第五章 处理机管理

处理机管理的工作是对CPU资源进行合理的分配使用，以提高处理机利用率，并使各用户公平地得到处理机资源。这里的主要问题是处理机调度算法和调度算法特征分析。

5.1 引言

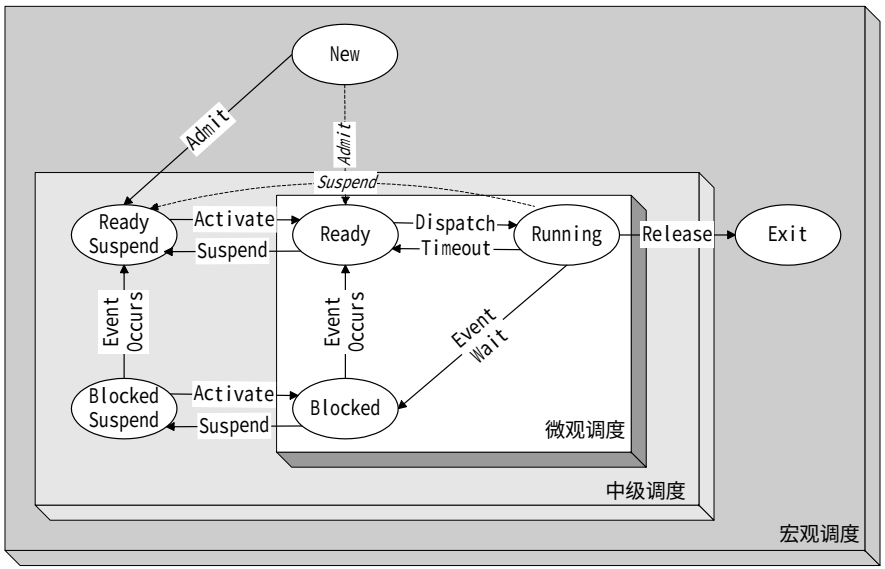
返回

5.1.1 调度的类型(scheduling)

从处理机调度的对象、时间、功能等不同角度，我们可把处理机调度分成不同类型。

1. 按照调度的层次

- 作业：又称为"宏观调度"、"高级调度"。从用户工作流程的角度，一次提交的若干个流程，其中每个程序按照进程调度。时间上通常是分钟、小时或天。
- 内外存交换：又称为"中级调度"。从存储器资源的角度。将进程的部分或全部换出到外存上，将当前所需部分换入到内存。指令和数据必须在内存里才能被CPU直接访问。
- 进程或线程：又称为"微观调度"、"低级调度"。从CPU资源的角度，执行的单位。时间上通常是毫秒。因为执行频繁，要求在实现时达到高效率。



处理机调度的层次

2. 按照调度的时间周期

- 长期(long-term)：将进程投入"允许执行"进程缓冲池中，或送到"退出"进程缓冲池中。进程状态：New －>Ready suspend, Running －>Exit
- 中期(medium-term)：将进程的部分或全部加载到内存中。进程状态：Ready <－>Ready suspend, Blocked <－>Blocked suspend
- 短期(short-term)：选择哪个进程在处理机上执行。进程状态：Ready <－>Running
- I/O调度：选择哪个I/O等待进程，使其请求可以被空闲的I/O设备进行处理。

3. 按照OS的分类

- 批处理调度－－应用场合：大中型主机集中计算，如工程计算、理论计算（流体力学）
- 分时调度、实时调度：通常没有专门的作业调度
- 多处理机调度

5.1.2 调度的性能准则

我们可从不同的角度来判断处理机调度算法的性能，如用户的角度、处理机的角度和算法实现的角度。实际的处理机调度算法选择是一个综合的判断结果。

1. 面向用户的调度性能准则

- 周转时间：作业从提交到完成（得到结果）所经历的时间。包括：在收容队列中等待，CPU上执行，就绪队列和阻塞队列中等待，结果输出等待－－批处理系统
  - 平均周转时间T
  - 平均带权周转时间（带权周转时间W是 T(周转)/T(CPU执行)〕
- 响应时间：用户输入一个请求（如击键）到系统给出首次响应（如屏幕显示）的时间－－分时系统
- 截止时间：开始截止时间和完成截止时间－－实时系统，与周转时间有些相似。
- 公平性：不因作业或进程本身的特性而使上述指标过分恶化。如长作业等待很长时间。
- 优先级：可以使关键任务达到更好的指标。

2. 面向系统的调度性能准则

- 吞吐量：单位时间内所完成的作业数，跟作业本身特性和调度算法都有关系－－批处理系统
  - 平均周转时间不是吞吐量的倒数，因为并发执行的作业在时间上可以重叠。如：在2小时内完成4个作业，而每个周转时间是1小时，则吞吐量是2个作业/小时
- 处理机利用率：－－大中型主机
- 各种设备的均衡利用：如CPU繁忙的作业和I/O繁忙（指次数多，每次时间短）的作业搭配－－大中型主机

3. 调度算法本身的调度性能准则

- 易于实现
- 执行开销比

5.1.3 进程调度

- 功能：调度程序(dispatcher)
  - 记录所有进程的运行状况（静态和动态）
  - 当进程出让CPU或调度程序剥夺执行状态进程占用的CPU时，选择适当的进程分派CPU
  - 完成上下文切换
- 进程的上下文切换过程：
  - 用户态执行进程A代码－－进入OS核心（通过时钟中断或系统调用）
  - 保存进程A的上下文，恢复进程B的上下文（CPU寄存器和一些表格的当前指针）
  - 用户态执行进程B代码
- 注：上下文切换之后，指令和数据快速缓存cache通常需要更新，执行速度降低

5.2 调度算法

通常将作业或进程归入各种就绪或阻塞队列。有的算法适用于作业调度，有的算法适用于进程调度，有的两者都适应。

返回

5.2.1 先来先服务 (FCFS, First Come First Service)

这是最简单的调度算法，按先后顺序进行调度。

1. FCFS算法

- 按照作业提交或进程变为就绪状态的先后次序，分派CPU；
- 当前作业或进程占用CPU，直到执行完或阻塞，才出让CPU（非抢占方式）。
- 在作业或进程唤醒后（如I/O完成），并不立即恢复执行，通常等到当前作业或进程出让CPU。最简单的算法。

2. FCFS的特点

- 比较有利于长作业，而不利于短作业。
- 有利于CPU繁忙的作业，而不利于I/O繁忙的作业。

5.2.2 短作业优先 (SJF, Shortest Job First)

又称为“短进程优先”SPN(Shortest Process Next)；这是对FCFS算法的改进，其目标是减少平均周转时间。

1. SJF算法

对预计执行时间短的作业（进程）优先分派处理机。通常后来的短作业不抢先正在执行的作业。

2. SJF的特点

- 优点：
  - 比FCFS改善平均周转时间和平均带权周转时间，缩短作业的等待时间；
  - 提高系统的吞吐量；
- 缺点：
  - 对长作业非常不利，可能长时间得不到执行；
  - 未能依据作业的紧迫程度来划分执行的优先级；
  - 难以准确估计作业（进程）的执行时间，从而影响调度性能。

3. SJF的变型

- "最短剩余时间优先"SRT(Shortest Remaining Time)
  - 允许比当前进程剩余时间更短的进程来抢占
- "最高响应比优先"HRRN(Highest Response Ratio Next)
  - 响应比R = (等待时间 + 要求执行时间) / 要求执行时间
  - 是FCFS和SJF的折衷

5.2.3 时间片轮转(Round Robin)算法

前两种算法主要用于宏观调度，说明怎样选择一个进程或作业开始运行，开始运行后的作法都相同，即运行到结束或阻塞，阻塞结束时等待当前进程放弃CPU 。本算法主要用于微观调度，说明怎样并发运行，即切换的方式；设计目标是提高资源利用率。

其基本思路是通过时间片轮转，提高进程并发性和响应时间特性，从而提高资源利用率；

1. 时间片轮转算法

- 将系统中所有的就绪进程按照FCFS原则，排成一个队列。
- 每次调度时将CPU分派给队首进程，让其执行一个时间片。时间片的长度从几个ms到几百ms。
- 在一个时间片结束时，发生时钟中断。
- 调度程序据此暂停当前进程的执行，将其送到就绪队列的末尾，并通过上下文切换执行当前的队首进程。
- 进程可以未使用完一个时间片，就出让CPU（如阻塞）。

2. 时间片长度的确定

- 时间片长度变化的影响
  - 过长－>退化为FCFS算法，进程在一个时间片内都执行完，响应时间长。
  - 过短－>用户的一次请求需要多个时间片才能处理完，上下文切换次数增加，响应时间长。
- 对响应时间的要求：
  - T(响应时间)=N(进程数目)*q(时间片)
- 时间片长度的影响因素：
  - 就绪进程的数目：数目越多，时间片越小（当响应时间一定时）
  - 系统的处理能力：应当使用户输入通常在一个时间片内能处理完，否则使响应时间，平均周转时间和平均带权周转时间延长。

5.2.4 多级队列算法 (Multiple-level Queue)

本算法引入多个就绪队列，通过各队列的区别对待，达到一个综合的调度目标；

- 根据作业或进程的性质或类型的不同，将就绪队列再分为若干个子队列。
- 每个作业固定归入一个队列。
- 各队列的不同处理：不同队列可有不同的优先级、时间片长度、调度策略等。如：系统进程、用户交互进程、批处理进程等。

5.2.5 优先级算法(Priority Scheduling)

本算法是多级队列算法的改进，平衡各进程对响应时间的要求。适用于作业调度和进程调度，可分成抢先式和非抢先式；

5.2.5.1 静态优先级

5.2.5.2 动态优先级

5.2.5.3 线性优先级调度算法（SRR, Selfish Round Robin）

5.2.5.1 静态优先级

创建进程时就确定，直到进程终止前都不改变。通常是一个整数。

- 依据：
  - 进程类型（系统进程优先级较高）
  - 对资源的需求（对CPU和内存需求较少的进程，优先级较高）
  - 用户要求（紧迫程度和付费多少）

5.2.5.2 动态优先级

在创建进程时赋予的优先级，在进程运行过程中可以自动改变，以便获得更好的调度性能。如：

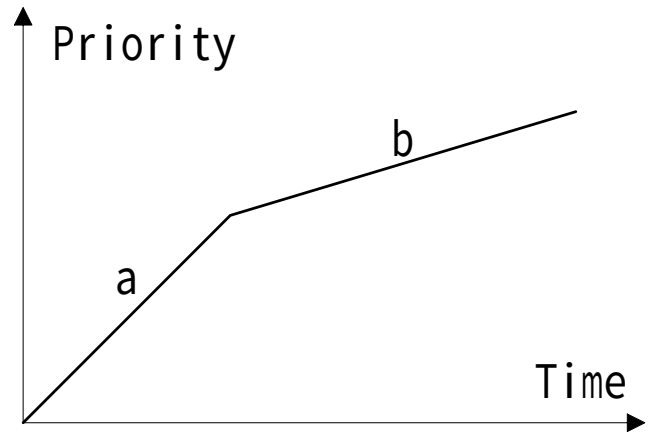
- 在就绪队列中，等待时间延长则优先级提高，从而使优先级较低的进程在等待足够的时间后，其优先级提高到可被调度执行；
- 进程每执行一个时间片，就降低其优先级，从而一个进程持续执行时，其优先级降低到出让CPU。

5.2.5.3 线性优先级调度算法（SRR, Selfish Round Robin）

本算法是优先级算法的一个实例，它通过进程优先级的递增来改进长执行时间进程的周转时间特征。

1. SRR算法

- 就绪进程队列分成两个：
  - 新创建进程队列：按FCFS方式排成；进程优先级按速率a增加；
  - 享受服务队列：已得到过时间片服务的进程按FCFS方式排成；进程优先级按速率b增加；
- 新创建进程等待时间的确定：当新创建进程优先级与享受服务队列中最后一个进程优先级相同时，转入享受服务队列；



SRR算法的优先级变化

2. SRR算法与FCFS算法和时间片轮转算法的关系

- 当b>a>0时，享受服务队列中永远只有一个进程；SRR算法退化成FCFS算法；
- 当a>b=0时，SRR算法就是时间片轮转算法；

5.2.6 多级反馈队列算法 (Round Robin with Multiple Feedback)

- 多级反馈队列算法是时间片轮转算法和优先级算法的综合和发展。优点：
  - 为提高系统吞吐量和缩短平均周转时间而照顾短进程
  - 为获得较好的I/O设备利用率和缩短响应时间而照顾I/O型进程
  - 不必估计进程的执行时间，动态调节

1. 多级反馈队列算法

- 设置多个就绪队列，分别赋予不同的优先级，如逐级降低，队列1的优先级最高。每个队列执行时间片的长度也不同，规定优先级越低则时间片越长，如逐级加倍
- 新进程进入内存后，先投入队列1的末尾，按FCFS算法调度；若按队列1一个时间片未能执行完，则降低投入到队列2的末尾，同样按FCFS算法调度；如此下去，降低到最后的队列，则按"时间片轮转"算法调度直到完成。
- 仅当较高优先级的队列为空，才调度较低优先级的队列中的进程执行。如果进程执行时有新进程进入较高优先级的队列，则抢先执行新进程，并把被抢先的进程投入原队列的末尾。

2. 几点说明

- I/O型进程：让其进入最高优先级队列，以及时响应I/O交互。通常执行一个小时间片，要求可处理完一次I/O请求的数据，然后转入到阻塞队列。
- 计算型进程：每次都执行完时间片，进入更低级队列。最终采用最大时间片来执行，减少调度次数。
- I/O次数不多，而主要是CPU处理的进程：在I/O完成后，放回优先I/O请求时离开的队列，以免每次都回到最高优先级队列后再逐次下降。
- 为适应一个进程在不同时间段的运行特点，I/O完成时，提高优先级；时间片用完时，降低优先级；

5.3 调度算法性能分析

调度算法的性能通常是通过实验或计算得到的。

- FCFS, Round Robin, 线性优先级算法 SRR(Selfish Round Robin)
- 周转时间
  - 长作业时：T(FCFS) < T(SRR) < T(RR)（运行时间是主要因素）
  - 短作业时：T(RR) < T(SRR) < T(FCFS)（等待时间是主要因素）

返回

5.4 实时调度

返回

5.4.1 实时调度的特点

- 要求更详细的调度信息：如，就绪时间、开始或完成截止时间、处理时间、资源要求、绝对或相对优先级（硬实时或软实时）。
- 采用抢先式调度。
- 快速中断响应，在中断处理时（硬件）关中断的时间尽量短。
- 快速上下文切换：相应地采用较小的调度单位（如线程）。

5.4.2 实时调度算法

- 静态表调度算法（Static table-driven scheduling）：适用于周期性的实时应用。通过对所有周期性任务的分析预测（到达时间、运行时间、结束时间、任务间的优先关系），事先确定一个固定的调度方案。这种方法的特点是有效但不灵活。
- 静态优先级调度算法（Static priority-driven scheduling）：把通用的优先级调度算法用于实时系统，但优先级的确定是通过静态分析（运行时间、到达频率）完成的。

- 动态分析调度算法（Dynamic planning-based scheduling）：在任务下达后执行前进行调度分析，要求满足实时性要求。
- 无保障动态调度算法（Dynamic best effort scheduling）：在任务下达时分配优先级，开始执行，在时限到达时未完成的任务被取消。用于非周期性任务的实时系统。

5.5 多处理机调度

返回

5.5.1 与单处理机调度的区别

- 注重整体运行效率（而不是个别处理机的利用率）
- 更多样的调度算法
- 多处理机访问OS数据结构时的互斥（对于共享内存系统）
- 调度单位广泛采用线程

5.5.2 对称式多处理系统(SMP)

按控制方式，SMP调度算法可分为集中控制和分散控制。下面所述静态和动态调度都是集中控制，而自调度是分散控制。

- 集中控制
  - 静态分配(static assignment)：每个CPU设立一个就绪队列，进程从开始执行到完成，都在同一个CPU上。
    - 优点：调度算法开销小。
    - 缺点：容易出现忙闲不均。
  - 动态分配(dynamic assignment)：各个CPU采用一个公共就绪队列，队首进程每次分派到当前空闲的CPU上执行。

- 分散控制
  - 自调度(self-scheduling)：各个CPU采用一个公共就绪队列，每个处理机都可以从队列中选择适当进程来执行。需要对就绪队列的数据结构进行互斥访问控制。是最常用的算法，实现时易于移植采用单处理机的调度技术。
    - 变型：Mach OS中局部和全局就绪队列相结合，其中局部就绪队列中的线程优先调度。

5.5.3 非对称式多处理系统(ASMP)的处理机调度

- 主－从处理机系统，由主处理机管理一个公共就绪队列，并分派进程给从处理机执行。
- 各个处理机有固定分工，如执行OS的系统功能，I/O处理，应用程序。

5.5.4 成组调度(gang scheduling)

将一个进程中的一组线程，每次分派时同时到一组处理机上执行，在剥夺处理机时也同时对这一组线程进行。

- 优点
  - 通常这样的一组线程在应用逻辑上相互合作，成组调度提高了这些线程的执行并行度，有利于减少阻塞和加快推进速度，最终提高系统吞吐量。
  - 每次调度可以完成多个线程的分派，在系统内线程总数相同时能够减少调度次数，从而减少调度算法的开销

5.5.5 专用处理机调度 (dedicated processor assignment)

- 为进程中的每个线程都固定分配一个CPU，直到该线程执行完成。
- 缺点：线程阻塞时，造成CPU的闲置。优点：线程执行时不需切换，相应的开销可以大大减小，推进速度更快。
- 适用场合：CPU数量众多的高度并行系统，单个CPU利用率已不太重要。

5.6 调度算法举例

返回

5.6.1 传统UNIX的进程调度

未设置作业调度，进程调度采用基于时间片的多级反馈队列算法，进程优先级分为核心优先级和用户优先级。

1. 调度时机

- 调度由0号进程完成（始终在核心态执行，与其他进程并不完全一样）。时机：
- 进程由核心态转入用户态时：
  - 在每次执行核心代码之后返回用户态之前，检查各就绪进程的优先级并进行调度。如中断——进程回到就绪队列
- 进程主动放弃处理机时：
  - 进程申请系统资源而未得到满足（如read），或进行进程间同步而暂停（如wait或pause），
  - 进程退出（如exit）——进程进入阻塞队列或exit状态。

2. 调度标志

- UNIX System V中有三个与调度有关的标志：
  - runrun：表示要求进行调度，当发现有就绪进程优先级高于当前进程时，设置该标识。在wakeup, setrun, setpri（设置优先级）过程和时钟中断处理例程进行设置。
  - runin：表示内存中没有适当的进程可以换出或内存无足够空间换入一个外存就绪进程。（内存紧张）
  - runout：表示外存交换区中没有适当的进程可以换入。（交换区紧张）

3. 用户优先级

- 进程在用户态和核心态的优先级是不同的，这里说的是用户态进程的优先级。它是基于执行时间的动态优先级，进程优先级可为0~127之间的任一整数。优先数越大，优先级越低。
  - 0~49之间的优先级为系统内核保留
  - 用户态下的进程优先级为50~127之间

在UNIX System V中，进程优先数：

$$P\_pri = P\_CPU / 2 + PUSER + P\_nice + NZERO$$

- 系统设置部分：PUSER和NZERO是基本用户优先数的阈值，分别为25和20
- CPU使用时间部分：P_CPU表示该进程最近一次CPU使用时间。每次时钟中断则该值加1（最多可达80）。如果时钟中断的周期为16.6ms，则每秒钟过后将该值为60。
  - 新创建进程的P_CPU值为0，因而具有较高的优先级。不同系统对P_CPU的计算方法有所不同。有的固定一个因子，有的会考虑系统负荷。
- 用户设置部分：P_nice是用户可以通过系统调用设置的一个优先级偏移值。默认为20。超级用户可以设置其在0到39之间，而普通用户只能增大该值（即降低优先级）。

4. 核心优先级

- 内核把进程阻塞事件与一个睡眠优先级（0~49）联系起来；当进程从阻塞中醒来时，可及时进行处理。如：
  - 磁盘I/O操作对应的睡眠优先级为20
  - 终端输入操作对应的睡眠优先级为29。
- 核心优先级分为可中断和不可中断两类优先级。当一个软中断信号到达时，若进程正在可中断优先级上阻塞，则进程立即被唤醒；若正在不可中断优先级上，则继续阻塞。其中：
  - 不可中断优先级：对换，等待磁盘I/O，等待缓冲区，等待文件索引结点——关键操作，应该很快完成
  - 可中断优先级：等待tty（虚终端）I/O，等待子进程退出

5. 调度的实现：分三个阶段

- 检查是否作上下文切换（runrun标志）和核心是否允许作上下文切换（对核心的各种数据结构的操作都已经完成，核心处于正确的状态）。如果允许作上下文切换，则保存当前进程的上下文。
- 恢复0号进程的上下文，然后执行0号进程，寻找最高优先级的就绪进程，如果没有这样的进程存在，则执行idle过程。如果有这样的进程存在，则该进程作为当前进程分派处理机，保存0号进程的上下文。
- 恢复当前进程的上下文，执行该进程。

5.6.2 Windows 2000

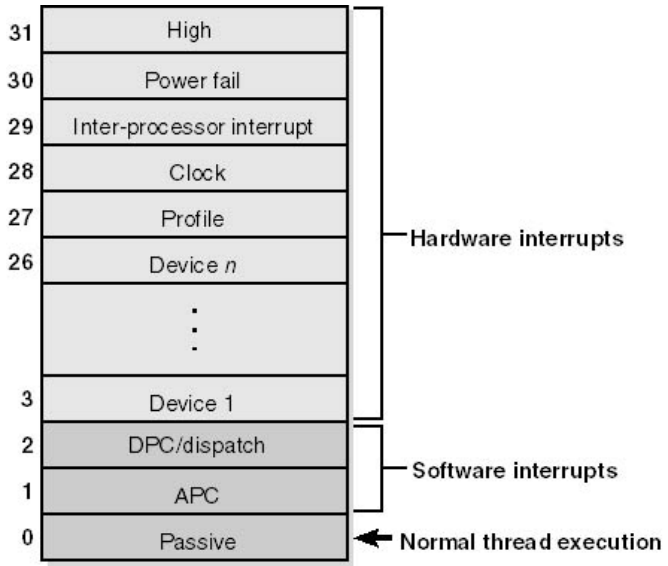
1. Windows 2000的线程调度概述

调度单位是线程而不是进程，采用严格的抢先式动态优先级调度，依据优先级和分配时间片来调度。

- 每个优先级的就绪进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，在被抢先前没有保证的运行时间；
  - 没有形式的调度循环；
  - 时间片用完事件；
  - 等待结束事件；
  - 主动挂起事件；
- 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；

2. Windows 2000的中断优先级

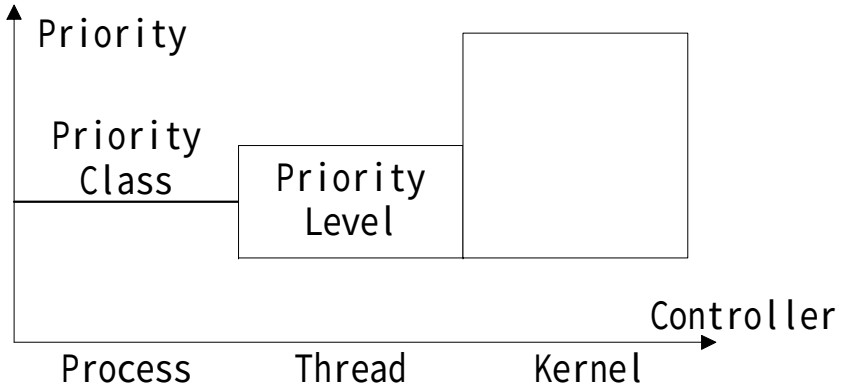


3. NT线程的优先级

从0到31，数值越大，优先级越高。分为两类

- 实时(real-time)：从16到31，如设备监控线程。
- 可变优先级(variable-priority)：从1到15（级别0保留为系统使用）。
  - 线程的基本优先级 = [进程的基本优先级 - 2，进程的基本优先级 + 2]，由应用程序控制
  - 线程的动态优先级 = [进程的基本优先级 - 2, 31]，由NT核心控制

- NT的线程优先级（Base Priority）由进程优先级类（Priority Class）和线程优先级偏移（Priority Level）构成，分别由相关函数控制。

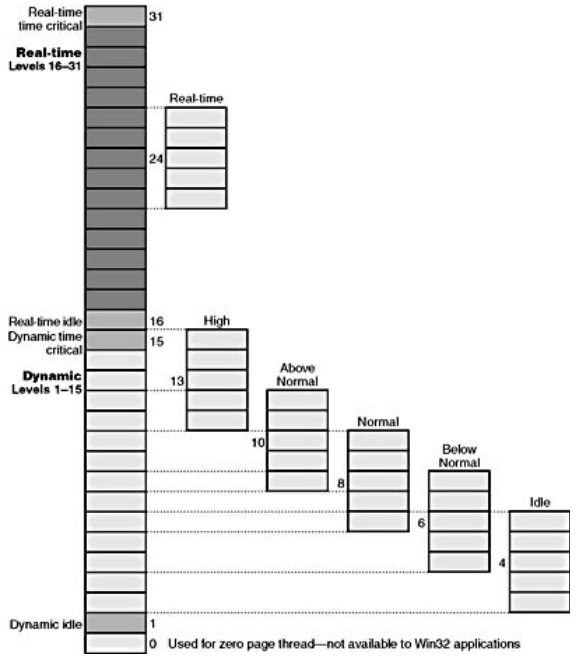


进程优先级类

Class	Normal base priority
IDLE_PRIORITY_CLASS	4
NORMAL_PRIORITY_CLASS	9 if the window of the process is in the foreground; and 7 if the window is in the background
HIGH_PRIORITY_CLASS	13
REALTIME_PRIORITY_CLASS	24

线程优先级偏移

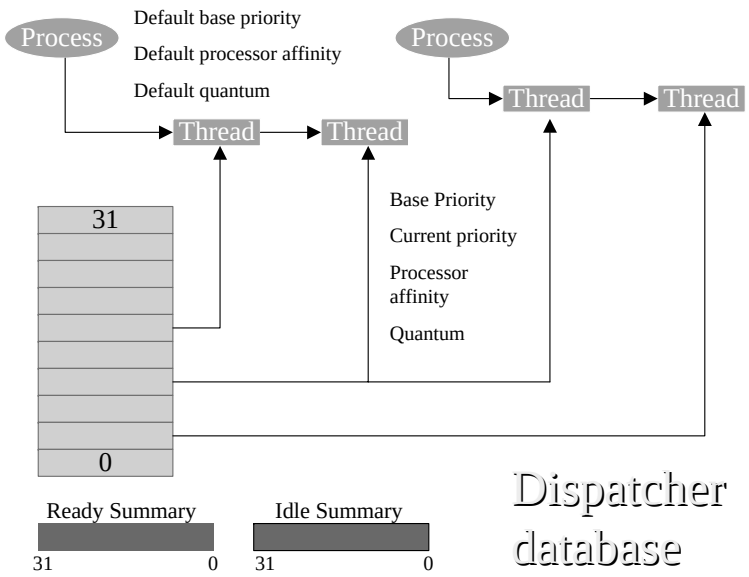
Priority	Meaning
THREAD_PRIORITY_ABOVE_NORMAL	Indicates 1 point above normal priority for the priority class.
THREAD_PRIORITY_BELOW_NORMAL	Indicates 1 point below normal priority for the priority class.
THREAD_PRIORITY_HIGHEST	Indicates 2 points above normal priority for the priority class.
THREAD_PRIORITY_LOWEST	Indicates 2 points below normal priority for the priority class.
THREAD_PRIORITY_NORMAL	Indicates normal priority for the priority class.
THREAD_PRIORITY_IDLE	Indicates a base priority level of 1 for IDLE_PRIORITY_CLASS, NORMAL_PRIORITY_CLASS, or HIGH_PRIORITY_CLASS processes, and a base priority level of 16 for REALTIME_PRIORITY_CLASS processes.
THREAD_PRIORITY_TIME_CRITICAL	Indicates a base priority level of 15 for IDLE_PRIORITY_CLASS, NORMAL_PRIORITY_CLASS, or HIGH_PRIORITY_CLASS processes, and a base priority level of 31 for REALTIME_PRIORITY_CLASS processes.



有关API

- 进程优先级类函数：
  - GetPriorityClass（读取）
  - SetPriorityClass（设置）
- 线程优先级偏移：
  - GetThreadPriority（读取）
  - SetThreadPriority（设置）

4. 线程调度数据结构



- 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位量。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(KiIdleSummary)
  - Windows 2000还维护一个称为空闲位图(KiIdleSummary)的32位量。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. Windows 2000中的start命令

- 可指定进程启动时的优先级；
  - start /high myprog

6. 线程时间配额(Quantum)

- 时间配额是一个线程从进入运行状态到Windows 2000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将重新给该线程分配一个新的时间配额，并继续运行。
- 每个线程都有一个代表本次运行最大时间长度的时间配额。时间配额不是一个时间长度值，而一个称为配额单位(quantum unit)的整数。

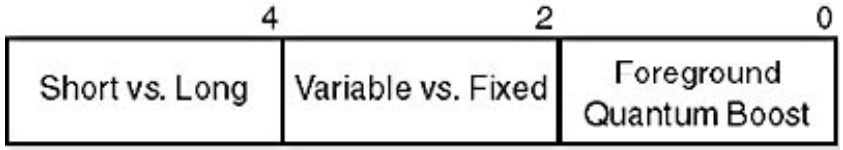
时间配额的计算

- 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。

时间配额的控制

在系统注册库中的一个注册项“HKLM\SYSTEM\CurrentControlSet\Control\PriorityControl\Win32PrioritySeparation”，允许用户指定线程时间配额的相对长度(长或短)和前台进程的进程的时间配额是否加长。该注册项为6位，分成3个字段，每个字段占2位。



- 时间配额长度字段(Short vs. Long)：
  - 1表示长时间配额，2表示短时间配额。0或3表示缺省设置(Windows 2000专业版的缺省设置为短时间配额，Windows 2000服务器版的缺省设置为长时间配额)。
- 前后台变化字段 (Variable vs. Fixed)：
  - 1表示改变前台进程时间配额，2表示前后台进程的时间配额相同。0或3表示缺省设置(Windows 2000专业版的缺省设置为改变前台进程时间配额，Windows 2000服务器版的缺省设置为前后台进程的时间配额相同)。
- 前台进程时间配额字段(Foreground Quantum Boost)：
  - 该字段的取值只能是0、1或2(取3是非法的，被视为2)。该字段是一个时间配额表索引，用于设置前后台进程的时间配额，后台进程的时间配额为第一项，前台进程的时间配额为第二项。该字段的值保存在内核变量PsPrioritySeparation。

时间配额的设置

	短时间配额			长时间配额		
改变前台进程时间配额	6	12	18	12	24	36
前后台进程的时间配额相同	18	18	18	36	36	36

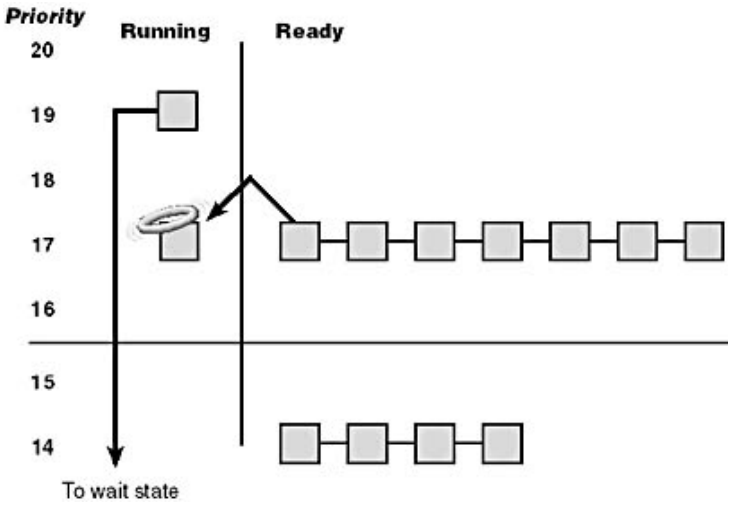
如果当前窗口切换到一个优先级高于空闲优先级类的进程中某线程，Win32子系统将用注册项Win32PrioritySeparation的前台进程时间配额字段作为索引，依据一个有3个元素的数组PspForegroundQuantum中取值，来设置该进程中所有线程的时间配额。该数组的内容由注册项Win32PrioritySeparation的另外2个字段确定。

提高前台线程优先级的潜在问题

- 假设用户首先启动了一个运行时间很长的电子表格计算程序，然后切换到一个计算密集型的应用(如一个需要复杂图形显示的游戏)。
- 如果前台的游戏进程提高它的优先级，后台的电子表格将会几乎得不到CPU时间。
- 但增加游戏进程的时间配额，则不会停止电子表格计算的执行，只是给游戏进程的CPU时间多一些。
- 如果用户希望运行一个交互式应用程序时的优先级比其他交互进程的优先级高，可利用任务管理器来修改进程的优先级类型为中上或高级，也可利用命令行在启动应用时使用命令“start /abovenormal”或“start /high”来设置进程优先级类型。

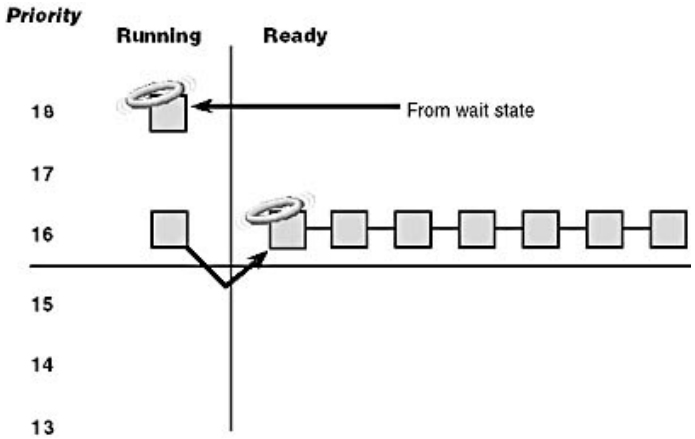
7. Windows 2000线程调度

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，线程的优先级被重置。

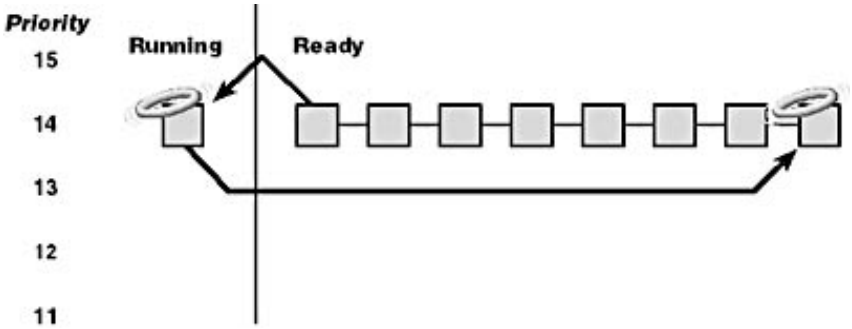
抢先



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先。

- 可能在以下两种情况下出现抢先：
  - 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先内核态下运行的线程。
  - 在判断一个线程是否被抢先时，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。
- 当线程被抢先时，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先时，时间配额被重置为一个完整的时间片；
  - 处于动态优先级的线程在被抢先时，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。

时间片用完



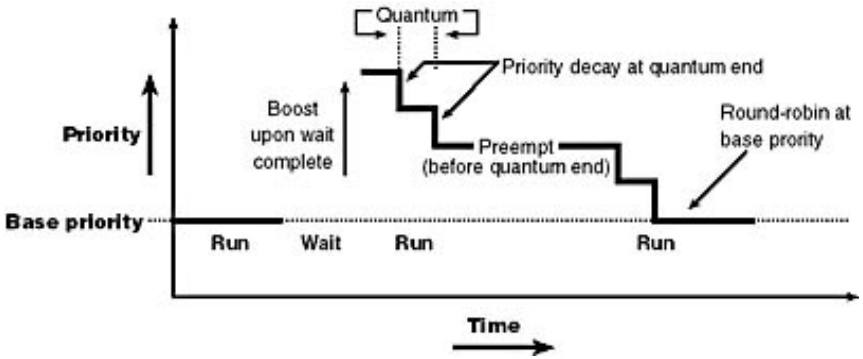
线程完整用完一个规定的时间片值时，重新赋予新时间片值，优先级降一级（不低于基本优先级），放在相应优先级就绪队列的尾部；

- 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。

结束(Termination)

- 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而从主函数中返回或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。

优先级调整



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。

- 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。

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。

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。

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。

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。

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。

8. 对称多处理机系统上Windows 2000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。

- 亲合关系Affinity
  - 描述该线程可在哪些处理机上运行。
  - 线程的亲合掩码是从进程的亲合掩码继承得到的。
  - 缺省时，所有进程(即所有线程)的
  - 亲合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函数来指定亲合掩码；

- 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；

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，判断它是否可抢先。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先，则新线程放入相应优先级的就绪队列，并等待调度执行。

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，它要在亲合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。

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，但不能立即抢先当前线程，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的线程不会抢先1号处理机上优先级为4的线程。

- 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先，系统会发出一个处理机间中断，以抢先正在运行的线程，让新线程在该处理机上运行。

小结

- 调度的类型（如调度单位的不同级别，时间周期，不同的OS），性能准则
- 进程调度：调度时机，上下文切换
- 调度算法：FCFS, SJF, RR, 多级队列，优先级，多级反馈队列，小结（队列，轮转，优先级）
- 调度算法的性能分析：周转时间和作业长短的关系
- 实时调度：概述，调度算法
- 多处理机调度：自调度，成组调度，专用处理机调度
- 算法举例：UNIX和Windows 2000

作业

实验一 处理机调度算法；