级。





3. 前台线程在等待结束后的优先级提升

- 对于前台进程中的线程，一个内核对象上的等待操作完成时，内核函数 `KiUnwaitThread` 会提升线程的当前优先级(不是线程的基本优先级)，提升幅度为变量 `PsPrioritySeparation` 的值。
- 在前台应用完成它的等待操作时小幅提升它的优先级，以使它更有可能马上进入运行状态，有效改进前台应用的响应时间特征。
- 用户不能禁止这种优先级提升，甚至是在用户已利用Win32的函数 `SetThreadPriorityBoost` 禁止了其他的优先级提升策略时，也是如此。





4. 图形用户接口线程被唤醒后的优先级提升

- 拥有窗口的线程在被窗口活动唤醒(如收到窗口消息)时将得到一个幅度为2的额外优先级提升。
- 窗口系统(Win32k.sys)在调用函数KeSetEvent时实施这种优先级提升，KeSetEvent函数调用设置一个事件，用于唤醒一个图形用户接口线程。
- 这种优先级提升的原因是改进交互应用的响应时间。





5. 对处理机饥饿线程的优先级提升

- 系统线程“平衡集管理器(balance set manager)”会每秒钟检查一次就绪队列，是否存在一直在就绪队列中排队超过300个时钟中断间隔的线程。
- 如果找到这样的线程，平衡集管理器将把该线程的优先级提升到15，并分配给它一个长度为正常值两倍的时间配额；
- 当被提升线程用完它的时间配额后，该线程的优先级立即衰减到它原来的基本优先级。





- 如果在该线程结束前出现其他高优先级的就绪线程，该线程会被放回就绪队列，并在就绪队列中超过另外300个时钟中断间隔后再次被提升优先级。
- 平衡集管理器只扫描16个就绪线程。如果就绪队列中有更多的线程，它将记住暂停时的位置，并在下一次开始时从当前位置开始扫描。
- 平衡集管理器在每次扫描时最多提升10个线程的优先级。如果在一次扫描中已提升了10个线程的优先级，平衡集管理器会停止本次扫描，并在下一次开始时从当前位置开始扫描。
- 这种算法并不能解决所有优先级倒置的问题，但它很有效。





5.6.2.9 对称多处理机系统上的线程调度

- 当Windows 2000试图调度优先级最高的可执行线程时，有几个因素会影响到处理机的选择。Windows 2000只保证一个优先级最高的线程处于运行状态。

[返回](#)





1. 亲和关系 (Affinity)

- 描述该线程可在哪些处理机上运行。
- 线程的亲和掩码是从进程的亲和掩码继承得到的。
- 缺省时，所有进程(即所有线程)的亲和掩码为系统中所有可用处理机的集合。应用程序通过调用 **SetProcessAffinityMask** 或 **SetThreadAffinityMask** 函数来指定亲和掩码；





2. 线程的首选处理机和第二处理机

- 首选处理机(ideal processor)：线程运行时的偏好处理机；
 - 线程创建后，Windows 2000系统不会修改线程的首选处理机设置；
 - 应用程序可通过SetThreadIdealProcessor函数来修改线程的首选处理机。
- 第二处理机(last processor)：线程第二个选择的运行处理机；





3. 就绪线程的运行处理机选择

- 当线程进入运行状态时，Windows 2000首先试图调度该线程到一个空闲处理机上运行。如果有多个空闲处理机，线程调度器的调度顺序为：
 - 线程的首选处理机
 - 线程的第二处理机
 - 当前执行处理机(即正在执行调度器代码的处理机)。
 - 如果这些处理机都不是空闲的，Windows 2000将依据处理机标识从高到低扫描系统中的空闲处理机状态，选择找到的第一个空闲处理机。





- 如果线程进入就绪状态时，所有处理机都处于繁忙状态，Windows 2000将检查一个处于运行状态或备用状态的线程，判断它是否可抢占。检查的顺序如下：
 - 线程的首选处理机
 - 线程的第二处理机
 - 如果这两个处理机都不在线程的亲合掩码中，Windows 2000将依据活动处理机掩码选择该线程可运行的编号最大的处理机。
- Windows 2000并不检查所有处理机上的运行线程和备用线程的优先级，而仅仅检查一个被选中处理机上的运行线程和备用线程的优先级。
- 如果在被选中的处理机上没有线程可被抢占，则新线程放入相应优先级的就绪队列，并等待调度执行。





- 如果被选中的处理机已有一个线程处于备用状态(即下一个在该处理机上运行的线程)，并且该线程的优先级低于正在检查的线程，则正在检查的线程取代原处于备用状态的线程，成为该处理机的下一个运行线程。
- 如果已有一个线程正在被选中的处理机上运行，Windows 2000将检查当前运行线程的优先级是否低于正在检查的线程；如果正在检查的线程优先级高，则标记当前运行线程为被抢占，系统会发出一个处理机间中断，以抢占正在运行的线程，让新线程在该处理机上运行。





4. 为特定的处理机调度线程

- 在多处理机系统，Windows 2000不能简单地从就绪队列中取第一个线程，它要在亲和掩码限制下寻找一个满足下列条件之一的线程。
 - 线程的上一次运行是在该处理机上；
 - 线程的首选处理机是该处理机；
 - 处于就绪状态的时间超过2个时间配额；
 - 优先级大于等于24；
- 如果Windows 2000不能找到满足要求的线程，它将从就绪队列的队首取第一个线程进入运行状态。





5. 最高优先级就绪线程可能不处于运行状态

- 有可能出现这种情况，一个比当前正在运行线程优先级更高的线程处于就绪状态，但不能立即抢占当前线程，进入运行状态。

例如：假设0号处理机上正运行着一个可在任何处理机上运行的优先级为8的线程，1号处理机上正运行着一个可在任何处理机上运行的优先级为4的线程；这时一个只能在0号处理机上运行的优先级为6的线程进入就绪状态。


在这种情况下，优先级为6的线程只能等待0号处理机上优先级为8的线程结束。因为Windows 2000不会为了让优先级为6的线程在0号处理机上运行，而把优先级为8的线程从0号处理机移到1号处理机。即0号处理机上的优先级为8的线程不会抢占1号处理机上优先级为4的线程。





5.6.2.10 空闲线程

- 如果在一个处理机上没有可运行的线程，Windows 2000/XP会调度相应处理机对应的**空闲线程**。
- 由于在多台处理机系统中可能两个处理机同时运行空闲线程，所以系统中的每个处理机都有一个对应的空闲线程。
- Windows 2000/XP给空闲线程指定的线程优先级为0，该空闲线程只在没有其他线程要运行时才运行。

名称	PID	状态	用户名	CPU	内存(专用工作...	基本优先级	线程
 系统空闲进程	0	正在运行	SYSTEM	95	8 K	无	4

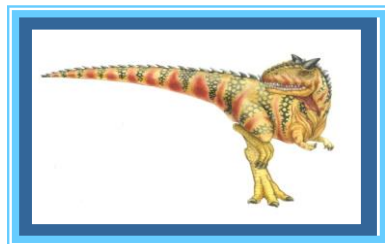
任务管理器中

[返回](#)





Linux进程调度概述

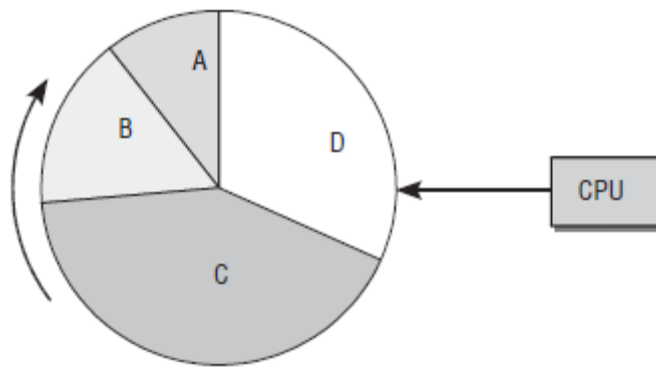




Linux的进程调度

Linux进程调度方式:

- Linux系统采用**抢占调度**方式。**Linux2.6内核是抢占式的**，这意味着进程无论是处于内核态还是用户态，都可能被抢占。
- Linux的调度基于**分时技术**（time-sharing）。对于优先级相同进程进程采用**时间片轮转法**。
- 根据进程的优先级对它们进行分类。进程的优先级是**动态的**。



不同的进程分配不同的时间片份额





Linux进程调度策略

- task_struct中与进程调度相关的一些变量有：

unsigned long policy: 进程调度策略

- Linux中现在有四种类型的调度策略：

include/uapi/linux/sched.h, (V4.x)

#define SCHED_NORMAL 0

#define SCHED_FIFO 1

#define SCHED_RR 2

#define SCHED_BATCH 3

#define SCHED_IDLE 5 //最不重要的进程

#define SCHED_DEADLINE 6 //实时进程调度算法，最早到期优先





Linux进程调度策略

- Linux的进程分为普通进程和实时进程，实时进程的优先级高于普通进程
- Linux进程的策略policy有：

符号常量	意义
SCHED_NORMAL	普通进程的时间片轮转算法
SCHED_FIFO	实时进程的先进先出算法
SCHED_RR	实时进程的时间片轮转算法
SCHED_BATCH	后台处理进程

- 普通进程按照SCHED_NORMAL调度策略进行进程调度。
- 实时进程按照SCHED_FIFO或SCHED_RR策略进行调度。
- SCHED_BATCH是2.6新加入的调度策略，这种类型的进程一般都是后台处理进程，总是倾向于跑完自己的时间片，没有交互性。所以对于这种调度策略的进程，调度器一般给的优先级比较低，这样系统就能在没什么事情做的时候运行这些进程，而一旦有交互性的进程需要运行，则立刻切换到交互性的进程，从用户的角度来看，系统的响应性/交互性就很好。





进程的调度算法

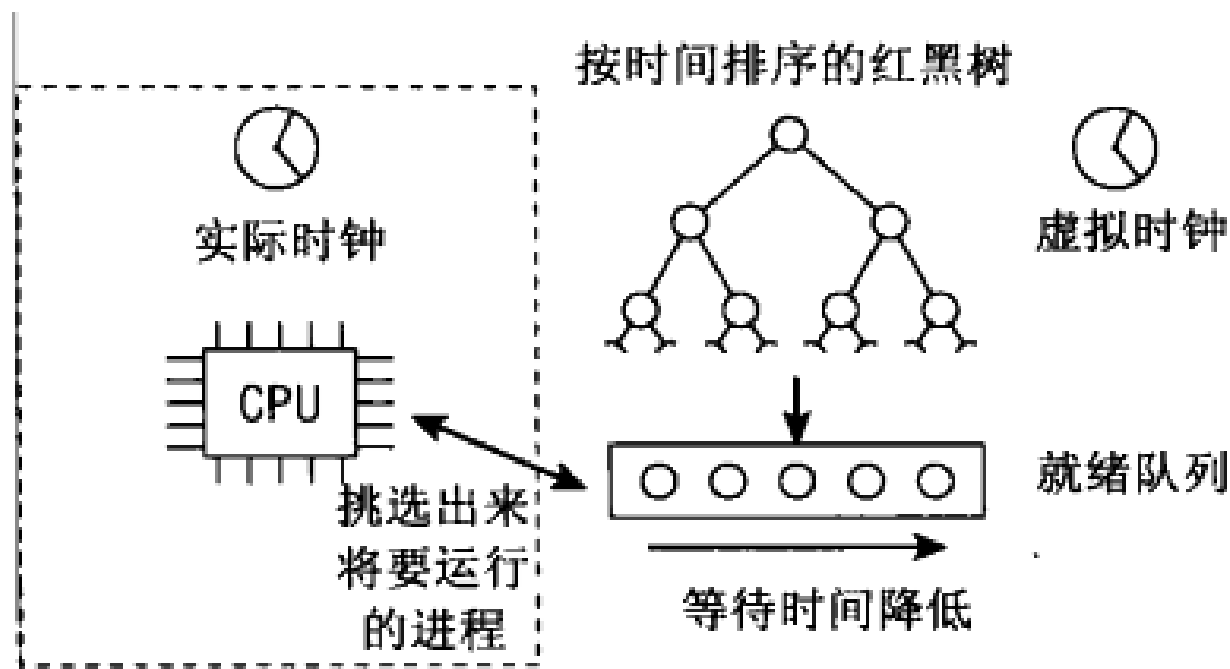
- 调度程序源代码在`kernel/sched.c`文件中。
- Linux 2.4的调度算法：
 - 遍历进程可运行队列，算法时间 $O(n)$
 - 时间片，动态优先级
- Linux 2.6.1-2.6.22 调度算法 ($O(1)$ 算法)：
 - 优先级0-139，使用active和expire两个队列，按照优先级调度，算法时间 $O(1)$
- Linux 2.6.23- 调度算法 (CFS算法)：
 - 非实时：CFS (完全公平调度器) 进程调度器，使用红黑树选取下一个被调度进程， $O(\lg N)$
 - 实时：优先级队列





一、CFS调度算法

- 在Linux 2.6.23 内核中发布了一种新的调度算法**CFS算法**（Completely Fair Scheduler 完全公平调度算法）。



！ 调度器通过将进程在红黑树中排序，跟踪进程的等待时间





二、Linux 2.6 O(1) 进程调度

■ 特性：

- ▶ O(1)调度算法，调度器开销恒定（与当前系统负载无关），实时性能更好
- ▶ 高可扩展性
- ▶ 新设计的 SMP 亲和方法
- ▶ 优化计算密集型的批处理作业的调度
- ▶ 重载条件下调度器工作更平滑
- ▶ 子进程先于父进程运行等其他改进





Linux 2.6 0(1) 进程调度

- Linux 2.6的进程设置140个优先级。实时进程优先级为0-99，普通进程优先级100-139的数。0为最高优先权，139为最低优先权。
- 优先级分为静态优先级和动态优先级
- 调度程序根据动态优先级来选择新进程运行。
- 基本时间片：
 - 静态优先权本质上决定了进程的基本时间片，即进程用完了以前的时间片时，系统分配给进程的时间片长度。静态优先权和基本时间片的关系用下列公式确定：
base time quantum (ms) :
 - ▶ $(140 - \text{static_priority}) * 20$ if $\text{static_priority} < 120$
 - ▶ $(140 - \text{static_priority}) * 5$ if $\text{static_priority} \geq 120$





task_struct 相关域

■ sleep_avg

- 进程的平均等待时间 (nanosecond)
- 反映交互式进程优先与分时系统的公平共享
- 值越大，计算出来的进程优先级也越高

■ run_list

- 串连在优先级队列中
- 优先级数组prio_array中按顺序排列了各个优先级下的所有进程
- 调度器在prio_array中找到相应的run_list，从而找到其宿主结构task_struct





task_struct 相关域

■ time_slice

- 进程的运行时间片剩余大小
- 进程的默认时间片与进程的静态优先级相关
- 进程创建时，与父进程平分时间片
- 运行过程中递减，一旦归零，则重置时间片，并请求调度
- 递减和重置在时钟中断中进行（scheduler_tick()）
- 进程退出时，如果自身并未被重新分配时间片，则将自己剩余的时间片返还给父进程





task_struct 相关域

■ static_prio 静态优先级

- 与2.4版本中的nice值意义相同，但取值区间不同，是用户可影响的优先级
- 通过set_user_nice()来改变
- $\text{static_prio} = \text{MAX_RT_PRIO} + \text{nice} + 20$
 - ▶ MAX_RT_PRIO 定义为100
- 进程初始时间片的大小仅决定于进程的静态优先级
- 核心将100~139的优先级映射到200ms~10ms的时间片上
- 优先级数值越大，优先级越低，分配的时间片越少
- 实时进程的static_prio不参与优先级prio的计算





task_struct 相关域

■ prio 动态优先级

- 相当于 2.4 中 goodness() 的计算结果，在 0~MAX_PRIO-1 之间取值（MAX_PRIO 定义为 140），其中：
 - ▶ 0~MAX_RT_PRIO-1（MAX_RT_PRIO 定义为 100）属于实时进程范围；
 - ▶ MAX_RT_PRIO~MAX_PRIO-1 属于非实时进程。数值越大，表示进程优先级越小。
- 2.6 中，动态优先级不再统一在调度器中计算和比较，而是独立计算，并存储在进程的 task_struct 中，再通过描述的 priority_array 结构自动排序。
- 普通进程 $\text{prio} = \max(100, \min(\text{static priority} - \text{bonus} + 5, 139))$
- Bonus 是范围从 0 到 10 的值，bonus 的值小于 5 表示降低动态优先权以示惩罚，bonus 的值大于 5 表示增加动态优先权以示额外奖赏。Bonus 的值依赖于进程过去的情况，说得更准确一些是与进程的平均睡眠时间相关。
- prio 的计算和很多因素相关。
 - ▶ 请参阅 [Understanding The Linux Kernel, 3rd Edition](#)





task_struct 相关域

- unsigned long **rt_priority**
 - 实时进程的优先级
 - sys_sched_setschedule()
 - 一经设定在运行时不变，作为其动态优先级
- prio_array_t ***array**
 - 记录当前CPU活动的就绪队列
 - 以优先级为序组成数组





runqueue结构 (kernel/sched.c)

- runqueue结构是 Linux2.6调度程序最重要的数据结构。系统中的每个 CPU 都有它自己的运行队列，所有的 runqueue 结构存放在 runqueues 每 CPU (per-CPU) 变量中。
- 系统中的每个可运行进程属于且只属于一个运行队列。只要可运行进程保持在同一个运行队列中，它就只可能在拥有该运行队列的 CPU 上执行。但是，可运行进程会从一个运行队列迁移到另一个运行队列。





The fields of the runqueue structure

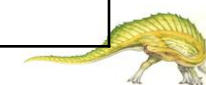
Type	Name	Description
spinlock_t	lock	保护进程链表的自旋锁
unsigned long	nr_running	运行队列链表中可运行进程的数量
unsigned long	cpu_load	基于运行队列中进程的平均数量的CPU负载因子
unsigned long	nr_switches	CPU执行进程切换的次数
unsigned long	nr_uninterruptible	先前在运行队列链表中而现在睡眠在TASK_UNINTERRUPTIBLE状态的进程的数量（对所有运行队列来说，这些字段的总数才是有意义的）
unsigned long	expired_timestamp	过期队列中最老的进程被插入队列的时间。
unsigned long long	timestamp_last_tick	最近一次定时器中断的时间戳的值
task_t *	curr	当前正在运行进程的进程描述符指针（对本地CPU，它与current相同）
task_t *	idle	当前CPU（this CPU）上交换进程的进程描述符指针。





The fields of the runqueue structure

struct mm_struct *	prev_mm	在进程切换期间用来存放被替换进程的内存描述符的地址
prio_array_t *	active	指向活动进程链表的指针
prio_array_t *	expired	指向过期进程链表的指针
prio_array_t [2]	arrays	活动和过期进程的两个集合
int	best_expired_prio	过期进程中静态优先权最高的进程（权值最小）。
atomic_t	nr_iowait	先前在运行队列的链表中而现在正等待磁盘I/O 操作结束的进程的数量。
struct sched_domain *	sd	指向当前CPU 的基本调度域
int	active_balance	如果要把一些进程从本地运行队列迁移到另外的运行队列（平衡运行队列），就设置这个标志。
int	push_cpu	未使用
task_t *	migration_thread	迁移内核线程的进程描述符指针。
struct list_head	migration_queue	从运行队列中被删除的进程的链表





The fields of the runqueue structure

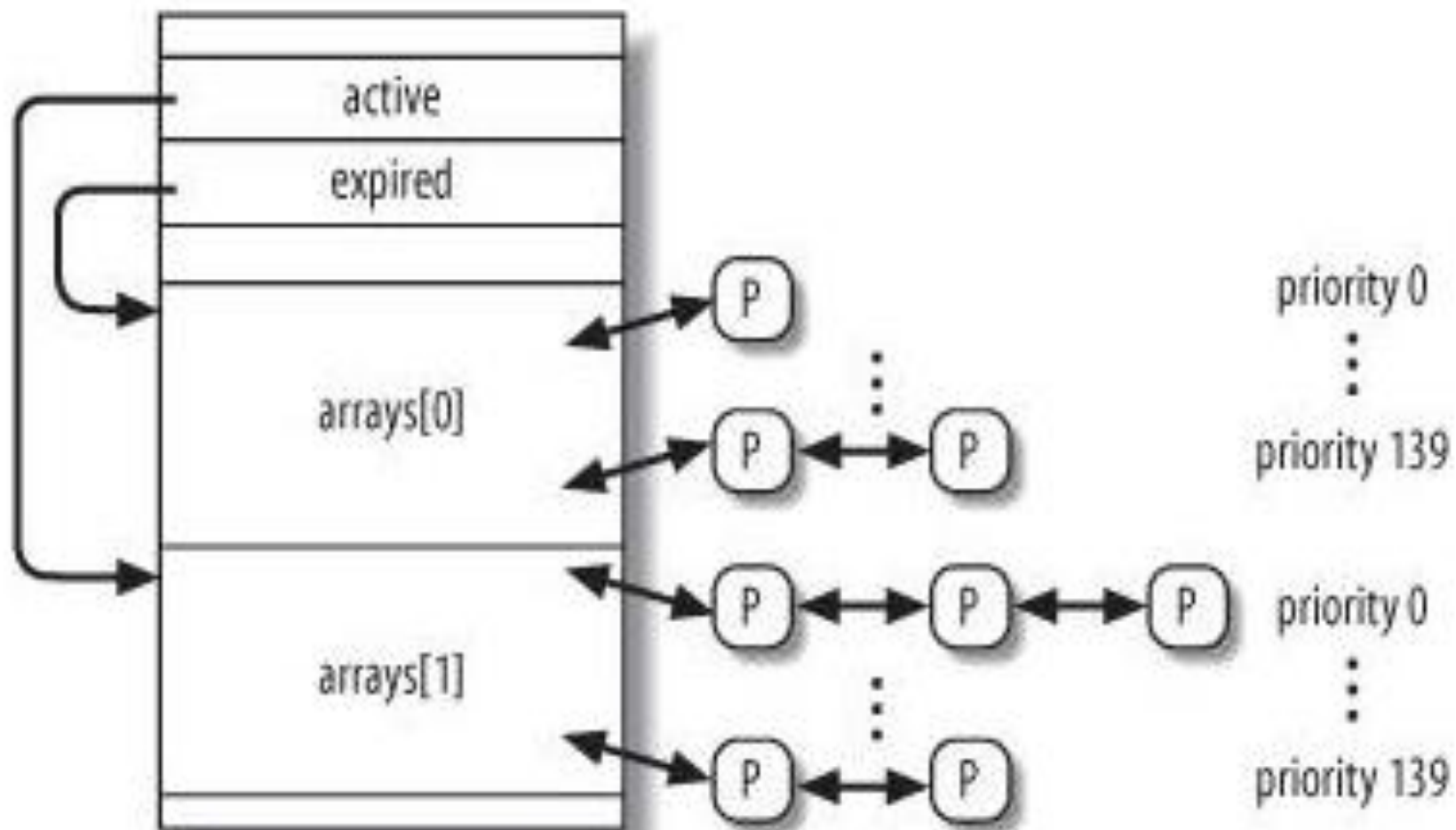
■ runqueue结构的字段：

● prio_array_t ***active**, ***expired**, **arrays**[2]

- ▶ 每个CPU均有两个具有优先级的队列，按时间片是否用完分为“**活动队列**”（**active**指针所指）和“**过期队列**”（**expired**指针所指）。**active** 指向时间片没用完、当前可被调度的就绪进程，**expired** 指向时间片已用完的就绪进程。
- ▶ 每一类队列用一个**struct prio_array**表示（优先级排序数组）
- ▶ 一个任务的时间片用完后，它会被转移到“过期”的队列中
- ▶ 在该队列中，任务仍然是按照优先级排好序的
- ▶ 当活动队列中的任务均被执行完时，就交换两个指针

这种调度方法会产生什么问题？







prio_array

- 每一类就绪进程都用一个 struct prio_array 的结构表示

```
struct prio_array{
```

```
    unsigned int nr_active; /*本进程组中的进程数*/
```

```
    unsigned long bitmap[BITMAP_SIZE];/*加速HASH表访问的位图快速定位第一个非空的  
    就绪进程链表*/
```

```
    struct list_headqueue[MAX_PRIO];/*以优先级为索引的HASH表*/
```

```
}
```





schedule()函数

- 进程调度由schedule()函数实现。首先，schedule()利用下面的代码定位优先级最高的就绪进程：
 - `prey=current;`
 - `array=rq->active;`
 - `idx=sched_find_first_bit(array->bitmap):`
 - `queue= array->queue+idx;`
 - `next =list_entry(queue->next, struct task_struct, run_list);`





schedule()函数

- `schedule()`通过调用`sched_find_first_bit()`函数找到当前CPU就绪进程队列`runqueue`的`active`进程数组中第一个非空的就绪进程链表。这个链表中的进程具有最高的优先级，`schedule()`选择链表中的第一个进程作为调度器下一时刻将要运行的进程。
- 如果`prev`(当前进程)和`next`(将要运行的进程)不是同一个进程，`schedule()`调用`context_switch()`将CPU切换到`next`进程运行。





调度程序所使用的函数

- 调度程序依靠几个函数来完成调度工作，其中最重要的函数有：
 - `scheduler_tick()`，维持当前最新的 `time_slice` 计数器
 - `try_to_wake_up()`，唤醒睡眠进程
 - `recalc_task_prio()`，更新进程的动态优先权
 - `schedule()`，选择要被执行的新进程
 - `load_balance()`，维持多处理器系统中运行队列的平衡。





三、Linux进程调度算法（2.4）

- Linux的进程调度程序是内核中的`schedule()`函数。`schedule()`函数需要确定：
 - **policy** 进程的调度策略
 - ▶ `SCHED_FIFO`、`SCHED_RR`的是实时进程，
 - ▶ `SCHED_NORMAL`是普通进程。
 - **rt_priority** 实时进程优先级。Linux2.6默认的变化范围[0, 99]。
 - **nice** 普通进程的优先级，取值[-20, 19]，默认0;nice为-20的进程获得时间片最长。2.6为**static_prio**，静态优先级，不参与优先级计算
 - **counter** 进程使用CPU运行时间的计数值(保存在jiffies 中)。进程首次运行时为进程优先级的数值，它随时间变化递减，随时钟滴答减1（即时钟中断的间隔时间，为10ms或1ms）。**counter=0**时，进程的时间片已用完。





Linux进程调度方法（2.4）

- Linux进程调度采用动态优先级法，调度对象是**可运行队列**，**每个处理器有一个可运行队列**；可运行队列中优先级大的进程首先得到CPU投入运行。
- 进程在运行中，**counter**的代表进程的动态优先级。
- Linux采取了加权的方法来保证实时进程优先于普通进程。
- **普通进程的权值就是它的counter的值，而实时进程的权值是它的rt_priority的值加1000（2.4）。**
- 在调度过程中，调度程序检查可运行队列中所有进程的权值，选择其中权值最大的进程做为下一个运行进程。





进程切换时机

当前进程放弃CPU的情况可以分为两种。

■ 进程主动地放弃CPU。大体可以分为两类：

- 隐式地主动放弃CPU。往往是因为需要的资源目前不能获取，如执行`read()`等系统调用的过程中，这种情况下的处理过程如下：
 - ▶ 将进程加入合适的等待队列。
 - ▶ 把当前进程的状态改为`TASK_INTERRUPTIBLE`或`TASK_UNINTERRUPTIBLE`。
 - ▶ 调用`schedule()`函数，该函数的执行结果往往是当前进程放弃CPU。
 - ▶ 检查资源是否可用，如果不可用，则跳转到第2)步。
 - ▶ 资源已可用，将该进程从等待队列中移去。
- 进程显式地主动放弃CPU，如系统调用`sched_yield()`，`sched_setscheduler()`，`pause()` 及 `nanosleep()` 均会导致当前进程让出CPU。





进程切换时机

- **进程被动放弃CPU。**下面两种情形均会导致当前进程控制块的need_resched被置1。
 - 当前进程的时间片已经用完；
 - 刚被唤醒进程的优先级别高于当前进程。





进程的调度时机

■ 进程的调度时机分成两种情形：

- 直接调用`schedule()`调度函数，例如进程主动放弃CPU的第一类情形；
- 间接调用`schedule()`。

■ 例如，进程被动放弃CPU的情形。当进程控制块的`need_resched`被置1时，并不立即直接调用`schedule()`调度函数。而是在随后的某个时刻，当进程从内核态返回用户态之前检查`need_resched`是否为1，如果为1，则调用`schedule()`调度函数。





进程的调度算法(2.4)

- 调度程序源代码见kernel/sched.c文件。调度程序的算法和相关的代码大部分都在2.5版内核的开发版中备重写了。
- 核心函数是schedule(), 该函数的任务是选出一个可运行的进程。
- 对于普通进程来讲CPU时间的分配是典型的时间片策略。
- 在某个时刻, 运行队列中的每个进程都有一个counter值, 当运行队列中进程的counter值都变为0以后, 表明一轮已经结束, 每个进程的counter根据其nice重新赋值, 开始新一轮执行过程。拥有CPU的进程每次“时钟滴答”counter值减一。
- Linux 2.4内核设定“时钟滴答”是10ms, 普通进程counter初值为21 (计算公式: $20 - (\text{nice}) + 1$), 所以默认的时间片为210ms。





schedule()函数执行过程:

- 1) 检查是否有软中断服务请求, 如果有, 则先执行这些请求。
- 2) 若当前进程调度策略是SCHED_RR, 且 counter为0, 则将该进程移到可执行进程队列的尾部并对counter重新赋值。
- 3) 检查当前进程的状态, 如为 TASK_INTERRUPTIBLE, 且该进程有信号接收, 则将进程状态置为 TASK_RUNNING。
- 4) 当前进程的状态不是TASK_RUNNING, 则将其从可执行进程队列中移出, 然后将当前进程控制块的need_resched恢复成0。





schedule()函数执行过程

- 5) 进入函数的核心部分。可运行进程队列的每个进程都将被计算出一个权值，主要是利用 **goodness()** 函数。最终最大的权值保存在变量 **c** 中，与之对应的进程控制块保存在变量 **next** 中。
- 6) 检查 **c** 是否为0。若为0则表明所有可执行进程的时间配额都已用完，因而对所有进程的 **counter** 重新计算（赋值）。
- 7) 如果 **next** 进程就是当前进程，则结束 **shedule()** 的运行。否则进行进程切换，CPU改由 **next** 进程占据。





goodness()函数

- **goodness()** 函数计算进程的当前权值。该函数的第一个参数是待估进程的控制块。
 - 如果该进程是实时进程，它的权值为 $1000 + \text{rt_priority}$ ，1000是普通进程权值无法到达的数字，因而实时进程总可以优先得到执行。
 - 对于普通进程，它的权值为counter，如果其又是内核线程，由于无需切换用户空间，则将权值加一作为奖励。





goodness()

```
static inline int goodness(struct task_struct * p, int this_cpu, struct mm_task *this_mm)
{
    int weight;
    if (p->policy & SCHED_YIELD)
        goto out          //如果设置的是SCHED_YIELD标志，则p自愿放弃CPU
    if (p->policy == SCHED_NORMAL) {
        //对于非实时进程，则返回当前的counter值
        weight = p->counter;
        if(!weight)
            goto out; //如果权值weight为0，则进程的时间片以用完，直接返回。
        .....      //对称多处理机代码
        if(p->mm==this_mm || !p->mm)
            //对于使用同一struct mm_task 结构的进程
            weight+=1; //将增加少许权值，因为他们使用同一页表，
            goto out; //减少在MMU上进行的上下文切换所花费的时间开销
    }
    weight= 1000 + p->rt_priority;
    //如果p是实时进程，则权值的weight基值为1000。
out:
```





与调度相关的其他系统调用

■ 系统调用

`nice()`

`getpriority()`

`setpriority()`

`sched_getscheduler()`

`sched_setscheduler()`

`sched_getparam()`

`sched_setparam()`

`sched_yield()`

`sched_get_priority_min()`

`sched_get_priority_max()`

描述

改变一个普通进程的优先级

取得一组普通进程的最大优先级

设置一组普通进程的优先级

取得一个进程的调度策略

设置一个进程的调度策略和优先级

取得一个进程的调度优先级

设置一个进程的优先级

不阻塞的情况下自愿放弃处理机

取得某种策略的最小优先级

取得某种策略的最大优先级





Q&A

