线程调度

Windows实现了一种由优先级驱动的抢占式调度系统。在这个系统中，最高优先级的可运行(就绪)线程将始终运行，某些做好运行准备的高优先级线程实际可以(或者首选)在哪些处理器上运行，这种现象也叫作处理器相关性(processor affinity)。处理器相关性是基于特定处理器组来定义的，每个处理器组最多可包含 64 颗处理器。默认情况下，线程只能在进程相关处理器组中的可用处理器上运行。(这是为了维持与老版本 Windows 的兼容性，因为老版 Windows 最多只支持 64 颗处理器。)开发者可以使用相应 API或在映像头部设置相关性掩码来调整处理器相关性设置，用户也可以通过各种工具在运行或创建进程时更改相关性。虽然一个进程中的多个线程可以关联到不同处理器组，但一个线程本身只能在所分配的处理器组内可用的处理器上运行。此外，开发者可以创建能够感知处理器组的应用程序，借此通过扩展的调度 API，把不同处理器组中的逻辑处理器关联到线程的相关性中。这样做可将进程转换为多处理器组(multigroup)进程，理论上将能在计算机中任何可用处理器上运行其线程。

当一个线程被选中准备运行后，它将可以会一个量程(quantum)。量程是指在轮到其他具备相同优先级的线程运行前，线程可以运行的时间度量单位。对于不同系统或不同进程，量程的值可能各不相同，主要原因为下列3点之一

1. 系统配置设置(量程的长短、可变或固定的量程、优先级分离)。
2. 进程的前台或后台状态。
3. 使用作业对象更改量程

由于 Windows 实现了一种抢占式调度器，线程可能无法完整用完自己的量程。也就是说，如果另一个优先级更高的线程已经就绪可以运行，当前运行的线程很可能会在用完自己的时间片 (time slice)之前就被抢占。实际上，对于被选择接下来运行的线程，很可能在它的量程真正开始之前就被抢占。

Windows调度代码是在内核中实现的。然而并不存在单独的“调度器”模块或例程调度代码分散在整个内核中与调度相关的事件各处。

执行这些工作的例程总称为内核的调度程序 (dispatcher)。下列事件可能需要线程调度程序的参与。

1. 线程变为执行就绪状态，例如已经新建完成的线程，或刚从等待状态释放的线程
2. 量程到期、线程终止、线程放弃执行或进入等待状态，导致线程离开运行状态
3. 线程优先级发生变化，这可能源自系统服务调用，或 Windows 自己更改了优先级值
4. 线程的处理器相关性发生变化，以至于无法在当前处理器上继续运行。

在上述每个交叉点时，Windows 必须决定(如果适用的话)在原本运行该线程的逻辑处理器上接下来要运行哪个线程，或者决定接下来用哪颗逻辑处理器运行该线程。逻辑处理器在选择运行一个新线程后，还需要通过上下文切换来切换到新线程。在上下文切换过程中，会保存当前运行中线程所关联的易失性(volatile)处理器状态信息，加载另一个线程的易失性状态信息，然后开始执行新线程。

优先级级别

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 优先级类相对优先级 | 实时 | 高 | 高于正常 | 正常 | 低于正常 | 空闲 |
| 时间关键(+饱和) | 31 | 15 | 15 | 15 | 15 | 15 |
| 最高(+2) | 26 | 15 | 12 | 10 | 8 | 6 |
| 高于正常(+1) | 25 | 14 | 11 | 9 | 7 | 5 |
| 正常(0) | 24 | 13 | 10 | 8 | 6 | 4 |
| 低于正常(-1) | 23 | 12 | 9 | 7 | 5 | 3 |
| 最低(-2) | 22 | 11 | 8 | 6 | 4 | 2 |
| 空闲(-饱和) | 16 | 1 | 1 | 1 | 1 | 1 |

表为Windows内核优先级别到Windows API的映射关系

在内核中，进程优先级类会使用PspPriorityTable全局数组以及PROCESS\_PRIORITY\_CLASS 索引转换为基本优先级

无论进程优先级类是什么，时间关键和空闲这两种相对线程优先级的值会始终不变(除非是实时优先级) 。因为Windows API会向内核请求优先级饱和值，为此会传入+16或-16作为所请求的相对优先级。用于计算该值的公式如下(HIGH\_PRIORITY等于31)

随后内核会将这些值视作对饱和值的请求，并在 KTHREAD 中设置 Saturation 字段对于正数饱和值，这会导致线程在自己的优先级类(动态或实时)的基础上获得可能的最高优先级值；对于负数饱和值，则会获得可能的最低优先级值。此外如果随后通过请求更改了进程的基本优先级，将不会影响到这些线程的基本优先级，因为进程代码中的饱和线程会被跳过。

从 WindowsAPI的角度来看，线程可以设置7种级别的优先级 (“高”优先级类可设置6个类别)。“实时”优先级类实际上可以设置介于 16 和 31 之间的任何优先级类别。表中未涉及的标准常数值可通过-7、-6、-5、-4、-3、3、4、5和6这几个值，将其作为 SetThreadPriority 的参数来指定。无论通过Windows API(将进程优先级类与相对线程优先级相结合)设置了怎样的线程优先级，对调度器来说，只需要关注最终结果即可。例如，优先级级别 10 可通过两种方式获得:“正常”优先级类进程(8) 与线程相对优先级“最高”(+2) 相结合，或“高于正常”优先级类进程(10) 与线程相对优先级“正常”(0)相结合。从调度器的角度来看，这些设置可以获得相同的优先级值(10)，因此这些线程在优先级方面是完全平等的。

虽然一个进程只能有一个基本优先级值，但其中的每个线程可以有两个优先级值:当前值(动态)和基本值。调度决策将基于当前优先级来进行。

线程的初始基本优先级是从进程基本优先级继承而来的。默认情况下，进程会从创建自己的进程处继承基本优先级。这种行为可通过 CreateProcess 函数改变，或也可以使用命令中的Start 命令来改变。我们还可以在进程创建完成后，使用SetPriorityClass 函数或各种提该功能的工具(如任务管理器或 Process Explorer)来更改进程优先级。(右击进程，选择新优先级类。例如，可以降低 CPU 密集型进程的优先级，使其不至于影响到常规系统操作更改进程的优先级可以增大或降低线程优先级，但线程的相对设置始终保持不变。

通常来说，用户应用程序和服务会以“正常”基本优先级启动，因此其中的初始线程通常会在优先级级别8下执行。然而有些 Windows 系统进程(如会话管理器、服务控制管理器、本地安全身份验证进程)的基本进程优先级略高于默认的“正常”类(8)。这个较高的默认值确保了这些进程中的线程启动时会获得比 8 这个默认值更高的优先级。

实时优先级

对于任何应用程序，我们都可以在一个动态范围内提高或降低线程优先级。然而必须具备提高调度优先级特权(SeIncreaseBasePriorityPrivilege)才能在“实时”范围内调整注意，很多重要的 Windows 内核模式系统线程都运行在“实时”优先级范围内，因此如果有线程在该范围内运行了很长的时间，可能会阻碍到重要的系统功能(例如内存管理器缓存管理器，或某些设备驱动程序)。

在使用标准Windows API的情况下，一旦某个进程进入实时范围，它的所有线程(哪怕空闲线程)也必须运行在这个实时优先级级别下。因此我们不可能通过标准接口将实时和动态线程混合包含在同一个进程中。这是因为SetThreadPriority API需要使用ThreadBasePriority信息类调用原生NtSetInformationThread API，而这个信息类只允许在同一个范围内设置优先级。此外，除非请求来自 CSRSS 或另一个实时进程，否则这个信息类只允许在公认的Windows API差值-2到2(或时间关键和空闲)之间更改优先级。换句话说，这意味着实时进程可以选择介于 16 到31 之间的任何线程优先级，而标准Windows API相对线程优先级的选择会受到一定限制。

使用这种特殊值调用 SetThreadPriority 会导致通过 ThreadActualBasePriority信息类调用 NtSetInformationThread，而该线程的内核基本优先级可直接设置，并可包含实时进程的动态范围。

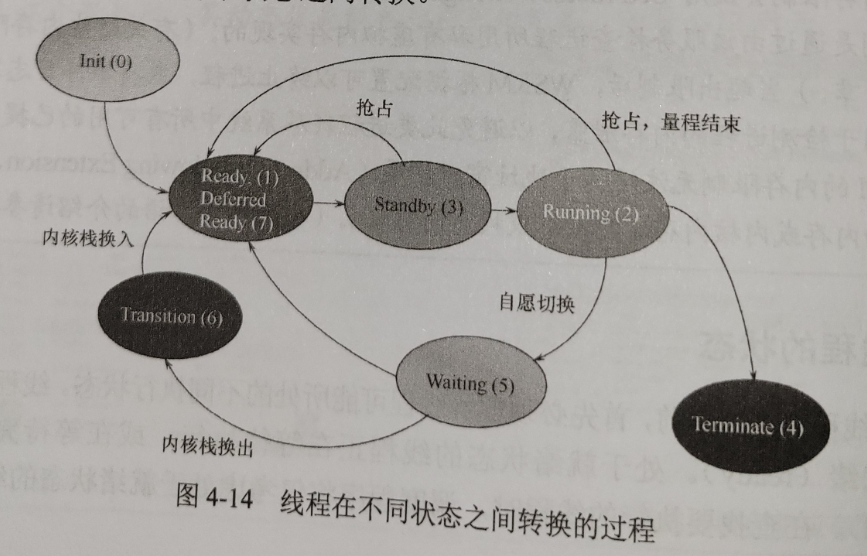
注意 “实时”这个词并不意味着 Windows 是一种常规意义上的实时操作系统。这是因为Windows 并未提供真正的实时操作系统基础构造，例如有保障的中断延迟，或让线程获得有保障的执行时间的方式。此处的“实时”仅仅意味着“高于其他一切”。

线程的状态

1. **就绪 (ready)。**处于就绪状态的线程正在等待执行，或在等待完成后被换入(in-swapped)。在查找要执行的线程时，调度程序将仅考虑处于就绪状态的线程。
2. **延迟就绪(deferred ready)**。该状态代表线程已被选择在特定处理器上运行尚未开始运行。该状态的存在，使得内核能够保持调度数据库被每颗处理器锁锁定的时间长度维持在最小范围内。
3. **待命 (standby)**。处于该状态的线程已被选中，随后在特定处理器上运行。在满足正确条件后，调度程序将针对该线程执行上下文切换。系统中每颗处理器只能有一个处于待命状态的线程。线程真正开始执行前，可能会在处于待命模式时就被抢占掉(例如在待命线程开始执行前，有更高优先级的线程变得可以运行了)。
4. **运行中(running)**。调度程序针对线程执行上下文切换后，线程将进入运行中状态并开始执行。线程的执行将持续进行，直到量程到期(随后将执行相同优先级并且已经就绪的其他线程)、被更高优先级的线程抢占、该线程终止、该线程放弃执行，或该线程自愿进入等待中状态为止。
5. **等待中(waiting)**。线程可通过多种方式进入等待中状态，例如线程自愿等待个对象以便与其同步执行，操作系统代表线程进行等待 (例如为了处理分页 IO)，或某个环境子系统可以让线程自行挂起。当线程的等待结束后，根据其优先级，线程可能立即开始运行，或重新恢复为就绪状态。
6. **转换 (transition)**。如果线程已执行就绪，但其内核栈从内存中被换出，此时线程将处于转换状态。在内核栈重新换入内存后，线程将进入就绪状态。

在Windows操作系统中，每个线程都有一个与之关联的内核栈。线程内核栈是一段连续的内存区域，用于在内核模式下存储线程的函数调用、局部变量和其他临时数据。线程内核栈对于操作系统的正常运行至关重要，因为它提供了在内核模式下执行代码所需的栈空间。以下是一些关于Windows线程内核栈的关键点：

1. 内核栈大小：在32位Windows系统中，线程内核栈的默认大小为12KB。在64位Windows系统中，线程内核栈的默认大小为24KB。这些大小可以在创建线程时进行调整，但不建议使用非默认大小，除非有充分的理由。
2. 内核栈和用户栈：每个线程都有两个栈：一个内核栈和一个用户栈。内核栈用于在内核模式下运行，而用户栈用于在用户模式下运行。当线程从用户模式切换到内核模式时，例如在执行系统调用时，操作系统会自动将栈指针从用户栈切换到内核栈。
3. 栈溢出检测：为了检测和防止内核栈溢出，Windows操作系统在内核栈的末端保留了一个“警戒区域”。当线程试图访问这个警戒区域时，操作系统会立即触发一个栈溢出异常。这种机制有助于保护系统免受潜在的栈溢出攻击。
4. 内核栈的分配和释放：线程内核栈在线程创建时分配，并在线程终止时释放。操作系统负责管理内核栈的生命周期，确保每个线程都有一个独立的内核栈。
5. 内核栈中的数据：线程内核栈包含许多重要的数据，包括线程的上下文信息、函数调用帧和局部变量。当线程在内核模式下运行时，这些数据对于正确执行代码和恢复线程的执行状态至关重要。
6. **已终止(terminated)**。执行完毕的线程将进入该状态。线程被终止后，执行体线程对象(用于在系统内存中描述该线程的数据结构)可能会被释放，或者也可能不释放对象管理器可通过策略决定何时删除该对象。例如，如果还有指向该线程的未关闭句柄对象将继续保留。如果被其他线程明确终止(例如调用 TerminateThread 这个 Windows API)，线程也可以从其他状态直接进入已终止状态。
7. **已初始化 (initialized)**。当线程创建完成后，内部将使用该状态。



调度程序数据库

为了做出线程调度决策，内核维护了一组总称为调度程序数据库 (dispatcher database)的数据结构。调度程序数据库会跟踪哪些线程正等待执行，以及哪颗处理器正在执行哪线程。

为了提高可缩放性以及线程调度的并发性，Windows 多处理器系统为每颗处理器供了一个调度程序就绪队列，以及一个共享的处理器组队列。借此，每颗CPU即可检查共享给自己的就绪队列，进而了解接下来需要执行的线程，而无须锁定整个系统范围的就绪队列。

注意 这种共享的数据结构必须受到保护[使用旋转锁 (spinlock)]，因此处理器组不能太大。通过这种方式，队列的争用就不那么重要了。在当前的实现中，处理器组最多可包含4颗逻辑处理器。如果逻辑处理器数量超过 4 颗，那么将创建多个处理器组，并将所有可用处理器平分给所有处理器组。例如，在包含6颗处理器的系统中，将创建两个处理器组，每个组包含3颗处理器。

就绪队列、就绪摘要以及其他相关信息都存储在PRCB 中一个名为KSHARED\_READY\_QUEUE 的内核结构中。虽然每颗处理器都具备该数据结构，但每颗处理器中仅第一颗处理器会使用该数据结构，并将其与同一个组中的其他处理器共享。

调度程序就绪队列(KSHARED\_READY\_QUEUE 中的 ReadListHead)包含处于就状态，等待调度执行的线程信息。Windows维护了一种名为就绪摘要(ReadySummary)的 32位掩码。该掩码中设置的每个位均代表该优先级级别下，就绪队列中的一个或多个线程(位0代表优先级0，位1代表优先级 1，以此类推)。

系统会以原生处理器命令的方式执行单个位的扫描，借此找到所设置的最高位。这样无论就绪队列中包含多少个线程，都可以用固定时长完成操作。

调度程序数据库可通过将 IRQL 提升至 DISPATCH\_LEVEL(2)进行同步，以防止其他线程打断处理器的线程调度过程。

量程

量程(quantum)是指在 Windows查是否有相同优先的其他线程等待运行前，一个线程可运行的时间长度。如果一个线程的量程已结束，并且该优先级下没有其他线程，Windows 将允许该线程继续运行一个量程。

在客户端版本 Windows 中，默认情况下线程可运行两个时钟间隔。在服务器系统中默认情况下线程可运行 12个时钟间隔。服务器系统默认使用更大的默认值，这是为了尽可能减少上下文切换次数。通过使用更长的量程，服务器应用程序一旦在收到客户端请求被唤醒后，会有更大可能在量程时段内完成请求并重新进入等待状态。

不同硬件平台的时钟间隔长度有所不同。时钟中断的频率取决于HAL而非内核。时钟间隔信息以纳秒为单位存储在内核变量KeMaximumIncrementas中。

线程的运行时间统计是基于处理器周期进行的系统启动时，会将以赫兹 (Hz)为单位的处理器速度(每秒的CPU 周期数)乘以单个时钟周期(clock cycle)(一个时间周期表示的时长为CPU执行一条指令需要的时间)的秒数，借此计算出每个量程的等效时钟周期数。该值会存储在内核变量 KiCyclesPerClockQuantum 中。

这种记账方式导致线程的量程长短并不取决于时钟周期数，而是要运行一个量程目标(target)，这种估计值代表了运行过程中，线程可以使用的 CPU 周期数。

每个进程都在进程控制块 (KPROCESS)中有一个量程重置值。在进程中新建线程时，会使用该值并将其复制到线程控制块(KTHREAD)中，并用它为新线程设置新的量程目标。

线程运行过程中，会针对诸如上下文切换、中断、某些调度决策等不同事件统计所用的CPU时钟周期。如果在某个时钟间隔计时器中断发生时，所统计的CPU时钟周期数已经达到(或超过)量程目标，将触发量程结束处理。

从内部来看，一个量程单元等于一个时钟计时周期 (clock tick)的三分之一。也就是说，一个时钟计时周期等于 3 个量程。这意味着在客户端 Windows 系统中，线程的量程因此对于前文提到的重置值为 6(2×3)，而服务器系统的量程重置值为 36(12×3)

Clock tick 是指系统时钟中断的发生。在操作系统中，系统时钟是由硬件时钟发出的周期性中断信号，一般是每秒发出一定数量的中断信号。每当发生一个系统时钟中断，CPU 就会停止当前正在执行的程序，转而去执行中断处理程序。中断处理程序负责更新系统时间、响应输入输出等任务。

当时钟间隔较短时，系统时钟中断发生的次数就会更频繁，中断处理程序的执行时间就需要更短，以避免中断处理程序过长而导致系统的延迟和卡顿。因此，时钟间隔越短，中断处理程序的执行时间就需要越短，才能保证系统的性能和稳定性。

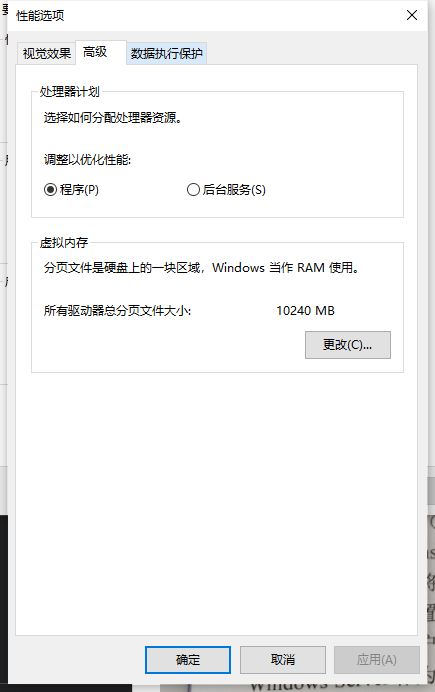
需要注意的是，时钟间隔不能过短，否则会导致系统的负载和开销增加，从而影响 CPU 的性能。此外，过于频繁的系统时钟中断也会影响系统的稳定性，因为中断处理程序可能会与正在执行的进程、线程争夺 CPU 资源，造成系统的延迟和卡顿。因此，时钟间隔的长度需要根据具体的应用场景和系统需求来确定。

量程的控制

所有进程的线程量程都可更改，但只能选择如下两种设置之一；短设置(2个时钟计时周期，客户端系统的默认值)；长设置(12 个时钟计时周期，服务器系统的默认值)

注意 如果在长设置量程系统中使用作业对象，还可以为作业中的进程选择其他量程值

若要更改该设置，可以到Windows系统的高级系统设置中的“高级”选项卡下的“性能”栏下的“高级”选项卡，随后便会看到下图的对话框



该对话框包含两个关键选项。

1. Programs(程序)。该设置将使用短的可变量程，是客户端版本Windows的默认值。如果在服务器系统上安装终端服务并将该服务器配置为应用程序服务器，那么也将使用该设置，这样使用终端服务器的用户就可以获得与桌面或客户端系统相同的量程设置。如果使用Windows Server 作为桌面操作系统，也可以手动选择该选项。
2. Background services(后台服务)。该设置将使用长的固定量程，是服务器系统的默认值。唯有在将工作站充当服务器系统的时候,才有必要对工作站系统选择该选项。不过由于该选项的改动会立刻生效因此也可以在计算机需要运行后端或服务器类的工作负载时再选择该选项。例如，如果需要长时间运行计算、编码或建模模拟等任务，甚至可能需要彻夜运行时，即可在夜间选择Background services 选项，并在白天重新设置为Programs 模式。

可变量程

如果启用可变量程，此时会将可变量程表 (PspVariableQuantums)中存储的，由6个量程成员组成的数组载入由PspComputeQuantum函数使用的PspForegroundQuantum表中。其算法会根据进程是否为前台进程(也就是说，是否包含拥有桌面前台窗口的线程)来选择最合适的量程索引。如果不是这种情况，将选择索引0，该索引值对应了前文介绍的默认线程量程。对于前台进程，该量程索引将等同于优先级分离值。

该优先级分离值决定了调度器应用给前台线程的优先级提升因此可与量程的相应扩展实现配对。每额外增加一个优先级级别(最多2级)，线程即可额外获得一个量程。举例来说，如果线程获得了一个优先级级别的提升，那么也将额外获得一个量程。默认情况下，Windows 会对前台线程设置可行的最大优先级提升，这意味着其优先级分离值为 2，因而会选择可变量程表中的量程索引2。结合起来，这会导致线程额外获得两个量程，共有3个可用量程。

下表描述了根据量程索引和不同量程配置，可以额外获得的量程值

这些数值均以一个时钟计时周期的1/3 为单位

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 短量程索引 | | | 长量程索引 | | |
| 可变 | 6 | 12 | 18 | 12 | 24 | 36 |
| 固定 | 18 | 18 | 18 | 36 | 36 | 36 |

因此在客户端系统中将一个窗口切换到前台后，包含了该前台窗口所属线程的进程中的有线程将获得3倍的量程。

与量程设置有关的注册表值

在图形界面下修改量程设置的选项实际上修改了注册表 HKLM\SYSTEM\CurrentControlSet\Control\PriorityControl键下的 Win32PrioritySeparation 值。除了指定线量程的相对长度(长或短)，该注册表值还决定了是否使用可变量程和优先级分离。该值包含 6个位，分为3个2位字段。

0-2位为优先级分离，3-4位为可变或固定，5-6位为长或短

1. **长或短**。该值设置为“1”代表长量程，“2”代表短量程，“0”或“3”代表采用适合当前系统的默认值(客户端系统为短，服务器系统为长).
2. **可变或固定**。该值设置为“1”代表启用可变量程表，设置为“0”或“3”代表采用适合当前系统的默认值(客户端系统为可变服务器系统为固定)。
3. **优先级分离**。该字段(存储在内核变量PsPrioritySeparation中)决定了优先级分离(最多可设置为“2”)

对于处于“空闲”进程优先级类的进程，其中运行的线程将始终收到一个线程量程并会忽略其他任何(由默认值或通过注册表设置的)量程配置设置。

优先级提升

Windows调度器会借助内部优先级提升(priority boost)机制周期性地调整进程的当前(动态)优先级。很多情况下，这是为了减少各种延迟(即为了让线程更快速响应正在等待的事件)并提高响应能力。其他情况下，这样做则可能是为了避免倒置 (inversion)和饥饿(starvation)等情况。

1. 由于调度器/调度程序事件而进行提升 (减少延迟)。
2. 由于IO操作完成而提升(减少延迟)。
3. 由于用户界面(UI)输入而提升(减少延迟/提高响应性)
4. 由于线程长时间等待执行体资源(ERESOURCE)而提升(避免饥饿)。
5. 运行就绪的线程一段时间内没有运行而进行提升(避免饥饿和优先级倒置).

然而与任何调度算法类似，这些措施并不完美，可能无法让所有应用程序从中获益

Inversion（倒置）

当一个低优先级的线程持有了一个共享资源，而高优先级的线程需要访问该共享资源时，由于低优先级线程一直占用该资源，高优先级的线程会被迫等待，从而导致低优先级线程“倒置”了高优先级线程的执行顺序。

Starvation（饥饿）

当一个线程无法获得所需的资源或 CPU 时间，导致它一直无法执行或一直处于等待状态，从而导致该线程“饥饿”。

客户端版本的 Windows 还包含一种适用于多媒体播放过程的伪提升机制。与其他优先级提升机制不同，多媒体播放提升是由名为 Multimedia Class Scheduler Service(mmcss.sys)的内核模式驱动程序管理的。然而这并不是真正的优先级提升。该驱动程序仅仅只是在需要时为线程设置新的优先级，因此与优先级提升有关的规则对它均不适用接下来我们首先介绍典型的、由内核管理的优先级提升。

1. **由于调度器/调度程序事件而进行提升**

发生调度程序事件时，将调用 KiExitDispatcher 例程。该例程的任务是调用 KiProcessThreadWaitList处理延迟就绪列表，随后调用 KzCheckForThreadDispatch检查当前处理器上是否有任何线程不应被调度。当此类事件发生时，调用方还可以指定要对线程进行哪种类型的优先级提升，以及与提升相关联的优先级增量。下列场景会被视作 AdjustUnwait调度程序事件，因为它们需要处理正在进入信号 (signaled) 状态的调度程序(同步)对象，这可能导致唤醒一个或多个线程。

1. 异步过程调用 (APC)被加入线程队列。
2. 事件被设置或触发 (pulsed)。
3. 计时器被设置，或系统时间有变化并且计时器必须重置
4. 互斥体(mutex)被释放或放弃。
5. 进程退出。
6. 项被插入队列 (KQUEUE)，或队列被清空
7. 信号量 (semaphore) 被释放。
8. 线程被警告、挂起、恢复、冻结或解冻
9. UMS 主线程正等待切换至已调度的UMS 线程

对于关联至公共 API的调度事件(如 SetEvent)，所用的优先级提升增量是由调用方决定的。对于警告，所用提升增量为 2(除非调用 KeAlertThreadByThreadId 将线程置于警告等待状态，此时所用提升增量为1) ，因为警告 API并没有提供调用方可用于设置自定义增量的参数。

调度器还有两个特殊的AdiustBoost 调度程序事件，它们是锁拥有权(lock-ownership)优先级机制的一部分。这些提升会试图解决倒置的情况，通过让释放锁的线程继续执行，尽管实际上它应该让新的所有者线程被唤醒并接管处理器的控制权。下列两种调度程序事件会导致 AdjustBoost调度程序退出。

1. 通过 KeSetEventBoostPriority 接口设置一个事件，并被ERESOURCE读取者-写入者内核锁使用。
2. 通过 KeSignalGate 接口设置一个门(gate)，并在释放入口锁后被各种内部机制使用。
3. **等待后提升**

等待后(unwait)提升会试图降低如下两个环节之间的延迟: 线程因为对象发送了信号而被唤醒(进而进入就绪状态)，以及线程实际开始执行等待后需要处理的任务(因而进入运行中状态)。一般来说，我们都希望从等待状态唤醒的线程能够尽可能快速地开始运行。

各种Windows头文件指定了内核模式API，如KeReleaseMutex、KeSetEvent和KeReleaseSemaphore 等的调用方应当使用的推荐值，这些值也符合诸如MUTANT\_INCREMENT、SEMAPHORE\_INCREMENT 和EVENT\_INCREMENT等的定义。这3个定义在头文件中总是会被设置为1，因此可以安全地假设，针对此类对象的大部分Unwait 会导致增量为1的提升。对于用户模式API，不仅无法指定增量，也无法让诸如NtSetEvent等原生系统调用通过参数指定增量。实际上，当此类 API 调用底层 Ke 接口时，会自动使用默认的INCREMENT 定义。当互斥体被放弃或计时器因为系统时间变化而被重置时，也是类似的情况，即系统将使用默认提升，类似于通常在互斥体被释放后所应用的提升。最后，APC提升则完全取决于调用方。稍后我们将在下文介绍与I/O 完成有关的APC 提升使用场景。

注意 一些调度程序对象没有相关联的提升，例如，当计时器被设置或过期时，或当进程发送了信号时，将不会应用提升。

所有这些增量为1的提升，在试图解决最初问题时都会假设释放和等待的线程均运行在相同优先级下。通过将等待的线程优先级提升一个级别，等待的线程将在操作完成后立即抢占释放的线程。然而在单处理器系统中，如果该假设不成立，提升将起不到太大的效果。举例来说，如果等待的线程为优先级 4，释放的线程为优先级8，那么在优先级5上的等待并不能在减少延迟和强制抢占方面起到多大作用。不过在多处理器系统中，由于窃取(stealing)和平衡算法的存在，更高优先级的线程将有更大可能被另一颗逻辑处理器选中。这种情况要归结于 NT 架构最初的一个设计选择，即不跟踪锁的所有权(但少数锁除外)。这意味着调度器无法确定事件到底归谁所有，以及它是否真的被当作锁来使用。即使对锁的所有权进行追踪，除非与执行体资源有关，否则为了避免护航问题(convoy issue)，通常也不会传递所有权。

某些类型的锁对象会使用事件或门作为自己的底层同步对象，而锁所有权提升解决了这种两难的问题。此外在多处理器系统中，就绪线程可能会被其他处理器选中，而此时高优先级可能会增加它在其他外理器上运行的概率。

1. **锁所有权提升**

由于执行体资源(ERESOURCE)和关键节的锁使用了底层的调度程序对象，释放这些锁会导致前文介绍过的等待后提升。另外，因为这些对象的高层实现会跟踪锁的所有者，所以在决定要应用的提升时，内核通过使用AdjustBoost 将可以做出更适合的决策。在此类提升中，AdjustIncrement 会被设置为: 所释放(或设置)的线程的当前优先级，减去任何图形用户界面(GUI)前台分离提升。此外在调用KiExitDispatcher 函数前，事件和门代码会先调用KiRemoveBoostThread 以便让释放的线程返回正常优先级。为避免锁护航(lock-convoy)问题，即两个线程相互之间反复传递锁进而得到不断增长的提升，这一步必不可少。

注意 推锁(pushlock)是一种很不公平的锁，因为在争夺获取路径上，此类锁的所有权是不可预测的 (与自旋锁相似，是随机的 )，优先级不会因为锁的拥有权而获得提升。这样做只能加重抢占和优先级扩散的情况，而这些情况并非必须，因为锁会在释放后立即可用(绕过了常规的等待/等待后路径 )。

1. **I/O操作完成后的优先级提升**

当某些IO 操作完成时，Windows 会提供临时的优先级提升，借此等待这些 IO 的线程将有更大机会可以立即运行以便处理正在等待的任务。虽然 Windows 驱动程序开发包WDK)头文件提供了推荐的提升值 (请在 Wdm.h或Ntddk.h 中搜索#define IO\_)，但实际使用的提升值取决于设备驱动程序。设备驱动程序完成IO请求后，会在对内核函数IoCompleteRequest 的调用中指定提升值。

|  |  |
| --- | --- |
| 设备 | 提升值 |
| 磁盘、CD-ROM、并口、视频（显卡） | 1 |
| 网络、邮件槽、命名管道、串口 | 2 |
| 键盘、鼠标 | 6 |
| 声卡 | 8 |

注意 很多人可能凭直觉认为显卡或磁盘需要比提升值 1更高的响应性。然而内核实际上是在尽可能对延迟进行优化，因为某些设备(以及人类的感官输入)比另一些设备更敏感。为了形象地理解，可以这样看: 为了在播放音乐的过程中不让人感觉到“卡顿”，声卡需要每 1ms均能获得数据;而显卡每秒钟只需要输出 24 帧画面，也就是说每画面约持续 42ms，超过这个时间才会产生可察觉的跳帧

IoCompleteRequest 这个 API 中信号发送代码的内核实现方式为: 使用APC(针对异步 IO)，或使用事件(针对同步I/O)。举例来说，当驱动程序将异步磁盘读取的IODISK INCREMENT传入IoCompleteRequest 时，内核会调用KeInsertQueueApc，并将提升参数设置为 IODISK INCREMENT。随后当该线程的等待因为APC而被打断后，即可得到值为1的提升。

表中的提升值仅仅是微软的建议，实际编写驱动时可以忽略

当任何文件系统驱动程序(可通过将自己的设备类型设置为FILE DEVICE DISK FILESYSTEM或FILE DEVICE NETWORK FILE SYSTEM加以识别)完成了自己的请求后，如果驱动程序传入了IO\_NO\_INCREMENT(0)那么将始终对IO\_DISK\_INCREMENT(1)进行提升。

1. **等待执行体资源时的提升**

当线程试图获取执行体资源，而该资源已经被另一个线程独占拥有时，该线程必须进入等待状态，直到其他线程释放了该资源。为降低死锁风险，执行体会以 500ms 为间隔进行等待，而不会无限制地等待该资源。每当500ms的等待结束后，如果该资源依然被独占，为了防止CPU饥饿，执行体会尝试获取调度程序锁，将独占该资源的一个或多个线程的优先级提升至 15 (如果原所有者线程的优先级低于等待者线程，并且小于 15) ，重置其量程，然后开始下一次等待。

由于执行体资源可被共享也可被独占，内核首先会提升独占的所有者，随后检查共享的所有者并提升所有这些所有者。当等待的线程再次进入等待状态后，调度器有可能调度一个或多个所有者线程，这些线程将有足够的时间完成自己的工作并释放资源。但是要注意，只有在资源未设置 Disable Boost(禁止提升)标志时才会使用这种提升机制。

此外，这种机制也并不完美。举例来说，如果资源有多个共享的所有者，执行体会将所有这些线程的优先级提升至 15。这会导致系统中突然激增大量高优先级线程，并且所有线程都具备完整量程。虽然最初的所有者线程会优先运行(因为它是被第一个提升的,所以会处于就绪列表首位)，但其他共享的所有者会随后运行，因为等待线程的优先级并未提升。只有在所有共享的所有者都有机会运行，并且它们的优先级被降低到低于等待线程后，等待线程才能最终有机会获取资源。由于在原本的独占所有者释放资源后，共享的所有者可将自己的所有权从共享所有提升或转换为独占所有,因此该机制可能无法取得预期效果。

1. **前台线程等待之后的提升**

当前台进程中的线程针对内核对象的等待操作完成后内核会将自己的当前(而非基本)优先级提升为 PsPrioritySeparation（注册表中的Win32PrioritySeparation键） 的当前值。(窗口系统负责决定哪些进程是前台进程。)

1. **GUI线程被唤醒后的优先级提升**

拥有窗口的线程，在被某些窗口活动(例如收到窗口消息)唤醒后将获得额外的提升值2。当窗口系统(Win32k.sys)调用 KeSetEvent 以设置用于唤醒GUI线程的事件时便会进行这样的提升。

1. **CPU饥饿导致的优先级提升**

优先级反转 (priority inversion)理想的解决方案(至少在理论上是理想的)应该是跟踪锁和它的所有者，并酌情对需要的线程进行提升以便进一步处理。这种理念是通过一种名为自动提升 (auto boost)的功能实现的。

Windows还提供了一种通用的CPU饥饿缓解机制，该机制包含在平衡集管理器 (balance-set manager)线程中。(这是一个系统线程，主要目的在于执行内存管理功能) 该线程会以每秒一次的频率扫描就绪队列，在其中查找已经处于就绪状态(也就是说，尚未运行)并持续了约4s 的线程。如果找到这样的线程，平衡集管理器会将该线程的优先级提升至 15，并将量程目标设置为与3个量程单元相等的 CPU时钟周期数。该量程到期后，线程的优先级会立即降低至最初的基本优先级。如果该线程还没有运行完毕并且更高优先级的线程已经准备好运行，优先级降低后的进程会返回就绪队列；如果继续维持这种状态 4s，则再次经历上述提升过程。

平衡集管理器并不会在每次运行时扫描所有已就绪线程。为了将自己所用的CPU时间降至最低，它只扫描 16 个就绪线程，如果在该优先级级别上还有更多线程，它会记住自己上次扫描到的位置，并在下次扫描时从这个位置继续进行。此外，每次运行只能提升10个线程。如果找到了超过 10个线程适合进行此类提升(也意味着整个系统真的是非常繁忙)，下次运行时会停止扫描并直接提升。

注意 如前所述，Windows 的调度决策不受线程数量影响，并且每次所需的时间都是固定不变的。由于平衡集管理器必须手动扫描就绪队列，该操作取决于系统中的线程数量，线程越多扫描所需的时间就越长。然而平衡集管理器及其算法并不算调度器的一部分，只是一种为了提高可靠性而采用的扩展机制。此外，因为要扫描的线程和队列存在上限，所以可保证平衡集管理器对性能的影响维持最低，并且在最糟糕的情况下也依然是可预测的

1. **应用提升**

继续回到KiExitDispatcher，我们已经介绍过，可以调用 KiProcessThreadWaitList来处理延迟就绪列表中的线程。而调用方传递的提升信息也是在这里处理的。为此需要循环遍历每个 DeferredReady 线程，解除到等待块的链接(仅 Active 和 Bypassed 块会被解除链接)，并在内核的线程控制块中设置两个重要值: AdjustReason 和AdjustIncrement。具体原因可能是前文提到的两个 Adjust 可能性之一，而增量对应着提升值。随后将调用KiDeferredReadyThread，并通过运行两个算法让该线程做好运行准备，这两个算法为: 量程和优先级选择算法(将分为两部分进行介绍)，以及处理器选择算法

首先看看该算法何时应用提升，要注意，该提升仅在线程未处于实时优先级范围内时才会生效。AdjustUnwait 提升的应用存在这样的前提要求: 线程并未处于任何非正常提升过程且该线程未通过调用 SetThreadPriorityBoost 在 KTHREAD 中设置 DisableBoost 标志以禁用提升。此外还有一种情况会禁用提升: 内核意识到线程已经耗尽了自己的量程，并且线程刚从不超过两个时钟计时周期的等待中脱离出来。

如果不符合上述任何情况，则会通过将 AdjustIncrement 与线程当前基本优先级相加的方式计算该线程的新优先级。此外，如果线程已知是某个前台进程的一部分(意味着其内存优先级被设置为MEMORY\_PRIORITY\_FOREGROUND，这是由Win32k.sys在焦点改变时设置的)，此时将应用优先级分离提升 (PsPrioritySeparation)，将该提升值增加到新优先级的基础上。这种做法也叫作前台优先级提升。

最后，内核会检查新计算出来的优先级是否高于线程的当前优先级，为避免新优先级进入实时优先级范围，还会限制该优先级不超过 15。随后内核会将这个值设置为线程最新的当前优先级。如果应用了任何前台分离提升，那么内核会在KTHREAD的FreroundBoost字段中设置该值，使得 PriorityDecrement 值与分离提升值相等。

对于AdjustBoost 提升，内核会检查线程的当前优先级是否低于AdjustIncrement值(根据前文介绍可知，这是负责进行设置的线程对应的优先级)以及线程的当前优先级是否低于13。如果情况相符，并且线程并未禁用优先级提升，则会使用AdjustIncrement 优先级作为新的当前优先级，但该值最高只能达到13。同时KTHREAD的UnusualBoost 字段还包含了提升值，这也会导致 PriorityDecrement 值与锁所有权提升值相等。在所有情况下，只要存在 PriorityDecrement值，线程的量程都会基于KiLockQuantumTarget值重新计算，使其等同于一个时钟计时周期。这确保了前台和非正常提升会在一个时钟计时周期(而非通常的两个，或配置的其他个数的时钟计时周期)后被移除。如果请求了AdjustBoost 但线程已经在优先级 13 或14 下运行，或提升被禁用，也会发生这种情况。上述工作完成后，AdjustReason 将被设置为 AdjustNone。

1. **移除提升**

提升的移除是在 KiDeferredReadyThread 中应用重新计算出的提升值和量程的同时进行的。算法首先会检查所做调整的类型。

对于 AdjustNone 场景，意味着线程可能是因为抢占而变为就绪状态，如果量程已耗尽但时钟中断尚未发现这种情况，此时将重新计算该线程的量程，但前提是该线程运行在动态优先级级别下。此外线程的优先级也会重新计算。对于非实时线程的 AdjustUnwait或AdjustBoost 场景，内核会检查线程是否已经悄悄用完了自己的量程。如果是，或如果该线程的基本优先级为14或更高，或如果不存在PriorityDecrement并且该线程已经完成了超过两个时钟计时周期的等待，该线程的量程和优先级都会重新计算。

非实时线程可以重新计算优先级。为此需要获得线程的当前优先级，减去非正常前台提升值(后面这两项的组合便是PriorityDecrement)，最后再减1。随后对于这个新优先级值，再使用基本优先级作为下限进行界定，并将当前存在的所有优先级降低量进行清零(借此清除非正常提升和前台提升)。这意味着对于锁所有权提升或前文介绍过的任何其他非正常提升，所有提升值均已丢弃。另外，对于常规的AdjustUnwait 提升，由于需要减1,其优先级自然会降低一级。由于下界检查的存在，这种降低最终会止于基本优先级。还有一种情况会导致提升移除，这种移除是由 KiRemoveBoostThread 函数进行的。这是一种锁所有权提升规则导致的提升移除特例。该规则决定了设置线程在将自己当前优先级提供给被唤醒线程时，为避免“锁护航”，必须失去自己的提升。这种机制也被用于撤销定向延迟过程调用(DPC)以及针对 ERESOURCE 锁饥饿引起的提升而导致的提升。这个例程唯一的特殊之处在于，在计算新优先级时，会通过特别谨慎的处理来区分PriorityDecrement的 ForegroundBoost 和UnusualBoost 组件，以便为任何 GUI前台分离提升维持线程的累积。这种行为是从 Windows 7引入的，确保了依赖锁所有权提升的线在前台运行，或不在前台运行时均不会产生异常行为。

1. **多媒体应用程序和游戏的优先级提升**

客户端版本的 Windows引入了MMCSS驱动程序(本章前文曾介绍过)，该驱动程序是过%SystemRootSystem32 DriverslMMCSSsys 实现的，其目的在于确保注册了该驱动程序的应用程序可以实现无卡顿的多媒体播放。

客户端应用程序可通过调用AvSetMmThreadCharacteristics向MMCSS注册注册时还必须提供一个与注册表 HKLM\SOFTWARE\Microsot\WindowsNT\CurrentVersion\Multimedia\SystemProfile\Tasks 键下任何一个子键相符的任务名称。出厂状态的 Windows 包含下列任务。

可以在多媒体线程头调用函数AvSetMmThreadCharacteristics将当前线程设置为多媒体特性，执行完多媒体操作后再调用函数 AvRevertMmThreadCharacteristics将线程恢复到正常线程，AvRevertMmThreadCharacteristics的参数为AvSetMmThreadCharacteristics返回的句柄

1. 音频 (audio)。
2. 捕获 (capture)。
3. 分配(distribution)。
4. 游戏(games)。
5. 低延迟(low latency)。
6. 回放(playback)。
7. 专业音频(pro audio)
8. 窗口管理器(window manager)。

上述每个任务都包含一些信息，可用于区分不同任务的各种属性。对调度来说，最重要的一条属性是调度分类(scheduling category)，这是确定向MMCSS注册的线程的优先级时主要的考虑因素。

|  |  |  |
| --- | --- | --- |
| 分类 | 优先级 | 描述 |
| 高 | 23~26 | 除了关键系统线程，优先级比系统中任何其他线程更高的专业音频线程 |
| 中 | 16~22 | 诸如Windows Media Player 等前台应用程序的线程 |
| 低 | 8~15 | 所有不属于上述分类的其他线程 |
| 已耗尽 | 4~6 | 已耗尽自己的CPU份额，只有在没有其他更高优先级进程已经准备好运行时才能继续运行的线程 |

默认情况下，多媒体线程会获得 80%的可用CPU 时间，其他线程获得剩余的20%。修改注册表HKLM\ SOFTWARE\Microsoft\Windows NT\CurrentVersion\Multimedia\SystemProfile键下的SystemResponsiveness 注册表值可以更改这个比例。（也就是说这个键的默认值为0x14(20)）

MMCSS的功能不仅是简单的优先级提升。由于 Windows网络驱动程序和NDIS栈的本质特性，在收到来自网卡的中断后，DPC 是一种常见的延迟处理机制。但DPC 运行在高于用户模式代码的IRQL级别上

MMCSS会向网络栈发送一个特殊命令，借此让网络栈在媒体播放过程中对网络数据包进行限流(throttle)。按照设计，这种限流可以实现最大化播放性能，但代价是牺牲少量网络吞量

1. **自动提升**

自动提升(auto boost)是一种以解决前文提到的优先级反转问题为目标的框架。它的想法在于: 使用一种能够对相应线程的优先级(如果需要还可用于IO优先级)进行提升的方式来追踪锁的所有者和锁的等待者，借此实现对线程的进一步处理。有关锁的信息存储在KTHREAD结构内部由KLOCK\_ENTRY对象组成的静态数组中该机制的当前实现最多可使用6个项。每个KLOCK\_ENTRY 维持两个二叉树: 一个用于由线程所拥有的锁另一个用于正在被线程等待的锁。这些树可通过优先级进行键控(keyed)，因此需要通过恒定的时间来判断每个提升需要应用的最高优先级。如果需要提升，会将所有者的优先级设置为等待者的优先级。如果优先级过低会造成问题,该机制还可提升I/O 的优先级。与所有优先级提升机制类似，自动提升可实现的最大优先级为15。(实时线程的优先级永远不会被提升。)

该机制当前的实现会为推锁 (push lock)和受保护的互斥体同步基元使用自动提升框架，它们只会暴露给内核代码。该框架还会被一些执行体组件用于某些特殊情况。

上下文切换

内核会将来自老线程的这些信息存储起来，具体过程为: 将其推送给当前(老线程的)内核模式栈，更新栈指针，将栈指针保存到老线程的 KTHREAD 结构。随后该内核栈指针会被设置给新线程的内核栈，并加载新线程的上下文。如果新线程位于不同进程中，还会将该进程的页表目录载入一个特殊的处理器寄存器中，这样新进程的地址空间就可用了。如果需要交付的内核 APC 正处于挂起状态随后还会请求IRQL为1的中断。否则，控制权会被传递给新线程中被还原的指令指针，随后新线程会恢复执行

直接切换

Windows 8 和 Windows Server 2012 引入了一种名为直接切换(direct switch)的优化机制，可以让线程让渡出自己的量程并提升给另一个线程，随后接受让渡的线程可以立即调度到同一颗处理器上运行。在同步客户端/服务器场景中，这种方式可大幅提高吞吐量，因为客户端/服务器线程无法迁移至其他可能闲置或休止(parked)的处理器上运行。这种机制还可以这样理解: 在任何特定时间里，只能运行客户端或服务器线程，因此线程调度器应当将其视作一个单一的逻辑线程。

如果可行，KiDirectSwitchThread函数会真正执行切换。如果通过传递标志指出要尽可能使用直接切换，KiExitDispatcher 会调用这个函数。如果KiExitDispatcher通过另一个位标志做出了指示，还将应用优先级让渡，此时第一个线程的优先级会被“让渡”给第二个线程(前提是后者的优先级低于前者)。在当前实现中，这两个标志总是会配合使用(全部使用，或全部不使用)，这意味着在任何直接切换尝试中，都会同时尝试进行优先级让渡。直接切换也可能失败，举例来说，目标线程的相关性可能会阻止它在当前处理器上运行。然而只要成功，第一个线程的量程就会传递给目标线程，第一个线程将失去自己的剩余量程。

直接切换目前主要用在下列场景中。

1. 线程用了 SignalObjectAndWait 这个WindowsAPI(或内核等价的NtSignalAndWaitForSingleObject)。
2. ALPC。
3. 同步远程过程调用(RPC)。

同步远程过程调用（Synchronous Remote Procedure Call，简称同步RPC）是一种分布式系统中的通信协议，允许一个计算机（客户端）请求另一个计算机（服务器）执行特定的函数或方法。在同步RPC中，客户端发起请求后会等待服务器响应。直到服务器处理完请求并返回结果之前，客户端会处于阻塞状态，无法进行其他操作。

同步RPC的工作流程通常包括以下几个步骤：

1. 客户端程序调用一个本地存根函数（Local Stub Function），该函数看起来与远程过程调用非常相似。
2. 本地存根函数将过程调用参数进行序列化（Marshal），将其编码为可以在网络上传输的格式。
3. 客户端将编码后的参数发送到服务器。
4. 服务器接收到请求后，将参数解码（Unmarshal）并执行相应的过程。
5. 执行完毕后，服务器将结果序列化并发送回客户端。
6. 客户端解码服务器返回的结果，并将其作为本地存根函数的返回值。

同步RPC的主要优点是简单和易于实现，因为客户端和服务器之间的通信模型非常直观。然而，它也存在一些缺点，如性能和可伸缩性方面的问题。由于客户端在等待服务器响应期间处于阻塞状态，应用程序的执行可能会受到影响。此外，如果服务器需要处理大量请求，同步RPC可能会导致服务器资源不足。

为了解决这些问题，可以使用异步远程过程调用（Asynchronous RPC）。在异步RPC中，客户端在发起请求后不会阻塞等待响应，而是继续执行其他任务。当服务器返回结果时，客户端会通过回调函数或其他机制处理响应。这种模型能更好地利用计算资源，提高性能和可伸缩性。

1. COM远程调用[目前仅支持多线程单元(multithreaded apartment，MTA)之间的调用]。

调度场景

1. **自愿切换**

线程可以调用 WaitForSingleObject 或 WaitForMultipleObjects 等 Windows 等待函数，进而等待某些对象(如事件、互斥体、信号量、I/O 完成端口、进程、线程、窗口消息等),从而让自己进入等待状态，自愿放弃对处理器的使用。

1. **抢占**

在这种调度场景中，当高优先级线程做好准备就绪后，低优先级的线程会被抢占。出现这种情况的原因有多种，如高优先级线程的等待已完成(另一个线程所等待的事件发生了)、一个线程的优先级被提高或降低了。

当一个线程被抢占后，会被放置到所处优先级对应的就绪队列的最顶部，当发起抢占的线程运行完毕后，被放置的线程即可继续在自己的量程内运行。

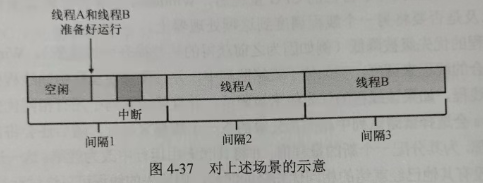
1. **量程结束**

当运行中的线程耗尽了自己的CPU量程后，Windows必须决定是否要降低该线程的优先级，以及是否要将另一个线程调度到这颗处理器上。

如果线程的优先级被降低(例如因为之前获得的某些提升而被降低)，Windows 会寻找另一个更适合的线程来调度，例如位于就绪队列中，并且比当前运行中的线程的新优先级更高的另一个线程。如果该线程的优先级未被降低，并且就绪队列中还有相同优先级的其他线程，Windows会选择就绪队列中相同优先级的下一个线程来运行，随后还会将前一个线程移动到队列尾部，为其分配一个新的量程值，并将其状态由运行中改为就绪。如果没有其他已经就绪的相同优先级的线程，则原先的线程可以继续运行一个量程。

如上所述，Windows 并不会简单地依赖基于时钟间隔计时器的量程来调度线程，而是会使用精确的CPU时钟周期计数来维持量程目标。Windows还会使用该计数器来确当前是否适合终止线程的量程。

Windows会精确记录所调度线程运行过程中使用的CPU时钟周期数量(不包含中断处理)。此外Windows还会记录在线程量程结束时，线程应该使用掉的时钟周期数量程目标。因此上述针对线程A 的那些不公平决策在 Windows 中将不再出现。此时会发生下列情况，如下图



1. 线程A和线程B在一个时钟间隔期间变为就绪状态
2. 线程A开始运行但中断了一段时间。处理该中断所用的时间不会被记账给该线程
3. 中断处理完成后线程 A 再次开始运行，但很快遇到了下一个时钟间隔。调度器检查为线程A记账的CPU 时钟周期数，并将其与量程计数时预计记账的CPU 时钟周期数进行对比。
4. 由于前一个数值远小于预期值，因此调度器会假设线程 A 是在一个时钟间隔的期间内运行的，并且可能遇到了额外的中断。
5. 线程A 的量程额外增加一个时钟间隔，量程目标重新计算。这样，线程A就机会运行一个完整时钟间隔了。
6. 在下一个时钟间隔，线程A 用完了自己的量程，线程 B 得到机会开始运行
7. **终止**

当一个线程运行完成(可能是因为它从主例程返回时调用了 ExitThread，或被TerminateThread终止)后，它会从运行中状态转换为已终止状态。如果该线程对象上没有打开的句柄，该线程会被从进程线程列表中删除，相关数据结构也会被撤销分配并释放

空闲线程

如果CPU上不存在可运行的线程，Windows 会调度该CPU的空闲线程每颗CPU都有自己专用的空闲线程。这是因为在多处理器系统中，当一颗CPU 正在执行线程时，其他CPU可能没有任何线程需要执行。每颗CPU的空闲线程均可通过该CPU的PRCB 中的指针找到。所有空闲线程都属于空闲进程。从很多方面来看，空闲进程和空闲线程都是一种特殊的情况。当然,它们可以用EPROCESS/KPROCESS和ETHREAD/KTHREAD 结构来表示,但它们并非执行体管理器进程和线程对象。空闲进程也不会出现在系统进程列表中。(因此它不会出现在内核模式调试器!process 0 0命令的输出结果中。)然而空闲线程或线程及其进程可通过其他方式找到。此外，空闲进程并非一种对象管理器进程对象，其空闲线程也不是对象管理器线程对象。相反，初始空闲线程和空闲进程结构是静态分配的，可用于在进程管理器和对象管理器被初始化之前实现系统自举。后续的空闲线程结构则是在系统中更多处理器上线后动态分配的(简单地从非换页池中分配，无须对象管理器参与)。一旦进程管理初始化完成，就会使用一个特殊变量PsIdleProcess 指向空闲进程。

空闲进程和线程的进程ID和线程ID(即客户端ID，或调试器输出结果中所示的 Cid)均为0，其PEB 和 TEB 指针也是0，空闲进程或其线程的很多其他字段也是0。因为空闲进程没有用户模式地址空间，其线程不执行任何用户模式代码，所以也不需要用于管理用户模式环境所需的各类数据。

对于空闲进程，最有趣的一个反常之处可能在于，Windows 会将空闲线程的优先级报告为0。然而实际上，空闲线程的优先级成员值是没有意义的，因为这些线程只有在没有其他线程运行时才会选中并进行分配。

正如空闲线程对于选择执行是一种特例，对于抢占，它们也是一种特例。空闲线程的例程 KiIdleLoop 会执行一系列操作，以避免自己被常规方式运行的其他线程抢占。当处理器上不存在可运行的非空闲线程时，该处理器会在自己的 PRCB 中被标记为空闲。随后，如果有线程被选中在该空闲处理器上执行，该线程的地址会存储到空闲处理器的PRCB中的NextThread 指针内。在每一次循环过程中，空闲线程都会检查该指针。

虽然不同体系结构之间的流程细节有所差异(这也是少数用汇编语言，而非 C 语言编写的例程之一)，但空闲线程的基本操作步骤如下。

1. 空闲线程短暂开启中断，进而让任何挂起的中断顺利交付，随后(使用 x86 和x64 处理器的 STI和 CLI指令)关闭中断。这是必需的，因为空闲线程执行过程中的大部分时间里中断都是关闭的。
2. 在某些体系结构的调试版系统 (debug build)中，空闲线程会检查是否有内核模式调试器正试图打断系统的执行。如果有，则会提供这样的访问。
3. 空闲线程会检查处理器上是否有任何挂起的 DPC。如果DPC在加入队列时没有生成DPC中断，那么该DPC会被挂起。如果 DPC 正在挂起，空闲循环会调用KiRetireDpcList来交付DPC。这样做还会执行计时器到期处理以及延迟就绪处理。KiRetireDpcList 必须在中断关闭的情况下进入，因此第1步结束时中断会关闭。KiRetireDpcList 退出时，中断依然处于关闭状态。
4. 空闲线程检查是否请求了量程终止处理。如果有，则调用 KiQuantumEnd处理该请求。
5. 空闲线程检查是否有线程被选中随后在该处理器上运行。如果有，则会分配该线程。出现这种情况的原因有多种，例如第3步所处理的DPC或计时器过期解决了等待线程的等待状态，或其他正处于空闲循环的处理器为该处理器选择了一个线程来运行。
6. 如果接到请求，空闲线程会检查已就绪要在其他处理器上运行的线程，并且在可行的情况下，会将其中一个线程放在本地进行调度。
7. 空闲线程调用已注册的电源管理处理器空闲例程(如果需要执行任何电源管理功能的话)，该例程可能位于处理器的电源驱动程序(例如intelppm.sys)中，如果此类驱动程序不可用，也可能位于HAL中。

但在实际的操作系统设计中，引入空闲线程具有一定的优势。以下是一些原因：

1. 简化调度逻辑：通过将空闲线程纳入调度机制中，操作系统可以简化线程调度的逻辑。这意味着操作系统不需要针对CPU空载的特殊情况进行单独处理。在任何时候，操作系统都可以假设至少有一个可运行的线程（即空闲线程）。
2. 低优先级系统任务：空闲线程可以在系统空闲时执行一些低优先级的任务，例如内存整理和管理。这使得操作系统可以在不影响关键任务性能的前提下，进行一些维护工作。
3. 节能和降温：空闲线程可以将处理器置于低功耗状态，从而节省能源并降低系统温度。这对于移动设备和服务器尤为重要，因为它们需要在有限的电源和散热条件下工作。
4. 处理器兼容性：不同的处理器有不同的低功耗模式和指令集，空闲线程允许操作系统根据当前处理器的特性来执行相应的操作。这有助于确保在多种处理器上实现良好的能源管理。
5. 系统性能统计：空闲线程的存在可以帮助操作系统收集有关系统性能的统计信息，例如CPU使用率。通过监控空闲线程的运行时间，操作系统可以计算出CPU的使用率和空闲率，从而更好地了解系统资源的使用情况。

线程的挂起

线程可分别使用 SuspendThread和 ResumeThread API函数显式挂起和恢复。每个线程有一个挂起计数，每次挂起会导致该计数增加，恢复会导致该计数减少。如果数值为0，意味着线程可以自由执行，否则将无法执行。

为了实现挂起，该线程需要在内核 APC 中排队。当线程切换为执行状态时，会首先执行该APC，这会让线程处于等待状态，只有在获得事件信号后线程才能最终恢复。如果处于等待状态的线程收到了挂起请求，这种挂起机制将产生显而易见的缺陷,因为这意味着为了挂起该线程，反而要首先将其唤醒。这可能会导致内核栈换入(如果该线程的内核栈之前处于换出状态)。Windows 8.1和Windows Server 2012 R2增加了一种名为轻型挂起(lightweight suspend)的机制，可以在不使用APC机制的前提下将处于等待状态的线程直接挂起，此时会直接操作线程在内存中的对象将其标记为已挂起。

（深度）冻结

在冻结机制中，进程会进入挂起状态，并且无法通过针对进程中的线程调用ResumeThread更改这种状态。如果系统需要挂起 UWP 应用，这种机制将非常有用。当一个 Windows应用切换到前台就会发生这种情况，例如在平板模式下有另一个应用被切换到前台，或在桌面模式下有应用被最小化。在这种情况下，系统会为应用提供大约 5s 的时间来完成工作，通常这段时间可以用来保存应用程序的状态。保存状态的操作很重要，因为如果内存资源变得非常低，Windows 应用可能会在不事先通知的情况下被终止。如果应用被终止，那么还可在下次启动时重新加载其状态，在用户感觉中，这个应用就好像从来不曾被关闭一样。冻结一个进程，意味着同样以无法通过 ResumeThread 唤醒的方式挂起进程中的有线程。KTHREAD 结构中的一个标志可以用来表示某个线程是否被冻结。为了让线程可以执行，其挂起计数必须为 0，并且必须清除该冻结标志。

深度冻结增加了一个额外约束: 进程中新建的线程同样无法启动。举例来说，如果通过调用CreateRemoteThreadEx 在深度冻结的进程中新建了线程，线程在实际启动前也会被冻结。这也是冻结功能最典型的用法。

进程和线程冻结功能并未直接暴露到用户模式。它们主要被进程状态管理器(Process State Manager，PSM)服务在内部使用，该服务负责向内核发出请求以便进行深度冻结和解冻。我们也可以使用作业进行进程冻结。冻结和解冻作业的能力尚未公开文档化，但只需使用标准的NtSetInformationJobObject系统调用即可实现。这一能力主要被 Windows 应用所用，毕竟所有 Windows 应用的处理都是通过作业的方式进行的。此类作业可能包含一个进程(Windows应用本身)，但也可能包含与该 Widows 应用有关的后台任务承载进程因此可通过一步操作冻结或解冻这种作业下的所有进程。

线程的选择

逻辑处理器需要选择接下来运行的线程时，会调用KiSelectNextThread调度器函数这种情况会发生在多种场景中。

1. 发生一次相关性硬变更，使得当前运行中或待命的线程无法符合在所选逻辑处理器上继续运行的条件，因而需要选择另一个线程来运行。
2. 当前运行中的线程量程结束，而它之前运行所用的对称多线程(Symmetric Multithreading，SMT)集已经变得非常忙碌，而理想节点上的其他SMT集处于完全空闲的状态。调度器针对当前线程执行量程终止迁移，因此必须选择另一个线程来运行。
3. 等待操作完成，等待状态寄存器中存在等待处理的调度操作(也就是说，其优先级或相关性位已经设置)。

在这些情况下，调度器将表现出如下行为。

1. 调度器调用 KiSelectReadyThreadEx，搜索处理器应该运行的下一个已就绪线程并检查是否找到这样的线程。
2. 如果未找到就绪线程，则会启用空闲调度器，并选择空闲线程来执行。如果找到了就绪线程，则会酌情将其以就绪状态放置在本地或共享的就绪队列中。

只有在逻辑处理器需要选择(但尚未运行)下一个已调度线程时，才需要执行KiSelectNextThread 操作(这也是该线程会进入就绪状态的原因)。不过其他情况下，逻辑处理器会立即运行下一个已就绪线程，或在此类线程不可用时执行其他操作(但不会进入空闲状态)，例如在遇到下列情况时。

1. 优先级变化导致当前待命或运行中的线程不再是所选逻辑处理器上优先级最高的已就绪线程，这意味着现在必须运行一个优先级更高的已就绪线程。
2. 线程已通过调用YieldProcessor或NtYieldExecution 明确放弃，另一个线程可能已做好执行准备。

YieldProcessor是一个编译器内联函数，它在Windows中用于为CPU提供一种提示，表明当前线程愿意放弃其剩余的时间片。这通常用于自旋锁和其他低级同步原语的实现中，以减少因忙等待导致的CPU资源浪费。

NtYieldExecution是一个Windows内核API，它允许当前执行的线程主动放弃其剩余时间片，让调度器选择下一个要执行的线程。这个API在某些情况下可以用于提高系统的响应性和整体性能，尤其是当一个线程完成了其关键任务，但在当前时间片内不需要继续执行其他操作时。

1. 当前线程的量程已到期，相同优先级级别的其他线程同样需要得到机会来运行
2. 线程失去了自己的优先级提升，导致类似上一种情况的优先级变化。
3. 空闲调度器正在运行，需要检查在空闲调度被请求，以及空闲调度器开始运行这两种状态期间，是否没有出现过已就绪线程。

有一种简单的方法可以用来记住所运行例程之间的差异: 检查逻辑处理器是否需要运行一个不同的线程(此时会调用 KiSelectNextThread)，或是只要可行就运行另一个不同的线程(此时会调用KiSelectReadyThreadEx)。无论任何情况，由于每颗处理器都属于一个共享的就绪队列(由 KPRCB 指向)，KiSelectReadyThreadEx 可以直接检查当前逻辑处理器(LP)的队列，移除所找到的第一个优先级最高的线程，除非该线程的优先级低于当前运行中的线程(取决于当前线程是否被允许继续运行，不过KiSelectNextThread 场景并不是这种情况)。如果不存在更高优先级的线程(或没有已就绪线程)，则不会返回任何线程。

空闲调度器

空闲线程运行时，会检查是否启用了空闲调度器。如果启用，空闲线程会开始扫描其他进程的就绪队列，以查找可由自己调用 KiSearchForNewThread 来运行的线程。与该操作有关的运行开销并不会记账给空闲线程时间，但会记账给中断和DPC 时间(记账给处理器)，因此空闲调度时间可以视作系统时间。

多处理器系统

多处理器Windows 系统会试图将线程调度到对线程来说最优的处理器上，所以不仅要考虑线程的首选处理器和之前运行时使用的处理器，还要考虑多处理器系统本身的配置。因此，虽然 Windows 会尝试着将优先级最高的可运行线程调度到所有可用CPU上，但只能保证在某个CPU上运行一个最高优先级的线程。对于共享的就绪队列(线程没有相关性限制)，这种保证更强一些。每个共享的处理器组都可以运行至少一个最高优先级线程。Windows 可支持的3种多处理器系统类型(SMT、多核、NUMA)。

1. **封装集和SMT集**

在处理逻辑处理器拓扑时，Windows 会使用KPRCB中的5个字段来确定适合的调度决策。

* 1. CoresPerPhysicalProcessor：每个物理处理器中的核心数量。通过CPUID指令查询得到。
  2. LogicalProcessorsPerCore：每个核心中的逻辑处理器数量。用于确定是否为SMT（Simultaneous Multi-Threading，如超线程）集的一部分。通过CPUID指令查询得到。
  3. PackageProcessorSet：相关性掩码，描述同一处理器组中的其他逻辑处理器是否属于同一物理处理器。
  4. CoreProcessorSet：将同一内核中的其他逻辑处理器连接在一起，也称为SMT集。
  5. GroupSetMember：定义处理器组中哪个位掩码(bitmask)用于标识当前逻辑处理器。

逻辑处理器拓扑描述了处理器内部组件之间的关系，以及它们在多核和多线程环境下的组织结构。处理器拓扑提供了有关处理器核心、超线程（如果支持）以及缓存等组件之间关系的信息。了解逻辑处理器拓扑有助于优化多线程应用程序的性能，因为可以根据拓扑信息来合理地分配线程和任务。

SMT（Simultaneous Multi-Threading，同时多线程）是一种处理器技术，它允许一个物理处理器核心同时执行多个线程。通过在一个核心中共享处理器资源，例如寄存器、缓存和执行单元，SMT 可以提高处理器的利用率和性能。SMT 技术的一个典型实现是 Intel 的超线程技术（Hyper-Threading）。

SMT 的主要优势是它能更有效地利用处理器资源。在传统的单线程处理器核心中，由于诸如内存延迟、分支预测错误等原因，某些处理器资源可能会在一段时间内闲置。通过允许一个核心在同一时间周期内处理多个线程，SMT 可以减少这些资源的闲置时间，从而提高处理器的吞吐量和性能。

然而，SMT 技术也有一些潜在的缺点。在多线程执行时，处理器资源需要在多个线程之间共享，这可能导致资源争用，从而降低某些线程的性能。此外，SMT 技术可能会导致缓存容量不足，因为多个线程可能需要访问相同的缓存资源。这可能导致缓存抖动现象，从而降低系统性能。尽管如此，在许多情况下，SMT 技术仍然能够带来性能上的提升。

1. **NUMA系统**

Windows还支持另一种多处理器系统类型:非一致内存访问架构(Non-Uniform Memo Architecture，NUMA)。在NUMA系统中，处理器会被分为一组组名为节点的较小单元，每个节点有自己的处理器和内存，通过一种缓存一致的互联总线连接组成大规模系统。说这些系统是“非一致”的，是因为每个节点都有自己的本地高速内存。虽然任何节点中的任何处理器都可以访问所有内存，但节点本地内存的访问速度通常更快。

内核会将NUMA系统中有关每个节点的信息保存在名为KNODE的数据结构中。内核变量 KeNodeBlock 是一种指针数组，指向每个节点的KNODE 结构。(dt nt!\_KNODE)

1. **处理器组的分配**

在查询系统拓扑并在逻辑处理器、SMT 集、多核封装以及物理处理器插槽之间构建各种关系时，Windows 会将处理器分配到相应的组，借此(通过前文提到的扩展相关性掩码)描述其相关性。这是由 KePerformGroupConfiguration 例程进行的，系统初始化过程中，在执行任何其他第1阶段工作前会调用该例程。整个过程如下。

1. 该函数查询检测到的所有节点 (KeNumberNodes)，计算每个节点的容量，即节点可以包含多少颗逻辑处理器。这个值会存储到KeNodeBlock数组的MaximumProcessors中，这个字段标识了系统中的所有NUMA节点。

如果系统支持NUMA 临近ID(ProximityD)，则会查询每个节点的临近ID并保存到节点块中。

1. 分配NUMA距离数组 (KeNodeDistance)，计算每个NUMA 节点之间的距离。接下来的一系列步骤将处理用于取代默认 NUMA 分配，由用户配置的特定选项。例如假设有系统安装了Hyper-V，并将虚拟机监控程序配置为自动启动。如果CPU不支持扩展的虚拟机监控程序接口，则只能启用一个处理器组，所有NUMA 节点都将关联到组0(只要能容纳在内)。因此这种情况下Hyper-V将无法发挥处理器数量超过64颗的计算机的全部性能
2. 该函数会检查加载器 (loader)是否传递了任何静态组分配数据(进而可由用户来配置)。这些数据决定了每个NUMA 节点的临近信息和组分配。
3. **每个组中的逻辑处理器**

一般来说，Windows会为每个组分配64颗处理器。但我们可以使用不同的加载选项

这些选项可以通过设置BCD（引导配置数据）来配置。要配置这些选项，请按照以下步骤操作：

打开命令提示符（cmd）或PowerShell，确保以管理员权限运行。

输入以下命令以编辑BCD设置：

bcdedit /set {current} groupsize 数值

将“数值”替换为想要的每个处理器组包含的处理器数量（2的某次幂，1到64之间）。例如8

bcdedit /set {current} groupaware on 设置可确保进程在组 0 以外的组中启动。 这增加了驱动程序和组件之间跨组交互的机会。 该选项还会修改旧函数 KeSetTargetProcessorDpc、KeSetSystemAffinityThreadEx 和 KeRevertToUserAffinityThreadEx 的行为，以便它们始终对包含可用逻辑处理器的最高编号组进行操作 。

完成设置后，需要重启计算机以使更改生效。

如果你想恢复默认设置，可以使用以下命令：

bcdedit /deletevalue {current} groupsize

bcdedit /deletevalue {current} groupaware

bcdedit /set onecpu on 用于将当前操作系统加载程序配置为使用一个逻辑处理器。

某些极端情况下，一个封装中的逻辑处理器可能无法纳入同一个组，此时 Windows会调整这些数字，以确保一个封装可纳入一个组。为此要减小 CoresPerPhysicalProcessor数值，如果SMT也无法容纳，则会减小 LogicalProcessorsPerCore 数值。但该规则有个外:系统实际上在一个封装里包含了多个 NUMA 节点(少见，但有可能)。在此类多芯片模组(Multi-chip module，MCM)中，会通过一个晶片 (die)/封装容纳两套内核与内存控制器。如果ACPI静态资源相关性表(Static Resource Afinity Table，SRAT)定义MCM 包含两个NUMA 节点，Windows 可能会将这两个节点关联到不同的组(取决于组配置算法)。此时一个MCM封装可能被分散到多个组中。

除了导致棘手的驱动程序和应用程序兼容性问题(其设计目的就是帮助开发者发现问题，确定根源)，这些选项还会对计算机造成更大影响:就算非 NUMA 类型的计算机也会迫使其表现出NUMA的相关行为。这是因为Windows 绝不允许一个NUMA节点越多个组，这一点从分配算法中就可以看出来。因此如果内核要创建过小的多个组，这两个组就必须拥有各自的NUMA 节点。

1. **逻辑处理器状态**

除了共享的和本地就绪的队列与摘要，Windows 还会通过两个位掩码追踪系统中的处理器状态。

1. KeActiveProcessors。这是活动处理器掩码，会为系统中每颗可用处理器设置一个位。如果许可方面的限制导致所运行 Windows 版本无法支持所有可用物理处理器，那么该掩码的数值可能会小于实际处理器数量。我们可以使用 KeRegisteredProcessors 变量查看计算机中已经获得许可的处理器数量，此处的处理器是指物理封装。
2. KeMaximumProcessors。这是在许可限制内允许的逻辑处理器(包括未来可以额外动态添加的处理器)数量最大值。通过调用 HAL并检查ACPI SRAT表(如果有的话)，还可借此了解平台本身在这方面的局限。
3. **调度器的伸缩性**

在多处理器系统中，一颗处理器可能需要修改另一颗处理器的每CPU调度数据结构,例如向某颗处理器插入想要在该处理器上运行的线程。因此我们需要使用针对每个PRCB排队的自旋锁同步这些结构，这是在DISPATCH\_LEVEL级别上进行的。因此可以在锁定特定处理器的 PRCB的同时进行线程的选择工作。如果有必要，还可以额外多锁定一颗处理器的PRCB，例如在线程窃取 (thread stealing)场景中(具体描述见下文)。通过使用细粒度的每个线程自旋锁，还可对线程上下文的切换进行同步。

对于每颗CPU上处于延迟就绪状态的线程，也会维持一个列表(DeferredReadyListHead)。这代表已经就绪可以运行，但执行体尚未就绪的线程，因此实际就绪操作被延迟到其他更合适的时间。由于每颗处理器只能操作它自己的每颗处理器延迟就绪列表，因此该列表并不会被 PRCB自旋锁同步。延迟就绪线程列表是由 KiProcessDeferredReadyList处理的，但在处理前，首先需要通过其他函数修改进程或线程相关性、优先级(包括提升后的优先级)或量程值。

该函数会为列表中的每个线程调用KiDeferredReadyThread。这可能导致线程立即运行，被放入处理器的就绪列表，或如果处理器不可用则可能会(以待命状态)放入另一颗处理器的延迟就绪列表，或通过另一颗处理器立即执行。该属性是由内核休止引擎在休止内核时使用的，休止内核时，所有线程均会放入延迟就绪列表，随后再加以处理。由于 KiDeferredReadyThread会跳过已休止的内核，因此这会导致处理器上的所有线程被转移至另一颗处理器。

1. **相关性**

线程相关性（Thread Affinity）是指一个线程在处理器之间的亲和力。在多处理器系统中，线程相关性决定了线程在哪些处理器上运行。这是一种优化方法，可以提高多处理器系统上的性能和资源利用率。

线程相关性是通过一个叫作“处理器亲和掩码”（Processor Affinity Mask）的值表示的。这个掩码是一个位图，其中每一位代表一个处理器。如果掩码中某一位的值为1，那么线程可以在对应的处理器上运行；如果为0，则表示线程不允许在该处理器上运行。

* 1. 调用SetThreadAffinityMask 函数为特定线程设置相关性。
  2. 调用SetProcessAfinityMask 函数为进程中的所有线程设置相关性
  3. 任务管理器和 Process Explorer 提供了该功能的GU版本。若要使用该功能,请右击进程，选择Set Afinity。此外(来自 Sysinternals 的)Psexec 工具还提供了该功能的命令行版本
  4. 让进程成为作业的成员，并使用SetInformationJobObject函数设置影响整个作业的相关性掩码。
  5. 编译应用程序时，在映像文件头指定相关性掩码。

我们还可在链接时为映像设置uniprocessor标志。如果设置了该标志，系统会在创建进程时选择一颗处理器(MmRotatingProcessorNumber)，并将其作为进程相关性掩码分配从第一颗处理器开始分配，随后对该组中的所有处理器进行轮询。

1. **扩展的相关性掩码**

为了支持64 颗以上的处理器(这是最初的相关性掩码结构本身存在的限制，64位系统上该掩码只有64位)，Windows 使用了一种扩展的相关性掩码: KAFFINITY\_EX。这是一种相关性掩码数组，每个可支持的处理器组对应其中一个元素(目前定义为20个).

当调度器需要从扩展的相关性掩码中引用一颗处理器时,首先会使用组编号解除对正确位掩码的引用(de-reference)，随后直接访问所需的相关性。内核API并不暴露扩展的相关性掩码，此时需要由 API 的调用方将组编号作为参数输入，随后即可获得该组的遗留相关性掩码。然而在 Windows API中，通常只能查询并获得有关一个组的信息，即正在运行当前线程的那个组(这是固定的)。

我们可以在创建时使用扩展的相关性，为此需要在创建新线程，或针对现有线程调用SetThreadGroupAffinity 时，在线程属性列表(PROC\_THREAD\_ATTRIBUTE GROUPAFFINITY)中指定线程编号。

1. **系统相关性掩码**

Windows 驱动程序通常会在调用线程或任意线程的上下文中执行(也就是说，通常并不在 System 进程的安全界限内)，因此目前运行的驱动程序代码可能会受制于应用程序开发者设置的相关性规则。这些规则与驱动程序的代码无关，甚至可能导致无法正确处中断或其他排队的工作。因此驱动程序开发者必须通过某种机制暂时绕过用户线程的相性设置，为此可使用 KeSetSystemAffinityThread(Ex)/KeSetSystemGroupAffinityThread元KeRevertToUser-AffinityThread(Ex)/KeRevertToUserGroupAffinityThread这些API

KeSetSystemAffinityThread：此函数用于将当前线程的处理器亲和性设置为系统范围内的所有处理器。通过这样做，驱动程序代码可以在所有可用处理器上运行，而不受应用程序设置的相关性规则的限制。

KeSetSystemAffinityThreadEx：此函数类似于KeSetSystemAffinityThread，但允许指定一个新的亲和性掩码，而不是简单地使用所有处理器。

KeSetSystemGroupAffinityThread：此函数用于在处理器组的范围内设置当前线程的处理器亲和性。它允许您为线程指定一个处理器组和一个亲和性掩码。

KeRevertToUserAffinityThread：在使用KeSetSystemAffinityThread或KeSetSystemAffinityThreadEx修改线程亲和性之后，使用此函数可以恢复线程的原始亲和性设置。

KeRevertToUserGroupAffinityThread：在使用KeSetSystemGroupAffinityThread修改线程亲和性之后，使用此函数可以恢复线程在组范围内的原始亲和性设置。

1. **理想处理器和上一颗处理器**

每个线程有3个CPU成员，这些信息存储在内核线程控制块中

* 1. **理想处理器**。这是运行该线程的首选处理器。
  2. **上一颗处理器**。这是上一次运行该线程所用的处理器。
  3. **下一颗处理器**。这是线程即将运行或准备好下次运行要使用的处理器。

这种方式会假设一个进程中的所有线程承担了等量工作。但多线程进程的实际情况往往并非如此，通常来说，此类进程会有一个或多个负责常规事务的线程，以及大量实际承担工作任务的线程。因此多线程应用程序如果希望充分利用整个平台，往往需要通过SetThreadIdealProcessor函数为自己的线程指定理想处理器编号。为了充分发挥处理器组的作用，开发者应当调用SetThreadIdealProcessorEx，该函数可用于为相关性选择组编号。

在64位Windows中，可以使用KNODE中的Stride字段均地分配进程中新建的线程。Stride(步幅)是一种标量数值，代表特定 NUMA 节点中，为了获得一个新的独立处理器片段，而需跳过的相关性位数，其中“独立”是指位于另一个内核(对于SMT 系统)或另一个封装(对于非SMT的多核系统)中。由于32位Windows不支持大型处理器配置系统，因此并不使用Stride，而是会直接选择下一个处理器编号，并尽可能试图避免共享同一个SMT集。

1. **理想节点**

在NUMA 系统中创建进程时，会为该进程选择理想节点。第一个进程会分配给节点0，第二个进程分配给节点 1，以此类推。随后，还会从进程的理想节点中选出该进程中线程的理想处理器。进程中第一个线程的理想处理器会分配给节点中的第一颗处理器。随着相同理想节点中同一个进程内创建出额外的线程,下一颗处理器将充当下一个线程的理想处理器，以此类推。

1. **CPU集**

Windows10和Windows Server 2016引入了一种名为CPU集(CPU set)的机制。该机制是相关性的一种体现形式，可以作为整体应用给系统(包括系统线程活动)、进程，甚至单个进程。例如，低延迟音频应用程序可能需要独占使用某颗处理器，而系统的其余部分可以使用其他处理器。CPU 集提供了实现这种做法的一种方式。

GetSystemCpuSetInformation可返回由包含了每个CPU集数据的SYSTEM\_CPU\_SET\_INFORMATION组成的数组在当前实现中，一个CPU 集可以等同于一颗 CPU。这意味着所返回的数组，其长度等同于系统中逻辑处理器的数量。每个CPU集可通过其ID加以标识，而具体数值可以任意选择为256(0x100)外加CPU索引(0、1······)。为了给进程和特定线程设置CPU集，必须将这些ID分别传递给 SetProcessDefaultCpuSets 和 SetThreadSelectedCpuSets 函数。

Windows API目前缺乏减少系统CPU集的能力。但调用NtSetSystemInformation 系统调用即可获得这样的能力。不过为了成功调用,调用方必须具备SelncreaseBasePriorityPrivilege特权

多处理器系统中线程的选择

KiSearchForNewThread 最初会检查是否已经有线程被选择在这颗处理器上运行(为此需要读取NextThread 字段)。如果有，则会立刻分发该线程使其进入运行中状态。否则会调用KiSelectReadyThreadEx例程，并且针对所找到的线程执行相同操作。

如果未找到任何线程，该处理器会被标记为空闲(即使空闲线程尚未开始执行)，随后会开始启动对其他(共享的)逻辑处理器上的队列所进行的扫描(与其他标准算法不同并不会在此时就放弃)。然而如果处理器的内核已经休止，该算法将不再试图检查其他逻辑处理器，此时更适当的做法是让内核进入休止状态，而非用新工作让它保持忙碌。

除了上述两种情况，其他情况下此时都会运行工作窃取 (work-stealing)循环。随后这些代码会检查当前 CPU 的共享就绪队列，并开始循环调用KiSearchForNewThreadOnProcessor。

工作窃取（work-stealing）是一种并行计算中的任务调度策略，主要用于多线程环境。它的目的是平衡处理器或线程之间的负载，使得处理器能够尽可能地被充分利用，从而提高系统的整体性能。

在工作窃取策略中，每个线程都有自己的任务队列，线程从自己的任务队列中获取任务并执行。当一个线程的任务队列为空时，它会尝试从其他线程的任务队列中“窃取”任务。这样，即使某个线程的任务队列为空，它仍然可以从其他繁忙线程那里获取任务，从而继续执行。这有助于减少线程空闲时间，并提高系统的并行度

处理器的选择

1. **在具备空闲处理器时，为线程选择处理器**

当一个线程变为运行就绪状态，将调用 KiDeferredReadyThread 调度器函数。这会让Windows 执行两个任务:酌情调整优先级并刷新量程；为该线程选择最佳逻辑处理器。

Windows 首先会查询该线程的理想处理器，并在该线程的硬相关性掩码范围内计算空闲处理器集。随后按照下列方式修剪这个集。

* 1. 任何由内核休止机制休止的空闲逻辑处理器均会被移除。(如果这个操作导致无法剩下任何空闲处理器，则空闲处理器的选择会被忽略，随后调度器的行为类似于无可用空闲处理器时的情况(具体描述请参阅下一节)。
  2. 任何不包含在理想节点中的空闲逻辑处理器均会被移除(除非这会导致所有空闲处理器被淘汰)。
  3. 在SMT系统中，任何非空闲 SMT 集会被移除，哪怕这可能会导致理想处理器本身被淘汰。换句话说，相比理想处理器，Windows 更倾向于选择非理想的空闲SMT集。
  4. Windows会检查理想处理器是否位于剩余空闲处理器集中。如果不在其中，则随后必须寻找最适合的理想处理器。为此首先需要检查该线程上一次运行时所用处理器是否位于剩余空闲集中。如果是，则会将该处理器看作临时理想处理器并选择它。(理想处理器需要尽可能实现最大化的处理器缓存命中率,而选择线程上一次运行所用的处理器是实现这一目标的一种好办法。)如果上次使用的处理器未包含在剩余的空闲集中，Windows 会检查当前处理器(即当前正在执行此调度代码的处理器)是否包含在这个集中。如果在，则会应用上一步所述逻辑。
  5. 如果上一颗以及当前处理器都不空闲，Windows 会再次执行一次修剪操作，删除未包含在同一个SMT 集中并充当理想处理器的任何逻辑处理器。如果没有剩下任何理器，Windows将移除未包含在当前处理器所在SMT集中的任何处理器(除非这样做同样会淘汰所有空闲处理器)。换句话说，Windows 更倾向于将共享同一个 SMT 集的空闲处理器，或一开始就可能选择的“上次运行所用”处理器，作为不可用的理想处理器。由于SMT的实现共享了内核缓存，从缓存的角度来看，这种做法可获得与选择理想处理器或上次所用处理器类似的效果。
  6. 如果经过上述步骤后空闲集依然剩余超过一颗处理器，Windows 会选择编号最小的处理器作为该线程的处理器。

在为线程选择了运行所用的处理器后，该线程会转为待命状态，空闲处理器的PRCB会被更新以指向该线程。如果该处理器为空闲但尚未停止 (halted)，则会发送一个 DPC中断，使得该处理器立即开始处理这个调度操作。每当发起上述调度操作后，还会调用KiCheckForThreadDispatch，它会检测到处理器上已经调度了新的线程，并在可能的情况下立刻进行上下文切换(并向Autoboost 机制通知这个切换，交付待处理的APC)。或者如果没有待处理的线程，则会发送一个 DPC 中断。

1. **在不具备空闲处理器时，为线程选择处理器**

如果线程想运行时不具备空闲处理器，或仅有的空闲处理器已被第一轮修剪(会淘汰掉已休止的空闲处理器)所淘汰，则 Windows 首先会检查是否发生了第二种情况。如果是这种情况，调度器会调用 KiSelectCandidateProcessor，通过内核休止引擎查询最佳候选处理器。内核休止引擎会选择理想节点中编号最大的未休止处理器。如果不存在此类处理器，该引擎会强制修改理想处理器的休止状态，使其不再处于已休止状态。返回调度器后,还会检查候选者是否已经处于空闲状态，如果是，则会选择该处理器以供线程运行，具体做法与上一个场景的最后几步类似。

如果该操作失败，Windows 必须决定是否要抢当前运行中的线程。首先必须选择一个目标处理器。具体选择偏好取决于如下优先级: 该线程的理想处理器、该线程上次运行所用的处理器、当前 NUMA 节点中第一颗可用处理器、另一个NUMA 节点中距离最近的处理器。当然所有这些顺序都会忽略相关性约束。

选择了处理器后，随后需要确定新线程是否可以抢占该处理器上当前运行的线程。这是通过比较两个线程的排名(rank)实现的。排名是一种内部调度编号，代表了基于调度组和其他因素为线程计算而来的相对“权力”。如新线程的排名为0(最高)或低于当前线程的排名，或如果排名相等但新线程的优先级高于当前执行中的线程，那么会进行抢占。当前运行中的线程会被标记为“已抢占”，Windows会向目标进程的队列中插入一个DPC 中断，以便抢占当前运行中的线程，供新线程运行。如果已就绪的线程无法立即运行，则会被放入共享或本地队列(根据相关性约束进行选择)的就绪队列，在队列中等待自己的运行机会。

注意 无论底层场景如何或出于任何可能性，大部分情况下线程都会被放入理想处理器的每颗处理器就绪队列，这样可以保证算法在确定如何为线程的运行选择逻辑处理器的过程中的一致性。

异质调度(big.LITTLE)

内核会假设自己运行在 SMP 系统中。然而某些 ARM 架构的处理器会包含多种不同内核。典型的ARM CPU(例如高通的CPU)会包含一些仅短时间运行的高性能内核(耗电量更大)，以及一些长时间运行的低性能内核(耗电量更小)。这种设计有时候也叫作big.LITTLE。

Windows 10引入了区分这些内核的能力，可以根据内核的尺寸和策略(包括线程的前台状态、线程优先级、线程的预期运行时间)调度线程。当电源管理器初始化时，Widows会调用PopInitializeHeteroProcessors对一组处理器进行初始化(前提是处理器是热添加到系统中的)。该函数可用于模拟异质系统(例如用于测试用途)，为此需要将如下键添加至注册表 HKLM\System\CurrentControlSet\Control\SessionManager\Kernel\KGroups 键下。

1. 通过一个键，使用两位十进制数识别处理器组编号。
2. 每个键会包含一个名为SmallProcessorMask的DWORD值，代需要视作“Small”的处理器掩码。举例来说，如果该值为3(前两位已打开)并且该组共有6 颗处理器，那么就意味着处理器0和处理器1(3=1位或2)为“Small”，另外4颗处理器为“Big”。这个机制基本上等同于相关性掩码。

对于异质系统，可通过全局变量中存储的多种策略选项调整内核。

异质系统（Heterogeneous System）是指在一个计算环境中，具有不同类型的处理器、内存、存储和硬件加速器等资源的组合。这些资源的类型、架构和特性可以有很大差异。与具有相同类型处理器和硬件的同质系统相比，异质系统旨在利用各种类型资源的优势，提高性能、能效和灵活性。

下表是部分全局变量及其含义

|  |  |  |
| --- | --- | --- |
| 变量名 | 含义 | 默认值 |
| KiHeteroSystem | 系统是异质的吗? | False |
| PopHeteroSystem | 系统的异质类型:  None(0)  Simulated (1)  EfficiencyClass(2)  FavoredCore(3) | None (0) |
| PpmHeteroPolicy | 调度策略:  None (0)  Manual(1)  SmallOnly (2)  LargeOnly(3)  Dynamic (4) | Dynamic (4) |
| KiDynamicHeteroCpuPolicyMask | 决定了在评估一个线程是否重要时的考虑因素 | 7(前台状态=1，优先级=2，预期运行时间=4) |
| KiDefaultDynamicHeteroCpuPolicy | 动态异质策略(见前文)行为:  A11(0)(全部可用)  Large(1)  LargeOrIdle(2)  Small (3)  SmallOrIdle(4)  Dynamic(5)(用优先级和其他指标决定)  BiasedSmall(6)(用优先级和其他指标决定，但倾向于Small)  BiasedLarge(7) | Small (3) |
| KiDynamicHeteroCpuPolicylmportant | 对被认为重要的动态线程(见上一行的可行值)所要采取的策略 | LargeOrldle(2) |
| KiDynamicHeterCpuPolicylmportantShort | 对被认为重要但只短暂运行的动态线程所要采取的策略 | Small(3) |
| KiDynamicCpuPolicyExpectedRuntime | 被认为“繁重”的运行时间值 | 5200ms |
| KiDynamicHeteroCpuPolicylmportantPriority | 如果选择基于优先级的动态策略，那么重要线程的优先级最小值应是多少 | 8 |

动态策略(见上表)必须被转换为基于KiDynamicHeteroPolicyMask和线程状态的重要性数值。这是由 KiConvertDynamicHeteroPolicy 函数负责的，该函数会按顺序检查线程的前台状态，线程相对于KiDynamicHeteroCpuPolicyImportantPriority 的优先级，以及线程的预期运行时间。如果该线程被认为是重要线程(如果运行时间是决定性因素，那么可以允许短间的运行)，则会在调度决策中使用与重要性有关的策略(即上表中的 KiDynamicHeteroCpuPolicyImportantShort 或 KiDynamicHeteroCpuPolicyImportant)。