第 7 章　进程和线程

操作系统的存在归根结底是为了提供一个运行程序的空间。按照 Unix 的术语，将正在运行的程序为进程。Linux 内核和其它 Unix 变种一样，都是采用了多任务技术；它可以在许多进程之间分配时间片从而使这些进程看起来似乎在同时运行一样。这里通常是内核对有关资源的访问作出仲裁；在这种情况下，资源就是 CPU 时间。

进程传统上都有唯一的执行程序的上下文——这是说明在某个时刻它正在处理一项内容的流行的方法。在给定的时刻，我们可以精确地知道代码的哪一部分正在执行。但是有时我们希望一个进程同时处理多件事情。例如，我们可能希望 Web 浏览器获取并显示 Web 页，同时也要监视用户是否点击停止按钮。只为监视停止按钮而运行一个全新的程序显然是不必要的，但是对于 Web 浏览器来说要对其时间进行分隔也并不总是非常方便——获取一些 Web 页信息，检测停止按钮，再获取一些 Web 页信息，再重新检测停止按钮，等等。

对于这个问题的比较流行的解决方法是线程。从概念上来说，线程是同一个进程中独立的执行上下文——更简单一点地说，它们为单一进程提供了一种同时处理多件事情的方法，就像是进程是一个自行控制的微缩化了的多任务操作系统。同一线程组中的线程共享它们的全局变量并有相同的堆（heap），因此使用 **malloc** 给线程组中的一个线程分配的内存可以被该线程组中的其它线程读写。但是它们拥有不同的堆栈（它们的局部变量是不共享的）并可以同时在进程代码不同的地方运行。这样，你的 Web 浏览器可以让一个线程来获取并显示 Web 页，同时另外一个线程观测停止按钮是否被点击，并且在停止按钮被点击时停止第一个线程。

和线程等价的一种观点——这是 Linux 内核使用的观点——线程只是偶然的共享相同的全局内存空间的进程。这意味着内核无需为线程创建一种全新的机制，否则必然会和现在已经编写完成的进程处理代码造成重复，而且有关进程的讨论绝大多数也都可以应用到线程上。

当然，以上的说明仅仅适用于内核空间的线程。实际中也有用户空间的线程，它执行相同的功能，但是却是在应用层实现的。用户空间的线程和内核空间的线程相比有很多优点，也有很多缺点，但是有关这些问题的讨论超出了本书的范围。而使人更加容易造成混淆是一个名为 **kernel_thread**（2426 行）的函数，尽管该函数被赋予了这样一个名字，但是它实际和内核空间的线程没有任何关系。

部分是由于历史的原因，部分是由于 Linux 内核并没有真正区分进程和线程这两者在概念上的不同，在内核代码中进程和线程都使用更通用的名字“ 任务 ”来引用。根据同样的思路，本书中所出现的 “ 任务 ”和“ 进程 ”具有相同的意义。

调度和时间片

对 CPU 访问的裁决过程被称为调度（Scheduling）。良好的调度决策要尊重用户赋予的优先级，这可以建立一种所有进程都在同时运行的十分逼真的假象。糟糕的调度决策会使操作系统变得沉闷缓慢。这是 Linux 调度程序必须经过高度优化的一个原因。

从概念上来说，调度程序把时间分为小片断，并根据一定的原则把这些片断分配给进程。你可能已经猜到，时间的这些小片断称为时间片。

实时进程

Linux 提供了三种调度算法：一种传统的 Unix 调度程序和两个由 POSIX.1b（原名为 POSIX.4）操作系统标准所规定的“ 实时 ”调度程序。因此，本书中有时会使用实时进程（从技术上考虑，系统使用术语 “ 非实时进程（nonrealtime process）” 来作为实时进程的对应，虽然我更倾向于使用另外一个术语 unrealtime process）。不要过分计较 “ 实时 ” 这个术语，虽然——如果从硬件的角度来看待这个问题，实时意味着你可以得到有关操作系统的某种性能保证，例如有关中断等待时间的承诺，但是这一点在 Linux 实时调度规则中并没有提供。相反的，Linux 的调度规则是 “ 软件实时 ”，也就是说如果实时进程需要，它们就只把 CPU 分配给实时进程；否则就把 CPU 时间让出给非实时进程。

但是如果你真正需要，一些 Linux 的变种也承诺提供一种 “ 硬实时 ”。但是，在当前的 Linux 内核中——因此也就是在本章中——“ 实时 ” 仅指 “ 软件实时 ”。

优先级

非实时进程有两种优先级，一种是静态优先级，另一种是动态优先级。实时进程又增加了第三种优先级，实时优先级。优先级是一些简单的整数，它代表了为决定应该允许哪一个进程使用 CPU 的资源时判断方便而赋予进程的权值——优先级越高，它得到 CPU 时间的机会也就越大：

- 静态优先级——被称为 “ 静态 ” 是因为它不随时间而改变，只能由用户进行修改。它指明了在被迫和其它进程竞争 CPU 之前该进程所应该被允许的时间片的最大值。（但是也可能由于其它原因，在该时间片耗尽之前进程就被迫交出了 CPU。）
- 动态优先级——只要进程拥有 CPU，它就随着时间不断减小；当它小于 0 时，标记进程重新调度。它指明了在这个时间片中所剩余的时间量。
- 实时优先级——指明这个进程自动把 CPU 交给哪一个其它进程：较高权值的进程总是优先于较低权值的进程。因为如果一个进程不是实时进程，其优先级就是 0，所以实时进程总是优先于非实时进程的。（这并不完全正确 如同后面论述的一样，实时进程也会明确地交出 CPU，而在等待 I/O 时也会被迫交出 CPU。前面的描述仅限于能够交付 CPU 运行的进程）

进程 ID（PIDs）

传统上每个 Unix 进程都有一个唯一的标志符，它是一个被称为进程标志符（PID）的，范围在 0 到 32,767 之间的整数。PID 0 和 PID 1 对于系统有特定的意义；其它的进程标识符都被认为是普通进程。在本章后面对 **get_pid** 的讨论中，你会看到 PID 是如何生成和赋值的。

在 Linux 中，PID 不一定非要唯一——虽然通常都是唯一的，但是两个任务也可以共享一个 PID。这是 Linux 对线程支持的一个副作用，这些线程从概念上讲应该共享一个 PID，因为它们是同一个进程的一部分。在 Linux 中，你可以创建两个任务，并且共享且仅共享它们的 PID——从实际使用角度讲它们不会是线程，但是它们可以使用同一个 PID。这并没有多大的意义，但是如果你希望这样处理，Linux 是支持的。

引用计数

引用计数是多个对象之间为共享普通信息而广泛使用的技术。使用更通用的术语来说，一个或多个“容器对象”携带指向共享数据对象的指针，其中包含了一个称为“引用计数（Reference Count）”的整数；这个引用计数的值和共享数据的容器对象的个数相同。希望共享数据的新容器对象将被赋予一个指向同一结构的指针，并且递增该共享数据对象的引用计数。

当容器对象离开时，就递减共享数据的引用计数，并做到“人走灯熄”——也就是当引用计数减小到 0 时，容器对象回收共享对象。图 7.1 阐述了这种技术。

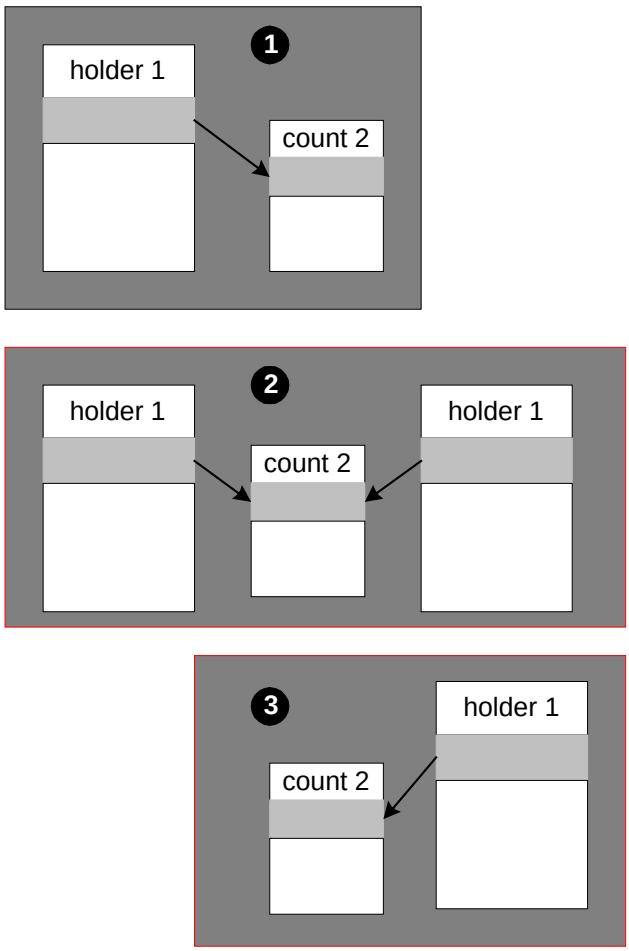

图 7.1 引用计数

就象你随后会看到的那样，Linux 通过使用引用计数技术来实现线程间的数据共享。

权能

在早期的 Unix 中，你或者是 root 用户，或者不是。如果你是 root，你几乎可以进行任何希望进行的操作，即使你的想法实际上十分糟糕，例如删除系统引导盘上的所有文件。如果你不是 root，那么你就不可能对系统造成太大的损害，但是你也不能执行任何重要的系统

管理任务。

不幸的是，很多应用程序的需要都介于这两个安全性极端之间。例如，修改系统时间是只有 root 才能执行的操作，因此实现它的程序必须作为 root 运行。但是因为是作为 root 运行的，修改系统时间的进程也就能处理 root 可以完成的任何事情。对于编写良好的程序来说并不会造成问题，但是程序仍然会有意无意地把系统搞得一团糟。（数不清的计算机攻击事件都是欺骗 root 去运行一些看似值得信任的可执行代码，造成了一些恶作剧。）

这些问题中有一些可以通过正确使用组和诸如 sudo 之类的程序而避免，但是有一些则不行。对于某些重要的操作，虽然你可能只想允许它们执行一两种权限操作，你也只能给予这些进程普通 root 访问许可。Linux 对于这个问题的解决方法是使用从现在已经舍弃了的 POSIX 草案标准中抽取出来的思想：权能。

权能使你可以更精确的定义经授权的进程所允许处理的事情。例如，你可以给一个进程授予修改系统时间的权力，而没有授予它可以杀掉系统中的其它进程、毁坏你的文件、并胡乱运行的权力。而且，为了帮助防止意外地滥用其优先级，长时间运行的进程可以暂时获得权能（如果允许），只要时间足够处理特殊的零碎工作就可以了，在处理完这个零碎的工作以后再收回权能。

在本书的编写期间，权能仍然处于开发状态。为了完全实现权能的预期功能，开发者们还必须要实现一些新的特性——例如，目前还没有内核支持将程序的权能附加到文件本身中。这样所造成的一个后果是 Linux 有时仍要检测进程是否作为 root 运行，而不是检测所进程需要的特殊权能。但是迄今为止已经实现了的内容仍然是十分有用的。

进程在内核中是如何表示的

内核使用几个数据结构来跟踪进程；其中有一些和进程自身的表示方法是密切相关的，另外一些则是独立的。图 7.2 阐述了这些数据结构，随后就会对它们进行详细介绍。

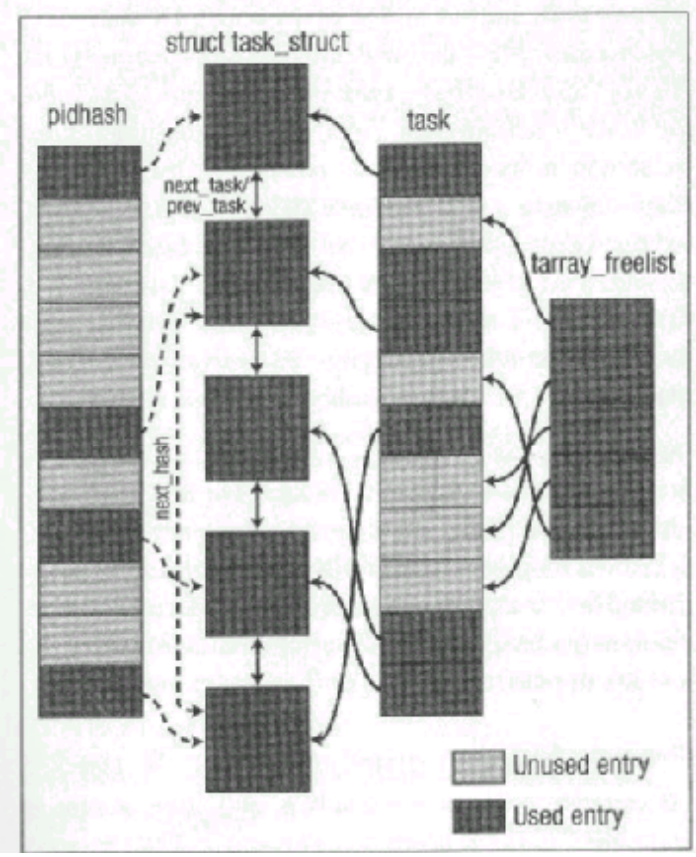

图 7.2 管理任务使用的内核数据结构

16325：表示进程的内核数据结构是 **struct task_struct**。我们暂时向前跳过这个结构的定义，继续往下看。它相当大，但是可以从逻辑上划分为很多部分。随着本章讨论的展开，你将会逐渐清楚它们每一部分的意义。在阅读的过程中，要注意这个结构的很多部分都是指向其它结构的指针；这在子孙进程和祖先进程希望共享指针所指向的信息时可以灵活运用——很多指针都指向正在被引用计数的信息。

16350：任务本身使用 **struct task_struct** 结构的 **next_task** 和 **prev_task** 成员组成一个循环的双向链接列表，它被称为任务队列。的确，这忽略了一个事实，它们在中心数组 **task**（很快就会讨论）中早已存在了。最初这看起来可能有些奇怪，但实际上这是十分有用的，因为这样允许内核代码可以遍历执行所有现存的任务——也就是 **task** 中所有经过填充的时间片——而无须浪费时间跳过空时间片。实际上对这个循环的访问是如此频繁，以至于在 16898 行单独为它定义了一个宏 **for_each_task**。

虽然 **for_each_task** 是单向的，但是它有一些值得注意的特性。首先，注意到循环的开始和末尾都是 **init_task**。这是很安全的，因为 **init_task** 从来不会退出；因此，作为标记它一直都是可用的。但是，注意到 **ini_task** 本身不是作为循环的一部分而访问的——这恰好就是你使用这个宏时所需要的东西。还有，作为我们关心的一小部分，你总是使用 **next_task** 成员直接向前遍历执行列表的；不存在相关的向后执行的宏。也没有必要需要这样一个宏——只有在需要及时把任务从列表中处理清除时才需要使用 **prev_task** 成员。

16351：Linux 还保持一个和这个任务列表类似的循环的双向任务列表。这个列表使用 **struct task_struct** 结构的 **prev_run** 成员和 **next_tun** 成员进行链接，基本上是作为队列来处理的（这真值得让人举杯庆祝）；出于这个原因，这个列表通常被称为运行队列（run

queue)。对于 **next_task** 来说，只是因为需要高效地将一个项移出队列才会使用到 **prev_run** 成员；对于这个列表的遍历循环执行通常都是使用 **next_run** 向前的。同样，在这个任务队列中也使用 **init_task** 来标记队列的开始和末尾。

通过使用 **add_to_runqueue**（26276 行）能够将任务加入队列，而使用 **del_from_runqueue**（26287 行）则把任务移出队列。有时候分别使用 **move_first_runqueue**（26318 行）和 **move_last_runqueue**（26300 行）把它们强制移动到队列的开头和末尾。注意这些函数都是局限于 kernel/sched.c 的，在别的文件中不会使用 **prev_run** 和 **next_run** 域（特别是在 kernel/fork.c 文件中的进程创建期间）；这是十分恰当的，因为只有在调度时才需要运行队列。

16370：首先，任务能够组成一个图，该图的结构表达了任务之间的家族关系；由于我不清楚这个图所使用的通用术语，我就称它为进程图（process graph）。这和 **next_task**/**prev_task** 之间的连接根本没有关系，在那里任务的位置是毫无意义的——只是一个偶然的历史事件而已。每一个 **struct task_struct** 中有五个指向进程图表中自己位置的指针。这五个指针在从 16370 行到 16371 行的代码中被定义。

- **p_opptr** 指向进程的原始祖先；通常和 **p_pptr** 类似。
- **p_pptr** 指向进程的当前祖先。
- **p_cptr** 指向进程的最年青（最近）子孙。
- **p_ysptr** 指向进程的下一个最年青（下一个最近）兄弟。
- **p_osptr** 指向进程的下一个最古老（下一个最远）兄弟。

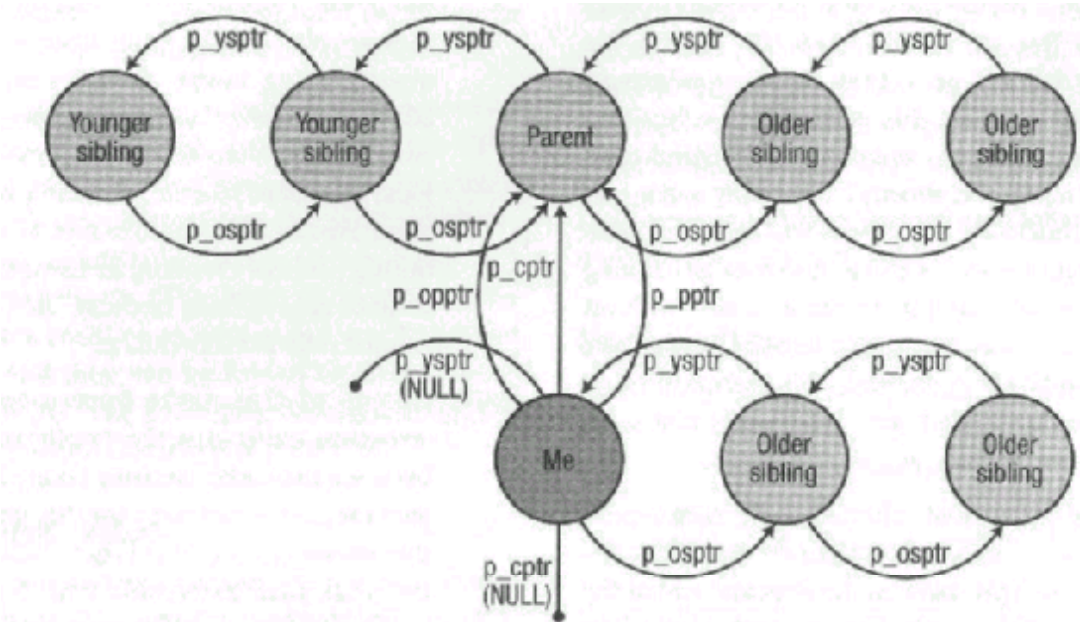

图 7.3　进程图

图 7.3 说明了它们之间的关系（整个链接集合都以标号为“Me”的节点为核心）。这个指针的集合还提供了浏览系统中进程集合的另外一种方法；显然，在处理诸如查找进程祖先或者查找列表中进程子孙时这个指针特别有用。这个指针是由两个宏维护的：

- **REMOVE_LINKS**（16876 行）从图中移出指针。
- **SET_LINKS**（16887 行）向图中插入指针。

这两个宏都可以调整 **next_task**/**prev_task** 的链接。如果你仔细研究一下这两个宏，你

就会发现它们只是增加或者删除叶子进程——而从不对拥有子孙进程的进程进行处理。

16517：**task** 定义为由指向 **struct task_struct** 结构的指针组成的数组。这个数组中的每一个项代表系统中的一个任务。数组的大小是 **NR_TASKS**（在 18320 行设置为 512），它规定了系统中可以同时运行的任务数量的上限。由于一共有 32,768 个可能的 PID，由于数组不够大，要通过它们的 PID 直接索引系统中所有任务显然是不可能的。（也就是 **task[i]**未必是由 PID **i** 指明的任务。）相反，Linux 使用其它的数据结构来帮助系统管理这种有限的资源。

16519：自由时间片列表 **tarray_freelist** 拥有一个说明 **task** 数组中自由位置的列表（实际上是一个堆栈）。它在 27966 行和 27967 行初始化，接着被两个在 16522 行到 16542 行定义的内联函数所使用。在 SMP 平台上，对于 **tarray_freelist** 的访问必须受自旋锁 **taskslot_lock**（23475 行）的限制。（自旋锁在第 10 章中详细讨论。）

16546：**pidhash** 数组有助于把 PID 映象到指向 **struct task_struct** 的指针。**pidhash** 在 27969 行和 27970 行初始化，此后它被一系列在 16548 行到 16580 行定义的宏和内联函数所操纵。这些最终实现了一个普通的哈希表。注意，为了处理 hush 记录，维护 **pidhash** 的函数使用了 **struct task_struct** 结构中的两个成员——**pidhash_next**（16374 行）和 **pidhash_pprev**（16375 行）。通过使用 **pidhash**，内核可以通过其 PID 有效地发现任务——虽然这种方式仍然比直接查找要慢。

仅仅是为了好玩，你可以自己证明这个哈希函数——**pid_hashfn**，16548 行——提供了一个均匀覆盖其域 0 到 32,767（所有有效的 PID）的发行版本。除非你所谓的“好玩”的概念和我不一样，否则你会同我一样感到有趣。

这些数据结构提供了有关当前运行系统的很多信息，但是这也需要付出代价：每当增加或删除进程时这些信息必须能够得到正确维护，否则系统就会变得混乱不堪。部分出于实现这种正确的维护非常困难的考虑，进程只在一个地方创建（使用 **do_fork**，后面会讨论），也只在一个地方删除（使用 **release**，也在后面中讨论）。

如果我们能把 **task** 处理为 32,768 个 **struct task_struct** 结构组成的数组，其中的每一项代表一个可能的 PID，那么至少可以消除一部分这种类型的复杂性。但是这样处理会大量增加内核对于内存的需求。每一个 **struct task_struct** 结构在 UP 平台上占用 964 字节，在 SMP 平台上占用 1,212 字节——取整以后，近似的数字是 1K。为了容纳所有这些结构，**task** 会像气球一样迅速膨胀到 32,768K，也就是 32M！（实际情况会更糟糕：我们尚未提到的有关任务的额外内存开销会把这个数字增长 8 倍——也就是 256M——而且不要忘记了，这些开销实际上都还没有运行一个任务。）此外，x86 的内存管理硬件把活动任务的数量限制在 4,000 左右；这一主题在下一章介绍。因此，数组中大多数的空间都会不可避免地被浪费了。

在目前的实现中，如果没有进程在运行，**task** 仅仅是 512 个 4 字节的指针，总共才 2K。如果我们考虑到那些附加的数据结构会占用一些额外开销，可能有一些超过这个数字，但是比起 32M 来还差得远呢。即使是 **task** 中的每一项都使用了，而且每个 **struct task_struct** 结构也都分配了，总共使用的内存也才不过大约 512K。应用程序能够忽略这种微小的区别。

进程状态

在一个给定的时间，进程处于下面注释中描述的六种状态中的一种。进程的当前状态被记录在 **struct task_struct** 结构的 **state** 成员中（16328 行）。

16188：**TASK_RUNNING** 意味着进程准备好运行了。即使是在 UP 系统中，也有不止一个任务同时处于 **TASK_RUNNING** 状态——**TASK_RUNNING** 并不意味着该进程可以立即获得 CPU（虽然有时候是这样），而是仅仅说明只要 CPU 一旦可用，进程就

可以立即准备好执行了。

16189：**TASK_INTERRUPTIBLE** 是两种等待状态的一种——这种状态意味着进程在等待特定事件，但是也可以被信号量中断。

16190：**TASK_UNINTERRUPTIBLE** 是另外一种等待状态。这种状态意味着进程在等待硬件条件而且不能被信号量中断。

16191：**TASK_ZOMBIE** 意味着进程已经退出了（或者已经被杀掉了），但是其相关的 **struct task_struct** 结构并没有被删除。这样即使子孙进程已经退出，也允许祖先进程对已经死去的子孙进程的状态进行查询。在本章后面我们会详细介绍这一点。

16192：**TASK_STOPPED** 意味着进程已经停止运行了。一般情况下，这意味着进程已经接收到了 **SIGSTOP**，**SIGSTP**，**SITTIN** 或者 **SIGTTOU** 信号量中的一个，但是它也可能意味着当前进程正在被跟踪（例如，进程正在调试器下运行，用户正在单步执行代码）。

16193：**TASK_SWAPPING** 主要用于表明进程正在执行磁盘交换工作。然而，这种状态似乎是没有什么用处的——虽然该标志符在整个内核中出现了好几次，但是其值从来没有被赋给进程的 **state** 成员。这种状态正在被逐渐淘汰。

进程来源：fork 和_ _clone

传统的 Unix 实现方法在系统运行以后只给出了一种创建新进程的方法：系统调用 **fork**。（如果你奇怪第一个进程是哪里来的，实际上该进程是 init，在第 4 章中已经讨论过。）当进程调用 **fork** 时，该进程从概念上被分成了两部分——这就像是道路中的分支——祖先和子孙可以自由选择不同的路径。在 **fork** 之后，祖先进程和其子进程几乎是等同的——它们所有的变量都有相同的值，它们打开的文件都相同，等等。但是，如果祖先进程改变了一个变量的值，子进程将不会看到这个变化，反之亦然。子进程是祖先进程的一个拷贝（至少最初是这样），但是它们并不共享内容。

Linux 保留了传统的 **fork** 并增加了一个更通用的函数_ **_clone**。（前面的两个下划线有助于强调普通应用程序代码不应该直接调用_ **_clone**，应该从在_ **_clone** 之上建立的线程库中调用这个函数。）鉴于 **fork** 创建一个新的子孙进程后，子孙进程虽然是其祖先进程的拷贝，但是它们并不共享任何内容，_ **_clone** 允许你定义祖先进程和子孙进程所应该共享的内容。如果你没有给_ **_clone** 提供它所能够识别的五个标志，子孙进程和祖先进程之间就不会共享任何内容，这样它就和 **fork** 类似。如果你提供了全部的五个标志，子孙进程就可以和祖先进程共享任何内容，这就和传统线程类似。其它标记的不同组合可以使你完成介于两者之间的功能。

顺便提一下，内核使用 **kernel_thread** 函数（2426 行）为了自己的使用创建了几个任务。用户从来不会调用这个函数——实际上，用户也不能调用这个函数；它只在创建例如 kswapd（在第 8 章中介绍）之类的特殊进程时才会使用，这些特殊进程有效地把内核分为很多部分，为了简单起见也把它们当作任务处理。使用 **kernel_thread** 创建的任务具有一些特殊的性质，这些性质我们在此不再深入介绍（例如，它们不能被抢占）；但是现在主要需要引起注意的是 **kernel_thread** 使用 **do_fork** 处理其垃圾工作。因此，即使是这些特殊进程，它们最终也要使用你我所使用的普通进程的创建方法来创建。

do_fork

23953：**do_fork** 是实现 **fork** 和_ **_clone** 的内核程序。

23963：分配 **struct task_struct** 结构以代表一个新的进程。

23967：给新的 **struct task_struct** 结构赋予初始值，该值直接从当前进程中拷贝而来。**do_fork** 的剩余工作主要包含为祖先进程和子孙进程不会共享的信息建立新的拷贝。（在本行和整个内核中你可以看到的 **current** 是一个宏，它把一个指针指向代表当前正在执行的进程的 **struct task_struct** 结构。这在 10285 行定义，但实际上只是对 **get_current** 函数的一个调用，而后者的定义在 10277 行。）

23981：新到达者需要 **task** 数组中的一个项；这个项是使用 **find_empty_process**（23598 行——它严格依赖于 16532 行的 **get_free_taskslot**）找到的。然而，它工作的方式有点不明显：**task** 数组没有使用的成员不是设置为空，而是设置为自由列表的下一个元素（使用 **add_free_taskslot**，16523 行）。因此，**task** 中没有使用的项指向链接列表中另外一个 **task** 没有使用的项，而 **tarray_freelist** 仅仅指向这个列表的表头。那么，返回一个自由位置就简单地变成了返回列表头的问题了（当然要把这个头指针指向下一个元素）。更传统的方法是使用一个独立的数据结构来管理这些信息，但是在内核中，空间总会显得有些不足。

23999：给新的任务赋 PID（其中的细节很快就会介绍）。

24045：本行和下面几行，使用该文件中别处定义的辅助函数，根据所提供的 **clone_flags** 参数的值为子孙进程建立祖先进程的数据结构中子孙进程所选择部分的拷贝。如果 **clone_flags** 指明相关的部分应该共享而不是拷贝，这时辅助函数（help function）就简单地增加引用计数接着返回；否则，它就创建新进程所独有的新的拷贝。

24078：到现在为止，所有进程所有的数据结构都已经设置过了，但是大部分跟踪进程的数据结构还没有被设置。系统将通过把进程增加到进程图表中开始设置它们。

24079：通过调用 **hash_pid** 把新的进程置入 **pidhash** 表中。

24088：通过调用 **wake_up_process**（26356 行）把新的进程设置为 **TASK_RUNNING** 状态并将其置入运行队列。

注意到现在不止是 **struct task_struct** 结构被设置了，而且所有相关的数据结构——自由时间片列表，任务列表、进程图、运行队列和 PID hash 表——这些都已经为新的到达者正确地进行了修改。恭喜你，你现在已经得到了一个健康的子孙任务。

PID 的分配

PID 是使用 **get_pid** 函数（23611 行）生成的，该函数能够返回一个没有使用的 PID。它从 **last_pid**（23464 行）开始——这是最近分配的 PID。

内核中使用的 **get_pid** 的版本是内核复杂性和速度频繁折中的一个例子；这里速度更为重要一些。**get_pid** 经过了高度优化——它比直接向前的实现方法要复杂的多，但是速度也要快的多。最直接的实现方法将遍历执行整个任务列表——典型的情况可能有几十项，有时候也可能成百上千项——对每一个可能的 PID 进程检测并找出适当的值。我们见到的版本有时是必须执行这些步骤的，但是在大多数情况下都可以跳过。这一结果被用来帮助加速进程创建的操作，它在 Unix 上慢得臭名卓著。

如果我们所需要的只是要为每一个运行进程都快速计算一个各不相同的整数，那么这里已经有现实可用的方法：只要取在 **task** 数组中进程的索引就可以了。平均说来，这肯定要比现在的 **get_pid** 速度要快——毕竟，这无须遍历任务列表。不幸的是，很多现存的应用程序都假定在一个 PID 可以再重用之前都需要等待一段时间。这种假定在任何情况下都是不安全的，但是在如果为了这些程序的问题而将内核牵涉进去可能仍然是一个很糟糕的思想。现存的 PID 分配策略速度仍然很快，并且它偶尔还有可以暴露这些应用程序中的潜在缺陷的优点，如果有的话（如果你认为这是一种优点）。

get_pid

23613：**next_safe** 变量是一个为加快系统运行速度而设定的变量；它保持记录了可能保留的次最低的的候选 PID。（更正确的应该把它命名为 **next_unsafe**。）当 **last_pid** 递增并超过这个范围时，系统应该检测整个任务列表来保证这个候选 PID 是否仍在被保留着（原来保留这个 PID 的进程现在可能已经运行完了）。由于遍历这个任务列表可能会很慢，所以只要可能就应该避免执行这样的操作。因此，在执行这个遍历的过程中，**get_pid** 要重新计算 **next_safe**——如果有些进程已经死掉了，这个数字可能现在更大了，因此 **get_pid** 可以避免一些将来对任务列表的遍历。（**next_safe** 是静态的，因此其值在下次 **get_pid** 需要分配 PID 时就会保留下来。）

23616：如果新的进程要和其祖先共享 PID，就返回祖先进程的 PID。

23620：开始搜寻候选 PID 寻找未使用的值。位与运算只是通过测试低 15 位是否置位来简单测试 **last_pid** 的新值是否超过了 32,767（最大允许的 PID）。我怀疑这些内核开发者真正需要通过这样做来获得微小的速度优势，但是你永远也不会知道；至少在这段代码编写期间，gcc 还不够敏锐到足以注意到它们的等价性并在生成的代码中选择稍微快速的形式。

23621：如果 **last_pid** 已经超出了允许的最大值，它就会滚动到 300。300 这个数字并没有什么魔力——它对于内核并没有特别的意义——这是另外一个加速变量。其思想是数字比较小的 PID 通常都属于系统开始运行时就已经创建的，从不会退出的长时间运行的后台监控程序。由于它们总是占据着数字比较小的 PID，所以如果不考虑对前面几百个值的重用问题，我们将会发现寻找可以使用的 PID 的过程会快许多。而且，由于 PID 的空间是同时允许的任务数（512）的 64 倍，为了追求速度而损失一些空间是一种非常值得的。

23622：由于 **last_pid** 超出了最大允许的 PID，它必然也就超出了 **next_safe**；因此，后面的 **if** 测试也可以跳过。

23624：如果 **last_pid** 仍然小于 **next_safe**，其值就可以再用。否则，必须检查任务列表。

23633：如果取得了 **last_pid** 的当前值，它就简单的递增，如果需要就跳转到 300，重新开始循环。初次看的时候，仿佛这个循环会一直运行下去——如果所有的 PID 都已经被使用了会出现什么情况呢？但是稍微考虑一下，我们就可以排除这种可能性：任务列表的最大值和同时并发的任务的最大数是相同的，有效的 PID 数目要比这两个数都大得多。因此，循环最终会找到有效的 PID；这仅仅是个时间的问题。

23651：**get_pid** 已经发现了一个没有被使用的 PID，随后返回该 PID。

运行新程序

如果我们能够进行的所有工作只是 **fork**（或者**__clone**），那么我们就只能一次次建立同一个进程的拷贝就可以了——这样我们的 Linux 系统就只能运行在系统中第一个创建的用户进程 init 了。Init 是很有用的，但是还没有功能如此强大；我们也还需要处理其它事情。

在我们创建新的进程以后，它通过调用 **exec** 就能够变成独立于其它进程的进程了。（这实际上不止是一个名为 **exec** 的函数；而是 **exec** 通常用作一个引用一系列函数的通用术语，所有这些函数基本上都处理相同的事情，但是使用的参数稍微有些不同。）

因此，创建一个“ 真正 ”的新进程——与其祖先不同的程序运行镜像——任务分为两步，一步是 **fork**，另一步是 **exec**，最后能够得出下面的风格非常熟悉的 C 代码：

P485 1

（**execl** 是 **exec** 家族若干函数中的一个。）

实现所有 **exec** 家族函数的底层内核函数是 10079 行到 10141 行的 **do_execve**。**do_execve** 处理三种工作：

- 把一些定义信息从文件读入内存。（**do_execve** 把这个工作交给 **prepare_binprm** 处理。）
- 准备新的参数和环境——这是 C 应用程序将它作为 **argc**，**argv** 和 **envp** 使用的内容。
- 装载可以解析可执行文件的二进制处理程序，并让它处理剩余的修改内核数据结构的工作。

记住这些任务，现在让我们开始仔细研究一下 **do_execve**。

do_execve

10082：代表在使用 **exec** 处理进程时所需要记录的全部信息的数据类型是 **struct linux_binprm** 结构（请参看 13786 行）——我确信 **binprm** 是“ binary parameters（二进制参数）”的缩写。**do_execve** 处理自己的工作，并使用这种类型的变量 **bprm** 同那些负责处理其部分工作的函数进行通信。注意到当 **do_execve** 返回时 **bprm** 就会被废弃——只有在执行 **exec** 时才需要 **bprm**，它并不在该进程的整个生命期中存在。

10087：**do_execve** 通过初始化一个记录新进程参数和环境分配的内存页的微型页表开始执行。它为这个目的总共需要申请 **MAX_ARG_PAGES**（在 13780 行宏定义为 32）个页，在 x86 平台上每一页是 4K，因此参数总共可以使用的空间加起来就是 32*4K=128K。作为我个人而言，我很高兴了解到这个内容，因为我偶而会超过这个限定，通常是在一个具有成百个文件的目录下运行 **cat*>/tmp/joined** 之类的东西的时候——所有这些文件名连接起来可能就超过了 128K。我通常是使用 xargs 程序解决这个问题，但是我现在也可以通过为 **MAX_ARG_PAGES** 重新定义一个比较大的值并重新编译内核来解决这个问题。至少现在如果这个问题再困扰我，我也知道该如何增加这一限制了。（可能一些热心的读者会重新编写程序来去掉这段糟糕的限制。）所以我非常喜欢拥有内核的源代码。

10091：下一步是要打开可执行文件。这不是简单的从文件中读出数据——现在的焦点是要确保文件存在，这样 **do_execve** 就可以清楚是否有必要继续进行处理。如果这是第一步，而不是首先填充 **bprm** 的页表的话，**do_execve** 在执行时有时能够获得很高的边际效应——如果这样失败了，用来初始化页表的时间就浪费了。然而，这只在文件不存在时才有用——这不是普通的情况，不值得优化。

10096：继续填充 **bprm**，特别是其 **argc** 和 **envc** 成员。为了填充这些成员，**do_execve** 使用 **count** 函数（9480 行），它通过使用被传递进来的 **argv** 和 **envp** 数组计算非空指针的个数。第一个空指针标志着列表结束，因此在到达空指针时就可以得到非空指针的个数并将其返回。这开始看起来似乎很可能因此而造成一些效率的损失：调用 **do_execve** 的函数有时早就知道了 **argv** 和 **envp** 数组的长度。因此可以再给 **do_execve** 增加两个整型参数 **argc** 和 **envc**。如果这两个参数都是非负的，那么它们就可以分别代表两个数组的长度。但是事情并没有这么简单：**count** 同时要检测它扫描的数组中是否有访问内存的错误发生。强迫（更多的情况是完全信任）**do_execve** 的调用者来对这些内容进行检测是不正确的。所以目前这样的处理方式要更好一些。

10115：主要使用 **copy_strings**（9519 行）把参数和环境变量拷贝到新进程中。**copy_strings** 看起来很复杂，但是它要处理的工作十分简单：把字符串拷贝到新进程的内存空间中，如果需要就给它们分配页。这种复杂性的增长主要出现在对页表的管理需要和跨越内核/用户空间限制的需要，这一点将在第 8 章中更详细地介绍。

10126：如果前面的工作可以很好地执行到此处，最后一步是要为新的可执行程序寻找一个二进制处理程序。如果 **search_binary_handler** 成功找到了这种程序，整个过程就成功运行结束，并返回一个非负值以说明成功。

10134：如果程序运行到了此处，那么前面的几步中肯定发生了错误。系统释放为新进程的参数和环境分配的所有页，接着必须返回一个负值通知调用者调用过程失败了。

prepare_binprm

9832： **prepare_binprm** 填写 **do_execve** 的重要部分 **bprm**。

9839： 本行开始一些健全性检测，例如要确保执行的是文件而不是目录，并且文件的可执行位已经设置了。

9858： 如果已经被设置过 setuid 和 setuid 位，就根据它们的提示新进程应该把当前执行的用户作为一个不同的用户（如果 setuid 被置位）并且/或者把它作为一个不同组的成员（如果 setgid 被置位）。

9933： 最后，**prepare_binprm** 从文件中读取前 128 个字节（而不是像该函数标题注释里说明的一样是前 512 个字节）到 **bprm** 的 **buf** 成员中。

顺便说一下，这里有一个延续已久的争论：在 13787 行，**struct linux_binprm** 结构的 buf 成员被声明为是 128 字节长，在 9933 行读入了 128 字节。但是字面上常量 128 用在两个地方——没有宏定义表示有必要保持两个数字的一致；因此，有可能会出现对其中一个进行改变而不改变相关的另一个的情况，这样就很可能摧毁系统。即使不从学术上考虑，这种忽略在保证效率的基础上是不能防止的——我不能想象出还有什么其它理由。

这是一个很好的对内核做点简短却有用的修改的机会：在每处这样使用 128 的地方都使用一个**#define** 语句（或者是使用类似于 **sizeof（bprm->buf）**的语句）代替；存在几个其它实例，我会让你把它们都找到。如果你实验一下，你就会发现在这种情况下**#define** 为什么比 **sizeof** 要好。（把这种重复出现的神奇数字加以定义和修正对于内核是更好的贡献。但是总体的修正工作要比看起来的困难，这只由于正确的对所有相关部分进行定位是很困难的；让我们一点一点地开始，最终会将其全部解决。）

search_binary_handler

二进制处理程序是 Linux 内核统一处理各种二进制格式的机制，这是我们需要的，因为不是所有的文件都是以相同的文件格式存储的。一个很合适的例子是 Java 的.class 文件。Java 定义了一种平台无关的二进制可执行格式——无论它们是在什么平台上运行，它们的文件本身都是相同的——因此这些文件显然应该和 Linux 特有的可执行格式一样构建。通过使用适当的二进制处理程序，Linux 可以把它们仿佛当作是自己特有的可执行文件一样处理。

后面我们会详细介绍二进制处理程序，但是现在你应该了解一些有关内容以便理解 **do_execve** 是如何发现匹配的。它把这一工作交给 **search_binary_handler**（9996 行）处理。

10037：开始遍历处理内核的二进制处理程序链接列表，依次将 **bprm** 传递给它们。（我们现在并不关心 **regs** 参数。）更确切的说，二进制处理程序的链接列表的每一个元素都包含一组指向函数的指针，这些函数一起提供了对一种二进制格式的支持。（13803 行定义的 **struct linux_binfmt** 结构显示了其中包含的内容：我们感兴趣的部分是装载二进制的部分 **load_binary**；装载共享库的部分 **load_shlib**；创建内核转储映象的部分 **core_dump**。）**search_binary_handler** 简单调用每一个 **load_binary** 函数，知道其中一个返回非负值指明它成功识别并装载了文件。**search_binary_handler** 返回负值指明发生的错误，其中包括不能找到匹配的二进制处理程序的错误。

10070：如果 10037 行开始的循环不能找到匹配的二进制处理程序，本行就试图装载新的二进制格式，它会引起第二次尝试，并应该取得成功。因此整个操作被包含在从 10036

行开始的两次执行的循环中。

可执行格式

正如前面一节中说明的一样，不是所有程序都使用相同的文件格式存储，Linux 使用二进制处理程序把它们之间的区别掩盖掉了。

Linux 当前“本地的”可执行格式（如果“本地”在系统中可以给各种格式提供良好支持）是可执行链接格式（ELF）。ELF 只是全部替换了原来的称为 a.out 的格式，替换之前的格式很难说是灵活的——除了有一些其它缺点以外，a.out 还很难适用于动态链接，这会使得共享库难于实现。Linux 仍然为 a.out 保留了一个二进制处理程序，但通常是使用 ELF。

二进制处理程序通过某种内嵌在文件开头的“magic 序列”（一个特殊字节序列）来识别文件，有时也会通过文件名的一些特性。例如，你会看到的 Java 处理程序可以保证文件名以.class 结尾并且前四个字节是（以十六进制）0xcafebabe，这是 Java 标准所定义的。

下面是 2.2 版本内核所提供的二进制处理程序（这是在我的 Intel 系统中的；Linux 的其它平台的移植移植版本，例如 PowerPC 和 SPARC 上，需要使用其它的处理程序）：

- a.out（在文件 fs/binfmt_aout.c 中）——这是为了支持原来风格的 Linux 二进制文件。这仍然是为了满足一些系统的向后兼容的需要，但是基本上 a.out 很快就会光荣退役了。
- EFL（在文件 fs/binfmt_elf.c 中）——只是为了支持现在新风格的 Linux 二进制文件。这在可执行文件和共享库中都广泛使用。最新的 Linux 系统（例如 Red Hat 5.2）一般只预装了 ELF 二进制文件，但是特殊情况下如果你决定装载 a.out 二进制文件，那么系统也可以对它提供支持。注意即使 ELF 被作为惯用的 Linux 本地格式，也要和其它格式一样使用二进制处理程序——内核并没有特殊的偏好。避免特殊情况的惯例能够简化内核代码。
- EM86（在文件 fs/binfmt_em86.c 中）——帮你在 Alpha 机器上运行 Intel 的 Linux 二进制文件，仿佛它们就是 Alpha 的本地二进制文件。
- Java（在文件 fs/binfmt_java.c 中）——使你可以不必每次都麻烦地定义 Java 字节码的解释程序就可以执行 Java 的.class 文件。这种机制和脚本中使用的机制类似；通过把.class 文件的文件名作为参数传递，处理程序返回来为你整型字节码处理程序。从用户的观点来看，Java 二进制文件是作为本地可执行文件处理的。在本章的后面内容中我们会详细介绍这个处理程序。
- Misc（在文件 fs/binfmt_misc.c 中）——这是最明智地使用二进制处理程序的方法，这个处理程序通过内嵌的特征数字或者文件名后缀可以识别出各种二进制格式——但是其最优秀的特性是它在运行期可以配置，而不是只能在编译器可以配置。因此，遵守这些限制，你就可以快速的增加对新二进制文件的支持，而不用重新编译内核，也无须重新启动机器。（这实在太棒了！）源程序文件中的注释建议最终使用它来取代 Java 和 M86 二进制处理程序。
- 脚本（在文件 fs/binfmt_script.c 中）——对于 shell 脚本，Perl 脚本等提供支持。宽松一点地说，所有前面两个字符是#!的可执行文件都规由这个二进制处理程序进行处理。

在上面这些二进制处理程序中，本书中只对 Java 和 ELF 处理程序进行了说明（分别从 9083 行和 7656 行开始），因为作为我们关心的基本内容，我们更关心内核如何处理各种不同格式间的区别，而不是每一种单个二进制处理程序的细节（虽然它自己也是一个很有趣的主题）。

一个例子：Java 二进制处理程序

如同前面你看到的一样，**do_execve** 遍历一个代表二进制处理程序的 **struct linux_binfmt** 结构的链接列表，调用每个结构的 **load_binary** 成员指向的函数直到其中一个成功（当然也或者到已经试验完了所有的格式为止）。但是这些结构又从何而来呢？函数 **load_binary** 是如何实现的？为了寻找这些答案，让我们来看一下 fs/binfmt_java.c 文件。

这个模块处理一些不是涉及在 Web 浏览器上使用 **java_format**（9236 行）执行的 Java 程序的 Java 二进制文件和相关的函数。它使用 **applet_format**（9254 行）及相关函数处理 Java 小程序（Applet）。在本节剩余部分的内容中，我们会集中看一下对于非 Java 小程序的支持；对于 Java 小程序的支持实际上是相同的。

如果重写 fs/binfmt_java.c 中的函数用来加强 Java 小程序函数和非 Java 小程序函数之间的相同代码的数量就更好了。虽然它注定最终要被“ misc ”二进制处理程序取代，但是现在还只是在讨论，尚未实行。

do_load_java

9108： 这是实际处理装载 Java 的.class 文件工作的函数。

9117： 通过检测特征数字 0xcafebabe 开始，这是因为 Java 标准规定所有有效的类文件都使用这个字符序列开始。接着开始执行健全性检测，一直到 9147 行，确保没有递归调用而且正在请求执行的可执行文件是以.class 结尾的。

9148： 此处，所有的健全性检测已经通过了。现在，**do_load_java** 取得文件的基本名字，将其和 Java 字节码解释程序一起放置到程序空间中，并试图执行 Java 字节码解释程序。

9165： 使用我们在 **do_execve** 中见到的同一个进程执行解释程序。特殊情况下，就像查询 **do_load_java** 的方法一样，使用 **search_binary_handler** 为解释程序查询二进制处理程序。（实际上，虽然它不一定非要是 ELF 二进制文件，但是它也可能是。）

记住其它处理程序不会分配新的 **struct task_struct** 结构——我们在使用 **fork** 的时候也碰到了这个问题。其它处理程序只是修改现存进程的 **struct task_struct** 结构。如果你希望细致地了解这是如何实现的，你的入手点应该是 **do_load_elf_binary**（8072 行）——我们关心的部分从 8273 行开始。

load_java

9226：**load_java** 是其它外部对象装载.class 文件时所使用的函数。它首先递增内核模块使用的计数（如果作为内核模块编译），随后又将其递减，但是实际的工作是由 **do_load_java**（9108 行）处理的。

java_format

9236： 通过比较 **java_format** 的初始化和 **struct linux_binfmt** 结构（13803 行）的定义，你可以看出这个模块没有提供对共享库和内核卸载的支持，只提供了对装载可执行程序的支持；而且这种支持是通过 **load_java** 函数实现的。

init_java_binfmt

9262： 指向这个模块的项是 **init_java_binfmt**，它把两个静态 **struct linux_binfmt** 结构 **java_format** 和 **applet_format** 的地址压入系统列表中。如果对 Java 二进制文件的支持被编译进了内核，就在 9355 行调用 **init_java_binfmt**，或者如果 Java 二进制文件

的支持被作为一个内核模块编译进了内核，就使用 kmod 任务。

调度：了解它们是如何运行的！

在应用程序被装载以后，必须获得对 CPU 的访问。这是调度程序涉及的领域。操作系统调度程序基本上划分为两类：

- 复杂调度程序——运行需要花费相当长的时间，但是希望可以全面提高系统性能。
- 快餐式（quick-and-dirty）调度程序——只是试图处理一些尽量简单的合理的工作就退出，从而进程本身将可以尽可能多的获得 CPU。

Linux 调度程序是后面一种情况。不要把“ quick-and-dirty ”解释成贬义的词，虽然实际的情况是：Linux 的调度程序在商业和自由领域中都从根本上痛击了其竞争者。

调度函数和调度策略

内核主要的调度函数经过仔细挑选使用 **schedule** 这个名字，该函数从 26686 行开始。这实际上是个很简单的函数，比它看起来还要简单，虽然由于它把三种调度策略合成了一种而其意义显得有些不是很明显。而且对于 SMP 的支持也增加了一定的复杂性，这一点将在第 10 章中详细讨论。

通常情况下使用的调度策略和进程有关。给定进程使用的调度算法称为调度策略，这在进程的 **struct task_struct** 结构的 **policy** 成员中有所反映。普通情况下，**policy** 是 **SCHED_OTHER**、**SCHED_FIFO**，或者 **SCHED_RR** 其中一个的位集。但是它也可能含有 **SCHED_YIED** 位集，如果进程决定交出 CPU——例如，通过调用 **sched_yield** 系统调用（请参看 **sched_yield**，27757 行）。

SCHED_XXX 常量在 16196 行到 16202 行宏定义。

16196：**SCHED_OTHER** 意味着传统 Unix 调度是使用它的——这不是一个实时进程。

16197：**SCHED_FIFO** 意味着这是一个实时进程，这要遵守 POSIX.1b 标准的 FIFO（先进先出）调度程序。它会一直运行，直到有一个进程在 I/O 阻塞，因而明确释放 CPU，或者是 CPU 被另一个具有更高 **rt_priority** 的实时进程抢占了。在 Linux 实现中，**SCHED_FIFO** 进程拥有时间片——只有当时间片结束时它们才被迫释放 CPU。因此，如同 POSIX.1b 中规定一样，这样的进程就像没有时间片一样运行。因此进程要保持对其时间片进行记录的这一事实主要是为了实现的方便，因此我们就不必使用 **if(!(current->policy & SCHED_FIFO)) { … }**来弄乱这些代码。还有，这样处理速度可能会快一些——其它实际可行的策略都需要记录时间片，并持续检测是否我们需要记录时间片会比简单的跟踪它速度更慢。

16198：**SCHED_RR** 意味着这是一个实时进程，要遵守 POSIX.1b 的 RR（循环：round-robin）调度规则。除了时间片有些不同之外，这和 **SCHED_FIFO** 类似。当 **SCHED_RR** 进程的时间片用完后，就使用相同的 **rt_priority** 跳转到 **SCHED_FIFO** 和 **SCHED_RR** 列表的最后。

16202：**SCHED_YIELD** 并不是一种调度策略，而是截取调度策略的一个附加位。如同前面说明的一样，如果有其它进程需要 CPU，它就提示调度程序释放 CPU。特别要注意的是这甚至会引起实时进程把 CPU 释放给非实时进程。

schedule

26689：**prev** 和 **next** 会被设置为 **schedule** 最感兴趣的两个进程：其中一个是在调用 **schedule** 时正在运行的进程（**prev**），另外一个应该是接着就给予 CPU 的进程（**next**）。记住 **prev** 和 **next** 可能是相同的——**schedule** 可以重新调度已经获得 CPU 的进程。

26706：如同第 6 章中介绍的一样，这就是中断处理程序的“ 下半部分 ”运行的地方。

26715：内核实时系统部分的实现，循环调度程序（**SCHED_RR**）通过移动“ 耗尽的 ”RR 进程——已经用完其时间片的进程——到队列末尾，这样具有相同优先级的其它 RR 进程就可以获得时间片了。同时这补充了耗尽进程的时间片。重要的是它并不是为 **SCHED_FIFO** 这样处理的，这样和预计的一样，后面的进程在其时间片偶然用完时就无须释放 CPU。

26720：由于代码的其它部分已经决定了进程必须被移进或移出 **TASK_RUNNING** 状态，所以会经常使用 **schedule**——例如，如果进程正在等待的硬件条件已经发生了——所以如果必要，这个 **switch** 会改变进程的状态。如果进程已经处于 **TASK_RUNNING** 状态，它就无须处理了。如果它是可以中断的（等待信号量）并且信号量到达了进程，就返回 **TASK_RUNNING** 状态。在所有其它情况下（例如，进程已经处于 **TASK_UNINTERUPTIBLE** 状态了），应该从运行队列中将进程移走。

26735：将 **p** 初始化为运行队列中的第一个任务；**p** 会遍历队列中的所有任务。

26736：c 记录了运行队列中所有进程的最好“ goodness ”——具有最好“ goodness ”的进程是最易获得 CPU 的进程。（我们很快就会讨论 **goodness**。）goodness 值越高越好，一个进程的 goodness 值永远不会为负——这是 Unix 用户经常见到的一种奇异情况，其中较高的优先级（通常称为较高“ niceness ”级）意味着进程会较少地获得 CPU 时间。（至少这在内核中是有意义的。）

26757：开始遍历执行任务列表，跟踪具有最好 goodness 的进程。注意只有在当前记录被破坏而不是当它简单地被约束时它才会改变最好进程的概念。因此，出于对队列中第一个进程的原因，这种约束就会被打破了。

26758：这个循环中只考虑了唯一一个可以调度的进程。**can_schedule** 宏的 SMP 版本在 26568 行定义；其定义使 SMP 内核只有任务尚未在 CPU 上运行才会把调度作为该 CPU 上的一个任务。（这样具有完美的意义——在几乎不必要的任务中造成混淆完全是一种浪费。）UP 版本在 26573 行，它总是真值——换而言之，在 UP 的情况下，运行队列中的每一个进程都需要竞争 CPU。

26767：值为 0 的 goodness 意味着进程已经用完它的时间片或者它已经明确说明要释放 CPU。如果所有运行队列中的所有进程都具有 0 值的 goodness，在循环结束后 **c** 的值就是 0。在这种情况下，**schedule** 要重新计算进程计数器；新计数器的值是原来值的一半加上进程的静态优先级——由于除非进程已经释放 CPU，否则原来计数器的值都是 0，**schedule** 通常只是把计数器重新初始化为静态优先级。（中断处理程序和由另外一个处理器引起的分支在 **schedule** 搜寻 goodness 最大值时都将增加此循环中的计数器，因此由于这个原因计数器可能不会为 0。虽然这有些罕见。）调度程序不必麻烦地重新计算现在哪一个进程具有最高的 goodness 值；它只是调度前面循环中遇到的第一个进程。此时，这个进程是它发现的第一个具有次高 goodness 值（0）的进程，因此 **schedule** 就能够计算出自己现在和以后所应该运行的任务。（记住，这就是“ quick-and-dirty ”的思想。）

26801：如果 **schedule** 已经选择了一个不同于前面正在运行的进程来调度，那么它就必须挂起原来的进程并允许新的进程运行。这是通过后面我们将介绍的 **switch_to** 处理的。**switch_to** 的一个重要结果对于应用程序开发者来说可能显得有些奇怪：对于

schedule 的调用并不返回。也就是它不是立即返回的；在系统条件判断语句返回到当前任务时调用就会返回。作为一个特殊情况，当任务退出而调用 **schedule** 时，对于 **schedule** 的调用从不会返回——因为内核不会返回已经退出的任务。还有另外一种特殊情况，如果 **schedule** 不会调度其它进程——也就是说，如果在 **schedule** 结束时 **next** 和 **prev** 是相同的——那么上下文中的跳转不会执行，**schedule** 实际上不会立即返回。

26809：**schedule** 末尾的**__schedule_tail** 和 **reacquire_kernel_lock** 函数在 UP 平台上不执行任何操作，因此现在我们就已经看完了调度程序的内核。顺便说一下，为了确保你已经正确的理解了这些代码，自己证明下面的性质：如果运行队列为空，那么下面就会调用 idle 任务。

switch_to

switch_to 处理从一个进程到下一个进程的跳转，称为上下文跳转（context-switching）；这是在不同处理器上会不同处理之间进行的低级特性。有趣的是，在 x86 平台上内核开发人员使用软件处理大多数的上下文跳转，这样就忽略了一些硬件的支持。这种机制背后的原因在**__switch_to** 函数（2638 行）上面的标题注释中有所说明，这个函数和 **switch_to** 宏（12939 行）一起处理上下文跳转。

由于很多上下文跳转要依赖于对内核处理内存方式的正确理解，这在下一章中才会详细介绍，本章只是稍微涉及一点。上下文跳转背后的基本思想是记忆当前位置和将要到达的位置——这是我们必须保存的当前上下文——接着跳转到另外一个前面已经存储过了的上下文。通过使用一部分汇编代码，**switch_to** 宏保存了后面将要介绍的上下文的两个重要的部分。

12945：首先，**switch_to** 宏保存 ESP 寄存器的内容，它指向进程的当前堆栈。堆栈在下一章中将深入介绍；现在你只需要简单了解堆栈中保存的局部变量和函数调用信息。**switch_to** 宏也保存 EIP 寄存器的内容，这是进程的当前指令指针——如果允许继续运行时所执行的为下一条指令的地址。

12948：把 **next->tss.eip**——保存指令的指针——压入返回堆栈，记录当后面紧跟的跳转到**__switch_to** 的 **jmp** 返回时的返回地址。这样做的最终结果是当**__switch_to** 返回时，我们又回到了新的进程。

12949：调用**__switch_to**（2638 行），它完成段寄存器和页表的保存和恢复工作。在你阅读完第 8 章以后这些特征数字就更有意义了。

12955：**tss** 代表 *task-state* 段，这是 Intel 使用的支持硬件上下文跳转的 CPU 特性的术语。虽然内核代码使用软件实现上下文跳转，但是开发人员仍然会使用 TSS 来记录进程的状态。**struct task_struct** 结构的 **tss** 成员的类型是 **struct thread_struct** 结构，本书中为了节省空间，忽略了它的定义。其成员仅仅对应于 x86 的 TSS——成员是为 EIP 和 ESP 而存在的，如此而已。

计算 goodness 值

进程的 goodness 值通过 **goodness** 函数（26388 行）计算。**goodness** 返回下面两类中的一个值：1,000 以下或者 1,000 以上。1,000 和 1,000 以上的值只能赋给“ 实时 ”进程，从 0 到 999 的值只能赋给“ 普通 ”进程。实际上普通进程的 goodness 值只使用了这个范围底部的一部分，从 0 到 41（或者对于 SMP 来说是 0 到 56，因为 SMP 模式会优先照顾等待同一个处理器的进程）。无论是在 SMP 还是在 UP 上，实时进程的 goodness 值的范围都是从 1,001

到 1,099。

有关这两类 goodness 结果的重要的一点是该值在实时系统中的范围肯定会比非实时系统的范围要高（因此偏移量（offset）是 100 而不是 1000）。POSIX..1b 规定内核要确保在实时进程和非实时进程同时竞争 CPU 时，实时进程要优先于非实时进程。由于调度程序总是选择具有最大 goodness 值的进程，又由于任何尚未释放 CPU 的实时进程的 goodness 值总是比非实时进程的 goodness 大，Linux 对这一点的遵守是很容易得到证明的。

尽管在 **goodness** 上面的标题注释中有所说明，该函数还是从不会返回-1,000 的，也不会返回其它的负值。由于 idle 进程的 **counter** 值为负，所以如果使用 idle 进程作为参数调用 **goodness**，就会返回负值，但这是不会发生的。

goodness 只是一个简单的函数，但是它是 Linux 调度程序必不可少的部分。运行对立中的每个进程每次执行 **schedule** 时都可能调用它，因此其执行速度必须很快。但是如果一旦它调度失误，那么整个系统都要遭殃了。考虑到这些冲突压力，我想改进现有的系统是相当困难的。

goodness

26394：如果进程已经释放了 CPU，就返回 0（在清除 **SCHED_YIELD** 位之后，这是因为进程只可能有一次想释放 CPU，现在它已经的确把 CPU 释放了）。

26402：如果这是一个实时进程，**goodness** 返回的值就属于数值较高的一类；这要精确地依赖于 **rt_priority** 的值。

26411：此处，代码识别出这是一个非实时进程，它把 goodness（在这个函数中被称为 **weight**）初始化为其当前的 **counter** 值，这样如果进程已经占用 CPU 一段时间了，或者进程开始的优先级比较低，那么进程就不太可能获得 CPU。

26412：如果权值 **weight** 的值为 0，那么进程的计数器就已经被用完了，因此 **goodness** 就不会再增加加权因素。其它进程就可以有机会运行。

26418：尽力优先考虑等待同一个处理器的进程（只在 SMP 系统中是这样——顺便说一下，考虑一下运行在一个双处理器的系统中的三个进程的实现情况）。

26423：给相关的当前进程或者当前线程增加了一些优点；这有助于合理使用缓存以避免使用昂贵的 MMU 上下文跳转。

26425：增加进程的 **priority**。这样，**goodness**（和其它类似的调度程序）就对较高优先级的进程比对较低优先级的进程更感兴趣，即使在前面进程已经部分用完了它们的时间片也是这样。

26428：返回计算出来的 goodness 值。

非实时优先级

每个 Linux 进程都有一个优先级，这是从 1 到 40 的一个整数，其值存储在 **struct task_struct** 结构的 **priority** 成员中。（对于实时进程，在 **struct task_struct** 结构中还会使用一个成员——**rt_priority** 成员。随后很快就会对它进行更详细的讨论。）它的范围使用 **PRIO_MIN**（在 16094 行宏定义为-20）和 **PRIO_MAX**（在 16095 行宏定义为 20）限定——理论上来说，的确是这样。但是非常令人气恼的是，控制优先级的函数——**sys_setpriority** 和 **sys_nice**——并没有注意到这些明显的常量，却相反宁愿使用一些固定的值。（它们也使用最大的完美值 19，而不是 20。）基于这个原因，**PRIO_MIN** 和 **PRIO_MAX** 两个常量并没有广泛使用。不过这又是一个热心读者改进代码的机会。

由于已经在文档中说明 **sys_nice**（27562 行）为要废弃不用了——可能会使用

sys_setpriority 来重新实现——我们就忽略前面一个函数，只讨论后面一个。

sys_setpriority

29213：**sys_setpriority** 使用三个参数——**which**，**who** 和 **niceval**。**which** 和 **who** 参数提供了一种可以用来指定一个给定用户所拥有的单个进程，一组进程或者所有进程的方法。**who** 要根据 **which** 的值做出不同的解释；它会作为一个进程 ID，进程组 ID 或者用户 ID 读取。

29220：这是确保 **which** 有效地进行健全性检测。我认为这里的模糊不清是不必要的。如果我们不使用

```
if   ( which > 2 || which > 0 )
```

而使用如下语句

```
if   ( which != PRIO_PROCESS && wich != PRIO_PGRP && which != PRIO_USER )
```

或者至少是

```
if   ( which > PRIO_USER || which < PRIO_PGRP )
```

另外，在 29270 行也可以使用同样的方法。

29226：**niceval** 是使用用户术语定义的——也就是说，它是在从-20 到 19 的范围中，而不是象内核中使用的一样，在从 1 到 40 的范围中。如同变量名说明的一样，这是一个完美的值，但不是一个优先级。因此，为了实现这种转化，**sys_setpriority** 应该跳过一些循环，同时要截断 **niceval** 超出允许范围的值。

我承认自己被这段代码的复杂性所困扰着。使用实际上使用的 **DEF_PRIORITY** 的值——20——以下的简化代码显然可以实现相同的效果：

```
if   ( niceval <-19 )
  priority = 40;
else if   ( niceval > 19 )
  priority = 1;
else
  priority = 20 - niceval;
```

在保持比 **sys_setpriority** 中的代码简单的同时，我的实现方法中当然也可以用于处理 **DEF_PRIORITY**。因此，或者我严重误解了一些内容，或者就象我提出的代码本身，它根本就不需要这么复杂。

29241：循环遍历系统的任务列表中的所有任务，执行它可以允许修改。**proc_sel**（29190 行）说明了给定的进程是否对所提供的 **which** 和 **who** 值满意，可以用它来选择进程；由于 **sys_getpriority** 也要使用这个函数，所以它也是 **sys_setpriority** 应该考虑的一个因素。

对于读取和设置单个进程优先级的普通情况（如果没有其它问题，就通过提早退出 **for_each_task** 循环），**sys_setpriority** 和 **sys_getpriority**（29274 行开始的代码和此处有相似的内部循环）都对它有一点加速作用。**sys_setpriority** 可能不会很频繁地被调用，但是 **sys_getpriority** 却可能被很频繁调用，因而这样努力的是值得的。

update_process_times

sys_setpriority 只会影响进程的 **priority** 成员——也就是其静态优先级。回忆一下进程也是具有动态优先级的，这由 **counter** 成员表示，这一点我们在对 **schedule** 和 **goodness** 的讨论中就已经清楚地看到了。我们已经可以看出在调度程序发现 **counter** 值为 0 时，**schedule** 会周期性地根据其静态优先级重新计算每一个进程的动态优先级。但是我们仍然还没有看到另外一部分困扰我们的问题：**counter** 是在哪里被递减的？它是怎样达到 0 的？

对于 UP，答案与 **update_process_times**（27382 行）有关。（和前面一样，我们把对于 SMP 问题的讨论延迟到第 10 章。）**update_process_times** 是作为 **update_time**（27412 行）的一部分被调用的，它还是第 6 章中讨论的定时器中断的一部分。作为一个结果，它被相当频繁地调用——每秒钟 100 次。（当然，这只是对人类的内力来说是相当频繁的，对于 CPU 来说这实在是很慢的。）在每一次调用的时候，它都会把当前进程的 **counter** 值减少从上次以来经过的“ 滴嗒 ”的数目（百分之一秒——请参看第 6 章）。通常，这只是一次跳动，但是如果内核正忙于处理中断，那么内核就可能会忽略定时器的跳动。当计数器减小到 0 以下时，**update_process_times** 就增加 **need_resched** 标志，说明这个进程需要重新调度。

现在，由于进程缺省的优先级（使用内核优先级的术语，而不使用用户空间的完美值）是 20，缺省情况下进程得到一个 21 次跳动的时间片。（的确这是 21 次跳动，而不是 20 次跳动，因为进程直到其动态优先级减少到 0 以下时才会为重新调度做出标记。）一次跳动是百分之一秒，或者是 10 微秒，因此缺省的时间片就是 210 微秒——大约是五分之一秒——在 16466 行有确切的描述。

我发现这个结果十分奇怪，因为原来以为理想的反应迅速的系统应该具有小很多的时间片——实际上我对这一点认识是如此强烈以至于开始的时候我还以为文档的说明发生了错误。但是，回顾一下，我觉得自己也不应该奇怪。毕竟，进程不会频繁地耗尽其整个时间片，因为它们经常都会因为 I/O 的原因而阻塞。在几个进程都绑定在 CPU 上时，在它们之间太频繁地跳转是没有必要的。（特别是在诸如 x86 之类的 CPU 上，这里的上下文跳转的代价是相当高的。）最后，我必须承认我从来没有注意到自己留意 Linux 逻辑单元的响应的迟缓特性，因此我觉得 210 微秒的时间片是个不错的选择——即使这在最初的时候看起来是太长了。

如果由于某些原因你需要时间片比当前最大值还长（410 微秒，优先级上长到了 40），你可以简单使用 **SCHED_FIFO** 调度策略，在你准备好以后就可以释放 CPU（或者重新编写 **sys_setpriority** 和 **sys_nice**）。

实时优先级

Linux 的实时进程增加了一级优先级。实时优先级保存在 **struct task_struct** 结构的 **rt_priority** 成员中，它是一个从 0 到 99 的整数。（值 0 意味着进程不是实时进程，在这种情况下其 **policy** 成员必须是 **SCHED_OTHER**。）

实时任务仍然使用相同的 **counter** 成员作为它们的非实时的计数器部分。实时任务为了某些目的甚至使用与非实时任务使用的 **priority** 成员相同的部分，这是当时间片用完时用来补充 **counter** 值使用的值。为了清晰起见，**rt_priority** 只是用来对实时进程划分等级以对它们进行区分——否则它们的处理方式就和非实时进程相同了。

进程的 **rt_priority** 被设定为使用 POSIX.1b 规定的函数 **sched_setscheduler** 和 **sched_setparam**（通常只有 root 才可以调用这两个函数，这一点我们在讨论权能时会看到）设置其调度策略。注意这意味着如果具有修改权限，进程的调度策略在进程生命期结束以后就可以改变。

实现这些 POSIX 函数的系统调用 **sys_sched_setscheduler**（27688 行）和 **sys_sched_setparam**（27694 行）都会把实际的工作交给 **setschedular**（27618 行）处理，这个函数我们现在就介绍。

setscheduler

27618：这个函数的三个参数是目标进程 **pid**（0 意味着当前进程），新的调度策略 **policy**，

和包含附加信息的一个结构 **param**——它记录了 **rt_priority** 的新值。

27630：在一些健全性检测之后，**setscheduler** 从用户空间中得到提供的 **struct sched_param** 结构的备份。在 16204 行定义的 **struct sched_param** 结构只有一个成员 **sched_priority**，它就是调用者为目标进程设计的 ***rt_priority***。

27639：使用 **find_process_by_pid**（27608 行）找到目标进程，如果 **pid** 是 0，这个函数就返回一个指向当前任务的指针；如果存在指向具有给定 PID 进程，就返回指向该进程的指针；或者如果不存在具有这个 PID 的进程，就返回 **NULL**。

27645：如果 **policy** 参数为负时，就保留当前的调度策略。否则，如果这是个有效值，那么现在就可以将其接收。

27657：确保优先级没有越界。这是通过使用一点小技巧来加强的。该行只是第一步，它被用来确保所提供的值没有大得超出了范围。

27659：现在已经确知新的实时优先级位于 0 到 99 的范围之内。如果 **policy** 是 **SCHED_OTHER**，但是新的实时优先级不是 0，那么这个测试就失败了。如果 **policy** 指明了一个实时调度程序但是新的实时优先级是 0（如果这里它不是 0，就应该是从 1 到 99），测试也会失败。否则，测试就能成功。这虽然并不是很易读，但它确实是正确的、最小的，（我想）速度也很快。我不确定这里我们是否对速度有所苛求，但是——到底一个进程需要多长时间需要设置它的调度程序？下面的代码就应该具有更好的可读性，而且当然也不会太慢：

P492 1

27663：不是每一个进程都可以设置自己的调度策略和其它进程的调度策略。如果所有进程都可以设置自己的调度策略，那么任何进程都可以简单地设置自己的调度策略为 **SCHED_FIFO** 并进入一个无限循环来抢占 CPU，这样必然会锁定系统。显然，是不能够允许这种做法的。因此，只有进程拥有这样处理的权能时，**setscheduer** 才会允许进程设置自己的调度策略。权能在下一节中将比较详细地介绍。

27666：在相同的行中，我们不希望别人可以修改其它用户进程的调度策略；普通情况下，只允许你修改你自己所有的进程的调度策略。因此，**setscheduer** 要确保或者用户是设置自己所有的进程的处理程序或者具有修改其它进程的调度策略的权能。

27672：这里才是 **setscheduler** 实际工作的地方，它在目标进程的 **struct task_struct** 结构中设置 **policy** 和 **rt_priority**。如果该进程在运行队列中（这通过检测其 **next_run** 成员非空来测试），就将它移到运行队列的顶部——这比较容易令人感到迷惑；可能这有助于 **SCHED_FIFO** 进程实现对 CPU 的抢占。进程为重新调度做出标记，**setscheduer** 清空并退出。

遵守限制

内核经常需要决定是否允许进程执行某个操作。进程可能被简单的禁止执行某些操作，但却被允许在受限的环境中执行一些别的操作；这些操作基本上可以由权能表示，并且/或者可以从用户 ID 和组 ID 中推导出来。在其它期间，允许进程处理一些操作，但只是在受限的环境中——例如，它对 CPU 的使用必须受到限制。

权能

在前面一节中，你已经看到了一个检测权能的例子——实际上是有两次相同的权能。这是 **CAP_SYS_NICE** 权能（14104 行），它决定是否应该允许进程设置优先级（完美级别）

或调度策略。由于这比仅仅的完美级别要更适用，**CAP_SYS_NICE** 是一个误用的位——虽然很容易就可以看出设置调度策略和相关的概念是紧密相关的，而且你一般也不会要一个权能而不要另外一个权能。

每一个进程都有三个权能，它们被存储在进程的 **struct task_struct** 结构中（在 16400 行到 16401 行中）：

- **cap_effective**——有效置位集合
- **cap_permitted**——允许位集合
- **cap_inheritable**——继承位集合

进程权能的有效位集合是当前可以处理的内容的集合；这是通过广泛使用的 **capable** 函数检测的集合，这个函数在 16738 行定义。

允许位集合规定进程正常地可以被赋予的权能。这个集合通常不会增加——只有一种情况例外：如果一个进程具有 **CAP_SETPCAP** 权能，那么它就可以将自己的允许位集合中的任何权能赋给其它进程，即使目标进程还没有拥有这个权能。

如果一个权能在允许位集合中，但是并不在有效位集合中，那么进程现在还没有马上拥有权能，但是它可以通过请求权能而获得。为什么要麻烦地区别它们呢？在本章开始我们第一次讨论权能的时候，我们简单地考虑了一个简单的例子：一个长期运行的进程只是偶然需要权能，而不是所有情况下都需要。为了保证进程不会偶然缺少权能，进程可以一直等待，直到它需要权能，接着请求权能，执行有权限的操作，并再次取消权能。这种方法比较安全。

继承位集合不像你想象的那么简单。它不是祖先继承在执行 **fork** 的同时传递的权能集合——实际上，在创建的那一刻（也就是紧随着 **fork**），子孙进程的权能的三个集合和其祖先的三个权能集合都是相同的。相反，继承位集合在 **exec** 运行期间才会起作用。进程在调用 **exec** 之前的继承位集合有助于决定它的允许位集合和继承位集合，它们在 **exec** 执行结束以后也会保留下来——仔细的介绍请参看 **compute_creds**（9948 行）。注意在 **exec** 之后权能是否保留要部分依赖于进程的继承位集合；它还要部分依赖于文件本身中的权能位集合（或者不管怎样，这至少是一个计划——虽然这种特性还没有完全实现）。

顺便提一下，注意到允许位集合必须总是有效位集合和继承位集合的超集（superset）（或者和有效位集合相同）。（只有对于有效位集合这才是严格正确的。一个进程可能会扩展另外一个进程的继承位集合从而它不再是其允许位集合的子集，但是就我知道的来说，这是无意义的，因此我们从现在就开始忽略这种可能性。）然而，和你可能希望的相反，有效位集合不一定要是继承位集合的超集（或者和继承位集合相同）。也就是说，在 **exec** 结束以后，进程可能会拥有一个以前不曾有过的权能（虽然这个权能必须在其允许位集合中——也就是说，这是一个原来进程自己可能已经得到了的权能）。我认为这种需要只是局部的，这样进程就不需要暂时获得不需要的权能，而能够获得足以执行 **exec** 程序的权能。

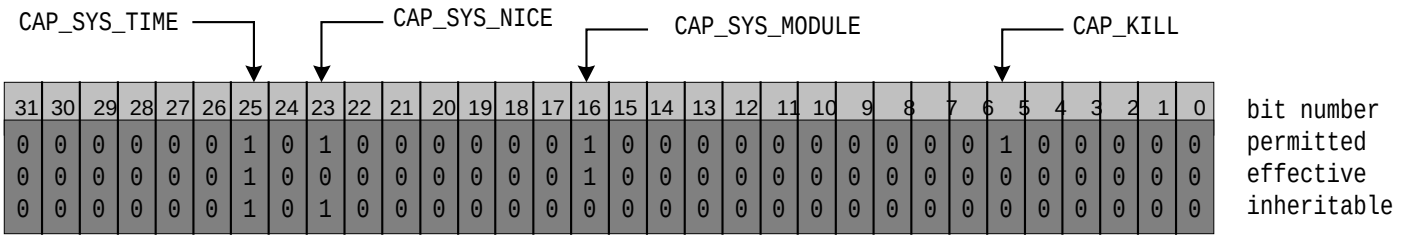

图 7.4 权能集

图 7.4 说明了各种可能性。它显示了一个理想进程的三种权能集合，位从左到右计数。

允许进程可以获得 **CAP_KILL** 权能，这样就允许它不考虑其它属主而杀掉别的进程，但是它还没有立即拥有权限，而且也不会在 **exec** 执行过程中自动获得。目前它具有增加和删除内核模块的权能（使用 **CAP_SYS_MODULE**），但是同样也不会在 **exec** 执行过程中自动获得。它可以获得 **CAP_SYS_NICE** 权能，但是直到 **exec** 执行完后才会获得（假定文件权能位允许）。最后，它可以立即修改系统时间（**CAP_SYS_TIME**），但是也是只有通过 **exec** 才能获得这个权能。除非其它具有 **CAP_SETPCAP** 权能的进程提供了这个权能，否则这个进程不能获得这个权能，它可能执行的其它进程也不可能获得这个权能。

保证这些不同性质的代码主要是在 kernel/capability.c 中，从 22460 行开始。两个主要的函数是读取权能的函数 **sys_capget**（22480 行）和设置权能的函数 **sys_capset**（22592 行）；它们在下一节中讨论。通过 **exec** 继承的权能使用 fs/exec.c 的 **compute_creds**（9948 行）处理，这一点已经介绍过了。

当然，root 肯定拥有所有的权能。内核权能特性给 root 提供了一种规则的方法来有所选择地只把需要的权能赋给特定的进程，而不用考虑该进程是否作为 root 用户运行。

权能一个有趣的特性是它们可以用来改变系统的“风格”。作为一个简单的例子，为所有的进程设置 **CAP_SYS_NICE** 权能会使所有进程都增加自己的优先级（并设置它们的调度规则，等等）。如果你修改了系统中每一个进程的运行方式，那么你就改变了系统本身。自己设想一下发明一种新的可以通过更令人兴奋的方式修补系统的内核权能。

权能的尚未为人所知的优点是它们使源程序代码非常清晰。当检测当前进程是否允许设置系统时间时，却反而要检测当前进程是否以 root 运行，这种方式看起来似乎有些不很好。权能使我们可以了解它们的意思。权能的存在甚至还能够使查询进程的用户 ID 或组 ID 的代码更为清晰，这是因为这样的处理代码对这个问题的答案比较感兴趣，而是对从其中可以推导出的结论更感兴趣。否则，代码应该已经使用权能查询它需要了解的内容了。由于权能更加一致地和 Linux 内核代码结合起来，这种特性就变得更加可靠了。

13916：内核可以识别的权能从这里开始。因为这些宏定义的解释已非常详细了，我们就不再详细介绍其中每一个的内容了。

14153：赋给每一个权能的数字是简单的连续整数，但是由于要使用无符号整数中的位来编址，所以就使用 **CAP_TO_MASK** 宏把它们转化为 2 的幂。

14154：设置和检测权能的核心只是一系列位操作；从这里到 include/linux/capability.h 中定义了用来使位操作更为清晰的宏和内联函数。

sys_capget

22480：**sys_capget** 有两个参数：**header** 和 **dataptr**。**header** 是 **cap_user_header_t** 类型（13878 行）的，它是一个指向定义权能使用的版本和目标进程的 PID 的结构的指针；**dataptr** 是 **cap_user_data_t** 类型（13884 行）的，它也是一个指向结构类型的指针——这个结构包含有效位、允许位和继承位集合。**sys_capget** 通过第二个指针返回信息。

22492：在版本不匹配的情况下，**sys_capget** 通过 **header** 指针返回使用的版本，接着返回 **EINVAL** 错误（或者如果它不能把版本信息拷贝到调用者的空间中就返回 **EFAULT**）。

22509：定义调用者希望了解其权能的进程；如果 **pid** 不是 0，也不是当前进程的 PID，**sys_capget** 就要查询它。

22520：如果它能装载目标进程，它就把自己的权能拷贝到临时变量 **data** 中。

22530：如果所有工作到目前为止都运行良好，它就把权能拷贝回用户空间中由 **dataptr** 参数提供的地址中。然后，它返回 **error** 变量——通常如果一切运行良好，这就是 0；否则就是一个错误号。

sys_capset

22592：**sys_capset** 的参数几乎和 **sys_capget** 的参数类似。不同之处是 **data**（不再称为 **dataptr** 了）是常量。

22600：和 **sys_capget** 一样，**sys_capset** 确保内核和调用进程使用一致的权能系统的版本。如果版本不一致，就拒绝尝试请求。

22613：如果 **pid** 不是 0，就说明调用者希望设置其它进程的权能，在大多数情况下这种尝试都会遭到拒绝。如果调用者具有 **CAP_SETPCAP** 权能，这意味着允许它设置任何进程的权能，**sys_capset** 就允许这种尝试。这种测试的前面部分有些太受限制了：如果它和当前进程的 **pid** 相等，就接收这个 **pid**。

22616：从用户空间中拷贝新的权能，如果失败就返回错误。

22627：和 22509 行开始的 **sys_capget** 代码类似，**sys_capset** 定义了调用者希望了解其权能的进程。这就是两者的区别所在，**sys_capset** 为了说明进程组（或者是-1 指明是所有进程）也允许其 **pid** 值为负。在这种情况下，**target** 仍然设置为 **current**，因此当前进程的权能要在后面的计算中使用。

22642：现在它必须保证合法地使用新的权能位集合，而且在内部保持一致。除非这种新特性在调用者的允许位集合中，否则这种测试会验证出新进程的继承位集合没有包含任何新鲜的东西。因此，它不会放弃调用者尚未拥有的任何权能。

22650：类似地，**sys_capset** 也要确保除非调用者的允许位中包含新的特性，否则目标进程的允许位集合也不会包含尚未具有的特性。因此，它也不会放弃调用者尚未拥有的任何权能。

22658：回想一下进程的有效位集合必须是其允许位集合的一个子集。这种性质在这里得到了保证。

22666：**sys_capset** 现在已经准备对请求做出修改。负的 **pid** 值意味着它正在给不止一个进程修改权能——如果 **pid** 是-1，就是所有的进程；如果 **pid** 是其它的负值，就是一个进程组中的所有进程。在这些情况下，实际工作分别由 **cap_set_all**（22561 行）和 **cap_set_pg**（22539 行）完成；这只是通过一些适当的进程集合循环，按照和单个进程相同的方法覆盖掉集合中的每一个进程的权能位集合。

22676：如果 **pid** 是正数（或者是 0，表示当前进程），权能位集合只赋给目标进程。

用户 ID 和组 ID

尽管权能功能强大、十分有用，但它并不是你实现访问控制的唯一武器。在一些情况中，我们需要了解哪个用户正在运行一个进程，或者进程是作为哪个用户来运行。用户使用整型的用户 ID 来区别，一个用户可以属于一个组或者多个组，每一个都有自己特有的整型 ID。

有两种风格的用户 ID 和组 ID：实际的 ID 和有效的 ID。一般说来，实际用户（或组）ID 为你说明了哪个用户创建了进程，有效用户（或组）ID 为你说明在情况改变时进程作为哪个用户运行。由于访问控制的决定要更多依赖于进程作为哪儿用户运行，而不是哪个用户创建了这个进程，因此内核会比检测实际用户（和组）ID 更加频繁地检测有效用户（或）ID——在我们现在关心的代码中就是这样处理的。**struct task_struct** 结构中的相关成员是 **uid**，**euid**，**gid**，和 **egid**（16396 行到 16397 行）。注意用户 ID 和用户名不同，前者是一个整数，而后者是一个字符串。/etc/passwd 文件把这两者关联起来。

让我们再回到 **sys_setpriority** 并看一下前面我们忽略了的从 29244 行到 29245 行的一些代码。**sys_setpriority** 通常执行的操作都是让用户降低自己进程的优先级，但是不能降低其它用户进程的优先级——除非用户具有 **CAP_SYS_NICE** 权能。因此，if 表达式的前面两个

术语要检测目标进程的用户 ID 是否和 **sys_setpriority** 的调用者的实际用户 ID 或者有效用户 ID 匹配。如果两个都不匹配，并且 **SYS_CAP_NICE** 没有设置，**sys_setpriority** 就正确地拒绝这种尝试。

如果允许，进程可以使用 **sys_setuid** 和 **sys_setgid**（29578 行和 29445 行）和其它一些函数修改它们的用户 ID 和组 ID。用户 ID 和组 ID 也可以通过执行可执行的 setuid 或 setgid 可执行程序进行修改。

资源限制

可以要求内核限制一个进程使用系统中的各种资源，包括内存和 CPU 时间。这可以通过 **sys_setrlimit** 实现（30057 行）。通过浏览 **struct rusage** 结构（16068 行）你对支持限制就可以有一个基本的概念。进程特有的限制在 **struct task_struct** 结构中记录——还可能在什么地方？请参看 16404 行的 **rlim** 数组成员。

违反限制的结果根据限制的不同也会有所不同。例如，对于 **RLIMIT_MPROC**（在本书的源程序代码中没有包括）——有关一个用户可以拥有的进程数目的限制——和你在 23974 行中看到的一样，结果仅仅和 **fork** 失败一样。超出其它限制的后果对于一些进程可能比较严重，这样进程会被杀死（请参看 27333 行）。进程可以使用 **sys_getrlimit**（30046 行）请求特殊限制，或者使用 **sys_getrusage**（30143 行）请求资源使用限制。

在 30067 行中，注意进程可以随意减少自己的资源限制，但是它增加自己的资源限制时只能增加到一个最大值，这个值可以根据每一个资源限制进行具体设置。因此，当前的资源限制和所有的资源限制是分别记录的（使用在 16089 行定义的 **struct rlimit** 结构的 **rlin_cur** 成员和 **rlim_max** 成员）。然而具有 **CAP_SYS_RESOURCE** 权能的进程可以覆盖这个最大值。

这和优先级的规则不同：允许进程可以减小自己的优先级，但是为增加其优先级需要特殊许可，即使是它减少了自己的优先级接着又要马上增加它也是如此。当前资源限制和最大资源限制这两个相互关联的概念并没有反映在内核优先级的调度中。还有，注意到一个进程可以改变另一个进程的优先级（当然是假定它有权这样处理），但是一个进程只能修改自己的资源限制。

所有美好的事物都会结束——这就是它们如何处理的

我们已经看到进程是如何生成的，怎样给它们赋予各自的生存周期。现在我们应该看一下它们是如何消亡的。

exit

同第 6 章中介绍的一样，你可以通过给进程发送信号量 9 强行杀掉进程，但是更普通的情况是进程自动退出。进程通过调用系统调用 **exit** 自动退出，它在内核中是由 **sys_exit** 实现的（23322 行）。（顺便说一下，当 C 程序从它的 **main** 部分返回时，就会潜在调用 **exit**。）当进程退出时，内核释放所有分配给这个进程的资源——内存、文件，等等——当然，还要停止给它继续使用 CPU 的机会。

然而内核不能立即回收代表进程的 **struct task_struct** 结构，这是因为该进程的祖先必须能够使用 **wait** 系统调用查询其子孙进程的退出状态。**wait** 返回它检测出的死亡状态的进程的 PID，因此如果死亡的子孙进程在祖先进程仍在等待时就已经重新分配了，那么应用程序

就会被搞乱（和其它问题一样，同一个祖先结束时可以有两个具有相同 PID 的子孙进程——一个进程是活动的，另一个进程是死亡的——祖先进程也不知道哪一个已经退出了）。因此，内核必须保留死亡子孙进程的 PID 直到 **wait** 发生为止——这通过完整地保持其 **struct task_struct** 结构来自动实现的；分配 PID 的代码就不用再查询它在任务列表中发现的进程是否是活动的。

处于这种在两种状态之间的进程——它既不是活动的，也没有真正死亡——被称为僵进程（zombies）。那么 **sys_exit** 的任务就是把活动进程转化为僵进程。

sys_exit 本身的工作很少；它只是简单地把现存退出代码转化为 **do_exit** 希望的格式，接着就会调用 **do_exit**，由它来处理实际的工作。（**do_exit** 也会作为发送信号量的一部分来调用，这一点我们在第 6 章中已经讨论过了。）

23267：**do_exit** 把退出代码作为参数处理，在其返回类型之前使用特殊符号 **NORET_TYPE**。虽然现在 **NORET_TYPE**（14955 行）定义为空——因此它也就不起作用——但是原来它经常被定义为**__volatile__**，用来提示 gcc 该函数不会返回。了解了这一点知识，gcc 就执行一些额外的优化工作并取消有关函数不能成功返回的警告信息。使用其新的定义，**NORET_TYPE** 对于编译器就没有用处了，但是它仍然给我们人类传递了很多有用的信息。

23285：释放它的信号量和其它 System V IPC 结构，这一点我们将在第 9 章中介绍。

23286：释放分配给它的内存，这一点我们在第 8 章中介绍。

23290：释放分配给它的文件，很快就会讨论。

23291：释放它的文件系统数据，它超出了本书的范围。

23292：释放它的信号量处理程序表，这一点我们在第 6 章中介绍过了。

23294：剩下的任务是进入 **TASK_ZOMBIE** 状态，其退出代码被记录下来以供将来祖先进程使用。

23296：调用 **exit_notify**（23198 行），它会警告当前退出任务的祖先进程和其进程组中的所有成员该进程正在退出。

23304：调用 **schedule**（26686 行）释放 CPU。这个对于 **schedule** 的调用从来不会返回，这是因为它跳转到下一个进程的上下文，从来不会再跳转回来，因此这是现在退出的进程的最后一次拥有 CPU 的机会。

__exit_files

进程如何和文件交互不是本书的主题。但是我们应该快速浏览一下**__exit_files**（23109 行），因为这样会有助于我们理解**__clone** 函数，这个函数使祖先进程和子孙进程可以共享特定的信息。祖先进程和子孙进程可以共享的一种信息是它们打开的文件列表。和当时说明的一样，Linux 使用引用计数器规则来保证进程退出之后可以正确地处理扫尾工作。这里就有个扫尾工作的很好的例子。

23115：假设进程已经打开了文件（几乎总会是这样的），**__exit_files** 会递减原来存储在 **tsk->files->count** 中的引用计数器。诸如 **atomic_dec_and_test** 之类的原子操作将在第 10 章详细介绍；知道 **atomic_dec_and_test**（10249 行）递减其参数值并当参数新值是 0 时返回真值就足够了。因此，如果 **tsk** 的对于目标 **struct files_struct** 结构的引用是最后一次时，这就是正确的。（如果这是一个私有拷贝，没有和其它任何进程共享，那么引用计数器的初始值就是 1，当然它被减小为 0。）

23116：在释放记录进程的打开文件的内存之前，必须把这些文件都关闭，这是通过调用 **close_files**（23081 行）实现的。

23118：释放保留进程的文件描述符数组 **fd** 的内存，这个数组是 **files** 的一个子域。打开文件（**NR_OPEN**，在 15067 行中定义 1,024）的最大数量要加以选择，这样本行中的

if 测试就能正确——**fd** 数组必须刚好适合一个内存页的大小。这样做可以使得内存的分配（或释放）速度快许多；否则，**__exit_files** 只好使用更通用但是速度却慢得多的内核的内存函数了。下一章会加深你对这种决策的理解。

23122：最后，**__exit_files** 释放 **files** 本身。

其它**__exit_xxx** 函数背后的概念是类似的：它们减少了任务自有的对于潜在共享信息的引用计数器，如果这是最后一次引用，它们要负责执行所有必须的工作来将其清除。

wait

和 **exec** 一样，**wait** 是一组函数，而不是一个函数。（但是和 **exec** 不同，**wait** 家族的函数实际包含一个名为 **wait** 的函数。）**wait** 家族中的其它函数最终都是使用一个系统调用 **sys_wait4**（23327 行）实现的，这个系统调用的名字反映出它实现了 **wait** 家族中最通用的函数 **wait4**。标准 C 库 libc 的实现必须重新组织对于其它 **wait** 函数调用的参数并调用 **sys_wait4**。（这还不是问题的全部：由于历史的原因，内核到 Alpha 的移植也会提供 **sys_waitpid**。但是即使是 **sys_waitpid** 也会反过来调用 **sys_wait4**。）

除了处理一些其它内容，**sys_wait4**——也只有 **sys_wait4**——最终把僵进程送进坟墓。然而从应用程序的观点来看，**wait** 和相关函数要检测子孙进程的状态：检测是否有进程死亡了，如果有，到底是哪一个进程，这个进程是怎样死亡的。

sys_wait4

23327：为了适合作为相当通用的一个函数，**sys_wait4** 有很多参数，其中一些是可选的。和通常情况一样，**pid** 是目标进程的 PID；和你看到的一样，0 和负值是特殊的。如果 **stat_addr** 非空，那么它就是所得子孙进程的退出状态应该拷贝到的地址。**options** 是一些可能定义 **sys_wait4** 的操作的标志的集合。如果 **ru** 非空，那么它就是所获得的子孙进程资源使用信息所应该拷贝到的地址。

23335：如果提供了无效选项，**sys_wait4** 就返回错误代码。这种决定看起来有点荒唐；我们可以简单忽略一些无关选项。当然，这样处理所需要的参数，如果调用者设置了自己不想设置的位，那么希望的操作是不要执行——在任何情况下，这都意味着调用者不能正确理解，在这种情况下发送一个失败信号量要比简单地忽略调用者的这种困惑要更多。

23342：循环遍历该进程的直接子进程（但不包括其孙进程，曾孙进程，等等）。如同本章中前面说明的一样，进程的最年轻（最近创建的）子孙进程通过 **struct task_struct** 结构的 **p_cptr** 成员是可访问的，这个最年轻进程原来的兄弟进程通过其 **p_osptr** 成员也是可以访问的；因此，**sys_wait4** 从这个最年轻子孙进程开始遍历其祖先的所有子孙进程，并逐渐遍历其原来的兄弟进程。

23343：根据 **pid** 参数的值筛选出不匹配的 PID。注意值为-1 的 **pid** 参数是如何潜在的对进程进行选择的，正如我们所期望的：**pid** 值在 23343，23346 和 23349 行中的测试没有成功，因此它就不会遭到拒绝。这样，系统需要对每一个子孙进程进行考虑。

23376：这就是我们现在感兴趣的情况——祖先进程正在等待一个已经结束了的进程。这是最后实际上得到僵进程的地方。它通过更新子孙进程使用的进程的用户时间和系统时间部分开始（这通过 29772 行的 **sys_times** 系统调用实现），因为子孙进程不会再参与计算了。

23382：其它资源使用信息被收集起来（如果要求这样处理），子孙进程的退出状态被传递到特定的地址中（同样，如果要求这样处理）。

23387：设置 **retval** 为当前得到的死亡子孙进程的 PID。这就是最后的结果；**retval** 不会再

改变了。

23388：如果这个垂死进程的当前祖先进程不是原来的祖先进程，那么进程就会离开进程图表中的当前位置（通过 **REMOVE_LINKS**，16876 行），在其原始祖先的控制下重新安装自己（通过 **SET_LINKS**，16887 行），接着给其祖先进程发送 **SIGCHLD** 信号量，这样祖先进程就知道其子孙进程已经退出了。这种通知是通过 **notify_parent**（28548 行，在第 6 章中介绍）传递的。

23396：否则——正常情况——最后可以调用 **release**（22951 行）释放所得子孙进程的 **struct task_struct** 结构。（在看完 **sys_wait4** 以后，我们马上就会看 **release**。）

23400：现在已经成功获取了子孙进程，因此 **sys_wait4** 只需要返回成功信息就完美地完成了工作；它跳转到 23418 行，从这儿返回 **retval**（所获得子孙进程的 PID）。

23401：注意特殊的流程控制；**default** 的情况需要继续执行从 23342 行开始的 **for** 循环。因为只有既没有停止运行也不是僵进程的进程才会执行到 **default** 的情况，所以这种流程控制是正确的，但是初次阅读时比较容易误解。而且，无论如何这也有些多余；没有它循环也一样能处理。

23406：如果代码能运行到此处，**for** 循环就可以完整地运行下来——正在调用的进程遍历执行其子孙进程没有发现匹配的整个列表——计算的结果是三种状态中的一种。或者由于该任务没有和所提供的 **pid** 参数匹配的子孙进程，因而还没有进程退出，或者（是前面情况的一个特例）该任务根本就没有子孙进程。

23408：如果 **flag** 不为 0，在 **for** 循环中就可以执行到 23358 行，这说明至少有一个进程和所提供的 **pid** 参数匹配——它不是僵进程，也没有被终止，因此它就不能被获取。在这种情况下，如果提供了 **WNOHANG** 选项——这意味着如果不能获取子孙进程，那么调用者就不会等待——它向前跳转到最后，返回 0。

23411：如果有信号量被接收，就退出并返回一个错误。这个信号量不是 **SIGCHLD**——如果它是 **SIGCHLD**，就应该已经发现了死亡的进程，因此就不可能执行到此处。

23413：否则，一切都没有问题；调用者只需要等待一个子孙进程退出。因此，进程的状态被设置为 **TASK_INTERRUPTIBLE** 并调用 **schedule** 释放 CPU 给另一个进程使用。正在等待的进程直到再次获得 CPU 时才会返回，同时要再次检测死亡子孙进程（通过向回跳转到 23339 行的 **repeat** 标号）。回想一下处于 **TASK_INTERRUPTIBLE** 状态的进程要等待信号量将其唤醒——在这种情况下，它特别希望 **SIGCHLD** 来指明子孙进程已经退出了，但是任何信号量都可以到达。

23417：**flag** 是 0，因为或者进程没有子孙进程，或者所提供的 **pid** 参数不能和它的任何子孙进程匹配——不管怎样，**sys_wait4** 都给调用者返回一个 **ECHILD** 错误。

release

22951：**release** 的唯一一个参数是指向要释放的 **struct task_struct** 结构的指针。

22953：确保该任务没有试图释放自身——这是会在内核中引起逻辑错误的一种无意义的情况。

22969：UP 代码实际上是通过调用 **free_uid**（23532 行）开始的，它用来释放潜在共享的 **struct user_struct** 结构，这个结构除了其它功能以外，还要帮助 **fork** 确保不会出现单个用户影响所有进程的情况。

22970：减小系统关于正在运行的任务总数的计数并释放 **tarray_freelist** 中的僵死进程的时间片。

22974：僵死进程的 PID 也会释放，并且使用 **REMOVE_LINKS**（16876 行）解除它同进程表和任务列表的关联。注意，由于内核数据结构在此处正在做出修正，**task** 数组中的进程项并不需要被设置为 **NULL**；把它的空槽增加到自由列表中就足够了。

22979：僵死进程有关次要页表错误，主要页表错误的总数以及向外交换所使用的时间的数量被增加到当前进程对应的 “ 子孙进程计数 ” 中——这是正确的；**release** 只能通过 **sys_wait4** 调用，这样只允许进程释放自己的子孙进程。因此，当前进程必须是僵死进程的祖先。

22982：最后，应该回收垂死进程的 **struct task_struct** 结构，这可以通过对 **free_task_struct** 的调用（2391 行）来实现。这个函数简单地回收存储在这个结构中的内存页。现在，进程最终功德圆满的寿终正寝了。