内核调度

2.6.23 内核中包含了一个重要的变化，用 CFS 替代了以前的调度器。CFS 被合并到 mainline 之前，关于内核调度器还有一个重要的 patch: RSDL。最终 2.6.23 决定将 CFS 合并到 mainline 而放弃了 RSDL。为什么要引入新的调度器，CFS 和 RSDL 有什么联系和区别？

引言

进程调度是操作系统的核心功能。调度器只是是调度过程中的一部分，进程调度是非常复杂的过程，需要多个系统协同工作完成。本文所关注的仅为调度器，它的主要工作是在所有 RUNNING 进程中选择最合适的一个。作为一个通用操作系统，Linux 调度器将进程分为三类:

交互式进程

此类进程有大量的人机交互，因此进程不断地处于睡眠状态，等待用户输入。典型的应用比如编辑器 vi。此类进程对系统响应时间要求比较高，否则用户会感觉系统反应迟缓。

批处理进程

此类进程不需要人机交互，在后台运行，需要占用大量的系统资源。但是能够忍受响应延迟。比如编译器。

实时进程

实时对调度延迟的要求最高，这些进程往往执行非常重要的操作，要求立即响应并执行。比如视频播放软件或飞机飞行控制系统，很明显这类程序不能容忍长时间的调度延迟，轻则影响电影放映效果，重则机毁人亡。根据进程的不同分类 Linux 采用不同的调度策略。对于实时进程，采用 FIFO 或者 Round Robin 的调度策略。对于普通进程，则需要区分交互式和批处理式的不同。传统 Linux 调度器提高交互式应用的优先级，使得它们能更快地被调度。而 CFS 和 RSDL 等新的调度器的核心思想是“完全公平”。这个设计理念不仅大大简化了调度器的代码复杂度，还对各种调度需求的提供了更完美的支持。在探讨 CFS 和 RSDL 之前，我们首先回顾一下 Linux2.4 和 Linux2.6.0 中所使用的调度器。内核调度器的简单历史

2.1 Linux2.4 的调度器

Linux2.4.18 中使用的调度器采用基于优先级的设计，这个调度器和 Linus 在 1992 年发布的调度器没有大的区别。该调度器的 pick next 算法非常简单：对 runqueue 中所有进程的优先级进行依次进行比较，选择最高优先级的进程作为下一个被调度的进程。(Runqueue 是 Linux 内核中保存所有就绪进程的队列) 。术语 pick next 用来指从所有候选进程中挑选下一个要被调度的进程的过程。每个进程被创建时都被赋予一个时间片。时钟中断递减当前运行进程的时间片，当进程的时间片被用完时，它必须等待重新赋予时间片才能有机会运行。 Linux2.4 调度器保证只有当所有 RUNNING 进程的时间片都被用完之后，才对所有进程重新分配时间片。这段时间被称为一个 epoch。这种设计保证了每个进程都有机会得到执行。各种进程对调度的需求并不相同，Linux2.4 调度器主要依靠改变进程的优先级，来满足不同进程的调度需求。事实上，所有后来的调度器都主要依赖修改进程优先级来满足不同的调度需求。

实时进程

实时进程的优先级是静态设定的，而且始终大于普通进程的优先级。因此只有当 runqueue 中没有实时进程的情况下，普通进程才能够获得调度。

实时进程采用两种调度策略：SCHED_FIFO 和 SCHED_RR。FIFO 采用先进先出的策略，对于所有相同优先级的进程，最先进入 runqueue 的进程总能优先获得调度；Round Robin 采用更加公平的轮转策略，使得相同优先级的实时进程能够轮流获得调度。

普通进程

对于普通进程，调度器倾向于提高交互式进程的优先级，因为它们需要快速的用户响应。普通进程的优先级主要由进程描述符中的 Counter 字段决定 (还要加上 nice 设定的静态优先级) 。进程被创建时子进程的 counter 值为父进程 counter 值的一半，这样保证了任何进程不能依靠不断地 fork() 子进程从而获得更多的执行机会。
Linux2.4 调度器是如何提高交互式进程的优先级的呢？如前所述，当所有 RUNNING 进程的时间片被用完之后，调度器将重新计算所有进程的 counter 值，所有进程不仅包括 RUNNING 进程，也包括处于睡眠状态的进程。处于睡眠状态的进程的 counter 本来就没有用完，在重新计算时，他们的 counter 值会加上这些原来未用完的部分，

从而提高了它们的优先级。交互式进程经常因等待用户输入而处于睡眠状态，当它们重新被唤醒并进入 runqueue 时，就会优先于其它进程而获得 CPU。从用户角度来看，交互式进程的响应速度就提高了。

该调度器的主要缺点:

可扩展性不好：调度器选择进程时需要遍历整个 runqueue 从中选出最佳人选，因此该算法的执行时间与进程数成正比。另外每次重新计算 counter 所花费的时间也会随着系统中进程数的增加而线性增长，当进程数很大时，更新 counter 操作的代价会非常高，导致系统整体的性能下降。

高负载系统上的调度性能比较低：2.4 的调度器预分配给每个进程的时间片比较大，因此在高负载的服务器上，该调度器的效率比较低，因为平均每个进程的等待时间于该时间片的大小成正比。

交互式进程的优化并不完善：Linux2.4 识别交互式进程的原理基于以下假设，即交互式进程比批处理进程更频繁地处于 SUSPENDED 状态。然而现实情况往往并非如此，有些批处理进程虽然没有用户交互，但是也会频繁地进行 IO 操作，比如一个数据库引擎在处理查询时会经常地进行磁盘 IO，虽然它们并不需要快速地用户响应，还是被提高了优先级。当系统中这类进程的负载较重时，会影响真正的交互式进程的响应时间。

对实时进程的支持不够：Linux2.4 内核是非抢占的，当进程处于内核态时不会发生抢占，这对于真正的实时应用是不能接受的。

为了解决这些问题，Ingo Molnar 开发了新的 O(1)调度器，在 CFS 和 RSDL 之前，这个调度器不仅被 Linux2.6 采用，还被 backport 到 Linux2.4 中，很多商业的发行版本都采用了这个调度器。

2.2 Linux2.6 的 O(1)调度器

从名字就可以看出 O(1)调度器主要解决了以前版本中的扩展性问题。O(1)调度算法所花费的时间为常数，与当前系统中的进程个数无关。此外 Linux2.6 内核支持内核态抢

占，因此更好地支持了实时进程。相对于前任，O (1) 调度器还更好地区分了交互式进程和批处理式进程。

Linux2.6 内核也支持三种调度策略。其中 SCHED_FIFO 和 SCHED_RR 用于实时进程，而 SCHED_NORMAL 用于普通进程。O(1)调度器在两个方面修改了 Linux2.4 调度器，一是进程优先级的计算方法；二是 pick next 算法。

2.2.1 进程的优先级计算

普通进程的优先级计算

普通进程优先级是动态计算的，计算公式中包含了静态优先级。一般来讲，静态优先级越高，进程所能分配到的时间片越长，用户可以通过 nice 系统调用修改进程的静态优先级。

动态优先级由 公式一 计算得出：

公式一

dynamic priority = max (100, min (static priority – bonus +5, 139))

其中 bonus 取决于进程的平均睡眠时间。由此可以看出，在 linux2.6 中，一个普通进程的优先级和平均睡眠时间的关系为：平均睡眠时间越长，其 bonus 越大，从而得到更高的优先级。

平均睡眠时间也被用来判断进程是否是一个交互式进程。如果满足下面的公式，进程就被认为是一个交互式进程：

公式二

Dynamic priority ≤ 3 x static priority /4 + 28

平均睡眠时间是进程处于等待睡眠状态下的时间，该值在进程进入睡眠状态时增加，而进入 RUNNING 状态后则减少。该值的更新时机分布在很多内核函数内：时钟中断 scheduler_tick ()；进程创建；进程从 TASK_INTERRUPTIBLE 状态唤醒；负载平衡等。

实时进程的优先级计算

实时进程的优先级由 sys_sched_setschedule()设置。该值不会动态修改，而且总是比普通进程的优先级高。在进程描述符中用 rt_priority 域表示。

2.2.2 pick next 算法

普通进程的调度选择算法基于进程的优先级，拥有最高优先级的进程被调度器选中。2.4 中，时间片 counter 同时也表示了一个进程的优先级。2.6 中时间片用任务描述符中的 time_slice 域表示，而优先级用 prio（普通进程）或者 rt_priority（实时进程）表示。

调度器为每一个 CPU 维护了两个进程队列数组：active 数组和 expire 数组。数组中的元素着保存某一优先级的进程队列指针。系统一共有 140 个不同的优先级，因此这两个数组大小都是 140。

当需要选择当前最高优先级的进程时，2.6 调度器不用遍历整个 runqueue，而是直接从 active 数组中选择当前最高优先级队列中的第一个进程。假设当前所有进程中最高优先级为 50(换句话说，系统中没有任何进程的优先级小于 50)。则调度器直接读取 active[49]，得到优先级为 50 的进程队列指针。该队列头上的第一个进程就是被选中的进程。这种算法的复杂度为 O(1)，从而解决了 2.4 调度器的扩展性问题。

为了实现上述算法 active 数组维护了一个 bitmap，当某个优先级别上有进程被插入列表时，相应的比特位就被置位。 Sched_find_first_bit()函数查询该 bitmap，返回当前被置位的最高优先级的数组下标。在上例中 sched_find_first_bit 函数将返回 49。在 IA 处理器上可以通过 bsfl 等指令实现。

为了提高交互式进程的响应时间，O(1)调度器不仅动态地提高该类进程的优先级，还采用以下方法:

每次时钟 tick 中断中，进程的时间片(time_slice)被减一。当 time_slice 为 0 时，调度器判断当前进程的类型，如果是交互式进程或者实时进程，则重置其时间片并重新插入 active 数组。如果不是交互式进程则从 active 数组中移到 expired 数组。这样实时进程和交互式进程就总能优先获得 CPU。然而这些进程不能始终留在 active 数组中，否则进入 expire 数组的进程就会产生饥饿现象。当进程已经占用 CPU 时间超过一个固定值后，即使它是实时进程或者交互式进程也会被移到 expire 数组中。

当 active 数组中的所有进程都被移到 expire 数组中后，调度器交换 active 数组和 expire 数组。当进程被移入 expire 数组时，调度器会重置其时间片，因此新的 active 数组又恢复了初始情况，而 expire 数组为空，从而开始新的一轮调度。

2.2.3 O(1)调度器小节

Linux2.6 调度器改进了前任调度器的可扩展性问题，schedule()函数的时间复杂度为 O(1)。这取决于两个改进:

一. Pick next 算法借助于 active 数组，无需遍历 runqueue;

二. 取消了定期更新所有进程 counter 的操作，动态优先级的修改分布在进程切换，时钟 tick 中断以及其它一些内核函数中进行。

O(1)调度器区分交互式进程和批处理进程的算法与以前虽大有改进,但仍然在很多情况下会失效。有一些著名的程序总能让该调度器性能下降，导致交互式进程反应缓慢:

fiftyp.c, thud.c, chew.c, ring-test.c, massive_intr.c

这些不足催生了 Con Kolivas 的楼梯调度算法 SD，以及后来的改进版本 RSDL。Ingo Molnar 在 RSDL 之后开发了 CFS，并最终被 2.6.23 内核采用。接下来我们开始介绍这些新一代调度器。

3 新一代调度器

Linux2.6.0 发布之前，很多人都担心调度器存在的问题将阻碍新版本的发布。它对于交互式应用仍然存在响应性差的问题，对 NUMA 支持也不完善。为了解决这些问题，大量难以维护和阅读的复杂代码被加入 Linux2.6.0 的调度器模块，虽然很多性能问题因此得到了解决，可是另外一个严重问题始终困扰着许多内核开发者。那就是代码的复杂度问题。

Con Kolivas，在 2004 年提出了第一个改进调度器设计的 patch：staircase scheduler。为调度器设计提供了一个新的思路。之后的 RSDL 和 CFS 都基于 SD 的许多基本思想。本章中，我们将简要探讨这三个主要的调度器算法。

3.1 楼梯调度算法 staircase scheduler

楼梯算法(SD)在思路上和 O(1)算法有很大不同，它抛弃了动态优先级的概念。而采用了一种完全公平的思路。前任算法的主要复杂性来自动态优先级的计算，调度器根据平均睡眠时间和一些很难理解的经验公式来修正进程的优先级以及区分交互式进程。这样的代码很难阅读和维护。

楼梯算法思路简单，但是实验证明它对应交互式进程的响应比其前任更好，而且极大地简化了代码。

和 O(1)算法一样，楼梯算法也同样为每一个优先级维护一个进程列表，并将这些列表组织在 active 数组中。当选取下一个被调度进程时，SD 算法也同样从 active 数组中直接读取。

与 O (1)算法不同在于，当进程用完了自己的时间片后，并不是被移到 expire 数组中。而是被加入 active 数组的低一优先级列表中，即将其降低一个级别。不过请注意这里只是将该任务插入低一级优先级任务列表中，任务本身的优先级并没有改变。当时间片再次用完，任务被再次放入更低一级优先级任务队列中。就象一部楼梯，任务每次用完了自己的时间片之后就下一级楼梯。

任务下到最低一级楼梯时，如果时间片再次用完，它会回到初始优先级的下一级任务队列中。比如某进程的优先级为 1，当它到达最后一级台阶 140 后，再次用完时间片时将

回到优先级为 2 的任务队列中，即第二级台阶。不过此时分配给该任务的 time_slice 将变成原来的 2 倍。比如原来该任务的时间片 time_slice 为 10ms，则现在变成了 20ms。基本的原则是，当任务下到楼梯底部时，再次用完时间片就回到上次下楼梯的起点的下一级台阶。并给予该任务相同于其最初分配的时间片。总结如下:

设任务本身优先级为 P，当它从第 N 级台阶开始下楼梯并到达底部后，将回到第 N+1 级台阶。并且赋予该任务 N+1 倍的时间片。

以上描述的是普通进程的调度算法，实时进程还是采用原来的调度策略，即 FIFO 或者 Round Robin。

楼梯算法能避免进程饥饿现象，高优先级的进程会最终和低优先级的进程竞争，使得低优先级进程最终获得执行机会。

对于交互式应用，当进入睡眠状态时，与它同等优先级的其他进程将一步一步地走下楼梯，进入低优先级进程队列。当该交互式进程再次唤醒后，它还留在高处的楼梯台阶上，从而能更快地被调度器选中，加速了响应时间。

楼梯算法的优点

从实现角度看，SD 基本上还是沿用了 O(1)的整体框架，只是删除了 O(1)调度器中动态修改优先级的复杂代码；还淘汰了 expire 数组，从而简化了代码。它最重要的意义在于证明了完全公平这个思想的可行性。

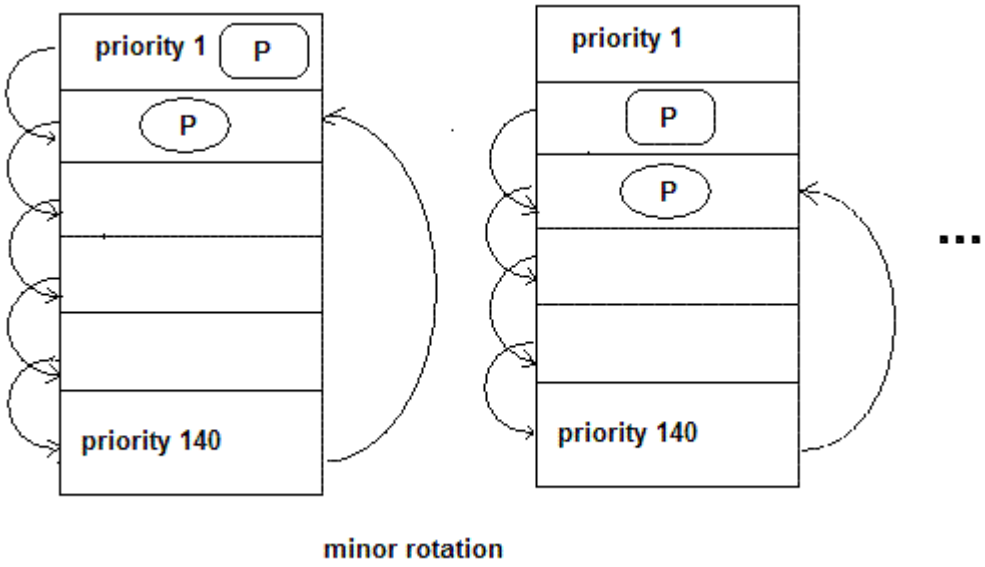
3.2 RSDL（The Rotating Staircase Deadline Schedule）

RSDL 也是由 Con Kolivas 开发的，它是对 SD 算法的改进。核心的思想还是“完全公平”。没有复杂的动态优先级调整策略。

RSDL 重新引入了 expire 数组。它为每一个优先级都分配了一个 “组时间配额”， 我们将组时间配额标记为 Tg；同一优先级的每个进程都拥有同样的"优先级时间配额"本文中用 Tp 表示，以便于后续描述。

当进程用完了自身的 Tp 时，就下降到下一优先级进程组中。这个过程和 SD 相同，在 RSDL 中这个过程叫做 minor rotation。请注意 Tp 不等于进程的时间片，而是小于进程的时间片。下图表示了 minor rotation。进程从 priority1 的队列中一步一步下到 priority140 之后回到 priority2 的队列中，这个过程如下图左边所示，然后从 priority 2 开始再次一步一步下楼，到底后再次反弹到 priority3 队列中，如图 1 所示。

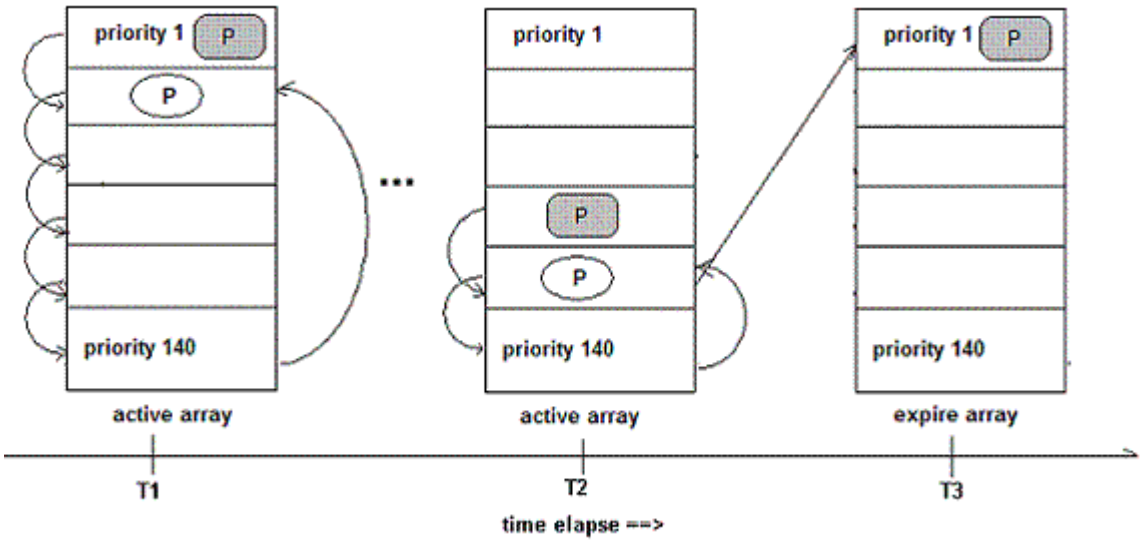
图 1.



在 SD 算法中，处于楼梯底部的低优先级进程必须等待所有的高优先级进程执行完才能获得 CPU。因此低优先级进程的等待时间无法确定。RSDL 中，当高优先级进程组用完了它们的 Tg(即组时间配额)时，无论该组中是否还有进程 Tp 尚未用完，所有属于该组的进程都被强制降低到下一优先级进程组中。这样低优先级任务就可以在一个可以预计的未来得到调度。从而改善了调度的公平性。这就是 RSDL 中 Deadline 代表的含义。

进程用完了自己的时间片 time_slice 时（下图中 T2），将放入 expire 数组中它初始的优先级队列中(priority 1)。

图 2



当 active 数组为空,或者所有的进程都降低到最低优先级时就会触发 major rotation: 。Major rotation 交换 active 数组和 expire 数组，所有进程都恢复到初始状态，再一次从新开始 minor rotation 的过程。

RSDL 对交互式进程的支持

和 SD 同样的道理，交互式进程在睡眠时间时，它所有的竞争者都因为 minor rotation 而降到了低优先级进程队列中。当它重新进入 RUNNING 状态时，就获得了相对较高的优先级，从而能被迅速响应。

3.3 CFS 完全公平调度器

CFS 是最终被内核采纳的调度器。它从 RSDL/SD 中吸取了完全公平的思想，不再跟踪进程的睡眠时间，也不再企图区分交互式进程。它将所有的进程都统一对待，这就是公平的含义。CFS 的算法和实现都相当简单，众多的测试表明其性能也非常优越。

按照作者 Ingo Molnar 的说法: "CFS 百分之八十的工作可以用一句话概括: CFS 在真实的硬件上模拟了完全理想的多任务处理器"。在“完全理想的多任务处理器 “下，每个进程都能同时获得 CPU 的执行时间。当系统中有两个进程时，CPU 的计算时间被分成

两份，每个进程获得 50%。然而在实际的硬件上，当一个进程占用 CPU 时，其它进程就必须等待。这就产生了不公平。

假设 runqueue 中有 n 个进程，当前进程运行了 10ms。在“完全理想的多任务处理器”中，10ms 应该平分给 n 个进程(不考虑各个进程的 nice 值)，因此当前进程应得的时间是(10/n)ms，但是它却运行了 10ms。所以 CFS 将惩罚当前进程，使其它进程能够在下次调度时尽可能取代当前进程。最终实现所有进程的公平调度。下面将介绍 CFS 实现的一些重要部分，以便深入地理解 CFS 的工作原理。

CFS 如何实现 pick next

CFS 抛弃了 active/expire 数组，而使用红黑树选取下一个被调度进程。所有状态为 RUNABLE 的进程都被插入红黑树。在每个调度点，CFS 调度器都会选择红黑树的最左边的叶子节点作为下一个将获得 cpu 的进程。

tick 中断

在 CFS 中，tick 中断首先更新调度信息。然后调整当前进程在红黑树中的位置。调整完成后如果发现当前进程不再是最左边的叶子，就标记 need_resched 标志，中断返回时就会调用 scheduler()完成进程切换。否则当前进程继续占用 CPU。从这里可以看到 CFS 抛弃了传统的时间片概念。Tick 中断只需更新红黑树，以前的所有调度器都在 tick 中断中递减时间片，当时间片或者配额被用完时才触发优先级调整并重新调度。

红黑树键值计算

理解 CFS 的关键就是了解红黑树键值的计算方法。该键值由三个因子计算而得：一是进程已经占用的 CPU 时间；二是当前进程的 nice 值；三是当前的 cpu 负载。

进程已经占用的 CPU 时间对键值的影响最大，其实很大程度上我们在理解 CFS 时可以简单地认为键值就等于进程已占用的 CPU 时间。因此该值越大，键值越大，从而使得当前进程向红黑树的右侧移动。另外 CFS 规定，nice 值为 1 的进程比 nice 值为 0 的进程多获得 10%的 CPU 时间。在计算键值时也考虑到这个因素，因此 nice 值越大，键值也越大。

CFS 为每个进程都维护两个重要变量：fair_clock 和 wait_runtime。在本文中，我们将为每个进程维护的变量称为进程级变量，为每个 CPU 维护的称作 CPU 级变量，为每个 runqueue 维护的称为 runqueue 级变量。

进程插入红黑树的键值即为 *fair_clock – wait_runtime*。

fair_clock 从其字面含义上讲就是一个进程应获得的 CPU 时间，即等于进程已占用的 CPU 时间除以当前 runqueue 中的进程总数；wait_runtime 是进程的等待时间。它们的差值代表了一个进程的公平程度。该值越大，代表当前进程相对于其它进程越不公平。

对于交互式任务，wait_runtime 长时间得不到更新，因此它能拥有更高的红黑树键值，更靠近红黑树的左边。从而得到快速响应。

红黑树是平衡树，调度器每次总最左边读出一个叶子节点，该读取操作的时间复杂度是 O(LgN)。

调度器管理器

为了支持实时进程，CFS 提供了调度器模块管理器。各种不同的调度器算法都可以作为一个模块注册到该管理器中。不同的进程可以选择使用不同的调度器模块。 2.6.23 中，CFS 实现了两个调度算法，CFS 算法模块和实时调度模块。对应实时进程，将使用实时调度模块。对应普通进程则使用 CFS 算法。Ingo Molnar 还邀请 Con Kolivas 可以将 RSDL/SD 写成一个调度算法模块。

CFS 源代码分析

Sched.c 中 scheduler_tick()函数会被时钟中断直接调用。它首先更新 runqueue 级变量 clock；然后调用 CFS 的 tick 处理函数 task_tick_fair()。task_tick_fair 在 sched_fair.c 中。它主要的工作是调用 entity_tick()

函数 entiry_tick 源代码如下：

```
static void entity_tick(struct cfs_rq *cfs_rq, struct sched_entity *curr)
{
        struct sched_entity *next;
        dequeue_entity(cfs_rq, curr, 0);
        enqueue_entity(cfs_rq, curr, 0);
        next = __pick_next_entity(cfs_rq);
        if (next == curr)
                return;
        __check_preempt_curr_fair(cfs_rq, next, curr,
                                  sched_granularity(cfs_rq));
}
```

首先调用 dequeue_entity ()函数将当前进程从红黑树中删除，再调用 enqueue_entity（）重新插入。这两个动作就调整了当前进程在红黑树中的位置。 _pick_next_entity() 返回红黑树中最左边的节点，如果不再是当前进程，就调用 _check_preempt_curr_fair。该函数设置调度标志，当中断返回时就会调用 schedule() 进行调度。

函数 enqueue_entity()的源码如下:

```
enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int wakeup)
{
        /*
         * Update the fair clock.
         */
```

```
        update_curr(cfs_rq);
        if (wakeup)
                enqueue_sleeper(cfs_rq, se);
        update_stats_enqueue(cfs_rq, se);
        __enqueue_entity(cfs_rq, se);
}
```

它的第一个工作是更新调度信息。然后将进程插入红黑树中。其中 update_curr()函数是核心。完成调度信息的更新。

```
static void update_curr(struct cfs_rq *cfs_rq)
{
        struct sched_entity *curr = cfs_rq_curr(cfs_rq);
        unsigned long delta_exec;
        if (unlikely(!curr))
                return;
        delta_exec = (unsigned long)(rq_of(cfs_rq)->clock - curr->exec_start);
        curr->delta_exec += delta_exec;
        if (unlikely(curr->delta_exec > sysctl_sched_stat_granularity)) {
                __update_curr(cfs_rq, curr);
                curr->delta_exec = 0;
        }
        curr->exec_start = rq_of(cfs_rq)->clock;
}
```

该函数首先统计当前进程所获得的 CPU 时间，rq_of(cfs_rq)->clock 值在 tick 中断中被更新，curr->exec_start 就是当前进程开始获得 CPU 时的时间戳。两值相减就是当前进程所获得的 CPU 时间。将该变量存入 curr->delta_exec 中。然后调用 __update_curr()

```
__update_curr(struct cfs_rq *cfs_rq, struct sched_entity *curr)
{
        unsigned long delta, delta_exec, delta_fair, delta_mine;
        struct load_weight *lw = &cfs_rq->load;
        unsigned long load = lw->weight;
        delta_exec = curr->delta_exec;
        schedstat_set(curr->exec_max, max((u64)delta_exec,
curr->exec_max));
        curr->sum_exec_runtime += delta_exec;
        cfs_rq->exec_clock += delta_exec;
        if (unlikely(!load))                    return;
        delta_fair = calc_delta_fair(delta_exec, lw);
        delta_mine = calc_delta_mine(delta_exec, curr->load.weight, lw);
        if (cfs_rq->sleeper_bonus > sysctl_sched_min_granularity) {
                delta = min((u64)delta_mine, cfs_rq->sleeper_bonus);
                delta = min(delta, (unsigned long)(
                        (long)sysctl_sched_runtime_limit -
curr->wait_runtime));
                cfs_rq->sleeper_bonus -= delta;
                delta_mine -= delta;
        }
        cfs_rq->fair_clock += delta_fair;
```

```
        add_wait_runtime(cfs_rq, curr, delta_mine - delta_exec);
}
```

__update_curr()的主要工作就是更新前面提到的 fair_clock 和 wait_runtime。这两个值的差值就是后面进程插入红黑树的键值。变量 Delta_exec 保存了前面获得的当前进程所占用的 CPU 时间。函数 calc_delta_fair()根据 cpu 负载（保存在 lw 变量中），对 delta_exec 进行修正，然后将结果保存到 delta_fair 变量中，随后将 fair_clock 增加 delta_fair。函数 calc_delta_mine()根据 nice 值（保存在 curr->load.weight 中）和 cpu 负载修正 delta_exec，将结果保存在 delta_mine 中。根据源代码中的注释，delta_mine 就表示当前进程应该获得的 CPU 时间。

随后将 delta_fair 加给 fair_clock 而将 delta_mine-delta_exec 加给 wait_runtime。函数 add_wait_runtime 中两次将 wait_runtime 减去 delta_mine-delta_exec。由于 calc_delt_xx()函数对 delta_exec 仅做了较小的修改，为了讨论方便，我们可以忽略它们对 delta_exec 的修改。最终的结果可以近似看成 fair_clock 增加了一倍的 delta_exec，而 wait_runtime 减小了两倍的 delta_exec。因此键值 fair_clock-wait_runtime 最终增加了一倍的 delta_exec 值。键值增加，使得当前进程再次插入红黑树中就向右移动了。

CFS 小结

以上的讨论看出 CFS 对以前的调度器进行了很大改动。用红黑树代替优先级数组；用完全公平的策略代替动态优先级策略；引入了模块管理器；它修改了原来 Linux2.6.0 调度器模块 70%的代码。结构更简单灵活，算法适应性更高。相比于 RSDL，虽然都基于完全公平的原理，但是它们的实现完全不同。相比之下，CFS 更加清晰简单，有更好的扩展性。

CFS 还有一个重要特点，即调度粒度小。CFS 之前的调度器中，除了进程调用了某些阻塞函数而主动参与调度之外，每个进程都只有在用完了时间片或者属于自己的时间配

额之后才被抢占。而 CFS 则在每次 tick 都进行检查，如果当前进程不再处于红黑树的左边，就被抢占。在高负载的服务器上，通过调整调度粒度能够获得更好的调度性能。

4 总结

通过对 Linux 调度器历史发展的探讨，能进一步了解 CFS 调度器开发的背景知识。其实目前任何调度器算法都还无法满足所有应用的需要，CFS 也有一些负面的测试报告。我们相信随着 Linux 的不断发展，还会出现新的调度算法，让我们拭目以待。有抱负的程序员也可以尝试着在这个领域为 Linux 作出贡献。