

第七章 进程调度

Linux与任何分时系统一样，通过一个进程到另一个进程的快速切换，达到表面上看来多个进程同时执行的神奇效果。进程切换本身已在第三章中讨论过，本章讨论进程调度，主要关心什么时候进行进程切换及选择哪一个进程来运行。

本章由三部分组成。“调度策略”一节从理论上介绍Linux进行进程调度所做的选择，“调度算法”一节讨论实现调度所采用的数据结构和相应的算法，最后，“与调度相关的系统调用”一节描述了影响进程调度的系统调用。

为了叙述起来更简单，我们仍以80x86体系结构为例，尤其是，我们假定系统采用统一内存访问模型（Uniform Memory Access），而且系统时钟设定为1ms。

调度策略

传统Unix操作系统的调度算法必须实现几个互相冲突的目标：进程响应时间尽可能快，后台作业的吞吐量尽可能高，尽可能避免进程的饥饿现象，低优先级和高优先级进程的需要尽可能调和等等。决定什么时候以怎样的方式选择一个新进程运行的这组规则就是所谓的调度策略（scheduling policy）。

Linux的调度基于分时技术（time-sharing）：多个进程以“时间多路复用”方式运行，因为CPU的时间被分成“片”，给每个可运行进程分配一片（注1）。当然，单处理器在任何给定的时刻只能运行一个进程。如果当前运行进程的时间片或时限（quantum）到期时，该进程还没有运行完毕，进程切换就可以发生。分时依赖于定时中断，因此，对进程是透明的。不需要在程序中插入额外的代码来保证CPU分时。

调度策略也是根据进程的优先级对它们进行分类。有时用复杂的算法求出进程当前的优先级，但最后¹的结果是相同的：每个进程都与一个值相关联，这个值表示把进程如何适当地分配给CPU。

在Linux中，进程的优先级是动态的。调度程序跟踪进程正在做什么，并周期性地调整它们的优先级。在这种方式下，在较长的时间间隔内没有使用CPU的进程，通过动态地增加它们的优先级来提升它们。相应地，对于已经在CPU上运行了较长时间的进程，通过减少它们的优先级来处罚它们。

当谈及有关调度问题时，传统上把进程分类为“I/O受限（I/O-bound）”或“CPU受限（CPU-bound）”。前者频繁地使用I/O设备，并花费很多时间等待I/O操作的完成；而后者是需要大量CPU时间的数值计算应用程序。

另一种分类法把进程区分为三类：

¹ 注1：调度算法不会选择已被停止和挂起的进程在CPU上运行。

交互式进程 (Interactive process)

这些进程经常与用户进行交互，因此，要花很多时间等待键盘和鼠标操作。当接受了输入后，进程必须被很快唤醒，否则，用户将发现系统反应迟钝。典型的情况是，平均延迟必须低于50到150ms。这样的延迟变化也必须进行限制，否则，用户将发现系统是不稳定的。典型的交互式程序是命令shell、文本编辑程序及图形应用程序。

批处理进程 (Batch process)

这些进程不必与用户交互，因此，它们经常在后台运行。因为这样的进程不必被很快地响应，因此，它们常受到调度程序的慢待。典型的批处理进程是程序设计语言的编译程序、数据库搜索引擎及科学计算。

实时进程 (Real-time process)

这些进程有很强的调度需要。这样的进程决不会被低优先级的进程阻塞，它们应该有一个短的响应时间，更重要的是，响应时间的变化应该很小。典型的实时程序有视频和音频应用程序、机器人控制程序及从物理传感器上收集数据的程序。

我们刚刚提到的两种分类法在一定程度上相互独立。例如，一个批处理进程可能是I/O受限型的（如数据库服务器），或是CPU受限型的（如图象着色程序）。在Linux中，调度算法可以明确地确认所有实时程序的身份，但没有办法区分交互式程序和批处理程序。Linux 2.6调度程序实现了基于进程过去行为的启发式算法，以确定进程应该被当作交互式进程还是批处理进程。当然，与批处理进程相比，调度程序有偏爱交互进程的倾向。

程序员可以通过表7-1所列的系统调用改变调度优先级。更详细的内容将在“与调度相关的系统调用”一节中给出。

表 7-1 与调度相关的系统调用

系统调用	说明
nice()	改变一个普通进程的静态优先级
getpriority()	获得一组普通进程的最大优先级
setpriority()	设置一组普通进程的静态优先级
sched_getscheduler()	获得一个进程的调度策略
sched_setscheduler()	设置一个进程的调度策略和实时优先级
sched_getparam()	获得一个进程的实时优先级
sched_setparam()	设置一个进程的实时优先级
sched_yield()	自愿放弃处理器而不阻塞
sched_get_priority_min()	获得一种策略的最小实时优先级
sched_get_priority_max()	获得一种策略的最大实时优先级
sched_rr_get_interval()	获得时间片轮转策略的时间片值

`sched_setaffinity()` 设置进程的CPU亲和力掩码
`sched_getaffinity()` 获得进程的CPU亲和力掩码

进程的抢占

如第一章所述，Linux的进程是抢占式的。如果进程进入TASK_RUNNING状态，内核检查它的动态优先级是否大于当前正运行进程的优先级。如果是，`current`的执行被中断，并调用调度程序选择另一个进程运行（通常是刚刚变为可运行的进程）。当然，进程在它的时间片到期时也可以被抢占。此时，当前进程`thread_info`结构中的TIF_NEED_RESCHED标志被设置，以便定时中断处理程序终止时调度程序被调用。

例如，让我们考虑一种情况，在这种情况下，只有两个程序——一个文本编辑程序和一个编译程序正在执行。文本编辑程序是一个交互式程序，因此，它的动态优先级高于编译程序。不过，因为编辑程序交替于暂停思考与数据输入之间，因此，它经常被挂起；此外，两次击键之间的平均延迟相对较长。然而，只要用户一按键，中断就发生，内核唤醒文本编辑进程。内核也确定编辑进程的动态优先级确实是高于`current`的优先级（当前正运行的进程，即编译进程），因此，编辑进程的TIF_NEED_RESCHED标志被设置，以此来强迫内核处理完中断时激活调度程序。调度程序选择编辑进程并执行进程切换；结果，编辑进程很快恢复执行，并把用户敲的字符回显在屏幕上。当处理完字符时，文本编辑进程自己挂起等待下一次击键，编译进程恢复执行。

注意被抢占的进程并没有被挂起，因为它还处于TASK_RUNNING状态，只不过不再使用CPU。此外，记住，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，我们曾在第5章“内核抢占”一节深入讨论过这个特征。

一个时间片²必须持续多长？

时间片的长短对系统性能是很关键的：它既不能太长也不能太短。

如果平均时间片太短，由进程切换引起的系统额外开销就变得非常高。例如，假定进程切换需要5ms，如果时间片也是5ms，那么，CPU至少把50%的时间花费在进程切换上（注2³）。

如果平均时间片太长，进程看起来就不再是并发执行。例如，让我们假定把时间片设置为5秒，那么，每个可运行进程运行大约5秒，但是暂停的时间更长（典型的是5秒乘以可运行进程的个数）。

通常认为长的时间片降低交互式应用程序的响应时间，但这往往是错误的。正如本章前面“

² 注2：实际上，情况可能更糟糕。例如，在进程的时间片中还要计算进程切换所需的时间，那么所有的CPU时间会花费在进程切换，就没有进程能执行完。

进程的抢占”一节中所描述的那样，交互式进程相对有较高的优先级，因此，不管时间片是多长，它们都会很快地抢占批处理进程。

在一些情况下，一个太长的时间片会降低系统的响应能力。例如，假定两个用户在各自的 `shell` 提示符下分别输入两条命令，其中一条启动一个 CPU 受限型的进程，而另一条启动一个交互式应用。两个 `shell` 都创建一个新进程，并把用户命令的执行委托给新进程。此外，又假定这样的新进程最初有相同的优先级（Linux 预先并不知道执行进程是批处理的还是交互式的）。现在，如果调度程序选择 CPU 受限型的进程执行，那么，另一个进程开始执行前就可能要等待一个时间片。因此，如果这样的时间片较长，那么，看起来系统就可能对用户的请求反应迟钝。

时间片大小的选择总是一种折衷。Linux 采取单凭经验的方法，即选择尽可能长、同时能保持良好响应时间的一个时间片。

调度算法

早期 Linux 版本中的调度算法非常简单易懂：在每次进程切换时，内核扫描可运行进程的链表，计算进程的优先权，然后选择“最佳”进程来运行。这个算法的主要缺点是选择“最佳”进程所要消耗的时间与可运行的进程数量相关，因此，这个算法的开销太大，在运行数千个进程的高端系统中，要消耗太多的时间。

Linux 2.6 的调度算法就复杂多了。通过设计，该算法较好地解决了与可运行进程数量的比例关系，因为它在固定的时间内（与可运行的进程数量无关）选中要运行的进程。它也很好地处理了与处理器数量的比例关系，因为每个 CPU 都拥有自己的可运行进程队列。而且，新算法较好地解决了区分交互式进程和批处理进程的问题。因此，在高负载的系统中，用户感到在 Linux 2.6 中交互应用的响应速度比早期的 Linux 版本要快。

调度程序总能成功地找到要执行的进程，事实上，总是至少有一个可运行进程：即 `swapper` 进程，它的 PID 等于 0，而且它只有在 CPU 不能执行其他进程时才执行。就象在第三章中提到的，每个多处理器系统的 CPU 都有它自己的 `swapper` 进程，其 PID 等于 0。

每个 Linux 进程总是按照下面的调度类型被调度：

SCHED_FIFO

先进先出的实时进程。当调度程序把 CPU 分配给进程的时候，它把该进程描述符保留在运行队列链表的当前位置。如果没有其他可运行的高优先权实时进程，进程就继续使用 CPU，想用多久就用多久，即使还有其他具有相同优先权的实时进程处于可运行状态。

SCHED_RR

时间片轮转的实时进程。当调度程序把 CPU 分配给进程的时候，把该进程的描述符放在运行队列链表的末尾。这种策略保证对所有具有相同优先权的 SCHED_RR 实时进程公平地分配 CPU 时间。

SCHED_NORMAL

普通的分时进程

调度算法根据进程是普通进程还是实时进程而有所不同。

普通进程的调度

每个普通进程都有它自己的静态优先权，调度程序使用静态优先权来估价系统中这个进程与其他普通进程之间调度的程度。内核用从 100（最高优先权）到 139（最低优先权）的数表示普通进程的静态优先权。注意，值越大静态优先权越低。

新进程总是继承其父进程的静态优先权。不过，通过把某些“让权值”传递给系统调用 `nice()` 和 `setpriority()`（参见本章稍后”与调度相关的系统调用），用户可以改变自己拥有的进程的静态优先权。

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：

$$\text{Base time quantum} = \begin{cases} (140 - \text{static priority}) \times 20 & \text{if static priority} < 120 \\ (140 - \text{static priority}) \times 5 & \text{if static priority} \geq 120 \end{cases} \quad (1)$$

如你所见，静态优先权越高，基本时间片就越长。其结果是，与优先权低的进程相比，通常优先权较高的进程获得更长的CPU时间片。表7-2说明了普通进程的静态优先权、基本时间片和对应的nice值，与最高静态优先权、缺省静态优先权、最低优先权相对应的nice值。（表中还列出了交互式的值和睡眠时间极限的值，在本章稍后给予说明）。

表 7-2 普通进程优先权的典型值

说明	静态优先权	Nice值	基本时间片	交互式的值	睡眠时间的 极限值
最高静态优先权	100	-20	800ms	-3	299ms
高静态优先权	110	-10	600ms	-1	499ms
缺省静态优先权	120	0	100ms	+2	799ms
低静态优先权	130	+10	50ms	+4	999ms
最低静态优先权	139	+19	5ms	+6	1199ms

动态优先权和平均睡眠时间

普通进程除了静态优先权，还有动态优先权，其值的范围是 100（最高优先权）到 139（最低优先权）。动态优先权是调度程序在选择新进程来运行的时候使用的数。它与静态优先权的关系用下面的经验公式表示。

$$\text{动态优先权} = \max(100, \min(\text{静态优先权} - \text{bonus} + 5, 139)) \quad (2)$$

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。

粗略地说讲，平均睡眠时间是进程在睡眠状态所消耗的平均纳秒数。注意，这绝对不是对过去时间的求平均值的操作。例如：在 TASK_INTERRUPTIBLE 状态与在 TASK_UNINTERRUPTIBLE 状态的所计算出的平均睡眠时间是不同的。而且，进程在运行的过程中，平均睡眠时间递减。最后，平均睡眠时间永远不会大于 1 秒。

表 7-3 说明平均睡眠时间和 bonus 值的关系。（表中还列出了相应的时间片粒度，这将在稍后讨论）

表 7-3 平均睡眠时间、bonus 值以及时间片粒度

平均睡眠时间	Bonus	粒度
大于或等于 0 小于 100ms	0	5120
大于或等于 100 小于 200ms	1	2560
大于或等于 200 小于 300ms	2	1280
大于或等于 300 小于 400ms	3	640

大于或等于 400 小于 500ms	4	320
大于或等于 500 小于 600ms	5	160
大于或等于 600 小于 700ms	6	80
大于或等于 700 小于 800ms	7	40
大于或等于 800 小于 900ms	8	20
大于或等于 900 小于 1000ms	9	10
1 秒	10	10

平均睡眠时间也被调度程序用来确定一个给定进程是交互进程还是批处理进程。更明确地来讲，如果一个进程满足下面的公式，就被看作是交互进程：

$$\text{动态优先权} \leq 3 \times \text{静态优先权} / 4 + 28 \quad (3)$$

它相当于下面公式：

$$\text{bonus} - 5 \geq \text{静态优先权} / 4 - 28$$

表达式：静态优先权/4-28 被叫做交互式的；交互式的一些典型值在表 7-2 中列出。应该注意，高优先权进程比低优先权进程更容易成为交互进程。例如，具有最高静态优先权（100）的进程，当它的 bonus 值超过 2，即睡眠时间超过 200ms 时，就被看作是交互进程。相反，具有最低优先权（139）的进程决不会被当作交互进程，因为 bonus 值总是小于 11，相应的需要交互式 等于 6。一个具有缺省静态优先权（120）的进程，一但其平均睡眠时间超过 700ms 就成为交互进程。

活动和过期进程

即使具有较高静态优先权的普通进程获得了较大的 CPU 时间片，也不应该使静态优先权较低的进程无法运行。为了避免进程饥饿，当一个进程用完它的时间片时，它应该被还没有用完时间片的低优先权进程取代。为了实现这种机制，调度程序维持两个不相交的可运行进程的集合。

活动进程

这些进程还没有用完他们的时间片，因此允许他们运行。

过期进程

这些可运行进程已经用完了他们的时间片，并因此被禁止运行，直到所有活动进程都过期。

不过，总体的方案要稍微复杂一些，因为调度程序试图提升交互进程的性能。用完其时间片的活动批处理进程总是变成过期进程。用完其时间片的交互进程通常仍然是活动进程：调度

程序重填它的时间片并把它留在活动进程集合中。但是，如果最老的过期进程已经等待了很长时间，或者过期进程比交互进程的静态优先权高，调度程序就把用完时间片的交互进程移到过期进程集合中。结果，活动进程集合最终会变为空，过期进程将有机会运行。

实时进程的调度

每个实时进程都与一个实时优先权相关，实时优先权是一个范围从 1（最高优先权）到 99（最低优先权）的值。调度程序总是让优先权高的进程运行，换句话说，实时进程运行的过程中，禁止低优先权进程的执行。与普通进程相反，实时进程总是被当成活动进程（参见上一节）。用户可以通过系统调用 `sched_setparam()` 和 `sched_setscheduler()` 改变进程的实时优先权（参见本章稍后“与调度相关的系统调用”一节）。

如果几个可运行的实时进程具有相同的最高优先级，调度程序选择第一个出现在与本地 CPU 的运行队列相应链表中的进程（参见第 3 章“TASK_RUNNING 进程的链表”）。

只有在下述事件发生时，实时进程才会被另外一个进程取代。

- * 进程被另外一个具有更高优先权的实时进程抢占。
- *-进程执行了阻塞操作并进入睡眠（处于 TASK_INTERRUPTIBLE 或 TASK_UNINTERRUPTIBLE 状态）
- * 进程停止（处于 TASK_STOPPED 或 TASK_TRACED 状态）或被杀死（处于 EXIT_ZOMBIE 或 EXIT_DEAD 状态）
- *-进程通过调用系统调用 `sched_yield()`（参见本章稍后的“与调度相关的系统调用”一节）自动放弃 CPU。
- *-进程是基于时间片轮转的实时进程（SCHED_RR），而且用完了它的时间片。

系统调用 `nice()` 和 `setpriority()`，当用于基于时间片轮转的实时进程时，不改变实时进程的优先权而会改变其基本时间片的长度。实际上，基于时间片轮转的实时进程的基本时间片的长度与实时进程的优先权无关，而依赖于进程的静态优先权，他们的关系见前面“普通进程的调度”一节中的公式 (1)。

调度程序所使用的数据结构

回忆第三章“标识进程”一节，进程链表链接所有的进程描述符，而运行队列链表链接所有的可运行进程（也就是处于 TASK_RUNNING 状态的进程）的进程描述符，`swapper` 进程（idle 进程）除外。

数据结构 runqueue

数据结构 runqueue 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 (参见第五章“每 CPU 变量”一节)。宏 this_rq() 产生本地 CPU 运行队列的地址，而宏 cpu_rq(n) 产生索引为 n 的 CPU 的运行队列的地址。

表 7-4 列出了 runqueue 数据结构所包括的字段，在下面的章节中我们将对其中的大部分进行讨论。

表 7-4 runqueue 结构的字段

类型	名称	说明
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。
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（见本章稍后“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

runqueue 数据结构中最重要的字段是与可运行进程的链表相关的字段。系统中的每个可运行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，正如我们将要看到的，可运行进程会从一个运行队列迁移到另一个运行队列。

运行队列的 arrays 字段是一个包含两个 prio_array_t 结构的数组。每个数据结构都表示一个可运行进程的集合，并包括 140 个双向链表头（每个链表对应一个可能的进程优先权）、一个优先权位图和一个集合中所包含的进程数量的计数器（参见第三章表 3-2）。

图 7-1 runqueue 结构和可运行进程的两个集合

如图 7-1 所显示的，runqueue 结构的 active 字段指向数组中两个 prio_array_t 数据结构之一：对应于包含活动进程的可运行进程的集合。相反，expired 字段指向数组中的另一个 prio_array_t 数据结构：对应于包含过期进程的可运行进程的集合。

数组中两个数据结构的作用会发生周期性的变化：活动进程突然变成过期进程，而过期进程

变为活动进程，调度程序简单地交换运行队列的 `active` 和 `expired` 字段的内容以完成这种变化

进程描述符

每个进程描述符都包括几个与调度相关的字段，他们被列在表 7-5 中

表 7-5 与调度程序相关的进程描述符字段

类型	名称	说明
unsigned long	thread_info->flags	存放 TIF_NEED_RESCHED 标志，如果必须调用调度程序，则设置该标志（见第四章“从中断和异常返回”一节）
unsigned int	thread_info->cpu	可运行进程所在运行队列的 CPU 逻辑号
unsigned long	state	进程的当前状态（见第 3 章“进程状态”一节）
int	prio	进程的动态优先权
int	static_prio	进程的静态优先权
struct list_head	run_list	指向进程所属的运行队列中的下一个和前一个元素。
prio_array_t *	array	指向包含进程的运行队列的集合: prio_array_t
unsigned long	sleep_avg	进程的平均睡眠时间
unsigned long long	timestamp	进程最近插入运行队列的时间，或涉及本进程的最近一次进程切换的时间。
int	activated	进程被唤醒时所使用的条件码
unsigned long	policy	进程的调度类型 (SCHED_NORMAL, SCHED_RR, 或 SCHED_FIFO)
cpumask_t	cpus_allowed	能执行进程的 CPU 的位掩码

unsigned int	time_slice	在进程的时间片中还剩余的时钟节拍数
unsigned int	first_time_slice Flag set to 1 if the process never exhausted its time quantum	如果进程肯定不会用完其时间片，就把该标志设置为1。
unsigned long	rt_priority Real-time priority of the process	进程的实时优先权

当新进程被创建的时候，由 `copy_process()` 调用的函数 `sched_fork()` 用下述方法设置 `current` 进程（父进程）和 `p` 进程（子进程）的 `time_slice` 字段。

```
p->time_slice = (current->time_slice + 1) >> 1;
current->time_slice >>= 1;
```

换句话说，父进程剩余的节拍数被划分成两等份：一份给父进程，另一份给子进程。这样做是为了避免用户通过下述方法获得无限的 CPU 时间：父进程创建一个运行相同代码的子进程，并随后杀死自己，通过适当地调节创建的速度，子进程就可以总是在父进程过期之前获得新的时间片。因为内核**不奖赏创建**，所以这种编程技巧不起作用。类似地，用户不能通过在 shell 中运行几个后台进程，或通过在图形桌面打开许多窗口来不公平地霸占处理器。更通俗地讲就是一个进程不能通过创建多个后来霸占资源。（除非它有给自己实时策略的特权）。

如果父进程的时间片只剩下一个时钟节拍，划分操作强行把 `current->time_slice` 置为 0，从而耗尽父进程的时间片。这种情况下，`copy_process()` 把 `current->time_slice` 重新置为 1，然后调用 `scheduler_tick()` 递减该字段（见下一节）。

函数 `copy_process()` 也初始化子进程描述符中与进程调度相关的几个字段：

```
p->first_time_slice = 1;
p->timestamp = sched_clock();
```

因为子进程没有用完它的时间片（如果一个进程在它的第一个时间片内终止或执行新的程序就把子进程的剩余时间奖励给父进程），所以 `first_time_slice` 标志被置为 1。用函数 `sched_clock()` 所产生的时间戳的值初始化 `timestamp` 字段：实际上，函数 `sched_clock()` 返回被转化成纳秒的 64 位寄存器 TSC（见第 6 章“时间戳计数器（TSC）”一节）的内容。

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，其中最重要的函数是：

`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。

`scheduler_tick()` 函数

我们已经在第 6 章“更新本地 CPU 统计数”一节说明：每次时钟节拍到来时，`scheduler_tick()` 是如何被调用以执行与调度相关的操作的。它执行的主要步骤如下：

1. 把转换为纳秒的 TSC 的当前值存入本地运行队列的 `timestamp_last_tick` 字段。这个时间戳是从函数 `sched_clock()`（见前一节）获得的。

2. 检查当前进程是否是本地 CPU 的 `swapper` 进程，执行下面的子步骤：

- a. 如果本地运行队列除了 `swapper` 进程外，还包括另外一个可运行的进程，就设置当前进程的 `TIF_NEED_RESCHED` 字段，以强迫进行重新调度。就像我们在本章稍后“`schedule()` 函数一节”将要看到的，如果内核支持超线程技术（见本章稍后“**多处理器系统中运行队列平衡**”一节），那么，只要一个逻辑 CPU 运行队列中的所有进程都有比另一个逻辑 CPU（两个逻辑 CPU 对应同一个物理 CPU）上已经在执行的进程有低得多的优先权，前一个逻辑 CPU 就可能空闲，即使它的运行队列中有可运行的进程。

- b. 跳转到第 7 步（没必要更新 `swapper` 进程的时间片计数器）。

3. 检查 `current->array` 是否指向本地运行队列的活动链表。如果不是，说明进程已经过期但还没有被替换：设置 `TIF_NEED_RESCHED` 标志以强制进行重新调度并跳转到第七步。

4. 获得 `this_rq()->lock` 自旋锁。

5. 递减当前进程的时间片计数器，并检查是否已经用完时间片。进程的调度类型不同，函数所执行的这一步操作也有很大的差别，我们马上会讨论它们。

6. 释放 `this_rq()->lock` 自旋锁

7. 调用 `rebalance_tick()` 函数，该函数应该保证不同 CPU 的运行队列包含数量基本相同的可运行进程。稍后在“多处理器系统中运行队列平衡”一节我们将讨论运行队列的平衡。

更新实时进程的时间片

如果当前进程是先进先出（FIFO）的实时进程，函数 `scheduler_tick()` 什么都不做。实际上在这种情况下，`current` 所表示的进程（当前进程）不可能被比它优先较低或与它优先权相等的进程所抢占，因此，维持当前进程的最新时间片计数器是没有意义的。

如果 `current` 表示基于时间片轮转的实时进程，`scheduler_tick()` 就递减它的时间片计数器并检查时间片是否被用完：

```
if (current->policy == SCHED_RR &&!--current->time_slice) {
    current->time_slice = task_timeslice(current);
    current->first_time_slice = 0;
    set_tsk_need_resched(current);
    list_del(&current->run_list);
    list_add_tail(&current->run_list,
                 this_rq( )->active->queue+current->prio);
}
```

如果函数确定时间片确实用完了，就执行一系列操作以达到抢占当前进程的目的，如果必要的话，就尽快抢占。

第一步操作包括调用 `task_timeslice()` 来重填进程的时间片计数器。该函数检查进程的静态优先权，并根据在前面“普通进程的调度”一节中说明的公式（1）返回相应的基本时间片。此外，`current` 的 `first_time_slice` 字段被清 0：该标志被 `fork()` 系统调用服务例程中的 `copy_process()` 设置，并在进程的第一个时间片一用完时立刻清 0。

第二步，`scheduler_tick()` 调用函数 `set_tsk_need_resched()` 设置进程的 `TIF_NEED_RESCHED` 标志。就象第四章“从中断和异常返回”所描述的，该标志强制调用 `schedule()` 函数，以便 `current` 指向的进程能被另外一个有相同优先权（或更高优先权）的进程（如果有这种进程）所取代。

`scheduler_tick()` 的最后一步操作包括把进程描述符移到与当前进程优先权相应的运行队列

活动链表的尾部。把 `current` 指向的进程放到链表的尾部可以保证在每个优先权与它相同的可运行实时进程获得 CPU 时间片以前，不会再次被选择来执行。这是基于时间片轮转的调度策略。进程描述符的移动通过两个步骤完成：先调用 `list_del()` 把进程从运行队列的活动链表中删除，然后调用 `list_add_tail()` 把进程重新插入到同一个活动链表的尾部。

更新普通进程的时间片

如果当前进程是普通进程，函数 `scheduler_tick()` 执行下列操作：

1. 递减时间片计数器(`current->time_slice`).
- 2.检查时间片计数器，如果时间片用完，函数执行下列操作：
 - a.调用 `dequeue_task()` 从可运行进程的 `this_rq()`->`active` 集合中删除 `current` 指向的进程。
 - b.调用 `set_tsk_need_resched()` 设置 `TIF_NEED_RESCHED` 标志。
 - c.更新 `current` 指向的进程的动态优先权
`current->prio = effective_prio(current);`

函数 `effective_prio()` 读 `current` 的 `static_prio` 和 `sleep_avg` 字段，并根据在前面“普通进程的调度”一节中的公式 (2) 计算进程的动态优先权。

- d.重填进程的时间片：
`current->time_slice = task_timeslice(current);`
`current->first_time_slice = 0;`
- e. 如果本地运行队列数据结构的 `expired_timestamp` 字段等于 0（即：过期进程集合为空），就把当前时钟节拍的值赋给 `expired_timestamp`
`if (!this_rq()->expired_timestamp)`
`this_rq()->expired_timestamp = jiffies;`
- f.把当前进程插入活动进程集合或过期进程集合。
`if (!TASK_INTERACTIVE(current) || EXPIRED_STARVING(this_rq())) {`
`enqueue_task(current, this_rq()->expired);`
`if (current->static_prio < this_rq()->best_expired_prio)`
`this_rq()->best_expired_prio = current->static_prio;`
`} else`
`enqueue_task(current, this_rq()->active);`

如果用前面“普通进程的调度”一节说明的公式(3)识别出进程是一个交互进程，TASK_INTERACTIVE宏就产生值1。宏EXPIRED_STARVING检查运行队列中的第一个过期进程的等待时间是否已经超过1000个时钟节拍乘以运行队列中的可运行进程数加1，如果是，宏产生值1。如果当前进程的静态优先权大于一个过期进程的静态优先权，EXPIRED_STARVING宏也产生值1。

3. 否则，如果时间片没有用完（current->time_slice不等于0），检查当前进程的剩余时间片是否太长：

```
if (TASK_INTERACTIVE(p) && !((task_timeslice(p) -
    p->time_slice) % TIMESLICE_GRANULARITY(p)) &&
    (p->time_slice >= TIMESLICE_GRANULARITY(p)) &&
    (p->array == rq->active)) {
list_del(&current->run_list);
list_add_tail(&current->run_list,
    this_rq()->active->queue+current->prio);
set_tsk_need_resched(p);
}
```

宏TIMESLICE_GRANULARITY产生两个数的乘积给当前进程的bonus（见本章前面的表7-3），其中一个数为系统中CPU数量，另一个为成比例的常量。基本上，具有高静态优先权的交互进程，其时间片被分成大小为TIMESLICE_GRANULARITY的几个片段，以使这些进程不会独占CPU。

try_to_wake_up()函数

try_to_wake_up()函数通过把进程状态设置为TASK_RUNNING，并插入本地CPU的运行队列来唤醒睡眠或停止的进程。例如：调用该函数唤醒等待队列中的进程（见第三章“进程是如何被组织的一节”）或恢复执行等待信号的进程（见第11章）。该函数接受的参数：

- * 被唤醒进程的描述符指针（p）
- * 可以被唤醒的进程状态掩码（state）
- * 一个标志(sync)，用来禁止被唤醒的进程抢占本地CPU上正在运行的进程。

该函数执行下列操作：

1. 调用函数task_rq_lock()禁用本地中断，并获得最后执行进程的CPU（它可能不同于本地CPU）所拥有的运行队列rq的锁。CPU的逻辑号赋给p->thread_info->cpu字段。

2. 检查进程的状态 `p->state` 是否属于被当作参数传递给函数的状态掩码 `state`,如果不是,就跳转到第 9 步终止函数。

3. 如果 `p->array` 字段不等于 `NULL`, 进程已经属于某个运行队列, 因此跳转到第 8 步。

4. 在多处理器系统中, 该函数检查要被唤醒的进程是否应该从最近运行的 CPU 的运行队列迁移到另外一个 CPU 的运行队列。实际上, 函数就是根据一些启发式规则选择一个目标运行队列, 例如:

*如果系统中某些 CPU 空闲, 就选择空闲 CPU 的运行队列作为目标。按照优先选择先前正在执行进程的 CPU 和本地 CPU 这种顺序来进行。

*如果先前执行进程的 CPU 的工作量远小于本地 CPU 的工作量, 就选择先前的运行队列作为目标。

*如果进程最近被执行过, 选择老的运行队列作为目标 (可能仍然用这个进程的数据填充硬件高速缓存)

*如果把进程移到本地 CPU 以缓解 CPU 之间的不平衡, 目标就是本地运行队列(见本章稍后 “多处理器系统中运行队列的平衡” 一节)。

执行完这一步, 函数已经确定了目标 CPU 和对应的目标运行队列 `rq`, 前者将执行被唤醒的进程, 后者就是进程插入的队列。

1. 如果进程处于 `TASK_UNINTERRUPTIBLE` 状态, 函数递减目标运行队列的 `nr_uninterruptible` 字段, 并把进程描述符的 `p->activated` 字段设置为-1。参见后面的 “`recalc_task_prio()`函数” 一节对 `activated` 字段的说明。

2. 调用 `activate_task()`函数, 它依次执行下面的子步骤:

a.调用 `sched_clock()`获取以纳秒为单位的当前时间戳。如果目标 CPU 不是本地 CPU, 就要补偿本地时钟器中断的偏差, 这是通过计算本地 CPU 和目标 CPU 上最近一次发生时钟中断的相对时间戳来达到的。

```
now = (sched_clock( ) - this_rq( )->timestamp_last_tick)
      + rq->timestamp_last_tick;
```

b. 调用 `recalc_task_prio()`, 把进程描述符的指针和上一步计算出的时间戳传递给它。下一节详细说明 `recalc_task_prio()`函数。

c.根据本章稍后的表 7-6 设置 `p->activated` 字段的值。

d.使用在步骤 6a 中计算的时间戳设置 `p->timestamp` 字段

e.把进程描述符插入活动进程集合

```
enqueue_task(p, rq->active);
rq->nr_running++;
```

3. 如果目标 CPU 不是本地 CPU，或者没有设置 `sync` 标志，就检查可运行的新进程的动态优先级是否比 `rq` 运行队列中当前进程的动态优先级高 (`p->prio < rq->curr->prio`)，如果是，就调用 `resched_task()` 抢占 `rq->curr`。在单处理器系统中，后面的函数只是执行 `set_tsk_need_resched()` 来设置 `rq->curr` 进程的 `TIF_NEED_RESCHED` 标志。在多处理器系统中 `resched_task()` 也检查 `TIF_NEED_RESCHED` 的旧值是否为 0、目标 CPU 与本地 CPU 是否不同、`rq->curr` 进程的 `TIF_POLLING_NRFLAG` 标志是否清 0（目标 CPU 没有轮询进程 `TIF_NEED_RESCHED` 标志的值）。如果是，`resched_task()` 调用 `smp_send_reschedule()` 产生 IPI，并强制目标 CPU 重新调度（参见第 4 章“处理器间中断的处理”）。

4.把进程的 `p->state` 字段设置为 `TASK_RUNNING` 状态。

5. 调用 `task_rq_unlock()` 打开 `rq` 运行队列的锁并打开本地中断。

6. 返回 1（如果成功唤醒进程），或 0（如果进程没有被唤醒）

recalc_task_prio() 函数

函数 `recalc_task_prio()` 更新进程的平均睡眠时间和动态优先级。它接收进程描述符的指针和由函数 `sched_clock()` 计算出的当前时间戳。

该函数执行下述操作：

1. 把 $\min(\text{now} - \text{p->timestamp}, 10^9)$ 的结果赋给局部变量 `sleep_time`。

`p->timestamp` 字段包含导致进程进入睡眠状态的进程切换的时间戳，因此，`sleep_time` 中存放的是从进程最后一次执行开始，进程消耗在睡眠状态的纳秒数（如果进程睡眠的时间更长，`sleep_time` 就等于 1 秒）。

2. 如果 `sleep_time` 不大于 0, 就不用更新进程的平均睡眠时间, 直接跳转到第 8 步。

3. 检查进程是否不是内核线程、进程是否从 `TASK_UNINTERRUPTIBLE` 状态 (`p->activated` 字段等于 -1, 见前一节的第 5 步) 被唤醒、进程连续睡眠的时间是否超过给定的睡眠时间极限。如果这三个条件都满足, 函数把 `p->sleep_avg` 设置为相当于 900 个时钟节拍的值 (用最大平均睡眠时间减去一个标准进程的基本时间片长度获得的一个经验值)。然后跳转到第 8 步。

睡眠时间极限依赖于进程的静态优先权, 表 7-2 说明了它的一些典型值。简而言之, 这个经验规则的目的是保证已经在不可中断模式上 (通常是等待磁盘 I/O 的操作) 睡眠了很长时间的进程获得一个预先确定而且足够长的平均睡眠时间, 以使这些进程即能尽快获得服务, 又不会因睡眠时间太长而引起其它进程的饥饿。

4. 执行 `CURRENT_BONUS` 宏计算进程原来的平均睡眠时间的 `bonus` 值 (见表 7-3)。如果 $(10 - \text{bonus})$ 大于 0, 函数用这个值与 `sleep_time` 相乘。因为要把 `sleep_time` 加到进程的平均睡眠时间上 (见下面的第 6 步), 所以当前平均睡眠时间越短, 它增加的就越快。

5. 如果进程处于 `TASK_UNINTERRUPTIBLE` 状态而且不是内核线程, 执行下述子步骤:

a. 检查平均睡眠时间 `p->sleep_avg` 是否大于或等于进程的睡眠时间极限 (见本章前面的表 7-2)。如果是, 把局部变量 `sleep_avg` 重新置为 0, 因此不用调整平均睡眠时间, 而直接跳转到第 6 步。

b. 如果和 `sleep_avg + p->sleep_avg` 大于或等于睡眠时间极限, 就把 `p->sleep_avg` 字段置为睡眠时间极限并把 `sleep_avg` 设置为 0。

通过对进程平均睡眠时间的轻微限制, 函数不会对睡眠时间很长的批处理进程给予过多的奖赏。

6. 把 `sleep_time` 加到进程的平均睡眠时间上 (`p->sleep_avg`)。

7. 检查 `p->sleep_avg` 是否超过 1000 个时钟节拍 (以纳秒为单位), 如果是, 函数就把它减到 1000 个时钟节拍 (以纳秒为单位)。

8. 更新进程的动态优先权:

```
p->prio = effective_prio(p);
```

函数 `effective_prio()` 已经在本章前面 “`scheduler_tick()` 函数” 一节讨论过。

`schedule()` 函数

函数 `schedule()` 实现调度程序。它的任务是从运行队列的链表中找到一个进程，并随后将 CPU 分配给这个进程。`schedule()` 可以由几个内核控制路径调用，可以采取直接调用或延迟调用（可延迟的）的方式。

直接调用

如果 `current` 进程因不能获得必须的资源而要立刻被阻塞，就直接调用调度程序。在这种情况下，要阻塞进程的内核路径按下述步骤执行：

1. 把 `current` 进程 `current` 插入适当的等待队列。
2. 把 `current` 进程的状态改为 `TASK_INTERRUPTIBLE` 或 `TASK_UNINTERRUPTIBLE`。
3. 调用 `schedule()`。
4. 检查资源是否可用，如果不可用就转到第 2 步。
5. 一旦资源可用就从等待队列中删除当前进程 `current`。

内核路径反复检查进程需要的资源是否可用，如果不可用，就调用 `schedule()` 把 CPU 分配给其它进程。稍后，当调度程序再次允许把 CPU 分配给这个进程时，要重新检查资源的可用性。这些步骤与 `wait_event()` 所执行的步骤很相似，也与第 3 章“如何组织进程”一节的函数很相似。

许多反复执行长任务的设备驱动程序也直接调用调度程序。每次反复循环时，驱动程序都检查 `TIF_NEED_RESCHED` 标志，如果需要就调用 `schedule()` 自动放弃 CPU。

延迟调用

也可以把 `TIF_NEED_RESCHED` 标志设置为 1，而以延迟方式调用调度程序。由于总是在恢复用户态进程的执行之前检查这个标志的值（见第 4 章从“中断和异常返回”一节），所以 `schedule()` 将在不久之后的某个时间被明确地调用。

延迟调用调度程序的典型例子：

*当 `current` 进程用完了它的 CPU 时间片时，由 `scheduler_tick()` 函数完成 `schedule()` 的延迟调用。

*当一个被唤醒进程的优先权比当前进程的优先权高时，由 `try_to_wake_up()` 函数完成 `schedule()` 的延迟调用。

*当发出系统调用 `sched_setscheduler()` 时（见本章稍后“与调度相关的系统调用”一节）

进程切换之前 `schedule()` 所执行的操作

`schedule()` 函数的任务之一是用另外一个进程来替换当前正在执行的进程。因此，该函数的关键结果是设置一个叫做 `next` 的变量，使它指向被选中的进程，该进程将取代当前进程。如果系统中没有优先权高于当前进程的可运行进程，最终 `next` 与 `current` 相等，不发生任何进程切换。

schedule()函数在一开始，先禁用内核抢占并初始化一些局部变量：

```
need_resched;
preempt_disable( );
prev = current;
rq = this_rq( );
```

正如你所见，把 current 返回的指针赋给 prev,并把与本地 CPU 相对应的运行队列数据结构的地址赋给 rq.

下一步，schedule()要保证 prev 不占用大内核锁（参见第 5 章“大内核锁”一节）：

```
if (prev->lock_depth >= 0)
    up(&kernel_sem);
```

注意，schedule()不改变 lock_depth 字段的值，当 prev 恢复执行的时候，如果该字段的值不等于负数，prev 重新获得 kernel_flag 自旋锁。因此，通过进程切换，会自动释放和重新获取大内核锁。

调用 sched_clock()函数以读取 TSC，并将它的值转换成纳秒，所获得的时间戳存放在局部变量 now 中。然后，schedule()计算 prev 所用的时间片长度：

```
now = sched_clock( );
run_time = now - prev->timestamp;
if (run_time > 1000000000)
    run_time = 1000000000;
```

通常使用限制在 1 秒（要转换成纳秒）的时间。run_time 的值用来限制进程对 CPU 的使用。不过，鼓励进程有较长的平均睡眠时间：

```
run_time /= (CURRENT_BONUS(prev) ? : 1);
```

记住，CURRENT_BONUS 返回 0 到 10 之间的值，它与进程的平均睡眠时间是成比例的。

在开始寻找可运行进程之前，schedule()必须关掉本地中断，并获得所要保护的运行队列的自旋锁：

```
spin_lock_irq(&rq->lock);
```

正如在第 3 章“进程终止”一节中所描述的，prev 可能是一个正在被终止的进程。为了确认这个事实，schedule()检查 PF_DEAD 标志：

```
if (prev->flags & PF_DEAD)
```

```
prev->state = EXIT_DEAD;
```

接下来，`schedule()` 检查 `prev` 的状态，如果不是可运行状态，而且它没有在内核态被抢占（见第四章“从中断和异常返回”一节），就应该从运行队列删除 `prev` 进程。不过，如果它是非阻塞挂起信号，而且状态为 `TASK_INTERRUPTIBLE`，函数就把该进程的状态设置为 `TASK_RUNNING`，并将它插入运行队列。这个操作与把处理器分配给 `prev` 是不同的，它只是给 `prev` 一次被选中执行的机会。

```
if (prev->state != TASK_RUNNING &&
    !(preempt_count() & PREEMPT_ACTIVE)) {
    if (prev->state == TASK_INTERRUPTIBLE && signal_pending(prev))
        prev->state = TASK_RUNNING;
    else {
        if (prev->state == TASK_UNINTERRUPTIBLE)
            rq->nr_uninterruptible++;
        deactivate_task(prev, rq);
    }
}
```

函数 `deactivate_task()` 从运行队列中删除该进程：

```
rq->nr_running--;
dequeue_task(p, p->array);
p->array = NULL;
```

现在，`schedule()` 检查运行队列中剩余的可运行进程数。如果有可运行的进程，`schedule()` 就调用 `dependent_sleeper()` 函数，在绝大多数情况下，该函数立即返回 0。但是，如果内核支持超线程技术（见本章稍后“多处理器系统中运行队列的平衡”一节），函数检查要被选中执行的进程，其优先级是否比已经在相同物理 CPU 的某个逻辑 CPU 上运行的兄弟进程的优先级低，在这种特殊的情况下，`schedule()` 拒绝选择低优先级的进程，而去执行 `swapper` 进程。

```
if (rq->nr_running) {
    if (dependent_sleeper(smp_processor_id(), rq)) {
        next = rq->idle;
        goto switch_tasks;
    }
}
```

如果运行队列中没有可运行的进程存在，函数就调用 `idle_balance()`，从另外一个运行队

列迁移一些可运行进程到本地运行队列中，`idle_balance()`与`load_balance()`类似，在稍后“`load_balance()`函数”一节中将对它进行说明。

```
if (!rq->nr_running) {
    idle_balance(smp_processor_id(), rq);
    if (!rq->nr_running) {
        next = rq->idle;
        rq->expired_timestamp = 0;
        wake_sleeping_dependent(smp_processor_id(), rq);
        if (!rq->nr_running)
            goto switch_tasks;
    }
}
```

如果`idle_balance()`没有成功地把进程迁移到本地运行队列中，`schedule()`就调用`wake_sleeping_dependent()`重新调度空闲CPU（即每个运行swapper进程的CPU）中的可运行进程。就象前面讨论`dependent_sleeper()`函数时所说明的，通常在内核支持超线程技术的时候可能会出现这种情况。然而，在单处理机系统中，或者当把进程迁移到本地运行队列的种种努力都失败的情况下，函数就选择swapper进程作为next进程并继续进行下一步骤。

我们假设`schedule()`函数已经肯定运行队列中有一些可运行的进程，现在它必须检查这些可运行进程中是否至少有一个进程是活动的，如果没有，函数就交换运行队列数据结构的`active`和`expired`字段的内容，因此，所有的过期进程变为活动进程，而空集合准备接纳将要过期的进程。

```
array = rq->active;
if (!array->nr_active) {
    rq->active = rq->expired;
    rq->expired = array;
    array = rq->active;
    rq->expired_timestamp = 0;
    rq->best_expired_prio = 140;
}
```

现在可以在活动的`prio_array_t`数据结构中搜索一个可运行进程了（参见第三章“标识进程”一节）。首先，`schedule()`搜索活动进程集合位掩码的第一个非0位。回忆一下，当对应的优先级链表不为空时，就把位掩码的相应位置1。因此，第一个非0位的下标对应包含最佳运行进程的链表，随后，返回该链表的第一个进程描述符：

```
idx = sched_find_first_bit(array->bitmap);
next = list_entry(array->queue[idx].next, task_t, run_list);
```

函数 `sched_find_first_bit()` 是基于 `bsf1` 汇编语言指令的，它返回 32 位字中被设置为 1 的最低位的位下标。

局部变量 `next` 现在存放将取代 `prev` 的进程描述符。`schedule()` 函数检查 `next->activated` 字段，该字段的编码值表示进程在被唤醒时的状态，如表 7-6 所示：

值	说明
0	进程处于 <code>TASK_RUNNING</code> 状态
1	进程处于 <code>TASK_INTERRUPTIBLE</code> 或 <code>TASK_STOPPED</code> 状态，而且正在被系统调用服务例程或内核线程唤醒。
2	进程处于 <code>TASK_INTERRUPTIBLE</code> 或 <code>TASK_STOPPED</code> 状态，而且正在被中断处理程序或可延迟函数唤醒。
-1	进程处于 <code>TASK_UNINTERRUPTIBLE</code> 状态而且正在被唤醒。

如果 `next` 是一个普通进程而且它正在从 `TASK_INTERRUPTIBLE` 或 `TASK_STOPPED` 状态被唤醒，调度程序就把自从进程插入运行队列开始所经过的纳秒数加到进程的平均睡眠时间中。换言之，进程的睡眠时间被增加了，以包含进程在运行队列中等待 CPU 所消耗的时间。

```
if (next->prio >= 100 && next->activated > 0) {
    unsigned long long delta = now - next->timestamp;
    if (next->activated == 1)
        delta = (delta * 38) / 128;
    array = next->array;
    dequeue_task(next, array);
    recalc_task_prio(next, next->timestamp + delta);
    enqueue_task(next, array);
}
next->activated=0;
```

要说明的是，调度程序把被中断处理程序和可延迟函数所唤醒的进程与被系统调用服务例程和内核线程所唤醒的进程区分开来，在前一种情况下，调度程序增加全部运行队列等待时间而在后一种情况下，它只增加等待时间的一部分。这是因为交互式进程更可能被异步事件（考虑用户在键盘上的按键操作）而不是同步事件唤醒。

schedule()完成进程切换时所执行的操作

现在 schedule()函数已经要让 next 进程投入运行。内核将立刻访问 next 进程的 thread_info 数据结构，它的地址存放在 next 进程描述符的接近顶部的位置。

```
switch_tasks;
prefetch(next);
```

prefetch 宏提示 CPU 控制单元把 next 进程描述符的第一部分字段的内容装入硬件高速缓存，正是这一点改善了 schedule()的性能，因为对于后续指令的执行（不影响 next），数据是并行移动的。

在替代 prev 之前，调度程序应该完成一些管理的工作：

```
clear_tsk_need_resched(prev);
rcu_qsctr_inc(prev->thread_info->cpu);
```

以防（万一）以延迟方式调用 schedule()，clear_tsk_need_resched()函数清除 prev 的 TIF_NEED_RESCHED 标志。然后，函数记录 CPU 正在经历静止状态（参见第 5 章“读-拷贝更新 (RCU)”一节）。

schedule()函数还必须减少 prev 的平均睡眠时间，并把它补充给进程所使用的 CPU 时间片：

```
prev->sleep_avg -= run_time;
if ((long)prev->sleep_avg <= 0)
    prev->sleep_avg = 0;
prev->timestamp = prev->last_ran = now;
随后更新进程的时间戳。
```

prev 和 next 很可能是同一个进程：在当前运行队列中没有优先权较高或相等的其他活动进程时，会发生这种情况。在这种情况下，函数不做进程切换：

```
if (prev == next) {
    spin_unlock_irq(&rq->lock);
    goto finish_schedule;
}
```

这里，prev 和 next 是不同的进程，进程切换确实发生了：

```
next->timestamp = now;
rq->nr_switches++;
rq->curr = next;
prev = context_switch(rq, prev, next);
```

context_switch()函数建立 next 的地址空间。正如我们将在第9章“内核线程的内存描述符”中将要看到的，进程描述符的 active_mm 字段指向进程所使用的内存描述符，而 mm 字段指向进程所拥有的内存描述符。对于一般的进程，这两个字段有相同的地址，但是，内核线程没有它自己的地址空间而且它的 mm 字段总是被设置为 NULL。context_switch()函数保证：如果 next 是一个内核线程，它使用 prev 所使用的地址空间：

```
if (!next->mm) {
    next->active_mm = prev->active_mm;
    atomic_inc(&prev->active_mm->mm_count);
    enter_lazy_tlb(prev->active_mm, next);
}
```

一直到Linux 2.2 版，内核线程都有自己的地址空间。那种设计选择不是最理想的，因为不管什么时候当调度程序选择一个新进程（即使是一个内核线程）运行时，都必须改变页表；因为内核线程都运行在内核态，它仅使用线性地址空间的第4个GB，其映射对系统的所有进程都是相同的。甚至最坏情况下，写cr3寄存器会使所有的TLB表项无效（参见第二章“转换旁路缓存器（TLB）一节），这将导致极大的性能损失。现在的Linux具有更高的效率，因为如果next是内核线程，就根本不触及页表。作为进一步的优化，如果next是内核线程，schedule()函数把进程设置为懒惰TLB模式（参见第二章“转换旁路缓存器（TLB）一节”）。

相反，如果next是一个普通进程，schedule()函数用next的地址空间替换prev的地址空间：

```
if (next->mm)
    switch_mm(prev->active_mm, next->mm, next);
```

如果prev是内核线程或正在退出的进程，context_switch()函数就把指向prev内存描述符的指针保存到运行队列的prev_mm 字段中，然后重新设置prev->active_mm：

```
if (!prev->mm) {
    rq->prev_mm = prev->active_mm;
    prev->active_mm = NULL;
}
```

现在，context_switch()终于可以调用 switch_to()执行 prev 和 next 之间的进程切换了（见第三章“执行进程切换”）一节：

```
switch_to(prev, next, prev);
return prev;
```

进程切换后 schedule()所执行的操作

`schedule()` 函数中在 `switch_to` 宏之后紧接着的指令并不由 `next` 进程立即执行，而是稍后当调度程序选择 `prev` 又执行时由 `prev` 执行。然而，在那个时刻，`prev` 局部变量并不指向我们开始描述 `schedule()` 时所替换出去的原来那个进程，而是指向 `prev` 被调度时由 `prev` 替换出的原来那个进程。（如果你被搞糊涂，请回到第三章阅读“执行进程切换”一节）。

进程切换后的第一部分指令是：

```
barrier( );
finish_task_switch(prev);
```

在 `schedule()` 中，紧接着 `context_switch()` 函数调用之后，宏 `barrier()` 产生一个代码优化屏障（见第 5 章“优化和内存屏障”一节）。然后，执行 `finish_task_switch()` 函数：

```
mm = this_rq( )->prev_mm;
this_rq( )->prev_mm = NULL;
prev_task_flags = prev->flags;
spin_unlock_irq(&this_rq( )->lock);
if (mm)
    mmdrop(mm);
if (prev_task_flags & PF_DEAD)
    put_task_struct(prev);
```

如果 `prev` 是一个内核线程，运行队列的 `prev_mm` 字段存放借给 `prev` 的内存描述符的地址。正如我们在第 9 章将要看到的，`mmdrop()` 减少内存描述符的使用计数器，如果该计数器等于 0 了，函数还要释放与页表相关的所有描述符和虚拟存储区。

`finish_task_switch()` 函数还要释放运行队列的自旋锁并打开本地中断。然后，检查 `prev` 是否是一个正在从系统中被删除的僵死任务（见第 3 章“进程的终止”）一节，如果是，就调用 `put_task_struct()` 以释放进程描述符引用计数器，并撤消所有其余对该进程的引用（见第 3 章“进程的删除”一节）。

`schedule()` 函数的最后一部分指令是：

```
finish_schedule:
prev = current;
if (prev->lock_depth >= 0)
    _ _reacquire_kernel_lock( );
preempt_enable_no_resched();
if (test_bit(TIF_NEED_RESCHED, &current_thread_info( )->flags)
    goto need_resched;
return;
```

如你所见，`schedule()`在需要的时候重新获得大内核锁、重新启用内核抢占、并检查是否一些其他的进程已经设置了当前进程的 `TIF_NEED_RESCHED` 标志，如果是，整个 `schedule()`函数重新开始执行，否则，函数结束。

多处理器系统中运行队列的平衡

我们在第 4 章已经看到，Linux 一直坚持采用对称多处理模式，这意味着，**与其他 CPU 相比**，内核不应该对一个 CPU 有任何偏向，但是，多处理器机器具有很多不同的风格，而且调度程序的实现随硬件特征的不同而有所不同，我们将特别关注下面三种不同类型的多处理器机器：

标准的多处理器体系结构

直到最近，这是多处理器机器最普通的体系结构。这些机器所共有的 RAM 芯片集被所有 CPU 共享。

超线程

超线程芯片是一个立刻执行几个执行线程的微处理器；它包括几个内部寄存器的拷贝，并快速在它们之间切换。这种由 Intel 发明的技术，使得当前线程在访问内存的间隙，处理器可以使用它的机器周期去执行另外一个线程。一个超线程的物理 CPU 可以被 Linux 看作几个不同的逻辑 CPU。

NUMA

把 CPU 和 RAM 以本地“结点”为单位分组，（通常一个结点包括一个 CPU 和几个 RAM 芯片）。内存仲裁器（一个使系统中的 CPU 以串型方式访问 RAM 的专用电路，见第 2 章“内存访问”一节）是典型的多处理器系统的瓶颈。在 NUMA 体系结构中，当 CPU 访问与它同在一个结点中的“本地”RAM 芯片时，几乎没有竞争，因此访问通常是非常快的。另一方面，访问它所属结点外的“远程”RAM 芯片就非常慢。我们将在第 8 章“非一致内存访问（NUMA）”一节讨论 Linux 内核内存分配器是如何支持 NUMA 体系结构的。

这些基本的多处理器系统类型经常被组合使用。例如，内核把一个包括两个不同超线程 CPU 的主板看作四个逻辑 CPU。

正如我们在上一节所看到的，`schedule()`函数从本地 CPU 的运行队列挑选新进程运行。因此，一个指定的 CPU 只能执行它相应的运行队列中的可运行进程。另外，一个可运行进程总是存放在某一个运行队列中：任何一个可运行进程都不可能同时出现在两个或多个运行队列中。因此，一个保持可运行状态的进程通常被限制在一个固定的 CPU 上。

这种设计通常对系统性能是有益的，因为，运行队列中的可运行进程所拥有的数据可能填满每个 CPU 的硬件高速缓存。但是，在有些情况下，把可运行进程限制在一个指定的 CPU 上可

能引起严重的性能**损失**。例如，考虑频繁使用 CPU 的大量批处理进程：如果他们绝大多数都在同一个运行队列中，那么系统中一个 CPU 将会超负荷，而其他一些 CPU 几乎处于空闲状态。

因此，内核周期性地检查运行队列的工作量是否平衡，并在需要的时候，把一些进程从一个运行队列迁移到另一个运行队列。但是，为了从多处理器系统获得最佳性能，负载平衡算法应该考虑系统中 CPU 的拓扑结构。从内核 2.6.7 版本开始，Linux **提出**一种基于“调度域”概念的复杂的运行队列平衡算法。正是有了调度域这一概念，使得这种算法能够很容易适应各种已有的多处理器体系结构（甚至诸如那些基于“多核”微处理器的新近出现的体系结构）。

调度域

调度域实际上是一个 CPU 集合，他们的工作量应当由内核保持平衡。一般来说，调度域采取分层的组织形式：最上层的调度域（通常包括系统中的所有 CPU）包括多个子调度域，每个子调度域包括一个 CPU 子集。正是调度域的这种分层结构，使工作量的平衡能以如下有效方式来实现：

每个调度域被依次划分成一个或多个组，每个组代表调度域的一个 CPU 子集。工作量的平衡总是在调度域的组之间来完成。换言之，只有在一些调度域的某些组的总工作量远远低于同一个调度域的另一个组的工作量时，才把进程从一个 CPU 迁移到另一个 CPU。

图 7-2 说明三个调度域分层实例，对应三种主要的多处理器机器体系结构：

图 7-2 调度域分层的三个实例

图 7-2 (a) 表示具有两 CPU 的**标准多处理器体系结构**中由单个调度域组成的一个层次结构，该调度域包括两个组，每个组有一个 CPU。

图 7-2 (b) 表示一个两层的层次结构，在使用超线程技术、有两 CPU 的多处理器结构中。最上层的调度域包括了系统中所有四个逻辑 CPU，它由两个组构成。上层域的每个组对应一个子调度域并包括一个物理 CPU。底层的调度域(也被称为基本调度域)包括两个组，每个组一个逻辑 CPU。

最后，图 7-2 (c) 表示有两个结点，每个结点有四个 CPU 的 8-CPU NUMA 体系结构上的两层层次结构。最上层的域由两个组构成，每个组对应一个不同的结点。每个基本调度域包括一个结点内的 CPU，包括四个组，每个组包括一个 CPU。

每个调度域由一个 `sched_domain` 描述符表示，而调度域中的每个组由 `sched_group` 描述符表示。每个 `sched_domain` 描述符包括一个 `groups` 字段，它指向组描述符链表中的第一个元素。此外，`sched_domain` 结构的 `parent` 字段指向父调度域的描述符（如果有的话）。

系统中所有物理 CPU 的 `sched_domain` 描述符都存放在每 CPU 变量 `phys_domains` 中。如果内核不支持超线程技术，这些域就在域层次结构的最底层，运行队列描述符的 `sd` 字段指向它们，即它们是基本的调度域。相反，如果内核支持超线程技术，底层调度域存放在每 CPU 变量 `cpu_domains` 中。

rebalance_tick()函数

为了保持系统中运行队列的平衡，每次经过一次时钟节拍 `scheduler_tick()`，就调用 `rebalance_tick()` 函数。它接受的参数有：本地 CPU 的索引 `this_cpu`、本地运行队列的地址 `this_rq` 以及一个标志 `idle`，该标志可以取下面的值：

`SCHED_IDLE`

CPU 当前空闲，即 `current` 是 `swapper` 进程。

`NOT_IDLE`

CPU 当前不空闲，即 `current` 不是 `swapper` 进程。

`rebalance_tick()` 函数首先确定运行队列中的进程数，并更新进程队列的平均工作量，为了完成这个工作，函数要访问运行队列描述符的 `nr_running` and `cpu_load` 字段。

随后，`rebalance_tick()` 开始在所有调度域上的循环，其路径是从基本域（本地运行队列描述符的 `sd` 字段所引用的域）到最上层的域。在每次循环中，函数确定是否已到调用函数 `load_balance()` 的时间，从而在调度域上执行**重新平衡**的操作。由存放在 `sched_domain` 域中的参数和 `idle` 值决定调用 `load_balance()` 的频率。如果 `idle` 等于 `SCHED_IDLE`，那么运行队列为空，`rebalance_tick()` 就以很高的频率调用 `load_balance()`（大概每一到两个节拍处理一次对应于逻辑和物理 CPU 的调度域）。相反，如果 `idle` 等于 `NOT_IDLE`，`rebalance_tick()` 就以很低的频率调度 `load_balance()`（大概每 10ms 处理一次逻辑 CPU 对应的调度域，每 100ms 处理一次物理 CPU 对应的调度域）。

load_balance()函数

`load_balance()` 函数检查是否调度域处于严重的不平衡状态。更确切地说，它检查是否可以通过把最繁忙的组中的一些进程迁移到本地 CPU 的运行队列来减轻不平衡的状况，如果是，函数尝试实现这个迁移。它接受四个参数：

`this_cpu`

本地 CPU 的索引

`this_rq`

本地运行队列的描述符的地址

`sd`

指向被检查的调度域的描述符

`idle`

取值为 `SCHED_IDLE`（本地 CPU 空闲）或 `NOT_IDLE`

函数执行下面的操作:

1. 获取 `this_rq->lock` 自旋锁。

2. 调用 `find_busiest_group()` 函数分析调度域中各组的工作量。函数返回最繁忙的组的 `sched_group` 描述符, 假设这个组不包括本地 CPU, 在这种情况下, 函数还返回为了恢复平衡而被迁移到本地运行队列中的进程数。另一方面, 如果最繁忙的组包括本地 CPU 或所有的组本来就是平衡的, 函数返回 `NULL`。这个过程不是微不足道的, 因为函数试图过滤掉统计工作量中的波动。

3. 如果 `find_busiest_group()` 在调度域中没有找到既不包括本地 CPU 又非常繁忙的组, 就释放 `this_rq->lock` 自旋锁, 调整调度域描述符的参数, 以延迟本地 CPU 上下一次对 `load_balance()` 的调度, 然后函数终止。

4. 调用 `find_busiest_queue()` 函数查找在第 2 步中找到的组中最繁忙的 CPU, 函数返回相应运行队列的描述符地址 `busiest`。

5. 获取另一个自旋锁, 也就是 `busiest->lock` 自旋锁。为了避免死锁, 这一操作必须非常小心: 首先释放 `this_rq->lock`, 然后通过增加 CPU 下标获得这两个锁。

6. 调用 `move_tasks()` 函数, 尝试从最繁忙的运行队列中把一些进程迁移到本地运行队列 `this_rq` 中 (见下一节)。

7. 如果函数 `move_task()` 没有成功地把某些进程迁移到本地运行队列, 调度域还是不平衡。把 `busiest->active_balance` 标志设置为 1, 并唤醒 `migration` 内核线程, 它的描述符存放在 `busiest->migration_thread` 中。Migration 内核线程顺着调度域的链搜索—从最繁忙运行队列的基本域到最上层域, 寻找空闲 CPU。如果找到一个空闲 CPU, 该内核线程就调用 `move_tasks()` 把一个进程迁移到空闲运行队列。

8. 释放 `busiest->lock` 和 `this_rq->lock` 自旋锁

9. 结束

move_tasks() 函数

`move_tasks()` 函数把进程从源运行队列迁移到本地运行队列。它接受 6 个参数: `this_rq` 和 `this_cpu` (本地运行队列描述符和本地 CPU 下标)、`busiest` (源运行队列描述符)、`max_nr_move` (被迁移进程的最大数)、`sd` (在其中执行平衡操作的调度域的描述符地址) 以及 `idle` 标志 (除了可以被设置为 `SCHED_IDLE` 和 `NOT_IDLE`, 在函数被 `idle_balance()` 间接

调用时，该标志还可以被设置为 `NEWLY_IDLE`。见本章前面“`schedule()`函数”一节)。

函数首先分析 `busiest` 运行队列的过期进程，从优先权高的进程开始。当扫描完所有过期进程后，函数扫描 `busiest` 运行队列的活动进程。函数对所有的后选进程调用 `can_migrate_task()`，如果下列条件都满足 `can_migrate_task()`返回 1:

- * 进程当前没有在远程 CPU 上执行
- * 本地 CPU 包含在进程描述符的 `cpus_allowed` 位图中
- * 至少满足下列条件之一：
 - * 本地 CPU 空闲。如果内核支持超线程技术，所有本地物理芯片中的逻辑 CPU 必须空闲。
 - * 内核在平衡调度域时因反复进行进程迁移都不成功而陷入困境。
 - * 被迁移的进程不是“高速缓存命中”的（最近不曾在远程 CPU 上执行，因此可以设想远程 CPU 上的硬件高速缓存中没有该进程的数据）。

如果 `can_migrate_task()`返回 1，`move_tasks()`就调用 `pull_task()`函数把后选进程迁移到本地运行队列中。实际上，`pull_task()`执行 `dequeue_task()`从远程运行队列删除进程，然后执行 `enqueue_task()`把进程插入本地运行队列，最后，如果刚被迁移的进程比当前进程拥有更高的动态优先权，就调用 `resched_task()`抢占本地 CPU 的当前进程。

与调度相关的系统调用⁴

已经介绍的几个系统调用允许进程改变它们的优先级及调度策略。作为一般原则，总是允许用户降低他们进程的优先级。然而，如果他们想修改属于其他某一用户进程的优先级，或者如果他们想增加他们自己进程的优先级，那么，他们必须拥有超级用户的特权。

`nice()`系统调用

`nice()`（注3）系统调用允许进程改变它们的基本优先级。包含在 `increment` 参数中的整数值用来修改进程描述符的 `nice` 字段。在 Unix 中的 `nice` 命令（允许用户用修改的调度优先级来运行程序）就是基于这个系统调用的。

`sys_nice()` 服务例程处理 `nice()` 系统调用。尽管 `increment` 参数可以有任意值，但是大于 40 的绝对值会被截为 40。从传统上来说，负值相当于请求优先级增加，并请求超级特权，而正值相当于请求优先级减少。在负增加的情况下，调用 `capable()` 函数核实进程是否有 `CAP_SYS_NICE` 权能。

而且，函数调用 `security_task_setnice()` 安全勾。我们在二十章讨论那个函数。如果用户想用请求的权能来改变优先级，`sys_nice()` 就把 `current->static_prio` 转换到 `nice` 值的范围，再加上 `increment` 的值，并调用 `set_user_nice()` 函数。然后依次执行几个操作：后面的函数

⁴ 注 3：因为这个系统调用用来降低进程的优先级，因此为了自己的优先级而调用它的用户对其他用户来说就是“美好的（`nice`）”

获得本地运行队列锁，更新current进程的静态优先权，调用resched_task()函数以允许其他进程抢占current进程，并释放运行队列锁。

nice()系统调用只维持向后兼容；它已经被下面描述的setpriority()系统调用取代。

getpriority() 和 setpriority() 系统调用

nice()系统调用只影响调用它的进程，而另外两个系统调用getpriority()和setpriority()则作用于给定组中所有进程的基本优先级。getpriority()返回20减去给定组中所有进程之中最低nice字段的值；setpriority()把给定组中所有进程的基本优先级都设置为一个给定的值。

内核对这两个系统调用的实现是通过sys_getpriority()和sys_setpriority()服务例程完成的。这两个服务例程本质上作用于同一组相同的参数：

which

指定进程组。它采用下列值之一：

PRIO_PROCESS

根据进程的ID选择进程（进程描述符的pid字段）

PRIO_PGRP

根据组ID选择进程（进程描述符的pgrp字段）

PRIO_USER

根据用户ID选择进程（进程描述符的uid字段）

who

用pid、pgrp及uid字段的值（取决于which的值）选择进程。如果who是0，把它的值设置为current进程相应字段的值。

niceval

新的基本优先级值（仅被sys_setpriority()所需要）。它的取值范围应该在-20（最高优先级）和+19（最小优先级）之间。

正如以前提到的，只有具有CAP_SYS_NICE 权能的进程才允许增加它们自己的基本优先级或修改其他进程的优先级。

正如我们将在第10章看到的，只有当出现了某些错误时，系统调用才返回一个负值。由于这个原因，getpriority()不返回-20到+19之间正常的nice值，而是0~40之间的一个非负值。

sched_getaffinity() 和 sched_setaffinity() 系统调用

sched_getaffinity() 和sched_setaffinity()系统调用分别返回和设置CPU进程亲合力掩码，

也就是允许对进程执行的CPU的位掩码。该掩码存放在进程描述符的cpus_allowed字段中。

sys_sched_getaffinity()系统调用服务例程通过调用 find_task_by_pid()搜索进程描述符, 然后返回的值为相应字段 cpus_allowed 与可用 CPU 位图做与运算的结果。

系统调用 sys_sched_setaffinity()有一点复杂。除了寻找目标进程的描述符并更新 cpus_allowed 字段, 该函数还必须检查进程所属的运行队列, 其对应的 CPU 亲和力掩码是否不再是最新值。在这种糟糕的情况下, 必须把进程从一个运行队列迁移到另一个运行队列。为了避免死锁和竞争条件的问题, 由迁移内核线程 (每个 CPU 有一个这样的线程) 完成这个工作。一旦必须把进程从运行队列 rq1 迁移到运行队列 rq2, sys_sched_setaffinity()系统调用就唤醒 rq1 的迁移线程(rq1->migration_thread), 它依次从 rq1 中删除被迁移的进程, 然后把它插入 rq2。

与实时进程相关的系统调用

现在我们介绍一组系统调用, 它们允许进程改变自己的调度规则, 尤其是可以变为实时进程。进程为了修改任何进程 (包括自己) 的描述符的rt_priority和policy字段, 同样必须具有 CAP_SYS_NICE 权能。

sched_getscheduler()和sched_setscheduler()系统调用

sched_getscheduler()查询由 pid 参数所表示的进程当前所用的调度策略。如果 pid 等于 0, 将检索调用进程的策略。如果成功, 这个系统调用为进程返回策略: SCHED_FIFO、SCHED_RR 或 SCHED_NORMAL (后者也称为 SCHED_OTHER)。相应的 sys_sched_getscheduler()服务例程调用 find_task_by_pid(), 后一个函数确定给定 pid 所对应的进程描述符, 并返回其 policy 字段的值。

sched_setscheduler()系统调用既设置调度策略, 也设置由参数pid所表示进程的相关参数。如果pid等于0, 调用进程的调度程序参数将被设置。

相应的 sys_sched_setscheduler()系统调用服务例程简单地调用 do_sched_setscheduler() 函数。后者检查由参数 policy 指定的调度策略和由参数 param->sched_priority 指定的新优先权是否有效。它还检查进程是否具有 CAP_SYS_NICE 权能或者进程的拥有者是否有超级用户的权限。如果每个条件都满足, 就把进程从它的运行队列 (如果进程是可运行的) 中删除; 更新进程的静态优先权、实时优先权和动态优先权; 把进程插回到运行队列; 最后, 在需要的情况下, resched_task()函数抢占运行队列的当前进程。

sched_getparam()和sched_setparam()系统调用

sched_getparam()系统调用为pid 所表示的进程检索调度参数。如果pid是0, current进程的参数被检索。正如你所期望的, 相应的sys_sched_getparam()服务例程找到与pid相关的进程描述符指针, 把它的rt_priority字段存放在类型为sched_param的局部变量中, 并调用 copy_to_user()把它拷贝到进程地址空间中由param参数指定的地址。

`sched_setparam()` 系统调用类似于 `sched_setscheduler()`，它与后者的不同在于不让调用者设置 `policy` 字段的值。(注4)⁵相应的 `sys_sched_setparam()` 服务例程几乎与 `sys_sched_setscheduler()` 相同的参数调用 `do_sched_setscheduler()`

`sched_yield()` 系统调用

`sched_yield()` 系统调用允许进程在未被挂起的情况下自愿放弃CPU，进程仍然处于 `TASK_RUNNING` 状态，但调度程序把它放在运行队列的过期进程集合中（如果进程是普通进程），或放在运行队列链表的末尾（如果进程是实时进程）。随后调用 `schedule()` 函数，在这种方式下，具有相同动态优先级的其他进程将有机会运行。这个调用主要由 `SCHED_FIFO` 实时进程使用。

`sched_get_priority_min()` 和 `sched_get_priority_max()` 系统调用

`sched_get_priority_min()` 和 `sched_get_priority_max()` 系统调用分别返回最小和最大实时静态优先级的值，这个值由 `policy` 参数所标识的调度策略来使用。

如果 `current` 是实时进程，`sys_sched_get_priority_min()` 服务例程返回1，否则返回0。

如果 `current` 是实时进程，`sys_sched_get_priority_max()` 服务例程返回99（最高优先级），否则返回0。

`sched_rr_get_interval()` 系统调用

`sched_rr_get_interval()` 系统调用把参数 `pid` 表示的实时进程的轮转时间片写入用户态地址空间的一个结构中。如果 `pid` 等于0，系统调用就写当前进程的时间片。

相应的 `sys_sched_rr_get_interval()` 服务例程同样调用 `find_process_by_pid()` 检索与 `pid` 相关的进程描述符。然后，把存放在所选中进程的基本时间片转换为秒数和纳秒数，并把它们拷贝到用户态的结构中，通常，`FIFO` 实时进程的时间片等于0。

⁵ 注4：POSIX 标准的一个特殊要求造成了这种异常情况