第 2 章 进程管理和调度

所有的现代操作系统都能够同时运行若干进程，至少用户错觉上是这样。如果系统只有一个处理器，那么在给定时刻只有一个程序可以运行。在多处理器系统中，可以真正并行运行的进程数目，取决于物理CPU的数目。

内核和处理器建立了多任务的错觉，即可以并行做几种操作，这是通过以很短的间隔在系统运行的应用程序之间不停切换而做到的。由于切换间隔如此之短，使得用户无法注意到短时间内的停滞，从而在感观上觉得计算机能够同时做几件事情。

这种系统管理方式引起了几个问题，内核必须解决这些问题，其中最重要的问题如下所示。

- 除非明确地要求，否则应用程序不能彼此干扰。例如，应用程序A的错误不能传播到应用程序B。由于Linux是一个多用户系统，它也必须确保程序不能读取或修改其他程序的内存，否则就很容易访问其他用户的私有数据。
- CPU时间必须在各种应用程序之间尽可能公平地共享，其中一些程序可能比其他程序更重要。

第一个需求——存储保护，将在第3章处理。在本章中，我主要讲解内核共享CPU时间的方法，以及如何在进程之间切换。这里有两个任务，其执行是相对独立的。

- 内核必须决定为各个进程分配多长时间，何时切换到下一个进程。这又引出了哪个进程是下一个的问题。此类决策是平台无关的。
- 在内核从进程A切换到进程B时，必须确保进程B的执行环境与上一次撤销其处理器资源时完全相同。例如，处理器寄存器的内容和虚拟地址空间的结构必须与此前相同。

这里的后一项工作与处理器极度相关。不能只用C语言实现，还需要汇编代码的帮助。

这两个任务是称之为*调度器*的内核子系统的职责。CPU时间如何分配取决于*调度器策略*，这与用于在各个进程之间切换的*任务切换*机制完全无关。

2.1 进程优先级

并非所有进程都具有相同的重要性。除了大多数读者熟悉的进程优先级之外，进程还有不同的关键度类别，以满足不同需求。首先进行比较粗糙的划分，进程可以分为实时进程和非实时进程。

- *硬实时进程*有严格的时间限制，某些任务必须在指定的时限内完成。如果飞机的飞行控制命令通过计算机处理，则必须尽快处理发送，即保证在确定的一段时间内完成。 例如，如果飞机处于着陆进场过程中，而飞行员想要拉起机头。如果计算机在几秒以后发送该命令，则什么用也没有！此时只能考虑飞机的后事了——一头扎到地上。硬实时进程的关键特征是，它们必须在可保证的时间范围内得到处理。请注意，这并不意味着所要求的时间范围特别短，而是系统必须保证决不会超过某一时间范围，即使在不大可能或条件不利的情况下也是如此。

Linux不支持硬实时处理，至少在主流的内核中不支持。但有一些修改版本如RTLinux、Xenomai、RATI提供了该特性。在这些修改后的方案中，Linux内核作为独立的“进程”运行来处理次重要的软件，而实时的工作则在内核外部完成。只有当没有实时的关键操作执行时，内核才会运行。

由于Linux是针对吞吐量优化，试图尽快地处理常见情形，其实很难实现可保证的响应时间。2007年我们在降低内核整体延迟（指向内核发出请求到完成之间的时间间隔）方面取得了相当多的进展。相关工作包括：可抢占的内核机制、实时互斥量以及本书将要讨论的完全公平的新调度器。

- 软实时进程是硬实时进程的一种弱化形式。尽管仍然需要快速得到结果，但稍微晚一点不会造成世界末日。软实时进程的一个例子是对CD的写入操作。CD写入进程接收的数据必须保持某一速率，因为数据是以连续流的形式写入介质的。如果系统负荷过高，数据流可能会暂时中断，这可能导致CD不可用，但比坠机好得多。不过，写入进程在需要CPU时间时应该能够得到保证，至少优先于所有其他普通进程。
- 大多数进程是没有特定时间约束的普通进程，但仍然可以根据重要性来分配优先级。

  例如，冗长的编译或计算只需要极低的优先级，因为计算偶而中断一两秒根本不会有什么后果，用户不太可能注意到。相比之下，交互式应用则应该尽快响应用户命令，因为用户很容易不耐烦。

图2-1给出了CPU时间分配的一个简图。进程的运行按时间片调度，分配给进程的时间片份额与其相对重要性相当。系统中时间的流动对应于圆盘的转动，而CPU则由圆周旁的“扫描器”表示。最终效果是，尽管所有的进程都有机会运行，但重要的进程会比次要的得到更多的CPU时间。

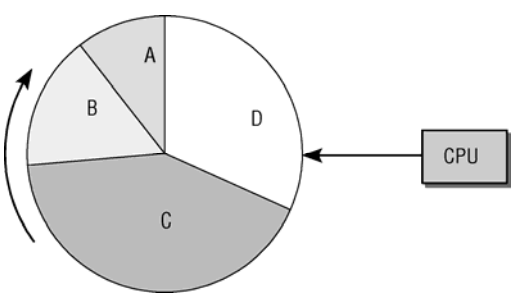

图2-1 通过时间片分配CPU时间

这种方案称之为抢占式多任务处理（preemptive multitasking），各个进程都分配到一定的时间段可以执行。时间段到期后，内核会从进程收回控制权，让一个不同的进程运行，而不考虑前一进程所执行的上一个任务。被抢占进程的运行时环境，即所有CPU寄存器的内容和页表，都会保存起来，因此其执行结果不会丢失。在该进程恢复执行时，其进程环境可以完全恢复。时间片的长度会根据进程重要性（以及因此而分配的优先级）的不同而变化。图2-1中分配给各个进程的时间片长度各有不同，即说明了这一点。

这种简化模型没有考虑几个重要问题。例如，进程在某些时间可能因为无事可做而无法立即执行。为使CPU时间的利益回报尽可能最大化，这样的进程决不能执行。这种情况在图2-1中看不出来，因为其中假定所有的进程都是可以立即运行的。另外一个忽略的事实是Linux支持不同的调度类别（在进程之间完全公平的调度和实时调度），调度时也必须考虑到这一点。此外，在有重要的进程变为就绪状态可以运行时，有一种选项是抢占当前的进程，图中也没有反映出这一点。

注意，进程调度在内核开发者之间引起了非常热烈的讨论，尤其是提到挑选最合适的算法时。为调度器的质量确立一种定量标准，可以讲，即使可能，也非常困难。另外调度器要满足Linux系统上许多不同工作负荷所提出的需求，这是非常具有挑战性的。自动化控制所需的小型嵌入式系统和大型计算机的需求非常不同，而多媒体系统的需求与前两者也颇为不同。实际上，调度器的代码近年来已经重写了两次。

(1) 在2.5系列内核开发期间，所谓的O(1)调度器代替了前一个调度器。该调度器一个特别的性质是，它可以在常数时间内完成其工作，不依赖于系统上运行的进程数目。该设计从根本上打破了先前使用的调度体系结构。

(2) 完全公平调度器（completely fair scheduler）在内核版本2.6.23开发期间合并进来。新的代码再一次完全放弃了原有的设计原则，例如，前一个调度器中为确保用户交互任务响应快速，需要许多启发式原则。该调度器的关键特性是，它试图尽可能地模仿理想情况下的公平调度。此外，它不仅可以调度单个进程，还能够处理更一般性的调度实体（scheduling entity）。例如，该调度器分配可用时间时，可以首先在不同用户之间分配，接下来在各个用户的进程之间分配。

我会在下文讨论该调度器的实现细节。

在关注内核如何实现调度之前，我们首先来讨论进程可能拥有的状态。

2.2　进程生命周期

进程并不总是可以立即运行。有时候它必须等待来自外部信号源、不受其控制的事件，例如在文本编辑器中等待键盘输入。在事件发生之前，进程无法运行。

当调度器在进程之间切换时，必须知道系统中每个进程的状态。将CPU时间分配到无事可做的进程，显然是没有意义的。进程在各个状态之间的转换也同样重要。例如，如果一个进程在等待来自外设的数据，那么调度器的职责是一旦数据已经到达，则需要将进程的状态由等待改为可运行。

进程可能有以下几种状态。

- **运行**：该进程此刻正在执行。
- **等待**：进程能够运行，但没有得到许可，因为CPU分配给另一个进程。调度器可以在下一次任务切换时选择该进程。
- **睡眠**：进程正在睡眠无法运行，因为它在等待一个外部事件。调度器无法在下一次任务切换时选择该进程。

系统将所有进程保存在一个进程表中，无论其状态是运行、睡眠或等待。但睡眠进程会特别标记出来，调度器会知道它们无法立即运行（具体实现，请参考2.3节）。睡眠进程会分类到若干队列中，因此它们可在适当的时间唤醒，例如在进程等待的外部事件已经发生时。

图2-2描述了进程的几种状态及其转换。

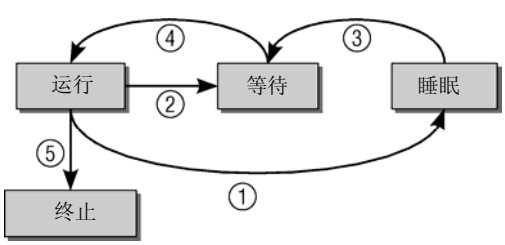

图2-2　进程状态之间的转换

对于一个排队中的可运行进程，我们来考察其各种可能的状态转换。该进程已经就绪，但没有运行，因为CPU分配给了其他进程（因此该进程的状态是“等待”）。在调度器授予CPU时间之前，进程会一直保持该状态。在分配CPU时间之后，其状态改变为“运行”（路径④）。

在调度器决定从该进程收回CPU资源时（可能的原因稍后讲述），过程状态从“运行”改变为“等待”（路径②），循环重新开始。实际上根据是否可以被信号中断，有两种“睡眠”状态。现在这种差别还不重要，但在更仔细地考察具体实现时，其差别就相对重要了。

如果进程必须等待事件，则其状态从“运行”改变为“睡眠”（路径①）。但进程状态无法从“睡眠”直接改变为“运行”。在所等待的事件发生后，进程先变回到“等待”状态（路径③），然后重新回到正常循环。

在程序执行终止（例如，用户关闭应用程序）后，过程状态由“运行”变为“终止”（路径⑤）。

上文没有列出的一个特殊的进程状态是所谓的“僵尸”状态。顾名思义，这样的进程已经死亡，但仍然以某种方式活着。实际上，说这些进程死了，是因为其资源（内存、与外设的连接，等等）已经释放，因此它们无法也决不会再次运行。说它们仍然活着，是因为进程表中仍然有对应的表项。

僵尸是如何产生的？其原因在于UNIX操作系统下进程创建和销毁的方式。在两种事件发生时，程序将终止运行。第一，程序必须由另一个进程或一个用户杀死（通常是通过发送SIGTERM或SIGKILL信号来完成，这等价于正常地终止进程）；进程的父进程在子进程终止时必须调用或已经调用wait4（读做wait for）系统调用。 这相当于向内核证实父进程已经确认子进程的终结。该系统调用使得内核可以释放为子进程保留的资源。

只有在第一个条件发生（程序终止）而第二个条件不成立的情况下（wait4），才会出现“僵尸”状态。在进程终止之后，其数据尚未从进程表删除之前，进程总是暂时处于“僵尸”状态。有时候（例如，如果父进程编程极其糟糕，没有发出wait调用），僵尸进程可能稳定地寄身于进程表中，直至下一次系统重启。从进程工具（如ps或top）的输出，可以看到僵尸进程。因为残余的数据在内核中占据的空间极少，所有这几乎不是一个问题。

抢占式多任务处理

Linux进程管理的结构中还需要另外两种进程状态选项：用户状态和核心态。这反映了所有现代CPU都有（至少）两种不同执行状态的事实，其中一种具有无限的权利，而另一种则受到各种限制。例如，可能禁止访问某些内存区域。这种区别是建立封闭“隔离罩”的一个重要前提，它维持着系统中现存的各个进程，防止它们与系统其他部分相互干扰。

进程通常都处于用户状态，只能访问自身的数据，无法干扰系统中的其他应用程序，甚至也不会注意到自身之外其他程序的存在。

如果进程想要访问系统数据或功能（后者管理着所有进程之间共享的资源，例如文件系统空间），则必须切换到核心态。显然这只能在受控情况下完成，否则所有建立的保护机制都是多余的，而且这种访问必须经由明确定义的路径。第1章简要提到“系统调用”是在状态之间切换的一种方法。第13章深入讨论了系统调用的实现。

从用户状态切换到核心态的第二种方法是通过中断，此时切换是自动触发的。系统调用是由用户应用程序有意调用的，中断则不同，其发生或多或少是不可预测的。处理中断的操作，通常与中断发生时执行的进程无关。例如，外部块设备向内存传输数据完毕会引发一个中断，但相关数据用于系统中运行的任何进程都是可能的。类似地，进入系统的网络数据包也是通过中断通知的。显然，该数据包也未必是用于当前运行的进程。因此，在Linux执行中断操作时，当前运行的进程不会察觉。

内核的抢占调度模型建立了一个层次结构，用于判断哪些进程状态可以由其他状态抢占。

- 普通进程总是可能被抢占，甚至是由其他进程抢占。在一个重要进程变为可运行时，例如编辑器接收到了等待已久的键盘输入，调度器可以决定是否立即执行该进程，即使当前进程仍然在正常运行。对于实现良好的交互行为和低系统延迟，这种抢占起到了重要作用。
- 如果系统处于核心态并正在处理系统调用，那么系统中的其他进程是无法夺取其CPU时间的。调度器必须等到系统调用执行结束，才能选择另一个进程执行，但中断可以中止系统调用。[①]
- 中断可以暂停处于用户状态和核心态的进程。中断具有最高优先级，因为在中断触发后需要尽快处理。

在内核2.5开发期间，一个称之为内核抢占（kernel preemption）的选项添加到内核。该选项支持在紧急情况下切换到另一个进程，甚至当前是处于核心态执行系统调用（中断处理期间是不行的）。尽管内核会试图尽快执行系统调用，但对于依赖恒定数据流的应用程序来说，系统调用所需的时间仍然太长了。内核抢占可以减少这样的等待时间，因而保证“更平滑的”程序执行。但该特性的代价是增加内核的复杂度，因为接下来有许多数据结构需要针对并发访问进行保护，即使在单处理器系统上也是如此。2.8.3节会讨论该技术。

2.3 进程表示

Linux内核涉及进程和程序的所有算法都围绕一个名为task_struct的数据结构建立，该结构定义在include/sched.h中。这是系统中主要的一个结构。在阐述调度器的实现之前，了解一下Linux管理进程的方式是很有必要的。

task_struct包含很多成员，将进程与各个内核子系统联系起来，下文会逐一讨论。此外，如果没有比较详细的知识，我们就很难解释某些结构成员的重要性，因此下面会频繁引用后续章节。

task_struct定义如下，当然，这里是简化版本：

<sched.h>

```
struct task_struct {
        volatile long state;          /* -1表示不可运行，0表示可运行，>0表示停止 */
        void *stack;
        atomic_t usage;
        unsigned long flags;          /* 每进程标志，下文定义 */
        unsigned long ptrace;
        int lock_depth;               /* 大内核锁深度 */

        int prio, static_prio, normal_prio;
        struct list_head run_list;
        const struct sched_class *sched_class;
        struct sched_entity se;

        unsigned short ioprio;

        unsigned long policy;
        cpumask_t cpus_allowed;
        unsigned int time_slice;

#if defined(CONFIG_SCHEDSTATS) || defined(CONFIG_TASK_DELAY_ACCT)
        struct sched_info sched_info;
#endif
```

① 在进行重要的内核操作时，可以停用几乎所有的中断。

```
        struct list_head tasks;
        /*
        * ptrace_list/ptrace_children链表是ptrace能够看到的当前进程的子进程列表。

        */
        struct list_head ptrace_children;

        struct list_head ptrace_list;

        struct mm_struct *mm, *active_mm;

/* 进程状态 */
        struct linux_binfmt *binfmt;
        long exit_state;
        int exit_code, exit_signal;
        int pdeath_signal; /* 在父进程终止时发送的信号 */

        unsigned int personality;
        unsigned did_exec:1;
        pid_t pid;
        pid_t tgid;
        /*
         * 分别是指向（原）父进程、最年轻的子进程、年幼的兄弟进程、年长的兄弟进程的指针。
         *（p->father可以替换为p->parent->pid）
         */
        struct task_struct *real_parent;    /* 真正的父进程（在被调试的情况下） */
        struct task_struct *parent;         /* 父进程 */
        /*
         * children/sibling链表外加当前调试的进程，构成了当前进程的所有子进程
         */
        struct list_head children;          /* 子进程链表 */
        struct list_head sibling;           /* 连接到父进程的子进程链表 */
        struct task_struct *group_leader;   /* 线程组组长 */

        /* PID与PID散列表的联系。 */
        struct pid_link pids[PIDTYPE_MAX];
        struct list_head thread_group;

        struct completion *vfork_done;      /* 用于vfork() */
        int __user *set_child_tid;          /* CLONE_CHILD_SETTID */
        int __user *clear_child_tid;        /* CLONE_CHILD_CLEARTID */

        unsigned long rt_priority;
        cputime_t utime, stime, utimescaled, stimescaled;;
        unsigned long nvcsw, nivcsw;        /* 上下文切换计数 */
        struct timespec start_time;         /* 单调时间 */
        struct timespec real_start_time;    /* 启动以来的时间 */
        /* 内存管理器失效和页交换信息，这个有一点争论。它既可以看作是特定于内存管理器的，
           也可以看作是特定于线程的 */
        unsigned long min_flt, maj_flt;

        cputime_t it_prof_expires, it_virt_expires;
        unsigned long long it_sched_expires;
        struct list_head cpu_timers[3];

/* 进程身份凭据 */
        uid_t uid,euid,suid,fsuid;
        gid_t gid,egid,sgid,fsgid;
        struct group_info *group_info;
        kernel_cap_t cap_effective, cap_inheritable, cap_permitted;

        unsigned keep_capabilities:1;
        struct user_struct *user;
```

```
        char comm[TASK_COMM_LEN]; /* 除去路径后的可执行文件名称
                                     -用[gs]et_task_comm访问（其中用task_lock()锁定它）
                                     -通常由flush_old_exec初始化 */
/* 文件系统信息 */
        int link_count, total_link_count;
/* ipc相关 */
        struct sysv_sem sysvsem;
/* 当前进程特定于CPU的状态信息 */
        struct thread_struct thread;
/* 文件系统信息 */
        struct fs_struct *fs;
/* 打开文件信息 */
        struct files_struct *files;
/* 命名空间 */
        struct nsproxy *nsproxy;
/* 信号处理程序 */
        struct signal_struct *signal;
        struct sighand_struct *sighand;

        sigset_t blocked, real_blocked;
        sigset_t saved_sigmask; /* 用TIF_RESTORE_SIGMASK恢复 */
        struct sigpending pending;

        unsigned long sas_ss_sp;
        size_t sas_ss_size;
        int (*notifier)(void *priv);
        void *notifier_data;
        sigset_t *notifier_mask;

#ifdef CONFIG_SECURITY
        void *security;
#endif

/* 线程组跟踪 */
        u32 parent_exec_id;
        u32 self_exec_id;

/* 日志文件系统信息 */
        void *journal_info;

/* 虚拟内存状态 */
        struct reclaim_state *reclaim_state;

        struct backing_dev_info *backing_dev_info;

        struct io_context *io_context;

        unsigned long ptrace_message;
        siginfo_t *last_siginfo; /* 由ptrace使用。*/
...
};
```

要弄清楚该结构中信息的数量诚然很困难。但该结构的内容可以分解为各个部分，每个部分表示进程的一个特定方面。

- 状态和执行信息，如待决信号、使用的二进制格式（和其他系统二进制格式的任何仿真信息）、进程ID号（pid）、到父进程及其他有关进程的指针、优先级和程序执行有关的时间信息（例如CPU时间）。
- 有关已经分配的虚拟内存的信息。

- 进程身份凭据，如用户ID、组ID以及权限[①]等。可使用系统调用查询（或修改）这些数据。在描述相关的特定子系统时，我会更详细地阐述。
- 使用的文件包含程序代码的二进制文件，以及进程所处理的所有文件的文件系统信息，这些都必须保存下来。
- 线程信息记录该进程特定于CPU的运行时间数据（该结构的其余字段与所使用的硬件无关）。
- 在与其他应用程序协作时所需的进程间通信有关的信息。
- 该进程所用的信号处理程序，用于响应到来的信号。

task_struct的许多成员并非简单类型变量，而是指向其他数据结构的指针，相关数据结构会在后续章节中讨论。在本章中，我会阐述task_struct中对进程管理的实现特别重要的一些成员。

state指定了进程的当前状态，可使用下列值（这些是预处理器常数，定义在<sched.h>中）。

- TASK_RUNNING意味着进程处于可运行状态。这并不意味着已经实际分配了CPU。进程可能会一直等到调度器选中它。该状态确保进程可以立即运行，而无需等待外部事件。
- TASK_INTERRUPTIBLE是针对等待某事件或其他资源的睡眠进程设置的。在内核发送信号给该进程表明事件已经发生时，进程状态变为TASK_RUNNING，它只要调度器选中该进程即可恢复执行。
- TASK_UNINTERRUPTIBLE用于因内核指示而停用的睡眠进程。它们不能由外部信号唤醒，只能由内核亲自唤醒。
- TASK_STOPPED表示进程特意停止运行，例如，由调试器暂停。
- TASK_TRACED本来不是进程状态，用于从停止的进程中，将当前被调试的那些（使用ptrace机制）与常规的进程区分开来。

下列常量既可以用于struct task_struct的进程状态字段，也可以用于exit_state字段，后者明确地用于退出进程。

- EXIT_ZOMBIE如上所述的僵尸状态。
- EXIT_DEAD状态则是指wait系统调用已经发出，而进程完全从系统移除之前的状态。只有多个线程对同一个进程发出wait调用时，该状态才有意义。

Linux提供资源限制（resource limit，rlimit）机制，对进程使用系统资源施加某些限制。该机制利用了task_struct中的rlim数组，数组项类型为struct rlimit。

```
<resource.h>
struct rlimit {
        unsigned long rlim_cur;
        unsigned long rlim_max;
}
```

上述定义设计得非常一般，因此可以用于许多不同的资源类型。

- rlim_cur是进程当前的资源限制，也称之为软限制（soft limit）。
- rlim_max是该限制的最大容许值，因此也称之为硬限制（hard limit）。

系统调用setrlimit来增减当前限制，但不能超出rlim_max指定的值。getrlimits用于检查当前限制。

rlim数组中的位置标识了受限制资源的类型，这也是内核需要定义预处理器常数，将资源与位置关联起来的原因。表2-1列出了可能的常数及其含义。关于如何最佳地运用各种限制，系统程序设计方

① 权限是授予进程的特定许可。它们使得进程可以执行某些本来只能由root进程执行的操作。

面的教科书提供了详细的说明，而setrlimit(2)的手册页详细描述了所有的限制。

表2-1 特定于进程的资源限制

常 数	语 义
RLIMIT_CPU	按毫秒计算的最大CPU时间
RLIMIT_FSIZE	允许的最大文件长度
RLIMIT_DATA	数据段的最大长度
RLIMIT_STACK	（用户状态）栈的最大长度
RLIMIT_CORE	内存转储文件的最大长度
RLIMIT_RSS	常驻内存的最大尺寸。换句话说，进程使用页帧的最大数目。目前未使用
RLIMIT_NPROC	与进程真正UID关联的用户可以拥有的进程的最大数目
RLIMIT_NOFILE	打开文件的最大数目
RLIMIT_MEMLOCK	不可换出页的最大数目
RLIMIT_AS	进程占用的虚拟地址空间的最大尺寸
RLIMIT_LOCKS	文件锁的最大数目
RLIMIT_SIGPENDING	待决信号的最大数目
RLIMIT_MSGQUEUE	信息队列的最大数目
RLIMIT_NICE	非实时进程的优先级（nice level）
RLIMIT_RTPRIO	最大的实时优先级

由于Linux试图建立与特定的本地UNIX系统之间的二进制兼容性，因此不同体系结构的数值可能不同。

因为限制涉及内核的各个不同部分，内核必须确认子系统遵守了相应限制。这也是为什么在本书以后几章里我们会屡次遇到rlimit的原因。

如果某一类资源没有使用限制(几乎所有资源的默认设置)，则将rlim_max设置为RLIM_INFINITY。例外情况包括下面所列举的。

- 打开文件的数目（RLIMIT_NOFILE，默认限制在1 024）。
- 每用户的最大进程数（RLIMIT_NPROC)，定义为max_threads/2。max_threads是一个全局变量，指定了在把八分之一可用内存用于管理线程信息的情况下，可以创建的线程数目。在计算时，提前给定了20个线程的最小可能内存用量。

init进程的限制在系统启动时即生效，定义在include/asm-generic-resource.h中的INIT_RLIMITS。

读者可以关注一下内核版本2.6.25，在本书编写时仍然在开发中，该版本的内核在proc文件系统中对每个进程都包含了对应的一个文件，这样就可以查看当前的rlimit值：

```
wolfgang@meitner> cat /proc/self/limits
Limit                     Soft Limit           Hard Limit           Units
Max cpu time              unlimited            unlimited            ms
Max file size             unlimited            unlimited            bytes
Max data size             unlimited            unlimited            bytes
Max stack size            8388608              unlimited            bytes
Max core file size        0                    unlimited            bytes
```

```
Max resident set          unlimited       unlimited       bytes
Max processes             unlimited       unlimited       processes
Max open files            1024            1024            files
Max locked memory         unlimited       unlimited       bytes
Max address space         unlimited       unlimited       bytes
Max file locks            unlimited       unlimited       locks
Max pending signals       unlimited       unlimited       signals
Max msgqueue size         unlimited       unlimited       bytes
Max nice priority         0               0
Max realtime priority     0               0
Max realtime timeout      unlimited       unlimited       us
```

内核版本2.6.24已经包含了用于生成该信息的大部分代码，但与/proc文件系统的关联可能只有后续的内核发布版本才会完成。

2.3.1 进程类型

典型的UNIX进程包括：由二进制代码组成的应用程序、单线程（计算机沿单一路径通过代码，不会有其他路径同时运行）、分配给应用程序的一组资源（如内存、文件等）。新进程是使用fork和exec系统调用产生的。

- fork生成当前进程的一个相同副本，该副本称之为*子进程*。原进程的所有资源都以适当的方式复制到子进程，因此该系统调用之后，原来的进程就有了两个独立的实例。这两个实例的联系包括：同一组打开文件、同样的工作目录、内存中同样的数据（两个进程各有一份副本），等等。此外二者别无关联。[①]
- exec从一个可执行的二进制文件加载另一个应用程序，来代替当前运行的进程。换句话说，加载了一个新程序。因为exec并不创建新进程，所以必须首先使用fork复制一个旧的程序，然后调用exec在系统上创建另一个应用程序。

上述两个调用在所有UNIX操作系统变体上都是可用的，其历史可以追溯到很久之前，除此之外Linux还提供了clone系统调用。clone的工作原理基本上与fork相同，但新进程不是独立于父进程的，而可以与其共享某些资源。可以指定需要共享和复制的资源种类，例如，父进程的内存数据、打开文件或安装的信号处理程序。

clone用于实现*线程*，但仅仅该系统调用不足以做到这一点，还需要用户空间库才能提供完整的实现。线程库的例子，有Linuxthreads和Next Generation Posix Threads等。

2.3.2 命名空间

命名空间提供了虚拟化的一种轻量级形式，使得我们可以从不同的方面来查看运行系统的全局属性。该机制类似于Solaris中的zone或FreeBSD中的jail。对该概念做一般概述之后，我将讨论命名空间框架所提供的基础设施。

1. 概念

传统上，在Linux以及其他衍生的UNIX变体中，许多资源是全局管理的。例如，系统中的所有进程按照惯例是通过PID标识的，这意味着内核必须管理一个全局的PID列表。而且，所有调用者通过uname系统调用返回的系统相关信息（包括系统名称和有关内核的一些信息）都是相同的。用户ID的

① 在2.4.1节中，读者会知道Linux使用了写时复制机制，直至新进程对内存页执行写操作才会复制内存页面，这比在执行fork时盲目地立即复制所有内存页要更高效。父子进程内存页之间的联系，只有对内核才是可见的，对应用程序是透明的。

管理方式类似，即各个用户是通过一个全局唯一的UID号标识。

全局ID使得内核可以有选择地允许或拒绝某些特权。虽然UID为0的root用户基本上允许做任何事，但其他用户ID则会受到限制。例如UID为*n*的用户，不允许杀死属于用户*m*的进程（$m \neq n$）。但这不能防止用户看到彼此，即用户*n*可以看到另一个用户*m*也在计算机上活动。只要用户只能操纵他们自己的进程，这就没什么问题，因为没有理由不允许用户看到其他用户的进程。

但有些情况下，这种效果可能是不想要的。如果提供Web主机的供应商打算向用户提供Linux计算机的全部访问权限，包括root权限在内。传统上，这需要为每个用户准备一台计算机，代价太高。使用KVM或VMWare提供的虚拟化环境是一种解决问题的方法，但资源分配做得不是非常好。计算机的各个用户都需要一个独立的内核，以及一份完全安装好的配套的用户层应用。

命名空间提供了一种不同的解决方案，所需资源较少。在虚拟化的系统中，一台物理计算机可以运行多个内核，可能是并行的多个不同的操作系统。而命名空间则只使用一个内核在一台物理计算机上运作，前述的所有全局资源都通过命名空间抽象起来。这使得可以将一组进程放置到容器中，各个容器彼此隔离。隔离可以使容器的成员与其他容器毫无关系。但也可以通过允许容器进行一定的共享，来降低容器之间的分隔。例如，容器可以设置为使用自身的PID集合，但仍然与其他容器共享部分文件系统。

本质上，命名空间建立了系统的不同视图。此前的每一项全局资源都必须包装到容器数据结构中，只有资源和包含资源的命名空间构成的二元组仍然是全局唯一的。虽然在给定容器内部资源是自足的，但无法提供在容器外部具有唯一性的ID。图2-3给出了此情况的一个概述。

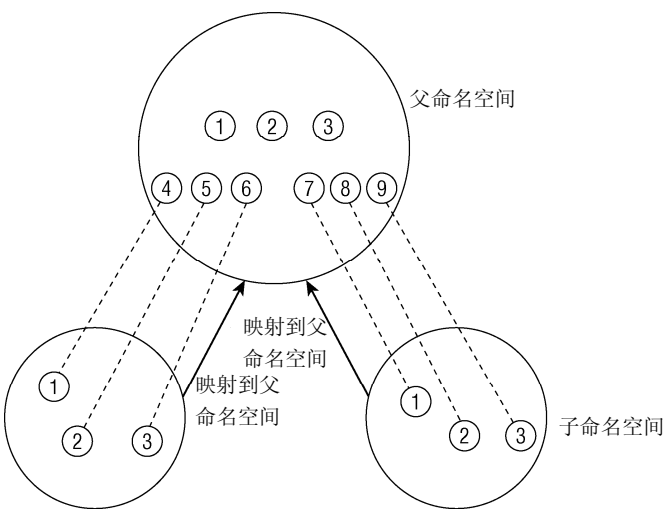

图2-3 命名空间可以按层次关联起来。每个命名空间都发源于一个父命名空间，一个父命名空间可以有多个子命名空间

考虑系统上有3个不同命名空间的情况。命名空间可以组织为层次，我会在这里讨论这种情况。一个命名空间是父命名空间，衍生了两个子命名空间。假定容器用于虚拟主机配置中，其中的每个容器必须看起来像是单独的一台Linux计算机。因此其中每一个都有自身的`init`进程，PID为0，其他进程的PID以递增次序分配。两个子命名空间都有PID为0的`init`进程，以及PID分别为2和3的两个进程。由于相同的PID在系统中出现多次，PID号不是全局唯一的。

虽然子容器不了解系统中的其他容器，但父容器知道子命名空间的存在，也可以看到其中执行的所有进程。图中子容器的进程映射到父容器中，PID为4到9。尽管系统上有9个进程，但却需要15个PID来表示，因为一个进程可以关联到多个PID。至于哪个PID是“正确”的，则依赖于具体的上下文。

如果命名空间包含的是比较简单的量，也可以是非层次的，例如下文讨论的UTS命名空间。在这种情况下，父子命名空间之间没有联系。

请注意，Linux系统对简单形式的命名空间的支持已经有很长一段时间了，主要是chroot系统调用。该方法可以将进程限制到文件系统的某一部分，因而是一种简单的命名空间机制。但真正的命名空间能够控制的功能远远超过文件系统视图。

新的命名空间可以用下面两种方法创建。

(1) 在用fork或clone系统调用创建新进程时，有特定的选项可以控制是与父进程共享命名空间，还是建立新的命名空间。

(2) unshare系统调用将进程的某些部分从父进程分离，其中也包括命名空间。更多信息请参见手册页unshare(2)。

在进程已经使用上述的两种机制之一从父进程命名空间分离后，从该进程的角度来看，改变全局属性不会传播到父进程命名空间，而父进程的修改也不会传播到子进程，至少对于简单的量是这样。而对于文件系统来说，情况就比较复杂，其中的共享机制非常强大，带来了大量的可能性，具体的情况会在第8章讨论。

在标准内核中命名空间当前仍然标记为试验性的，为使内核的所有部分都能够感知到命名空间，相关开发仍然在进行中。但就内核版本2.6.24而言，基本的框架已经建立就绪。[①]当前的实现仍然存在一些问题，相关的信息可以参见Documentation/namespaces/compatibility-list.txt文件。

2. 实现

命名空间的实现需要两个部分：每个子系统的命名空间结构，将此前所有的全局组件包装到命名空间中；将给定进程关联到所属各个命名空间的机制。图2-4说明了具体情形。

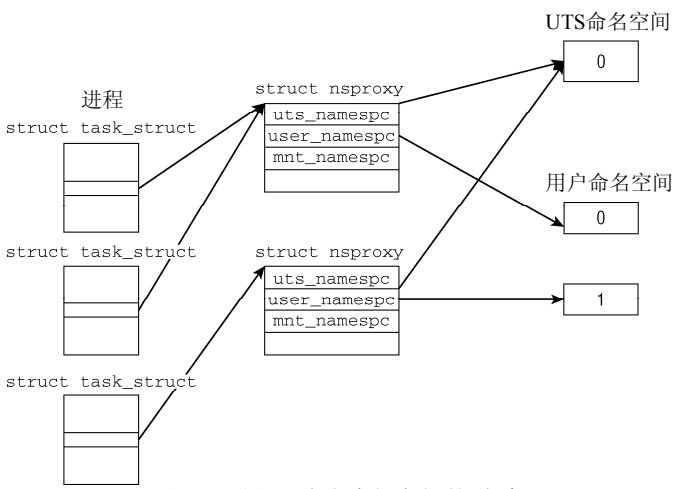

图2-4 进程和命名空间之间的联系

① 但这并不意味着相关实现是最近开发的。实际上，该方法已经用于产品系统多年，但一直以外部内核补丁的形式存在。

子系统此前的全局属性现在封装到命名空间中，每个进程关联到一个选定的命名空间。每个可以感知命名空间的内核子系统都必须提供一个数据结构，将所有通过命名空间形式提供的对象集中起来。struct nsproxy用于汇集指向特定于子系统的命名空间包装器的指针：

<nsproxy.h>

```
struct nsproxy {
        atomic_t count;
        struct uts_namespace *uts_ns;
        struct ipc_namespace *ipc_ns;
        struct mnt_namespace *mnt_ns;
        struct pid_namespace *pid_ns;
        struct user_namespace *user_ns;
        struct net *net_ns;
};
```

当前内核的以下范围可以感知到命名空间。

- UTS命名空间包含了运行内核的名称、版本、底层体系结构类型等信息。UTS是UNIX Timesharing System的简称。
- 保存在struct ipc_namespace中的所有与进程间通信（IPC）有关的信息。
- 已经装载的文件系统的视图，在struct mnt_namespace中给出。
- 有关进程ID的信息，由struct pid_namespace提供。
- struct user_namespace保存的用于限制每个用户资源使用的信息。
- struct net_ns包含所有网络相关的命名空间参数。读者在第12章中会看到，为使网络相关的内核代码能够完全感知命名空间，还有许多工作需要完成。

当我讨论相应的子系统时，会介绍各个命名空间容器的内容。在本章中，我们主要讲解UTS和用户命名空间。由于在创建新进程时可使用fork建立一个新的命名空间，因此必须提供控制该行为的适当的标志。每个命名空间都有一个对应的标志：

<sched.h>

```
#define CLONE_NEWUTS     0x04000000       /* 创建新的utsname组  */
#define CLONE_NEWIPC     0x08000000       /* 创建新的IPC命名空间  */
#define CLONE_NEWUSER    0x10000000       /* 创建新的用户命名空间 */
#define CLONE_NEWPID     0x20000000       /* 创建新的PID命名空间  */
#define CLONE_NEWNET     0x40000000       /* 创建新的网络命名空间 */
```

每个进程都关联到自身的命名空间视图：

<sched.h>

```
struct task_struct {
...
/* 命名空间 */

        struct nsproxy *nsproxy;
...
}
```

因为使用了指针，多个进程可以共享一组子命名空间。这样，修改给定的命名空间，对所有属于该命名空间的进程都是可见的。

请注意，对命名空间的支持必须在编译时启用，而且必须逐一指定需要支持的命名空间。但对命名空间的一般性支持总是会编译到内核中。 这使得内核不管有无命名空间，都不必使用不同的代码。除非指定不同的选项，否则每个进程都会关联到一个默认命名空间，这样可感知命名空间的代码总是可以使用。但如果内核编译时没有指定对具体命名空间的支持，默认命名空间的作用则类似于不启用

命名空间，所有的属性都相当于全局的。

init_nsproxy定义了初始的全局命名空间，其中维护了指向各子系统初始的命名空间对象的指针：

<kernel/nsproxy.c>

```
struct nsproxy init_nsproxy = INIT_NSPROXY(init_nsproxy);
```

<init_task.h>

```
#define INIT_NSPROXY(nsproxy) { \
        .pid_ns = &init_pid_ns, \
        .count = ATOMIC_INIT(1), \
        .uts_ns = &init_uts_ns, \
        .mnt_ns = NULL, \
        INIT_NET_NS(net_ns) \
        INIT_IPC_NS(ipc_ns) \
        .user_ns = &init_user_ns, \
}
```

- UTS命名空间

UTS命名空间几乎不需要特别的处理，因为它只需要简单量，没有层次组织。所有相关信息都汇集到下列结构的一个实例中：

<utsname.h>

```
struct uts_namespace {
        struct kref kref;
        struct new_utsname name;

};
```

kref是一个嵌入的引用计数器，可用于跟踪内核中有多少地方使用了struct uts_namespace的实例（回想第1章，其中讲述了更多有关处理引用计数的一般框架信息）。uts_namespace所提供的属性信息本身包含在struct new_utsname中：

<utsname.h>

```
struct new_utsname {
        char sysname[65];
        char nodename[65];
        char release[65];
        char version[65];
        char machine[65];
        char domainname[65];
};
```

各个字符串分别存储了系统的名称（Linux...）、内核发布版本、机器名，等等。使用uname工具可以取得这些属性的当前值，也可以在/proc/sys/kernel/中看到：

```
wolfgang@meitner> cat /proc/sys/kernel/ostype
Linux
wolfgang@meitner> cat /proc/sys/kernel/osrelease
2.6.24
```

初始设置保存在init_uts_ns中：

init/version.c

```
struct uts_namespace init_uts_ns = {
...
        .name = {
                .sysname = UTS_SYSNAME,
                .nodename = UTS_NODENAME,
                .release = UTS_RELEASE,
                .version = UTS_VERSION,
```

```
                .machine = UTS_MACHINE,
                .domainname = UTS_DOMAINNAME,
        },
};
```

相关的预处理器常数在内核中各处定义。例如，UTS_RELEASE在<utsrelease.h>中定义，该文件是连编时通过顶层Makefile动态生成的。

请注意，UTS结构的某些部分不能修改。例如，把sysname换成Linux以外的其他值是没有意义的，但改变机器名是可以的。

内核如何创建一个新的UTS命名空间呢？这属于copy_utsname函数的职责。在某个进程调用fork并通过CLONE_NEWUTS标志指定创建新的UTS命名空间时，则调用该函数。在这种情况下，会生成先前的uts_namespace实例的一份副本，当前进程的nsproxy实例内部的指针会指向新的副本。如此而已!由于在读取或设置UTS属性值时，内核会保证总是操作特定于当前进程的uts_namespace实例，在当前进程修改UTS属性不会反映到父进程，而父进程的修改也不会传播到子进程。

● *用户命名空间*

用户命名空间在数据结构管理方面类似于UTS：在要求创建新的用户命名空间时，则生成当前用户命名空间的一份副本，并关联到当前进程的nsproxy实例。但用户命名空间自身的表示要稍微复杂一些：

<user_namespace.h>

```
struct user_namespace {
        struct kref kref;
        struct hlist_head uidhash_table[UIDHASH_SZ];
        struct user_struct *root_user;
};
```

如前所述，kref是一个引用计数器，用于跟踪多少地方需要使用user_namespace实例。对命名空间中的每个用户，都有一个struct user_struct的实例负责记录其资源消耗，各个实例可通过散列表uidhash_table访问。

对我们来说user_struct的精确定义是无关紧要的。只要知道该结构维护了一些统计数据（如进程和打开文件的数目）就足够了。我们更感兴趣的问题是：每个用户命名空间对其用户资源使用的统计，与其他命名空间完全无关，对root用户的统计也是如此。这是因为在克隆一个用户命名空间时，为当前用户和root都创建了新的user_struct实例：

kernel/user_namespace.c

```
static struct user_namespace *clone_user_ns(struct user_namespace *old_ns)
{
        struct user_namespace *ns;
        struct user_struct *new_user;
...
        ns = kmalloc(sizeof(struct user_namespace), GFP_KERNEL);
...
        ns->root_user = alloc_uid(ns, 0);

        /* 将current->user替换为新的 */
        new_user = alloc_uid(ns, current->uid);

        switch_uid(new_user);
        return ns;
}
```

alloc_uid是一个辅助函数，对当前命名空间中给定UID的一个用户，如果该用户没有对应的

user_struct实例，则分配一个新的实例。在为root和当前用户分别设置了user_struct实例后，switch_uid确保从现在开始将新的user_struct实例用于资源统计。实质上就是将struct task_struct的user成员指向新的user_struct实例。

请注意，如果内核编译时未指定支持用户命名空间，那么复制用户命名空间实际上是空操作，即总是会使用默认的命名空间。

2.3.3 进程 ID 号

UNIX进程总是会分配一个号码用于在其命名空间中唯一地标识它们。该号码被称作进程ID号，简称PID。用fork或clone产生的每个进程都由内核自动地分配了一个新的唯一的PID值。

1. 进程ID

但每个进程除了PID这个特征值之外，还有其他的ID。有下列几种可能的类型。

- 处于某个线程组（在一个进程中，以标志CLONE_THREAD来调用clone建立的该进程的不同的执行上下文，我们在后文会看到）中的所有进程都有统一的线程组ID（TGID）。如果进程没有使用线程，则其PID和TGID相同。

  线程组中的主进程被称作组长（group leader）。通过clone创建的所有线程的task_struct的group_leader成员，会指向组长的task_struct实例。
- 另外，独立进程可以合并成进程组（使用setpgrp系统调用）。进程组成员的task_struct的pgrp属性值都是相同的，即进程组组长的PID。进程组简化了向组的所有成员发送信号的操作，这对于各种系统程序设计应用（参见系统程序设计方面的文献，例如［SR05］）是有用的。请注意，用管道连接的进程包含在同一个进程组中。
- 几个进程组可以合并成一个会话。会话中的所有进程都有同样的会话ID，保存在task_struct的session成员中。SID可以使用setsid系统调用设置。它可以用于终端程序设计，但和我们这里的讨论不相干。

命名空间增加了PID管理的复杂性。回想一下，PID命名空间按层次组织。在建立一个新的命名空间时，该命名空间中的所有PID对父命名空间都是可见的，但子命名空间无法看到父命名空间的PID。但这意味着某些进程具有多个PID，凡可以看到该进程的命名空间，都会为其分配一个PID。 这必须反映在数据结构中。我们必须区分局部ID和全局ID。

- 全局ID是在内核本身和初始命名空间中的唯一ID号，在系统启动期间开始的init进程即属于初始命名空间。对每个ID类型，都有一个给定的全局ID，保证在整个系统中是唯一的。
- 局部ID属于某个特定的命名空间，不具备全局有效性。对每个ID类型，它们在所属的命名空间内部有效，但类型相同、值也相同的ID可能出现在不同的命名空间中。

全局PID和TGID直接保存在task_struct中，分别是task_struct的pid和tgid成员：

```
<sched.h>
struct task_struct {
...
        pid_t pid;
        pid_t tgid;
...
}
```

这两项都是pid_t类型，该类型定义为__kernel_pid_t，后者由各个体系结构分别定义。通常定义为int，即可以同时使用2^{32}个不同的ID。

会话和进程组ID不是直接包含在task_struct本身中，但保存在用于信号处理的结构中。task_

struct->signal->__session表示全局SID，而全局PGID则保存在task_struct->signal->__pgrp。辅助函数set_task_session和set_task_pgrp可用于修改这些值。

2. 管理PID

除了这两个字段之外，内核还需要找一个办法来管理所有命名空间内部的局部量，以及其他ID（如TID和SID）。这需要几个相互连接的数据结构，以及许多辅助函数，并将在下文讨论。

- *数据结构*

下文我将使用ID指代提到的任何进程ID。在必要的情况下，我会明确地说明ID类型（例如，TGID，即线程组ID）。

一个小型的子系统称之为PID分配器（pid allocator）用于加速新ID的分配。此外，内核需要提供辅助函数，以实现通过ID及其类型查找进程的task_struct的功能，以及将ID的内核表示形式和用户空间可见的数值进行转换的功能。

在介绍表示ID本身所需的数据结构之前，我需要讨论PID命名空间的表示方式。我们所需查看的代码如下所示：

<pid_namespace.h>

```
struct pid_namespace {
...
        struct task_struct *child_reaper;
...
        int level;
        struct pid_namespace *parent;
};
```

实际上PID分配器也需要依靠该结构的某些部分来连续生成唯一ID，但我们目前对此无需关注。我们上述代码中给出的下列成员更感兴趣。

- 每个PID命名空间都具有一个进程，其发挥的作用相当于全局的init进程。init的一个目的是对孤儿进程调用wait4，命名空间局部的init变体也必须完成该工作。child_reaper保存了指向该进程的task_struct的指针。
- parent是指向父命名空间的指针，层次表示当前命名空间在命名空间层次结构中的深度。初始命名空间的level为0，该命名空间的子空间level为1，下一层的子空间level为2，依次递推。level的计算比较重要，因为level较高的命名空间中的ID，对level较低的命名空间来说是可见的。从给定的level设置，内核即可推断进程会关联到多少个ID。

回想图2-3的内容，命名空间是按层次关联的。这有助于理解上述的定义。

PID的管理围绕两个数据结构展开：struct pid是内核对PID的内部表示，而struct upid则表示特定的命名空间中可见的信息。两个结构的定义如下：

<pid.h>

```
struct upid {
        int nr;
        struct pid_namespace *ns;
        struct hlist_node pid_chain;
};

struct pid
{
        atomic_t count;
        /* 使用该pid的进程的列表 */
        struct hlist_head tasks[PIDTYPE_MAX];
        int level;
```

```
        struct upid numbers[1];
};
```

由于这两个结构与其他一些数据结构存在广泛的联系，在分别讨论相关结构之前，图2-5对此进行了概述。

对于struct upid，nr表示ID的数值，ns是指向该ID所属的命名空间的指针。所有的upid实例都保存在一个散列表中，稍后我们会看到该结构。pid_chain用内核的标准方法实现了散列溢出链表。

struct pid的定义首先是一个引用计数器count。tasks是一个数组，每个数组项都是一个散列表头，对应于一个ID类型。这样做是必要的，因为一个ID可能用于几个进程。所有共享同一给定ID的task_struct实例，都通过该列表连接起来。PIDTYPE_MAX表示ID类型的数目：

```
<pid.h>
enum pid_type
{
        PIDTYPE_PID,
        PIDTYPE_PGID,
        PIDTYPE_SID,
        PIDTYPE_MAX
};
```

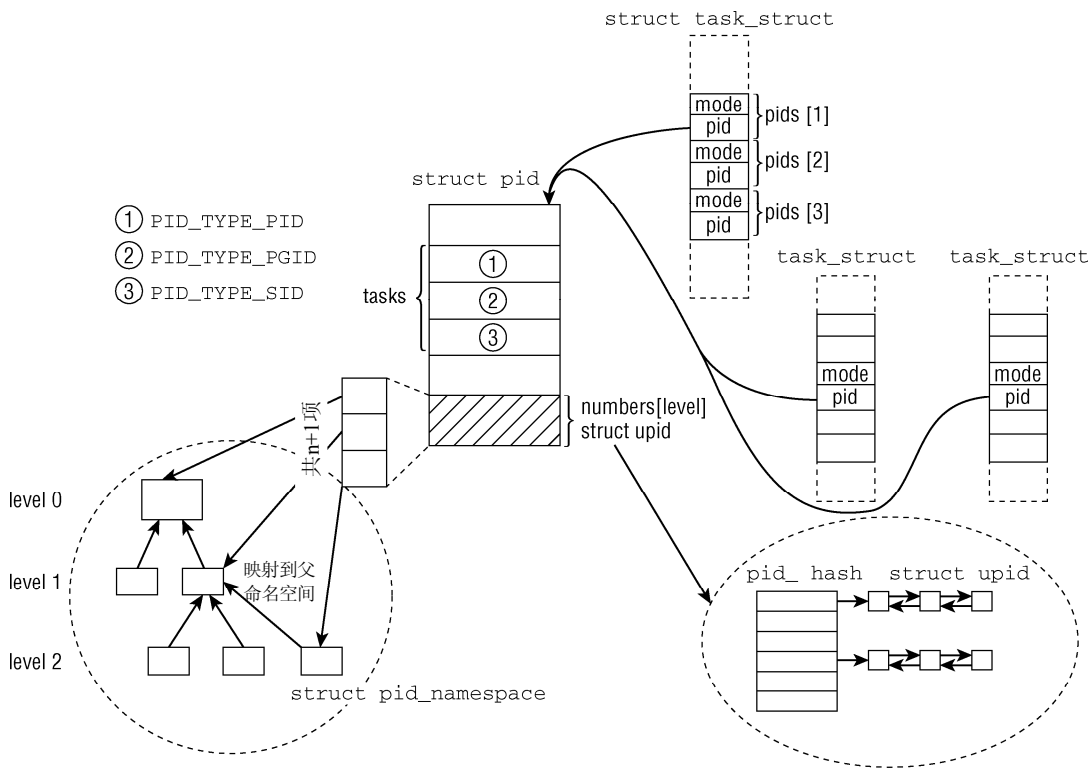

图2-5 实现可感知命名空间的ID表示所用的数据结构

请注意，枚举类型中定义的ID类型不包括线程组ID！这是因为线程组ID无非是线程组组长的PID

而已，因此再单独定义一项是不必要的。

一个进程可能在多个命名空间中可见，而其在各个命名空间中的局部ID各不相同。level表示可以看到该进程的命名空间的数目（换言之，即包含该进程的命名空间在命名空间层次结构中的深度），而numbers是一个upid实例的数组，每个数组项都对应于一个命名空间。注意该数组形式上只有一个数组项，如果一个进程只包含在全局命名空间中，那么确实如此。由于该数组位于结构的末尾，因此只要分配更多的内存空间，即可向数组添加附加的项。

由于所有共享同一ID的task_struct实例都按进程存储在一个散列表中，因此需要在struct task_struct中增加一个散列表元素：

<sched.h>

```
struct task_struct {
...
        /* PID与PID散列表的联系。 */
        struct pid_link pids[PIDTYPE_MAX];
...
};
```

辅助数据结构pid_link可以将task_struct连接到表头在struct pid中的散列表上：

<pid.h>

```
struct pid_link
{
        struct hlist_node node;
        struct pid *pid;
};
```

pid指向进程所属的pid结构实例，node用作散列表元素。

为在给定的命名空间中查找对应于指定PID数值的pid结构实例，使用了一个散列表：

kernel/pid.c

```
static struct hlist_head *pid_hash;
```

hlist_head是一个内核的标准数据结构，用于建立双链散列表（附录C描述了该散列表的结构，并介绍了用于处理该数据结构的几个辅助函数）。

pid_hash用作一个hlist_head数组。数组的元素数目取决于计算机的内存配置，大约在2^4=16和2^{12}=4096之间。pidhash_init用于计算恰当的容量并分配所需的内存。

假如已经分配了struct pid的一个新实例，并设置用于给定的ID类型。它会如下附加到task_struct：

kernel/pid.c

```
int fastcall attach_pid(struct task_struct *task, enum pid_type type,
                struct pid *pid)
{
        struct pid_link *link;

        link = &task->pids[type];
        link->pid = pid;
        hlist_add_head_rcu(&link->node, &pid->tasks[type]);

        return 0;
}
```

这里建立了双向连接：task_struct可以通过task_struct->pids[type]->pid访问pid实例。而从pid实例开始，可以遍历tasks[type]散列表找到task_struct。hlist_add_head_rcu是遍历散列表的标准函数，此外还确保了遵守RCU机制（参见第5章）。因为，在其他内核组件并发地操作散列

表时，可防止竞态条件（race condition）出现。

● 函数

内核提供了若干辅助函数，用于操作和扫描上面描述的数据结构。本质上内核必须完成下面两个不同的任务。

(1) 给出局部数字ID和对应的命名空间，查找此二元组描述的task_struct。

(2) 给出task_struct、ID类型、命名空间，取得命名空间局部的数字ID。

我们首先专注于如何将task_struct实例变为数字ID。这个过程包含下面两个步骤。

(1) 获得与task_struct关联的pid实例。辅助函数task_pid、task_tgid、task_pgrp和task_session分别用于取得不同类型的ID。获取PID的实现很简单：

<sched.h>

```
static inline struct pid *task_pid(struct task_struct *task)
{
        return task->pids[PIDTYPE_PID].pid;
}
```

获取TGID的做法类似，因为TGID不过是线程组组长的PID而已。只要将上述实现替换为task->group_leader->pids[PIDTYPE_PID].pid即可。

找出进程组ID则需要使用PIDTYPE_PGID作为数组索引，但该ID仍然需要从线程组组长的task_struct实例获取：

<sched.h>

```
static inline struct pid *task_pgrp(struct task_struct *task)
{
        return task->group_leader->pids[PIDTYPE_PGID].pid;
}
```

(2) 在获得pid实例之后，从struct pid的numbers数组中的uid信息，即可获得数字ID：

kernel/pid.c

```
pid_t pid_nr_ns(struct pid *pid, struct pid_namespace *ns)
{
        struct upid *upid;
        pid_t nr = 0;

        if (pid && ns->level <= pid->level) {
                upid = &pid->numbers[ns->level];
                if (upid->ns == ns)
                        nr = upid->nr;
        }
        return nr;
}
```

因为父命名空间可以看到子命名空间中的PID，反过来却不行，内核必须确保当前命名空间的level小于或等于产生局部PID的命名空间的level。

同样重要的是要注意到，内核只需要关注产生全局PID。因为全局命名空间中所有其他ID类型都会映射到PID，因此不必生成诸如全局TGID或SID。

除了在第2步使用的pid_nr_ns之外，内核还可以使用下列辅助函数：

❑ pid_vnr返回该ID所属的命名空间所看到的局部PID；

❑ pid_nr则获取从init进程看到的全局PID。

这两个函数都依赖于pid_nr_ns，并自动选择适当的level：0用于获取全局PID，而pid->level则用于获取局部PID。

内核提供了几个辅助函数，合并了前述步骤：

kernel/pid.c

```
pid_t task_pid_nr_ns(struct task_struct *tsk, struct pid_namespace *ns)
pid_t task_tgid_nr_ns(struct task_struct *tsk, struct pid_namespace *ns)
pid_t task_pgrp_nr_ns(struct task_struct *tsk, struct pid_namespace *ns)
pid_t task_session_nr_ns(struct task_struct *tsk, struct pid_namespace *ns)
```

从函数名可以明显推断其语义，因此我们不再赘述。

现在我们把注意力转向内核如何将数字PID和命名空间转换为pid实例。同样需要下面两个步骤。

(1) 给出进程的局部数字PID和关联的命名空间（这是PID的用户空间表示），为确定pid实例（这是PID的内核表示），内核必须采用标准的散列方案。首先，根据PID和命名空间指针计算在pid_hash数组中的索引，[①]然后遍历散列表直至找到所要的元素。这是通过辅助函数find_pid_ns处理的：

kernel/pid.c

```
struct pid * fastcall find_pid_ns(int nr, struct pid_namespace *ns)
```

struct upid的实例保存在散列表中，由于这些实例直接包含在struct pid中，内核可以使用container_of机制（参见附录C）推断出所要的信息。

(2) pid_task取出pid->tasks[type]散列表中的第一个task_struct实例。

这两个步骤可以通过辅助函数find_task_by_pid_type_ns完成：

kernel/pid.c

```
struct task_struct *find_task_by_pid_type_ns(int type, int nr,
                struct pid_namespace *ns)
{
        return pid_task(find_pid_ns(nr, ns), type);
}
```

一些简单一点的辅助函数基于最一般性的find_task_by_pid_type_ns：

- find_task_by_pid_ns(pid_t nr, struct pid_namespace * ns)根据给出的数字PID和进程的命名空间来查找task_struct实例。
- find_task_by_vpid(pid_t vnr)通过局部数字PID查找进程。
- find_task_by_pid(pid_t nr)通过全局数字PID查找进程。

内核源代码中许多地方都需要find_task_by_pid，因为很多特定于进程的操作（例如，使用kill发送一个信号）都通过PID标识目标进程。

3. 生成唯一的PID

除了管理PID之外，内核还负责提供机制来生成唯一的PID（尚未分配）。在这种情况下，可以忽略各种不同类型的PID之间的差别，因为按一般的UNIX观念，只需要为PID生成唯一的数值即可。所有其他的ID都可以派生自PID，在下文讨论fork和clone时会看到这一点。在随后的几节中，名词PID还是指一般的UNIX进程ID（PIDTYPE_PID）。

为跟踪已经分配和仍然可用的PID，内核使用一个大的位图，其中每个PID由一个比特标识。PID的值可通过对应比特在位图中的位置计算而来。

因此，分配一个空闲的PID，本质上就等同于寻找位图中第一个值为0的比特，接下来将该比特设置为1。反之，释放一个PID可通过将对应的比特从1切换为0来实现。这些操作使用下述两个函数实现：

---

① 为达到该目的，内核使用了乘法散列法，用的是与机器字所能表示的最大数字成黄金分割比率的一个素数。具体细节可参见[Knu97]。

kernel/pid.c

```
static int alloc_pidmap(struct pid_namespace *pid_ns)
```

用于分配一个PID，而

kernel/pid.c

```
static fastcall void free_pidmap(struct pid_namespace *pid_ns, int pid)
```

用于释放一个PID。我们这里不关注具体的实现方式，但它们必须能够在命名空间下工作。

在建立一个新进程时，进程可能在多个命名空间中是可见的。对每个这样的命名空间，都需要生成一个局部PID。这是在alloc_pid中处理的：

kernel/pid.c

```
struct pid *alloc_pid(struct pid_namespace *ns)
{
        struct pid *pid;
        enum pid_type type;
        int i, nr;
        struct pid_namespace *tmp;
        struct upid *upid;
...
        tmp = ns;
        for (i = ns->level; i >= 0; i--) {
                nr = alloc_pidmap(tmp);
...
                pid->numbers[i].nr = nr;
                pid->numbers[i].ns = tmp;
                tmp = tmp->parent;
        }
        pid->level = ns->level;
...
```

起始于建立进程的命名空间，一直到初始的全局命名空间，内核会为此间的每个命名空间分别创建一个局部PID。包含在struct pid中的所有upid都用重新生成的PID更新其数据。每个upid实例都必须置于PID散列表中：

kernel/pid.c

```
        for (i = ns->level; i >= 0; i--) {
                upid = &pid->numbers[i];
                hlist_add_head_rcu(&upid->pid_chain,
                                &pid_hash[pid_hashfn(upid->nr, upid->ns)]);
        }
...
        return pid;
}
```

2.3.4 进程关系

除了源于ID连接的关系之外，内核还负责管理建立在UNIX进程创建模型之上“家族关系”。相关讨论一般使用下列术语。

- 如果进程A分支形成进程B，进程A称之为*父进程*而进程B则是*子进程*。[①]
  如果进程B再次分支建立另一个进程C，进程A和进程C之间有时称之为*祖孙关系*。
- 如果进程A分支若干次形成几个子进程B_1，B_2，…，B_n，各个B_i进程之间的关系称之为*兄弟关系*。

① 不同于自然的家庭，进程只有一个父母系。

图2-6说明了可能的进程家族关系。

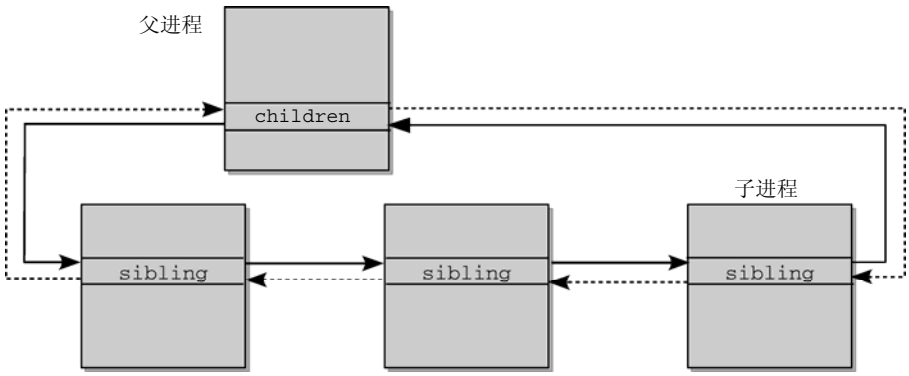

图2-6 进程之间的家族关系

task_struct数据结构提供了两个链表表头，用于实现这些关系：

```
<sched.h>
struct task_struct {
...
        struct list_head children;   /* 子进程链表 */
        struct list_head sibling;    /* 连接到父进程的子进程链表 */
...
}
```

- children是链表表头，该链表中保存有进程的所有子进程。
- sibling用于将兄弟进程彼此连接起来。

新的子进程置于sibling链表的起始位置，这意味着可以重建进程分支的时间顺序。[①]

2.4 进程管理相关的系统调用

在本节中，我将讨论fork和exec系列系统调用的实现。通常这些调用不是由应用程序直接发出的，而是通过一个中间层调用，即负责与内核通信的C标准库。

从用户状态切换到核心态的方法，依不同的体系结构而各有不同。在附录A中，我详细讲述了用于在这两种状态之间切换的机制，并解释了用户空间和内核空间之间如何交换参数。就目前而言，将内核视为由C标准库使用的“程序库”即可，我在第1章简要地提到过这一点。

2.4.1 进程复制

传统的UNIX中用于复制进程的系统调用是fork。但它并不是Linux为此实现的唯一调用，实际上Linux实现了3个。

(1) fork是重量级调用，因为它建立了父进程的一个完整副本，然后作为子进程执行。为减少与该调用相关的工作量，Linux使用了写时复制（copy-on-write）技术，下文中会讨论。

① 2.6.21之前的内核版本有3个辅助函数：younger_sibling、older_sibling和eldest_child，在访问上述链表及其元素时能够有所帮助。它们用于生成调试输出，但不是很有用，因此在后续版本中被删除了。补丁作者Ingo Molnar注意到对应的代码是内核最古老的成分之一，并作了相应的注记。这导致另一个著名的开发者Linus Torvalds取消了这个补丁。

(2) vfork类似于fork，但并不创建父进程数据的副本。相反，父子进程之间共享数据。这节省了大量CPU时间（如果一个进程操纵共享数据，则另一个会自动注意到）。

vfork设计用于子进程形成后立即执行execve系统调用加载新程序的情形。在子进程退出或开始新程序之前，内核保证父进程处于堵塞状态。

引用手册页vfork(2)的文字，“非常不幸，Linux从过去复活了这个幽灵”。由于fork使用了写时复制技术，vfork速度方面不再有优势，因此应该避免使用它。

(3) clone产生线程，可以对父子进程之间的共享、复制进行精确控制。

1. 写时复制

内核使用了写时复制（Copy-On-Write，COW）技术，以防止在fork执行时将父进程的所有数据复制到子进程。该技术利用了下述事实：进程通常只使用了其内存页的一小部分。[①]在调用fork时，内核通常对父进程的每个内存页，都为子进程创建一个相同的副本。这有两种很不好的负面效应。

(1) 使用了大量内存。

(2) 复制操作耗费很长时间。

如果应用程序在进程复制之后使用exec立即加载新程序，那么负面效应会更严重。这实际上意味着，此前进行的复制操作是完全多余的，因为进程地址空间会重新初始化，复制的数据不再需要了。

内核可以使用技巧规避该问题。并不复制进程的整个地址空间，而是只复制其页表。这样就建立了虚拟地址空间和物理内存页之间的联系，我在第1章简要地讲过，具体过程请参见第3章和第4章。因此，fork之后父子进程的地址空间指向同样的物理内存页。

当然，父子进程不能允许修改彼此的页，[②]这也是两个进程的页表对页标记了只读访问的原因，即使在普通环境下允许写入也是如此。

假如两个进程只能读取其内存页，那么二者之间的数据共享就不是问题，因为不会有修改。

只要一个进程试图向复制的内存页写入，处理器会向内核报告访问错误（此类错误被称作缺页异常）。内核然后查看额外的内存管理数据结构（参见第4章），检查该页是否可以用读写模式访问，还是只能以只读模式访问。如果是后者，则必须向进程报告段错误。读者会在第4章看到，缺页异常处理程序的实际实现要复杂得多，因为还必须考虑其他方面的问题，例如换出的页。

如果页表项将一页标记为“只读”，但通常情况下该页应该是可写的，内核可根据此条件来判断该页实际上是COW页。因此内核会创建该页专用于当前进程的副本，当然也可以用于写操作。直至第4章我们才会讨论复制操作的实现方式，因为这需要内存管理方面广泛的背景知识。

COW机制使得内核可以尽可能延迟内存页的复制，更重要的是，在很多情况下不需要复制。这节省了大量时间。

2. 执行系统调用

fork、vfork和clone系统调用的入口点分别是sys_fork、sys_vfork和sys_clone函数。其定义依赖于具体的体系结构，因为在用户空间和内核空间之间传递参数的方法因体系结构而异（更多细节请参见第13章）。上述函数的任务是从处理器寄存器中提取由用户空间提供的信息，调用体系结构无关的do_fork函数，后者负责进程复制。该函数的原型如下：

kernel/fork.c

```
long do_fork(unsigned long clone_flags,
             unsigned long stack_start,
```

① 进程访问最频繁的页的集合被称为工作区（working set）。

② 两个进程显示共享的页除外。

```
                struct pt_regs *regs,
                unsigned long stack_size,
                int __user *parent_tidptr,
                int __user *child_tidptr)
```

该函数需要下列参数。

- clone_flags是一个标志集合，用来指定控制复制过程的一些属性。最低字节指定了在子进程终止时被发给父进程的信号号码。其余的高位字节保存了各种常数，下文会分别讨论。
- start_stack是用户状态下栈的起始地址。
- regs是一个指向寄存器集合的指针，其中以原始形式保存了调用参数。该参数使用的数据类型是特定于体系结构的struct pt_regs，其中按照系统调用执行时寄存器在内核栈上的存储顺序，保存了所有的寄存器（更详细的信息，请参考附录A）。
- stack_size是用户状态下栈的大小。该参数通常是不必要的，设置为0。
- parent_tidptr和child_tidptr是指向用户空间中地址的两个指针，分别指向父子进程的TID。NPTL（Native Posix Threads Library）库的线程实现需要这两个参数。我将在下文讨论其语义。

不同的fork变体，主要是通过标志集合区分。在大多数体系结构上，[①]典型的fork调用的实现方式与IA-32处理器相同。

arch/x86/kernel/process_32.c

```
asmlinkage int sys_fork(struct pt_regs regs)
{
        return do_fork(SIGCHLD, regs.esp, &regs, 0, NULL, NULL);
}
```

唯一使用的标志是SIGCHLD。这意味着在子进程终止后发送SIGCHLD信号通知父进程。最初，父子进程的栈地址相同（起始地址保存在IA-32系统的esp寄存器中）。但如果操作栈地址并写入数据，则COW机制会为每个进程分别创建一个栈副本。

如果do_fork成功，则新建进程的PID作为系统调用的结果返回，否则返回错误码（负值）。

sys_vfork的实现与sys_fork只是略微不同，前者使用了额外的标志（CLONE_VFORK和CLONE_VM，其语义下文讨论）。

sys_clone的实现方式与上述调用相似，差别在于do_fork如下调用：

arch/x86/kernel/process_32.c

```
asmlinkage int sys_clone(struct pt_regs regs)
{
        unsigned long clone_flags;
        unsigned long newsp;
        int __user *parent_tidptr, *child_tidptr;

        clone_flags = regs.ebx;
        newsp = regs.ecx;
        parent_tidptr = (int __user *)regs.edx;
        child_tidptr = (int __user *)regs.edi;
        if (!newsp)
                newsp = regs.esp;
        return do_fork(clone_flags, newsp, &regs, 0, parent_tidptr, child_tidptr);
}
```

① 例外：Sparc(64)系统通过sparc_do_fork访问do_fork；IA-64只提供了一个系统调用sys_clone2，用于在用户空间实现fork、vfork和clone系统调用。sys_clone2和sparc_do_fork最终都依赖于do_fork。

标志不再是硬编码的，而是可以通过各个寄存器参数传递到系统调用。因而该函数的第一部分负责提取这些参数。另外，也不再复制父进程的栈，而是可以指定新的栈地址（newsp）。在生成线程时，可能需要这样做，线程可能与父进程共享地址空间，但线程自身的栈可能在另一个地址空间。另外还指定了用户空间中的两个指针（parent_tidptr和child_tidptr），用于与线程库通信。其语义在2.4.1节讨论。

3. do_fork的实现

所有3个fork机制最终都调用了kernel/fork.c中的do_fork（一个体系结构无关的函数），其代码流程如图2-7所示。

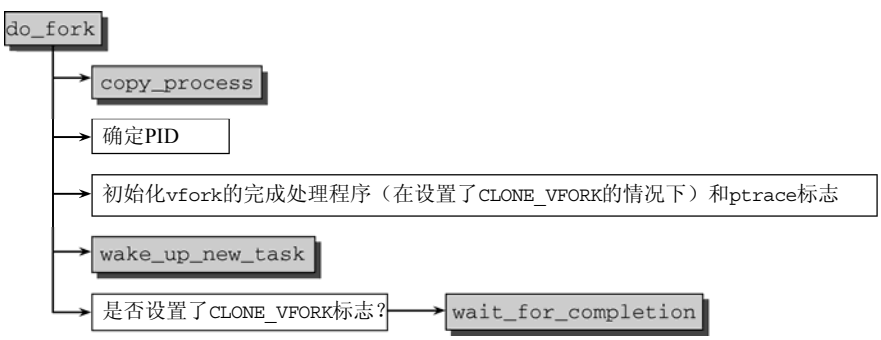

图2-7 do_fork的代码流程图

do_fork以调用copy_process开始，后者执行生成新进程的实际工作，并根据指定的标志重用父进程的数据。在子进程生成之后，内核必须执行下列收尾操作：

- 由于fork要返回新进程的PID，因此必须获得PID。这是比较复杂的，因为如果设置了CLONE_NEWPID标志，fork操作可能创建了新的PID命名空间。如果是这样，则需要调用task_pid_nr_ns获取在父命名空间中为新进程选择的PID，即发出fork调用的进程所在的命名空间。如果PID命名空间没有改变，调用task_pid_vnr获取局部PID即可，因为新旧进程都在同一个命名空间中。

  kernel/fork.c

  ```
  nr = (clone_flags & CLONE_NEWPID) ?
          task_pid_nr_ns(p, current->nsproxy->pid_ns) :
                  task_pid_vnr(p);
  ```

- 如果将要使用Ptrace（参见第13章）监控新的进程，那么在创建新进程后会立即向其发送SIGSTOP信号，以便附接的调试器检查其数据。
- 子进程使用wake_up_new_task唤醒。换言之，即将其task_struct添加到调度器队列。调度器也有机会对新启动的进程给予特别处理，这使得可以实现一种策略以便新进程有较高的几率尽快开始运行，另外也可以防止一再地调用fork浪费CPU时间。

  如果子进程在父进程之前开始运行，则可以大大地减少复制内存页的工作量，尤其是子进程在fork之后发出exec调用的情况下。但要记住，将进程排到调度器数据结构中并不意味着该子进程可以立即开始执行，而是调度器此时起可以选择它运行。
- 如果使用vfork机制（内核通过设置的CLONE_VFORK标志识别），必须启用子进程的完成机制（completions mechanism）。子进程的task_struct的vfork_done成员即用于该目的。借助于

wait_for_completion函数，父进程在该变量上进入睡眠状态，直至子进程退出。在进程终止（或用execve启动新应用程序）时，内核自动调用complete（vfork_done）。这会唤醒所有因该变量睡眠的进程。在第14章中，我会非常详细地讨论完成机制的实现。

- 通过采用这种方法，内核可以确保使用vfork生成的子进程的父进程会一直处于不活动状态，直至子进程退出或执行一个新的程序。父进程的临时睡眠状态，也确保了两个进程不会彼此干扰或操作对方的地址空间。

4. 复制进程

在do_fork中大多数工作是由copy_process函数完成的，其代码流程如图2-8所示。请注意，该函数必须处理3个系统调用（fork、vfork和clone）的主要工作。

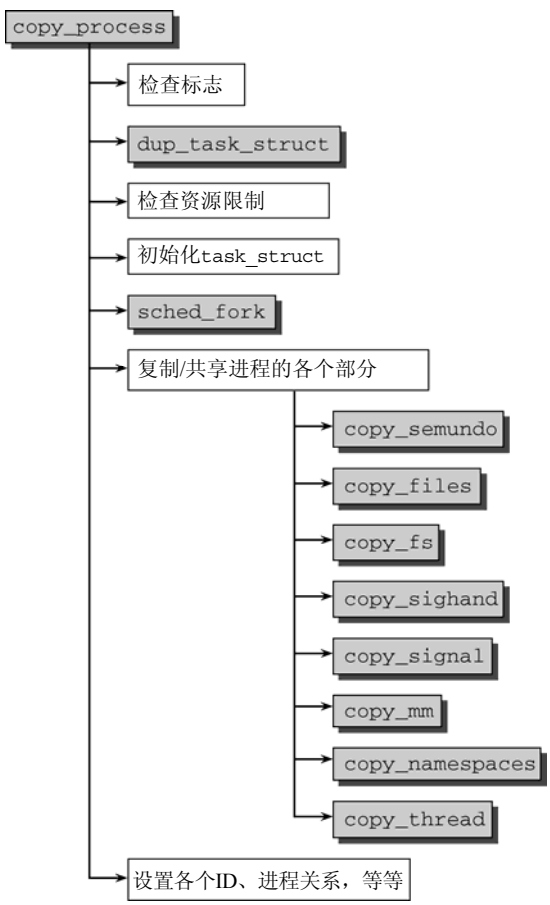

图2-8 copy_process的代码流程图

由于内核必须处理许多特别和具体的情形，我们只讲述该函数的一个略微简化的版本，免得迷失于无数的细节而忽略最重要的方面。

复制进程的行为受到相当多标志的控制。clone(2)的手册页详细讲述了这些标志，这里不再赘述，我建议读者看一下手册页，或者Linux系统程序设计方面的任何好书都可以。我们更感兴趣的是，

某些标志组合没有意义，内核必须捕获这种情况。例如，一方面请求创建一个新命名空间（CLONE_NEWNS），而同时要求与父进程共享所有的文件系统信息（CLONE_FS），就是没有意义的。捕获这种组合并返回错误码并不复杂：

kernel/fork.c

```
static struct task_struct *copy_process(unsigned long clone_flags,
                                        unsigned long stack_start,
                                        struct pt_regs *regs,
                                        unsigned long stack_size,
                                        int __user *child_tidptr,
                                        struct pid *pid)

{
        int retval;
        struct task_struct *p;
        int cgroup_callbacks_done = 0;

        if ((clone_flags & (CLONE_NEWNS|CLONE_FS)) == (CLONE_NEWNS|CLONE_FS))
                return ERR_PTR(-EINVAL);
...
```

此处很适宜回忆简介部分提到的：Linux有时候在操作成功时需要返回指针，而在失败时则返回错误码。遗憾的是，C语言每个函数只允许一个直接的返回值，因此任何有关可能错误的信息都必须编码到指针中。虽然一般而言指针可以指向内存中的任意位置，而Linux支持的每个体系结构的虚拟地址空间中都有一个从虚拟地址0到至少4 KiB的区域，该区域中没有任何有意义的信息。因此内核可以重用该地址范围来编码错误码。如果fork的返回值指向前述的地址范围内部，那么该调用就失败了，其原因可以由指针的数值判断。ERR_PTR是一个辅助宏，用于将数值常数（例如EINVAL，非法操作）编码为指针。

还需要进一步检查一些标志。

- 在用CLONE_THREAD创建一个线程时，必须用CLONE_SIGHAND激活信号共享。通常情况下，一个信号无法发送到线程组中的各个线程。
- 只有在父子进程之间共享虚拟地址空间时（CLONE_VM），才能提供共享的信号处理程序。因此类似的想法是，要想达到同样的效果，线程也必须与父进程共享地址空间。

在内核建立了自洽的标志集之后，则用dup_task_struct来建立父进程task_struct的副本。用于子进程的新的task_struct实例可以在任何空闲的内核内存位置分配（更多细节请参见第3章，其中讲解了这里提到的分配机制）。

父子进程的task_struct实例只有一个成员不同：新进程分配了一个新的核心态栈，即task_struct->stack。通常栈和thread_info一同保存在一个联合中，thread_info保存了线程所需的所有特定于处理器的底层信息。

<sched.h>

```
union thread_union {
        struct thread_info thread_info;
        unsigned long stack[THREAD_SIZE/sizeof(long)];
};
```

原则上，只要设置了预处理器常数__HAVE_THREAD_FUNCTIONS通知内核，那么各个体系结构可以随意在stack数组中存储什么数据。在这种情况下，它们必须自行实现task_thread_info和task_stack_page，这两个函数用于获取给定task_struct实例的线程信息和核心态栈。另外，它们必须实现dup_task_struct中调用的函数setup_thread_stack，以便确定stack成员的具体内存布局。当

前只有IA-64和m68k不依赖于内核的默认方法。

在大多数体系结构上，使用一两个内存页来保存一个thread_union的实例。在IA-32上，两个内存页是默认设置，因此可用的内核栈长度略小于8 KiB，其中一部分被thread_info实例占据。不过要注意，配置选项4KSTACKS会将栈长度降低到4 KiB，即一个页面。如果系统上有许多进程在运行，这样做是有利的，因为每个进程可以节省一个页面。另一方面，对于经常趋向于使用过多栈空间的外部驱动程序来说，这可能导致问题。标准发布版所提供的内核，其所有核心部分都已经设计为能够在4 KiB栈长度配置下运转流畅，但一旦需要只提供二进制代码的驱动程序，就可能引发问题（糟糕的是，过去已经发生过这类问题），此类驱动通常习于向可用的栈空间乱塞数据。

thread_info保存了特定于体系结构的汇编语言代码需要访问的那部分进程数据。尽管该结构的定义因不同的处理器而不同，大多数系统上该结构的内容类似于下列代码。

<asm-arch/thread_info.h>

```
struct thread_info {
        struct task_struct      *task;          /* 当前进程task_struct指针 */
        struct exec_domain      *exec_domain;   /* 执行区间 */
        unsigned long           flags;          /* 底层标志 */
        unsigned long           status;         /* 线程同步标志 */
        __u32                   cpu;            /* 当前CPU */
        int                     preempt_count;  /* 0 => 可抢占，  <0 => BUG */

        mm_segment_t            addr_limit;     /* 线程地址空间 */
        struct restart_block    restart_block;
}
```

- task是指向进程task_struct实例的指针。
- exec_domain用于实现执行区间（execution domain），后者用于在一类计算机上实现多种的ABI（Application Binary Interface，应用程序二进制接口）。例如，在AMD64系统的64bit模式下运行32bit应用程序。
- flags可以保存各种特定于进程的标志，我们对其中两个特别感兴趣，如下所示。
  - 如果进程有待决信号则置位TIF_SIGPENDING。
  - TIF_NEED_RESCHED表示该进程应该或想要调度器选择另一个进程替换本进程执行。

  其他可用的常数是特定于硬件的，几乎从不使用，可以参见<asm-arch/thread_info.h>。
- cpu说明了进程正在其上执行的CPU数目（在多处理器系统上很重要，在单处理器系统上非常容易判断）。
- preempt_count实现内核抢占所需的一个计数器，我将在2.8.3节讨论。
- addr_limit指定了进程可以使用的虚拟地址的上限。如前所述，该限制适用于普通进程，但内核线程可以访问整个虚拟地址空间，包括只有内核能访问的部分。这并不意味着限制进程可以分配的内存数量。回想第1章提到的用户和内核地址空间之间的分隔，我会在第4章详细讨论该主题。
- restart_block用于实现信号机制（参见第5章）。

图2-9给出了task_struct、thread_info和内核栈之间的关系。在内核的某个特定组件使用了过多栈空间时，内核栈会溢出到thread_info部分，这很可能会导致严重的故障。此外在紧急情况下输出调用栈回溯时将会导致错误的信息出现，因此内核提供了kstack_end函数，用于判断给出的地址是否位于栈的有效部分之内。

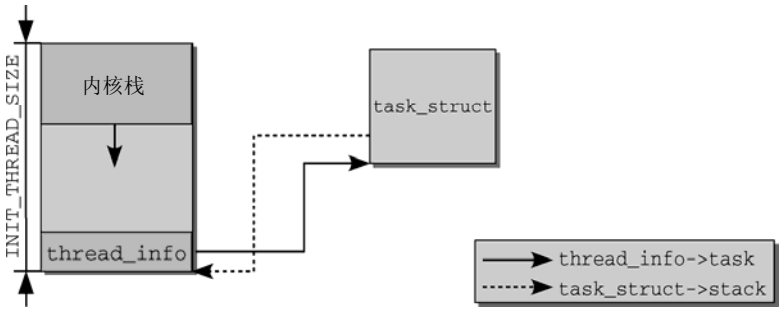

图2-9 进程的task_struct、thread_info和内核栈之间的关系

dup_task_struct会复制父进程task_struct和thread_info实例的内容，但stack则与新的thread_info实例位于同一内存区域。这意味着父子进程的task_struct此时除了栈指针之外是完全相同的，但子进程的task_struct实例会在copy_process过程中修改。

此外所有体系结构都将两个名为current和current_thread_info的符号定义为宏或函数。其语义如下所示。

- current_thread_info可获得指向当前执行进程的thread_info实例的指针。其地址可以根据内核栈指针确定，因为thread_info实例总是位于栈顶。[①]因为每个进程分别使用各自的内核栈，进程到栈的映射是唯一的。
- current给出了当前进程task_struct实例的地址。该函数在源代码中出现非常频繁。该地址可以使用current_thread_info()确定：current = current_thread_info()->task。

我们继续讨论copy_process。在dup_task_struct成功之后，内核会检查当前的特定用户在创建新进程之后，是否超出了允许的最大进程数目：

kernel/fork.c

```
        if (atomic_read(&p->user->processes) >=
                        p->signal->rlim[RLIMIT_NPROC].rlim_cur) {
                if (!capable(CAP_SYS_ADMIN) && !capable(CAP_SYS_RESOURCE) &&
                    p->user != current->nsproxy->user_ns->root_user)
                        goto bad_fork_free;
        }
...
```

拥有当前进程的用户，其资源计数器保存一个user_struct实例中，可通过task_struct->user访问，特定用户当前持有进程的数目保存在user_struct->processes。如果该值超出rlimit设置的限制，则放弃创建进程，除非当前用户是root用户或分配了特别的权限（CAP_SYS_ADMIN或CAP_SYS_RESOURCE）。检测root用户很有趣：回想上文，每个PID命名空间都有各自的root用户。上述检测必须考虑这一点。

如果资源限制无法防止进程建立，则调用接口函数sched_fork，以便使调度器有机会对新进程进行设置。在内核版本2.6.23引入CFQ调度器之前，该过程要更加复杂，因为父进程的剩余时间片必须在父子进程之间分配。由于新的调度器不再需要时间片，现在简单多了。本质上，该例程会初始化一些统计字段，在多处理器系统上，如果有必要可能还会在各个CPU之间对可用的进程重新均衡一下。

① 指向内核栈的指针通常保存在一个特别保留的寄存器中。有些体系结构特别是IA-32和AMD64使用了不同的解决方案，我将在A.10.3节讨论。

此外进程状态设置为TASK_RUNNING，由于新进程事实上还没运行，这个状态实际上不是真实的。但这可以防止内核的任何其他部分试图将进程状态从非运行改为运行，并在进程的设置彻底完成之前调度进程。

接下来会调用许多形如copy_xyz的例程，以便复制或共享特定的内核子系统的资源。task_struct包含了一些指针，指向具体数据结构的实例，描述了可共享或可复制的资源。由于子进程的task_struct是从父进程的task_struct精确复制而来，因此相关的指针最初都指向同样的资源，或者说同样的具体资源实例，如图2-10所示。

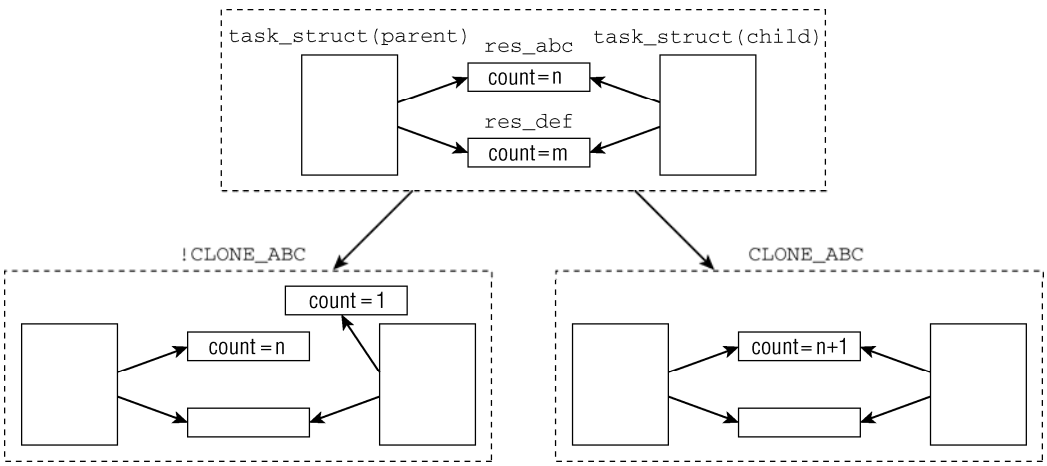

图2-10 在创建新线程时，父进程的资源可以共享或复制

假定我们有两个资源：res_abc和res_def。最初父子进程的task_struct中的对应指针都指向了资源的同一个实例，即内存中特定的数据结构。

如果CLONE_ABC置位，则两个进程会共享res_abc。此外，为防止与资源实例关联的内存空间释放过快，还需要对实例的引用计数器加1，只有进程不再使用内存时，才能释放。如果父进程或子进程修改了共享资源，则变化在两个进程中都可以看到。

如果CLONE_ABC没有置位，接下来会为子进程创建res_abc的一份副本，新副本的资源计数器初始化为1。因此在这种情况下，如果父进程或子进程修改了资源，变化不会传播到另一个进程。

通常，设置的CLONE标志越少，需要完成的工作越少。但多设置一些标志，则使得父子进程有更多机会相互操作彼此的数据结构，在编写应用程序时必须考虑到这一点。

判断资源是共享还是复制需要通过许多辅助例程完成，每个辅助例程对应一种资源。我不打算在此讨论各个copy_xyz函数的实现（相当无趣），但会概述其作用。在后续章节中详细论述各个子系统时，我会介绍与进程每个组件相关的数据结构。

- 如果COPY_SYSVSEM置位，则copy_semundo使用父进程的System V信号量。
- 如果CLONE_FILES置位，则copy_files使用父进程的文件描述符；否则创建新的files结构（参见第8章），其中包含的信息与父进程相同。该信息的修改可以独立于原结构。
- 如果CLONE_FS置位，则copy_fs使用父进程的文件系统上下文（task_struct->fs）。这是一个fs_struct类型的结构，包含了诸如根目录、进程的当前工作目录之类的信息（更多细节请参见第8章）。

- 如果CLONE_SIGHAND或CLONE_THREAD置位，则copy_sighand使用父进程的信号处理程序。第5章会更详细地论述使用的struct sighand_struct结构。
- 如果CLONE_THREAD置位，则copy_signal与父进程共同使用信号处理中不特定于处理程序的部分（task_struct->signal，参见第5章）。
- 如果COPY_MM置位，则copy_mm让父进程和子进程共享同一地址空间。在这种情况下，两个进程使用同一个mm_struct实例（参见第4章），task_struct->mm指针即指向该实例。
- 如果copy_mm没有置位，并不意味着需要复制父进程的整个地址空间。内核确实会创建页表的一份副本，但并不复制页的实际内容。这是使用COW机制完成的，仅当其中一个进程将数据写入页时，才会进行实际复制。
- copy_namespaces有特别的调用语义。它用于建立子进程的命名空间。回想前文提到的几个控制与父进程共享何种命名空间的CLONE_NEWxyz标志，但其语义与所有其他标志都相反。如果没有指定CLONE_NEWxyz，则与父进程共享相应的命名空间，否则创建一个新的命名空间。copy_namespaces相当于调度程序，对每个可能的命名空间，分别执行对应的复制例程。但各个具体的复制例程就没什么趣味了，因为本质上就是复制数据或通过引用计数的管理来共享现存的实例，因此我不会详细讨论各例程的实现。
- copy_thread与这里讨论的所有其他复制操作都大不相同，这是一个特定于体系结构的函数，用于复制进程中特定于线程（thread-specific）的数据。

> 这里的特定于线程并不是指某个CLONE标志，也不是指操作对线程而非整个进程执行。其语义无非是指复制执行上下文中特定于体系结构的所有数据（内核中名词线程通常用于多个含义）。

重要的是填充task_struct->thread的各个成员。这是一个thread_struct类型的结构，其定义是体系结构相关的。它包含了所有寄存器（和其他信息），内核在进程之间切换时需要保存和恢复进程的内容，该结构可用于此。

为理解各个thread_struct结构的布局，需要深入了解各种CPU的相关知识。对这些结构的详尽讨论则超过了本书的范围。但附录A包含了几种系统上该结构内容的一些相关信息。

回到对copy_process的讨论，内核必须填好task_struct中对父子进程不同的各个成员。包含下列一些：

- task_struct中包含的各个链表元素，例如sibling和children；
- 间隔定时器成员cpu_timers（参见第15章）；
- 待决信号列表（pending），将在第5章讨论。

在用之前描述的机制为进程分配一个新的pid实例之后，则保存在task_struct中。对于线程，线程组ID与分支进程（即调用fork/clone的进程）相同：

kernel/fork.c

```
        p->pid = pid_nr(pid);
        p->tgid = p->pid;
        if (clone_flags & CLONE_THREAD)
                p->tgid = current->tgid;
...
```

回想一下，pid_nr函数对给定的pid实例计算全局数值PID。

对普通进程，父进程是分支进程。对于线程来说有些不同：由于线程被视为分支进程内部的第二

（或第三、第四，等等）个执行序列，其父进程应是分支进程的父进程。关于这一点，代码的表述比文字要容易：

kernel/fork.c

```
        if (clone_flags & (CLONE_PARENT|CLONE_THREAD))
                p->real_parent = current->real_parent;
        else
                p->real_parent = current;
        p->parent = p->real_parent;
```

非线程的普通进程可通过设置CLONE_PARENT触发同样的行为。对线程来说还需要另一个校正，即普通进程的线程组组长是进程本身。对线程来说，其组长是当前进程的组长：

kernel/fork.c

```
    p->group_leader = p;

        if (clone_flags & CLONE_THREAD) {
                p->group_leader = current->group_leader;
                list_add_tail_rcu(&p->thread_group, &p->group_leader->thread_group);
...
}
```

新进程接下来必须通过children链表与父进程连接起来。这是通过辅助宏add_parent处理的。此外，新进程必须被归入2.3.3节描述的ID数据结构体系中。

kernel/fork.c

```
        add_parent(p);

        if (thread_group_leader(p)) {
                if (clone_flags & CLONE_NEWPID)
                        p->nsproxy->pid_ns->child_reaper = p;

                set_task_pgrp(p, task_pgrp_nr(current));
                set_task_session(p, task_session_nr(current));
                attach_pid(p, PIDTYPE_PGID, task_pgrp(current));
                attach_pid(p, PIDTYPE_SID, task_session(current));
        }

        attach_pid(p, PIDTYPE_PID, pid);
...
        return p;
}
```

thread_group_leader只检查新进程的pid和tgid是否相同。倘若如此，则该进程是线程组的组长。在这种情况下，还需要完成更多必要的工作。

- 回想一下，在非全局命名空间的进程命名空间中，各个进程有特定于该命名空间的init进程。如果通过置位CLONE_NEWPID创建一个新的PID命名空间，那么init进程的角色必须由调用clone的进程承担。
- 新进程必须被加到当前进程组和会话。这样就需要用到前文讨论过的一些函数。

最后，PID本身被加到ID数据结构的体系中。创建新进程的工作就此完成！

5. 创建线程时的特别问题

用户空间线程库使用clone系统调用来生成新线程。该调用支持（上文讨论之外的）标志，对copy_process（及其调用的函数）具有某些特殊影响。为简明起见，我在上文中省去了这些标志。但有一点应该记住，在Linux内核中，线程和一般进程之间的差别不是那么刚性，这两个名词经常用作同义词（如前所述，线程也经常用于指进程的体系结构相关部分）。在本节中，我重点讲解用户线

程库（尤其是NPTL）用于实现多线程功能的标志。

- CLONE_PARENT_SETTID将生成线程的PID复制到clone调用指定的用户空间中的某个地址（parent_tidptr，传递到clone的指针）[①]：

  kernel/fork.c

```
        if (clone_flags & CLONE_PARENT_SETTID)
                put_user(nr, parent_tidptr);
```

  复制操作在do_fork中执行，此时新线程的task_struct尚未初始化，copy操作尚未创建新线程的数据。

- CLONE_CHILD_SETTID首先会将另一个传递到clone的用户空间指针（child_tidptr）保存在新进程的task_struct中。

  kernel/fork.c

```
p->set_child_tid = (clone_flags & CLONE_CHILD_SETTID) ? child_tidptr : NULL;
```

  在新进程第一次执行时，内核会调用schedule_tail函数将当前PID复制到该地址。

  kernel/schedule.c

```
asmlinkage void schedule_tail(struct task_struct *prev)
{
...
        if (current->set_child_tid)
                put_user(task_pid_vnr(current), current->set_child_tid);
...
}
```

- CLONE_CHILD_CLEARTID首先会在copy_process中将用户空间指针child_tidptr保存在task_struct中，这次是另一个不同的成员。

  kernel/fork.c

```
p->clear_child_tid = (clone_flags & CLONE_CHILD_CLEARTID) ? child_tidptr: NULL;
```

  在进程终止时，[②] 将0写入clear_child_tid指定的地址。[③]

  kernel/fork.c

```
void mm_release(struct task_struct *tsk, struct mm_struct *mm)
{
        if (tsk->clear_child_tid
            && atomic_read(&mm->mm_users) > 1) {
                u32 __user * tidptr = tsk->clear_child_tid;
                tsk->clear_child_tid = NULL;

                put_user(0, tidptr);
                sys_futex(tidptr, FUTEX_WAKE, 1, NULL, NULL, 0);
        }
...
}
```

  此外，sys_futex，一个快速的用户空间互斥量，用于唤醒等待线程结束事件的进程。

上述标志可用于从用户空间检测内核中线程的产生和销毁。CLONE_CHILD_SETTID和CLONE_PARENT_SETTID用于检测线程的生成。CLONE_CHILD_CLEARTID用于在线程结束时从内核向用户空间

① put_user用于在内核地址空间和用户地址空间之间复制数据，将在第4章讨论。

② 或更精确地说，在进程终止过程中，使用mm_release自动释放其用于内存管理的数据结构时。

③ 条件mm->mm_users > 1意味着系统中至少有另一个进程在使用该内存管理数据结构。因此当前进程是一般意义上的一个线程，其地址空间来自另一个进程，且只有一个控制流。

传递信息。在多处理器系统上这些检测可以真正地并行执行。

2.4.2 内核线程

内核线程是直接由内核本身启动的进程。内核线程实际上是将内核函数委托给独立的进程，与系统中其他进程“并行”执行（实际上，也并行于内核自身的执行）。[①]内核线程经常称之为（内核）守护进程。它们用于执行下列任务。

- 周期性地将修改的内存页与页来源块设备同步（例如，使用mmap的文件映射）。
- 如果内存页很少使用，则写入交换区。
- 管理延时动作（deferred action）。
- 实现文件系统的事务日志。

基本上，有两种类型的内核线程。

- **类型1**：线程启动后一直等待，直至内核请求线程执行某一特定操作。
- **类型2**：线程启动后按周期性间隔运行，检测特定资源的使用，在用量超出或低于预置的限制值时采取行动。内核使用这类线程用于连续监测任务。

调用kernel_thread函数可启动一个内核线程。其定义是特定于体系结构的，但原型总是相同的。

<asm-*arch*/processor.h>

```
int kernel_thread(int (*fn)(void *), void * arg, unsigned long flags)
```

产生的线程将执行用fn指针传递的函数，而用arg指定的参数将自动传递给该函数。[②]flags中可指定CLONE标志。

kernel_thread的第一个任务是构建一个pt_regs实例，对其中的寄存器指定适当的值，这与普通的fork系统调用类似。接下来调用我们熟悉的do_fork函数。

```
p = do_fork(flags | CLONE_VM | CLONE_UNTRACED, 0, &regs, 0, NULL, NULL);
```

因为内核线程是由内核自身生成的，应该注意下面两个特别之处。

(1) 它们在CPU的管态（supervisor mode）执行，而不是用户状态（参见第1章）。

(2) 它们只可以访问虚拟地址空间的内核部分（高于TASK_SIZE的所有地址），但不能访问用户空间。

回想上文的内容，可知task_struct中包含了指向mm_structs的两个指针：

<sched.h>

```
struct task_struct {
...
        struct mm_struct *mm, *active_mm;
...
}
```

大多数计算机上系统的全部虚拟地址空间分成两个部分：底部可以由用户层程序访问，上部则专供内核使用。在内核代表用户层程序运行时（例如，执行系统调用），虚拟地址空间的用户空间部分由mm指向的mm_struct实例描述（该结构的具体内容与当前无关，会在第4章讨论）。每当内核执行上下文切换时，虚拟地址空间的用户层部分都会切换，以便与当前运行的进程匹配。

这为优化提供了一些余地，可遵循所谓的惰性TLB处理（lazy TLB handling）。由于内核线程不与任何特定的用户层进程相关，内核并不需要倒换虚拟地址空间的用户层部分，保留旧设置即可。由

① 在多处理系统上，进程是真正并行执行的。在单处理器系统上，调度器模拟并行执行。

② 通过参数表示需要完成的工作，这使得函数可用于不同目的。

于内核线程之前可能是任何用户层进程在执行，因此用户空间部分的内容本质上是随机的，内核线程绝不能修改其内容。为强调用户空间部分不能访问，mm设置为空指针。但由于内核必须知道用户空间当前包含了什么，所以在active_mm中保存了指向mm_struct的一个指针来描述它。

为什么没有mm指针的进程称作惰性TLB进程？假如内核线程之后运行的进程与之前是同一个。在这种情况下，内核并不需要修改用户空间地址表，地址转换后备缓冲器（即TLB）中的信息仍然有效。只有在内核线程之后执行的进程是与此前不同的用户层进程时，才需要切换（并对应清除TLB数据）。

请注意，当内核在进程上下文下运转时，mm和active_mm的值相同。

内核线程可以用两种方法实现。古老的方法：内核中一些地方仍然在使用该方法，将一个函数直接传递给kernel_thread。该函数接下来负责帮助内核调用daemonize以转换为守护进程。这依次引发下列操作.

(1) 该函数从内核线程释放其父进程（用户进程）的所有资源（例如，内存上下文、文件描述符,等等），不然这些资源会一直锁定到线程结束，这是不可取的，因为守护进程通常运行到系统关机为止。因为守护进程只操作内核地址区域，它甚至不需要这些资源。

(2) daemonize阻塞信号的接收。

(3) 将init用作守护进程的父进程。

创建内核线程更现代的方法是辅助函数kthread_create。

kernel/kthread.c

```
struct task_struct *kthread_create(int (*threadfn)(void *data),
                                   void *data,
                                   const char namefmt[],
                                   ...)
```

该函数创建一个新的内核线程，其名称由namefmt给出。最初该线程是停止的，需要使用wake_up_process启动它。此后，会调用通过threadfn给出的线程函数，而data则作为参数。

另一个备选方案是宏kthread_run（参数与kthread_create相同），它会调用kthread_create创建新线程，但立即唤醒它。还可以使用kthread_create_cpu代替kthread_create创建内核线程，使之绑定到特定的CPU。

内核线程会出现在系统进程列表中，但在ps的输出中由方括号包围，以便与普通进程区分。

```
wolfgang@meitner> ps fax
  PID TTY STAT TIME COMMAND
    2?    S<   0:00 [kthreadd]
    3?    S<   0:00 _ [migration/0]
    4?    S<   0:00 _ [ksoftirqd/0]
    5?    S<   0:00 _ [migration/1]
    6?    S<   0:00 _ [ksoftirqd/1]
...
   52?    S<   0:00 _ [kblockd/3]
   55?    S<   0:00 _ [kacpid]
   56?    S<   0:00 _ [kacpi_notify]
...
```

如果内核线程绑定到特定的CPU，CPU的编号在斜线后给出。

2.4.3 启动新程序

通过用新代码替换现存程序，即可启动新程序。Linux提供的execve系统调用可用于该目的。[①]

① C标准库中有其他exec变体，但最终都基于execve。在前述章节中，exec经常用于指代这些变体之一。

1. execve的实现

该系统调用的入口点是体系结构相关的sys_execve函数。该函数很快将其工作委托给系统无关的do_execve例程。

kernel/exec.c

```
int do_execve(char * filename,
              char __user *__user *argv,
              char __user *__user *envp,
              struct pt_regs * regs)
```

这里不仅用参数传递了寄存器集合和可执行文件的名称（filename），而且还传递了指向程序的参数和环境的指针。[①]这里的记号稍微有点笨拙，因为argv和envp都是指针数组，而且指向两个数组自身的指针以及数组中的所有指针都位于虚拟地址空间的用户空间部分。回想一下第1章的内容，可知在内核访问用户空间内存时需要多加小心，而__user注释则允许自动化工具来检测是否所有相关事宜都处理得当。

图2-11给出了do_execve的代码流程图。

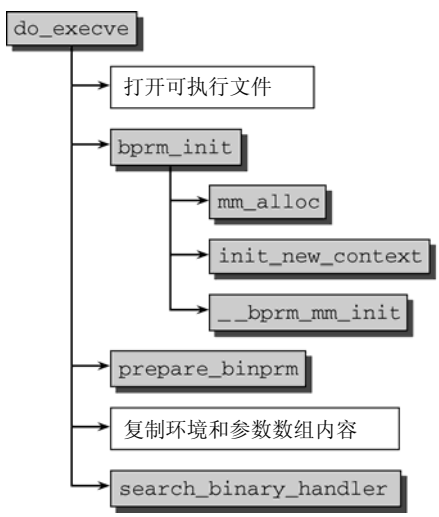

图2-11 do_execve的代码流程图

首先打开要执行的文件。换言之，按第8章的说法，内核找到相关的inode并生成一个文件描述符，用于寻址该文件。

bprm_init接下来处理若干管理性任务：mm_alloc生成一个新的mm_struct实例来管理进程地址空间（参见第4章）。init_new_context是一个特定于体系结构的函数，用于初始化该实例，而__bprm_mm_init则建立初始的栈。

新进程的各个参数（例如，euid、egid、参数列表、环境、文件名，等等）随后会分别传递给其他函数，此时为简明起见，则合并成一个类型为linux_binprm的结构。prepare_binprm用于提供一些父进程相关的值（特别是有效UID和GID）。剩余的数据，即参数列表，接下来直接复制到该结构

① argv包含在命令行上传递给该程序的所有参数（例如，对于ls -l /usr/bin来说，就是-l和/usr/bin）。环境则包括了在程序执行时定义的所有环境变量。在大多数shell中，可以使用set输出这些变量的列表。

中。要注意prepare_binprm也维护了对SUID和SGID位的处理：

fs/exec.c

```
int prepare_binprm(struct linux_binprm *bprm)
{
...
        bprm->e_uid = current->euid;
        bprm->e_gid = current->egid;

        if(!(bprm->file->f_vfsmnt->mnt_flags & MNT_NOSUID)) {
                /* Set-uid? */
                if (mode & S_ISUID) {
                        bprm->e_uid = inode->i_uid;
                }

                /* Set-gid? */
                /*
                 *如果setgid置位但组执行位没有置位，那么这可能是强制锁定，
                 *而不是setgid的可执行文件。
                 */
                if ((mode & (S_ISGID | S_IXGRP)) == (S_ISGID | S_IXGRP)){
                        bprm->e_gid = inode->i_gid;
                }
        }
...
}
```

在确认文件来源卷在装载时没有置位MNT_NOSUID之后，内核会检测SUID或SGID位是否置位。第一种情况很容易处理：如果S_ISUID置位，那么有效UID与inode相同（否则，使用进程的有效UID）。SGID的情况类似，但内核还需要确认组执行位也已经置位。

Linux支持可执行文件的各种不同组织格式。标准格式是ELF（Executable and Linkable Format），我会在附录E详细论述。其他的备选格式是表2-2列出的各种变体（表中列出了内核中对应的linux_binfmt实例的名称）。

尽管在不同的体系结构上可能使用许多二进制格式（ELF尽可能设计得与系统无关），这并不意味着特定二进制格式中的程序能够在多个体系结构上运行。不同处理器使用的汇编语言语句仍然非常不同，而二进制格式只表示如何在可执行文件和内存中组织程序的各个部分（数据、代码，等等）。

search_binary_handler用于在do_execve结束时查找一种适当的二进制格式，用于所要执行的特定文件。这种查找是可能的，因为各种格式可根据不同的特点来识别（通常是文件起始处的一个“魔数”）。二进制格式处理程序负责将新程序的数据加载到旧的地址空间中。附录E针对ELF格式描述了加载的步骤。通常，二进制格式处理程序执行下列操作。

- 释放原进程使用的所有资源。
- 将应用程序映射到虚拟地址空间中。必须考虑下列段的处理（涉及的变量是task_struct的成员，由二进制格式处理程序设置为正确的值）。
  - text段包含程序的可执行代码。start_code和end_code指定该段在地址空间中驻留的区域。
  - 预先初始化的数据（在编译时间指定了具体值的变量）位于start_data和end_data之间，映射自可执行文件的对应段。
  - 堆（heap）用于动态内存分配，亦置于虚拟地址空间中。start_brk和brk指定了其边界。
  - 栈的位置由start_stack定义。几乎所有的计算机上栈都是自动地向下增长。唯一的例外是当前的PA-Risc。对于栈的反向增长，体系结构相关部分的实现必须告知内核，可通过设置配置符号STACK_GROWSUP完成。

- 程序的参数和环境也映射到虚拟地址空间中，分别位于arg_start和arg_end之间，以及env_start和env_end之间。

❑ 设置进程的指令指针和其他特定于体系结构的寄存器，以便在调度器选择该进程时开始执行程序的main函数。

有关ELF格式到虚拟地址空间的映射，将在4.2.1节更详细地讨论。

表2-2 Linux支持的二进制格式

名 称	含 义
flat_format	平坦格式用于没有内存管理单元（MMU）的嵌入式CPU上。为节省空间，可执行文件中的数据还可以压缩（如果内核可提供zlib支持）
script_format	这是一种伪格式，用于运行使用#!机制的脚本。检查文件的第一行，内核即知道使用何种解释器，启动适当的应用程序即可（例如，如果是#! /usr/bin/perl，则启动Perl）
misc_format	这也是一种伪格式，用于启动需要外部解释器的应用程序。与#!机制相比，解释器无须显式指定，而可以通过特定的文件标识符（后缀、文件头，等等）确定。例如，该格式用于执行Java字节代码或用Wine运行Windows程序
elf_format	这是一种与计算机和体系结构无关的格式，可用于32位和64位。它是Linux的标准格式
elf_fdpic_format	ELF格式变体，提供了针对没有MMU系统的特别特性
irix_format	ELF格式变体，提供了特定于Irix的特性
som_format	在PA-Risc计算机上使用，特定于HP-UX的格式
aout_format	a.out是引入ELF之前Linux的标准格式。因为它太不灵活，所以现在很少使用

2. 解释二进制格式

在Linux内核中，每种二进制格式都表示为下列数据结构（已经简化过）的一个实例：

<binfmts.h>

```
struct linux_binfmt {
        struct linux_binfmt * next;
        struct module *module;
        int (*load_binary)(struct linux_binprm *, struct pt_regs * regs);
        int (*load_shlib)(struct file *);
        int (*core_dump)(long signr, struct pt_regs * regs, struct file * file);
        unsigned long min_coredump;      /* minimal dump size */
};
```

每种二进制格式必须提供下面3个函数。

(1) load_binary用于加载普通程序。

(2) load_shlib用于加载共享库，即动态库。

(3) core_dump用于在程序错误的情况下输出内存转储。该转储随后可使用调试器（例如，gdb）分析，以便解决问题。min_coredump是生成内存转储时，内存转储文件长度的下界（通常，这是一个内存页的长度）。

每种二进制格式首先必须使用register_binfmt向内核注册。该函数的目的是向一个链表增加一种新的二进制格式，该链表的表头是fs/exec.c中的全局变量formats。linux_binfmt实例通过其next成员彼此连接起来。

2.4.4 退出进程

进程必须用exit系统调用终止。这使得内核有机会将该进程使用的资源释放回系统。[①] 该调用的

① 程序员可以显式调用exit。但编译器会在main函数（或特定语言使用的main函数）末尾自动添加相应的调用。

入口点是sys_exit函数，需要一个错误码作为其参数，以便退出进程。其定义是体系结构无关的，见kernel/exit.c。我们对其实现没什么兴趣，因为它很快将工作委托给do_exit。

简而言之，该函数的实现就是将各个引用计数器减1，如果引用计数器归0而没有进程再使用对应的结构，那么将相应的内存区域返还给内存管理模块。

2.5 调度器的实现

内存中保存了对每个进程的唯一描述，并通过若干结构与其他进程连接起来。调度器面对的情形就是这样，其任务是在程序之间共享CPU时间，创造并行执行的错觉。正如以上的讨论，该任务分为两个不同部分：一个涉及调度策略，另一个涉及上下文切换。

2.5.1 概观

内核必须提供一种方法，在各个进程之间尽可能公平地共享CPU时间，而同时又要考虑不同的任务优先级。完成该目的有许多方法，各有其利弊，我们无须在此讨论（对可能方法的概述，请参见[Tan07]）。我们主要关注Linux内核采用的解决方案。

schedule函数是理解调度操作的起点。该函数定义在kernel/sched.c中，是内核代码中最常调用的函数之一。调度器的实现受若干因素的影响而稍显模糊。

- 在多处理器系统上，必须要注意几个细节（有一些非常微妙），以避免调度器自相干扰。
- 不仅实现了*优先调度*，还实现了Posix标准需要的其他两种软实时策略。
- 使用goto以生成最优的汇编语言代码。这些语句在C代码中来回地跳转，与结构化程序设计的所有原理背道而驰。但如果小心翼翼地使用它，该特性就可以发挥作用（调度器就是一个例子）。

下面我暂时忽略实时进程，只考虑完全公平调度器（稍后再考虑实时进程）。Linux调度器的一个杰出特性是，它不需要时间片概念，至少不需要传统的时间片。经典的调度器对系统中的进程分别计算时间片，使进程运行直至时间片用尽。在所有进程的所有时间片都已经用尽时，则需要重新计算。相比之下，当前的调度器只考虑进程的等待时间，即进程在就绪队列（run-queue）中已经等待了多长时间。对CPU时间需求最严格的进程被调度执行。

调度器的一般原理是，按所能分配的计算能力，向系统中的每个进程提供最大的公正性。或者从另一个角度来说，它试图确保没有进程被亏待。这听起来不错，但就CPU时间而论，*公平与否*意味着什么呢？考虑一台理想计算机，可以并行运行任意数目的进程。如果系统上有N个进程，那么每个进程得到总计算能力的1/N，所有的进程在物理上真实地并行执行。假如一个进程需要10分钟完成其工作。如果5个这样的进程在理想CPU上同时运行，每个会得到计算能力的20%，这意味着每个进程需要运行50分钟，而不是10分钟。但所有的5个进程都会刚好在该时间段之后结束其工作，没有哪个进程在此段时间内处于不活动状态！

在真正的硬件上这显然是无法实现的。如果系统只有一个CPU，至多可以同时运行一个进程。只能通过在各个进程之间高频率来回切换，来实现多任务。对用户来说，由于其思维比转换频率慢得多，切换造成了并行执行的错觉，但实际上不存在并行执行。虽然多CPU系统能改善这种情况并完美地并行执行少量进程，但情况总是CPU数目比要运行的进程数目少，这样上述问题又出现了。

如果通过轮流运行各个进程来模拟多任务，那么当前运行的进程，其待遇显然好于哪些等待调度器选择的进程，即等待的进程受到了不公平的对待。不公平的程度正比于等待时间。

每次调用调度器时，它会挑选具有最高等待时间的进程，把CPU提供给该进程。如果经常发生这

种情况，那么进程的不公平待遇不会累积，不公平会均匀分布到系统中的所有进程。

图2-12说明了调度器如何记录哪个进程已经等待了多长时间。由于可运行进程是排队的，该结构称之为就绪队列。

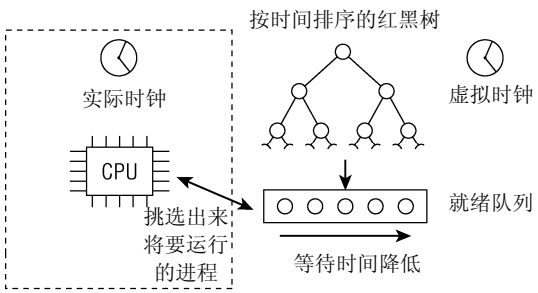

图2-12　调度器通过将进程在红黑树中排序，跟踪进程的等待时间

所有的可运行进程都按时间在一个红黑树中排序，所谓时间即其等待时间。等待CPU时间最长的进程是最左侧的项，调度器下一次会考虑该进程。等待时间稍短的进程在该树上从左至右排序。

如果读者不熟悉红黑树，知道以下这些也足够了。该数据结构对所包含的项提供了高效的管理，该树管理的进程数目增加时，查找、插入、删除操作需要的时间只会适度地增加。[①]红黑树是内核的标准数据结构，附录C提供了更多有关的信息。此外，红黑树的内容在每一本数据结构教科书中都可以找到。

除了红黑树外，就绪队列还装备了虚拟时钟。[②]该时钟的时间流逝速度慢于实际的时钟，精确的速度依赖于当前等待调度器挑选的进程的数目。假定该队列上有4个进程，那么虚拟时钟将以实际时钟四分之一的速度运行。如果以完全公平的方式分享计算能力，那么该时钟是判断等待进程将获得多少CPU时间的基准。在就绪队列等待实际的20秒，相当于虚拟时间5秒。4个进程分别执行5秒，即可使CPU被实际占用20秒。

假定就绪队列的虚拟时间由`fair_clock`给出，而进程的等待时间保存在`wait_runtime`。为排序红黑树上的进程，内核使用差值`fair_clock - wait_runtime`。`fair_clock`是完全公平调度的情况下进程将会得到的CPU时间的度量，而`wait_runtime`直接度量了实际系统的不足造成的不公平。

在进程允许运行时，将从`wait_runtime`减去它已经运行的时间。这样，在按时间排序的树中它会向右移动到某一点，另一个进程将成为最左边，下一次会被调度器选择。但请注意，在进程运行时`fair_clock`中的虚拟时钟会增加。这实际上意味着，进程在完全公平的系统中接收的CPU时间份额，是推演自在实际的CPU上执行花费的时间。这减缓了削弱不公平状况的过程：减少`wait_runtime`等价于降低进程受到的不公平对待的数量，但内核无论如何不能忘记，用于降低不公平性的一部分时间，实际上属于处于完全公平世界中的进程。 再次假定就绪队列上有4个进程，而一个进程实际上已经等待了20秒。现在它允许运行10秒：此后的`wait_runtime`是10，但由于该进程无论如何都会得到该时

① 确切地说，时间复杂度是$O(\log n)$，n是树中结点的数目。这比原调度器的性能要差，后者以$O(1)$调度器著称，即其运行时间与需要处理的进程的数目无关。但除非大量进程同时处于可运行状态，否则新调度器的对数级时间造成的性能下降是可以忽略的。实际上，这种情况不会发生。

② 请注意，内核2.6.23的调度机制确实使用了虚拟时钟的概念，但当前版本对虚拟时间的计算稍有不同。由于用虚拟时钟来说明易于理解该方法，我会一直使用该概念。在讨论调度器实现时，我将讲述如何模拟虚拟时钟。

间段中的10/4 = 2秒，因此实际上只有8秒对该进程在就绪队列中的新位置起了作用。

遗憾的是，该策略受若干现实问题的影响，已经变得复杂了。

- 进程的不同优先级（即，nice值）必须考虑，更重要的进程必须比次要进程更多的CPU时间份额。
- 进程不能切换得太频繁，因为上下文切换，即从一个进程改变到另一个，是有一定开销的。在切换发生得太频繁时，过多时间花费在进程切换的过程中，而不是用于实际的工作。另一方面，两次相邻的任务切换之间，时间也不能太长，否则会累积比较大的不公平值。对多媒体系统来说，进程运行太长时间也会导致延迟增大。

在下面的讨论中，我们会看到调度器解决这些问题的方案。

理解调度决策的一个好方法是，在编译时激活调度器统计。这会在运行时生成文件/proc/sched_debug，其中包含了调度器当前状态所有方面的信息。

最后要注意，Documentation/目录下包含了一些文件，涉及调度器的各个方面。但切记，其中一些仍然讲述的是旧的*O*(1)调度器，已经过时了！

2.5.2 数据结构

调度器使用一系列数据结构，来排序和管理系统中的进程。调度器的工作方式与这些结构的设计密切相关。几个组件在许多方面彼此交互，图2-13概述了这些组件的关联。

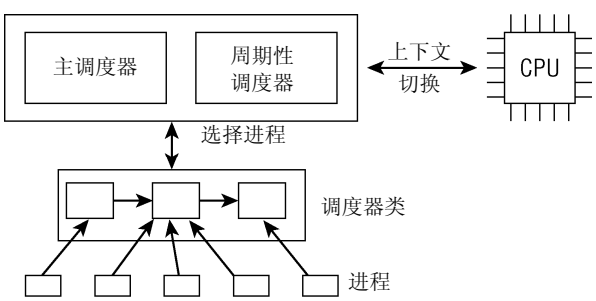

图2-13 调度子系统各组件概观

可以用两种方法激活调度。一种是直接的，比如进程打算睡眠或出于其他原因放弃CPU；另一种是通过周期性机制，以固定的频率运行，不时检测是否有必要进行进程切换。在下文中我将这两个组件称为通用调度器（generic scheduler）或核心调度器（core scheduler）。本质上，通用调度器是一个分配器，与其他两个组件交互。

(1) 调度类用于判断接下来运行哪个进程。内核支持不同的调度策略（完全公平调度、实时调度、在无事可做时调度空闲进程），调度类使得能够以模块化方法实现这些策略，即一个类的代码不需要与其他类的代码交互。

在调度器被调用时，它会查询调度器类，得知接下来运行哪个进程。

(2) 在选中将要运行的进程之后，必须执行底层任务切换。这需要与CPU的紧密交互。

每个进程都刚好属于某一调度类，各个调度类负责管理所属的进程。通用调度器自身完全不涉及进程管理，其工作都委托给调度器类。

1. task_struct的成员

各进程的task_struct有几个成员与调度相关。

<sched.h>

```
struct task_struct {
...
        int prio, static_prio, normal_prio;
        unsigned int rt_priority;

        struct list_head run_list;
        const struct sched_class *sched_class;
        struct sched_entity se;

        unsigned int policy;
        cpumask_t cpus_allowed;
        unsigned int time_slice;
...
}
```

- 并非系统上的所有进程都同样重要。不那么紧急的进程不需要太多关注，而重要的工作应该尽可能快速完成。为确定特定进程的重要性，我们给进程增加了相对优先级属性。

  但task_struct采用了3个成员来表示进程的优先级：prio和normal_prio表示动态优先级，static_prio表示进程的静态优先级。静态优先级是进程启动时分配的优先级。它可以用nice和sched_setscheduler系统调用修改，否则在进程运行期间会一直保持恒定。

  normal_priority表示基于进程的静态优先级和调度策略计算出的优先级。因此，即使普通进程和实时进程具有相同的静态优先级，其普通优先级也是不同的。进程分支时，子进程会继承普通优先级。

  但调度器考虑的优先级则保存在prio。由于在某些情况下内核需要暂时提高进程的优先级，因此需要第3个成员来表示。由于这些改变不是持久的，因此静态和普通优先级不受影响。这3个优先级彼此的依赖关系稍微有一点微妙，我会在下文详细讲述。
- rt_priority表示实时进程的优先级。该值不会代替先前讨论的那些值！最低的实时优先级为0，而最高的优先级是99。值越大，表明优先级越高。这里使用的惯例不同于nice值。
- sched_class表示该进程所属的调度器类。
- 调度器不限于调度进程，还可以处理更大的实体。这可以用于实现组调度：可用的CPU时间可以首先在一般的进程组（例如，所有进程可以按所有者分组）之间分配，接下来分配的时间在组内再次分配。

  这种一般性要求调度器不直接操作进程，而是处理可调度实体。一个实体由sched_entity的一个实例表示。

  在最简单的情况下，调度在各个进程上执行，这也是我们最初关注的情形。由于调度器设计为处理可调度的实体，在调度器看来各个进程必须也像是这样的实体。因此se在task_struct中内嵌了一个sched_entity实例，调度器可据此操作各个task struct（请注意se不是一个指针，因为该实体嵌入在task_struct中）。
- policy保存了对该进程应用的调度策略。Linux支持5个可能的值。
  - SCHED_NORMAL用于普通进程，我们主要讲述此类进程。它们通过完全公平调度器来处理。SCHED_BATCH和SCHED_IDLE也通过完全公平调度器来处理，不过可用于次要的进程。SCHED_BATCH用于非交互、CPU使用密集的批处理进程。调度决策对此类进程给予“冷处理”：它们决不会抢占CF调度器处理的另一个进程，因此不会干扰交互式进程。如果不打算用nice降低进程的静态优先级，同时又不希望该进程影响系统的交互性，此时最适合使用该调度类。

在调度决策中SCHED_IDLE进程的重要性也比较低，因为其相对权重总是最小的（在论述内核如何计算反映进程优先级的权重时，这一点就很清楚了）。

要注意，尽管名称是SCHED_IDLE，但SCHED_IDLE不负责调度空闲进程。空闲进程由内核提供单独的机制来处理。

- SCHED_RR和SCHED_FIFO用于实现软实时进程。SCHED_RR实现了一种循环方法，而SCHED_FIFO则使用先进先出机制。这些不是由完全公平调度器类处理，而是由实时调度器类处理，2.7节会详细论述。

  辅助函数rt_policy用于判断给出的调度策略是否属于实时类（SCHED_RR和SCHED_FIFO）。task_has_rt_policy用于对给定进程判断该性质。

kernel/sched.c

```
static inline int rt_policy(int policy)
static inline int task_has_rt_policy(struct task_struct *p)
```

- cpus_allowed是一个位域，在多处理器系统上使用，用来限制进程可以在哪些CPU上运行。[①]
- run_list和time_slice是循环实时调度器所需要的，但不用于完全公平调度器。run_list是一个表头，用于维护包含各进程的一个运行表，而time_slice则指定进程可使用CPU的剩余时间段。

前文讨论的TIF_NEED_RESCHED标志，对调度器而言，和task_struct中上述与调度相关的成员同样重要。如果对活动进程设置该标志，调度器即知道CPU将从该进程收回并授予新进程，这可能是自愿的，也可能是强制的。

2. 调度器类

调度器类提供了通用调度器和各个调度方法之间的关联。调度器类由特定数据结构中汇集的几个函数指针表示。全局调度器请求的各个操作都可以由一个指针表示。这使得无需了解不同调度器类的内部工作原理，即可创建通用调度器。

除去针对多处理器系统的扩展（我在后文再考虑这些），该结构如下所示：

<sched.h>

```
struct sched_class {
        const struct sched_class *next;

        void (*enqueue_task) (struct rq *rq, struct task_struct *p, int wakeup);
        void (*dequeue_task) (struct rq *rq, struct task_struct *p, int sleep);
        void (*yield_task) (struct rq *rq);

        void (*check_preempt_curr) (struct rq *rq, struct task_struct *p);

        struct task_struct * (*pick_next_task) (struct rq *rq);
        void (*put_prev_task) (struct rq *rq, struct task_struct *p);
        void (*set_curr_task) (struct rq *rq);
        void (*task_tick) (struct rq *rq, struct task_struct *p);
        void (*task_new) (struct rq *rq, struct task_struct *p);
};
```

对各个调度类，都必须提供struct sched_class的一个实例。调度类之间的层次结构是平坦的：实时进程最重要，在完全公平进程之前处理；而完全公平进程则优先于空闲进程；空闲进程只有CPU

① 可使用sched_setaffinity系统调用设置该位图。

无事可做时才处于活动状态。next成员将不同调度类的sched_class实例，按上述顺序连接起来。要注意这个层次结构在编译时已经建立：没有在运行时动态增加新调度器类的机制。

下面是各个调度类可以提供的操作。

- enqueue_task向就绪队列添加一个新进程。在进程从睡眠状态变为可运行状态时，即发生该操作。
- dequeue_task提供逆向操作，将一个进程从就绪队列去除。事实上，在进程从可运行状态切换到不可运行状态时，就会发生该操作。内核有可能因为其他理由将进程从就绪队列去除，比如，进程的优先级可能需要改变。

  尽管使用了术语就绪队列（run queue），各个调度类无须用简单的队列来表示其进程。实际上，回想上文，可知完全公平调度器对此使用了红黑树。
- 在进程想要自愿放弃对处理器的控制权时，可使用sched_yield系统调用。这导致内核调用yield_task。
- 在必要的情况下，会调用check_preempt_curr，用一个新唤醒的进程来抢占当前进程。例如，在用wake_up_new_task唤醒新进程时，会调用该函数。
- pick_next_task用于选择下一个将要运行的进程，而put_prev_task则在用另一个进程代替当前运行的进程之前调用。要注意，这些操作并不等价于将进程加入或撤出就绪队列的操作，如enqueue_task和dequeue_task。相反，它们负责向进程提供或撤销CPU。但在不同进程之间切换，仍然需要执行一个底层的上下文切换。
- 在进程的调度策略发生变化时，需要调用set_curr_task。还有其他一些场合也调用该函数，但与我们的目的无关。
- task_tick在每次激活周期性调度器时，由周期性调度器调用。
- new_task用于建立fork系统调用和调度器之间的关联。每次新进程建立后，则用new_task通知调度器。

标准函数activate_task和deactivate_task调用前述的函数，提供进程在就绪队列的入队和离队功能。此外，它们还更新内核的统计数据。

kernel/sched.c

```
static void enqueue_task(struct rq *rq, struct task_struct *p, int wakeup)
static void dequeue_task(struct rq *rq, struct task_struct *p, int sleep)
```

在进程注册到就绪队列时，嵌入的sched_entity实例的on_rq成员设置为1，否则为0。

此外，内核定义了便捷方法check_preempt_curr，调用与给定进程相关的调度类的check_preempt_curr方法：

kernel/sched.c

```
static inline void check_preempt_curr(struct rq *rq, struct task_struct *p)
```

用户层应用程序无法直接与调度类交互。它们只知道上文定义的常量SCHED_xyz。在这些常量和可用的调度类之间提供适当的映射，这是内核的工作。SCHED_NORMAL、SCHED_BATCH和SCHED_IDLE映射到fair_sched_class，而SCHED_RR和SCHED_FIFO与rt_sched_class关联。fair_sched_class和rt_sched_class都是struct sched_class的实例，分别表示完全公平调度器和实时调度器。当我详细论述相应的调度器类时，会给出相关实例的内容。

3. 就绪队列

核心调度器用于管理活动进程的主要数据结构称之为就绪队列。各个CPU都有自身的就绪队列，

各个活动进程只出现在一个就绪队列中。在多个CPU上同时运行一个进程是不可能的。[①]

就绪队列是全局调度器许多操作的起点。但要注意，进程并不是由就绪队列的成员直接管理的！这是各个调度器类的职责，因此在各个就绪队列中嵌入了特定于调度器类的子就绪队列。[②]

就绪队列是使用下列数据结构实现的。为简明起见，我省去了几个用于统计、不直接影响就绪队列工作的成员，以及在多处理器系统上所需要的成员。

kernel/sched.c

```
struct rq {
        unsigned long nr_running;
        #define CPU_LOAD_IDX_MAX 5
        unsigned long cpu_load[CPU_LOAD_IDX_MAX];
...
        struct load_weight load;

        struct cfs_rq cfs;
        struct rt_rq rt;

        struct task_struct *curr, *idle;
        u64 clock;
...
};
```

- nr_running指定了队列上可运行进程的数目，不考虑其优先级或调度类。
- load提供了就绪队列当前负荷的度量。队列的负荷本质上与队列上当前活动进程的数目成正比，其中的各个进程又有优先级作为权重。每个就绪队列的虚拟时钟的速度即基于该信息。由于负荷及其他相关数量的计算是调度算法的一个重要部分，下文的2.5.3节会详细讨论涉及的机制。
- cpu_load用于跟踪此前的负荷状态。
- cfs和rt是嵌入的子就绪队列，分别用于完全公平调度器和实时调度器。
- curr指向当前运行的进程的task_struct实例。
- idle指向空闲进程的task_struct实例，该进程亦称之为空闲线程，在其他可运行进程时执行。
- clock和prev_raw_clock用于实现就绪队列自身的时钟。每次调用周期性调度器时，都会更新clock的值。另外内核还提供了标准函数update_rq_clock，可在操作就绪队列的调度器中多处调用，例如，在用wakeup_new_task唤醒新进程时。

系统的所有就绪队列都在runqueues数组中，该数组的每个元素分别对应于系统中的一个CPU。在单处理器系统中，由于只需要一个就绪队列，数组只有一个元素。

kernel/sched.c

```
static DEFINE_PER_CPU_SHARED_ALIGNED(struct rq, runqueues);
```

内核也定义了一些便利的宏，其含义很明显。

kernel/sched.c

```
#define cpu_rq(cpu)      (&per_cpu(runqueues, (cpu)))
#define this_rq()        (&__get_cpu_var(runqueues))
#define task_rq(p)       cpu_rq(task_cpu(p))
#define cpu_curr(cpu)    (cpu_rq(cpu)->curr)
```

① 但发源于同一进程的各线程可以在不同处理器上执行，因为进程管理对进程和线程不作重要的区分。

② 对于熟悉内核早期版本的读者来说，了解调度器类和就绪队列代替了先前的*O*(1)调度器使用的活动和到期进程列表，还是颇有趣味的。

4. 调度实体

由于调度器可以操作比进程更一般的实体，因此需要一个适当的数据结构来描述此类实体。其定义如下：

<sched.h>

```
struct sched_entity {
        struct load_weight load; /* 用于负载均衡 */
        struct rb_node run_node;
        unsigned int on_rq;

        u64 exec_start;
        u64 sum_exec_runtime;
        u64 vruntime;
        u64 prev_sum_exec_runtime;
...
}
```

如果编译内核时启用了调度器统计，那么该结构会包含很多用于统计的成员。如果启用了组调度，那么还会增加一些成员。但我们目前感兴趣的内容主要是上面列出的几项。各个成员的含义如下。

- load指定了权重，决定了各个实体占队列总负荷的比例。计算负荷权重是调度器的一项重任，因为CFS所需的虚拟时钟的速度最终依赖于负荷，因此我会在2.5.3节详细讨论该方法。
- run_node是标准的树结点，使得实体可以在红黑树上排序。
- on_rq表示该实体当前是否在就绪队列上接受调度。
- 在进程运行时，我们需要记录消耗的CPU时间，以用于完全公平调度器。sum_exec_runtime即用于该目的。跟踪运行时间是由update_curr不断累积完成的。调度器中许多地方都会调用该函数，例如，新进程加入就绪队列时，或者周期性调度器中。每次调用时，会计算当前时间和exec_start之间的差值，exec_start则更新到当前时间。差值则被加到sum_exec_runtime。

  在进程执行期间虚拟时钟上流逝的时间数量由vruntime统计。
- 在进程被撤销CPU时，其当前sum_exec_runtime值保存到prev_exec_runtime。此后，在进程抢占时又需要该数据。但请注意，在prev_exec_runtime中保存sum_exec_runtime的值，并不意味着重置sum_exec_runtime！原值保存下来，而sum_exec_runtime则持续单调增长。

由于每个task_struct都嵌入了sched_entity的一个实例，所以进程是可调度实体。但请注意，其逆命题一般是不正确的，因为可调度的实体不见得一定是进程。但在下文中我们只关注进程调度，因此我们暂时将调度实体和进程视为等同。不过要记住，这在一般意义上是不正确的！

2.5.3 处理优先级

从用户的角度来看，优先级也太简单了。因为，他们看来优先级似乎只是某个范围内的数字。令人遗憾的是，内核内部对优先级的处理并没有我们想象中那么简单。事实上，处理优先级相当复杂。

1. 优先级的内核表示

在用户空间可以通过nice命令设置进程的静态优先级，这在内部会调用nice系统调用。[①] 进程的nice值在-20和+19之间（包含）。值越低，表明优先级越高。为什么选择这个诡异的范围，真相已经淹没在历史中。

① setpriority是另一个用于设置进程优先级的系统调用。它不仅能够修改单个线程的优先级，还能修改线程组中所有线程的优先级，或者通过指定UID来修改特定用户的所有进程的优先级。

内核使用一个简单些的数值范围，从0到139（包含），用来表示内部优先级。同样是值越低，优先级越高。从0到99的范围专供实时进程使用。nice值[−20, +19]映射到范围100到139，如图2-14所示。实时进程的优先级总是比普通进程更高。

图2-14 内核优先级标度

下列宏用于在各种不同表示形式之间转换（MAX_RT_PRIO指定实时进程的最大优先级，而MAX_PRIO则是普通进程的最大优先级数值）：

<sched.h>

```
#define MAX_USER_RT_PRIO        100
#define MAX_RT_PRIO             MAX_USER_RT_PRIO
#define MAX_PRIO                (MAX_RT_PRIO + 40)
#define DEFAULT_PRIO            (MAX_RT_PRIO + 20)
```

kernel/sched.c

```
#define NICE_TO_PRIO(nice)      (MAX_RT_PRIO + (nice) + 20)
#define PRIO_TO_NICE(prio)      ((prio) -MAX_RT_PRIO -20)
#define TASK_NICE(p)            PRIO_TO_NICE((p)->static_prio)
```

2. 计算优先级

回想一下，可知只考虑进程的静态优先级是不够的，还必须考虑下面3个优先级。即动态优先级（task_struct->prio）、普通优先级（task_struct->normal_prio）和静态优先级（task_struct->static_prio）。这些优先级按有趣的方式彼此关联，下文中我会具体讨论。

static_prio是计算的起点。假定它已经设置好，而内核现在想要计算其他优先级。一行代码即可：

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

辅助函数effective_prio执行了下列操作：

kernel/sched.c

```
static int effective_prio(struct task_struct *p)
{
        p->normal_prio = normal_prio(p);
        /*
         * 如果是实时进程或已经提高到实时优先级，则保持优先级不变。否则，返回普通优先级：
         */
        if (!rt_prio(p->prio))
                return p->normal_prio;
        return p->prio;
}
```

这里首先计算了普通优先级，并保存在normal_priority。这个副效应使得能够用一个函数调用设置两个优先级（prio和normal_prio）。另一个辅助函数rt_prio，会检测普通优先级是否在实时范围中，即是否小于RT_RT_PRIO。请注意，该检测与调度类无关，它只涉及优先级的数值。

现在假定我们在处理普通进程，不涉及实时调度。在这种情况下，normal_prio只是返回静态优先级。结果很简单：所有3个优级都是同一个值，即静态优先级！

实时进程的情况有所不同。注意普通优先级的计算方法：

kernel/sched.c

```
static inline int normal_prio(struct task_struct *p)
```

```
{
        int prio;

        if (task_has_rt_policy(p))
                prio = MAX_RT_PRIO-1 -p->rt_priority;
        else
                prio = __normal_prio(p);
        return prio;
}
```

普通优先级需要根据普通进程和实时进程进行不同的计算。__normal_prio的计算只适用于普通进程。而实时进程的普通优先级计算，则需要根据其rt_priority设置。由于更高的rt_priority值表示更高的实时优先级，内核内部优先级的表示刚好相反，越低的值表示的优先级越高。因此，实时进程在内核内部的优先级数值，正确的算法是MAX_RT_PRIO - 1 - p->rt_priority。这一次请注意，与effective_prio相比，实时进程的检测不再基于优先级数值，而是通过task_struct中设置的调度策略来检测。

__normal_priority做什么呢？该函数实际上很简单，它只是返回静态优先级：

kernel/sched.c

```
static inline int __normal_prio(struct task_struct *p)
{
        return p->static_prio;
}
```

读者现在可以很奇怪，为什么对此增加一个额外的函数。这是有历史原因的：在原来的*O*(1)调度器中，普通优先级的计算涉及相当多技巧性的工作。必须检测交互式进程并提高其优先级，而必须“惩罚”非交互进程，以便使系统获得良好的交互体验。这需要大量的启发式计算，它们可能完成得很好，也可能不工作。感谢新的调度器，已经不再需要此类魔法式计算。

但还有一个问题：为什么内核在effective_prio中检测实时进程是基于优先级数值，而非task_has_rt_policy？对于临时提高至实时优先级的非实时进程来说，这是必要的，这种情况可能发生在使用实时互斥量（RT-Mutex）时。[①]

最后，表2-3综述了针对不同类型进程上述计算的结果。

表2-3 对各种类型的进程计算优先级

进程类型 / 优先级	static_prio	normal_prio	prio
非实时进程	static_prio	static_prio	static_prio
优先级提高的非实时进程	static_prio	static_prio	prio不变
实时进程	static_prio	MAX_RT_PRIO-1-rt_priority	prio不变

在新建进程用wake_up_new_task唤醒时，或使用nice系统调用改变静态优先级时，则用上文给出的方法设置p->prio。

请注意，在进程分支出子进程时，子进程的静态优先级继承自父进程。子进程的动态优先级，即task_struct->prio，则设置为父进程的普通优先级。这确保了实时互斥量引起的优先级提高不会传递到子进程。

① 实时互斥量能够保护内核的一些部分，防止多处理器并发访问。但有一种现象会发生，称作*优先级反转*（priority inversion）。其中一个低优先级进程在执行，而较高优先级的进程则在等待CPU。这可以通过临时提高进程的优先级解决。有关该问题的更多细节，请参考5.2.8节的讨论。

3. 计算负荷权重

进程的重要性不仅是由优先级指定的，而且还需要考虑保存在task_struct->se.load的负荷权重。set_load_weight负责根据进程类型及其静态优先级计算负荷权重。

负荷权重包含在数据结构load_weight中：

<sched.h>

```
struct load_weight {
        unsigned long weight, inv_weight;
};
```

内核不仅维护了负荷权重自身，而且还有另一个数值，用于计算被负荷权重除的结果。①

一般概念是这样，进程每降低一个nice值，则多获得10%的CPU时间，每升高一个nice值，则放弃10%的CPU时间。为执行该策略，内核将优先级转换为权重值。我们首先看一下转换表：

kernel/sched.c

```
static const int prio_to_weight[40] = {
  /* -20 */     88761,   71755,   56483,   46273,   36291,
  /* -15 */     29154,   23254,   18705,   14949,   11916,
  /* -10 */     9548,    7620,    6100,    4904,    3906,
  /*  -5 */     3121,    2501,    1991,    1586,    1277,
  /*   0 */     1024,    820,     655,     526,     423,

  /*   5 */     335,     272,     215,     172,     137,
  /*  10 */     110,     87,      70,      56,      45,
  /*  15 */     36,      29,      23,      18,      15,
};
```

对内核使用的范围[0, 39]中的每个nice级别，该数组中都有一个对应项。各数组之间的乘数因子是1.25。要知道为何使用该因子，可考虑下列例子。两个进程A和B在nice级别0运行，因此两个进程的CPU份额相同，即都是50%。nice级别为0的进程，其权重查表可知为1024。每个进程的份额是1024/（1024+1024）=0.5，即50%。

如果进程B的优先级加1，那么其CPU份额应该减少10%。换句话说，这意味着进程A得到总的CPU时间的55%，而进程B得到45%。优先级增加1导致权重减少，即1024/1.25≈820。因此进程A现在将得到的CPU份额是1024/(1024+820)≈0.55，而进程B的份额则是820/(1024+820)≈0.45，这样就产生了10%的差值。

执行转换的代码也需要考虑实时进程。实时进程的权重是普通进程的两倍。另一方面，SCHED_IDLE进程的权重总是非常小：

kernel/sched.c

```
#define WEIGHT_IDLEPRIO          2
#define WMULT_IDLEPRIO           (1 << 31)
static void set_load_weight(struct task_struct *p)
{
        if (task_has_rt_policy(p)) {
                p->se.load.weight = prio_to_weight[0] * 2;
                p->se.load.inv_weight = prio_to_wmult[0] >> 1;
                return;
        }

        /*
```

① 由于使用了普通的long类型，因此内核无法直接存储1/weight，而必须借助于利用乘法和位移来执行除法的技术。但这里并不关注相关的细节。

```
        * SCHED_IDLE进程得到的权重最小:
        */
        if (p->policy == SCHED_IDLE) {
                p->se.load.weight = WEIGHT_IDLEPRIO;
                p->se.load.inv_weight = WMULT_IDLEPRIO;
                return;
        }

        p->se.load.weight = prio_to_weight[p->static_prio -MAX_RT_PRIO];
        p->se.load.inv_weight = prio_to_wmult[p->static_prio -MAX_RT_PRIO];
}
```

内核不仅计算出权重本身，还存储了用于除法的值。请注意，每个优先级变化关联10%的CPU时间的特征，导致了权重（和相关的CPU时间）的指数特征，见图2-15。图中上方的插图给出了对应于普通优先级的某个受限区域内的曲线图。下方的插图在Y轴上则采用了对数标度。要注意，该函数在普通到实时进程间的临界点上是不连续的。

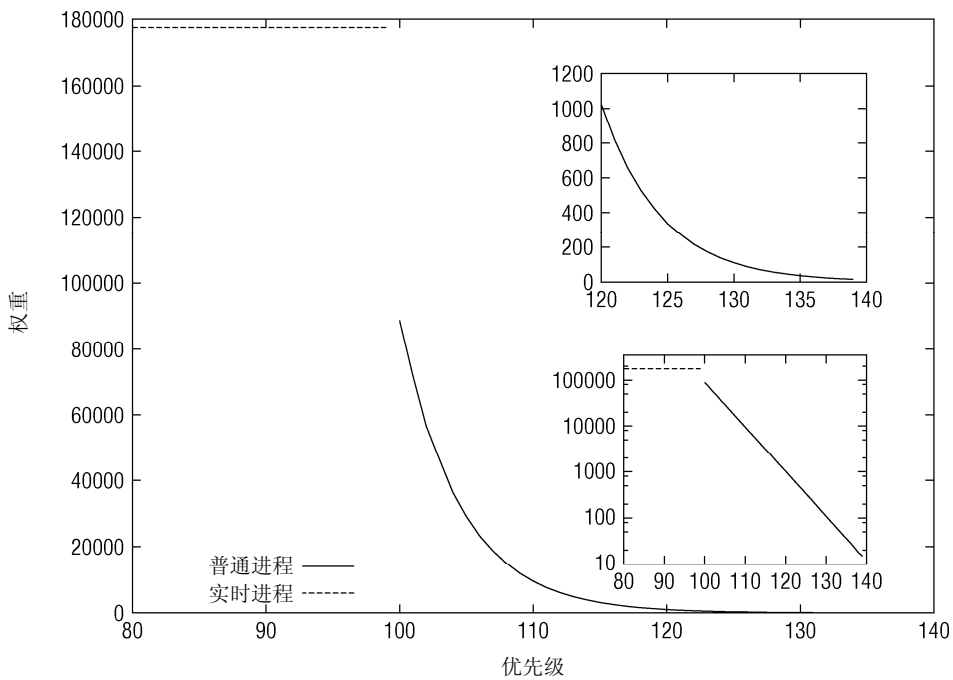

图2-15 静态优先级和负荷之间关系，分普通和实时进程两种情况

回想一下可知，不仅进程，而且就绪队列也关联到一个负荷权重。每次进程被加到就绪队列时，内核会调用inc_nr_running。这不仅确保就绪队列能够跟踪记录有多少进程在运行，而且还将进程的权重添加到就绪队列的权重中：

kernel/sched.c

```
static inline void update_load_add(struct load_weight *lw, unsigned long inc)
{
        lw->weight += inc;
}
```

```
static inline void inc_load(struct rq *rq, const struct task_struct *p)
{
        update_load_add(&rq->load, p->se.load.weight);
}

static void inc_nr_running(struct task_struct *p, struct rq *rq)
{
        rq->nr_running++;
        inc_load(rq, p);
}
```

在进程从就绪队列移除时，会调用对应的函数（dec_nr_running、dec_load、update_load_sub）。

2.5.4 核心调度器

如前所述，调度器的实现基于两个函数：周期性调度器函数和主调度器函数。这些函数根据现有进程的优先级分配CPU时间。这也是为什么整个方法称之为优先调度的原因，不过其实也是一个非常一般的术语。我在本节将论述优先调度的实现方式。

1. 周期性调度器

周期性调度器在scheduler_tick中实现。如果系统正在活动中，内核会按照频率HZ自动调用该函数。如果没有进程在等待调度，那么在计算机电力供应不足的情况下，也可以关闭该调度器以减少电能消耗。例如，笔记本电脑或小型嵌入式系统。周期性操作的底层机制将在第15章讨论。该函数有下面两个主要任务。

(1) 管理内核中与整个系统和各个进程的调度相关的统计量。其间执行的主要操作是对各种计数器加1，我们对此没什么兴趣。

(2) 激活负责当前进程的调度类的周期性调度方法。

kernel/sched.c

```
voidscheduler_tick(void)
{
        int cpu = smp_processor_id();
        struct rq *rq = cpu_rq(cpu);
        struct task_struct *curr = rq->curr;

...
        __update_rq_clock(rq)
        update_cpu_load(rq);
```

该函数的第一部分处理就绪队列时钟的更新。该职责委托给__update_rq_clock完成，本质上就是增加struct rq当前实例的时钟时间戳。该函数必须处理硬件时钟的一些奇异之处，这与我们的目标不相干。update_cpu_load负责更新就绪队列的cpu_load[]数组。本质上相当于将数组中先前存储的负荷值向后移动一个位置，将当前就绪队列的负荷记入数组的第一个位置。另外，该函数还引入了一些取平均值的技巧，以确保负荷数组的内容不会呈现出太多的不连续跳变。

由于调度器的模块化结构，主体工程实际上比较简单，因为主要的工作可以完全委托给特定调度器类的方法：

kernel/sched.c

```
        if (curr != rq->idle)
                curr->sched_class->task_tick(rq, curr);
}
```

task_tick的实现方式取决于底层的调度器类。例如，完全公平调度器会在该方法中检测是否进

程已经运行太长时间，以避免过长的延迟，我会在下文详细讨论。如果读者熟悉旧的基于时间片的调度方法，那么应该会知道，这里的做法实际上不等价于到期的时间片，因为完全公平调度器中不再存在所谓时间片的概念。

如果当前进程应该被重新调度，那么调度器类方法会在task_struct中设置TIF_NEED_RESCHED标志，以表示该请求，而内核会在接下来的适当时机完成该请求。

2. 主调度器

在内核中的许多地方，如果要将CPU分配给与当前活动进程不同的另一个进程，都会直接调用主调度器函数（schedule）。在从系统调用返回之后，内核也会检查当前进程是否设置了重调度标志TIF_NEED_RESCHED，例如，前述的scheduler_tick就会设置该标志。如果是这样，则内核会调用schedule。该函数假定当前活动进程一定会被另一个进程取代。

在详细论述schedule之前，需要说明一下__sched前缀。该前缀用于可能调用schedule的函数，包括schedule自身。其声明如下所示：

```
void __sched some_function(...) {
...
        schedule();
...
}
```

该前缀目的在于，将相关函数的代码编译之后，放到目标文件的一个特定的段中，即.sched.text中（有关ELF段的更多信息，请参见附录C）。该信息使得内核在显示栈转储或类似信息时，忽略所有与调度有关的调用。由于调度器函数调用不是普通代码流程的一部分，因此在这种情况下是没有意义的。

我们现在回到主调度器schedule的实现。该函数首先确定当前就绪队列，并在prev中保存一个指向（仍然）活动进程的task_struct的指针。

kernel/sched.c

```
asmlinkage void __sched schedule(void)
{
        struct task_struct *prev, *next;
        struct rq *rq;
        int cpu;

need_resched:
        cpu = smp_processor_id();
        rq = cpu_rq(cpu);
        prev = rq->curr;
...
```

类似于周期性调度器，内核也利用该时机来更新就绪队列的时钟，并清除当前运行进程task_struct中的重调度标志TIF_NEED_RESCHED。

kernel/sched.c

```
        __update_rq_clock(rq);
        clear_tsk_need_resched(prev);
...
```

同样因为调度器的模块化结构，大多数工作可以委托给调度类。如果当前进程原来处于可中断睡眠状态但现在接收到信号，那么它必须再次提升为运行进程。否则，用相应调度器类的方法使进程停止活动（deactivate_task实质上最终调用了sched_class->dequeue_task）：

kernel/sched.c

```
        if (unlikely((prev->state & TASK_INTERRUPTIBLE) &&
```

```
        unlikely(signal_pending(prev)))) {
                prev->state = TASK_RUNNING;
        } else {
                deactivate_task(rq, prev, 1);
        }
...
```

put_prev_task首先通知调度器类当前运行的进程将要被另一个进程代替。要注意，这不等价于把进程从就绪队列移除，而是提供了一个时机，供执行一些簿记工作并更新统计量。调度类还必须选择下一个应该执行的进程，该工作由pick_next_task负责：

```
        prev->sched_class->put_prev_task(rq, prev);
        next = pick_next_task(rq, prev);
...
```

不见得必然选择一个新进程。也可能其他进程都在睡眠，当前只有一个进程能够运行，这样它自然就被留在CPU上。但如果已经选择了一个新进程，那么必须准备并执行硬件级的进程切换。

kernel/sched.c

```
        if (likely(prev != next)) {
                rq->curr = next;
                context_switch(rq, prev, next);
        }
...
```

context_switch一个接口，供访问特定于体系结构的方法，后者负责执行底层上下文切换。

下列代码检测当前进程的重调度位是否设置，并跳转到如上所述的标号，重新开始搜索一个新进程：

kernel/sched.c

```
        if (unlikely(test_thread_flag(TIF_NEED_RESCHED)))
                goto need_resched;
}
```

请注意，上述代码片段可能在两个不同的上下文中执行。在没有执行上下文切换时，它在schedule函数的末尾直接执行。但如果已经执行了上下文切换，当前进程会正好在这以前停止运行，新进程已经接管了CPU。但稍后在前一进程被再次选择运行时，它会刚好在这一点上恢复执行。在这种情况下，由于prev不会指向正确的进程，所以需要通过current和test_thread_flag找到当前线程。

3. 与fork的交互

每当使用fork系统调用或其变体之一建立新进程时，调度器有机会用sched_fork函数挂钩到该进程。在单处理器系统上，该函数实质上执行3个操作：初始化新进程与调度相关的字段、建立数据结构（相当简单直接）、确定进程的动态优先级。

kernel/sched.c

```
/*
 * fork()/clone()时的设置:
 */
void sched_fork(struct task_struct *p, int clone_flags)
{

        /* 初始化数据结构 */
...
        /*
         * 确认没有将提高的优先级泄漏到子进程
         */
```

```
        p->prio = current->normal_prio;
        if (!rt_prio(p->prio))
                p->sched_class = &fair_sched_class;
...
}
```

通过使用父进程的普通优先级作为子进程的动态优先级，内核确保父进程优先级的临时提高不会被子进程继承。回想一下，可知在使用实时互斥量时进程的动态优先级可以临时修改。该效应不能转移到子进程。如果优先级不在实时范围中，则进程总是从完全公平调度类开始执行。

在使用wake_up_new_task唤醒新进程时，则是调度器与进程创建逻辑交互的第二个时机：内核会调用调度类的task_new函数。这提供了一个时机，将新进程加入到相应类的就绪队列中。

4. 上下文切换

内核选择新进程之后，必须处理与多任务相关的技术细节。这些细节总称为上下文切换（context switching）。辅助函数context_switch是个分配器，它会调用所需的特定于体系结构的方法。

kernel/sched.c

```
static inline void
context_switch(struct rq *rq, struct task_struct *prev,
               struct task_struct *next)
{
        struct mm_struct *mm, *oldmm;

        prepare_task_switch(rq, prev, next);
        mm = next->mm;
        oldmm = prev->active_mm;
..
```

紧接着进程切换之前，prepare_task_switch会调用每个体系结构都必须定义的prepare_arch_switch挂钩。这使得内核执行特定于体系结构的代码，为切换做事先准备。大多数支持的体系结构（Sparc64和Sparc除外）都不需要该选项，因此并未使用。

上下文切换本身通过调用两个特定于处理器的函数完成。

(1) switch_mm更换通过task_struct->mm描述的内存管理上下文。该工作的细节取决于处理器，主要包括加载页表、刷出地址转换后备缓冲器（部分或全部）、向内存管理单元（MMU）提供新的信息。由于这些操作深入到CPU的细节中，我不打算在此讨论其实现。

(2) switch_to切换处理器寄存器内容和内核栈（虚拟地址空间的用户部分在第一步已经变更，其中也包括了用户状态下的栈，因此用户栈就不需要显式变更了）。此项工作在不同的体系结构下可能差别很大，代码通常都使用汇编语言编写。

由于用户空间进程的寄存器内容在进入核心态时保存在内核栈上（更多细节请参见第14章），在上下文切换期间无需显式操作。而因为每个进程首先都是从核心态开始执行（在调度期间控制权传递到新进程），在返回用户空间时，会使用内核栈上保存的值自动恢复寄存器数据。

但要记住，内核线程没有自身的用户空间内存上下文，可能在某个随机进程地址空间的上部执行。其task_struct->mm为NULL。从当前进程“借来”的地址空间记录在active_mm中：

kernel/sched.c

```
if (unlikely(!mm)) {
        next->active_mm = oldmm;
        atomic_inc(&oldmm->mm_count);
        enter_lazy_tlb(oldmm, next);
} else
        switch_mm(oldmm, mm, next);
...
```

enter_lazy_tlb通知底层体系结构不需要切换虚拟地址空间的用户空间部分。这种加速上下文切换的技术称之为惰性TLB。

如果前一进程是内核线程（即prev->mm为NULL），则其active_mm指针必须重置为NULL，以断开与借用的地址空间的联系：

kernel/sched.c

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

最后用switch_to完成进程切换，该函数切换寄存器状态和栈，新进程在该调用之后开始执行：

kernel/sched.c

```
        /* 这里我们只是切换寄存器状态和栈。 */
        switch_to(prev, next, prev);

        barrier();
        /*
         * this_rq必须重新计算，因为在调用schedule()之后prev可能已经移动到其他CPU，
         * 因此其栈帧上的rq可能是无效的。
         */
        finish_task_switch(this_rq(), prev);
}
```

switch_to之后的代码只有在当前进程下一次被选择运行时才会执行。finish_task_switch完成一些清理工作，使得能够正确地释放锁，但我们不会详细讨论这些。它也向各个体系结构提供了另一个挂钩上下文切换过程的可能性，但只在少量计算机上需要。barrier语句是一个编译器指令，确保switch_to和finish_task_switch语句的执行顺序不会因为任何可能的优化而改变（更多细节请参见第5章）。

- switch_to的复杂之处

finish_task_switch的有趣之处在于，调度过程可能选择了一个新进程，而清理则是针对此前的活动进程。请注意，这不是发起上下文切换的那个进程，而是系统中随机的某个其他进程！内核必须想办法使得该进程能够与context_switch例程通信，这可以通过switch_to宏实现。每个体系结构都必须实现它，而且有一个异乎寻常的调用约定，即通过3个参数传递两个变量！这是因为上下文切换不仅涉及两个进程，而是3个进程。该情形如图2-16所示。

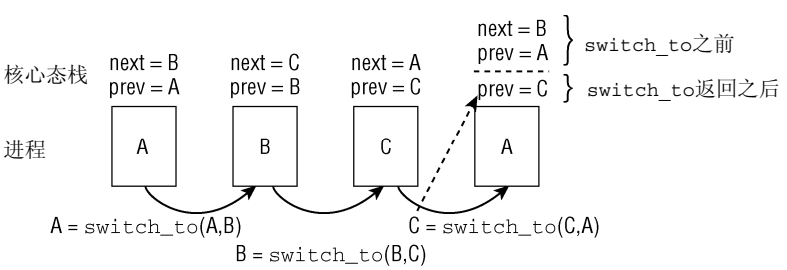

图2-16 上下文切换期间prev和next变量的行为特性

假定3个进程A、B和C在系统上运行。在某个时间点，内核决定从进程A切换到进程B，然后从进程B到进程C，再接下来从进程C切换回进程A。在每个switch_to调用之前，next和prev指针位于各

进程的栈上，prev指向当前运行的进程，而next指向将要运行的下一个进程。为执行从prev到next的切换，switch_to的前两个参数足够了。对进程A来说，prev指向进程A而next指向进程B。

在进程A被选中再次执行时，会出现一个问题。控制权返回至switch_to之后的点，如果栈准确地恢复到切换之前的状态，那么prev和next仍然指向切换之前的值，即next = B而prev = A。在这种情况下，内核无法知道实际上在进程A之前运行的是进程C。

因此，在新进程被选中时，底层的进程切换例程必须将此前执行的进程提供给context_switch。由于控制流会回到该函数的中间，这无法用普通的函数返回值来做到，因此使用了一个3个参数的宏。但逻辑上的效果是相同的，仿佛switch_to是带有两个参数的函数，而且返回了一个指向此前运行进程的指针。switch_to宏实际上执行的代码如下：

```
prev = switch_to(prev,next)
```

其中返回的prev值并不是用作参数的prev值，而是上一个执行的进程。在上述例子中，进程A提供给switch_to的参数是A和B，但恢复执行后得到的返回值是prev = C。内核实现该行为特性的方式依赖于底层的体系结构，但内核显然可以通过考虑两个进程的核心态栈来重建所要的信息。对可以访问所有内存的内核而言，这两个栈显然是同时可用的。

- 惰性FPU模式

由于上下文切换的速度对系统性能的影响举足轻重，所以内核使用了一种技巧来减少所需的CPU时间。浮点寄存器（及其他内核未使用的扩充寄存器，例如IA-32平台上的SSE2寄存器）除非有应用程序实际使用，否则不会保存。此外，除非有应用程序需要，否则这些寄存器也不会恢复。这称之为惰性FPU技术。由于使用了汇编语言代码，因此其实现依平台而有所不同，但基本原理总是同样的。也应注意到，如果不考虑平台，浮点寄存器的内容不是保存在进程栈上，而是保存在线程数据结构中。我将通过一个例子来说明该技术。

为简明起见，我们假定这一次系统中只有进程A和进程B。进程A在运行并使用浮点操作。在调度器切换到进程B时，进程A的浮点寄存器的内容保存到进程的线程数据结构中。但这些寄存器中的值不会立即被来自进程B的值替换。

如果进程B在其时间片内并不执行任何浮点操作，那么在进程A下一次激活时，会看到CPU浮点寄存器内容与此前相同。内核因此节省了显式恢复寄存器值的工作量，这节省了时间。

但如果进程B确实执行了浮点操作，该事实会报告给内核，它会用来自线程数据结构的适当值填充寄存器。因此，只有在需要的情况下，内核才会保存和恢复浮点寄存器内容，不会因为多余的操作浪费时间。

2.6 完全公平调度类

核心调度器必须知道的有关完全公平调度器的所有信息，都包含在fair_sched_class中：

kernel/sched_fair.c

```
static const struct sched_class fair_sched_class = {
        .next = &idle_sched_class,
        .enqueue_task = enqueue_task_fair,
        .dequeue_task = dequeue_task_fair,
        .yield_task = yield_task_fair,

        .check_preempt_curr = check_preempt_wakeup,

        .pick_next_task = pick_next_task_fair,
        .put_prev_task = put_prev_task_fair,
```

```
...
        .set_curr_task = set_curr_task_fair,
        .task_tick = task_tick_fair,
        .task_new = task_new_fair,
};
```

在先前的讨论中，我们已经看到主调度器调用这些函数，接下来我们将考察这些函数在CFS中的实现方式。

2.6.1 数据结构

首先，我们需要介绍一下CFS的就绪队列。回想一下，可知主调度器的每个就绪队列中都嵌入了一个该结构的实例：

kernel/sched.c

```
struct cfs_rq {
        struct load_weight load;
        unsigned long nr_running;

        u64 min_vruntime;

        struct rb_root tasks_timeline;
        struct rb_node *rb_leftmost;

        struct sched_entity *curr;
}
```

各个成员的语义如下。

- nr_running计算了队列上可运行进程的数目，load维护了所有这些进程的累积负荷值。回想一下在2.5.3节已经遇到的负荷计算相关内容。
- min_vruntime跟踪记录队列上所有进程的最小虚拟运行时间。这个值是实现与就绪队列相关的虚拟时钟的基础。其名字很容易会产生一些误解，因为min_vruntime实际上可能比最左边的树结点的vruntime大些。因为它是单调递增的，在我详细论述该值的设置时会继续讨论该问题。
- tasks_timeline是一个基本成员，用于在按时间排序的红黑树中管理所有进程。rb_leftmost总是设置为指向树最左边的结点，即最需要被调度的进程。该成员理论上可以通过遍历红黑树获得，但由于我们通常只对最左边的结点感兴趣，因为这可以减少搜索树花费的平均时间。
- curr指向当前执行进程的可调度实体。

2.6.2 CFS 操作

我们现在把注意力转向如何实现CF调度器提供的调度方法。

1. 虚拟时钟

我在2.5.1节提到，完全公平调度算法依赖于虚拟时钟，用以度量等待进程在完全公平系统中所能得到的CPU时间。但数据结构中任何地方都没找到虚拟时钟！这是由于所有的必要信息都可以根据现存的实际时钟和与每个进程相关的负荷权重推算出来。所有与虚拟时钟有关的计算都在update_curr中执行，该函数在系统中各个不同地方调用，包括周期性调度器之内。图2-17的代码流程图提供了该函数所完成工作的概述。

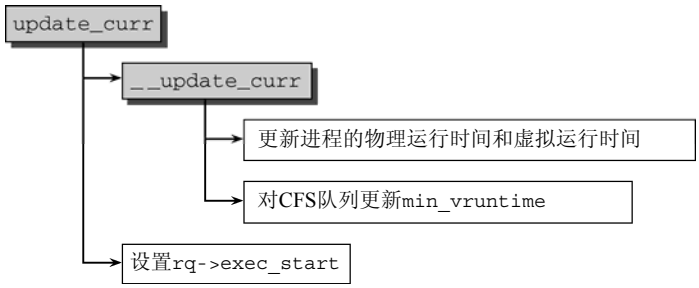

图2-17 update_curr的代码流程图

首先，该函数确定就绪队列的当前执行进程，并获取主调度器就绪队列的实际时钟值，该值在每个调度周期都会更新（rq_of是一个辅助函数，用于确定与CFS就绪队列相关的struct rq实例）：

```
static void update_curr(struct cfs_rq *cfs_rq)
{
        struct sched_entity *curr = cfs_rq->curr;
        u64 now = rq_of(cfs_rq)->clock;
        unsigned long delta_exec;

        if (unlikely(!curr))
                return;
...
```

如果就绪队列上当前没有进程正在执行，则显然无事可做。否则，内核会计算当前和上一次更新负荷统计量时两次的时间差，并将其余的工作委托给__update_curr。

kernel/sched_fair.c

```
        delta_exec = (unsigned long)(now -curr->exec_start);

        __update_curr(cfs_rq, curr, delta_exec);
        curr->exec_start = now;
}
```

根据这些信息，__update_curr需要更新当前进程在CPU上执行花费的物理时间和虚拟时间。物理时间的更新比较简单，只要将时间差加到先前统计的时间即可：

kernel/sched_fair.c

```
static inline void
__update_curr(struct cfs_rq *cfs_rq, struct sched_entity *curr,
              unsigned long delta_exec)
{
        unsigned long delta_exec_weighted;
        u64 vruntime;

        curr->sum_exec_runtime += delta_exec;
...
```

有趣的事情是如何使用给出的信息来模拟不存在的虚拟时钟。这一次内核的实现仍然是非常巧妙的，针对最普遍的情形节省了一些时间。对于运行在nice级别0的进程来说，根据定义虚拟时间和物理时间是相等的。在使用不同的优先级时，必须根据进程的负荷权重重新衡定时间（回想2.5.3节讨论的进程优先级与负荷权重之间的关联）：

kernel/sched_fair.c

```
        delta_exec_weighted = delta_exec;
```

```
        if (unlikely(curr->load.weight != NICE_0_LOAD)) {
                delta_exec_weighted = calc_delta_fair(delta_exec_weighted,
                                                      &curr->load);
        }
        curr->vruntime += delta_exec_weighted;
...
```

忽略舍入和溢出检查，calc_delta_fair所作的就是根据下列公式计算：

$$\text{delta\_exec\_weighted} = \text{delta\_exec} \times \frac{\text{NICE\_0\_LOAD}}{\text{Curr->load.weight}}$$

前文提到的逆向权重值，在该计算中可以派上用场了。回想一下，可知越重要的进程会有越高的优先级（即，越低的nice值），会得到更大的权重，因此累加的虚拟运行时间会小一些。图2-18给出了不同优先级的实际时间和虚拟时间之间的关系。根据公式可知，nice 0进程优先级为120，则虚拟时间和物理时间是相等的，即current->load.weight等于NICE_0_LOAD的情况。请注意图2-18的插图，其中使用了双对数坐标来对各种优先级绘图。

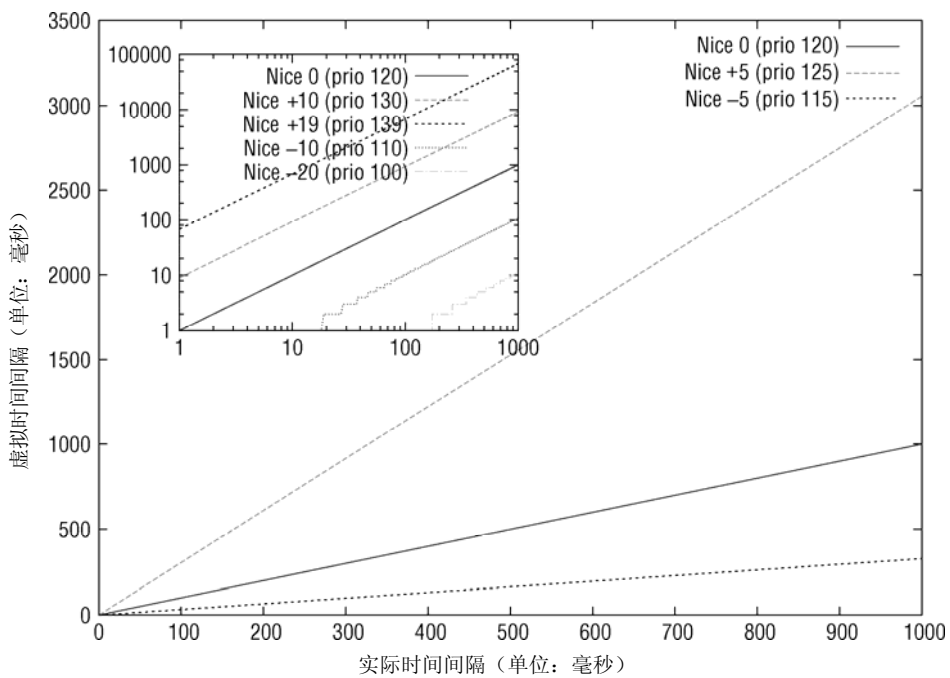

图2-18 不同nice级别/优先级的进程，实际时间和虚拟时间的关系

最后，内核需要设置min_vruntime。必须小心保证该值是单调递增的。

kernel/sched_fair.c

```
        /*
         * 跟踪树中最左边的结点的vruntime，维护cfs_rq->min_vruntime的单调递增性
         */
        if (first_fair(cfs_rq)) {
                vruntime = min_vruntime(curr->vruntime,
                        __pick_next_entity(cfs_rq)->vruntime);
```

```
        } else
                vruntime = curr->vruntime;

        cfs_rq->min_vruntime =
                max_vruntime(cfs_rq->min_vruntime, vruntime);
}
```

first_fair是一个辅助函数，检测树是否有最左边的结点，即是否有进程在树上等待调度。倘若如此，则内核获取其vruntime，即树中所有结点最小的vruntime值。如果因为树是空的而没有最左边的结点，则使用当前进程的虚拟运行时间。为保证每个队列的min_vruntime是单调递增的，内核将其设置为二者中的较大者。这意味着，每个队列的min_vruntime只有被树上某个结点的vruntime超出时才更新。利用该策略，内核确保min_vrtime只能增加，不能减少。

完全公平调度器的真正关键点是，红黑树的排序过程是根据下列键进行的：

kernel/sched_fair.c

```
static inline s64 entity_key(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
        return se->vruntime -cfs_rq->min_vruntime;
}
```

键值较小的结点，排序位置就更靠左，因此会被更快地调度。用这种方法，内核实现了下面两种对立的机制。

(1) 在进程运行时，其vruntime稳定地增加，它在红黑树中总是向右移动的。

因为越重要的进程vruntime增加越慢，因此它们向右移动的速度也越慢，这样其被调度的机会要大于次要进程，这刚好是我们需要的。

(2) 如果进程进入睡眠，则其vruntime保持不变。因为每个队列min_vruntime同时会增加（回想一下，它是单调的!），那么睡眠进程醒来后，在红黑树中的位置会更靠左，因为其键值变得更小了。[①]

实际上上述两种效应是同时发生作用的，但这并不影响解释。图2-19针对红黑树上不同的移动机制，作出了图解。

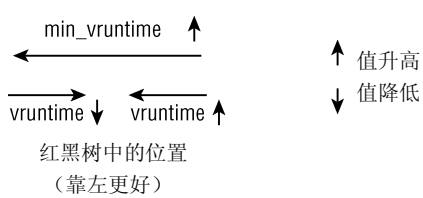

图2-19 每个调度实体和每个队列的虚拟时间对进程在红黑树中位置的影响

2. 延迟跟踪

内核有一个固有的概念，称之为良好的调度延迟，即保证每个可运行的进程都应该至少运行一次的某个时间间隔。[②] 它在sysctl_sched_latency给出，可通过/proc/sys/kernel/sched_latency_ns控制，默认值为20 000 000纳秒或20毫秒。第二个控制参数sched_nr_latency，控制在一个延迟周期中处理的最大活动进程数目。如果活动进程的数目超出该上限，则延迟周期也成比例地线性扩展。sched_nr_latency可以通过sysctl_sched_min_granularity间接地控制，后者可通过/proc/sys/

① 对短时间睡眠的进程来说，稍有不同，在我讨论具体机制时会考虑这种情况。

② 切记：这与时间片无关，旧的调度器才使用时间片！

kernel/sched_min_granularity_ns设置。默认值是4 000 000纳秒，即4毫秒，每次sysctl_sched_latency/sysctl_sched_min_granularity之一改变时，都会重新计算sched_nr_latency。

__sched_period确定延迟周期的长度，通常就是sysctl_sched_latency，但如果有更多进程在运行，其值有可能按比例线性扩展。在这种情况下，周期长度是：

$$\text{sysctl\_sched\_latency} \times \frac{\text{nr\_running}}{\text{sched\_nr\_latency}}$$

通过考虑各个进程的相对权重，将一个延迟周期的时间在活动进程之间进行分配。对于由某个可调度实体表示的给定进程，分配到的时间如下计算：

kernel/sched_fair.c

```
static u64 sched_slice(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
        u64 slice = __sched_period(cfs_rq->nr_running);

        slice *= se->load.weight;
        do_div(slice, cfs_rq->load.weight);

        return slice;
}
```

回想一下，就绪队列的负荷权重是队列上所有活动进程负荷权重的累加和。结果时间段是按实际时间给出的，但内核有时候也需要知道等价的虚拟时间。

kernel/sched_fair.c

```
static u64 __sched_vslice(unsigned long rq_weight, unsigned long nr_running)
{
        u64 vslice = __sched_period(nr_running);

        vslice *= NICE_0_LOAD;
        do_div(vslice, rq_weight);

        return vslice;
}

static u64 sched_vslice(struct cfs_rq *cfs_rq)
{
  return __sched_vslice(cfs_rq->load.weight, cfs_rq->nr_running);
}
```

回想一下，对权重weight的进程来说，实际时间段time对应的虚拟时间长度为：

$$\text{time} \times \frac{\text{NICE\_0\_LOAD}}{\text{weight}}$$

该公式也用于转换分配到的延迟时间间隔。

现在万事俱备，可以开始讨论CFS与全局调度器交互所必须实现的各个方法了。

2.6.3 队列操作

有两个函数可用来增删就绪队列的成员：enqueue_task_fair和dequeue_task_fair。我们首先关注如何向就绪队列放置新进程。

除了指向所述的就绪队列和task_struct的指针外，该函数还有另一个参数wakeup。这使得可以指定入队的进程是否最近才被唤醒并转换为运行状态（在这种情况下wakeup为1），还是此前就是可运行的（那么wakeup是0）。enqueue_task_fair的代码流程图如图2-20所示。

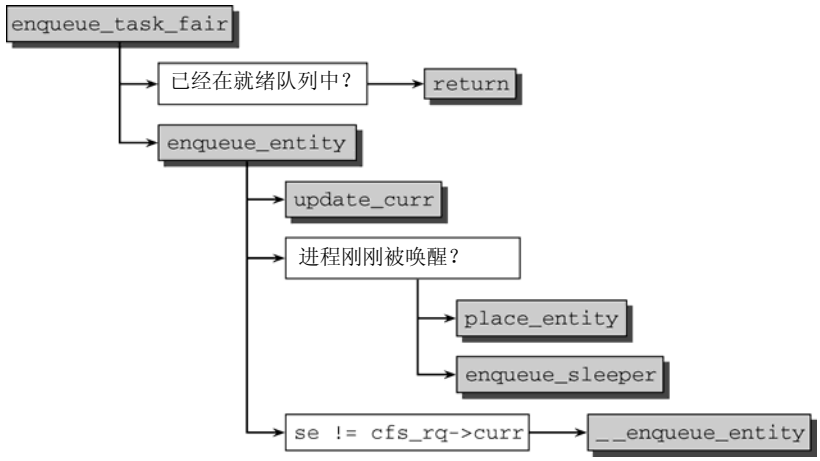

图2-20 enqueue_task_fair的代码流程图

如果通过struct sched_entity的on_rq成员判断进程已经在就绪队列上，则无事可做。否则，具体的工作委托给enqueue_entity完成，其中内核会借机用updater_curr更新统计量。

如果进程最近在运行，其虚拟运行时间仍然有效，那么（除非它当前在执行中）它可以直接用__enqueue_entity加入红黑树中。该函数需要一些处理红黑树的机制，但这可以依靠内核的标准方法（更多信息请参见附录C），无需多虑。函数的要点在于将进程置于正确的位置，这可以通过以下两点保证：此前已经设置过进程的vruntime字段，内核会不断更新队列的min_vruntime值。

如果进程此前在睡眠，那么在place_entity中首先会调整进程的虚拟运行时间[①]：

kernel/sched_fair.c

```
static void
place_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int initial)
{
        u64 vruntime;

        vruntime = cfs_rq->min_vruntime;

        if (initial)
                vruntime += sched_vslice_add(cfs_rq, se);

        if (!initial) {
                vruntime -= sysctl_sched_latency;
                vruntime = max_vruntime(se->vruntime, vruntime);
        }

        se->vruntime = vruntime;
}
```

函数根据initial的值来区分两种情况。只有在新进程被加到系统中时，才会设置该参数，但这里的情况并非如此：initial是零（在下文讨论task_new_fair时，我会说明另一种情况）。

① 要注意，在实际的内核源代码中，会根据sched_feature查询的结果来执行部分代码。CF调度器支持一些“可配置”特性，这些只能在调试状态下打开或关闭，否则特性集合是固定的。因此我忽略了特性选择机制，只考虑那些总是编译到内核中，处于活动状态的代码。

由于内核已经承诺在当前的延迟周期内使所有活动进程都至少运行一次，队列的min_vruntime用作基准虚拟时间，通过减去sysctl_sched_latency，则可以确保新唤醒的进程只有在当前延迟周期结束后才能运行。

但如果睡眠进程已经累积了比较大的不公平值（即se_vruntime值比较大），则内核必须考虑这一点。如果se->vruntime比先前计算的差值更大，则将其作为进程的vruntime，这会导致该进程在红黑树中处于比较靠左的位置，回想一下可知具有较大vruntime值的进程可以更早调度执行。

我们回到enqueue_entity：在place_entity确定了进程正确的虚拟运行时间之后，则用__enqueue_entity将其置于红黑树中。我在此前已经注意到，这是个纯粹机械性的函数，它使用了内核的标准方法将进程排序到红黑树中。

2.6.4 选择下一个进程

选择下一个将要运行的进程由pick_next_task_fair执行。其代码流程图在图2-21给出。

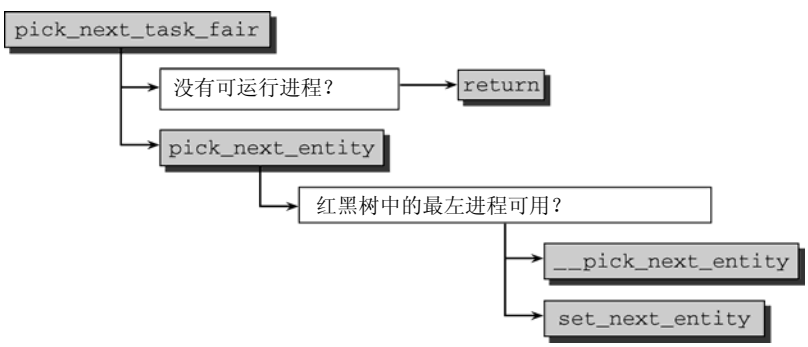

图2-21 pick_next_task_fair的代码流程图

如果nr_running计数器为0，即当前队列上没有可运行进程，则无事可做，函数可以立即返回。否则将具体工作委托给pick_next_entity。

如果树中最左边的进程可用，可以使用辅助函数first_fair立即确定，然后用__pick_next_entity从红黑树中提取出sched_entity实例。这是使用container_of机制完成的，因为红黑树管理的结点是rb_node的实例，而rb_node即嵌入在sched_entity中。

现在已经选择了进程，但还需要完成一些工作，才能将其标记为运行进程。这是通过set_next_entity处理的。

kernel/sched_fair.c

```
static void
set_next_entity(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
        /* 树中不保存“当前”进程。 */
        if (se->on_rq) {
                __dequeue_entity(cfs_rq, se);
        }
...
```

当前执行进程不保存在就绪队列上，因此使用__dequeue_entity将其从树中移除。如果当前进程是最左边的结点，则将leftmost指针设置到下一个最左边的进程。请注意在我们的例子中，进程确实已经在就绪队列上，但set_next_entity可能从不同地方调用，所以情况会有所不同。

尽管该进程不再包含在红黑树中，但进程和就绪队列之间的关联没有丢失，因为curr标记了当前运行的进程：

kernel/sched_fair.c

```
        cfs_rq->curr = se;
        se->prev_sum_exec_runtime = se->sum_exec_runtime;
}
```

因为该进程是当前活动进程，在CPU上花费的实际时间将记入sum_exec_runtime，因此内核会在prev_sum_exec_runtime保存此前的设置。要注意进程中的sum_exec_runtime没有重置。因此差值sum_exec_runtime - prev_sum_exec_runtime确实表示了在CPU上执行花费的实际时间。

2.6.5　处理周期性调度器

在处理周期调度时前述的差值很重要。形式上由函数task_tick_fair负责，但实际工作由entity_tick完成。图2-22给出了代码流程图。

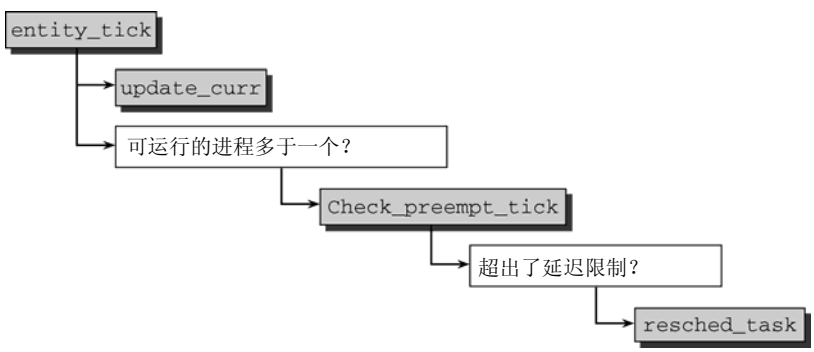

图2-22　entity_tick的代码流程图

首先，一如既往地使用update_curr更新统计量。如果队列的nr_running计数器表明队列上可运行的进程少于两个，则实际上无事可做。如果某个进程应该被抢占，那么至少需要有另一个进程能够抢占它。如果进程数目不少于两个，则由check_preempt_tick作出决策：

kernel/sched_fair.c

```
static void
check_preempt_tick(struct cfs_rq *cfs_rq, struct sched_entity *curr)
{

        unsigned long ideal_runtime, delta_exec;

        ideal_runtime = sched_slice(cfs_rq, curr);
        delta_exec = curr->sum_exec_runtime -curr->prev_sum_exec_runtime;
        if (delta_exec > ideal_runtime)
                resched_task(rq_of(cfs_rq)->curr);
}
```

该函数的目的在于，确保没有哪个进程能够比延迟周期中确定的份额运行得更长。该份额对应的实际时间长度在sched_slice中计算，如上文上述，进程在CPU上已经运行的实际时间间隔由sum_exec_runtime-prev_sum_exec_runtime给出。因此抢占决策很容易作出：如果进程运行时间比期望的时间间隔长，那么通过resched_task发出重调度请求。这会在task_struct中设置TIF_NEED_RESCHED标志，核心调度器会在下一个适当时机发起重调度。

2.6.6 唤醒抢占

当在try_to_wake_up和wake_up_new_task中唤醒进程时，内核使用check_preempt_curr看看是否新进程可以抢占当前运行的进程。请注意该过程不涉及核心调度器！对完全公平调度器处理的进程，则由check_preempt_wakeup函数执行该检测。

新唤醒的进程不必一定由完全公平调度器处理。如果新进程是一个实时进程，则会立即请求重调度，因为实时进程总是会抢占CFS进程：

kernel/sched_fair.c

```
static void check_preempt_wakeup(struct rq *rq, struct task_struct *p)
{
        struct task_struct *curr = rq->curr;
        struct cfs_rq *cfs_rq = task_cfs_rq(curr);
        struct sched_entity *se = &curr->se, *pse = &p->se;
        unsigned long gran;

        if (unlikely(rt_prio(p->prio))) {
                update_rq_clock(rq);
                update_curr(cfs_rq);
                resched_task(curr);
                return;
        }
...
```

最便于处理的情况是SCHED_BATCH进程，根据定义它们不抢占其他进程。

kernel/sched.c

```
        if (unlikely(p->policy == SCHED_BATCH))
                return;
...
```

当运行进程被新进程抢占时，内核确保被抢占者至少已经运行了某一最小时间限额。该最小值保存在sysctl_sched_wakeup_granularity，我们此前已经遇到。回想可知其默认值设置为4毫秒。这指的是实际时间，因此在必要的情况下内核首先需要将其转换为虚拟时间：

kernel/sched_fair.c

```
        gran = sysctl_sched_wakeup_granularity;
        if (unlikely(se->load.weight != NICE_0_LOAD))
                gran = calc_delta_fair(gran, &se->load);
...
```

如果新进程的虚拟运行时间，加上最小时间限额，仍然小于当前执行进程的虚拟运行时间（由其调度实体se表示），则请求重调度：

kernel/sched_fair.c

```
        if (pse->vruntime + gran < se->vruntime)
                resched_task(curr);
}
```

增加的时间“缓冲”确保了进程不至于切换得太频繁，避免了花费过多的时间用于上下文切换，而非实际工作。

2.6.7 处理新进程

我们对完全公平调度器需要考虑的最后一个操作是创建新进程时调用的挂钩函数：task_new_fair。该函数的行为可使用参数sysctl_sched_child_runs_first控制。顾名思义，该参数用

于判断新建子进程是否应该在父进程之前运行。这通常是有益的，特别是在子进程随后会执行exec系统调用的情况下。该参数的默认设置是1，但可以通过/proc/sys/kernel/sched_child_runs_ first修改。

该函数先用update_curr进行通常的统计量更新，然后调用此前讨论过的place_entity：

kernel/sched_fair.c

```
static void task_new_fair(struct rq *rq, struct task_struct *p)
{
        struct cfs_rq *cfs_rq = task_cfs_rq(p);
        struct sched_entity *se = &p->se, *curr = cfs_rq->curr;
        int this_cpu = smp_processor_id();

        update_curr(cfs_rq);
        place_entity(cfs_rq, se, 1);
...
```

在这种情况下，调用place_entity时的initial参数设置为1，以便用sched_vslice_add计算初始的vruntime。回想一下，可知这实际上确定了进程在延迟周期中所占的时间份额，只是转换为虚拟时间。这是调度器最初向进程欠下的债务。

kernel/sched_fair.c

```
        if (sysctl_sched_child_runs_first && curr->vruntime < se->vruntime) {
                swap(curr->vruntime, se->vruntime);
        }

        enqueue_task_fair(rq, p, 0);
        resched_task(rq->curr);
}
```

如果父进程的虚拟运行时间（由curr表示）小于子进程的虚拟运行时间，则意味着父进程将在子进程之前调度运行。回想一下前文的内容，可知虚拟运算时间比较小，则在红黑树中的位置比较靠左。如果子进程应该在父进程之前运行，则二者的虚拟运算时间需要换过来。

然后子进程按常规加入就绪队列，并请求重调度。

2.7 实时调度类

按照POSIX标准的强制要求，除了“普通”进程之外，Linux还支持两种实时调度类。调度器结构使得实时进程可以平滑地集成到内核中，而无需修改核心调度器，这显然是调度类带来的好处。[①]

现在比较适合于回想一些很久以前讨论过的事实。实时进程的特点在于其优先级比普通进程高，对应地，其static_prio值总是比普通进程低，如图2-14所示。rt_task宏通过检查其优先级来证实给定进程是否是实时进程，而task_has_rt_policy则检测进程是否关联到实时调度策略。

2.7.1 性质

实时进程与普通进程有一个根本的不同之处：如果系统中有一个实时进程且可运行，那么调度器总是会选中它运行，除非有另一个优先级更高的实时进程。

现有的两种实时类，不同之处如下所示。

- 循环进程（SCHED_RR）有时间片，其值在进程运行时会减少，就像是普通进程。在所有的时间段都到期后，则该值重置为初始值，而进程则置于队列的末尾。这确保了在有几个优先级

① 完全公平调度器在唤醒抢占代码部分需要了解实时进程的存在，但这需要的工作量微乎其微。

相同的SCHED_RR进程的情况下，它们总是依次执行。

- 先进先出进程（SCHED_FIFO）没有时间片，在被调度器选择执行后，可以运行任意长时间。

很明显，如果实时进程编写得比较差，系统可能变得无法使用。只要写一个无限循环，循环体内不进入睡眠即可。在编写实时应用程序时，应该多加小心。[①]

2.7.2 数据结构

实时进程的调度类定义如下：

kernel/sched-rt.c

```
const struct sched_class rt_sched_class = {
        .next = &fair_sched_class,
        .enqueue_task = enqueue_task_rt,
        .dequeue_task = dequeue_task_rt,
        .yield_task = yield_task_rt,

        .check_preempt_curr = check_preempt_curr_rt,

        .pick_next_task = pick_next_task_rt,
        .put_prev_task = put_prev_task_rt,

        .set_curr_task = set_curr_task_rt,
        .task_tick = task_tick_rt,
};
```

实时调度器类的实现比完全公平调度器简单。大约只需要250行代码，而CFS则需要1 100行！

核心调度器的就绪队列也包含了用于实时进程的子就绪队列，是一个嵌入的struct rt_rq实例：

kernel/sched.c

```
struct rq {
...
        t_rq rt;
...
}
```

就绪队列非常简单，链表就足够了[②]：

kernel/sched.c

```
struct rt_prio_array {
        DECLARE_BITMAP(bitmap, MAX_RT_PRIO+1); /* 包含1比特用于间隔符 */
        struct list_head queue[MAX_RT_PRIO];
};

struct rt_rq {
        struct rt_prio_array active;
};
```

具有相同优先级的所有实时进程都保存在一个链表中，表头为active.queue[prio]，而active.bitmap位图中的每个比特位对应于一个链表，凡包含了进程的链表，对应的比特位则置位。如果链表中没有进程，则对应的比特位不置位。图2-23说明了具体情形。

① 请注意，在内核2.6.25引入实时组调度之后，这种情况会有所缓解。在本书撰写时，该特性仍然在开发中。

② SMP系统需要更多的结构成员，用于负载均衡，但我们在此不关心这些。

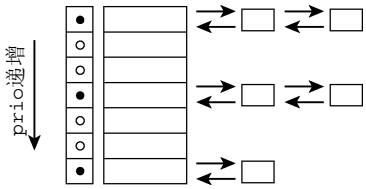

图2-23　实时调度器的就绪队列

实时调度器类中对应于update_cur的是update_curr_rt，该函数将当前进程在CPU上执行花费的时间记录在sum_exec_runtime中。所有计算的单位都是实际时间，不需要虚拟时间。这样就简化了很多。

2.7.3　调度器操作

进程的入队和离队都比较简单。只需以p->prio为索引访问queue数组queue[p->prio]，即可获得正确的链表，将进程加入链表或从链表删除即可。如果队列中至少有一个进程，则将位图中对应的比特位置位；如果队列中没有进程，则清除位图中对应的比特位。请注意，新进程总是排列在每个链表的末尾。

两个比较有趣的操作分别是，如何选择下一个将要执行的进程，以及如何处理抢占。首先考虑pick_next_task_rt，该函数放置选择下一个将执行的进程。其代码流程图在图2-24给出。

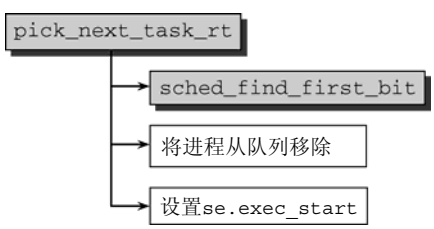

图2-24　pick_next_task_rt的代码流程图

sched_find_first_bit是一个标准函数，可以找到active.bitmap中第一个置位的比特位，这意味着高的实时优先级（对应于较低的内核优先级值），因此在较低的实时优先级之前处理。取出所选链表的第一个进程，并将se.exec_start设置为就绪队列的当前实际时钟值，即可。

周期调度的实现同样简单。SCHED_FIFO进程最容易处理。它们可以运行任意长的时间，而且必须使用yield系统调用将控制权显式传递给另一个进程：

kernel/sched.c

```
static void task_tick_rt(struct rq *rq, struct task_struct *p)
{
        update_curr_rt(rq);

        /*
         * 循环进程需要一种特殊形式的时间片管理。
         * 先进先出进程没有时间片。
         */
        if (p->policy != SCHED_RR)
                return;
...
```

如果当前进程是循环进程，则减少其时间片。在尚未超出时间段时，没什么可作的，进程可以继续执行。计数器归0后，其值重置为DEF_TIMESLICE，即100 * HZ / 1000，亦即100毫秒。如果该进程不是链表中唯一的进程，则重新排队到末尾。通过用set_tsk_need_resched设置TIF_NEED_RESCHED标志，照常请求重调度：

kernel/sched-rt.c

```
        if (--p->time_slice)
                return;

        p->time_slice = DEF_TIMESLICE;

        /*
         * 如果不是队列上的唯一成员，则重新排队到末尾。
         */
        if (p->run_list.prev != p->run_list.next) {
                requeue_task_rt(rq, p);
                set_tsk_need_resched(p);
        }
}
```

为将进程转换为实时进程，必须使用sched_setscheduler系统调用。这里不详细讨论该函数了，因为它只执行了下列简单任务。

- 使用deactivate_task将进程从当前队列移除。
- 在task_struct中设置实时优先级和调度类。
- 重新激活进程。

如果进程此前不在任何就绪队列上，那么只需要设置调度类和新的优先级数值。停止进程活动和重激活则是不必要的。

要注意，只有具有root权限（或等价于CAP_SYS_NICE）的进程执行了sched_setscheduler系统调用，才能修改调度器类或优先级。否则，下列规则适用。

- 调度类只能从SCHED_NORMAL改为SCHED_BATCH，或反过来。改为SCHED_FIFO是不可能的。
- 只有目标进程的UID或EUID与调用者进程的EUID相同时，才能修改目标进程的优先级。此外，优先级只能降低，不能提升。

2.8 调度器增强

到目前为止，我们只考虑了实时系统上的调度。事实上，Linux可以做得更好些。除了支持多个CPU之外，内核也提供其他几种与调度相关的增强功能，在以后几节里会论述。但请注意，这些增强功能大大增加了调度器的复杂性，因此我主要考虑简化的情形，目的在于说明实质性的原理，而不考虑所有的边界情形和调度中出现的奇异情况。

2.8.1 SMP 调度

多处理器系统上，内核必须考虑几个额外的问题，以确保良好的调度。

- CPU负荷必须尽可能公平地在所有的处理器上共享。如果一个处理器负责3个并发的应用程序，而另一个只能处理空闲进程，那是没有意义的。
- 进程与系统中某些处理器的亲合性（affinity）必须是可设置的。例如在4个CPU系统中，可以将计算密集型应用程序绑定到前3个CPU，而剩余的（交互式）进程则在第4个CPU上运行。
- 内核必须能够将进程从一个CPU迁移到另一个。但该选项必须谨慎使用，因为它会严重危害

性能。在小型SMP系统上CPU高速缓存是最大的问题。对于真正大型系统，CPU与迁移进程此前使用的物理内存距离可能有若干米，因此对该进程内存的访问代价高昂。

进程对特定CPU的亲合性，定义在task_struct的cpus_allowed成员中。Linux提供了sched_setaffinity系统调用，可修改进程与CPU的现有分配关系。

1. 数据结构的扩展

在SMP系统上，每个调度器类的调度方法必须增加两个额外的函数：

<sched.h>

```
struct sched_class {
...
#ifdef CONFIG_SMP
        unsigned long (*load_balance) (struct rq *this_rq, int this_cpu,
                        struct rq *busiest, unsigned long max_load_move,
                        struct sched_domain *sd, enum cpu_idle_type idle,
                        int *all_pinned, int *this_best_prio);

        int (*move_one_task) (struct rq *this_rq, int this_cpu,
                              struct rq *busiest, struct sched_domain *sd,
                              enum cpu_idle_type idle);
#endif
...
}
```

虽然其名字称之为load_balance，但这些函数并不直接负责处理负载均衡。每当内核认为有必要重新均衡时，核心调度器代码都会调用这些函数。特定于调度器类的函数接下来建立一个迭代器，使得核心调度器能够遍历所有可能迁移到另一个队列的备选进程，但各个调度器类的内部结构不能因为迭代器而暴露给核心调度器。load_balance函数指针采用了一般性的函数load_balance，而move_one_task则使用了iter_move_one_task。这些函数用于不同的目的。

- iter_move_one_task从最忙碌的就绪队列移出一个进程，迁移到当前CPU的就绪队列。
- load_balance则允许从最忙的就绪队列分配多个进程到当前CPU，但移动的负荷不能比max_load_move更多。

负载均衡处理过程是如何发起的？在SMP系统上，周期性调度器函数scheduler_tick按上文所述完成所有系统都需要的任务之后，会调用trigger_load_balance函数。这会引发SCHEDULE_SOFTIRQ软中断softIRQ（硬件中断的软件模拟，更多细节请参见第14章），该中断确保会在适当的时机执行run_rebalance_domains。该函数最终对当前CPU调用rebalance_domains，实现负载均衡。时序如图2-25所示。

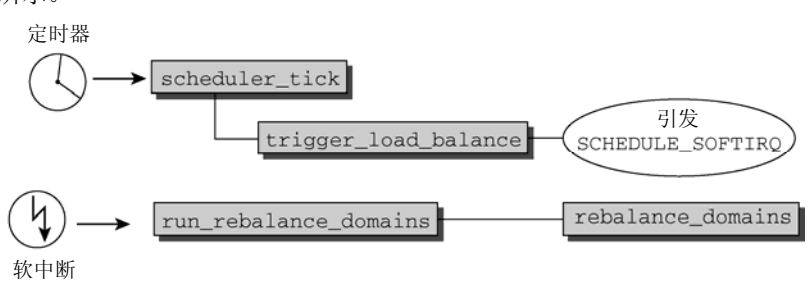

图2-25 在SMP系统上发起负载均衡的时序

为执行重新均衡的操作，内核需要更多信息。因此在SMP系统上，就绪队列增加了额外的字段：

kernel/sched.c

```
struct rq {
...
#ifdef CONFIG_SMP
        struct sched_domain *sd;
        /* 用于主动均衡 */
        int active_balance;
        int push_cpu;
        /*该就绪队列的CPU:  */
        int cpu;

        struct task_struct *migration_thread;
        struct list_head migration_queue;
#endif
...
}
```

就绪队列是特定于CPU的，因此cpu表示了该就绪队列所属的处理器。内核为每个就绪队列提供了一个*迁移线程*，可以接收迁移请求，这些请求保存在链表migration_queue中。这样的请求通常发源于调度器自身，但如果进程被限制在某一特定的CPU集合上，而不能在当前执行的CPU上继续运行时，也可能出现这样的请求。内核试图周期性地均衡就绪队列，但如果对某个就绪队列效果不佳，则必须使用主动均衡（active balancing）。如果需要主动均衡，则将active_balance设置为非零值，而cpu则记录了从哪个处理器发起的主动均衡请求。

此外，所有的就绪队列组织为*调度域*（scheduling domain）。这可以将物理上邻近或共享高速缓存的CPU群集起来，应优先选择在这些CPU之间迁移进程。但在“普通”的SMP系统上，所有的处理器都包含在一个调度域中。因此我不会详细讨论该结构，要提的一点是该结构包含了大量参数，可以通过/proc/sys/kernel/cpuX/domainY设置。其中包括了在多长时间之后发起负载均衡（包括最大/最小时间间隔），导致队列需要重新均衡的最小不平衡值，等等。此外该结构还管理一些字段，可以在运行时设置，使得内核能够跟踪记录上一次均衡操作在何时执行，下一次将在何时执行。

那么load_balance做什么呢？该函数会检测在上一次重新均衡操作之后是否已经过去了足够的时间，在必要的情况下通过调用load_balance发起一轮新的重新均衡操作。该函数的代码流程图如图2-26所示。请注意，我在该图中描述的是一个简化的版本，因为SMP调度器必须处理大量边边角角的情况。如果都画出来，相关的细节会扰乱图中真正的实质性操作。

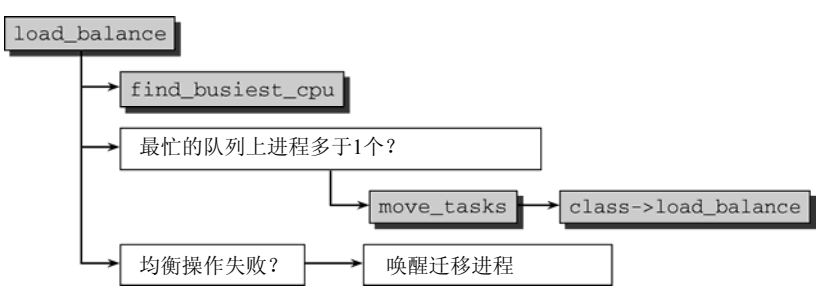

图2-26 load_balance的代码流程图

首先该函数必须标识出哪个队列工作量最大。该任务委托给find_busiest_queue，后者对一个特定的就绪队列调用。函数迭代所有处理器的队列（或确切地说，当前调度组中的所有处理器），比较其负荷权重。最忙的队列就是最后找到的负荷值最大的队列。

在find_busiest_queue标识出一个非常繁忙的队列之后，如果至少有一个进程在该队列上执行（否则负载均衡就没多大意义），则使用move_tasks将该队列中适当数目的进程迁移到当前队列。move_tasks函数接下来会调用特定于调度器类的load_balance方法。

在选择被迁移的进程时，内核必须确保所述的进程：

- 目前没有运行或刚结束运行，因为对运行进程而言，CPU高速缓存充满了进程的数据，迁移该进程则完全抵消了高速缓存带来的好处；
- 根据其CPU亲合性，可以在与当前队列关联的处理器上执行。

如果均衡操作失败（例如，远程队列上所有进程都有较高的内核内部优先级值，即较低的nice值），那么将唤醒负责最忙的就绪队列的迁移线程。为确保主动负载均衡执行得比上述方法更积极一点，load_balance会设置最忙的就绪队列的active_balance标志，并将发起请求的CPU记录到rq->cpu。

2. 迁移线程

迁移线程用于两个目的。一个是用于完成发自调度器的迁移请求，另外一个是用于实现主动均衡。迁移线程是一个执行migration_thread的内核线程。该函数的代码流程图如图2-27所示。

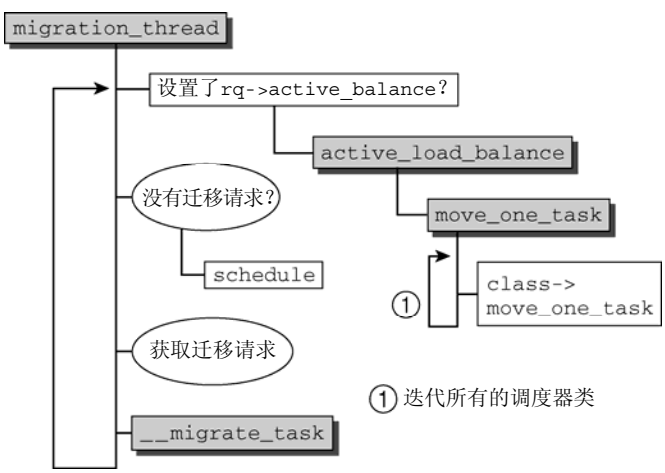

图2-27 migration_thread的代码流程图

migration_thread内部是一个无限循环，在无事可做时进入睡眠状态。首先，该函数检测是否需要主动均衡。如果需要，则调用active_load_balance满足该请求。该函数试图从当前就绪队列移出一个进程，且移至发起主动均衡请求CPU的就绪队列。它使用move_one_task完成该工作，后者又对所有的调度器类，分别调用特定于调度器类的move_one_task函数，直至其中一个成功。注意，这些函数移动进程时会尝试比load_balance更激烈的方法。例如，它们不进行此前提到的优先级比较，因此它们更有可能成功。

完成主动负载均衡之后，迁移线程会检测migrate_req链表中是否有来自调度器的待决迁移请求。如果没有，则线程发出重调度请求。否则，用__migrate_task完成相关请求，该函数会直接移出所要求的进程，而不再与调度器类进一步交互。

3. 核心调度器的改变

除了上述增加的特性之外，在SMP系统上还需要对核心调度器的现存方法作一些修改。虽然到处都是一些小的细节变化，与单处理器系统相比最重要的差别如下所示。

- 在用exec系统调用启动一个新进程时，是调度器跨越CPU移动该进程的一个良好的时机。事实上，该进程尚未执行，因此将其移动到另一个CPU不会带来对CPU高速缓存的负面效应。exec系统调用会调用挂钩函数sched_exec，其代码流程图如图2-28所示。sched_balance_self挑选当前负荷最少的CPU（而且进程得允许在该CPU上运行）。如果不是当前CPU，那么会使用sched_migrate_task，向迁移线程发送一个迁移请求。
- 完全公平调度器的调度粒度与CPU的数目是成比例的。系统中处理器越多，可以采用的调度粒度就越大。sysctl_sched_min_granularity和sysctl_sched_latency都乘以校正因子$1+\log_2(\text{nr_cpus})$，其中nr_cpus表示现有的CPU的数目。但它们不能超出200毫秒。sysctl_sched_wakeup_granularity也需要乘以该因子，但没有上界。

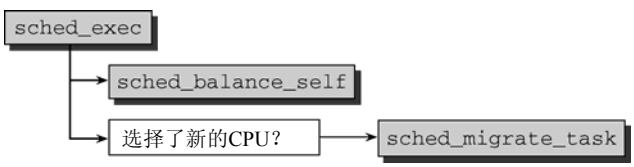

图2-28 sched_exec的代码流程图

2.8.2 调度域和控制组

在此前对调度器代码的讨论中，调度器并不直接与进程交互，而是处理可调度实体。这使得可以实现组调度：进程置于不同的组中，调度器首先在这些组之间保证公平，然后在组中的所有进程之间保证公平。举例来说，这使得可以向每个用户授予相同的CPU时间份额。在调度器确定每个用户获得多长时间之后，确定的时间间隔以公平的方式分配到该用户的进程。事实上，这意味着一个用户运行的进程越多，那么每个进程获得的CPU份额就越少。但用户获得的总时间不受进程数目的影响。

把进程按用户分组不是唯一可能的做法。内核还提供了控制组（control group），该特性使得通过特殊文件系统cgroups可以创建任意的进程集合，甚至可以分为多个层次。该情形如图2-29所示。

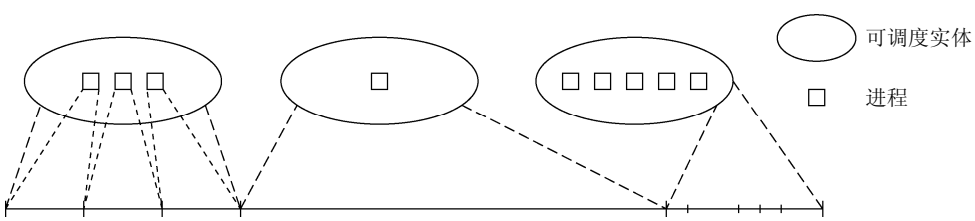

图2-29 公平的组调度概观：可用的CPU时间首先在调度组之间公平地分配，然后在每个组内的进程之间分配

为反映内核中的此种层次化情形，struct sched_entity增加了一个成员，用以表示这种层次结构：

<sched.h>

```
struct sched_entity {
...
#ifdef CONFIG_FAIR_GROUP_SCHED
        struct sched_entity *parent;
...
#endif
...
```

```
}
```

所有调度类相关的操作，都必须考虑到调度实体的这种子结构。举例来说，考虑一下在完全公平调度器将进程加入就绪队列的实际代码：

kernel/sched_fair.c

```
static void enqueue_task_fair(struct rq *rq, struct task_struct *p, int wakeup)
{
        struct cfs_rq *cfs_rq;
        struct sched_entity *se = &p->se;

        for_each_sched_entity(se) {
                if (se->on_rq)
                        break;
                cfs_rq = cfs_rq_of(se);
                enqueue_entity(cfs_rq, se, wakeup);
                wakeup = 1;
        }
}
```

for_each_sched_entity会遍历由sched_entity的parent成员定义的调度层次结构，每个实体都加入到就绪队列。

请注意，for_each_sched_entity实际上是一个平凡的循环。如果未选择支持组调度，则会退化为只执行一次循环体中的代码，因此又恢复了先前的讨论所描述的行为特性。

2.8.3 内核抢占和低延迟相关工作

我们现在把注意力转向内核抢占，该特性用来为系统提供更平滑的体验，特别是在多媒体环境下。与此密切相关的是内核进行的低延迟方面的工作，我会稍后讨论。

1. 内核抢占

如上所述，在系统调用后返回用户状态之前，或者是内核中某些指定的点上，都会调用调度器。这确保除了一些明确指定的情况之外，内核是无法中断的，这不同于用户进程。如果内核处于相对耗时较长的操作中，比如文件系统或内存管理相关的任务，这种行为可能会带来问题。内核代表特定的进程执行相当长的时间，而其他进程则无法运行。这可能导致系统延迟增加，用户体验到“缓慢的”响应。如果多媒体应用长时间无法得到CPU，则可能发生视频和音频漏失现象。

在编译内核时启用对*内核抢占*的支持，则可以解决这些问题。如果高优先级进程有事情需要完成，那么在启用内核抢占的情况下，不仅用户空间应用程序可以被中断，内核也可以被中断。切记，内核抢占和用户层进程被其他进程抢占是两个不同的概念！

内核抢占是在内核版本2.5开发期间增加的。尽管使内核可抢占所需的改动非常少，但该机制不像抢占用户空间进程那样容易实现。如果内核无法一次性完成某些操作（例如，对数据结构的操作），那么可能出现*竞态条件*而使得系统不一致。在多处理器系统上出现的同样的问题会在第5章论述。

因此内核不能在任意点上被中断。幸运的是，大多数不能中断的点已经被SMP实现标识出来了，并且在实现内核抢占时可以重用这些信息。内核的某些易于出现问题的部分每次只能由一个处理器访问，这些部分使用所谓的*自旋锁*保护：到达危险区域（亦称之为临界区）的第一个处理器会获得锁，在离开该区域时释放该锁。另一个想要访问该区域的处理器在此期间必须等待，直到第一个处理器释放锁为止。只有此时它才能获得锁并进入临界区。

如果内核可以被抢占，即使单处理器系统也会像是SMP系统。考虑正在临界区内部工作的内核被抢占的情形。下一个进程也在核心态操作，凑巧也想要访问同一个临界区。这实际上等价于两个处理

器在临界区中工作，我们必须防止这种情形。每次内核进入临界区时，我们必须停用内核抢占。

内核如何跟踪它是否能够被抢占？回想一下，可知系统中的每个进程都有一个特定于体系结构的struct thread_info实例。该结构也包含了一个抢占计数器（preemption counter）：

<asm-arch/thread_info.h>

```
struct thread_info {
...
        int preempt_count; /* 0 => 可抢占，  <0 => BUG */
...
}
```

该成员的值确定了内核当前是否处于一个可以被中断的位置。如果preempt_count为零，则内核可以被中断，否则不行。该值不能直接操作，只能通过辅助函数dec_preempt_count和inc_preempt_count，这两个函数分别对计数器减1和加1。每次内核进入重要区域，需要禁止抢占时，都会调用inc_preempt_count。在退出该区域时，则调用dec_preempt_count将抢占计数器的值减1。由于内核可能通过不同路线进入某些重要的区域，特别是嵌套的路线，因此preempt_count使用简单的布尔变量是不够的。在陆续进入多个临界区时，在内核再次启用抢占之前，必须确认已经离开所有的临界区。

dec_preempt_count和inc_preempt_count调用会集成到SMP系统的同步操作中（参见第5章）。无论如何，对这两个函数的调用都已经出现在内核的所有相关点上，因此抢占机制只需重用现存的基础设施即可。

还有更多的例程可用于抢占处理。

- preempt_disable通过调用inc_preempt_count停用抢占。此外，会指示编译器避免某些内存优化，以免导致某些与抢占机制相关的问题。
- preempt_check_resched会检测是否有必要进行调度，如有必要则进行。
- preempt_enable启用内核抢占，然后用preempt_check_resched检测是否有必要重调度。
- preempt_disable_no_resched停用抢占，但不进行重调度。

> 在内核中的某些点，普通SMP同步方法提供的保护是不够的。例如，在修改per-cpu变量时可能会发生这种情况。在真正的SMP系统上，这不需要任何形式的保护，因为根据定义只有一个处理器能够操作该变量，系统中其他的每个CPU都有自身的变量实例，不需要访问当前处理器的实例。但内核抢占的出现，使得同一处理器上的两个不同代码路径可以“准并发”地访问该变量，这与两个独立的处理器操作该值的效果是相同的。因此在这些情况下，必须手动调用preempt_disable显式停用抢占。
>
> 但要注意，第1章提到的get_cpu和put_cpu函数会自动停用内核抢占，因此如果使用该机制访问per-cpu变量，则没有必要特别注意。

内核如何知道是否需要抢占？首先，必须设置TIF_NEED_RESCHED标志来通知有进程在等待得到CPU时间。这是通过preempt_check_resched来确认的：

<preempt.h>

```
#define preempt_check_resched() \
do {\
        if (unlikely(test_thread_flag(TIF_NEED_RESCHED))) \
                preempt_schedule(); \
} while (0)
```

我们知道该函数是在抢占停用后重新启用时调用的，此时检测是否有进程打算抢占当前执行的内

核代码，是一个比较好的时机。如果是这样，则应尽快完成，而无需等待下一次对调度器的例行调用。

抢占机制中主要的函数是preempt_schedule。设置了TIF_NEED_RESCHED标志，并不能保证一定可以抢占内核，内核有可能正处于临界区中，不能被干扰。可以通过preempt_reschedule检查：

kernel/sched.c

```
asmlinkage void __sched preempt_schedule(void)
{
        struct thread_info *ti = current_thread_info();
        /*
         * 如果preempt_count非零，或中断停用，
         * 我们不想要抢占当前进程，返回即可。
         */
        if (unlikely(ti->preempt_count || irqs_disabled()))
                return;
...
```

如果抢占计数器大于0，那么抢占仍然是停用的，因此内核不能被中断，该函数立即结束。如果在某些重要的点上内核停用了硬件中断，以保证一次性完成相关的处理，那么抢占也是不可能的。irqs_disabled会检测是否停用了中断，如果已经停用，则内核不能被抢占。

如果可以抢占，则需要执行下列步骤：

kernel/sched.c

```
do {
        add_preempt_count(PREEMPT_ACTIVE);

        schedule();

        sub_preempt_count(PREEMPT_ACTIVE);
        /*
         * 再次检查，以免在schedule和当前点之间错过了抢占的时机。
         */
} while (unlikely(test_thread_flag(TIF_NEED_RESCHED)));
```

在调用调度器之前，抢占计数器的值设置为PREEMPT_ACTIVE。这设置了抢占计数器中的一个标志位，使之有一个很大的值，这样就不受普通的抢占计数器加1操作的影响了，如图2-30所示。它向schedule函数表明，调度不是以普通方式引发的，而是由于内核抢占。在内核重调度之后，代码流程回到当前进程。此时标志位已经再次移除，这可能是在一段时间之后，此间的这段时间供抢先的进程执行。

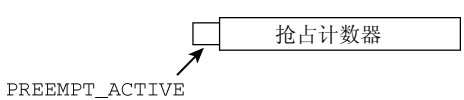

图2-30 进程的抢占计数器

此前我忽略了该标志与schedule的关系，因此必须在这里讨论。我们知道，如果进程目前不处于可运行状态，则调度器会用deactivate_task停止其活动。实际上，如果调度是由抢占机制发起的（查看抢占计数器中是否设置了PREEMPT_ACTIVE），则会跳过该操作：

kernel/sched.c

```
asmlinkage void __sched schedule(void) {
...
        if (prev->state && !(preempt_count() & PREEMPT_ACTIVE)) {
                if (unlikely((prev->state & TASK_INTERRUPTIBLE) &&
                                unlikely(signal_pending(prev)))) {
```

```
                prev->state = TASK_RUNNING;
        } else {
                deactivate_task(rq, prev, 1);
        }
    }
...
}
```

这确保了尽可能快速地选择下一个进程，而无需停止当前进程的活动。如果一个高优先级进程在等待调度，则调度器类将会选择该进程，使其运行。

该方法只是触发内核抢占的一种方法。另一种激活抢占的可能方法是在处理了一个硬件中断请求之后。如果处理器在处理中断请求后返回核心态（返回用户状态则没有影响），特定于体系结构的汇编例程会检查抢占计数器值是否为0，即是否允许抢占，以及是否设置了重调度标志，类似于preempt_schedule的处理。如果两个条件都满足，则调用调度器，这一次是通过preempt_schedule_irq，表明抢占请求发自中断上下文。该函数和preempt_schedule之间的本质区别是，preempt_schedule_irq调用时停用了中断，防止中断造成递归调用。

根据本节讲述的方法可知，启用了抢占特性的内核能够比普通内核更快速地用紧急进程替代当前进程。

2. 低延迟

当然，即使没有启用内核抢占，内核也很关注提供良好的延迟时间。例如，这对于网络服务器是很重要的。尽管此类环境不需要内核抢占引入的开销，但内核仍然应该以合理的速度响应重要的事件。例如，如果一网络请求到达，需要守护进程处理，那么该请求不应该被执行繁重IO操作的数据库过度延迟。我已经讨论了内核提供的一些用于缓解该问题的措施：CFS和内核抢占中的调度延迟。第5章中将讨论的实时互斥量也有助于解决该问题，但还有一个与调度有关的操作能够对此有所帮助。

基本上，内核中耗时长的操作不应该完全占据整个系统。相反，它们应该不时地检测是否有另一个进程变为可运行，并在必要的情况下调用调度器选择相应的进程运行。该机制不依赖于内核抢占，即使内核连编时未指定支持抢占，也能够降低延迟。

发起有条件重调度的函数是cond_resched。其实现如下：

kernel/sched.c

```
int __sched cond_resched(void)
{
        if (need_resched() && !(preempt_count() & PREEMPT_ACTIVE))
                __cond_resched();
                return 1;
        }
        return 0;
}
```

need_resched检查是否设置了TIF_NEED_RESCHED标志，代码另外还保证内核当前没有被抢占[①]，因此允许重调度。只要两个条件满足，那么__cond_resched会处理必要的细节并调用调度器。

如何使用cond_resched？举例来说，考虑内核读取与给定内存映射关联的内存页的情况。这可以通过无限循环完成，直至所有需要的数据读取完毕：

```
for (;;)
        /* 读入数据 */
```

① 另外，该函数还确认系统完全处于正常运行状态，例如系统尚未启动完成就不属于正常运行状态。由于这是不重要的边角情况，我从代码中省略了对应的检查。

```
        if (exit_condition)
                continue;
```

如果需要大量的读取操作，可能耗时会很长。由于进程运行在内核空间中，调度器无法象在用户空间那样撤销其CPU，假定也没有启用内核抢占。通过在每个循环迭代中调用cond_resched，即可改进此种情况。

```
for (;;)
        cond_resched();
        /* 读入数据 */
        if (exit_condition)
                continue;
```

内核代码已经仔细核查过，以找出长时间运行的函数，并在适当之处插入对cond_resched的调用。即使没有显式内核抢占，这也能够保证较高的响应速度。

遵循长期以来的UNIX内核传统，Linux的进程状态也支持可中断的和不可中断的睡眠。但在2.6.25的开发周期中，又添加了另一个状态：TASK_KILLABLE。[①]处于此状态进程正在睡眠，不响应非致命信号，但可以被致命信号杀死，这刚好与TASK_UNINTERRUPTIBLE相反。在撰写本书时，内核中适用于TASK_KILLABLE睡眠之处，都还没有修改。

在内核2.6.25和2.6.26开发期间，调度器的清理相对而言是比较多的。 在这期间增加的一个新特性是实时组调度。这意味着，通过本章介绍的组调度框架，现在也可以处理实时进程了。

另外，调度器相关的文档移到了一个专用目录Documentation/scheduler/下，旧的*O*(1)调度器的相关文档都已经过时，因而删除了。有关实时组调度的文档可以参考Documentation/scheduler/sched-rt-group.txt。

2.9 小结

Linux是一个多用户、多任务操作系统，因而必须管理来自多个用户的多个进程。在本章中，读者已经了解到进程是Linux的一个非常重要和基本的抽象。用于表示进程的数据结构与内核中几乎每个子系统都有关联。

读者已经看到Linux如何实现继承自UNIX传统的fork/exec模型，这种模型下创建的新进程与父进程形成层次关系，Linux还使用命名空间和clone系统调用对传统的UNIX模型进行了扩展。这两种模型中，进程如何感知到系统，以及哪些资源在父子进程之间共享，都可以微调。本来无关的进程之间进行通信需要采用显式的方法，并将在第5章讨论。

另外，读者已经看到，如何通过调度器在进程之间分配可用的计算资源。Linux支持可插入的调度模块，这些用于实现完全公平调度和POSIX软实时调度等策略。调度器会判断何时在进程之间切换，而上下文切换则由特定于体系结构的例程实现。

最后，我讨论了如何增强调度器来适应具有多CPU的系统，以及内核抢占和低延迟特性如何使Linux更好地处理有时间限制的情形。

① 实际上TASK_KILLABLE不是一个全新的进程状态，而是TASK_UNINTERRUPTIBLE的扩展。但其效果等价于一个全新的进程状态。