基于组的调度

Windows 8和Windows Server 2012引入了一种基于组的调度机制，该机制围绕调度组(KSCHEDULING\_GROUP)这一概念而来。调度组会为每颗处理器维护一条策略，多个调度参数,以及一个内核调度控制块(Kernel Scheduling Control Block, KSCB)列表，这些均属于调度组的一部分。从另一面来看，线程会指向自己所属的调度组。如果该指针为 Null，则意味着线程脱离了所有调度组的控制。

调度组涉及以下术语

1. **世代**(generation)。这是用于跟踪CPU用量的时间量。
2. **配额** (quota)。这是一个组的每个世代允许运行的 CPU 用量。超过该配额意味着这个组已经耗尽了自己的所有“预算”，不足该配额意味着这个组没能用完自己的所有“预算”。
3. **权重**(weight)。这是组的相对重要性，其值介于1和9之间，默认值为5。
4. **公平共享调度**(fair-share scheduling)。借助此类调度，如果配额未用尽的线程没有运行，则此时的空闲CPU周期可以分配给已经超出配额的线程。

KSCB结构包含了如下与CPU有关的信息。

1. 这个世代的周期用量。
2. 长期平均周期用量，这样即可通过真正的“拱形”发现激增的线程活动。
3. 诸如硬封顶(hard capping)等控制标志，这意味着就算在已分配的配额基础上还有可用CPU时间，这些时间也无法为线程提供额外的 CPU 时间。
4. 基于标准优先级(仅限0到15这个范围，因为实时线程永远不可能包含到调度组中)的就绪队列。

调度组还维持了一个名为排名(rank)的重要参数，它可以视作整个组中所有线程的调度优先级。0为最高，排名越高，意味着该组有更多 CPU 时间可以使用,但同时额外获得更多CPU 时间的可能性就越低。排名将始终胜过优先级。这意味着对于两个不同排名的线程，将忽略优先级首选数值较小的那个排名。排名相等的线程则会根据优先级进行对比。随着 CPU周期用量的增加，排名也会酌情做出调整。

面对各种调度选择(如KiQuantumEnd)，在决定接下来要调度哪个线程时还需要考虑当前和就绪线程的线程组(如果有的话)。如果存在调度组，那么数值更小的排名会胜出，随后(如果排名相等)则会按照优先级进行选择，最后则会考虑选择优先抵达的线程(如果优先级也相等，则会在量程结束时进行轮询)。

KiQuantumEnd 是 Windows 操作系统内核中的一个内部函数，它在一个线程完成其分配的量程后被调用。当线程用完其量程时，内核会调用 KiQuantumEnd 函数来处理线程的调度。

KiQuantumEnd 的主要职责包括：

1. 检查当前线程是否已完成其任务，如果是，则将其标记为终止状态。
2. 检查当前线程是否还有剩余的时间量子，如果没有，则将其移到相应的就绪队列末尾，等待重新获得处理器时间。
3. 从就绪队列中选择下一个要执行的线程。
4. 如果选定的下一个线程与当前线程不同，则进行上下文切换，将新线程调度到处理器上。

动态公平共享调度

动态公平共享调度(Dynamic Fair Share Scheduling，DFSS)这种机制可用于将CPU时间公平地分配给计算机上运行的不同会话。借此可以防止一个会话因为其中运行了多个高优先级进程而独占CPU。

在系统初始化的最后几步中，随着 Smss.exe 初始化注册表 SOFTWARE 配置单元，进程管理器会在PsBootPhaseComplete 中发起最后的引导后初始化过程并调用PspIsDfssEnabled。系统会从两种CPU 配额机制(DFSS 或遗留机制)中择一使用。

PsBootPhaseComplete是Windows内核函数之一，其作用是在系统启动的最后阶段进行系统初始化，以确保所有的系统服务和驱动程序都已成功加载。该函数还负责将系统状态从引导状态切换到正常运行状态。具体来说，PsBootPhaseComplete会执行一系列初始化操作，例如创建系统进程，初始化内核对象，创建系统线程，初始化系统中断处理程序，以及加载和初始化系统驱动程序等。

PspIsDfssEnabled是另一个Windows内核函数，其作用是检查操作系统是否启用了动态公平共享调度（DFSS）功能。

为了启用 DFSS，需要将两个配额键对应的 EnableCpuQuota 注册表值(这两个注册表键只适用于Windows Server)设置为非零值，一个配额键位于注册表

HKLM\SOFTWARE\Policies\Microsoft\Windows\SessionManager\QuotaSystem 键下，适用于基于策略的设置；另一个配额键位于注册表

HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\QuotaSystem键下，位于系统键下。借此即可确定系统是否支持此功能

如果DFSS被启用，PsCpuFairShareEnabled全局变量将被设置为TRUE，借此所有线程都可属于调度组(但会话0中的进程除外)。随后会调用PspReadDfssConfigurationValues从上述注册表键中读取 DFSS 配置参数，并将其存储在全局变量中。系统会监视这些键。如果键被修改，通知回调会再次调用PspReadDfssConfigurationValues 以更新配置值。

下表列出了这些值及其含义

|  |  |  |  |
| --- | --- | --- | --- |
| 注册表值名称 | 内核变量名称 | 含义 | 默认值 |
| DfssShortTermSharingMS | PsDfssShortTermSharingMS | 在一个世代周期内组排名增加所需的时间 | 30 ms |
| DfssLongTermSharingMS | PsDfssLongTermSharingMS | 在一个世代周期内，当线程超出自己的配额后，从排名0调至非零排名所需的时间 | 15 ms |
| DfssGenerationLengthMS | PsDfssGenerationLengthMS | 用于跟踪CPU用量的时代时间 | 600 ms |
| DfssLongTermFraction1024 | PsDfssLongTermFraction1024 | 在长期周期计算过程中，指数移动平均值计算公式中所用的值 | 512 |

启用 DFSS后，一旦新建了会话 (会话0除外)，MiSessionObjectCreate 会以默认权重5来分配需要关联给该会话的调度组，而 5这个权重恰好为最小值1和最大值9的中间值。调度组会根据策略结构(KSCHEDULING\_GROUP\_POLICY)管理DFSS或CPU速率控制信息，该策略结构也是调度组的一部分。其中的 Type 成员决定了该策略是为DFSS(WeightBased=0)还是为速率控制(RateControl=1)配置的。随后MiSessionObjectCreate 会调用 KeInsertSchedulingGroup，借此将调度组插入全局系统列表(维护于全局变量 KiSchedulingGroupList 中，对于热添加的处理器，需要借此重新计算权重)。最终这会导致调度组也同时指向特定会话的SESSION\_OBJECT结构

CPU速率限制

我们可对作业施加多种限制，CPU速率控制(rate control)就是其中之一，为此可调用 SetInformationJobObject 并使用JobObjectCpuRateControlInformation作为作业的信息类，使用JOBOBJECT\_CPU\_RATE\_CONTROL\_INFORMATION这种结构类型来包含实际的控制数据。该结构包含的一系列标志可供我们为 CPU 时间限制应用下列3种之一的设置。

1. **CPU速率** 该值可以介于1和 10000之间，代表百分率与100的乘积(例如如果要设置40%的限制，值应该设置为 4000)。
2. **基于权重** 该值可以介于 1和9之间，代表相对于其他作业的权重。(DFSS 实际配置的就是这个设置。)
3. **最小和最大CPU速率** 这些值的用途与第一个选项类似。当作业中的线程达到了测量间隔(默认为600ms)内的最大百分率时，在下一个间隔开始前，将无法继续获得更多CPU 时间。可以使用控制标志指定是否通过硬上限强制设置限制，哪怕目前还有多余的CPU时间。

设置这些限制的最终结果是将一个作业内所有进程中的所有线程放入一个新的调度组中，并按需对整个组进行配置。

处理器的动态添加和替换

现在的服务器主板和系统已经可以支持在计算机运行时添加或更换CPU，计算机中的ACPI BIOS 和相关硬件在构建时就已设计为可允许并感知此类需求，但还需要操作系统的支持才能完整实现这种做法。

动态处理器支持是通过HAL(硬件抽象层, Hardware Abstraction Layer)提供的，HAL可以借助 KeStartDynamicProcessor函数通知内核系统中新增了处理器。该例程的作用类似于系统在启动时检测到有超过一颗处理器，需要对相关结构进行初始化时所做的工作。在添加了动态处理器后，会有大量系统组件执行一些额外工作。例如内存管理器分配针对该CPU 优化过的新内存页和内存结构。当内核对全局描述符表(Global Descriptor Table，GDT)、中断分发表(IDT)、处理器控制区域(PCR)、处理器控制块 (Processor Control Block，PRCB)，以及与处理器有关的其他结构进行初始化时，同时会初始化一个新的DPC 内核栈。

此外还会调用内核中的其他执行体部件，这主要是为了针对新增加的处理器初始化每颗处理器旁视(look-aside)列表。例如I/O管理器、执行体旁视列表代码、缓存管理器以及对象管理器，都会为自己频繁分配的结构使用每颗处理器旁视列表。

最后，内核对处理器的线程化 DPC 支持进行初始化，并调整已导出的内核变量，借此报告新处理器。根据处理器数量计算而来的各种内存管理器掩码和进程种子也会酌情更新，同时还会对新处理器的功能进行更新以便与系统的其他部分保持匹配，例如为新添加的处理器启用虚拟化支持。这个初始化过程的最后一步会通知 Windows 硬件错误架构(Windows Hardware Error Architecture，WHEA)新处理器已经上线。

这一过程也有 HAL的参与。当内核感知到动态处理器后，会调用一次HAL；当内核完成该处理器的初始化工作后，会再次调用一次 HAL。

为了处理驱动程序，系统使用了一个默认的执行体回调对象ProcessorAdd，驱动程序可以借此注册通知。与通知驱动程序电源状态或系统时间变化的回调类似，该回调可以让驱动程序代码执行所需工作，例如在需要时创建新的工作者线程，以便同时处理更多工作

驱动程序获得通知后，最后需要调用即插即用管理器这个内核组件，借此将新处理器添加到系统的设备节点中，并对中断进行重新平衡，这样新处理器就可以注册到其他处理器的中断了。随后，CPU密集型应用程序也将能充分利用新增的处理器。

然而相关性的突然变化可能会对运行中的应用程序产生潜在的不利影响(尤其是在从单处理器环境变为多处理器环境时)，此时可能会出现潜在的竞争条件，或导致工作的错误分配(因为进程可能在启动时就已经根据当时所感知到的 CPU 数量计算好了最优比例)。因此应用程序默认将无法从动态添加的处理器中获益，必须主动请求新处理器。

Windows的SetProcessAffinityUpdateMode和QueryProcessAffinityUpdateMode 这两API可以(使用未文档化的NtSet/QueryInformationProcess 系统调用)告诉进程管理器这些应用程序的相关性需要更新(为此需要在 EPROCESS 中设置AffinityUpdateEnable标志)；或告诉进程管理器，自己并不想处理相关性更新(为此需要在 EPROCESS 中设置AffinityPermanent 标志)。这是一种一次性变更。在告知系统后，应用程序的相关性就无法再次更改，不能随后再改变主意并重新请求更新相关性。

作为 KeStartDynamicProcessor 的一部分，中断重新平衡后还增加了一个新步骤:调用处理器管理器，通过PsUpdateActiveProcessAffinity 执行相关性更新。一些 Windows核心进程和服务已经启用了相关性更新，但第三方软件可能需要重新编译才能从新 API调用中获益。System进程、Svchost 进程和Smss 均已通过编译能够支持动态添加的处理器