介 绍

《Linux内核注释》旨在给程序员和学生提供比以前更详细和更易理解的Linux内核代码注释。作者分析了核心代码，并对重要的函数、系统调用和数据结构提供了大量的注释。

对《注释》系列丛书的写作灵感都来源于John Lions所著的大量流行的《Lions' Commentary on Unix》一书。无数的计算机专业的学生在复制和使用这本书。这本书对AT&T的Unix操作系统的早期版本的内幕进行了深刻的剖析。

《Liunx内核注释》同样提供了对流行的功能强大的Liunx操作系统的结构和函数实现的内幕介绍。本书的主要目标是：

1.提供一个最新的和完整的服务器版本的完整源代码。（这本书分析的版本是2.2.5版,也是写这本书时发布的最新版本。）

2.提供一个对每个子系统功能的一般性概述。

3.研究各个子系统主要的函数和数据结构。

4.对开发者应怎样通过修改源代码来改进和扩展内核提出建议。

本书的最后一项目标—定制--是你学习内核代码的最有说服力的原因。通过理解内核是怎样工作的，你能够编写自己的代码用以在你的操作系统中实现所需要的功能。如果允许其他人共享你的改进，你的代码甚至会在官方发行的内核代码中出现，被全世界数百万计的人们所使用。

开放源代码是指让开发者研究源代码并实现功能性扩展。Linux是全世界成长最快的操作系统,开放源代码是其主要的原因之一。从玩游戏，到网上冲浪，到为大大小小的ISP们提供稳定的Web服务器平台以至解决最庞大的科学难题，Linux都能胜任全部工作。它之所以能如此强大是因为有像你一样的开发者在研究、学习并且扩充这个系统。

你能从本书中学到什么

这本书集中解释了Linux内核源代码的核心中专用代码行是如何运行的。你将学习到内核最内部的子系统是怎样构造和这种构造能够实现系统功能的理由。

本书的第一部分以易于阅读和交叉引用的格式复制了一个经过筛选的linux 内核源代码的子集。在这本书稍后的注释中，无论一行代码在何处被引用，你都会在这一行前面发现一个小箭头。这个箭头指出了对此行进行注释处的页号。

源代码后是这本书的第二部分，即注释部分，注释部分对源代码进行了讨论。注释部分的每一章讨论了一个不同的内核子系统，或者是其它的功能性逻辑组件，例如系统调用或内存管理。注释部分大量的行号引用为你指明了所讨论代码行的确切行号。

在本书正文后的附录部分,简洁地覆盖了自本书主要部分完成以后内核的变化。在附录中还包含了被内核用做软件许可证的完整的GNU常规公众许可证。最后,本书为你提供了一个索引。通过该索引你可以查询术语或主题。这将让你更快更有效的使用这本参考工具书。

本书的使用对象

本书假设你能阅读C语言的代码, 不怕偶尔读一些汇编语言代码。并且你想知道一个快速的、坚固的、可靠的、健壮的、现代的、实用的操作系统是如何工作的。一些读者也许是这样的程序员, 他们想为前进中的Linux内核发展工作提供他们自己的改进和添加内容。

如何使用本书

用最适合你自己的方法放松地去看这本《linux 内核注释》。因为写这本书的目的是为提供一个参考资料，你不必从头看到尾。因为注释和代码是一一对应的，你可以从另外一个方向接近内核。

欢迎你对我的第一本书提出意见。你可以通过e-mail和我联系。地址是：lckc@ScottMaxwell.org。 勘误表、更新和其它一些有用信息可以通过访问 http://www.ScottMaxwell.org/lckc.html 得到。

第1章 Linux 简介

让用户很详细地了解现有操作系统的实际工作方式是不可能的，因为大多数操作系统的源代码都是严格保密的。其例外是一些研究用的系统，另外一些是明确为操作系统教学而设计的系统。（还有一些系统则是同时出于这两种目的。）尽管研究和教学这两个目的都很好，但是这类系统很少能够通过对正式操作系统的小部分实现来体现操作系统的实际功能。对于操作系统的一些特殊问题，这种折衷系统所能够表现的就更是少得可怜了。

在以实际使用为目标的操作系统中，让任何人都可以自由获取系统源代码，无论目的是要了解、学习还是改进，这样的现实系统并不多。本书的主题就是这些少数操作系统中的一个：Linux。

Linux 的工作方式类似于 Uinx，是免费的，源代码也是开放的，符合标准规范的 32 位（在 64 位 CPU 上是 64 位）操作系统。Linux 拥有现代操作系统的所具有的内容，例如：

- 真正的抢先式多任务处理，支持多用户
- 内存保护
- 虚拟内存
- 支持对称多处理机 SMP（symmetric multiprocessing），即多个 CPU 机器，以及通常的单 CPU（UP）机器
- 符合 POSIX 标准
- 联网
- 图形用户接口和桌面环境（实际上桌面环境并不只一个）
- 速度和稳定性

严格说来，Linux 并不是一个完整的操作系统。当我们在安装通常所说的 Linux 时，我们实际安装的是很多工具的集合。这些工具协同工作以组成一个功能强大的实用系统。Linux 本身只是这个操作系统的内核，是操作系统的心脏、灵魂、指挥中心。（整个系统应该称为 GNU/Linux，其原因在本章的后续内容中将会给以介绍。）内核以独占的方式执行最底层任务，保证系统正常运行——协调多个并发进程，管理进程使用的内存，使它们相互之间不产生冲突，满足进程访问磁盘的请求等等。

在本书中，我们给大家揭示的就是 Linux 是如何完成这一具有挑战性的工作的。

Linux（和 Unix）的简明历史

为了让大家对本书所讨论的内容有更清楚的了解，让我们先来简要回顾一下 Linux 的历史。由于 Linux 是在 Unix 的基础上发展而来的，我们的话题就从 Unix 开始。

Unix 是由 AT&T 贝尔实验室的 Ken Thompson 和 Dennis Ritchie 于 1969 年在一台已经废弃了的 PDP-7 上开发的；它最初是一个用汇编语言写成的单用户操作系统。不久，Thompson 和 Ritchie 成功地说服管理部门为他们购买更新的机器，以便该开发小组可以实现一个文本处理系统，Unix 就在 PDP-11 上用 C 语言重新编写（发明 C 语言的部分目的就在于此）。它果真变成了一个文本处理系统——不久之后。只不过问题是他们先实现了一个操作系统而已…

最终，他们实现了该文本处理工具，而且 Unix（以及 Unix 上运行的工具）也在 AT&T 得到广泛应用。在 1973 年，Thompson 和 Ritchie 在一个操作系统会议上就这个系统发表了一篇论文，该论文引起了学术界对 Unix 系统的极大兴趣。

由于 1956 年反托拉斯法案的限制，AT&T 不能涉足计算机业务，但允许它可以以象征性的费用发售该系统。就这样，Unix 被广泛发布，首先是学术科研用户，后来又扩展到政府和商业用户。

伯克利（Berkeley）的加州大学是学术用户中的一个。在这里 Unix 得到了计算机系统研究小组（CSRG）的广泛应用。并且在这里所进行的修改引发了 Unix 的一大系列，这就是广为人知的伯克利软件开发（BSD）Unix。除了 AT&T 所提供的 Unix 系列之外，BSD 是最有影响力的 Unix 系列。BSD 在 Unix 中增加了很多显著特性，例如 TCP/IP 网络，更好的用户文件系统（UFS），工作控制，并且改进了 AT&T 的内存管理代码。

多年以来，BSD 版本的 Unix 一直在学术环境中占据主导地位，但最终发展成为 System V 版本的 AT&T 的 Unix 则成为商业领域的主宰。从某种程度上来说，这是有社会原因的：学校倾向于使用非正式但通常更好用的 BSD 风格的 Unix，而商业界则倾向于从 AT&T 获取 Unix。

在用户需求驱动和用户编程改进特性的促进下，BSD 风格的 Unix 一般要比 AT&T 的 Unix 更具有创新性，而且改进也更为迅速。但是，在 AT&T 发布最后一个正式版本 System V Release 4（SVR4）时，System V Unix 已经吸收了 BSD 的大多数重要的优点，并且还增加了一些自己的优势。这种现象的部分原因在于从 1984 年开始，AT&T 逐渐可以将 Unix 商业化，而伯克利 Unix 的开发工作在 1993 年 BSD4.4 版本完成以后就逐渐收缩以至终止了。然而，BSD 的进一步改进由外界开发者延续下来，到今天还在继续进行。正在进行的 Unix 系列开发中至少有四个独立的版本是直接起源于 BSD4.4，这还不包括几个厂商的 Unix 版本，例如惠普的 HP-UX，都是部分地或者全部地基于 BSD 而发展起来的。

实际上 Unix 的变种并不止 BSD 和 System V。由于 Unix 主要使用 C 语言来编写，这就使得它相对比较容易地移植到新的机器上，它的简单性也使其相对比较容易重新设计与开发。Unix 的这些特点大受商业界硬件供应商的欢迎，比如 Sun、SGI、惠普、IBM、DEC（数字设备公司）、Amdahl 等等；IBM 还不止一次对 Unix 进行了再开发。厂商们设计开发出新的硬件并简单地将 Unix 移植到新的硬件上，这样新的硬件一经发布便具备一定的功能。经过一段时间之后，这些厂商都拥有了自己的专有 Unix 版本。而且为了占有市场，这些版本故意以不同的侧重点发布出来以更好的占有用户。

版本混乱的状态促进了标准化工作的进行。其中最主要的就是 POSIX 系列标准，它定义了一套标准的操作系统接口和工具。从理论上说，POSIX 标准代码很容易移植到任何遵守 POSIX 标准的操作系统中，而且严格的 POSIX 测试已经把这种理论上的可移植性转化为现实。直到今天，几乎所有的正式操作系统都以支持 POSIX 标准为目标。

现在让我们回顾一下，在 1984 年，杰出的电脑黑客 Richard Stallman 独立开发出一个类 Unix 的操作系统，该操作系统具有完全的内核、开发工具和终端用户应用程序。在 GNU（“GNU’s Not Unix”首字母的缩写）计划的配合下，Stallman 开发这个产品有自己的技术理想：他想开发出一个质量高而且自由的操作系统。Stallman 使用了“自由”（free）这个词，不仅意味着用户可以免费的获取软件；而且更重要的是，它将意味着某种程度的“解放”：用户可以自由使用、拷贝、查询、重用、修改甚至是分发这份软件，完全没有软件使用协议的限制。这也正是 Stallman 创建自由软件基金会（FSF）资助 GNU 软件开发的本意（FSF 也在资助其它科研方面的开发工作）。

15 年以来，GNU 工程已经吸收、产生了大量的程序，这不仅包括 Emacs，gcc（GNU 的 C 编译器），bash（shell 命令），还有大部分 Linux 用户所熟知的许多应用程序。现在正在进行开发的项目是 GNU Hurd 内核，这是 GNU 操作系统的最后一个主要部件（实际上 Hurd 内核早已能够使用了，不过当前的版本号为 0.3 的系统在什么时候能够完成，还是未知数）。

尽管 Linux 大受欢迎，但是 Hurd 内核还在继续开发。这种情况的原因有几个方面，其

一是 Hurd 的体系结构十分清晰的体现了 Stallman 关于操作系统工作方式的思想，例如，在运行期间，任何用户都可以部分的改变或替换 Hurd（这种替换不是对每个用户都是可见的，而是只对申请修改的用户可见，而且还必须符合安全规范）。另一个原因是据介绍 Hurd 对于多处理器的支持比 Linux 本身的内核要好。还有一个简单的原因是兴趣的驱动，因为程序员们希望能够自由地进行自己所喜欢的工作。只要有人希望为 Hurd 工作，Hurd 的开发就不会停止。如果他们能够如愿以偿，Hurd 有朝一日将成为 Linux 的强劲对手。不过在今天，Linux 还是自由内核王国里无可争议的主宰。

在 GNU 发展的中期，也就是 1991 年，一个名叫 Linus Torvalds 的芬兰大学生想要了解 Intel 的新 CPU——80386。他认为比较好的学习方法是自己编写一个操作系统的内核。出于这种目的，加上他对当时 Unix 变种版本对于 80386 类机器的脆弱支持十分不满，他决定要开发出一个全功能的、支持 POSIX 标准的、类 Unix 的操作系统内核，该系统吸收了 BSD 和 System V 的优点，同时摒弃了它们的缺点。Linus（虽然我知道我应该称他为 Torvalds，但是所有人都称他为 Linus）独立把这个内核开发到 0.02 版，这个版本已经可以运行 gcc，bash 和很少的一些应用程序。这些就是他开始的全部工作了。后来，他又开始在因特网络上寻求广泛的帮助。

不到三年，Linus 的 Unix—Linux—已经升级到 1.0 版本。它的源代码量也呈指数形式增长，实现了基本的 TCP/IP 功能（网络部分的代码后来重写过，而且还可能会再次重写）。此时 Linux 就已经拥有大约 10 万用户了。

现在的 Linux 内核由 150 多万行代码组成，Linux 也已经拥有了大约 1000 万用户（由于 Linux 可以自由获取和拷贝，获取具体的统计数字是不可能的）。Linux 内核 GNU/Linux 附同 GNU 工具已经占据了 Unix 50%的市场。一些公司正在把内核和一些应用程序同安装软件打包在一起，生产出 Linux 的 distribution（发行版本），这些公司包括 Red Hat 和 Calera prominent 公司。现在的 GNU/Linux 已经备受注目，得到了诸如 Sun、IBM、SGI 等公司的广泛支持。SGI 最近决定在其基于 Intel 的 Merced 的系列机器上不再搭载自己的 Unix 变种版本 IRIX，而是直接采用 GNU/Linux；Linux 甚至被指定为 Amiga 将要发布的新操作系统的基础。

GNU 通用公共许可证

这样一个如此流行大受欢迎的操作系统当然值得我们学习。按照通用公共许可证(GPL，（General Public License）)的规定，Linux 的源代码可以自由获取，这使得我们学习该系统的强烈愿望得以实现。GPL 这份非同寻常的软件许可证，充分体现了上面提到的 Stallman 的思想：只要用户所做的修改是同等自由的，用户可以自由地使用、拷贝、查询、重用、修改甚至重新发布这个软件。通过这种方式，GPL 保证了 Linux（以及同一许可证保证下的大量其它软件）不仅现在自由可用，而且以后经过任何修改之后都仍然可以自由使用。

请注意这里的自由并不是说没有人靠这个软件盈利，有一些日益兴起的公司，比如发行最流行的 Linux 发行版本的 Red Hat，就是一个例子。（Red Hat 自从面世以来，市值已经突破数十亿美元，每年盈利数十万美元，而且这些数字还在不断增长）。但是任何人都不能限制其它用户涉足本软件领域，而且所作的修改不能减少其自由程度。

本书的附录 B 中收录有 GNU 通用公共许可证协议的全文。

Linux 开发过程

如上所述，由于 Linux 是一款自由软件，它可以免费获取以供学习研究。Linux 之所以值得学习研究，是因为它是相当优秀的操作系统。如果 Linux 操作系统相当糟糕，那它就根本不值得被我们使用，也就没有必要去研究相关的书籍。（除非一种可能，为了追求刺激）。Linux 是一款十分优秀的操作系统还在于几个相互关联的原因。

Linux 优秀的原因之一在于它是基于天才的思想开发而成的。在学生时代就开始推动整个系统开发的 Linus Torvads 是一个天才，他的才能不仅展现在编程能力方面，而且组织技巧也相当杰出。Linux 的内核是由世界上一些最优秀的程序员开发并不断完善的，他们通过 Internet 相互协作，开发理想的操作系统；他们享受着工作中的乐趣，而且也获得了充分的自豪感。

Linux 优秀的另外一个原因在于它是基于一组优秀的概念。Unix 是一个简单却非常优秀的模型。在 Linux 创建之前，Unix 已经有 20 年的发展历史。Linux 从 Unix 的各个流派中不断吸取成功经验，模仿 Unix 的优点，抛弃 Unix 的缺点。这样做的结果是 Linux 成为了 Unix 系列中的佼佼者：高速、健壮、完整，而且抛弃了历史包袱。

然而，Linux 最强大的生命力还在于其公开的开发过程。每个人都可以自由获取内核源程序，每个人都可以对源程序加以修改，而后他人也可以自由获取你修改后的源程序。如果你发现了缺陷（bug），你可以对它进行修正，而不用去乞求不知名的公司来为你修正。如果你有什么最优化或者新特点的创意，你也可以直接在系统中增加功能，而不用向操作系统供应商解释你的想法，指望他们将来会增加相应的功能。当发现一个安全漏洞后，你可以通过编程来弥补这个漏洞，而不用关闭系统直到你的供应商为你提供修补程序。由于你拥有直接访问源代码的能力，你也可以直接阅读代码来寻找缺陷，或是效率不高的代码，或是安全漏洞，以防患于未然。

除非你是一个程序员，否则这一点听起来仿佛没有多少吸引力。实际上即使你不是程序员，这种开发模型也将使你受益匪浅，这主要体现在以下两个方面：

- 可以间接受益于世界各地成千上万的程序员随时进行的改进工作。
- 如果你需要对系统进行修改，你可以雇用程序员为你完成工作。这部分人将根据你的需求定义单独为你服务。可以设想，这在源程序不公开的操作系统中它将是什么样子。

Linux 这种独特的自由流畅的开发模型已被命名为 bazaar（集市模型），它是相对于 cathedral（教堂）模型而言的。在 cathedral 模型中，源程序代码被锁定在一个保密的小范围内。只有开发者（很多情况下是市场）认为能够发行一个新版本，这个新版本才会被推向市场。这些术语在 Eric S. Raymond 的 The Cathedral and the Bazaar 一文中有所介绍，大家可以在 **http://www.tuxedo.org/~esr/writings/**找到这篇文章。Bazaar 开发模型通过重视实验，征集并充分利用早期的反馈，对巨大数量的脑力资源进行平衡配置，可以开发出更优秀的软件。（顺便说一下，虽然 Linux 是最为明显的使用 bazaar 开发模型的例子，但是它却远不是第一个使用这个模型的系统。）

为了确保这些无序的开发过程能够有序地进行，Linux 采用了双树系统。一个树是稳定树（stable tree），另一个树是非稳定树（unstable tree）或者开发树（development tree）。一些新特性、实验性改进等都将首先在开发树中进行。如果在开发树中所做的改进也可以应用于稳定树，那么在开发树中经过测试以后，在稳定树中将进行相同的改进。按照 Linus 的观点，一旦开发树经过了足够的发展，开发树就会成为新的稳定树，如此周而复始的进行下去。

源程序版本号的形式为 x.y.z。对于稳定树来说，y 是偶数；对于开发树来说，y 比相应

的稳定树大一（因此，是奇数）。截至到本书截稿时，最新的稳定内核版本号是 2.2.10，最新的开发内核的版本号是 2.3.12。对 2.3 树的缺陷修正会回溯影响（back-propagated）2.2 树，而当 2.3 树足够成熟的时候会发展成为 2.4.0。（顺便说一下，这种开发会比常规惯例要快，因为每一版本所包含的改变比以前更少了，内核开发人员只需花很短的时间就能够完成一个实验开发周期。）

http://www.kernel.org 及其镜像站点提供了最新的可供下载的内核版本，而且同时包括稳定和开发版本。如果你愿意的话，不需要很长时间，这些站点所提供的最新版本中就可能包含了你的一部分源程序代码。

第2章 代码初识

本章首先从较高层次介绍 Linux 内核源程序的概况，这些都是大家关心的一些基本特点。随后将简要介绍一些实际代码。最后以如何编译内核来检验个人所进行的修改的讨论来作为本章的收尾。

Linux 内核源程序的部分特点

在过去的一段时期，Linux 内核同时使用 C 语言和汇编语言实现的。这两种语言需要一定的平衡：C 语言编写的代码移植性较好、易于维护，而汇编语言编写的程序则速度较快。一般只有在速度是关键因素或者一些因平台相关特性而产生的特殊要求（例如直接和内存管理硬件进行通讯）时才使用汇编语言。

正如同实际中所做的，即使内核并未使用 C++的对象特性，部分内核也可以在 g++（GNU 的 C++编译器）下进行编译。同其它面向对象的编程语言相比较，相对而言 C++的开销是较低的，但是对于内核开发人员来说，这已经足够甚至太多了。

内核开发人员不断发展编程风格，形成了 Linux 代码独有的特色。本节将讨论其中的一些问题。

gcc 特性的使用

Linux 内核被设计为必须使用 GNU 的 C 编译器 gcc 来编译，而不是任何一种 C 编译器都可以使用。内核代码有时要使用 gcc 特性，伴随着本书的进程，我们将陆续介绍其中的一部分。

一些 gcc 特有代码只是简单地使用 gcc 语言扩展，例如允许在 C（不只是 C++）中使用 **inline** 关键字指示内联函数。也就是说，代码中被调用的函数在每次函数调用时都会被扩充，因而就可以节约实际函数调用的开销。

更为普遍的情况是代码的编写方式比较复杂。因为对于某些类型的输入，gcc 能够产生比其它输入效率更高的执行代码。从理论上讲，编译器可以优化具有相同功能的两种对等的方法，并且得到相同的结果。因此，代码的编写方式是无关紧要的。但在实际上，用一些方法编写所产生的代码要比用其它方法编写所产生的代码的执行速度快得多。内核开发人员清楚如何才能产生更高效的执行代码的方法，而且这种知识也不断在他们编写的代码中反映出来。

例如，考虑内核中经常使用的 **goto** 语句——为了提高速度，内核中经常大量使用这种一般要避免使用的语句。在本书中所包含的不到 40,000 行代码中，一共有 500 多条 **goto** 语句，大约是每 80 行一个。除汇编文件外，精确的统计数字是接近每 72 行一个 **goto** 语句。公平的说，这是选择偏向的结果：比例如此高的原因之一是本书中涉及的是内核源程序的核心，在这里速度比其它因素都需要优先考虑。整个内核的比例大概是每 260 行一个 **goto** 语句。然而，这仍然是我不再使用 Basic 进行编程以来见过的使用 **goto** 频率最高的地方。

代码必需受特定编译器限制的特性不仅与普通应用程序的开发有很大不同，而且也不同于大多数内核的开发。大多数的开发人员使用 C 语言编写代码来保持较高的可移植性，即使在编写操作系统时也是如此。这样做的优点是显而易见的，最为重要的一点是一旦出现更

好的编译器，程序员们可以随时进行更换。

内核对于gcc特性的完全依赖使得内核向新的编译器上的移植工作更加困难。最近Linus对这一问题在有关内核的邮件列表上表明了自己的观点。“记住，编译器只是一个工具。”这是对依赖于 gcc 特性的一个很好的基本思想的表述：编译器只是为了完成工作。如果通过遵守标准还不能达到工作要求，那么就不是工作要求有问题，而是对于标准的依赖有问题。

在大多数情况下，这种观点是不能够被人所接受的。通常情况下，为了保证和程序语言标准的一致，开发人员可能需要牺牲某些特性、速度或者其它相关因素。其它的选择可能会为后期开发造成很大的麻烦。

但是，在这种特定的情况下，Linus 是正确的。Linux 内核是一个特例，因为其执行速度要比向其它编译器的可移植性远为重要。如果设计目标是编写一个可移植性好而不要求快速运行的内核，或者是编写一个任何人都可以使用自己喜欢的编译器进行编译的内核，那么结论就可能会有所不同了；而这些恰好不是 Linux 的设计目标。实际上，gcc 几乎可以为所有能够运行 Linux 的 CPU 生成代码，因此，对于 gcc 的依赖并不是可移植性的严重障碍。

在第 3 章中我们将对内核设计目标进行详细说明。

内核代码习惯用语

内核代码中使用了一些显著的习惯用语，本节将介绍常用的几个。当你通读源程序代码时，真正重要的问题是并不在这些习惯用语本身，而是这种类型的习惯用语的确存在，而且是不断被使用和发展的。如果你需要编写内核代码，你应该注意到内核中所使用的习惯用语，并把这些习惯用语应用到你的代码中。当通读本书（或者代码）时，注意你还能找到多少习惯用语。

为了讨论这些习惯用语，我们首先需要对它们进行命名。为了便于讨论，笔者创造了这些名字。而在实际中，大家不一定非要参考这些用语，它们只是对内核工作方式的描述而已。

一个普通的习惯用语笔者称之为“资源获取”（resource acquisition idiom）。在这个用语中，一个函数必须实现一系列资源的获取，包括内存、锁等等（这些资源的类型未必相同）。只有成功地获取当前所需要的资源之后，才能处理后面的资源请求。最后，该函数还必须释放所有已经获取的资源，而不必对没有获取的资源进行考虑。

我采用“错误变量”这一用语（error variable idiom）来辅助说明资源获取用语，它使用一个临时变量来记录函数的期望返回值。当然，相当多的函数都能实现这个功能。但是错误变量的不同点在于它通常是用来处理由于速度的因素而变得非常复杂的流程控制中的问题。错误变量有两个典型的值，0（表示成功）和负数（表示有错）。

这两个用语结合使用，我们就可以十分自然地得到符合模式的代码如下：

```
Int f (void)
{
   int err;
   resource *r1, *r2;
   err = -ERR1      /*assume failure*/
   r1=acquire_ resource();
   if (!r1)            /*not aquired*/
       goto out          /*returns    -ERR1*/

   Got resource r1,try for r2.*/
```

```
    err = - ERR2;
    r2 = acquire_ resource2();
    if (!r2)            /*not aquired*/
        goto out1           /*returns –ERR2*/

    /*have both r1 and r2.*/
    err = 0;

    /* … use r1 and r2 … */

out2:
    release_resource(r2)

out2:
    release_resource(r2)

out:
        return err;
}
```

（注意变量 **err** 是使用错误变量的一个明确实例，同样，诸如 **out** 之类的标号则指明了资源获取用语的使用。）

如果执行到标号 **out2**，则都已经获取了 **r1** 和 **r2** 资源，而且也都需要进行释放。如果执行到标号 **out1**（不管是顺序执行还是使用 **goto** 语句进行跳转到），则 **r2** 资源是无效的（也可能刚被释放），但是 **r1** 资源却是有效的，而且必需在此将其释放。同理，如果标号 **out** 能被执行，则 **r1** 和 **r2** 资源都无效，**err** 所返回的是错误或成功标志。

在这个简单的例子中，对于 **err** 的一些赋值是没有必要的。在实践中，实际代码必须遵守这种模式。这样做的原因主要在于同一行中可能包含有多种测试，而这些测试应该返回相同的错误代码，因此对错误变量统一赋值要比多次赋值更为简单。虽然在这个例子中对于这种属性的必要性并不非常迫切，但是我还是倾向于保留这种特点。有关的实际应用可以参考 **sys_shmctl**（第 21654 行），在第 9 章中还将详细介绍这个例子。

减少#if 和#ifdef 的使用

现在的 Linux 内核已经移植到不同的平台上，但是我们还必须解决移植过程中所出现的问题。大部分支持各种不同平台的代码由于包含许多预处理代码现都已变得非常不规范，例如：

```
#if defined(SOLARIS)
/* … do things the solaris way … */
#elif defined(HPUX)
/* … do things the HP-UX way … */
#elif defined(LINUX)
/* … do things the right way … */
```

#endif

这个例子试图实现操作系统的可移植性，虽然 Linux 关注的焦点很明显是实现代码在各种 CPU 上的可移植性，但是二者的基本原理是一致的。对于这类问题来说，预处理器是一种错误的解决方式。这些杂乱的问题使得代码晦涩难懂。更为糟糕的是，增加对新平台的支持有可能要求重新遍历这些杂乱分布的低质量代码段（实际上你很难能找到这类代码段的全部）。

与现有方式不同的是，Linux 一般通过简单函数（或者是宏）调用来抽象出不同平台间的差异。内核的移植可以通过实现适合于相应平台的函数（或宏）来实现。这样不仅使代码的主体简单易懂，而且在移植的过程中还可以比较容易地自动检测出你没有注意到的内容：如引用未声明函数时会出现链接错误。有时用预处理器来支持不同的体系结构，但这种方式并不常用，而相对于代码风格的变化就更是微不足道了。

顺便说一下，我们可以注意到这种解决方法和使用用户对象（或者 C 语言中充满函数指针的 **struct** 结构）来代替离散的 **switch** 语句处理不同类型的方法十分相似。在某些层次上，这些问题和解决方法是统一的。

可移植性的问题并不仅限于平台和 CPU 的移植，编译器也是一个重要的问题。此处为了简化，假设 Linux 只使用 gcc 来编译。由于 Linux 只使用同一个编译器，所以就没有必要使用**#if** 块（或者**#ifdef** 块）来选择不同的编译器。

内核代码主要使用**#ifdef** 来区分需要编译或不需要编译的部分，从而对不同的结构提供支持。例如，代码经常测试 **SMP** 宏是否定义过，从而决定是否支持 SMP 机。

代码样例

上一节仅仅是一些讨论，了解 Linux 代码风格最好的方法就是实际研究一下它的部分代码。即使你不完全理解本节所讨论代码的细节也无关紧要，毕竟本节的主要目的不是理解代码，一些读者可以只对本节进行浏览。本节的主要目的是让读者对 Linux 代码进行初步了解，对今后的工作提供必要基础。而讨论将涉及部分广泛使用到的内核代码。

printk

printk（25836 行）是内核内部消息日志记录函数。在出现诸如内核检测到其数据结构出现不一致的事件时，内核会使用 **printk** 把相关信息打印到系统控制台上。对于 **printk** 的调用一般分为如下几类：

- 紧急事件（emergency）－－例如，**panic** 函数（25563 行）多次使用了 **printk**。当内核检测到发生不可恢复的内部错误时就会调用 **panic** 函数，然后尽其所能的安全关闭计算机。这个函数中调用 **printk** 以提示用户系统将要关闭。
- 调试－－从 3816 行开始的**#ifdef** 块使用 **printk** 来打印 SMP 逻辑单元（box）中每一个处理器的相关配置信息，但是此过程只有在使用 **SMP_DEBUG** 标志编译代码的情况下才能够被执行。
- 普通信息－－例如，当机器启动时，内核必需估计系统速度以确保设备驱动程序能够忙等待（busy-waiting）一个精确的极短周期。计算这种估计值的函数名为 **calibrate_delay**（19654 行），它既在 19661 行使用 **printk** 声明马上开始计算，又在 19693 行报告计算结果。另外，在第 4 章将详细的介绍 **calibrate_delay** 函数。

如果你已经浏览过这些参照代码，你可能已经注意到 **printk** 和 **printf** 的参数十分类似：

一个格式化字符串，后跟零个或者多个参数加入字符串中。格式化字符串可能是以一组“<N>”开始，这里的 N 是从 0 到 7 的数字，包括 0 和 7 在内。数字区分了消息的日志等级（log level），只有当日志等级高于当前控制台定义的日志等级（**console_loglevel**，25650 行）时，才会打印消息。root 可以通过适当减小控制台的日志等级来过滤不是很紧急的消息。如果内核在格式化字符串中检测不到日志等级序列，那么就会一直打印消息。（实际上，日志等级序列并不一定要在格式化字符串中出现，可以在格式化文本中查找到它的代码。）

从 14946 行开始的**#define** 块说明了这些特殊序列，这些定义可以帮助调用者正确区分对 **printk** 的调用。简单的说，我称日志等级 0 到 4 为“紧急事件”，从等级 5 到等级 6 为“普通信息”，等级 7 自然就是我所说的“调试”。（这种分类方法并不意味着其它更好的分类方法没有用处，而只是目前我们还不关心它而已。）

在上面讨论的基础上，我们研究一下代码本身。

printk

25836：参数 **fmt** 是 **printf** 类型的格式化字符串。如果你对“…”部分的内容不熟悉，那就需要参阅一本好的 C 语言参考书（在其索引中查找“变参函数，variadic function”）。另外，在安装的 GNU/Linux 中的 **stdarg** 帮助里也包含了一个有关变参函数的简明描述，在这儿只需要敲入“man stdarg”就可以看到。

简单的说，“…”部分提示编译器 **fmt** 后面可能紧跟着数量不定的任何类型的参数。由于这些参数在编译的时候还没有类型和名字，内核使用由三个宏 **va_start**，**va_arg** 和 **va_end** 组成的特殊组以及一个特殊类型－－**va_list** 对它们进行处理。

25842：**msg_level** 记录了当前消息的日志等级。它是静态的，这看起来可能会有些奇怪－－为什么下一次对 **printk** 的调用需要记录日志等级呢？问题的答案是只有打印出新行（\n）或者赋给一个新的日志等级序列以后，当前消息才会结束。这样通过在包含消息结束的新行里调用 **printk**，就保证了在多个短期冲突的情况下，调用者只打印唯一一个长消息。

25845：在 SMP 逻辑单元中，内核可能试图从不同的 CPU 向控制台同时打印信息。（有时在单处理机（UP）逻辑单元中也会发生同样问题，但由于中断还未被覆盖掉，所以问题也并不十分明显。）如果不进行任何协同的话，结果就将处于完全无法让人了解的杂乱无章的状态，每个消息的各个部分都和其它消息的各个部分混杂交织在一起。相反，内核使用旋转锁（spin-lock）来控制对控制台的访问。旋转锁将在第 10 章对它进行深入的介绍。

如果你对 **flags** 在传送给 **spin_lock_irqsave** 之前为什么不对它初始化感到疑惑，请不要担心：**spin_lock_irqsave**（对于不同的版本请分别参看 12614 行，12637 行，12716 行，和 12837 行）是一个宏，而不是一个函数。该宏实际上是将值写入 **flags** 中，而不是从 **flags** 中读出值。（在 25895 行中，存储在 **flags** 中的信息被 **spin_lock_irqsave** 回读，请参看 12616 行，12639 行，12728 行和 12841 行）

25846：初始化变量 **args**，该变量代表 **printk** 参数中的“…”部分。

25848：调用内核自身的 **vsprintf**（为节省空间而省略）实现。该函数的功能与标准 **vsprintf** 函数非常相似，向 **buf** 中写入格式化文本（25634 行）并返回写入字符串的长度（长度不包括最后一位终止字符 0 字节）。很快，你将可以看到为什么这种机制会忽略 **buf** 的前三个字符。

（正如 25847 行的注释中所述）我们应该注意到在这里并没有采取严格的措施来保证缓冲器不会过载。这里系统假定 1024 个字符长度的 **buf** 已经足够使用（参阅 25634 行）。如果内核在这里能够使用 **vsnprintf** 函数的话，情况就会好许多。然而，**vsnprintf** 还有另外一个参数限制了它能够写入缓冲器的字符长度。

25849：计算 **buf** 中最近使用的元素，调用 **va_end** 终止对“…”参数的处理。

25851：开始格式化消息的循环。其中存在一个内部循环能够处理更多内容（这一点随后就能看到），因此，每次内循环开始，都开始一个新的打印行。由于通常情况下 **printk** 只用于打印单行，所以在每次调用中这种循环通常只执行一次。

25853：如果预先不知道消息的日志等级，**printk** 会检查当前行是否以日志等级序列开头。

25860：如果不是，**buf** 中开始未使用的三个字符就能够起作用了。（第一次以后的每次循环，都会覆盖部分消息文本，但是这样并不会引起问题，因为这里的文本只是前面行中的一部分，它们已经被打印过，而且以后也不再需要了。）这样，就可以将日志等级插入 **buf** 中。

25866：此处有如下属性：**p** 指向日志等级序列（消息文本紧随其后），**msg** 指向消息文本——请注意 25852 行和 25865 行中对 **msg** 的赋值。

由于已知 **p** 用来指示日志等级序列的开头－－该日志等级序列可能是由函数自身所创建的－－日志等级可以从 **p** 中抽出并存到 **msg_level** 中。

25868：没有检测到新行，清空 **line_feed** 标志。

25869：这是前面谈到过的内循环，循环将运行到本行结束（也就是检测到新行标志）或者缓冲器的末尾为止。

25870：除了将消息打印到控制台之外，**printk** 还能够记录最近打印的长度为 **LOG_BUF_LEN** 的字符组。（**LOG_BUF_LEN** 为 16K，请参看 25632 行。）如果在控制台打开之前，内核就已经调用 **printk**，则显然不能在控制台上正确打印消息，但是这些消息将被尽可能的存储到 **log_buf** 中（25656 行）。当控制台打开以后，缓存在 **log_buf** 中的数据就可以转储并在控制台上打印出来，请参看 25988 行。

log_buf 是一个循环缓冲器，**log_start** 和 **log_size** 变量（25657 行和 25646 行）分别记录当前缓冲器的开始位置和长度。本行中的按位与（AND）操作实际上是快速求模（%）运算，它的正确性依赖于 **LOG_BUF_LEN** 的值是 2 的幂。

25872：保存变量跟踪记录循环日志的值。显然，日志大小会不断增长，直至达到 **LOG_BUF_LEN** 的值为止。此后，**log_size** 将保持不变，而插入新字符将导致 **log_start** 的增长。

25878：请注意 **logged_chars**（25658 行）记录从机器启动之后 **printk** 写入的所有字符的长度，它在每次循环中都会被更新，而不是在循环结束后才改变一次。基于同样的道理，**log_start** 和 **log_size** 的处理方式也是一样。这实际上是一种优化的时机，但是我们将在结束对函数的介绍之后再对它详细讨论。

25879：消息被分为若干行，这当然要使用新行标志符来进行分割。一旦内核检测到新行标志符，就写入一个完整行，从而内循环的执行也可以提前终止。

25884：在这里我们先不考虑内部循环是否会提前退出，从 **msg** 到 **p** 的字符序列是专门提供给控制台使用的。（这种字符序列我称之为行，但是不要忘了，这里的行可能并不意味着新行终止，因为 **buf** 也许还没有终止。）如果该行的日志等级高于系统控制台定义的日志等级，而且当前又有控制台可供打印，那么就能够正确打印该行。（记住，**printk** 可能在所有控制台打开之前就已经被调用过了。）

如果在该信息块中没有发现日志等级序列，并且在前面的 **printk** 调用中也没有对 **msg_level** 赋值，那么本行中的 **msg_level** 就是-1。由于 **console_leglevel** 总不小于 1（除非 root 通过 sysctl 接口锁定），于是总是可以打印这些行。

25886：本行应该能够被打印。**printk** 通过遍历打开的控制台驱动链表告知每一个控制台驱动去打印当前行。（因为虽然设备驱动在本书的讨论范围之外，但是控制台驱动代码则并不包含在内。）

25888：请注意这里消息文本的开头使用的是 **msg** 而不是 **p**，这样就在没有日志等级序列的情况下写入消息了。然而，日志等级序列已经被存储到 **log_buf** 缓冲器中了。这样就可以使后来访问 **log_buf** 以获取信息日志等级的代码（请参看 25998 行）能够正确执行，不会再产生显示混乱信息序列的现象。

25892：如果内层 **for** 循环发现一新行，那么 **buf** 中的剩余字符（如果有的话）将被认为是新的消息，因此 **msg_level** 会被重置。但是无论怎样，外层循环都会持续到 **buf** 清空为止。

25895：释放在 25845 行获取的控制台锁（console lock）。

25896：唤醒等待被写入控制台日志的所有进程。注意即使没有文本被实际写入任何控制台，这个过程也仍然会发生。这样处理是正确的，因为无论是否要往控制台中写入文本，等待进程实际上都是在等待从 **log_buf** 中读出信息。在 25748 行，进程被转入休眠状态以等待 **log_buf** 的活动。在休眠、唤醒和等待队列中所使用的机制将在下一节中进行讨论。

25897：返回日志中写入的字符长度。

如果对于每个字符的处理工作都能减少一点，那么从 25869 行开始的 **for** 循环就能执行得更快一点。当循环存在时，我们可以通过只在循环退出时将 **logged_chars** 更新一次来稍微提高运行速度。然而我们还可以通过其它努力来提高速度。由于我们可以预知消息的长度，因此 **log_size** 和 **log_start** 可以到最后再增长。让我们来实验一下这样能否提高速度，下面是一段经过理想优化的代码：

```
do {
static int wrapped = 0;
const int x = wrapped
   ? log_start
   : log_size;
const int lim = LOG_BUF_LEN – x;
int n = buf_end – p;
if ( n>= lim)
   n = lim;

memcpy(log_buf + x, p, n);
p += n;

if(log_size < LOG_BUF_LEN)
   log_size += n;
else {
   wrapped = 1;
   log_start += n;
   ;og_start &= LOG_BUF_LEN – 1;
}
} while (p < buf_end);
```

请注意循环通常只需要执行一次，只有在 **log_buf** 末尾写入信息需要折行时才会多次执行。因而 **log_size** 和 **log_buf** 只需要更新一次（或者当写入需要换行时是两次）。

这时速度的确提高了，但是有两个原因使我们并不能这样做。首先，内核可能有自己特有的 **memcpy** 函数，我们必须确保对 **memcpy** 的调用不会再次进入对 printk 的调用。（有一

部分内核移植版定义了自己特有的速度较快的 **memcpy** 函数版本，因此所有的移植都要在这一点上保持一致。）如果 **memecpy** 调用 **printk** 来报告失败，那么就有可能触发无限循环。

然而在这一点上也并不是真的无药可救。使用这种解决方案的最大问题在于该内核循环的形式中也要留意新行标志符，因此使用 **memcpy** 将整个消息拷贝到 **log_buf** 中是不正确的：如果此处存在新行，我们将无法对其进行处理。

我们可以试验一个一箭双雕的办法。下面这种替代的尝试虽然可能比前面那种初步解决方法速度要慢，但是它保持了内核版本的语意：

```
/* in declaration section:*/
int n;
char *start;
static char *log = log_buf;
/*……*/

for (start = p;p < buf_end;p++) {
   *log++ = *p;
   if    (log >= (log_buf + LOG_BUF_LEN))
      log = log_buf ;    /* warp*/

   if (*p == '/n') {
      line_feed = 1;
      break;
   }
}

/* p - start is number of chars copied. */
n = p – start;
logged_chars += n ;
/*
*exercise for the reader:
*also use n to update log size and log_ start.
*(it's not as simple as may look.)
*/
```

（请注意 gcc 的优化器十分灵敏，它足以能检测到循环内部的表达式 **log_buf+LOG_BUF_LEN** 并没有改变，因此在上面的循环中试图手工加速计算是没有任何效果的。）

不幸的是，这种方法并不能比现在内核版本在速度上快许多，而且那样会使得代码晦涩难懂（如果你编写过更新 **log_size** 和 **log_start** 的代码，你就能清楚地了解这一点）。你可以自己决定这种折衷是否值得。然而无论怎样，我们学到了一些东西，这是通常的成果：不管成功与否，改进内核代码都可以加深你对内核工作原理的理解。

等待队列

前一节我们曾简要的提到进程（也就是正在运行的程序）可以转入休眠状态以等待某个特定事件，当该事件发生时这些进程能够被再次唤醒。内核实现这一功能的技术要点是把等

待队列（wait queue）和每一个事件联系起来。需要等待事件的进程在转入休眠状态后插入到队列中。当事件发生之后，内核遍历相应队列，唤醒休眠的任务让它投入运行状态。任务负责将自己从等待队列中清除。

等待队列的功能强大得令人吃惊，它们被广泛应用于整个内核中。更重要的是，实现等待队列的代码量并不大。

wait_queue 结构

18662：简单的数据结构就是等待队列节点，它包含两个元素：

- **tast**－－指向 **struct task_struct** 结构的指针，它代表一个进程。从 16325 行开始的 **struct task_struct** 结构将在第 7 章中进行介绍。
- **next**－－指向队列中下一节点的指针。因而，等待队列实际上是一个单链表。

通常，我们用指向等待队列队首的指针来表示等待队列。作为一个例子，请参看 **printk** 使用的等待队列 **log_wait**（25647 行）。

wait_event

16840：通过使用这个宏，内核代码能够使当前执行的进程在等待队列 **wq** 中等待直至给定 **condition**（可能是任何的表达式）得到满足。

16842：如果条件已经为真，当前进程显然也就无需等待了。

16844：否则，进程必须等待给定条件转变为真。这可以通过调用**__wait_event** 来实现（16824 行），我们将在下一节介绍它。由于 **__wait_event** 已经同 **wait_event** 分离，已知条件为假的部分内核代码可以直接调用**__wait_queue**，而不用通过宏来进行冗余的（特别是在这些情况下）测试，实际上也没有代码会真正这样处理。更为重要的是，如果条件已经为真，**wait_event** 会跳过将进程插入等待队列的代码。

注意 wait_event 的主体是用一个比较特殊的结构封闭起来的：

```
do {
   /* … */
} while (0)
```

使我惊奇的是，这个小技巧并没有得到应有的重视。这里的主要思路是使被封闭的代码能够像一个单句一样使用。考虑下面这个宏，该宏的目的是如果 **p** 是一个非空指针，则调用 **free**：

```
#define FREE1(p)    if    (p)    free (p)
```

除非你在如下所述的情况下使用 **FREE1**，否则所有调用都是正确有效的：

```
if    (expression)
      FREE1(p)
else
      printf(“expression was false.\n”) ;
```

FREE1 经扩展以后，**else** 就和错误的 **if**－－**FREE1** 的 **if**－－联系在一起。

我曾经发现有些程序员通过如下途径解决这种问题：

```
#define FREE2(p)    if (p) { free(p); }
#define FREE3(p)    { if (p) { free(p); } }
```

这两种方法都不尽人意，程序员在调用宏以后自然而然使用的分号会把扩展信息弄乱。以 **FREE2** 为例，在宏展开之后，为了使编译器能更准确的识别，我们还需要进行一定的缩进调节，最终代码如下所示：

```
if (expression)
   if (p) { free(p);}
else
   printf(“expression was false./n”);
```

这样就会引起语法错误－－**else** 和任何一个 **if** 都不匹配。**FREE3** 从本质上讲也存在同样的问题。而且在研究问题产生原因的同时，你也能够明白为什么宏体里是否包含 **if** 是无关紧要的。不管宏体内部内容如何，只要你使用一组括号来指定宏体，你就会碰到相同的问题。

这里是我们能够引入 **do/while(0)**技巧的地方。现在我们可以编写 **FREE4**，它能够克服前面所出现的所有问题。

```
#define FREE4(P)    \
do {                    \
      if (p)            \
         free(p);       \
      while (0)
```

将 **FREE4** 和其它宏一样插入相同代码之后，宏展开后其代码如下所示（为清晰起见，我们再次调整了缩进格式）：

```
if (expression)
   do {
      if (p)
         free(p);
   } while (0);      /* “;” following macro.*/
```

这段代码当然可以正确执行。编译器能够优化这个伪循环，舍弃循环控制，因此执行代码并没有速度的损失，我们也从而得到了能够实现理想功能的宏。

虽然这是一个可以接受的解决方案，但是我们不能不提到的是编写函数要比编写宏好得多。不过如果你不能提供函数调用所需的开销，那么就需要使用内联函数。这种情况虽然在内核中经常出现，但是在其它地方就要少得多。（无可否认，当使用 C++，gcc 或者任何实现了将要出现的修正版 ISO 标准 C 的编译器时，这种方案只是一种选择，就是最后为 C 增加内联函数。）

__wait_event

16842：**__wait_event** 使当前进程在等待队列 **wq** 中等待直至 **condition** 为真。

16829：通过调用 **add_wait_queue**（16791 行），局部变量**__wait** 可以被链接到队列上。注意**__wait** 是在堆栈中而不是在内核堆中分配空间，这是内核中常用的一种技巧。在宏运行结束之前，**__wait** 就已经被从等待队列中移走了，因此等待队列中指向它的指针总是有效的。

16830：重复分配 CPU 给另一个进程直至条件满足，这一点将在下面几节中讨论。

16831：进程被置为 **TASK_UNINTERRUPTIBLE** 状态（16190 行）。这意味着进程处于休眠状态，不应被唤醒，即使是信号量也不能打断该进程的休眠。信号量在第 6 章中介绍，而进程状态则在第 7 章中介绍。

16832：如果条件已经满足，则可以退出循环。

请注意如果在第一次循环时条件就已经满足，那么前面一行的赋值就浪费了（因为在循环结束之后进程状态会立刻被再次赋值）。**__wait_event** 假定宏开始执行时条件还没有得到满足。而且，这种对进程状态变量 **state** 的延迟赋值也并没有什么害处。在某些特殊情况下，这种方法还十分有益。例如当**__wait_event** 开始执行时条件为假，但是在执行到 16832 行时就为真了。这种变化只有在为有关进程状态的代码计算 **condition** 变量值时才会出现问题。但是在代码中这种情况我一处也没有发现。

16834：调用 **schedule**（26686 行，在第 7 章中讨论）将 CPU 转移给另一个进程。直到进程再次获得 CPU 时，对 **schedule** 的调用才会返回。这种情况只有当等待队列中的进程被唤醒时才会发生。

16836：进程已经退出了，因此条件必定已经得到了满足。进程重置 **TASK_RUNNING** 的状态（16188 行），使其适合 CPU 运行。

16837：通过调用 **remove_wait_queue**（16814 行）将进程从等待队列中移去。**wait_event_interruptible** 和**__wait_event_interruptible**（分别参见 16868 行和 16847）基本上与 **wait_event** 和**__wait_event** 相同，但不同的是它们允许休眠的进程可以被信号量中断。如前所述，信号量将在第 6 章中介绍。

请注意 **wait_event** 是被如下结构所包含的。

```
({
  /* … */
})
```

和 **do/while(0)**技巧一样，这样可以使被封闭起来的代码能够像一个单元一样运行。这样的封闭代码就是一个独立的表达式，而不是一个独立的语句。也就是说，它可以求值以供其它更复杂的表达式使用。发生这种情况的原因主要在于一些不可移植的 gcc 特有代码的存在。通过使用这类技巧，一个程序块中的最后一个表达式的值将定义为整个程序块的最终值。当在表达式中使用 **wait_event_interruptible** 时，执行宏体后赋**__ret** 的值为宏体的值（参看 16873 行）。对于有 Lisp 背景知识的程序员来说，这是个很常见的概念。但是如果你仅仅了解一点 C 和其它一些相关的过程性程序设计语言，那么你可能就会觉得比较奇怪。

__wake_up

26829：该函数用来唤醒等待队列中正在休眠的进程。它由 **wake_up** 和 **wake_up_interruptible** 调用（请分别参看 16612 行和 16614 行）。这些宏提供 **mode** 参数，只有状态满足 **mode** 所包含的状态之一的进程才可能被唤醒。

26833：正如将在第 10 章中详细讨论的那样，锁（lock）是用来限制对资源的访问，这在 SMP 逻辑单元中尤其重要，因为在这种情况下当一个 CPU 在修改某数据结构时，另一个 CPU 可能正在从该数据结构中读取数据，或者也有可能两个 CPU 同时对同一个数据结构进行修改，等等。在这种情况下，受保护的资源显然是等待队列。非常有趣的是所有的等待队列都使用同一个锁来保护。虽然这种方法要比为每一个等待队列定义一个新锁简单得多，但是这就意味着 SMP 逻辑单元可能经常会发现自己正在等待一个实际上并不必须的锁。

26838：本段代码遍历非空队列，为队列中正确状态的每一个进程调用 **wake_up_process**（26356 行）。如前所述，进程（队列节点）在此可能并没有从队列中移走。这在很大程度上是由于即使队列中的进程正在被唤醒，它仍然可能希望继续存在于等待队列中，这一点正如我们在**__wait_event** 中发现的问题一样。

内核模块（Kernel Modules）

整个内核并不需要同时装入内存。应该确认，为保证系统能够正常运行，一些特定的内核必须总是驻留在内存中，例如，进程调度代码就必须常驻内存。但是内核其它部分，例如大部分的设备驱动就应该仅在内核需要的时候才装载，而在其它情况下则无需占用内存。

举例来说，只有在内核真正和 CD-ROM 通讯时才需要使用完成内核与 CD-ROM 通讯的设备驱动程序，因此内核可以被设置为在和设备通讯之前才装载相应代码。内核完成和设备的通讯之后可以将这部分代码丢弃。也就是说，一旦代码不再需要，就可以从内存中移走。系统运行过程中可以增减的这部分内核称为内核模块。

内核模块的优点是可以简化内核自身的开发。假设你购买了一个新的高速 CD-ROM 驱动器，但是现有的 CD-ROM 驱动程序并不支持该设备。你自然就希望增加对这种高速模式的支持以提高系统光驱设备的性能。如果作为内核模块来编译驱动程序，你的工作将会方便得多：编译驱动程序，加载到内核，测试，卸载驱动程序，修改驱动程序，再次加载驱动程序到内核，测试，如此周而复始。如果你的驱动程序是直接编辑在内核中的，那么你就必须重新编译整个内核并且在每次修改驱动程序之后重新启动机器。这样慢得很多。

自然，你也必须留意内核模块。对于指明其它内核模块在磁盘上的驻留位置的那些模块，一定不能从内存中卸载，否则，内核将只能通过访问磁盘来装载处理磁盘访问的内核模块，这是不可能实现的。这也是我们要选择把部分内核作为模块编译还是直接编译进内核使其常驻内存的又一个原因。你知道自己系统的设置方式，因而也就可以自己选择正确使用的方式。（如果为了确保安全，你可以简单的忽略内核模块系统的优点，而把所有的内容都编译到内核里面。）

内核模块会带来一些速度上的损失，这是因为一些必需的代码现在并不在 RAM 中，必需要从磁盘读入。但是整个系统的性能通常会有所提高，这主要是因为通过丢弃暂时不使用的模块可以释放出额外的 RAM 供应用程序使用。如果这部分内存被内核所占用，应用程序将只能更加频繁地进行磁盘交换（swap），而这种磁盘交换会显著的降低应用程序的性能。（磁盘交换将在第 8 章中讨论。）

内核模块还会带来因复杂度的增加所造成的开销，这是因为在系统运行的过程中移进移出部分内核需要额外的代码。然而，正如你将在本节中看到的，复杂度的开销是可以管理的。通过使用外部程序来代理一些必需的工作还可以更进一步降低复杂度的开销。（更为确切的说法是，这样做不是减少了复杂度的开销，而是把复杂度的开销重新分配了一下。）这是对内核模块原理的一个小小的扩展：即使是内核的支持模块对于内核来说也只是外部的，部分可用的，只有在需要的时候才被装入内存。

通常用于这种目的程序称为 modprobe。有关的 modprobe 代码超出了本书的范围，但是在 Linux 的每个发行版本中都有包含有它。本节的剩余部分将讨论同 modprobe 协同工作以装载内核模块的内核代码。

request_module

24432：作为函数说明之前的注释，**request_module** 是一个函数。内核的其它模块在需要装载其它内核模块的时候，都必须调用这个函数。就像内核处理其它工作一样，这种调用也是为当前运行的进程进行的。从进程的角度来看，这种调用的请求通常是隐含的——正在执行进程其它请求的内核可能会发现必须调入一个模块才能够完成该请求。例如，请参看 10070 行，这里是一些将在第 7 章中讨论的代码。

24446：以内核中的一个独立进程的形式执行 **exec_modprobe** 函数（24384 行，马上就会讨论到）。这并不能只通过函数的简单调用实现，因为 **exec_modprobe** 要继续调用 **exec**

来执行一个程序。因此，对函数 **exec_modprobe** 的简单调用将永远不会有返回。这和使用 **fork** 以准备 **exec** 调用十分类似，你可以认为 **kernel_thread** 对内核来说就是较低版本的 **fork**，虽然两者有很大不同。**fork** 是从指定函数开始执行新的进程，而不是从调用者的当前位置开始运行。正如 **fork** 一样，**kernel_thread** 返回的值是新进程的进程号。

24448：和 **fork** 一样，从 **kernel_thread** 返回的负值表示内部错误。

24455：正如函数中论述的一样，大部分的信号量将因当前进程而被暂时阻塞。

24462：等待 **exec_modprobe** 执行完毕，同时指出所需要的模块是已经成功装入内存还是装载失败了。

24465：结束运行，恢复信号量。如果 **exec_modpro** 返回错误代码，则打印错误消息。

exec_modprobe

24384：**exec_modprobe** 运行为内核增加内核模块的程序。这里的模块名是一个 **void***的指针，而不是 **char***的指针。原因简单说来就是 **kernel_thread** 产生的函数通常都使用 **void***指针参数。

24386：设置 modprobe 的参数列表和环境。**Modprobe_path**（24363 行）用来定位 modprobe 程序的位置。它可以通过内核的 sysctl 特性来修改，这一点将在第 11 章中介绍（请参看 30388 行）。这意味着 root 可以动态选择不同于/sbin/modprobe 的程序来运行，以适应当 modprobe 被安装到其它地方或者使用修改过的 modprobe 替换掉了原有的 modprobe 之类的情况。

24400：（正如代码中描述的一样）出于安全性考虑，丢弃所有挂起的信号量和信号量句柄（handlers）。这里最重要的部分是对 **flush_signal_handlers** 的调用（28041 行），它使用内核默认的信号量句柄代替所有用户定义的信号量句柄。如果在此时有信号量被传送到内核，它将获得默认响应——通常是忽略信号量或杀死进程。但是不管怎样都不会引起安全风险。由于该函数从触发它的进程中分离出来（如前所述），所以不管原始进程在此处是否改变其原来分配的信号量句柄都不会产生任何影响。

24405：关闭调用进程打开的所有文件。最重要的是，这意味着 modprobe 程序不再从调用进程中继承标准输入输出和标准错误。这很有可能会引起安全漏洞。（这可能在替代 modprobe 的程序中引起的问题，但是 modprobe 本身实际上并不关心这个差异。）

24413：modprobe 程序作为 root 运行，它拥有 root 所拥有的所有权限。和整个内核中其它地方一样，请注意 root 使用用户 ID 号 0 的假定在这里已经被写入程序。用户 ID 号和权能系统（capability system）（在接下来的几行中会用到）将在第 7 章中介绍。

24421：试图执行 modprobe 程序。如果尝试失败，内核将使用 **printk** 打印错误消息并返回错误代码。这里是可能产生 **printk** 的缓冲器过载的地点之一。**module_name** 的长度并没有明确限制，就我们对该调用的看法而言，它可能长达一百万个字符。为防止 **printk** 缓冲器过载，你必需遍历所有对于该函数的调用（实际上是对 **request_module** 的调用）以保证每个调用者使用足够短的不会为 printk 造成麻烦的模块名。

24427：当 **execve** 成功执行时，它不会返回任何结果，因此本处是不可能执行到的。但是编译器却并不知道这一点，因此此处使用了 **return** 语句以保证 gcc 不出错。

对于内核的进一步讨论将超出本章的既定范围，因此在这个问题上我们到此为止。然而本书中也包括了其它必需的内核代码。在读完第 4 章和第 5 章之后，也许你会希望再次仔细研读一下这部分内容。有关这个问题的两个文件是 include/linux/module.h（从 15529 行开始）和/kernel/module.c（从 24476 行开始）。和 **sys_create_module**（24586 行），**sys_init_module**（24637 行），**sys_delete_module**（24860 行）和 **sys_query_module**（25148 行）四个函数需要特别注意一样，**struct module**（15581 行）也要特别引起注意。这些函数实现了 modprobe

以及 insmod，lsmod 和 rmmod 所使用的系统调用以完成模块的装载、定位和卸载。

内核触发直接回调内核程序的现象看起来很令人奇怪。但是，实际上进行的工作不止于此。例如，modprobe 必须实际访问磁盘以搜寻要装载的模块。而且更为重要的一点是这种方法赋予 root 对内核模块系统更多的控制能力。这主要是因为 root 也可以运行 modprobe 以及相关程序。因此，root 既可以手工装载、查询、卸载模块，也可以由内核自动完成。

配置与编译内核

你可能仅仅研读、欣赏而并不修改 Linux 内核源代码。但是，更普遍的情况是，用户有强烈的愿望去改进内核代码并完成相应的测试，这样我们就需要知道如何重建内核。本节就是要告诉你如何实现这一点，而最终则归结于如何把你所做的修改发行给别人，以使得每个人都能从你的工作中受益。

配置内核

编译内核的第一步就是配置内核，这是增加或者减少对内核特性的支持以及修改内核的一些特性发挥作用的方式的必要步骤。例如，你可以要求内核为自己的声卡指定一个不同的 DMA 通道。如果内核配置和你的需要相同，那么你可以直接跳过本节，否则请继续阅读以下内容。

为了完成内核的配置，请先切换到 root 用户，然后转入如下内核源程序目录：

```
cd /usr/src/linux
```

接着敲入如下命令组：

```
make config
make menuconfig
make xconfig
```

这三条命令都可以让你来配置内核，但它们发挥作用的方式各不相同：

- **make config**－－三种方法中最简单也是最枯燥的一种。但是最基本的一点是，它可以适应任何情况。这种方法通过为每一个内核支持的特性向用户提问的方式来决定在内核中需要包含哪些特性。对于大多数问题，你只要回答 y（yes，把该特性编译进内核中），m（作为模块编译）或者 n（no，根本不对该特性提供支持）。在决定之前用户应该考虑清楚，因为这个过程是不可逆的。如果你在该过程中犯了错误，就只能按 Ctrl+C 退出。你也可以敲入?以获取帮助。图 2.1 显示了这种方法正在 X 终端上运行的情况。

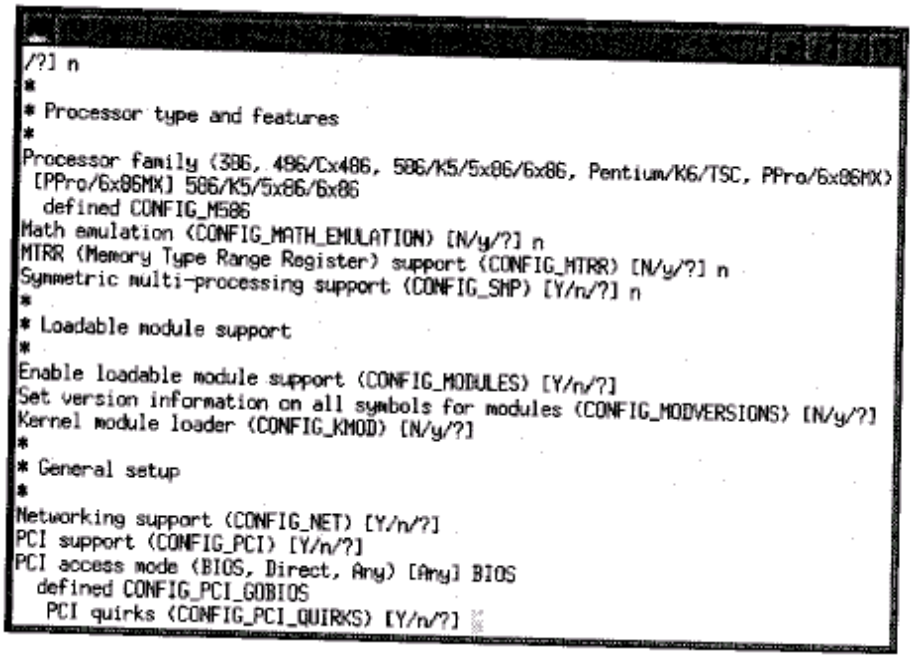

图 2.1 运行中的 **make config**

幸运的是，这种方法还有一些智能。例如，如果你对 SCSI 支持回答 no，那么系统就不会再询问你有关 SCSI 的细节问题了。而且你可以只按回车键以接受缺省的选择，也就是当前的设置（因此，如果当前内核将对于 SCSI 的支持编译进了内核，在这个问题上按回车键就意味着继续把对 SCSI 的支持编译进内核中）。即使是这样，大部分用户还是宁愿使用另外的两种方法。

- **make menuconfig**－一种基于终端的配置机制，用户拥有通过移动光标来进行浏览等功能。图 2.2 显示了在 X 终端上运行的 **make menuconfig**。虽然在控制台上显示的是彩色，但是在终端上的显示仍然相当单调。使用 menuconfig 必须要有相应的 ncurses 类库。

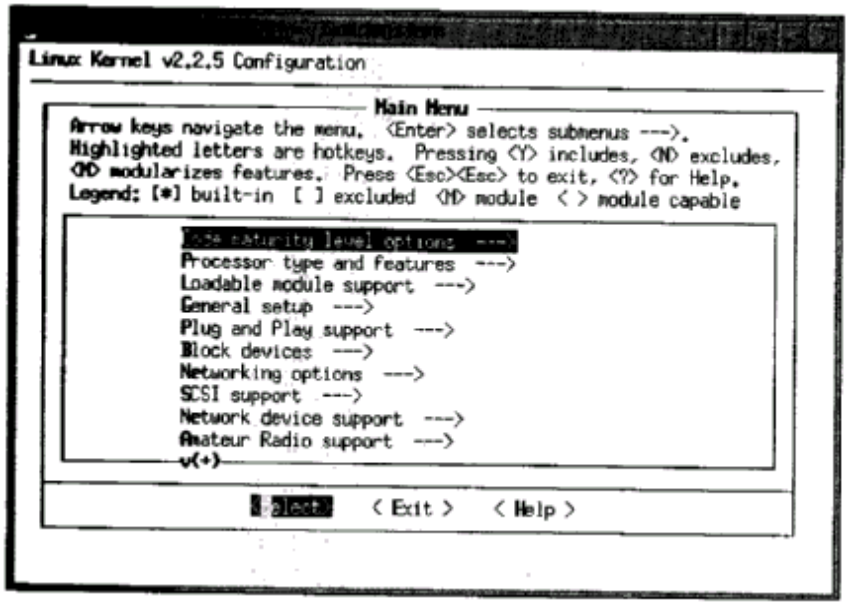

图 2.2 运行中的 **make menuconfig**

- **make xconfig**－－这是我最喜欢的一种配置方式。只有你能够在 X server 上用 root 用户身份运行 X 应用程序时，这种配置方式才可以使用（有些偏执的用户就不愿意使用这种方式）。你还必须拥有 Tcl 窗口系统（Tcl windowing system），这实际上还意味着你必须拥有 Tcl，Tk 以及一个正在运行的 X 安装程序。作为补偿，用户获得的是更漂亮的，基于 X 系统的以及和 menuconfig 功能相同的配置方法。图 2.3 显示这种方法运行过程中打开“可装载模块支持（Loadable module support）”子窗口的情况。


t_5.JPG

图 2.3 运行中的 **make xconfig**

如上所述，这三种方法都实现了相同的功能：它们都生成在构建内核时使用的.config 文件。而唯一的区别是在于创建这个文件时的难易程度不同。

构建内核

构建内核要做的工作要比配置内核所做的工作少得多。虽然有几种方式都能实现这一功能，但是选择哪一种依赖于你希望怎样对系统进行设置。长期以来，我已经形成了如下的习惯。虽然这种习惯比我所必须要做的略微多一些，但是它包含了所有基本的问题。首先，如果你还不在内核源程序目录中，请先再次转入这一目录：

```
cd  /usr/src/linux
```

现在，切换到 root 用户，使用下面显示的命令生成内核。现在在 shell 中敲入下面的命令，注意 make 命令因为空间关系分成了两行，但实际上这在 shell 输入时是一个只有一行的命令：

```
make dep clean zlilo boot
    modules modules_install
```

当给出了如上多个目标时，除非前面所有的目标都成功了，否则 make 能够知道没有必要继续尝试下面的目标。因此，如果 make 能够运行结束，成功退出，那么这就意味着所有的目标都正确构建了。现在你可以重新启动机器以运行新的内核。

备份的重要性

当修改（fooling）内核时，你必须准备一个能够启动的备用内核。实现该目的的一种方式是通过配置 Linux 加载程序（LILO）以允许用户选择启动的内核映象，其中之一是从没有修改过的内核的备份（我总是这样做的）。

如果你比较有耐心，那么你就可以使用 zdisk 目标而不使用 zlilo 目标；它可以把能够启动的内核映象写入软盘中。这样你就可以通过在启动时插入软盘的方式启动你的测试内核；如果没有插入软盘，则启动正常的内核。

但是请注意：内核模块并没有被装载到软盘中，它们实际上是装在硬盘中的（除非你愿意承担更多的麻烦）。因此，如果你弄乱了内核模块，即使是 zdisk 目标也救不了你。实际上，上面提到的这两种方法都存在这个问题。虽然有比较好的解决方法可用，但是最简单的方法（也就是我所使用的方法）是把备份内核作为严格独立的内核来编译，而不使用可装载模块的支持。通过这种方法，即使我弄乱了内核而不得不使用备份启动系统，那么不管问题是实验性内核不正确还是内核模块的原因都无关紧要。不管怎样，在备份的内核中已经有我

需要的所有东西了。

由于用户所作的修改可能导致系统的崩溃，如损坏磁盘上的数据等等，并不仅仅只是打乱设备驱动程序或文件系统，在测试新内核之前，备份系统的最新数据也是一个英明的决策。（虽然设备驱动程序的开发不是本书的主题，但是必需指出的是，设备驱动程序的缺陷可能会引起系统的物理损坏。例如显示器是不能备份的，而且因价格昂贵而不易替换。）作为一个潜在的内核黑客，你的最佳投资（当然是读过本书以后）是一个磁带驱动器和充足的磁带。

发行你的改进

下面是有关发行你所做修改的一些基本规则：

- 检查最新发行版本，确保你所处理的不是已经解决了的问题
- 遵守 Linux 内核代码编写的风格。简要的说就是 8 字符缩进以及 K&R 括号风格（**if**，**else**，**for**，**while**，**switch** 或者 **do** 后面同一行中紧跟着开括号）。在内核源程序目录下面的文档编写和代码风格文件给出了完整的规则，不过我们已经介绍了其中的关键部分。注意本书中包含的源程序代码为节省空间而进行了大量的重新编辑，在该过程中我可能打破了其中的一些规则。
- 独立发行相对无关的修改。这样，只想使用你所做的某部分修改的人就可以十分方便地获得想要的东西，而不用一次检验所有的修改内容。
- 让使用你所做修改的用户清楚他们可以从你的修改中获取什么。同样地，你也应该给出这些问题的可信度。你是 15 分钟之前才匆匆完成你的修改，甚至还没有时间对它们进行编译，还是已经在你和你的朋友的系统中从去年 3 月开始就长期稳定的运行过这个修改？

假设现在你已经准备好发行自己的修改版本了，那么要做的第一步是建立一个说明你所做的修改的文件。你可以使用 diff 程序自动创建这个文件。结果或者被称为 diffs，也或者在 Linux 中更普遍的被称为补丁（patch）。

发布的过程十分简单。假设原来没有修改过的源程序代码在 linux-2.2.5 目录下，而你修改过的源程序代码在 linux-my 目录下，那么只要进行如下的简单工作就可以了（只有在链接不存在的情况下才需要执行 ln）：

```
ln –s linux-my linux
make –C linux-2.2.5 distclean
make –C linux distclean
diff –urN linux-2.2.5 linux >my.patch
```

现在，输出文件 my.patch 包含了其它用户应用这个修改程序时所必须的一切内容。（警告：如上所述，两个源程序间的所有差别都会包含在这个补丁文件中。Diff 不能区分修改部分之间的关系，所以就把它们都罗列了出来。）如果补丁文件相对较小，你可以使用邮件直接发往内核邮件列表。如果补丁很大，那么就需要通过 FTP 或者 Web 站点发布。这时发给邮件列表的信件中就只需要包含一个 URL。

Linux 内核邮件列表的常见问题解答（FAQ）文件位于 **http://www.ececs.uc.edu/~rreilova/linux/lkmlfaq.html**。该 FAQ 中包含了邮件列表的订阅，邮件发布以及阅读邮件列表的注意事项等等。

顺便提一下，如果你想随时了解内核更新开发的进程，我向你强烈推荐下面这个具有很高价值的内核交流站点 Kernel Traffic：**http://www.kt.opensrc.org**。

第3章 内核体系结构概述

本章从较高层次上对内核进行说明。从顺序上来说，本章首先介绍内核设计目标，接下来介绍内核体系结构，最后介绍内核源程序目录结构。

内核设计目标

Linux 的内核展现出了几个相互关联的设计目标，它们依次是：清晰性（clarity），兼容性（compatibility），可移植性（portability），健壮性（robustness），安全性（security）和速度（speed）。这些目标有时是互补的，有时则是矛盾的。但是它们被尽可能的保持在相互一致的状态，内核设计和实现的特性通常都要回归到这些问题上来。本节接下来的部分将分别讨论这些设计目标，同时还将对它们之间的取舍与平衡进行简要的说明。

清晰性

稍微过于简化的说，内核目标是在保证速度和健壮性的前提下尽量清晰。这和现在的大多数应用程序的开发有所区别，后者的目标通常是在保证清晰性和健壮性的基础上尽量提高速度。因而在内核内部，速度和清晰性经常是一对矛盾。

在某种程度上，清晰性是健壮性的必要补充：一个很容易理解的实现方法比较容易证明是正确的；或者即使不正确，也能比较容易的找出其问题所在。从而这两个目标很少会发生冲突。

但是清晰性和速度通常却是一对矛盾。经过仔细手工优化的算法通常都使用了编译器生成代码的类似技术，很少可能是最清晰的解决方案。当内核中清晰性和速度要求不一致时，通常都是以牺牲清晰性来保证速度的。即便如此，程序员仍然清楚的知道清晰性的重要性，而且他们也做了大量完美的工作以使用最清晰的方法保证速度。

兼容性

正如第 1 章中所述，Linux 最初的编写目的是为了实现一个完整的、与 Unix 兼容的操作系统内核。随着开发过程的展开，它也开始以符合 POSIX 标准为目标。就内核而言，兼容 Unix（至少是同某一现代的 Unix 实现相兼容）和符合 POSIX 标准并没有什么区别，因此我们也不会在这个问题上详细追究。

内核提供了另外一种类型的兼容性。基于 Linux 的系统能够提供可选择的对 Java.class 文件的本地运行支持。（据说 Linux 是第一个提供这种支持的操作系统。）尽管实际负责 Java 程序解释执行的是另外一个 Java 虚拟机进程，该虚拟机并没有内置到内核中。但是内核提供的这种机制可以使得这种支持对用户是透明的。通过内核本身提供的程度不同的支持（这并不代表大部分工作像 Java 的解决方式一样能够通过外部进程实现），对其它可执行文件格式的支持也能够以同样的方式插入内核中。这方面的内容将在第 7 章中详细介绍。

另外需要说明的是，GNU/Linux 系统作为一个整体通过 DOSEMU 仿真机器提供了对 DOS 可执行程序的支持，而且也通过 WINE 设计提供了对 Windows 可执行程序的部分支持。系统还以同样的方式通过 SAMBA 提供了对 Windows 兼容文件和打印服务的支持。但是这些都不是同内核密切相关的问题，因此在本书中我们不再对它们进行讨论。

兼容性的另外一个方面是兼容异种文件系统，本章中稍后会有更为详细的介绍，但是大

部分内容已经超出了本书的范围。Linux 能够支持很多文件系统，例如 ext2（“本地”文件系统），ISO-9660（CD-ROM 使用的文件系统），MS-DOS，网络文件系统（NFS）等许多其它文件系统。如果你有使用其它操作系统格式的磁盘或者一个网络磁盘服务器，那么 Linux 将能够和这些不同的文件系统进行交互。

兼容性的另外一个问题是网络，这在当今 Internet 流行的时代尤为重要。作为 Unix 的一个变种，Linux 自然从很早就开始提供对 TCP/IP 的支持。内核还支持其它许多网络协议，它们包括 AppleTalk 协议的代码，这使得 Linux 单元（box）可以和 Macintosh 机自由通讯；Novell 的网络协议，也就是网络报文交换（IPX），分组报文交换（SPX），和 NetWare 核心协议（NCP）；IP 协议的新版本 IPv6；以及其它一些不太出名的协议。

兼容性考虑的最后一个方面是硬件兼容性。似乎每个不常见的显卡，市场份额小的网卡，非标准的 CD-ROM 接口和专用磁带设备都有 Linux 的驱动程序。（只要它不是专为特定操作系统设计的专用硬件。）而且只要越来越多的厂商也逐渐认识到 Linux 的优势，并能够为更容易地实现向 Linux 上移植而开放相应的源程序代码，Linux 对硬件支持会越来越好。

这些兼容性必须通过一个重要的子目标：模块度（Modularity）来实现。在可能的情况下，内核只定义子系统的抽象接口，这种抽象接口可以通过任何方法来实现。例如，内核对于新文件系统的支持将简化为对虚拟文件系统（VFS）接口的代码实现。第 7 章中介绍的是另外一个例子，内核对二进制句柄的抽象支持是实现对诸如 Java 之类的新可执行格式的支持的方法。增加新的可执行格式的支持将转变为对相应的二进制句柄接口的实现。

可移植性

与硬件兼容性相关的设计目标是可移植性（portability），也就是在不同硬件平台上运行 Linux 的能力。系统最初是为运行在标准 IBM 兼容机上的 Intel x86 CPU 而设计的，当时根本没有考虑到可移植性的问题。但是情况从那以后已经发生了很大的变化。现在正式的内核移植包括向基于 Alpha，ARM，Motorola 69x0，MIPS，PowerPC，SPARC 以及 SPARC-64 CPU 系统的移植。因而，Linux 可以在 Amigas，旧版或新版的 Macintosh，Sun 和 SGI 工作站以及 NeXT 机等机器上运行。而且这些还只是标准内核发行版本的移植范围。从老的 DEC VAX 到 3Com 掌上系列个人数字助理（例如 Palm III）的非正式的移植工作也在不断进行中。成功的非正式移植版本后来通常都会变成正式的移植版本，因此这些非正式的移植版本很多最终都会出现在主开发树中。

广泛平台支持之所以能够成功的部分原因在于内核把源程序代码清晰地划分为体系结构无关部分和体系结构相关部分。在本章的后续部分将对这个问题进行更深入的讨论。

健壮性和安全性

Linux 必须健壮、稳定。系统自身应该没有任何缺陷，并它还应该可以保护进程（用户）以防止互相干扰，这就像把整个系统从其它系统中隔离开来加以保护一样。后一种考虑很大程度上是受信任的用户空间应用程序领域的问题，但是内核至少也应该提供支撑安全体系的原语（primitive）。健壮性和安全性比任何别的目标都要重要，包括速度。（系统崩溃的速度很快又有什么好处呢？）

保证 Linux 健壮性和安全性的唯一一个最重要的因素是其开放的开发过程，它可以被看作是一种广泛而严格的检查。内核中的每一行代码、每一个改变都会很快由世界上数不清的程序员检验。还有一些程序员专门负责寻找和报告潜在的缺陷——他们这样做完全是出于自己的个人爱好，因为他们也希望自己的 Linux 系统能够健壮安全。以前检查中所没有发现的缺陷可以通过这类人的努力来定位、修复，而这种修复又合并进主开发树以使所有的人都能

够受益。安全警告和缺陷报告通常在几天甚至几个小时内就能够得到处理和修复。

Linux 可能并不一定是现有的最安全的操作系统（很多人认为这项桂冠应该属于 OpenBSD，它是一个以安全性为主要目标的 Unix 变种），但是它是一个有力的竞争者。而且 Linux 健壮性远没有发展到尽头。

速度

这个术语经常自己就可以说明问题。速度几乎是最重要的衡量标准，虽然其等级比健壮性、安全性和（在有些时候）兼容性的等级要低。然而它却是代码最直观的几个方面之一。Linux 内核代码经过了彻底的优化，而最经常使用的部分－－例如调度－－则是优化工作的重点。几乎在任何时候都有一些不可思议的代码，这是由于这种方式的执行速度比较快。（这并不总是很明显，但是你经常不得不通过自己的试验来对这种优化代码进行确认。）虽然有时一些更直接的实现方法速度也很快，但是我所见过的这种情况屈指可数。

在某些情况下，本书推荐用可读性更好的代码来替代那些以速度的名义而被故意扭曲了的代码。虽然速度是一个设计目标，但我基本上只在以下两种情况时才会这样做：a) 在所考虑的问题中，速度明显不是关键问题 b) 没有其它的办法。

内核体系结构初始

图 3.1 是一种类 Unix 操作系统的相当标准的视图，实际上，更细致的来说，该图能够说明所有期望具有平台无关特性的操作系统。它着重强调了内核的下面两个特性：

- 内核将应用程序和硬件分离开来。
- 部分内核是体系结构和硬件特有的，而部分内核则是可移植的。

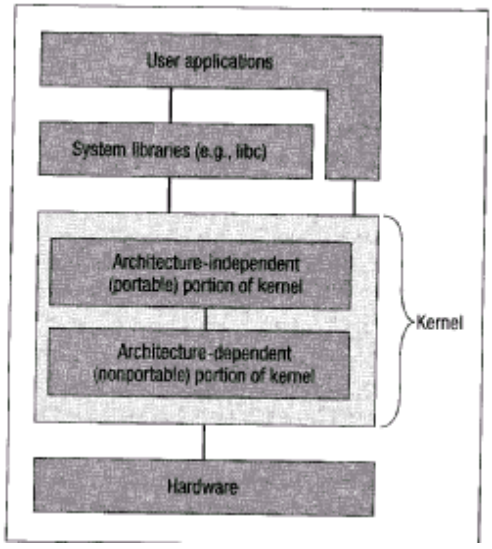

图 3.1 内核体系结构基本结构图

第一点我们在前面章节中已经讨论清楚了，在这里没有必要重复说明。第二点，也就是与体系结构无关和与体系结构相关代码的内容对于我们的讨论比较有意义。内核通过使用与处理用户应用程序相同的技巧来实现部分可移植性。这也就是说，如同内核把用户应用程序和硬件分离一样，部分内核将会因为与硬件的联系而同其它内核分离开来。通过这种分离，

用户应用程序和部分内核都成为可移植的。

虽然这通常并不能够使得内核本身更清楚，但是源程序代码的体系结构无关部分通常定义了与低层，也就是体系结构相关部分（或假定）的接口。作为一个简单的例子，内存管理代码中的体系结构无关部分假定只要包含特定的头文件就可以获得合适的 **PAGE_SIZE** 宏（参看 10791 行）的定义，该宏定义了系统的内存管理硬件用于分割系统地址空间的内存块的大小（参看第 8 章）。体系结构无关代码并不关心宏的确切定义，而把这些问题都留给体系结构相关代码去处理。（顺便一提，这比到处使用**#ifdef/#endif** 程序块来定义平台相关代码要清晰易懂得多。）

这样，内核向新的体系结构的移植就转变成为确认这些特性以及在新内核上实现它们的问题。

另外，用户应用程序的可移植性还可以通过它和内核的中间层次－－标准 C 库（libc）－－的协助来实现。应用程序实际上从不和内核直接通讯，而只通过 libc 来实现。图 3.1 中显示应用程序和内核直接通讯的唯一原因在于它们能够和内核通讯。虽然在实际上应用程序并不同内核直接通讯－－这样做是毫无意义的。通过直接和内核通讯所能处理的问题都可以通过使用 libc 实现，而且更容易。

Libc 和内核通讯的方式是体系结构相关的（这和图中有一点矛盾），libc 负责将用户代码从实现细节中解放出来。有趣的是，甚至大部分 libc 都不了解这些细节。大部分的 libc，例如 **atoi** 和 **rand** 的实现，都根本不需要和内核进行通讯。剩余部分的大部分 libc，例如 **printf** 函数，在涉及到内核之前或之后就已经处理大量的工作。（**printf** 必需首先解释格式化字符串，分析相应参数，设定打印方法，在临时内部缓冲器中记录预期输出。直到此时它才调用底层系统调用 **write** 来实际打印该缓冲区。）其它部分的 libc 则只是相应系统调用的简单代理。因而一旦发生函数调用时，它们会立即调用内核相应函数以完成主要工作。在最低层次上，大部分 libc 通过单通道同内核进行交流，而它们所使用的机制将第 5 章中进行详细介绍。

由于这种设计，所有的用户应用程序，甚至大部分的 C 库，都是通过体系结构无关的方式和内核通讯的。

内核体系结构的深入了解

图 3.2 显示了内核概念化的一种可能方式。该图和区分内核的体系结构无关和体系结构相关的方法有所不同，它是一种更具有普遍性的结构视图。在“Kernel”框内的本书中有所涉及的内核部分都用括号注明了相应的章节编号。虽然有关对称多处理（SMP）的支持也属于本书的范围，但是在这里我们却没有标明章号。部分原因在于相当多的 SMP 代码广泛地分布于整个内核中，因此很难将它与某一个模块联系起来。同样的道理，对于内核初始化的支持也属于本书的范围，但是也没有标明章号。这样做仅仅是因为从设计的观点上看，该问题并不重要。最后，虽然在图中我们将第 6 章和“进程间通讯”框联系在一起，但是该章只涉及一部分进程间通讯的内容。

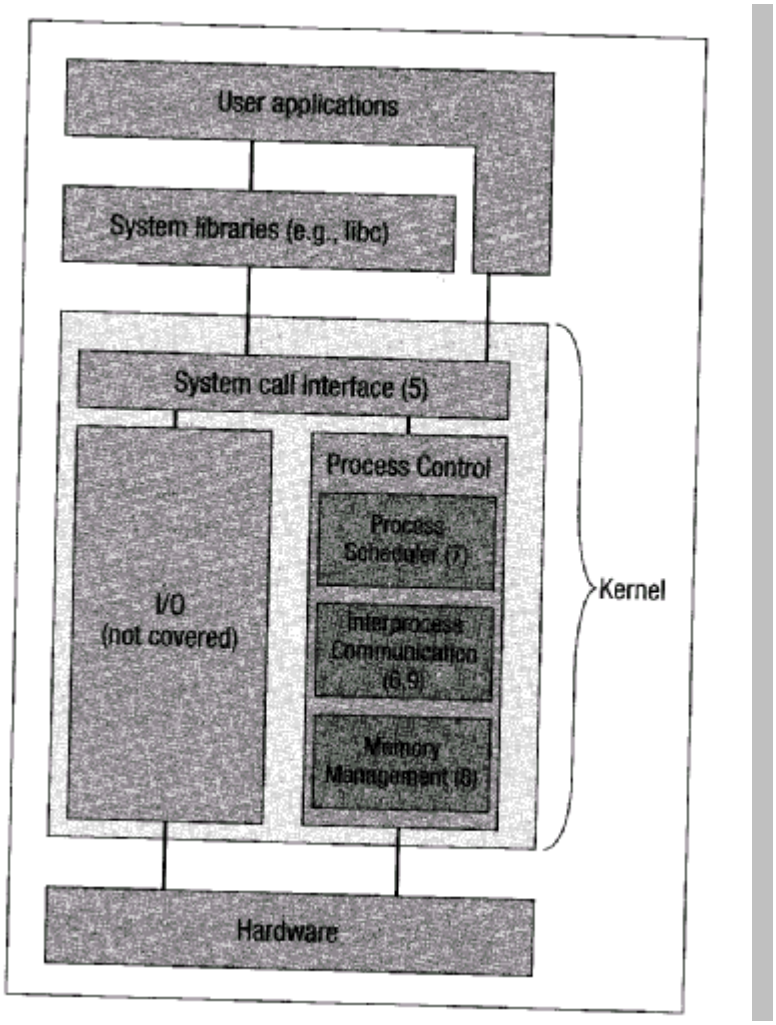

图 3.2 详细的内核体系结构图

进程和内核的交互通常需要通过如下步骤：

1. 用户应用程序调用系统调用，通常是使用 libc。
2. 该调用被内核的 **system_call** 函数截获（第 5 章，171 行），此后该函数会将调用请求转发给另外的执行请求的内核函数。
3. 该函数随即和相关内部代码模块建立通讯，而这些模块还可能需要和其它的代码模块或者底层硬件通讯。
4. 结果按照同样的路径依次返回。

然而，并不是所有内核和进程间的交互都是由进程发起的。内核有时也会自行决定同哪个进程交互，例如通过释放信号量或者简单的采用直接杀死进程的方法终止该进程的执行（如当进程用完所有可用的 CPU 时间片），以便使其它进程有机会运行。这些交互过程在该图中并没有表示，主要是因为它们通常都只是内核对自己的内部数据结构的修改（信号量传递对于这种规则来说是一个例外）。

是层次化（Layered），模块化（Modular）还是其它？

解决复杂性的所有方法都基于一个基本原理：问题分解和各个击破。也就是说，都是把

大型的、难以解决的问题（或系统）分解成一定数量的复杂度较低的子问题（或子系统），再根据需要重复这一过程直到每一部分都小到可以解决为止，而各种方法只是这种原理的一些不同运用而已。

计算机科学中有三种经典的方法比较适合于构建大型系统（我首先必须说明的是，这些定义都是经过我深思熟虑的讨论对象）。

- 层次（Layer）－－将解决方案分解成若干部分，在这些部分中存在一个问题域的最底层，它为上层的抽象层次较高的工作提供基础。较高层建立在其低层基础之上。OSI 和 TCP/IP 协议堆栈是众所周知的层次化软件设计的成功的例子。操作系统设计的层次化解决方案可能会包含一个可以直接和硬件通讯的层次、然后在其上提供为更高层提供抽象支持的层次。这样更高层就可以对磁盘、网卡等硬件进行访问，而并不需要了解这些设备的具体细节。

  层次化设计的一个特征是要逐步构建符号集（vocabulary）。随着层次的升高，符号集的功能将越来越强大。层次化设计的另外一个特征是完全可以在对其上下层透明的条件下替换某一层次。在最理想的情况下，移植层次化的操作系统只需要重写最低层的代码。纯层次化模型实现的执行速度可能会很慢，因为高层必须（间接的）通过调用一系列连续的低层才能处理完自己的任务－－N 层调用 N-1 层，N-1 层调用 N-2 层，等等，直到实际的工作在 0 层被处理完成。接着，结果当然是通过同样的路径反向传递回来。因此，层次化设计通常会包含对某些高层直接和某些低层通讯的支持；这样虽然提高了速度，但是却使得各个层次的替换工作更加困难（因为不止一个高层会直接依赖于这个你所希望进行替换的层次）。
- 模块（Module）－－模块将具体的一部分功能块隐藏在抽象的接口背后。模块的最大特点是将接口和其实现分离开来，这样就能够保证一个模块可以在不影响其它模块的情况下进行改变。这样也将模块之间的依赖关系仅仅限定于接口。模块的范围是试图反映求解域内一些方面的自然的概念性界限。纯模块化的操作系统因而就可能有一个磁盘子系统模块，一个内存管理子系统模块，等等。纯模块化和纯层次化的操作系统之间的主要区别是一个可以由其它模块自由调用，模块间没有上层和下层的概念。（从这个意义上来说，模块是广义的层次。按照纯粹的观点，层次是最多可供一个其它模块调用的模块，这个模块也就是它的直接上层模块。）
- 对象（Object）－－对象和模块不同，因为对于初学者来说它们具有不同的问题考虑方式，实现的方法也可能各自独立。但是，就我们当前的目的来说，对象不过是结构化使用模块的方法。组件（Component）作为对象思想的进一步改进目前还没有在操作系统设计中广泛使用。即便如此（按照我们的观点），我们也没有足够的理由将其和模块划分在不同的范畴中。

图 3.1 强调了内核的层次化的视图，而且是体系结构无关层次位于体系结构相关层次之上。（更为精确的视图是在顶层增加一个附加的体系结构相关的层次。这是因为系统调用接口位于应用程序和内核之间，而且是体系结构相关的。）图 3.2 着重强调了更加模块化的内核视图。

从合理的表述层次上看，这两种观点都是正确的。但也可以说这两种观点都是错误的。我可以用大量的图片向你证明内核是遵从所有你所能够指出的设计原则集合的，因为它就是从众多思想中抽取出来的。简单说来，事实是 Linux 内核既不是严格层次化的，也不是严格模块化的，也不是严格意义上的任何类型，而是以实用为主要依据的。（实际上，如果要用一个词来概括 Linux 从设计到实现的所有特点，那么实用就是最确切的。）也许最保守的观点是内核的实现是模块化的，虽然这些模块有时会为了追求速度而有意跨越模块的界限。

这样，Linux 的设计同时兼顾了理论和实际。Linux 并没有忽视设计方法；相反，在 Linux

的开发基本思想中，设计方法的作用就像是编译器：它是完成工作的有力工具。选择一个基本的设计原则（例如对象）并完全使用这种原则，不允许有任何例外，这对于测试该原则的限制，或者构建以说明这些方法为目的的教学系统来说都是一个不错的方法。但是如果要用它来达到 Linux 的设计目标则会引起许多问题。而且 Linux 的设计目标中也并不包括要使内核成为一个完全纯化的系统。Linux 开发者为了达到设计目标宁愿违背妨碍目标实现的原则。

实际上，如果对于 Linux 来说是正确的，那么它们对于所有最成功的设计来说都是正确的。最成功、应用最广泛的实际系统必然是实用的系统。有些开发人员试图寻找功能强大的可以解决所有问题的特殊方法。他们一旦找到了这种方法，所有的问题就都迎刃而解了。像 Linux 内核一样的成功设计通常需要为系统的不同部分和描述上的不同层次使用不同的方法。这样做的结果可能不是很清晰，也不是很纯粹，但是这种混合产物比同等功能的纯粹系统要强大而且优秀得多。

Linux 大部分都是单内核的

操作系统内核可能是微内核，也可能是单内核（后者有时称之为宏内核 Macrokernel）。按照类似封装的形式，这些术语定义如下：

- 微内核（Microkernel kernel）－－在微内核中，大部分内核都作为独立的进程在特权状态下运行，它们通过消息传递进行通讯。在典型情况下，每个概念模块都有一个进程。因此，如果在设计中有一个系统调用模块，那么就必然有一个相应的进程来接收系统调用，并和能够执行系统调用的其它进程（或模块）通讯以完成所需任务。

  在这些设计中，微内核部分经常只不过是一个消息转发站：当系统调用模块要给文件系统模块发送消息时，消息直接通过内核转发。这种方式有助于实现模块间的隔离。（某些时候，模块也可以直接给其它模块传递消息。）在一些微内核的设计中，更多的功能，如 I/O 等，也都被封装在内核中了。但是最根本的思想还是要保持微内核尽量小，这样只需要把微内核本身进行移植就可以完成将整个内核移植到新的平台上。其它模块都只依赖于微内核或其它模块，并不直接直接依赖硬件。

  微内核设计的一个优点是在不影响系统其它部分的情况下，用更高效的实现代替现有文件系统模块的工作将会更加容易。我们甚至可以在系统运行时将开发出的新系统模块或者需要替换现有模块的模块直接而且迅速的加入系统。另外一个优点是不需要的模块将不会被加载到内存中，因此微内核就可以更有效的利用内存。
- 单内核（Monolithic kernel）－－单内核是一个很大的进程。它的内部又可以被分为若干模块（或者是层次或其它）。但是在运行的时候，它是一个独立的二进制大映象。其模块间的通讯是通过直接调用其它模块中的函数实现的，而不是消息传递。

单内核的支持者声称微内核的消息传递开销引起了效率的损失。微内核的支持者则认为因此而增加的内核设计的灵活性和可维护性可以弥补任何损失。

我并不想讨论这些问题，但必须说明非常有趣的一点是，这种争论经常会令人想到前几年 CPU 领域中 RISC 和 CISC 的斗争。现代的成功 CPU 设计中包含了所有这两种技术，就像 Linux 内核是微内核和单一内核的混合产物一样。Linux 内核基本上是单一的，但是它并不是一个纯粹的集成内核。前面一章所介绍的内核模块系统将微内核的许多优点引入到 Linux 的单内核设计中。（顺便提一下，我考虑过一种有趣的情况，就是 Linux 的内核模块系统可以将系统内核转化成为简单的不传递消息的微内核设计。虽然我并不赞成，但是它仍然是一个有趣的想法。）

为什么 Linux 必然是单内核的呢？一个方面是历史的原因：在 Linus 的观点看来，通过把内核以单一的方式进行组织并在最初始的空间中运行是相当容易的事情。这种决策避免了有关消息传递体系结构，计算模块装载方式等方面的相关工作。（内核模块系统在随后的几

年中又进行了不断地改进。）

另外一个原因是充足的开发时间的结果。Linux 既没有开发时间的限制，也没有深受市场压力的发行进度。 所有的限制只有并不过分的对内核的修改与扩充。内核的单一设计在内部实现了充分的模块化，在这种条件下的修改或增加都并不怎么困难。而且问题还在于没有必要为了追求尚未证实的可维护性的微小增长而重写Linux的内核。（Linus曾多次特别强调了如下的观点：为了这点利益而损耗速度是不值得的。）后面章节中的部分内容将详细的重新考虑充足开发时间的效果。

如果 Linux 是纯微内核设计，那么向其它体系结构上的移植将会比较容易。实际上，有一些微内核，如 Mach 微内核，就已经成功的证明了这种可移植性的优点。实际的情况是，Linux 内核的移植虽然不是很简单，但也绝不是不可能的：大约的数字是，向一个全新的体系结构上的典型的移植工作需要 30,000 到 60,000 行代码，再加上不到 20,000 行的驱动程序代码。（并不是所有的移植都需要新的驱动程序代码。）粗略的计算一下，我估计一个典型的移植平均需要 50,000 行代码。这对于一个程序员或者最多一个程序小组来说是力所能及的，可以在一年之内完成。虽然这比微内核的移植需要更多的代码，但是 Linux 的支持者将会提出，这样的 Linux 内核移植版本比微内核更能够有效的利用底层硬件，因而移植过程中的额外工作是能够从系统性能的提高上得到补偿的。

这种特殊设计的权衡也不是很轻松就可以达到的，单内核的实现策略公然违背了传统的看法，后者认为微内核是未来发展的趋势。但是由于单一模式（大部分情况下）在 Linux 中运行状态良好，而且内核移植相对来说比较困难，但没有明显地阻碍程序员团体的工作，他们已经热情高涨地把内核成功的移植到了现存的大部分实际系统中，更不用说类似掌上型电脑的一些看起来很不实际的目标了。只要 Linux 的众多特点仍然值得移植，新的移植版本就会不断涌现。

设计和实现的关系

接下来的部分将介绍一些内核设计和实现之间的关系。本部分最重要的内容是对于内核源程序目录结构的概述，这一点随后就会提到。本章最后以实现中体系结构无关代码和体系结构相关代码的相对大小的估算作为总结。

内核源程序目录结构

按照惯例，内核源程序代码安装在/usr/src/linux 目录下。在该目录下还有几个其它目录，每一个都代表一个特定的内核功能性子集（或者非常粗略的说是高层代码模块）。

Documentation

这个目录下面没有内核代码，只有一套有用的文档。但是这些文档的质量不一。有一部分内核文档，例如文件系统，在该目录下有相当优秀而且相当完整的文档；而另外一部分内核，例如进程调度，则根本就没有文档。但是在这里你可以不时的发现自己所最需要的东西。

arch

arch 目录下的所有子目录中都是体系结构相关的代码。每个体系结构特有的子目录下都又至少包含三个子目录：kernel，存放支持体系结构特有的诸如信号量处理和 SMP 之类特征的实现；lib，存放高速的体系结构特有的诸如 **strlen** 和 **memcpy** 之类的通用函数的实现；以及 mm，存放体系结构特有的内存管理程序的实现。

除了这三个子目录以外，大多数体系结构在必要的情况下还都有一个 boot 子目录，该目录中包含有在这种平台上启动内核所使用的部分或全部平台特有代码。这些启动代码中的部分或全部也可以在平台特有的内核目录下找到。

最后，大部分体系结构所特有的目录还可以根据需要包含了供附加特性或改进的组织使用的其它子目录。例如，i386 目录包含一个 math-emu 子目录，其中包括了在缺少数学协处理器（FPU）的 CPU 上运行模拟 FPU 的代码。作为另外一个例子，m68k 移植版本中为每一个该移植版本所支持的基于 680x0 的机器建立了一个子目录，从而这些机器所特有的代码都有一个自然的根目录。

下面几个是 arch 目录下的子目录：

- arch/alpha/——Linux 内核到基于 DEC Alphs CPU 工作站的移植。
- arch/arm/——Linux 到 ARM 系列 CPU 的移植，该类 CPU 主要用于诸如 Corel 的 NetWinder 和 Acorn RiscPC 之类的机器。
- arch/i386/——最接近于 Linux 内核原始平台或标准平台。这是为 Intel 的 80386 结构使用的，当然包括对同一系列后来的 CPU（80486，Pentium 等等）的支持。它还包括了对 AMD，Cyrix 和 IDT 等公司的一些兼容产品的支持。

  本书基本上将这种体系结构称为“x86”。即使这样，严格说来“x86”对于我们的目标来说还是要求得过于宽泛。早期的 Intel CPU，例如 80286，并没有包括 Linux 运行所需的所有特性。对于这些机器，Linux 也没有正式的支持版本。（顺便提一下，Linux 对这种 CPU 的独立移植版本是存在的，不过它在功能上有部分损失。）当本书中提到“x86 平台”时，通常是指 80386 或更新的 CPU。
- arch/m68k/——到 Motorola 的 680x0 CPU 系列的移植。该版本可以提供对基于从 68020（只要它同内存管理单元（MMU）68851 一起使用）到 68060 的一切机器的支持。很多公司在他们的产品中使用 680x0 系列芯片，例如 Commodore（现在是 Gateway）的 Amiga，Apple 的 Macintosh，Atari ST，等等。这些老机器中的很多现在正充当可靠的 Linux 工作站。另外，到 NeXT 工作站和 SUN 3 工作站的移植也正在进行中。
- arch/mips/——到 MIPS 的 CPU 系列的移植。虽然有其它几个厂商也使用 MIPS 开发了一些系统，但是基于这种 CPU 的最出名的机器是 Silicon Graphics（SGI）工作站。
- arch/ppc/——到 Motorola/IBM 的 PowerPC 系列 CPU 的移植。这包括对基于 PowerPC 的 Macintosh 和 Amiga 以及 BeBox、IBM 的 RS/6000 等其它一些机器的支持。
- arch/sparc/——到 32 位 SPARC CPU 的移植。这包括对从 Sun SPARC 1 到 SPARC 20 的全部支持。
- arch/sparc64/——到基于 64 位 SPARC CPU（UltraSPARC 系）系统的移植。这里所能够支持的机器包括 Sun 的 Ultra 1，Ultra 2 和更高配置的机器，直到 Sun 的最新产品 Enterprise 10000。注意 32 位和 64 位的 SPARC 的移植版本正在合并中。

不幸的是，本书必须将注意力集中在 x86 上，因此只应用到了 arch/i386/目录下的代码，而其它体系结构所特有的代码将不再涉及了。

drivers

这个目录是内核中非常大的一块。实际上，drivers 目录下包含的代码占整个内核发行版本代码的一半以上。它包括显卡、网卡、SCSI 适配器、软盘驱动器，PCI 设备和其它任何你可以说出的 Linux 支持的外围设备的软件驱动程序。

Drivers 目录下的一些子目录是平台特有的，例如，zorro 子目录中包含有和 Zorro 总线通讯的代码。而 Zorro 总线只在 Amiga 中使用过，因此这些代码必然是 Amiga 特有的。而其它一些子目录，例如 pci 子目录，则至少是部分平台无关的。

fs

Linux 支持的所有文件系统在 fs 目录下面都有一个对应的子目录。一个文件系统（file system）是存储设备和需要访问存储设备的进程之间的媒介。

文件系统可能是本地的物理上可访问的存储设备，例如硬盘或 CD-ROM 驱动器；在这两种情况下将分别使用 ext2 和 isofs 文件系统。文件系统也可能是可以通过网络访问的存储设备；这种情况下使用的文件系统是 NFS。

还有一些伪文件系统，例如 proc 文件系统，可以以伪文件的形式提供其它信息（例如，在 proc 的情况下是提供内核的内部变量和数据结构）。虽然在底层并没有实际的存储设备与这些文件系统相对应，但是进程可以像有实际存储设备一样处理（NFS 也可以作为伪文件系统来使用）。

include

这个目录包含了 Linux 源程序树中大部分的包含（.h）文件。这些文件按照下面的子目录进行分组：

- include/asm-*/——这样的子目录有多个，每一个都对应着一个 arch 的子目录，例如 include/asm-alpha，include/asm-arm，include/asm-i386 等等。每个目录下的文件中包含了支持给定体系结构所必须的预处理器宏和短小的内联函数。这些内联函数很多都是全部或部分地使用汇编语言实现的，而且在 C 或者汇编代码中都会应用到这些文件。

  当编译内核时，系统将建立一个从 include/asm 到目标体系结构特有的目录的符号链接。结果是体系结构无关内核源程序代码可以使用如下形式的代码来实现所需功能：

  ```
  #include <asm/some-file>
  ```

  这样就能够将适当地体系结构特有的文件包含（**#include**）进来。
- include/linux/——内核和用户应用程序请求特定内核服务时所使用的常量和数据结构在头文件中定义，而该目录中就包含了这些头文件。这些文件大都是平台独立的。这个目录被全部复制（更多的情况是链接）到/usr/inlude/linux 下。这样用户应用程序就可以使用#include 包含这些头文件，而且能够保证所包含进来的头文件的内容和内核中的定义一致。第 9 章将会给出有关的一个样例。
- 对这些文件的移植只有对于内核来说才是必须的，对用户应用程序则没有必要。移植工作可以按照如下的方式封装处理：

  ```
  /* …   Stuff for user apps and kernel … */
  #ifdef   __KERNEL__

  /* … Stuff for kernel only … */
  #endif   /* __KERNEL__ */
  ```
- include/net/——这个目录供与网络子系统有关的头文件使用。
- include/scsi/——这个目录供与 SCSI 控制器和 SCSI 设备有关的头文件使用。
- include/video/——这个目录供与显卡和帧显示缓存有关的头文件使用。

init

这个目录下面的两个文件中比较重要的一个是 main.c，它包含了大部分协调内核初始化的代码。第 4 章将详细介绍这部分代码。

ipc

这个目录下的文件实现了 System V 的进程间通讯（IPC）。在第 9 章中将会对它们进行详细介绍。

kernel

这个目录中包含了 Linux 中最重要的部分：实现平台独立的基本功能。这部分内容包括进程调度（kernel/sched.c）以及创建和撤销进程的代码（kernel/fork.c 和 kernel/exit.c）；以上所有的以及其它部分内容将在第 7 章中有所涉及。但是我并不想给你留下这样的印象：需要了解的内容都在这个目录下。实际上在其它目录下也有很多重要的内容。但是，不管怎样说，最重要部分的代码是在这个目录下的。

lib

lib 目录包含两部分的内容。lib/inflate.c 中的函数能够在系统启动时展开经过压缩的内核（请参看第 4 章）。lib 目录下剩余的其它文件实现一个标准 C 库的有用子集。这些实现的焦点集中在字符串和内存操作的函数（**strlen**，**mmcpy** 和其它类似的函数）以及有关 **sprintf** 和 **atoi** 的系列函数上。

这些文件都是使用 C 语言编写的，因此在新的内核移植版本中可以立即使用这些文件。正如本章前面部分说明的那样，一些移植提供了它们独有的高速的函数版本，这些函数通常是经过手工调整过的汇编程序，在移植后的系统使用这些函数来代替原来的通用函数。

mm

该目录包含了体系结构无关的内存管理代码。正如我们前面说明的那样，为每个平台实现最低层的原语的体系结构特有的内存管理程序是存储在 arch/platform/mm 中的。大部分平台独立和 x86 特有的内存管理代码将在第 8 章中介绍。

net

这个目录包含了 Linux 应用的网络协议代码，例如 AppleTalk，TCP/IP，IPX 等等。

scripts

该目录下没有内核代码，它包含了用来配置内核的脚本。当运行 **make menuconfig** 或者 **make xconfig** 之类的命令配置内核时，用户就是和位于这个目录下的脚本进行交互的。

体系结构相关和体系结构无关的代码

现在我们来估计一下体系结构相关和体系结构无关代码的相对大小。我们首先给出一些数字。完整的 2.2.5 的内核总共有 1,725,645 行代码。（顺便一提，请注意本书只包含了 39,000 行代码，但是我们仍然努力涵盖了相当部分的核心函数。）其中一共有 392,884 行代码在体系结构特有的目录之内，也就是 arch/*和 include/asm-*下面。我估计还有超过 64,000 行的代码是仅供一种体系结构专用的驱动程序。这意味着大约 26%的代码是专用于特定体系结构的。

但是，对于单一一种体系结构，体系结构相关代码比例相对较小。不妨理想一点，如果某种体系结构所需要的特有代码约有 50,000 行，而体系结构无关代码则大约有 1,250,000 行，那么体系结构相关代码大概只占到 4%。当然，在特定的一个内核中，并不是所有这些体系结构无关代码都会被用到，因此体系结构相关代码在特定内核中所占的比重与内核的配置有关。但是不管怎样，很显然大部分内核代码是平台独立的。

第 4 章　系统初始化

当你想要运行程序时，你需要把程序的文件名敲入 shell－－或者更为流行的，在如 GNOME 或者 KDE 等之类桌面环境中点击相应的图标－－这样就能将其装载进内核并运行。但是，首先必须有其它的软件来装载并运行内核；这通常是诸如 LOADLIN 或者 LILO 之类的内核引导程序。更进一步，我们还需要其它的软件来装载运行内核引导程序－－称之“内核引导程序的引导程序”－－而且看起来似乎运行内核引导程序的引导程序也需要内核引导程序的引导程序的引导程序，等等，这个过程是无限的。

这个无限循环的过程必然最终在某个地方终止，这就是硬件。因此，在最低的层次上，启动系统的第一步是从硬件中获得帮助。该硬件总是运行一些短小的内置程序－－软件，但是这些软件是被固化在只读存储器中，存储在已知地址中。因此，在这种情况下就不需要软件引导程序了－－它能够运行更大更复杂的程序，直到内核自身装载成功为止。按照这种方式，系统自己的引导过程（bootstrap）会引发系统的启动，当然这只是术语“系统引导（booting）”的一个比喻。虽然不同体系结构的引导过程的具体细节差异很大，但是它们的原则都基本相同。

前面的工作都完成以后，内核就已经成功装载了。随后内核可以初始化自身以及系统的其它部分。

本章首先将简单介绍基于 x86 PC 机的典型自启动方式，接着回顾一下每一步工作在什么时机发生，最后我们还要介绍的是内核的相应部分。

引导 PC 机

本节简要介绍 x86 PC 是如何引导的。本节的目的不是让你精通 PC 是怎样引导的－－这超出了本书的范围－－而是向你展示特定体系结构一般的引导方式，为下文中的内核初始化进行铺垫。

首先，机器中的每个 CPU 都要自行初始化，接着可能要用几分之一秒的时间来执行自测试。在多处理器的系统中，这个过程会更复杂些－－但是实际上也并不多。在双处理器的 Pentium 系统中，一个 CPU 总是作为主 CPU 存在，另外一个 CPU 则是辅 CPU。主 CPU 执行启动过程中的剩余工作，随后内核才会激活辅 CPU。在多处理器的 Pentium Pro 系统中，CPU 必须根据 Intel 定义的算法“抢夺标志”－－来动态决定由哪个 CPU 启动系统。取得标志的 CPU 启动系统，随后内核激活其它的 CPU。无论是哪种情况，启动程序的剩余部分只与一个 CPU 有关。这样，在随后的一段时间内，我们可以认为该系统中只有一个 CPU 是可用的，而不考虑其它的 CPU，或者说这些 CPU 被暂时隐藏了。另一方面，内核还需要明确的激活所有其它的 CPU－－这一点你可以在本章后续部分看到。

接下来，CPU 从 0xfffffff0 单元中取得指令并执行，这个地址非常接近于 32 位 CPU 的最后可用的地址。因为大多数 PC 都没有 4GB 的 RAM，所以通常在这个地址上并没有实际内存的。内存硬件可以虚拟使用它。对那些确实有 4GB 内存的机器来说，它们也只是仅仅损失了供 BIOS 使用的顶端地址空间末尾的少量内存（实际上 BIOS 在这里只保留了 64K 的空间－－这种损失在 4GB 的机器中是可以忽略的）。

该地址单元中存储的指令是一条跳转指令，这条指令跳转到基本输入输出（BIOS）代码的首部。BIOS 内置在主板中，它主要负责控制系统的启动。请注意 CPU 实际上并不真正关心 BIOS 是否存在，这样就使得在诸如用户定制的嵌入系统之类的非 PC 体系结构的计算

机中使用 Intel 的 CPU 成为可能。CPU 执行在目标地址中发现的任何指令，在这里使用跳转指令转移到 BIOS 只是 PC 体系结构的一部分。（实际上，跳转指令自己是 BIOS 的一部分，但是这不是考虑这个问题的最方便的方法。）

BIOS 使用内置的规则来选择启动设备。通常情况下，这些规则是可以改变的，方法是在启动过程开始时按下一个键（例如，在我的系统中是 Delete 键）并通过一些菜单选项浏览选择。但是，通常的过程是 BIOS 首先试图从软盘启动，如果失败了，就再试图从主硬盘上启动。如果又失败了，就再试图从 CD-ROM 上启动。为了使问题更具体，这里讨论的情况假定是最普通的，也就是启动设备是硬盘。

从这种启动设备上启动，BIOS 读取第一个扇区的信息－－首 512 个字节－－称之为主引导记录（MBR）。接下来发生的内容有赖于 Linux 是怎样在系统上安装的。为使讨论形象具体，我们假定 LILO 是内核的载入程序。在典型的设置中，BIOS 检测 MBR 中的关键数字（为了确认该数据段的确是 MBR）并在 MBR 中检测引导扇区的位置。这一扇区包含了 LILO 的开始部分，然后 BIOS 将其装入内存，开始执行。

注意我们现在已经实现了从硬件和内置软件的范围到实际软件范围的转变，从有形范围到无形范围，也就是说从你可以接触的部分到不可接触的部分。

下面就是 LILO 的责任了。它把自己其余的部分装载进来，在磁盘上找到配置数据，这些数据指明从什么地方可以得到内核，启动时要通过什么选项。LILO 接着装载内核到内存并跳转到内核。

通常，内核以压缩形式存储，只有少量指令足以完成解压缩的任务，也就是自解压可执行文件，是以非压缩形式存储的。因此，内核的下一步工作是自解压缩内核镜像。到这里，内核就已经完成了装载的过程。

下面是到现在进行的步骤地简要描述：

1. CPU 初始化自身，接着在固定位置执行一条指令。
2. 这条指令跳转到 BIOS 中。
3. BIOS 找到启动设备并获取 MBR，该 MBR 指向 LILO。
4. BIOS 装载并把控制权转交给 LILO。
5. LILO 装载压缩内核。
6. 压缩内核自解压并把控制权转交给解压的内核

正如你所见到的，引导过程每一步都将你带入更大量更复杂的代码块中，一直到最后成功地运行了内核为止。

依赖于你计算层次的方式，CPU 成为内核引导程序的引导程序的引导程序的引导程序（CPU 装载 BIOS，BIOS 装载 LILO，LILO 装载压缩内核，压缩内核装载解压内核；但是你可以合理的考虑是否这些步骤都满足引导程序的定义）。

初始化 Linux 内核

在内核成功装入内存（如果需要就解压缩）以及一些关键硬件，例如已经在低层设置过的内存管理器（MMU，请参看第 8 章）之后，内核将跳转到 **start_kernel**（19802 行）。这个函数完成其余的系统初始化工作－－实际上，几乎所有的初始化工作都是由这个函数实现的。因此，**start_kernel** 就是本节的核心。

start_kernel

19802：**__init** 标示符在 gcc 编译器中指定将该函数置于内核的特定区域。在内核完成自身初始化之后，就试图释放这个特定区域。实际上，内核中存在两个这样的区

域，.text.init 和.data.init－－第一个是代码初始化使用的，另外一个是数据初始化使用的。（诸如可以在进程间共享的代码和字符串常量之类的“文本（Text）”是在可执行程序中的“纯区域”中使用的一个术语。）另外你也可以看到**__initfunc** 和**__initdata** 标志，前者和**__init** 类似，标志初始化专用代码，后者则标志初始化专用数据。

19807：如前所述，即使在多处理器系统中，在启动时也只使用一个 CPU。Intel 称之为引导程序处理器（Bootstrap Processor，简称为 BSP），它在内核代码的某些地方有时也称之为 BP。BSP 首次运行这一行时，跳过后面的 **if** 语句，并减小 **boot_cpu** 标志，从而当其它 CPU 运行到此处时，都要运行 if 语句。等到其它 CPU 被激活执行到这里时，BSP 已经在 idle 循环中了（本章稍后会更详细的讨论这个问题），**initialize_secondary**（4355 行）负责把其它 CPU 加入到 BSP 中。这样，其它 CPU 就不用执行 **start_kernel** 的剩余部分了－－这也是一件好事，因为这意味着不用再进行对许多硬件进行冗余初始化等工作了。

顺便说一下，这种奇异的小小的改动只有对于 x86 是必需的；对于其它平台，调用 **smp_init** 完全可以处理 SMP 设置的其它部分，这一点马上就会讨论。因此，其它平台的 **initialize_secondary** 的定义都是空的。

19816：打印内核标题信息（20099 行），这里显示了有关内核如何编译的信息，包括在什么机器上编译，什么时间编译，使用什么版本的编译器，等等。如果中间任何一步发生了错误，在寻找机器不能启动的原因时查明内核的来源是一个有用的线索。

19817：初始化内核自身的部分组件－－内存，硬件中断，调度程序，等等。尤其是 **setup_arch** 函数（19765 行）完成体系结构相关的设置，此后在 **command_line**（传递到内核的参数，在下面讨论）、**memory_start** 和 **memory_end**（内核可用物理地址范围）中返回结果。下面这些函数都希望驻留在内存低端的；它们使用 **memory_start** 和 **memory_end** 来传递该信息。在函数获得所希望的值后，返回值指明了新的 **memory_start** 的值。

19823：分析传给内核的各种选项。**parse_options** 函数（19707 行，在随后的分析内核选项一节中讨论）也设置了 **argv** 和 **envp** 的初值。

19833：内核运行过程中也可以自行对所进行的工作进行记录，周期性地对所执行的指令进行抽样，并使用所获得的结果更新表格。这在定时器中断过程中通过调用 **x86_do_profile**（1896 行）来实现，该部分将在第 6 章中介绍。

如图 4.1 中说明的那样，这个表格把内核划分为几个大小相同的范围，并简单跟踪在一次中断的时间内每个范围中运行多少条指令。这种记录当然是非常粗糙的－－甚至不是依据函数和行号进行划分的，而只是使用近似的地址－－但是这样代价很低、快速、短小，而且有助于专家判断最关键的问题要点。每个表格条目所涉及到地址的多少－－还有问题发生地点的不确定性－－可以通过简单修改 **prof_shift**（26142 行）来调节。**profile_setup**（19076 行，在本章中后面讨论）可以让你在启动的时候设置 **prof_shift** 的值，这样比为修改这个数字而重新编译内核要清晰方便得多。

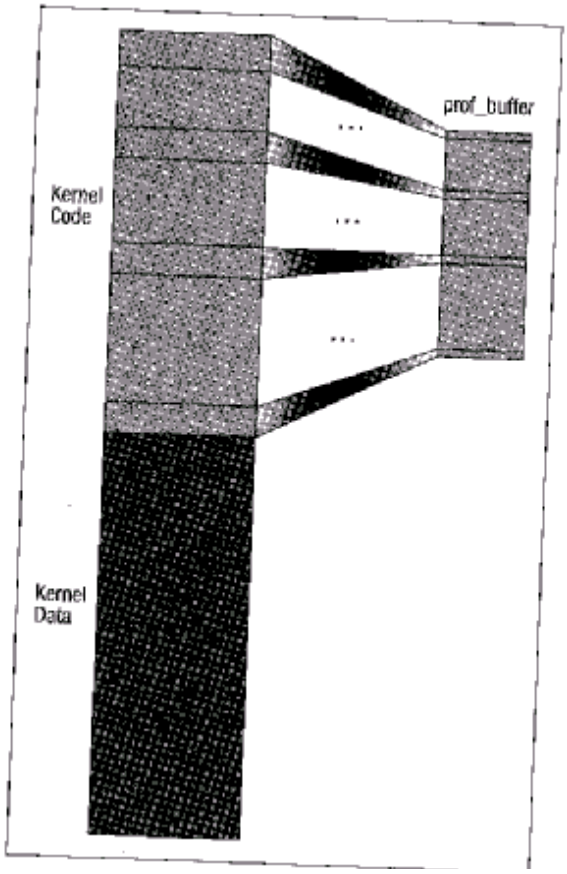

图 4.1 描述用缓存（profiling buffer）

这个 **if** 程序块为记录表格分配内存，并把所有项都清零。注意到如果 **prof_shift** 是 0（缺省值），那么记录功能就被关掉了，**if** 程序段不再被执行，也不为表格分配空间。

19846：内核通过调用 **sti**（13104 行是 UP 版本的－－注意该主题在第 6 章中有更详细的介绍）开始接收硬件中断。首先需要激活定时器中断，以便后来对 **calibrate_delay**（19654 行）的调用可以计算机器的 BogoMIPS 的值（在下一节“BogoMIPS”中介绍）。因为一些设备驱动程序需要 BogoMIPS 的值，所以内核必需在大部分硬件、文件系统等等初始化之前计算出这个值来。

19876：测试该 CPU 的各种缺陷，比如 Pentium F00F 缺陷（请参看第 8 章），记录检测到的缺陷，以便于内核的其它部分以后可以使用它们的工作。（为了节省空间起见，我们省略掉了 **check_bugs** 函数。）

19882：调用 **smp_init**（19787 行），它又调用了其它的函数来激活 SMP 系统中其它 CPU：在 x86 的平台上，**smp_boot_cpus**（4614 行）初始化一些内核数据结构，这些数据结构跟踪检测另外的 CPU 并简单的将其改为保持模式；最后 **smp_commence**（4195 行）使这些 CPU 继续执行。

19883：把 init 函数作为内核线程终止，这比较复杂；请参看本章后面有关 **init** 的讨论。

19885：增加 idle 进程的 **need_resched** 标志，这样做的原因在此时可能还比较模糊。直到读完了第 5、6、7 章以后，才能有个清楚的概念；但是，在下一个定时器中断结束之前（在第 6 章中讨论），**system_call**（171 行，在第 5 章中讨论）函数中会注意到 idle 进程的 **need_fesched** 标志增加了，并且调用 **schedule**（26686 行，第 7 章）释放 CPU，并将其赋给更应该获取 CPU 的进程。

19886：已经完成了内核初始化的工作——或者不管怎样，已经把需要完成的少量责任传递给了 **init**——所剩余的工作不过是进入 idle 循环以消耗空闲的 CPU 时间片。因此，本行调用 **cpu_idle**（2014 行）——idle 循环。正如你可以从 **cpu_idle** 本身可以发现的一样，该函数从不返回。然而，当有实际工作要处理时，该函数就会被抢占。

注意到 **cpu_idle** 只是反复调用 **idle** 系统调用（下一章将讨论系统调用），它通过 **sys_idle**（2064 行）实现真正的 idle 循环——2014 行对应 UP 版本，2044 行针对 SMP 版本。它们通过执行 **hlt**（对应“halt”）指令把 CPU 转入低功耗的“睡眠”状态。只要没有实际的工作处理，CPU 都将转入这种状态。

BogoMIPS

BogoMIPS 的数字由内核计算并在系统初始化的时候打印。它近似的给出了每秒钟 CPU 可以执行一个短延迟循环的次数。在内核中，这个结果主要用于需要等待非常短周期的设备驱动程序——例如，等待几微秒并查看设备的某些信息是否已经可用。

由于没有正确理解 BogoMIPS 的含义，BogoMIPS 在各处都被滥用，就仿佛它可以满足人类最原始、最深层次的需求：把所有计算机性能的信息简化为一个数字。“BogoMIPS”中的“Bogo”部分来源于“伪（bogus）”，就正是为了防止这种用法：虽然这个数字比大多数性能比较有效很多，但是它仍然是不准确的、容易引起误解的、无用的和不真实的，根本不适合将它用于机器间差别的对比。但是这个数字仍然非常吸引人，这也正是我们在这里讨论这个问题的原因。（顺便说一下，BogoMIPS 中“MIPS”部分是“millions of instructions per second（百万条指令每秒）”的缩写，这是计算机性能对比中的一个常用单位。）

calibrate_delay

19654：**calibrate_delay** 是近似计算 BogoMIPS 数字的内核函数。

19622：作为第一次估算，**calibrate_delay** 计算出在每一秒内执行多少次**__delay** 循环（6866 行），也就是每个定时器滴答（timer tick）——百分之一秒——内延时循环可以执行多少次。

19664：计算一个定时器滴答内可以执行多少次循环需要在滴答开始时就开始计数，或者应该尽可能与它接近。全局变量 **jiffies**（16588 行）中存储了从内核开始保持跟踪时间开始到现在已经经过的定时器滴答数；第 6 章中将介绍它的实现方式。**jiffies** 保持异步更新，在一个中断内——每秒一百次，内核暂时挂起正在处理的内容，更新变量，然后继续刚才的工作。如果不这样处理，下一行的循环就永远不可能退出。从而，如果 **jiffies** 不声明为 **volatile**——简单的说，这个值变化的原因对于编译器是透明的——gcc 仍然可能对该循环进行优化，并引起该循环进入不能退出的状态。虽然目前的 gcc 还没有如此高的智能，然而它的维护者应该完全能够为它实现这种智能。

19669：定时器又前移了一个滴答，因此又产生一个新的滴答。下一步是要等待 **loops_per_sec** 延时循环调用定时器循环，接着检测是否最少有一个完整的滴答已经完成。如果是这样，就退出首次近似估算循环；如果没有，就把 **loops_per_sec** 的值加倍，然后重新启动这个过程。

这个循环的正确性依赖于如下的事实：现有的机器在任何地方都不能每秒执行 2^{32} 次延时循环——对于 64 位机来说则远低于每秒 2^{64} 次——虽然这只是一个微不足道的问题。

19677：现在内核已经清楚 **loops_per_sec** 循环调用延时循环在这台机器上要花费超过百分之

一秒的时间才能完成，因此，内核将重新开始进行估算。为了提高效率，内核使用折半查找算法计算 **loops_per_sec** 的实际值，我们假定开始的时候，实际值在现在计算结果和其一半之间——实际值不可能比现在计算值还大，但是可以（而且可能）稍微小一点。

19681：和前面使用的方式一样，**calibrate_delay** 查看是否这个 **loops_per_sec** 已经减小了的值还是比较大，需要耗费一个完整的定时器间隔。如果还是相当大，实际值应该小于当前计算值或者就是当前值，因此，使用更小的值继续查询；如果不够大，就使用一个更大的值继续查询。

19691：内核有一种很好的方法来计算一个定时器滴答中执行延时循环的次数。这个数字乘以一秒内滴答的数量就得到了每秒内可以执行的延时循环的次数。这种计算只是一种估算，乘法也累积了误差，因此结果并不能精确到纳秒。但是这个数字供内核使用已经足够精确了。

19693：为了让用户感到激动，内核打印出这个数字。注意这里明显省略了%f 的格式限定——内核尽量避免浮点数运算。这个计算过程中最有用的常量是 500,000；它是用一百万除以 2 得来，理由是每秒钟执行一百万条指令，而每个 delay 循环的核心是 2 条指令（**decl** 和一条跳转指令）。

分析内核选项

parse_options 函数分析由内核引导程序发送给内核的启动选项，在初始化过程中按照某些选项运行，并将剩余部分传送给 init 进程（在本章后面部分提到）。这些选项可能已经存储在配置文件中了，也可能是由用户在系统启动时敲入的——内核并不关心这些。类似的细节全部是内核引导程序应该关注的内容。

parse_options

19707：参数已经收集在一条长的命令行中，内核被赋给指向该命令行头部的一个指针；内核引导程序在前面已经将该行存储在一个指定地址中。

19718：中断下一个参数，保持指向下一个参数的指针以供下一次循环使用。注意系统使用空格而不是通常的空白来分隔内核参数；制表符并不能把当前参数和下一个参数分隔开。如果发现了分隔字符空格，下一行就使用字节 0 覆盖，这样 **line** 可以作为包含有唯一一个内核选项的标准 C 字符串来使用了。如果没有发现空格，就该函数关心的内容而言，其余的部分都具有相同的属性——这只有在处理 **line** 中最后一个选项的情况下才会发生，循环就会在下次开始时结束。

注意该代码不会跳过多个空格。假设 **line** 值如下所述（两个空格）：

rw debug

这会被当作三个选项：“rw”，“”（空字符串）和“debug”。因为空字符串不是有效的内核选项，它将会被传递到初始化的过程（这一点随后就可以看到）——这当然不是用户所希望的。因此，内核引导程序应该负责对多个空格进行压缩。LILO 通过忽略用户多敲的空格，完美的解决了这个问题。

19721：现在开始解释这些选项。最前面的两个选项——**ro** 和 **rw**——指明内核要装载根文件系统，也就是根目录（ / 目录）所在的位置，而分别处于只读和读/写模式。

19729：第三种可能性，**debug**，增加了调试信息的数量；这些调试信息要通过调用 **do_syslog** 打印出来（25724 行）。

19733：开始几个选项是简单的独立标志，它们并不使用参数。内核也可以辨认形为

option=value 的选项。本行就是一个例子，这里内核引导程序定义了一个命令来代替 init 运行；它使用 **init=*/some/other/program*** 的形式。这里的代码舍弃了 **init=** 部分，为随后 **init** 的使用而把剩余部分在 **execute_command** 中保存起来（20044 行，后面会讨论到）。和其它大部分参数的处理方法不同，本处功能不能在 **checksetup**（19612 行，马上就讨论到）中实现，这是因为它改变了该函数的局部变量。很快，你就可以看到前面三个选项之所以也在这里处理而不是在 **checksetup** 中处理的原因。

19745：大部分内核选项都是由 **checksetup** 函数分析的。如果 **checksetup** 处理了某个选项，就返回真值，循环继续进行。

19750：否则，**line** 中没有已经被辨认的内核选项。在这种情况下，它被作为一个供 init 进程使用的选项或者环境变量来处理——如果其形式为 **envar=*value***，就作为环境变量处理；否则，就作为选项处理。如果 **argv_init** 和 **envp_init**（分别见 19057 和 19059 行）数组中有足够的空间，选项和环境变量就存储在里面供以后 **init** 函数使用。

这解释了从 19736 行开始的注释。字符串 **auto** 并不是任何内核选项的前缀，因此它应该被作为 init 的一个参数存储在 **argv_init** 数组中——这在大多数情况下都是可行的，因为 **auto** 是 init 可以识别的选项。但是，当使用 **init=**的形式给出内核选项时，通常是执行 shell 而不是 init，**auto** 会使 shell 混淆；因此，安全一点的方法是，**parse_options** 在此处忽略所有与此有关的 init 参数。

奇怪的是，当 **argv_init** 或者 **envp_init** 空间用完时，整个循环就结束了。仅仅因为 **argv_init** 的空间用完了并不意味着 line 中就不再含有 init 使用的环境变量，反之亦然。此外，可能还剩下许多内核选项没有处理。当你考虑到 **MAX_INIT_ARGX**（19029 行）和 **MAX_INIT_ENVS**（19030 行）都通过使用**#define** 被预定义为 8——这是一个很容易超过的下限——这种行为就更奇怪了。如果在 19752 行和 19756 行的 **break** 改成 **continue**，那么循环可以继续处理内核选项，而不会写入超过 **argv_init** 和 **envp_init** 数组界限的空间。如果 **command_line** 中仍然包含有并不是为 init 而定义的内核选项，那么这一点就是非常重要的。

19760：所有的内核选项都处理完成了。最后一步是要使用 **NULL** 填充 **argv_init** 和 **envp_init** 数组的末尾，从而使得 init 可以知道在哪里终止。

checksetup

19612：**checksetup** 函数负责进行大部分内核选项的处理过程。它把这些内核选项分为三类：一类使用内核普通参数来分析=sign 之后的部分；另一类自行分析=sign 之后的部分；还有一类自行分析整个行，包括= sign 前面的部分和= sign 后面的部分。第一类被认为是使用“现成”的参数，这与为第二类提供的“原始”参数相对应。最后一类只由一个 IDE 驱动程序组成；内核首先在 19619 行检查并处理这种情况，以使其不会在随后的处理中造成麻烦。

19625：接下来，**checksetup** 扫描整个 **raw_params** 数组（19552 行）并判断是否该内核选项应该不加处理的保留。**raw_params** 中的元素是 **struct kernel_param** 类型（19223 行）的，它把内核选项前缀和装载选项时调用的函数联系起来。如果数组中的某些项的 **str** 成员以 **line** 为前缀，就会调用 **line** 后面的相应函数（也就是前缀之后的部分），随后 **checksetup** 会返回一个非零值以表明它已经对该内核选项进行了处理。**raw_params** 数组以两个 **NULL** 结束，因此在检测到 **str** 成员是 **NULL** 时，循环就可以结束了。在这种情况下，显然循环已经到达了 **raw_params** 数组的结尾，但是仍然没有找到匹配的情况。当然，测试 **setup_func** 成员也可以取得同样好的效果。

这个循环说明了一点：与大多数内核非常不同的是，这里的初始化并不需要尽可能

的快。如果内核比从前多用几微秒来启动，这并没有什么实际的损失——毕竟用户应用程序还没有开始运行，所以他们并没有损失什么东西。

最终结果是代码效率很低，而且存在很多优化的可能。例如，**raw_params** 数组中字符串的长度可以在 **raw_params** 中暂存，而不用在 19626 行多次重复计算。更好的解决方法是，可以把 raw_params 数组中的项按照字符顺序排序，这样 **checksetup** 就可以进行折半查找。

在 **raw_params** 的情况中实现排序并没有什么障碍，但是这样也可能并不能获得很大的优势，因为折半查找的优点只有在比较大的数组中才能充分表现出来（所谓比较大的确切值在不同的环境中也有所不同）。**raw_params** 的姊妹数组 **cooked_params**（19228 行）当然是足够大的，可以显示出折半查找的优势；但是这样就引发了一个新的问题：对 **cooked_params** 进行排序比较难用，因为这可能需要分隔一些**#ifdef** 程序段——请参看从 19268 行到 19272 行的例子。进一步说，因为算法只是查找前缀，而不使用完全匹配，在遍历数组中的各个项时对遍历次序比较敏感，所以这种特性在使用不同的查找次序时就很难再保持了。然而，这些问题并不是不可克服的（程序员可以预先静态地为引导程序建立一颗前缀树），如果性能在这里是主要因素，那么这种努力也是值得的。但是，由于性能在这里并不是主要问题，所以简单性才被作为最重要的因素体现出来。

即使这样，在类似的 **root_dev_names** 数组（19085 行）中——这个数组把硬件设备名的前缀映射到它们的主 ID 号上——开发者仍然可以简单地通过把比较常用的项（IDE 和 SCSI 磁盘）放在不太常用的项（串口 IDE CDs）的前面以节省出一点性能。但是我在 **raw_params** 或 **cooked_params** 中并没有发现与之类似的模式。

另外一件需要注意的事是：现在你可以猜想一下为什么 **ro**，**rw** 和 **debug** 选项在 **parse_options** 中测试而不在这里测试——**parse_options** 要检测精确的匹配，但是 **checksetup** 只检测前缀。作为一个特殊的情况，**ro** 选项碰巧正好是 **root=**（19553 行）的前缀，这样如果这三个选项彼此合并，就需要仔细处理了。这似乎仍然是一个相当无力的原因。考虑一下 **noinitrd** 选项（19251 行）。这是 **cooked_params** 的一个项，因而只需要匹配前缀，而且与之相关联的设置函数（**no_initrd**，19902 行）将忽略所有可能已经传递给它们的参数——这正像 **ro**，**rw** 和 **debug** 被包含在 **cooked_params** 中时所可能进行的工作一样。

19632：这个循环为 **cooked_params** 数组的处理工作和前面一个循环为 **raw_params** 数组的处理工作相同。这两个循环（当然不包括循环使用的数组）间的唯一区别是本循环在调用设置函数之前，使用 **get_options**（19062 行）处理 **line** 中=sign 后面的部分。简单的说，**get_options** 使用 10 个负整数填充 **ints[1]**到 **ints[10]**。**ints[0]**中是 **ints** 中使用元素的个数——也就是，它记录了存储在 **ints** 中的 **intsget_options** 数量。接着这个数组将被传递给设置函数，该设置函数则会按照自己喜欢的方式对该数组内容进行解释。

19640：返回 0，说明 **line** 中所包含的内核选项不能被函数理解。

profile_setup

19076：**profile_setup** 是 **checksetup** 调用的设置函数的一个完美的例子：这个函数十分短小，使用 **ints** 参数处理了部分内容。而且到目前为止你也应该对它的目的有了一定了解。正如前面提到的一样，用户可以在启动的时候设置 **prof_shift** 的值——好，这里正是它的实现方式。当内核启动过程提供 **profile=**选项时，就调用 **profile_setup** 函数。前缀字符串和函数在 19235 行被联系在一起。注意这是在 **cooked_params** 中，因此 **profile_setup** 取得的是处理过的参数。

19079：如果参数中存在 **profile=**的形式，就使用 **profile=**后面的第一个数字作为 **prof_shift** 的新值。选项给出的其它参数都被简单的忽略了。

19081：如果给出了 **profile=**选项，但是没有为它提供参数，**prof_shif**t 的缺省值就是 2。这个缺省值有些奇怪，因为我们已经知道，这意味着使用四分之一的内核可用内存来配置其余部分－－这是一个很大的开销。但是另一方面，使用这些内存有助于更精确的定位问题热点－－只有很少的几条指令存在不确定性，这样应该比较容易地把问题限制在一两行源程序代码内。那张图也并不是像我所画的那样简单：因为图中只描述了内核代码，这种开销还不到内核所有内存空间的 25%，但是对于所覆盖的代码量来说却并不止 25%。

init

init 从许多方面看都是一个非常特殊的进程。这是内核运行的第一个用户进程，它要负责触发其它必需的进程以使系统作为一个整体进入可用的状态。这些工作由/etc/inittab 文件控制，通常包括设置 getty 进程以接受用户登录；建立网络服务，例如 FTP 和 HTTP 守护进程；等等。如果没有这些进程，用户就不可能完成多少工作，这样成功启动内核就显得没有多大意义了。

这种设计的另外一个重要的副作用是 init 是系统中所有进程的祖先。init 产生 getty 进程，getty 进程产生 login 进程，login 进程产生你自己的 shell，使用自己的 shell，可以产生每一个你运行的进程。在所有的结果中，这有助于确保内核进程表中的所有项最终都能够得到处理。进程结束以后将其清除（回收）的工作首先应由其父进程完成；如果父进程已经退出，那么祖父进程就要担负起这种责任；如果祖父进程已经退出，那么曾祖父进程就要担负起这种责任，周而复始。通过这种方式，从不退出的 init 进程就可能要负责回收其它进程。

因此，为了确保这些重要的工作都能正确执行，内核初始化进程所需要做的最后一步工作就是创建 init 进程，接下来就加以描述。

init

20044：**unused** 参数来源于该函数的非常规调用。**init** 函数－－不要和 *init 进程*搞混了，后者是它随后要创建的－－作为内核线程开始生命周期，一个作为内核的一部分运行的进程。（如果你编写过多线程的程序，这里的内核线程可能会同你所已经知道的线程意义有所不同－－在那种意义下，它不是一个内核线程。）实际上，**init** 函数就像是新进程使用的剥离出来了的 **main** 函数，**unused** 参数是一个独立的指针，其值指向为给定进程所提供的信息－－这比通常使用 **argc**，**argv** 和 **envp** 参数传递的信息要少得多。**init** 函数碰巧不需要额外的信息，因此这个参数命名为 **unused**，就是要强调这一点。

为了确保在这一点上你不会产生困惑，我们在这里再对整个机制进行扼要重复：**init** 函数是内核的一部分；它在内核中作为内核的一个独立的执行部分运行；也就是说，无论从哪个方面看它都是内核代码。但是，init 进程就不是这样了。在某些方面，init 进程是一个特殊的进程，但是不属于内核本身；其代码存储在磁盘上单独的可执行映像中，这和其它程序一样。因为 **init** 函数后来产生 init 进程，而它自己又恰好作为进程运行，这样就很容易产生混淆。

因为 idle 进程已经占据了进程 ID 号（PID）0，**init**（当然是 init）就被赋值为下一个可用的 PID，也就是 1。（进程 ID 在第 7 章中讨论。）内核重复假定 PID 为 1 的进程是 init，因此这种特性在没有充分地相互作用，也就是没有同步地进行修改的情况

下是不能改变的。

20046：调用 **lock_kernel**（17492 行对应 UP 版本；10174 行对应 SMP 版本）执行后续几行，而不会受到其它会受到随后工作的影响的内核模块的干扰。内核锁随后在 20053 行被释放。

20047：调用 **do_basic_setup**（19916 行）初始化总线并随同其它工作产生一些其它内核线程。

20052：内核已完全完成初始化了，因此 **free_initmen**（7620 行）可以舍弃内核的.text.init 节的函数和.data.init 节的数据。所有使用**__initfunc** 标记过的函数和使用**__initdata** 标记过的数据现在都不能使用了，它们曾经获得的内存现在也可能重新用于其它目的了。

20055：如果可能，打开控制台设备，这样 init 进程就拥有一个控制台，可以向其中写入信息，也可以从其中读取输入信息。实际上 init 进程除了打印错误信息以外，并不使用控制台，但是如果调用的是 shell 或者其它需要交互的进程，而不是 init，那么就需要一个可以交互的输入源。如果成功执行 **open**，/dev/console 就成为 init 的标准输入源（文件描述符 0）。

20059：调用 **dup** 打开/dev/console 文件描述符两次，这样，init 就也使用它供标准输出和标准错误使用（文件描述符 1 和 2）。假设 20055 行的 **open** 成功执行（正常情况），init 现在就有三个文件描述符－－标准输入、标准输出以及标准错误－－全都加载在系统控制台之上。

20067：如果内核命令行中给出了到 init 的直接路径（或者别的可替代的程序），现在就试图执行 **init**。

因为当 **execve** 成功执行目标程序时并不返回，只有当前面的所有处理过程都失败时，才能执行相关的表达式。接下来的几行在几个地方查找 init，按照可能性由高到低的顺序依次是：首先是/sbin/init，这是 init 标准的位置；接下来是两个可能的位置，/etc/init 和/bin/init。

20072：这些是 init 可能出现的所有地方。如果现在还没有出现，**init** 就无法找到它的这个同名者了，机器可能就崩溃了。因此，它就会试图建立一个交互的 shell（/bin/sh）来代替。现在 **init** 最后的希望就是 root 用户可以修复这种错误并重新启动机器。（可以肯定，root 也正是希望如此。）

20073：**init** 甚至不能创建 shell－－一定是发生了什么问题！好，按照它们所说的，当所有其它情况都失败时，调用 **panic**（25563 行）。这样内核就会试图同步磁盘，确保其状态一致，然后暂停进程的执行。如果超过了内核选项中定义的时间，它也可能会重新启动机器。

第 5 章 系统调用

大部分介绍 Unix 内核的书籍都没有仔细说明系统调用，我认为这是一个失误。实际上，我们实际需要的系统调用现在已经十分完美。因此，从某种意义上来说，研究系统调用的实现是无意义的——如果你想为 Linux 内核的改进贡献自己的力量，还有其它许多方面更值得投入精力。

然而，对于我们来说，仔细研究少量系统调用是十分值得的。这样就有机会初步了解一些概念，这些概念将随本书发展而进行详细介绍，例如进程处理和内存。这使得你可以趁机详细了解一下 Linux 内核编程的特点。这包括一些和你过去在学校里（或工作中）所学的内容不同的方法。和其它编程任务相比，Linux 内核编程的一个显著特点是它不断同三个成见进行斗争——这三个成见就是速度、正确和清晰——我们不可能同时获取这三个方面…至少并不总是能够。

什么是系统调用

系统调用发生在用户进程（比如 emacs）通过调用特殊函数（例如 **open**）以请求内核提供服务的时候。在这里，用户进程被暂时挂起。内核检验用户请求，尝试执行，并把结果反馈给用户进程，接着用户进程重新启动，随后我们就将详细讨论这种机制。

系统调用负责保护对内核所管理的资源的访问，系统调用中的几个大类主要有：处理 I/O 请求（**open**，**close**，**read**，**write**，**poll** 等等），进程（**fork**，**execve**，**kill**，等等），时间（**time**，**settimeofday** 等等）以及内存（**mmap**，**brk**，等等）的系统调用。几乎所有的系统调用都可以归入这几类中。

然而，从根本上来说，系统调用可能和它表面上有所不同。首先，在 Linux 中，C 库中对于一些系统调用的实现是建立在其它系统调用的基础之上的。例如，**waitpid** 是通过简单调用 **wait4** 实现的，但是它们两个都是作为独立的系统调用说明的。其它的传统系统调用，如 **sigmask** 和 **ftime** 是由 C 库而不是由 Linux 内核本身实现的；即使不是全部，至少大部分是如此。

当然，从技巧的一面来看这是无害的——从应用程序的观点来看，系统调用就和其它的函数调用一样。只要结果符合预计的情况，应用程序就不能确定是否真正使用到了内核。（这种处理方式还有一个潜在的优点：用户可以直接触发的内核代码越少，出现安全漏洞的机会也就越少。）但是，由于使用这种技巧所引起的困扰将会使我们的讨论更为困难。实际上，系统调用这一术语通常被演讲者用来说明在第一个 Unix 版本中的任何对系统的调用。但是在本章中我们只对“真正”的系统调用感兴趣——真正的系统调用至少包括用户进程对部分内核代码的调用。

系统调用必须返回 **int** 的值，并且也只能返回 **int** 的值。为了方便起见，返回值如果为零或者为正，就说明调用成功；为负则说明发生了错误。就像老练的 C 程序员所知道的一样，当标准 C 库中的函数发生错误时会通过设置全局整型变量 **errno** 指明发生错误的属性，系统调用的原理和它相同。然而，仅仅研究内核源程序代码并不能够获得这种系统调用方式的全部意义。如果发生了错误，系统调用简单返回自己所期望的负数错误号，其余部分则由标准 C 库实现。（正常情况下，用户代码并不直接调用内核系统函数，而是要通过标准 C 库中专门负责翻译的一个小层次（thin layer）实现。）我们随便举一个例子，27825 行（**sys_nanosleep** 的一部分）返回**-EINVAL** 指明所提供的值越界了。标准 C 库中实际处理

sys_nanosleep 的代码会注意到返回的负值，从而设置 **errno** 和 **EINVAL**，并且自己返回-1 给原始的调用者。

在最近的内核版本中，系统调用返回负值偶尔也不一定表示错误了。在目前的几个系统调用中（例如 **lseek**），即使结果正确也会返回一个很大的负值。最近，错误返回值是在-1 到-4095 范围之内。现在，标准 C 库实现能够以更加成熟和高级的方式解释系统调用的返回值；当返回值为负时，内核本身就不用再做任何特殊的处理了。

中断、内核空间和用户空间

我们将在第 6 章中介绍中断和在第 8 章中介绍内存时再次明确这些概念。但是在本章中，我们只需要粗略地了解一些术语。

第一个术语是中断（interrupt），它来源于两个方面：硬件中断，例如磁盘指明其中存放一些数据（这与本章无关）；和软件中断，一种等价的软件机制。在 x86 系列 CPU 中，软件中断是用户进程通知内核需要触发系统调用的基本方法（出于这种目的使用的中断号是 0x80，对于 Intel 芯片的研究者来说更为熟悉的是 INT 80）。内核通过 **system_call**（171 行）函数响应中断，这一点我们马上就会介绍。

另外两个术语是内核空间（kernel space）和用户空间（user space），它们分别对应内核保留的内存和用户进程保留的内存。当然，多用户进程也经常同时运行，而且各个进程之间通常不会共享它们的内存，但是，任何一个用户进程使用的内存都称为用户空间。内核在某一个时刻通常只和一个用户进程交互，因此实际上不会引起任何混乱。

由于这些内存空间是相互独立的，用户进程根本不能直接访问内核空间，内核也只能通过 **put_user**（13278 行）和 **get_user**（13254 行）宏和类似的宏才可以访问用户空间。因为系统调用是进程和进程所运行的操作系统之间的接口，所以系统调用需要频繁地和用户空间交互，因此这些宏也就会不时的在系统调用中出现。在通过数值传递参数的情况下并不需要它们，但是当用户把指针——内核通过这个指针进行读写——传递给系统调用时，就需要这些宏了。

如何激活系统调用

系统调用的的激活有两种方法：**system_call** 函数和 **lcall7** 调用门（call gate）（请参看 135 行）。（你可能听说过还有一种机制，**syscall** 函数，是通过调用 **lcall7** 实现的——至少在 x86 平台上是如此——因此，它并不是一个特有的方法。）本节将细致地讨论一下这两种机制。

在阅读的过程中请注意系统调用本身并不关心它们是由 **system_call** 还是由 **lcall7** 激活的。这种把系统调用和其实现方式区别开来的方法是十分精巧的。这样，如果出于某种原因我们不得不增加一种激活系统调用的方法，我们也不必修改系统调用本身来支持这种方法。

在你浏览这些汇编代码之前要注意这些机器指令中操作数的顺序和普通 Intel 的次序相反。虽然还有一些其它的语法区别，但是操作数反序是最令人迷惑的。如果你还记得 Intel 的语法：

```
mov eax, 0
```

（本句代码的意思是把常数 0 传送到寄存器 EAX 中）在这里应该写作：

```
mov1 $0, %eax
```

这样你就能够正确通过。（内核使用的语法是 AT&T 的汇编语法。在 GNU 汇编文档中有更多资料。）

system_call

system_call（171 行）是所有系统调用的入口点（这是对于内部代码来说的；**lcall7** 用来支持 iBCS2，这一点我们很快就会讨论）。正如前面标题注释中说明的一样，目的是为普通情况简单地实现直接的流程，不采用跳转，因此函数的各个部分都是离散的——整体的流量控制已经因为要避免普通情况下的多分支而变得非常复杂。（分支的避免是十分值得的，因为它们引起的代价非常昂贵。它们可以清空 CPU 管道，使现存 CPU 的并行加速机制失效。）

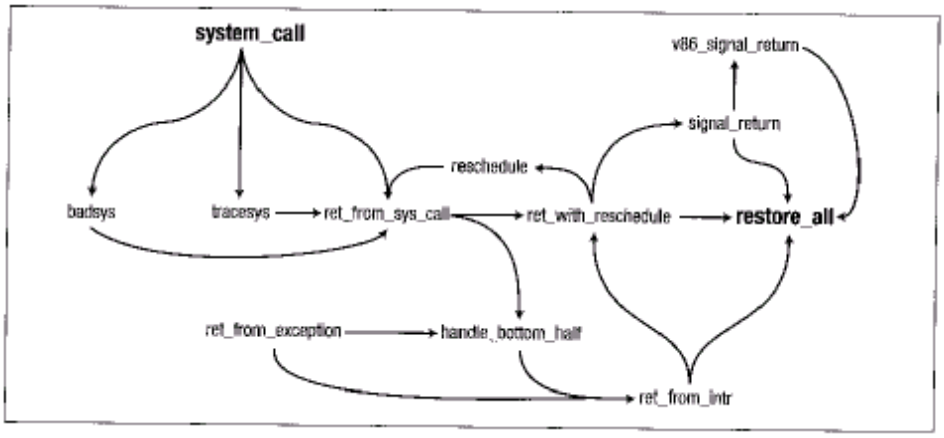

图 5.1 **system_call** 的流程控制

图 5.1 显示了作为 **system_call** 的一部分出现的分支目标标签以及它们之间的流程控制方向，该图可以在你阅读本部分讨论内容时提供很大的帮助。图中 **system_call** 和 **restore_all** 两个标签比其它标签都要大，因为这两处是该函数正常的出口点和入口点；然而，还有另外两个入口点，这一点在本章的后续内容中很快就可以看到。

system_call 是由标准 C 库激活的，该标准 C 库会把自己希望传递的参数装载到 CPU 寄存器中，并触发 0x80 软件中断。（**system_call** 在这里是一个中断处理程序。）内核记录了软件中断和 6828 行的 **system_call** 函数的联系（**SYSCALL_VECTOR** 是在 1713 行宏定义为 0x80 的）。

system_call

172： **system_call** 的第一个参数是所希望激活的系统调用的数目；它存储在 EAX 寄存器中。**system_call** 还允许有多达四个的参数和系统调用一起传送。在一些极其罕见的情况下使用四个参数的限制是负担繁重的，通常可以建立一个指向结构的指针参数来巧妙地完成同样功能，指针指向的结构中可以包含你所需要的一切信息。

随后可能需要 EAX 值的一个额外拷贝，因此通过将其压栈而保存起来；这个值就是 218 行的 **ORIG_EAX**（%**esp**）表达式的值。

173： **SAVE_ALL** 宏是在 85 行定义的；它把所有寄存器的值压入 CPU 的堆栈。随后，就在 **system_call** 返回之前，使用 **RESTALL_ALL**（100 行）把栈中的值弹出。在这中间，**system_call** 可以根据需要自由使用寄存器的值。更重要的是，任何它所调用的 C 函数都可以从栈中查找到所希望的参数，因为 **SAVE_ALL** 已经把所有寄存器的值都压入栈中了。

结果栈的结构从 26 行开始描述。象 **0（%esp）**和 **4（%esp）**一样的表达式指明了堆栈指针（ESP 寄存器）的一种替换形式——分别表示 ESP 上的 0 字节，ESP 上的 4 字节，等等。特别要注意的是在前面一行中压入堆栈的 EAX 的拷贝已经变成本标

题注释作为 **orig_eax** 所描述的内容；它们是由 **SAVE_ALL** 压入寄存器之上的堆栈的（**orig_eax** 之上的寄存器在这里早已就绪了）。

还需注意：这可能有点令人迷惑——由于我们调用 **orig_eax** 时 EAX 的拷贝已经压入了堆栈，它是否有可能在其它寄存器下面而不是在其它寄存器上面呢？答案既是肯定的，也是否定的。x86 的堆栈指针寄存器 ESP 在有数据压入堆栈时会减少——堆栈会向内存低地址发展。因此，**orig_eax** 逻辑上是在其它值的下面，但是物理上却是在其它值的上面。

从 51 行开始的一系列宏有助于使这些替换更容易理解。例如，**EAX（%esp）**就和 **18（%esp）**相同——然而前一种方法通过表达式引用存储在堆栈中的 EAX 寄存器副本的决定可以使整个过程更加简单。

174： 从 EBX 寄存器中取得指向当前任务的指针。完成这个工作的宏 **GET_CURRENT**（131 行）对于在大部分代码中使用的 C 函数 **get_current**（10277 行）来说是一个无限循环。

此后，当看到类似于 **foo（%ebx）**或者 **foo（%esp）**的表达式时，这意味着这些的代码正在引用代表当前进程的结构的字段——16325 行的 **struct task_struct**——这在第 7 章中将对它进行更详细的介绍。（更确切的描述是， **%ebx** 的置换在 **struct task_struct** 中，**%esp** 的置换在与 **struct task_struct** 相关联的 **struct pt_regs** 结构中。但是这些细节在这里都并不重要。）

175： 检查（EAX 中的）系统调用的数目是否超过系统调用的最大数量。（此处 EAX 为一个无符号数，因此不可能为负值。）如果的确超过了，就向前跳转到 **badsys**（223 行）。

177： 检测系统调用是否正被跟踪。如 strace 之类的程序为有兴趣的人提供了系统调用的跟踪工具，或者额外的调试信息：如果能够监测到正在执行的系统调用，那么你就可以了解到当前程序正在处理的内容。如果系统调用正被跟踪，控制流程就向前跳转到 **tracesys**（215 行）。

179： 调用系统函数。此处有很多工作需要处理。首先，**SYSMOL_NAME** 宏不处理任何工作，只是简单的为参数文本所替换，因此可以将其忽略。**sys_call_table** 是在当前文件（arch/i386/kernel/entry.S）的末尾从 373 行开始定义的。这是一张由指向实现各种系统调用的内核函数的函数指针组成的表。

本行中第二对圆括号中包含了三个使用逗号分割开的参数（第一个参数为空）；这里就是实现数组索引的地方。当然，这个数组是以 **sys_call_table** 作为索引的，这称为偏移（displacement）。这三个参数是数组的基地址、索引（EAX，系统调用的数目）和大小，或者每个数组元素中的字节数——在这里就是 4。由于数组基地址为空，就将其当作 0——但是它要和偏移地址，**sys_call_table，**相加，简单的说就是 **sys_call_table** 被当作数组的基地址。本行基本上等同于如下的 C 表达式：

```
/* Call a function in an array of functions. */
(sys_call_table[eax])();
```

然而， C 当然还要处理许多繁重的工作，例如为你记录数组元素的大小。不要忘记，系统调用的参数早已经存储在堆栈中了，这主要由调用者提供给 **system_call** 并使用 **SAVE_ALL** 把它们压栈。

180： 系统调用已经返回。它在 EAX 寄存器中的返回值（这个值同时也是 **system_call** 的返回值）被存储起来。返回值被存储在堆栈中的 EAX 内，以使得 **RESTORE_ALL** 可以迅速地恢复实际的 EAX 寄存器以及其它寄存器的值。

182： 接下来的代码仍然是 **system_call** 的一部分，它是一个也可以命名为 **ret_from_sys_call** 和 **ret_from_intr** 的独立入口点。它们偶尔会被 C 直接调用，也可

以从 **system_call** 的其它部分跳转过来。

185： 接下来的几行检测“下半部分（bottom half）”是否激活；如果激活了，就跳转到 **handle_bottom_half** 标号（242 行）并立即开始处理。下半部分是中断进程的一部分，将在下一章中讨论。

189： 检查该进程是否为再次调度做了标记（记住表达式**$0** 就是常量 0 的系统简单表示）。如果的确如此，就跳转到 **reschedule** 标号（247 行）。

191： 检测是否还有挂起的信号量，如果有的话，下一行就向前跳转到 **signal_return**（197 行）。

193： **restore_all** 标号是 **system_call** 的退出点。其主体就是简单的 **RESTORE_ALL** 宏（100 行），该宏将恢复早先由 **SAVE_ALL** 存储的参数并返回给 **system_call** 的调用者。

197： 当 **system_call** 从系统调用返回前，如果它检测到需要将信号量传送给当前的进程时，才会执行到 **signal_return**。它通过使中断再次可用开始执行，有关内容将在第 6 章中介绍。

199： 如果返回虚拟 8086 模式（这不是本书的主题），就向前跳转到 **v86_signal_return**（207 行）。

202： **system_call** 要调用 C 函数 **do_signal**（3364 行，在第 6 章中讨论）来释放信号量。**do_signal** 需要两个参数，这两个参数都是通过寄存器传递的；第一个是 EAX 寄存器，另一个是 EDX 寄存器。**system_call**（在 200 行）早已把第一个参数的值赋给了 EAX；现在，就把 EDX 寄存器和寄存器本身进行 XOR 操作，从而将其清 0，这样 **do_signal** 就认为这是一个空指针。

203： 调用 **do_signal** 传递信号量，并且跳回到 **restore_all**（193 行）结束。

207： 由于虚拟 8086 模式不是本书的主题，我们将忽略大部分 **v86_signal_return**。然而，它和 **signal_return** 的情况非常类似。

215： 如果当前进程的系统调用正由其祖先跟踪，就像 strace 程序中那样，那么就可以执行到 **tracesys** 标号。这一部分的基本思想如同 179 行一样是通过 **syscall_table** 调用系统函数，但是这里把该调用和对 **syscall_trace** 函数的调用捆绑在一起。后面的这个函数在本书中并没有涉及到，它能够中止当前进程并通知其祖先注意当前进程将要激活一个系统调用。

EAX 操作和这些代码的交错使用最初可能容易令人产生困惑。**system_call** 把存储在堆栈中的 EAX 拷贝赋给**-ENOSYS**，调用 **syscall_trace**，在 172 行再从所做的备份中恢复 EAX 的值，调用实际的系统调用，把系统调用的返回值置入堆栈中 EAX 的位置，再次调用 **syscall_trace**。

这种方式背后的原因是 **syscall_trace**（或者更准确的说是它所要使用到的跟踪程序）需要知道在它是在实际系统调用之前还是之后被调用的。**-ENOSYS** 的值能够用来指示它是在实际系统调用执行之前被调用的，因为实际中所有实现的系统调用的执行都不会返回**-ENOSYS**。因此，EAX 在堆栈中的备份在第一次调用 **syscall_trace** 之前是**-ENOSYS**，但是在第二次调用 **syscall_trace** 之前就不再是了（除非是调用 **sys_ni_syscall** 的时候，在这种情况下，我们并不关心是怎样跟踪的）。218 行和 219 行中 EAX 的作用只是找出要调用的系统调用，这和无须跟踪的情况是一致的。

222： 被跟踪的系统调用已经返回；流程控制跳转回 **ret_from_sys_call**（184 行）并以与普通的无须跟踪的情况相同的方式结束。

223： 当系统调用的数目越界时，就可以执行到 **badsys** 标号。在这种情况下，**system_call** 必须返回**-ENOSYS**（**ENOSYS** 在 82 行将它赋值为 38）。正如前面提到的一样，调用者会识别出这是一个错误，因为返回值在-1 到-4,095 之间。

228： 在诸如除零错误（请参看 279 行）之类的 CPU 异常中断情况下将执行到 **ret_from_exception** 标号；但是 **system_call** 内部的所有代码都不会执行到这个标号。如果有下半部分是激活的，现在就是它在起作用了。

233： 处理完下半部分之后或者从上面的情况简单的执行下来（虽然没有下半部分是激活的，但是同样也触发了 CPU 异常），就执行到了 **ret_from_intr** 标号。这是一个全局符号变量，因此可能在内核的其它部分也会有对它的调用。

237： 被保存的 CPU 的 EFLAGS 和 CS 寄存器在此已经被并入 EAX，因而高 24 位的值（其中恰好包含了一位在 70 行定义的非常有用的 **VM_MASK**）来源于 EFLAGS，其它低 8 位的值来源于 CS。该行隐式的同时对这两部分进行测试以判断进程到底返回虚拟 8086 模式（这是 **VM_MASK** 的部分）还是用户模式（这是 3 的部分——用户模式的优先等级是 3）。下面是近似的等价 C 代码：

```
/* Mix eflags and cs in eax. */
eax = eflags & ~0xff;
eax |= cs & ~0xff
/* Simultaneously test lower 2 bits
* and VM_MASK bit. */
if    (eax & (VM_MASK | 3))
    goto ret_with_reschedule;
goto restore_all;
```

238： 如果这些条件中有一个能得到满足，流程控制就跳转到 **ret_with_reschedule**（188 行）标号来测试在 **system_call** 返回之前进程是否需要再次调度。否则，调用者就是一个内核任务，因此 **system_call** 通过跳转到 **restore_all** (193 行)来跳过重新调度的内容。

242： 无论何时 **system_call** 使用一个下半部分服务时都可以执行到 **handle_bottom_half** 标号。它简单的调用第 6 章中介绍的 C 函数 **bottom_half**（29126 行），然后跳回到 **ret_from_intrr**（233 行）。

248： **system_call** 的最后一个部分在 **reschedule** 标号之下。当产生系统调用的进程已经被标记为需要进行重新调度时，就可以执行到这个标号；典型地，这是因为进程的时间片已经用完了——也就是说，进程到目前为止已经尽可能的拥有 CPU 了，应该给其它进程一个机会来运行了。因此，在必要的情况下就可以调用 C 函数 **schedule**（26686 行）交出 CPU，同时流程控制转回 249 行。CPU 调度是第 7 章中讨论的一个主题。

lcall7

Linux 支持 Intel 二进制兼容规范标准的版本 2（iBCS2）。（iBCS2 中的小写字母 i 显然是有意的，但是该标准却没有对此进行解释；这样看来似乎和现实的 Intel 系列的 CPU 例如 i386，i486 等等是一致的。）iBCS2 的规范中规定了所有基于 x86 的 Unix 系统的应用程序的标准内核接口，这些系统不仅包括 Linux，而且还包括其它自由的 x86 Unix（例如 FreeBSD），也还包括 Solaris/x86，SCO Unix 等等。这些标准接口使得为其它 Unix 系统开发的二进制商业软件在 Linux 系统中能够直接运行，反之亦然（而且，近期新开发软件向其它 Unix 移植的情况越来越多）。例如，Corel 公司的 WordPerfect 的 SCO Unix 的二进制代码在还没有 Linux 的本地版本的 WordPerfect 之前就可以使用 iBCS2 在 Linux 上良好地运行。

iBCS2 标准有很多组成部分，但是我们现在关心的是这些系统调用如何协调一致来适应这些迥然不同的 Unix 系统。这是通过 **lcall7** 调用门实现的。它是一个相当简单的汇编函数

（尤其是和 **system_call** 相比而言更是如此），仅仅定位并全权委托一个 C 函数来处理细节。（调用门是 x86 CPU 的一种特性，通过这种特性用户任务可以在安全受控的模式下调用内核代码。）这种调用门在 6802 行进行设定。

lcall7

136： 前面的几行将通过调整处理器堆栈以使堆栈的内容和 **system_call** 预期的相同——**system_call** 中的一些代码将会完成清理工作，这样所有的内容都可以连续存放了。

145： 基于同样的思想，**lcall7** 把指向当前任务的指针置入 EBX 寄存器，这一点和 **system_call** 的情况是相同的。但是，它的执行方式却与 **system_call** 不同，这就比较奇怪了。这三行可以等价地按如下形式书写：

push1 %esp

GET_CURRENT（%ebx）

这种实现的执行速度并不比原有的更快；在将宏展开以后，实际上这还是同样的三条指令以不同的次序组合在一起而已。这样做的优点是可以和文件中的其它代码更为一致，而且代码也许会更清晰一些。

148： 取得指向当前任务 **exec_domain** 域的指针，使用这个域以获取指向其 **lcall7** 处理程序的指针，接着调用这个处理程序。

本书中并没有对执行域（execution domains）进行详细说明——但是简单说来，内核使用执行域实现了部分 iBCS2 标准。在 15977 行你可以找到 **struct exec_domain** 结构。**default_exec_domain**（22807 行）是缺省的执行域，它拥有一个缺省的 **lcall7** 处理程序。它就是 **no_lcall7**（22820 行）。其基本的执行方式类似于 SVR4 风格的 Unix，如果调用进程没有成功，就传送一个分段违例信号量（segmentation violation signal）给调用的进程，。

152： 跳转到 **ret_from_sys_call** 标号（184 行——注意这是在 **system_call** 内部的）清除并返回，就像是正常的系统调用一样。

系统调用样例

现在你已经知道了系统调用是如何激活的，接下来我们将通过几个系统调用例子的剖析来了解一下它们的工作方式。注意系统调用 **foo** 几乎都是使用名为 **sys_foo** 的内核函数实现的，但是在某些情况下该函数也会使用一个名为 **do_foo** 的辅助函数。

sys_ni_syscall

29185：**sys_ni_syscall** 的确是最简单的系统调用；它只是简单的返回 **ENOSYS** 错误。最初的时候这可能显得没有什么作用，但是它的确是有用的。实际上，**sys_ni_syscall** 在 **sys_call_table** 中占据了很多位置——而且其原因并不只有一个。开始的时候，**sys_ni_syscall** 在位置 0（374 行），因为如果漏洞百出的代码错误地调用了 **system_call**——例如，没有初始化作为参数传递给 **system_call** 的变量——在这种偶然的变量定义中，0 是最可能的值。如果我们能够避免这种情况，那么在错误发生时就不用采取象杀掉进程一样的剧烈措施。（当然，只要允许有用工作的进行，就不可能防止所有的错误。）这种使用表的元素 0 作为抵御错误的手段在内核中被作为良好的经验而广泛使用。

而且，你还会发现 **sys_ni_syscall** 在表中明显出现的地方就多达十几处。这些条目代表了那些已经从内核中移出的系统调用——例如在 418 行，就代替了已经废弃了的

prof 系统调用。我们不能简单地把另外的实际系统调用放在这里，因为老的二进制代码可能还会使用到这些已经废弃了的系统调用号。如果一个程序试图调用这些老的系统调用，但是结果却与预期的完全不同，例如打开了一个文件，这会比较令人感到奇怪的。

最后，**sys_ni_syscall** 将占据表尾部所有未用的空间；这一点是在从 572 行到 574 行的代码实现的，它根据需要重复使用这些项来填充表。由于 **sys_ni_syscall** 只是简单返回 **ENOSYS** 错误号，对它的调用和跳转到 **system_call** 中的 **badsys** 标号作用是相同的——也就是说，使用指向这些表项的系统调用号和在表外对整个表进行全部索引具有相同的作用。因此，我们不用改变 **NR_syscalls** 就可以在表中增加（或者删除）系统调用，但是其效果与我们真的对 **NR_syscalls** 进行了修改一样（不管怎样，这都是由 **NR_syscalls** 所建立的限制条件所决定的）。

到现在也许你已经猜到了，**sys_ni_syscall** 中的“ni”并不是指 Monty Python 的“说‘Ni’ 的骑士”；而是指“not implemented（没有实现）”这一相较而言并不太诙谐的短语。

对于这个简单的函数我们需要研究的另外一个问题是 **asmlinkage** 标签。这是为一些 gcc 功能定义的一个宏，它告诉编译器该函数不希望从寄存器中（这是一种普通的优化 ）取得任何参数，而希望仅仅从 CPU 堆栈中取得参数。回忆一下我们前面提到过 **system_call** 使用第一个参数作为系统调用的数目，同时还允许另外四个参数和系统调用一起传递。**system_call** 通过把其它参数（这些参数是通过寄存器传递过来的）滞留在堆栈中的方法简单的实现了这种技巧。所有的系统调用都使用 **asmlinkage** 标签作了标记，因此它们都要查找堆栈以获得参数。当然，在 **sys_ni_syscall** 的情况下这并没有任何区别，因为 **sys_ni_syscall** 并不需要任何参数。但是对于其它大部分系统调用来说这就是个问题了。并且，由于在其它很多函数前面都有 **asmlinkage** 标签，我想你也应该对它有些了解。

sys_time

31394：**sys_time** 是包含几个重要概念的简单系统调用。它实现了系统调用 **time**，返回值是从某个特定的时间点（1970 年 1 月 1 日午夜 UTC）以来经过的秒数。这个数字被作为全局变量 **xtime**（请参看 26095 行；它被声明为 **volatile** 型的变量，因为它可以通过中断加以修改，这一点我们在第 6 章中就会看到）的一部分，通过 **CURRENT_TIME** 宏（请参看 16598 行）可以访问它。

31400：该函数非常直接的实现了它的简单定义。当前时间首先被存储在局部变量 **i** 中。

31402：如果所提供的指针 **tloc** 是非空的，返回值也将被拷贝到指针指向的位置。该函数的一个微妙之处就在于此；它把 i 拷贝到用户空间中而不是使用 **CURRENT_TIME** 宏来重新对其进行计算，这基于两个原因：

- **CURRENT_TIME** 宏的定义以后可能会改变，新的实现方法可能会由于某种原因而速度比较慢，但是对于 i 的访问至少应该和 **CURRENT_TIME** 宏展开的速度同样快。
- 使用这种方式处理，确保结果的一致性：如果代码刚好执行到 31400 行和 31402 行之间时时间发生了改变，**sys_time** 可能把一个值拷贝到***tloc** 中，但是在结束之后却返回另一个值。

另外还有一个小的方面需要注意，此处的代码不使用**&&**来编写而是使用两个 **if**，这可能有一点令人奇怪。内核中采用这些看起来非常特殊的代码的一般原因都是由于速度的要求，但是 gcc 为**&&**版本和两个 **if** 版本的代码生成的代码是等同的，因此这里的原因就不可能是速度的要求——除非这些代码是在早期 gcc 版本下开发的，

这样才有些意义。

31403：如果 **sys_time** 不能访问所提供的位置（一般都是因为 **tloc** 无效），它就把**-EFAULT** 的值赋给 **i**，从而在 31405 行返回错误代码。

31405：为调用者返回的 **i** 或者是当前时间，或者是**-EFAULT**。

sys_reboot

29298：内核中其他地方可能都没有 **sys_reboot** 的实现方法这样先进。其原因是可以理解为：根据调用的名字我们就可以知道，**reboot** 系统调用可以用来重新启动机器。根据所提供的参数，它还能够挂起机器，关闭电源，允许或者禁止使用 Ctrl+Alt_Del 组合键来重启机器。如果你要使用这个函数编写代码，需要特别注意它上面的注释标题的警告：首先同步磁盘，否则磁盘缓冲区中的数据可能会丢失。

由于它可能为系统引发的潜在后果，**sys_reboot** 需要几个特殊参数，这一点马上就会讨论。

29305：如果调用者不具有 **CAP_SYS_BOOT**（14096 行）权能（capability），系统就会返回 **EPERM** 错误。权能在第 7 章中会详细讨论。现在，简单的说就是：权能是检测用户是否具有特定权限的方法。

29309：在这里，这种偏执的思想充分发挥了作用。**syst_reboot** 根据从 16002 到 16005 行定义的特殊数字检测参数 **magic1** 和 **magic2**。这种思想是如果 **sys_reboot** 在某种程度上是被偶然调用的，那么就不太可能再从由 **magic1** 和 **magic2** 组成的小集合中同时提取值。注意这并不意味着这是一个防止粗心的安全措施。

顺便说一下，这些特殊数字并不是随机选取的。第一个参数的关系是十分明显的，它是“感受死亡（feel dead）”的双关语。后面的三个参数要用十六进制才能了解它们全部的意思：它们分别是 0x28121969，0x5121996，0x16041998。这似乎代表 Linus 的妻子（或者就是 Linus 自己）和他两个女儿的生日。由此推论，当 Linus 和他的妻子养育了更多儿女的时候，重启动需要的特殊参数可能在某种程度上会增加。不过我想在他们用尽 32 位可能空间之前，他的妻子就会制止他的行为了。

29315：请求内核锁，这样能保证这段代码在某一时间只能由一个处理器执行。使用 **lock_kernel/unlock_kernel** 函数对所保护起来的任何其它代码对其它 CPU 都同样是不可访问的。在单处理器的机器中，这只是一个 no-op（不处理任何事情）；而详细讨论它在多处理器上的作用则是第 10 章的内容。

29317：在 **LINUX_REBOOT_CMD_RESTART** 的情况中，**sys_reboot** 调用一系列基于 **reboot_notifier_list** 的函数来通知它们系统正在重新启动。正常情况下，这些函数都是操作系统关闭时需要清除的模块的一部分。这个列表函数似乎并不在内核中的其它地方使用——至少在标准内核发行版本中是这样，也许此外的其它模块可能使用这个列表。不管怎样，这个列表的存在可以方便其他人使用。

LINUX_REBOOT_CMD_RESTART 和其它 **cmd** 识别出的值从 16023 行开始通过 **#define** 进行宏定义。这些值并没有潜在的意义，选用它们的简单原因是它们一般不会发生意外并且相互之间各不相同。（有趣的是，**LINUX_REBOOT_CMD_OFF** 是零，这是在意外情况下最不可能出现的一个值。但是，由于 **LINUX_REBOOT_CMD_OFF** 简单的禁止用户使用 Ctrl+Alt+Del 重新启动机器，它就是一种“安全”的意外了。）

29321：打印警告信息以后，**sys_reboot** 调用 **machine_restart**（2185 行）重启机器。正如你从 2298 行中所看到的一样，**machine_restart** 函数从来不会返回。但是不管怎样，对于 **machine_restart** 的调用后面都跟着一个 **break** 语句。

这仅仅是经典的良好的编程风格吗？的确如此，但是却又不仅仅如此。文件

kernel/sys.c 的代码是属于体系结构无关部分的。但是 **machine_restart**，它显然是体系结构所特有的，属于代码的体系结构特有的部分（arch/i386/kernel/process.c）。因而对于不同的移植版本也有所不同。我们并不清楚以后内核的每个移植版本的实现都不会返回——例如，它可能调度底层硬件重启但是本身要仍然持续运行几分钟，这就需要首先从函数中返回。或者更为确切的说法是，由于某些特定的原因，系统可能并不总是能够重启；或许某些软件所控制的硬件根本就不能重启。在这种平台上，**machine_restart** 就应该可以返回，因此体系结构无关的代码应该对这种可能性有所准备。

针对这个问题，正式的发行版本中都至少包含一个退出端口，使 **machine_restart** 函数可以从这个端口返回：m68k 端口。不同的基于 m68k 的机器支持的代码也各不相同，由于本书主要是针对 x86 的，我不希望花费过多的时间来解析所有的细节。但是这的确是可能的。（在其它情况下，**machine_restart** 简单进入一个无限循环——既不重新启动机器，也不返回。但是这里我们担心的是需要返回的情况。）

因此，我们毕竟是需要 **break** 的。前面看起来只是简单的习惯甚至是偏执的思想在这里为了内核的移植性已经变成必须的了。

29324：接下来的两种情况分别允许和禁止臭名卓著的 Ctrl+Alt+Del 组合键（这三个组合键也被称为“Vulcan 神经收缩（Vulcan nerve pinch）”，“黑客之手（hacker’s claw）”，“三指之礼（three-fingered salute）”，我最喜欢后面这个）。这些只是简单的设置全局 **C_A_D** 标志（在 29160 行定义，在 29378 行检测）。

29332：这种情况和 **LINUX_REBOOT_CMD_RESTART** 类似，但只是暂停系统而不是将其重新启动。两者之间的一个区别是它调用 **machine_halt**（2304 行）——这是 x86 上的一条 no-op 指令，但是在其它平台上却要完成关闭系统的实际工作——而不是调用 **machine_restart**。并且它会把 **machine_halt** 不能使之暂停的机器转入低功耗模式运行。它使用 **do_exit**（23267 行）杀死内核本身。

29340：到现在为止，这已经是一种比较熟悉的模式了。这里，**sys_reboot** 关闭机器电源，除了为可以使用软件自行关闭电源的系统调用 **machine_power_off**（2307 行）之外，其它的应该和暂停机器情况完全相同。

29348：**LINUX_REBOOT_CMD_RESTART2** 的情况是已建立主题的一个变种。它接收命令，将其作为 ASCII 字符串传递，该字符串说明了机器应该如何关闭。字符串不会由 **sys_reboot** 本身来解释，而是使用 **machine_restart** 函数来解释；因而这种模式的意义，如果有的话，就是这些代码是平台相关的。（我使用“如果有”的原因是启动机器——特别是在 x86 中——一般只有一种方法，因此其它的信息都可以被 **machine_restart** 忽略。）

29365：调用者传递了一个无法识别的命令。**sys_reboot** 不作任何处理，仅仅返回一个错误。因此，即使由 **magic1** 和 **magic2** 传递给 **sys_reboot** 正确的 magic 数值，它也无须处理任何内容。

29369：一个可识别的命令被传递给 **sys_reboot**。如果流程执行到这里，它可能就是两个设置 **C_A_D** 的命令之一，因为其它情况通常都是停止或者重新启动机器。在任何情况下，**sys_reboot** 都简单把内核解锁并返回 0 以表示成功。

sys_sysinfo

24142：一个只能返回一个整型值的系统调用。如果需要返回更多的信息，我们只需要使用类似于在系统调用中传递多于四个参数时所使用的技巧就可以了：我们通过一个指向结构的指针将结果返回。收集系统资源使用情况的 **sysinfo** 系统调用就是这种函数的一个样例。

24144：分配并清空一个 **struct sysinfo** 结构（15004 行）以暂时存储返回值。**sys_sysinfo** 可以把结构中的每个域都独立地拷贝出来，但是这样会速度很慢、很不方便，而且必然不容易阅读。

24148：禁止中断。这在第 6 章中会有详细的介绍；作为目前来说，我们只要说明这种模式在使用的过程中能够确保 **sys_sysinfo** 正在使用的值不会改变就足够了。

24149：**struct sysinfo** 结构的 **uptime** 域用来指明系统已经启动并运行了的秒数。这个值是使用 **jiffies**（26146 行）和 **HZ** 来计算的。**jiffies** 计算了系统运行过程中时钟的滴答次数；**HZ** 是系统相关的一个参数，它十分简单，就是每秒内部时钟滴答的次数。

24151：数组 **avenrun**（27116 行）记录了运行队列的平均长度——也就是等待 CPU 的平均进程数——在最后的 1 秒钟，5 秒钟和 15 秒钟。**calc_load**（27135 行）周期性的重复计算它的值。由于内核中是要严格禁止浮点数运算的，所以只能通过计算变化的次数这一修正值来计算。

24155：同样记录系统中当前运行的进程数。

24158：**si_meminfo**（07635 行）写入这个结构中的内存相关成员，**si_swapinfo**（38544 行）写入与虚拟内存相关的部分。

24161：现在整个结构都已经全部填充了。**sysinfo** 试图将其拷贝回用户空间，如果失败就返回 **EFAULT**，如果成功就返回 0。

第 6 章 信号量，中断和时间

信号量（Signal）是进程间通讯（IPC）的一种形式——是一个进程给另一个进程发送信息的方法。但是信息不可能很多——一个信号量不可能携带详细的信息，即使是传送者的身份也不能被传递；唯一能够确定的事实是信号量的确被发送了。（然而和经典信号量不同，POSIX 实时信号量允许传送稍微多一点的信息。）实际上，信号量对于双向通讯是没有用处的。还有，根据某些限定，信号量的接受者不必以任何方式作出响应，甚至可以直接忽略大部分信号量。

虽然有这么多的限制，然而信号量仍然是一种功能强大的十分有用的机制——勿庸置疑，这是 Unix IPC 中使用最频繁的机制。每当进程退出或者废弃一个空指针时，每当使用 Ctrl+C 键终止程序运行时，都要传递信号量。

第 9 章会更详细的讨论 IPC 机制。对于本章的讨论来说，信号量的内容就足够讨论了。

正如在 Linux 内核本身的代码注释中所说明的一样，中断（Interrupt）对于内核来说和信号量是类似的。中断一般都是从磁盘之类的硬件设备送往内核，用以提示内核该设备需要加以注意。一个重要的硬件中断源就是定时器设备，它周期性地通知内核已经通过的时间。如同第 5 章中阐述的一样，中断也可以由用户进程通过软件产生。

在本章中，我们首先讨论一下 Linux 中信号量和中断的实现，最后再浏览一下 Linux 的时间处理方式。

虽然内核对代码的要求标准非常严格，本章所涉及的代码仍然特别清晰明白。本章使用的一般方法是首先介绍相关的数据结构和它们之间的关系，接下来讨论操纵和查询它们的函数。

锁的概述

锁的基本思想是限制对共享资源的访问——共享资源包括共享的文件，共享的内存片，以及在一次只能由一个 CPU 执行的代码段。概括的说，在单处理器上运行的 Linux 内核并不需要锁，这是因为在编写 Linux 内核时就已经注意到要尽量避免各种可能需要锁的情况了。但是，在多处理器机器上，一个处理器有时需要防止其它处理器对它的有害的介入。

include/asm-i386/spinlock.h 文件（从 12582 行开始）并不使用难看的**#ifdef** 把所有对锁函数的调用封装起来，它包含一系列对单处理器平台（UP）基本为空的宏，然而在多处理器平台（SMP）上这些宏将展开成为实际代码。因而内核的其它代码对 UP 和 SMP（当涉及到这种特性时）都是相同的，但是它们两个的效果却是迥然不同的。

第 10 章中涉及 SMP 的部分会对锁做深入的介绍。但是，由于你在代码中将到处都能够看到对锁宏的调用，特别是在本章所讨论到的代码中这一点尤为明显，所以你应该首先对宏的用途有初步了解——以及为什么现在在大多数情况下我们都可以安全地将其忽略（我们将在讨论的过程中对其中的异常情况进行说明）。

信号量

Linux 内核将信号量分为两类：

- 非实时的（Nonrealtime）——大部分是些传统的信号量，例如 **SIGSEGV**，**SIGHUP**

和 **SIGKILL**。

- 实时的（realtime）——由 POSIX 1003.1b 标准规定，它们同非实时信号量有细微的区别。特别是实时信号量具有进程可以配置的意义——就像是非实时信号量 **SIGUSR1** 和 **SIGUSR2** 一样——额外的信息能够和这些信号量一起传送。它们也会排队，因此如果在第一个信号量处理完成之前有多个信号量实例到达，所有的信号量都能够被正确传送；这对于非实时信号量则是不可能的。

在第 7 章中我们将会对实时性对于 Linux 内核的意义进行更详细的介绍——特别是实时性所不能够说明的内容。

信号量数目的宏定义从 12048 行开始。实时信号量的数目在 **SIGRTMIN** 和 **SIGRTMAX**（分别在 12087 行和 12088 行）所定义的范围之内。

数据结构

本节讨论信号量代码使用的最重要的数据结构。

sigset_t

12035：**sigset_t** 表示信号量的集合。根据使用地点的不同，它的意思也不同——例如，它可能记录着正在等待某一个进程的信号量（如 16425 行 **struct task_struct** 的 **signal** 成员）的集合，也可能是某个进程已经请求阻塞了的信号量（如同一行中定义的同一结构的 **blocked** 成员）的集合。随着本书的进行，我们会逐渐看到这些类似的应用。

12036：**sigset_t** 的唯一一个组成部分是一组 **unsigned long**（无符号长整型数），其中的每一位都代表一个信号量。注意到无符号长整型类型在整个内核代码中是作为一个字来处理的，这和你所希望的可能有所出入——即使是在当前 x86 CPU 的讨论中，有时候字也被用于说明 16 位类型。由于 Linux 是一个真 32 位操作系统，将 32 位看作是一个字在绝大多数情况下是正确的。（将 Linux 称为真 32 位操作系统也有一些不准确，因为在 64 位 CPU 上它也是一个真 64 位操作系统。）

这个数组的大小 **_NSIG_WORDS** 在 12031 行直接计算。（**_NSIG_BPW** 中的“BPW”是“bits per word（每字位数）”的缩写。）在不同的平台上，**_NSIG_WORDS** 的大小从 1（Alpha 平台中）到 4（MIPS 平台中）不等。如你所见，在 x86 平台中，该值正好是 2，这意味着在 x86 平台上 2 个无符号数就可以包含足够的位数来代表所有 Linux 使用的信号量。

struct sigaction

12165：**struct sigaction** 代表信号量到达时进程应该执行的动作。它被封装在 **struct k_sigaction**（12172 行）结构中，而该结构又是被封装在 **struct signal_struct** 结构中的，后者是 **struct task_struct** 结构的 **sig** 成员所指向的一个实例（16424 行）。如果这个指针为空，进程就会退出而不必接受任何信号量。否则，每个进程对于每个信号量数目都需要若干 **_NSIG struct sigaction** 结构和一个 **struct sigaction** 结构。

12166：**sa_handler**（**__sighandler_t** 类型——一个在 12148 行定义的函数指针类型）描述了进程希望处理信号量的方式。其值可以是下面中的一个：

- **SIG_DFL**（12151 行）申请处理信号量的缺省操作，不管该操作是什么——这是由信号量所决定的。注意它和 **NULL** 是等同的。
- **SIG_IGN**（12153 行）意味着信号量应该忽略。但是，并不是所有的信号量都可以被忽略的。
- 所有的其它值都是在信号量到达时所需要调用的用户空间函数的地址。

12167：**sa_flags** 进一步调整信号量处理代码所完成的工作。可能的标志集合从 12108 行开始定义。这些标志允许用户代码在信号量实例发送以后（或者保留用户定制的操作时）请求恢复缺省操作，等等。这一点在宏定义块前面的标签注释中已经说明了。

12168：**sa_restorer** 是本书中所没有涉及的一些信号量处理代码细节所使用的。

12169：**sa_mask** 是一系列其它信号量的集合，进程在处理这些信号量的过程中可能需要进行锁定。例如，如果一个进程在处理 **SIGCHLD** 的时候希望锁定 **SIGHUP** 和 **SIGINT**，进程的第 **SIGCHLD** 个 **sa_mask** 就会对与 **SIGHUP** 和 **SIGINT** 相关的位进行置位。

siginfo_t

11851：**struct siginfo**（也称为 **siginfo_t**）是伴随着信号量，特别是在实时信号量，所传递的额外信息。

11852：勿庸置疑，**si_signo** 是信号量的数目。

11853：**si_errno** 应该是信号量传递时传送者的 **errno** 的值，这样接收者就可以对它进行检测。内核本身并不关心这个值；当在某些情况下需要设置这个值时，内核将其设置为 0。我推测如果这样，即使调用者没有设置这个值，它们仍然会发现 **si_error** 的值被设为已知状态。

11854：**si_code** 记录了信号量的来源（不是发送者的进程 ID 号，也就是 PID——它在别处记录）。有效的信号量来源在 11915 行及其随后部分使用宏进行了定义。

11856：该结构的最后一部分是 **union** 类型的；该 **union** 类型依赖于 **si_code** 的值。

11857：**union** 的第一部分是**_pad**，它将 **siginfo_t** 的长度扩展填充为 128***sizeof（int）**字节（在 x86 平台上一共是 512 个字节）。留意一下这个数组的大小，也就是 **SI_PAD_SIZE**（11849 行），代表了该结构的前三个成员——如果增加了更多的成员，**SI_PAD_SIZE** 就需要进行相应修改。

struct signal_queue

17132：**struct signal_queue** 结构用来确保所有的实时信号量都被正确传送了，如果可能，每一个都包含着额外信息（**siginfo_t**）。如同后面你将会看到的一样，内核会为每个进程都设置一个队列，用来存放该进程的挂起的实时信号量。这个队列类型本身很小，仅仅由一个指向下一个节点的指针和 **siginfo_t** 本身组成。

应用函数

有关信号量的一个最重要的数据结构是 **sigset_t**，它是由一系列在 include/linux/signal.h 文件中定义的简单函数所操纵的，这些函数的定义从 17123 行开始。在 x86 平台上，这些相同的函数可以——而且已经——使用汇编语言更加有效的实现了；这些更高效的版本从 12204 行开始。（m68k 端口是唯一一个例外的端口，它使用体系结构特有的代码实现。）由于平台无关的版本和 x86 特有的版本都很重要，我们会对两者都加以介绍。

平台无关的 sigset_t 函数

配合 **sigset_t** 使用的平台无关的函数在 include/linux/sigal.h 文件中，从 17123 行开始。称为“bitops”（位级的操作）的函数将在后面介绍。

sigaddset

17145：**sigaddset** 把一个信号量加入集合——也就是说，它修改了集合中的一位。

17147：为了便于位操作，将基于 0 的信号量转化为基于 1 的信号量。

17149：如果信号量中填入一个无符号长整型数，恰当的位就会被设置。

17151：否则，**sigaddset** 就需要绕很多弯路，首先装入恰当的数组元素，接着设置该元素中相关位。

17148 行的代码和该文件中后面的其它代码一样，第一次见到时可能会令人感到有些困惑。在内核代码中，速度是压倒一切的因素。从而，也许你并不会看到类似于下面的运行期间进行决定的代码：

```
if (_NSIG_WORDS == 1)
    set->sig[0] |=1UL << sig;
else
    set->sig[sig / NSIGBPW] |= 1UL << (sig % NSIGBPW);
```

而你看到的是类似于下面的在编译期间决定的代码：

```
#if (_NSIG_WORDS == 1)
    set->sig[0] |=1UL << sig;
#else
    set->sig[sig / NSIGBPW] |= 1UL << (sig % NSIGBPW);
#endif
```

难道这样不会运行的更快些吗？不要忘了，**if** 条件是能够在编译期间进行计算的，因此预处理器可以使系统没有必要在运行期间执行检测工作。

当你认识到优化工作的实现方式时，这也就没有什么神秘的了。gcc 的优化器的敏锐程度足以注意到 **if** 表达式只有一个出口，因此它可以把那些不必要的代码移走。作为内核“运行期间”版本的结果代码和“编译期间”的版本是等同的。

但是在我们使用优化器很糟糕的编译器时，基于预处理器的版本还会更好吗？这一点并不确定。问题之一是，基于预处理器的（编译期间的）版本有一点更难懂。当代码的复杂程度比前面的简单例子要高时，可读性的差别就会明显的显示出来。例如，让我们考虑一下 **sigemptyse**t 中的从 17264 行开始的 **switch**。现在的 **switch** 类似于这样：

```
swithc (_NSIG_WORDS) {
default:
   memset(set, 0, sizeof(sigset_t));
   break;
case 2:   set->sig[1] = 0;
case 1:   set->sig[0] = 0;
   break;
}
```

（请注意经周密考虑的 case 2 随 case 1 连续执行的情况。）为了更好的利用预处理器而将其重写，它就可能类似于：

```
#if ((_NSIG_WORDS != 2)) && \ (_NSIG_WORDS != 1)
   memset(set, 0, sizeof(sigset_t));
#else /* (_NSIG_WORDS is 2 or 1). */
#if (_NSIG_WORDS == 2)
   set->sig[1] = 0;
#endif
```

```
    set->sog[0] = 0;
#endif /* _NSIG_WORDS test. */
```

gcc 的优化器为两者产生的目标代码是相同的。你更希望读哪一种版本的源程序代码呢？

另外，即使编译器的优化器并没有这么好——这种优化实在相当简单——那么编译器就不可能生成很好的代码。不管我们提供多少帮助都注定是不够的，因此我们可能要编写一些更容易读、更容易维护的代码——这是又一项工程技术的权衡。最后，就象我们在前面的内容中已经看到而且还要不断看到的那样，使用除 gcc 之外的编译器编译内核本身就是个挑战——增加一段 gcc 特有代码不会引起更多问题的。

sigdelset

17154：这些代码和 **sigaddset** 非常类似；区别在于这里从集合中删去了一位——就是把相应的位设置为关。

sigismember

17163：这些代码和 **sigaddset** 也非常类似；这里是要测试某一位是否被设置。注意到 17167 行可能和下面的这种写法有同样的好处：

```
return set->sig[0] & (1UL << sig);
```

这种写法与 17169 行非常相似。虽然这样能够和其它函数的编写风格更加一致，但是这并不是什么改进。

这些修改将对函数的行为方式稍有改动：它现在返回 0 或 1，经过这种修改，就可以在一个位被设置时返回其它的非 0 值。但是，这种改变不会终止没有退出的代码，因为其调用者只关心返回值是否为 0（它们并不特别在意是否为 1）。

sigfindinword

17172：这个函数返回 **word** 中设置的第一个位的位置。函数 **ffz**（在本书中没有涉及）返回其参数中第一个 0 位的位置。在将位求补的字中的第一个 0 的位置——这正是这个函数搜寻的内容——显然是原始顺序中的第一个 1 的位置。它从最小位 0 开始计算。

sigmask

17177：最后，这个有用的 **sigmask** 宏简单的把信号量数目通过一个相应的位集合转化为一个位掩码。

平台相关的 sigset_t 函数

即使平台无关的版本已经使用了简单有效的 C 代码，它也可以通过使用 x86 CPU 家族的方便而功能强大的位集指令在 x86 平台上更加有效地实现，。这些函数中的大部分都可以减少为单独的机器指令，因此这里的讨论也都很精简。

在 x86 平台（例如 m68k）上平台无关的函数对于编译器甚至是不可见的。17126 行包含进了 asm/signal.h 文件，在 x86 上这个文件被分解为 include/asm-i386/signal.h，这都应该归功于设置文件所建立的符号链接。12202 行定义了预处理器符号 **__HAVE_ARCH_SIG_BITOPS**，它消除了这些平台无关的函数的定义（请参看 17140 行）。

sigaddset

12204：x86 特有的使用 **btsl** 指令的 **sigaddset** 实现，它仅对操作数的一个位进行设置。

sigdelset

12210：同样，这是 x86 特有的使用 **btrl** 指令的 **sigdelset** 实现，它对操作数的一个位进行重置（清除）。

sigismember

12233：**sigismember** 根据其 **sig** 参数是否是一个编译期常量表达式来选择实现方法。文档中所没有说明的 gcc 编译器的强大的特殊参数**__builtin_constant_p** 是一个编译期操作符（就象 **sizeof** 一样），它能够报告是否可以在编译期间计算其参数值。

如果可以，**sigismember** 使用**__const_sigismember** 函数（12216 行）完成这项工作，因为它的大部分表达式都可以在编译期间计算。否则就使用更为普遍的版本**__gen_sigismember** 函数（12224 行）来代替。更普遍的版本中使用的是 x86 的 **btl** 指令，它需要测试其操作数中的某一位。

注意到在编译期的常量合并和死锁代码消除通常意味着这样的完整测试只能在编译期间执行——关键是 **sigismember** 要根据需要使用**__const_sigismember** 或者**__gen_sigismember** 替换，在作为结果的目标代码中甚至完全看不出来根本就没有对另一部分进行考虑。这样相当精简，难道不是吗？

sigmask

12238：x86 特有的 **sigmask** 的实现，这与平台无关的版本是等同的。

sigfindinword

12240：最后，x86 特有的 **sigfindinword** 实现只使用了 x86 的 **bsfl** 指令，它在自己的操作数中寻找一个设置位。

设置函数

除了前面的那一组函数之外，还有一组对 **sigset_t** 执行设置操作的函数和宏。和前面一组类似，这些函数使用**__HAVE_ARCH_SIG_SETOPS** 预处理器符号保护起来。然而现在没有一种体系结构能够提供自己独有的这些函数的实现，正因为如此，体系结构无关的版本是现存的唯一版本。

_SIG_SET_BINOP

17184：我们希望定义的全部三个二进制操作——**sigorsets**，**sigandsets** 和 **signandsets**——的实现方式从本质上来说是相同的。这个宏简单的把这三个函数的共同代码分解出来，从而只给它们提供一个操作和一个名字。当然，这同 C++模版函数类似，不过这样我们不得不自己处理一些记录工作，而不能完全信任编译器——这是我们使用 C 工作所付出的一部分代价。

17191：程序开始在 **sigset_t** 中全部四个字节的无符号长整型数的循环，同时对这些操作数进行应用。这个循环是为了速度的原因而展开的——通过减少循环控制开销来提高速度，这是很出名的一种增加速度的方法。然而，大多数情况下，这个循环根本就不执行。例如，在 x86 平台上，编译器可以在运行期间就证实不会执行该循环体，因为截断取整以后，**_NSIG_WORDS**/4 的结果是 0。（回忆一下**_NSIG_WORDS** 在 x86 平台上的值为 2。）

17201：**switch** 从循环末尾处理剩余工作的这行开始。如果在某些平台上**_NSIG_WORDS** 正好为 6，那么该循环就可以执行一次，而且 **switch** 的情况 2 也可以被执行。在 x86

平台上，循环永远不会执行；只有 **switch** 的情况 2 才可能执行。

顺便说一下，我并不清楚为什么 **switch** 不和其类似的**_SOG_SET_OP** 一样使用直接流程的方式实现。通常情况下，现存的版本可以更充分的利用缓存（如果你试图重新编写它，那么你就可以清楚的认识到这一点）——但是如果实际原因的确如此，那么**_SIG_SET_OP** 也应该使用相同的参数。

_SIG_SET_OP

17238：**_SIG_SET_OP** 和**_SIG_SET_BINOP** 类似，但是它使用的是一元操作而不是二元操作，因此我们并不需要详细地介绍它。但是你应该注意的是，这只能使用一次——在 17257 行生成 **signotset**——这和**_SIG_SET_BINOP** 不同。因此，在某种程度上这是不需要的——其实现者可以直接编写 **signotset**，而不必借助**_SIG_SET_OP**，这并没有产生任何重复代码。然而，二者生成的目标代码是相同的，这样如果我们以后选择增加一元操作，意义也就不大了。

sigemptyset

17262：**sigemptyset** 清空所提供的集合——要把其中的每一位都清空。（下面一个函数 **sigfillset** 和这个函数功能相同，不过它要设置所有的位而不是清除所有的位，因此我们就不再详细介绍了。）

17265：普通情况下使用 **memset** 把集合中的每一位都置为 0。

17268：对于**_NSIG_WORDS** 的一些比较小的值来说，简单的直接设置 **sigset_t** 的一两个元素可能速度更快。在这里采用的就是这种直接流程实现。

sigaddsetmask

17292：该函数和下面的几个函数是更简单快速设置和读取最低的 32 位（或者根据字的大小）信号量的一系列函数。**sigaddsetmask** 简单地把 **mask** 所指定的位置位，而不对剩余的位进行任何处理——这是一个集合的联合操作。

siginitset

17310：根据提供的掩码对最低 32 位（或者是别的）置位，并将其它位设置为 0。下面一个函数 **siginitsetinv**（17323 行）正好相反：它根据掩码的补数设置最低 32 位（或者别的），并对其余的位置位。

传送信号量

从用户的观点来看，传送信号量相当简单：调用系统调用 **kill**，该调用只需要进程 ID 号和一个信号量。但是，正如本节中所显示的那样，其实现要复杂得多。

sys_kill

28768：**sys_kill** 是系统调用 **kill** 的一个具有欺骗性的实现样例；真正的实际工作是在 **kill_somethig-info** 中实现的，我们随后就将对这个方面进行研究。**sys_kill** 的参数是要传递的信号量 **sig** 和信号量的目的 **pid**。就象你将看到的那样，参数 **pid** 并不仅是进程 ID。（PID 和进程的其它概念都在第 7 章中详细介绍。）

28770：根据提供给 **sys_kill** 的信息声明并填充 **struct siginfo** 结构。特别要注意的是 **si_code** 是 **SI_USER**（因为只有用户进程才可以调用该系统调用；内核本身是不会调用系统调用的，它更倾向于使用低层函数）。

28778：传递这些信息给 **kill_something_info**，该函数处理实际的工作。

kill_something_info

28484：该函数的参数和 **sys_kill** 类似，但是增加了一项 **siginfo** 结构的指针。

28487：如果 **pid** 为 0，就意味着当前进程希望把信号量传递给整个进程组，该工作由 **kill_pg_info**（28408 行）完成。

28489：如果 **pid** 是-1，信号量（几乎）被送往系统中的每一个进程，这在下面的段落中介绍。

28494：使用 **for_ech_tas**k 宏（在 16898 行宏定义，第 7 章中详细介绍）开始循环处理现存进程列表的每一项。

28496：如果这不是 idle 进程（或 init），就使用 **send_sig_info**（28218 行，后面将会讨论）传递信号量。每次发现合适的任务时 **count** 的值都会增加，虽然 **kill_something_info** 并不关心 **count** 的实际值，而是在意是否能够发现合适的进程。如果所有试图发送信号量的努力都失败了，将记录失败的过程以使得 **kill_something_info** 可以在 28503 行返回错误代码；如果发生了多次错误，则只返回最后一次失败的情况。

28503：如果发现了合适的候选进程，**kill_something_info** 就返回最近失败的错误代码，或者成功就返回 0。如果没有发现任何合适的候选进程，就返回 **ESRCH** 错误。

28504：其它负的 **pid**（是负值，但不是-1）定义了接收信号量的进程组；**pid** 的绝对值是进程组号。和前面一样，**kill_pg_info** 的使用就是出于这种目的。

28506：其它的所有可能性都已经进行了说明；**pid** 必须为正数。在这种情况下，它是信号量传送的目的进程的 PID。这由 **kill_proc_info** 实现（28463 行，很快就会讨论）。

kill_pg_info

28408：这个函数给进程组中的每一个进程发送一个信号量和一个 **struct siginfo** 结构。其函数体和前面介绍的 **kill_something_info** 类似，因此我只是简单介绍一下。

28417：开始循环处理系统中的所有进程。

28418：如果进程在正确的进程组中，那么信号量就发送给它。

28427：如果信号量成功发送给任何进程，**retval** 就设置为 0，从而在 28430 行成功返回。如果信号量不能被发往任何进程，那么要么是所给的进程组中没有进程，在这种情况下，**reval** 仍然会在 28415 行赋值为**-ESRCH**；或者 **kill_pg_info** 发送信号量给一个或多个进程，但是每次都失败了，在这种情况下 **retval** 值为从 **send_sig_info** 得到的最近错误代码。注意它和 **kill_something_info** 的细微区别，后者如果发送信号量失败时就返回错误。但是这里的 **kill_pg_info** 即使在某些情况下出错了，只要信号能成功地传递给任意进程，就会返回成功信息。

28430：在 28410 行中，如果进程组号无效，**retval** 或者是如前所述的赋值，或者就是**-EINVAL**。

kill_proc_info

28463：**kill_proc_info** 是一个相当简单的函数，它把信号量和 **struct siginfo** 结构传递给由 PID 定义的单个进程。

28469：通过所提供的 PID 查找相应的进程；如果成功 **find_task_by_pid**（16570 行）返回一个指向该进程的指针，如果没有找到该进程就返回 **NULL**。

28472：如果找到匹配进程，就使用 **send_sig_info** 把信号量传送给目的进程。

28474：返回错误指示，或者是在 28470 行由于没有发现匹配进程而返回**-ESRCH**，或者是其它情况下从 **send_sig_info** 中返回的值。

send_sig_info

28218：我们最后看的几个函数中最重要的显然是 **send_sig_info**。这个函数使用不同的方法装载进程并处理实际的工作。现在应该了解一下实际的工作是如何完成的。**send_sig_info** 将使用 **info** 指针（该指针也可能为 **NULL**）指向额外信息的信号量 **sig** 传送给 **t** 指针（调用者应该保证 **t** 不会为 **NULL**）指向的进程。

28229：确保 **sig** 在范围之内。注意使用的是如下的测试

sig > _NSIG

而不是你可能预期的

sig >= _NSIG

这是因为信号量的计数是从 1 开始的，而不是从 0 开始的。因此虽然不存在对这个信号量编号的定义，有效信号量的编号的标识符 **_NSIG** 本身也是有效的信号量编号。

28233：这是另外一个严密性检查——实际上包含多个检验。基本的思想是检测信号量的传送是否合法。虽然内核本身可以给任何进程传送信号量，但是除了在涉及 **SIGCONT** 的情况之外，除 root 之外的用户都不能给其它用户的进程传送信号量。总之，这个长长的 **if** 条件说明了如下问题：

- （28233 行）如果不存在补充信息，或者虽然存在补充信息，但是信号量来源于用户而不是内核，并且…
- （28235 行）…信号量不是 **SIGCONT**，或者虽然信号量是 **SIGCONT**，但是并不是传送给同一会话过程中的其它进程，并且…
- （28237 行和 28238 行）…发送者有效的用户 ID 既不是已经存储了的目标进程的用户 ID，也不是目标进程的当前用户 ID，并且…
- （28239 行和 28240 行）…发送者的当前用户 ID 既不是已经存储了的目标进程的用户 ID，也不是目标进程的当前用户 ID，并且…
- （28241 行）…此处不会允许用户超越普通许可（例如，由于用户是 root）…那么就不应该发送信号量了；可以跳过这段发送信号量的代码。

对于前面的 **if** 条件必须明白两点。首先，当将 **info** 映射为无符号长整型数的时候，如果它为 1，这就不是一个实际指向 **struct siginfo** 结构的指针。相反的，它是说明信号量来自于内核的特殊值，但是并没有进一步的附加信息可供使用。内核本身在最低的页（内存页在第 8 章中讨论）中并不分配空间，因此在 4,096 之下的任何地址（除了 0，**NULL** 之外）都可以作为这种特殊值使用。

其次，在这几种条件的情况中，位 XOR 运算操作符（^）比不等运算操作符（**!=**）使用得更为普遍。在这些情况下，两个操作符意义相同，因为如果两个相比较的正数之间有一位不同，在 XOR 运算的结构中就至少有一位被置位，所以结果非空，逻辑值为真。推测起来，cc 的早期版本为^生成的代码比为**!=**生成的代码更为有效，但是在现在的编译器版本中就不是这样了。

28248：忽略信号量 0 并拒绝将信号量传送给僵进程（已经退出但是还尚未从系统的数据结构中移走的进程；请参看第 7 章的“进程状态”一节，它讨论了函数 **exit**）。

28252：对于一些信号量，在实际发送之前必须进行一些额外的工作。这些工作是在这里的 **switch** 中处理的。

28253：如果正在发送 **SIGKLL** 或者 **SIGCONT**，**send_sig_info** 就唤醒进程（也就是说，如果当前被停止了就允许它再次运行）。

28257：设置进程的返回代码为 0——如果进程已经使用 **SIGSTOP** 停止了，返回代码域就被用来在停止等待的信号量和其祖先间建立通讯。

28258：取消任何挂起的 **SIGSTOP**（被调试器阻塞），**SIGSTP**（由键盘输入的 Ctrl+Z 终止），

SIGTTIN（试图从 TTY 中读取信息的后台运行进程），**SIGTTOU**（试图向 TTY 中写入信息的后台运行的进程）；这些是所有可能中断进程的条件，也是 **SIGCONT** 或者 **SIGKILL** 最可能作为响应出现的情况。

28263：在一些信号量被删除之后，调用 **recalc_sigpending**（16654 行，将在后面讨论）来判断是否还有信号量仍然处于挂起状态以等待进程。

28266：在前面的情况中，如果 **SIGCONT** 或者 **SIGKILL** 到达了，这四个信号量就会都被取消。但是看起来有些不太对称，如果这四个信号量有一个到达了，任何挂起等待的 **SIGCONT** 都会被取消。然而 **SIGKILL** 却不会被取消，这遵循 **SIGKILL** 永远不会被锁定或者取消的规律。

28281：如果目标进程希望忽略信号量并且允许不接收信号量，那么就跳过了信号量的接收过程。

28284：非实时信号量并不排队等待，这就意味着如果在进程处理第一个信号量实例之前，同一信号量的第二个实例就到达了，那么第二个实例就会被忽略。这一点就是在这里确保的(回想一下 **struct task_struct** 结构的 **singnal** 成员中保存着一个进程的当前正在挂起等待的信号量的集合)。

28304：在限制条件的控制下，实时信号量需要排队等待。最重要的限制是可以同时排队等待的实时信号量总数的可配置限制；这一限制值为 **max_queued_signal**，它是在 28007 行定义的，而且可以使用 Linux 的系统控制特性加以修改。如果有空间来容纳更多的信号量，就分配 **struct signal_queue** 结构来容纳排队等待的信息。

为什么所要首先限制排队等待的信号量的数目呢？这是为了防止服务拒绝的攻击：如果没有这个限制，用户可以持续发送实时信号量直到内核内存溢出，这样就会阻碍内核为其它进程提供该服务及其它服务。

28310：如果一个队列节点已经被分配，现在 **send_sig_info** 就必须使有关这个信号量的信息进入队列。

28311：把信息加入队列是很直接的：**send_sig_info** 把挂起等待的信号量数量（全局变量）增加 1，接着把新的节点增加到目标进程的信号量队列中。

28315：根据提供给 **send_sig_info** 的 **info** 参数填充队列节点的 **info** 成员。

28316：0（**NULL**）意味着信号量是从用户发送而来的，而且可能使用了从 28513 行到 28544 行定义的向后兼容的信号量发送函数。目标 **siginfo_t** 使用相对比较明确的值来填写。

28323：值 1 是指示信号量来源于内核的一个特殊值——再一次的使用了向后兼容的函数。和前面的情况一样，目标 **siginfo_t** 使用相对比较明确的值来填写。

28331：正常情况下，**send_sig_info** 得到一个实际的 **siginfo_t**，它可以简单地将其拷贝到队列节点中。

28334：没有分配队列节点——或者因为系统内存溢出而造成 **kmem_cache_alloc** 在 28306 行返回 **NULL**；或者因为已经达到了信号量队列的最大值，**send_sig_info** 根本就没有试图分配节点。不管怎样，**send_sig_info** 所处理的内容是相同的：除非该信号量是通过内核或者老式的信号量函数（例如 **kill**）发出的，否则 **send_sig_info** 就返回 **EAGAIN** 错误，通知调用者现在信号量不能排队等待，但是后来调用者应该可以再次使用相同的参数成功执行调用。否则，**send_sig_info** 就传送该信号量但并不将其排入队列中。

28345：最后，**send_sig_info** 从实际上准备好发送信号量。首先，信号量进入该进程的挂起等待的信号量的集合中。注意即使信号量被锁定了这个过程也要执行，这可能有点奇怪。但是这样处理是有原因的：内核必须提供 **sys_sigpending**（28981 行，本章中后面部分将讨论），它允许进程查询在锁定时传送进来什么信号量。

28346：如果信号量没有被锁定，那么进程应该被通知有信号量到达了。相应的，其 **sigpendig** 标志被置位。

28370：如果进程正在等待信号量的到达并且有信号量也正在等待它，那么这个进程就被唤醒（使用 **wake_up_process**，26356 行）来处理信号量。

force_sig_info

28386：这个函数被内核用来保证不管进程是否需要，它都确实接收了一个信号量。例如，在进程释放未用指针时，可以使用这个函数来确保该进程接收了 **SIGSEGV**（请参看 7070 行——实际上是调用了向后兼容的函数 **force_sig**，但是 **force_sig** 完全是按照 **force_sig_info** 实现的）。**force_sig_info** 的参数和 **send_sig_info** 的参数相同，两者的意义也相同。

28392：如果目标进程是僵进程，即使是内核也不应该给它发送任何信号量；所进行的尝试将被拒绝。

28397：如果进程将要忽略这个信号量，**force_sig_info** 将通过强制它执行缺省操作的方式进行纠正。实际上它并不像外表所表现出来的那样无害：在内核使用该函数的情况下，对这个信号量的缺省操作是杀死进程。

28399：把信号量从 **t** 所锁定的集合中移走。

28402：**force_sig_info** 现在已经建立了一些条件使得 **t** 必须接收信号量，因此该信号量就可以使用 **send_sig_info** 进行发送。如果 **send_sig_info** 的实现改变了，这仍然可能造成信号量不能发送，因此这两个函数必须保持同步。

recalc_sigpending

16654：这个函数重新计算进程的 **sigpending** 标志；当进程的 **signal** 或 **blocked** 集合改变时就调用该函数。

16676：在最简单的情况中，**recalc_sigpending** 将信号量和锁定集合求补的结果执行位 AND 操作。（对锁定集合求补就是允许的集合。）其它的情况仅仅是这种情况的泛化。

16679：如果前面操作中的任何一个在 **ready** 中遗留下了任何一位，那么挂起等待的信号量集合中最少有一个信号量现在还被锁定；因此 **recalc_sigpending** 将增加 **sigpending** 标志的值。

由于 **recalc_sigpending** 所实际需要了解的全部内容只是是否至少有一个信号量在挂起等待——例如，如果不止一个，也并不需要知道有多少信号量在挂起等待——非平凡情况下的代码只要发现 **ready** 的值被置为非 0 值就应该停止对其进行修改（例如，前面 16662 行通过中断循环）。但是，任何可能来对此优化所产生的效率增进都必须要同为此而进行的额外测试进行权衡。正是由于这个原因，又加上 **_NSIG_WORDS** 很小（在实际中无论如何都是如此），改进的版本可能要比标准情况快一点。

ignored_signal

28183：**ignored_signal** 有助于 **send_sig_info** 决定是否给一个进程发送信号量。

28189：如果进程正被其祖先跟踪（可能是调试器），或者信号量是在进程锁定的集合中，那么它就不能被忽略。第二种情况可能是我们过去所没有考虑过的；如果信号量被锁定了，**send_sig_info**（还有 **ignored_signal**）难道不应该将其忽略吗？如果情况的确如此，还真不应该忽略。这个函数通过信号量是否应该被忽略表明了进程的信号量的 **signal** 集合的相应位是否应该被置位。如同前面我们已经看到的那样，对 **sigpending** 系统调用的支持要求如果在锁定过程中有信号量到达，内核就应该设置相应的位。因此，被锁定信号量不能简单地忽略。

28194：如果进程是一个僵进程，信号量就应该被忽略。这种测试是不必要的，因为这种情况甚至在 28248 行的 **ignored_signal** 调用之前就会被发现。

28199：在大多数情况下，**SIG_DFL**（缺省的）操作是处理信号量而不是将其忽略。你所能看到的例外是 **SIGCONT**，**SIGWINCH**，**SIGCHLD**，和 **SIGURG**。

28207：进程允许忽略大部分信号量，但是不能忽略 **SIGCHLD**。对于 **SIGCHLD**，POSIX 赋予 **SIG_IGN** 一种特殊的意义，这一点在 28831 行将会说明。这里所提到的“automatic child reaping”（自动子进程空间回收）将在 3426 行执行。

28211：在缺省的情况下，可以假定 **ignored_signal** 有一个实际的函数指针，而不是 **SIG_DFL** 或者 **SIG_IGN** 两个伪值的一个。这样，信号量就和用户定义的处理句柄联系起来，这意味着进程希望处理这个信号量。它通过返回 0 来指明信号量不应该被忽略。

do_signal

3364：**do_signal** 在信号量到达进程时使用。这个函数在内核中被调用的地方不止一次——如我们在第 5 章中看到的从 203 行和 211 行，还有从 2797 行和 2827 行。通常所有这些情况都是当前进程希望处理挂起等待的信号量（如果有的话）。

3375：如果非空，**oldset** 用来返回当前进程锁定的信号量集合。由于 **do_signal** 不会修改锁定的集合，它可以简单的返回一个指向现有锁定集合的指针。

3378：进入几乎扩展到该函数末尾（3478 行）的循环。退出该循环的方法只有两种：把所有可能的信号量都处理了，或者处理唯一一个信号量。

3382：使用 **dequeue_signal** 使信号量出队列（28060，后面将会介绍）。**dequeue_signal** 或者返回 0，或者返回需要处理的信号量的编号，并且它还会填充 **info** 中的附加信息。

3385：如果没有信号量处于等待状态，将在这里中断循环。正常情况下，它在循环第一次执行过程中是不会发生的。

3388：如果当前进程正在被其祖先跟踪（可能是调试器），而且信号量也并不是不可锁定的 **SIGKILL**，那么在信号量到达之前，进程的祖先就必须已经得到通知了。

3391：将传递给子孙进程的信号量编号传送到祖先进程中对应子孙进程的 **exit_code** 域；祖先使用 **sys_wait4** 收集这些信息（23327 行，在第 7 章中介绍）。**do_signal** 停止子孙进程的运行，然后使用 **notify_parent**（3393 行）给祖先进程发送 **SIGCHLD** 信号量，接着调用调度函数 **schedule**（26686 行，第 7 章中介绍），给其它进程——尤其是其祖先进程——运行的机会。**schedule** 会把 CPU 分配给其它进程，因此直到内核跳转回这个进程才会返回。

3397：如果调试器取消了信号量，**do_signal** 在这里就不应该处理它；循环继续进行。

3402：**SIGSTOP** 可能只是由于进程正在被跟踪而产生。这里没有必要处理它；循环继续进行。

3406：如果调试器修改了 **do_signal** 要处理的信号量编号，**do_signal** 将根据新的信息填充 **info**。

3415：正如注释中所说明的一样，如果新的信号量被锁定了，就需要重新排队，循环继续进行。否则，控制流程将直接执行下面的代码。

3421：在这里，或者进程未被跟踪，或者进程正被跟踪但是得到了一个 **SIGKILL** 信号量，或者控制流程直接从前面的代码块中执行下来。在任何一种情况中，**do_signal** 都有一个应该现在处理的信号量。它从获取 **struct k_sigaction** 结构开始，这个结构指明了怎样处理这个信号量编号。

3423：如果进程试图忽略信号量，那么除非这个信号量是 **SIGCHLD**，否则 **do_signal** 就继续执行循环从而忽略该信号量。为什么这个测试不能同时保证该进程不会忽略掉 **SIGKILL** 这个注定不可忽略也不可锁定的信号量呢？答案在于和 **SIGKILL** 相关的

操作永远不会是 **SIG_IGN** 的，实际上也不会是除 **SIG_DFL** 之外的任何操作——28807 行就保证了这一点（在 **do_sigaction** 函数中）。这样，如果操作是 **SIG_IGN**，那么信号量编号就不可能是 **SIGKILL**。

3426：如同在从 28820 行开始的标题注释中说明的一样，POSIX 标准说明了忽略 **SIGCHLD** 的操作实际上意味着自动回收其子孙进程。子孙进程是通过使用 **sys_wait4** 来回收的（23327 行，在第 7 章中介绍），此后循环继续运行。

3435：进程为这个信号量采用缺省操作。专用的初始化进程接收到全部信号量所对应的缺省操作是把信号量整个忽略掉。

3439：对信号量 **SIGCONT**、**SIGCHLD** 和 **SIGWINCH** 所采取的缺省操作是不加处理，只是简单地继续执行循环。

3442：对于信号量 **SIGSTP**，**SIGTTIN** 和 **SIGTTOU**，缺省的操作各自不同。如果该进程所归属的进程组是孤立的——简单的说就是没有连接到 TTY 上——那么 POSIX 规定对于这些基于终端的信号量的缺省操作是将其忽略。如果进程的进程组不是孤立的，缺省的操作是停止进程的运行——这和 **SIGSTOP** 的情况是相同的，在这种情况下控制流程直接向下运行。

3447：在对 **SIGSTOP** 的响应中（或者是从前面情况中直接执行下来），**do_signal** 终止了进程。另外，除非祖先进程已经规定对其子孙进程的终止运行不加理会，否则祖先进程将会在其子孙进程退出时被通知。和 3394 行一样，调用 **schedule** 交出 CPU 给其它某一进程。当内核把 CPU 再次分配给当前进程的时候，该循环继续运行以处理队列中的另外一个信号量。

这不是我们希望的——我认为当 **schedule** 返回时，循环应该退出，因为信号量已经处理完了。其原理在于如果进程被终止了，唤醒进程的最可能的原因是进程又得到了信号量，可能是 **SIGCONT**，因此该进程现在就可以检测并处理信号量了。

3456：对于其它信号量的缺省操作是退出进程。它们中的一些将使进程首先清空内核（详细的介绍请参看第 8 章），这些信号量就是 **SIGQUIT**，**SIGILL**，**SIGTRRAP**，**SIGABRT**，**SIGFPE** 和 **SIGSEGV**。如果此二进制格式（详细的介绍请参看第 7 章）知道如何清空内核并且成功地清空了内核，那么在进程的返回代码中就会有一位被设置来指明进程在退出之前就已经清空了内核。接着流程按照 **default** 的情况继续执行，终止进程的运行。注意 **do_exit**（23267 行，在第 7 章中也会有介绍）是从来不会返回的——因而在 3471 行中会有“NOTREACHED”注释。

3476：此处，**do_signal** 从队列中取出一个信号量，该信号量既不和 **SIG_IGN** 的操作有关，也不和 **SIG_DFL** 的操作有关。唯一的另外一种可能性是这是用户定义的信号量处理函数。**do_signal** 调用 **handle_signal**（3314 行，本章随后将会更为详细地讨论）来触发这个信号量处理函数，接着返回 1 向调用者声明这个信号量已经处理过了。

3481：此处，**do_signal** 不能为当前进程从队列中取出信号量。（只有从 3386 行的 **break** 退出时才能执行到本行。）如果在系统调用的处理过程中间被中断了，**do_signal** 就要调整寄存器，从而系统调用将可以重新执行。

3490：返回 0 以通知调用者 **do_signal** 没有处理任何信号量。

dequeue_signal

28060：**dequeue_signal** 将信号量从进程信号量队列中移出，同时忽略那些由掩码说明的信号量。它返回信号量的编号并使用指针参数 **info** 返回相关的 **siginfo_t**。

28071：为了避免重复参照而建立一些别名：**s** 是进程的挂起等待的信号量的集合（记住其中可能包括了一些锁定的信号量），**m** 是掩码的集合。特别要注意的是*s 表达式，在该函数中这个表达式出现了不止一次，但是它只是 **current->signal.sig[0]**的一种简

单写法。

28073：在这个 **switch** 条件分支中，**sig** 被设置为第一个挂起等待的信号量。从最简单的情况入手最容易理解；其它的情况只是这种情况的泛化。

28091：最简单的情况是：它把挂起等待的信号量和掩码求补后的结果进行位 AND 运算，结果被存储在临时变量 **x** 中；**x** 现在就是掩码不能忽略的挂起等待的信号量的集合。如果 **x** 不为 0，那么就存在挂起等待的信号量（**x** 至少有一位被置位）；**dequeue_signal** 使用 **ffz**（本书中没有涉及）取得相应的信号量编号，并将其转化为从 1 开始计数的信号量编号，将结果存储在 **sig** 中。正如前面所说明的一样，其它情况只是这种情况的泛化；最重要的结果是 **sig** 被置位，如果可能的话在每种情况下都是如此——此后其它变量（**i**，**s**，**m** 或者 **x**）的状态就不难理解了。如果在 **switch** 之后的 **sig** 是 0，掩码中就没有传递挂起等待的信号量。

28097：如果一个信号量正在挂起等待，那么 **dequeue_signal** 应该试图将其从队列中释放出来。**reset** 跟踪 **dequeue_signal** 以判定是否应该把信号量从进程的挂起等待的信号量队列中删除。将 **reset** 初始化为 1 仅仅是由于在函数处理过程中它可能会改变的假定。

28107：对于非实时信号量，内核不会保持原始的 **siginfo_t**（如果曾经有过的话），因此 **dequeue_signal** 应该尽可能的重新组织有关的信息。不幸的是，当前实现方法中并没有多少信息——只有信号量编号自身而已。**info** 的其它成员都简单地被设置为 0。

28118：在另一种情况，也就是实时信号量情况下，**siginfo_t** 只是一种点缀。**dequeue_signal** 会在进程的 sigqueue 中进行扫描以确定其值。

28122：如果找到了 **siginfo_t**，**dequeue_signal** 现在就使其出队列，将 **siginfo_t** 的内容拷贝到 **info** 中，并释放为这个队列节点分配的内存。

28129：如果队列中没有这个信号量的更多实例，那么信号量就不会在挂起等待了。但是为了弄清楚队列中是否还有信号量的实例，**dequeue_signal** 需要遍历整个队列。因此，该函数需要扫描这个队列的其余元素来查询是否存在相同信号量的其它实例。如果发现了实例，**dequeue_signal** 就清空 **reset** 标志——只有在这种独特的情况下才会进行的操作。

28142：正在出队的信号量是实时信号量，但是在进程的挂起等待的实时信号量队列中却没有发现它，其原因在代码中已经进行了阐述。现在，**dequeue_signal** 和它有非实时信号量的情况相同了——它知道信号量是可以访问的，但是没有方法可以访问其原始值——并且其响应过程处理的工作也完全相同，仅仅使用信号量编号来填充 **info**，而没有其它属性值。

28150：除非 **reset** 标志被清空了——也就是说除非这是一个实时信号量并且同一个信号量的其它实例仍然在挂起等待队列中——该信号量已经被处理过；它应该从进程的挂起等待集合中删除。

28152：信号量脱离队列，因此 **dequeue_signal** 应该重新计算进程的 **sigpending** 标志。我认为这里有一个可以进行少量优化的机会：只用当 **reset** 为真的时候 **dequeue_signal** 才需要这样处理。**recalc_sigpending** 从进程的锁定集合和挂起等待集合中计算结果；锁定的集合没有改变，因此只有当挂起等待的集合发生改变时，**dequeue_signal** 才需要调用 **recalc_sigpending**。如果 **reset** 为假，挂起等待的集合就不会改变，因此对于 **recalc_sigpending** 的调用就是不必要的。

28163：**switch** 没有发现信号量，因此没有信号量正在挂起等待。作为内部正确性的检测，**dequeue_signal** 确保内核不会认为有信号量正在为某任务挂起等待。

28174：返回出队的信号量编号，或者如果没有信号量出队，就返回 0。

notify_parent

28548：**notify_parent** 寻找进程的祖先进程并通知它其子孙进程的状态发生了改变——通常情况是其子孙进程或者被终止了，或者被杀死了。

28553：使用有关信号量发生的上下文的信息填充局部变量 **info**。

28564：如果子孙进程已经退出，**why** 被赋以适当的值以指明其原因是因为它清空了内核，或者被某信号量将其杀死，或者因为执行了非法操作。

28572：类似地，如果使用信号量终止了进程，对 **why** 赋值以说明发生的情况。

28578：前面的情况几乎覆盖了所有的可能性。如果没有，函数打印出警告信息并继续运行；在这种情况下，系统会在 28562 行将 **why** 的值赋为 **SI_KERNEL**。

28586：给进程的祖先进程发送信号量。下面一行唤醒任何等待这个子孙进程的进程并为其提供 **CPU**。

handle_signal

3314：　**handle_signal** 在需要调用用户定义的信号量处理程序时由 **do_signal** 调用。

3338：　建立一个用户处理程序可以在其中运行的堆栈帧。如果进程已经请求了内核所拥有的有关信号量的原始值和其上下文的附加信息，那么堆栈帧就使用 **setup_rt_frame**（3231 行）构建起来；否则就使用 **setup_frame**（3161 行）构建。这两种方法都可以实现构建工作，这样控制流程会返回信号量处理程序。当它返回时，实际返回的是信号量到达的时候正在执行的代码。

3343：　如果 **SA_ONESHOT** 标志被设置，则信号量处理程序应该只执行一次。（注意 **sys_signal** 是 **signal** 系统调用的实现，它使用 **SA_ONESHOT** 类型的信号量处理程序——请参看 29063 行。）在这种情况中，缺省的操作是立即将其恢复。

3346：　**SA_NODEFER** 意味着在执行这个信号量的处理程序时，不应该有其它信号量被锁定。如果位没有设置，其它的位现在就会被加入进程的锁定的集合中。

其它有关信号量的函数

其它一些有关信号量处理的函数。

sys_sigpending

28981：这个简短的系统调用允许进程询问在信号量锁定期间是否有非实时信号量到达。通过所提供的指针，该函数返回一个位集以指明它们是哪些信号量。

28987：这个函数的核心是进程的 **blocked** 集合和 **signal** 集合间的简单位 AND 操作。它只对最低 32 位感兴趣，这些都是非实时信号量。

28992：使用所提供的指针把挂起等待的集合拷贝回用户空间。如果失败就返回**-EFAULT**，如果成功就返回 0。注意是否有信号量正在挂起等待——也就是说，返回值是否为空——并不是成功的判据之一。

do_sigaction

28801：**do_sigaction** 实现了系统调用 **sigaction** 有意义的部分。（其余部分在 2833 行的 **sys_sigaction** 中。）**sigaction** 是 POSIX 中等价于 ISO C 的函数 **signal**——它把信号量和操作关联起来，这样进程接收到信号量时就能够执行相应的操作。

28806：健全性检测：确保 **sig** 在范围之内并且进程没有试图把 **SIGKILL** 或者 **SIGCONT** 和某种操作相关联。进程被简单地剥夺了覆盖这两个信号量的缺省操作的权力。然而，

和 **signal** 实现的处理程序不同，使用 **sigaction** 实现的处理程序不是 **SA_ONESHOT** 类型的，因此在处理程序被调用的时候就不用每次都将其重新装载。

28811：获取和这个信号量相关的指向 **k_sigaction** 结构的指针。

28813：**sigaction** 可以通过一个过去所提供的指针返回旧有的操作。这在以堆栈方式存在的处理程序中是很有用的，在这里处理程序被临时覆盖，以后再恢复出来。如果 **oact** 指针非空，旧有的操作就会被拷贝到其中。（但是这并不会把信息拷贝会用户空间；调用者必须执行这样的处理。）

28815：如果 **do_sigaction** 被赋予一个需要同信号量相关联的操作，那么二者现在就相互关联起来。**SIGKILL** 和 **SIGSTOP** 也必须被从操作的掩码中删除，为了确保这些信号量不会被锁定或者覆盖。

28836：正如在 28820 行开始的标题中注释的一样，为了遵守 POSIX 标准，下面的几行代码必须要经过一定变形，并且在必要情况下还会舍弃某些信号量。对于这些细节情况，我们即使跳过也不会有什么损失。

sys_rt_sigtimedwait

28694：**sys_rt_sigtimedwait** 等待信号量的到达，它可能在经过一段特定的时间间隔以后超时退出。并不是所有的信号量都会接收；指针 **uthese** 所指明的 **sigset_t** 说明了调用者所感兴趣的信号量。

28714：**uthese**（它已经被拷贝到局部变量 **these** 中了）是允许的信号量的集合，于是内核元语只知道如何锁定信号量。但这样也没有关系：对允许的信号量集合进行求补运算就得到了应该锁定的信号量，所得到的结果就可以直接使用了。

28717：如果调用者提供了超时时间，该超时时间就将被拷贝到用户空间中，而且其值也必须经过健全检测。

28726：检查是否已经有信号量正在等待了——如果有，就没有必要为其等待了。否则，调用者必须等待。

28731：保存原来的锁定信号量集合，然后阻塞由 **these** 定义的所有信号量。

28737：如果用户没有提供超时时间，那么超时时间会是 **MAX_SCHEDULE_TIMEOUT**（宏定义为 **LONG_MAX**，或者是 2^{31}-1，16228 行）。但是并不永远都是如此——超时时间是以瞬间（jiffy）计数的，它的系统时钟以每秒 100 次的速度跳动着，因此大约有 248 天，超时时间就耗尽了。（在 64 位机器中，这大约需要三十亿年。）

28739：如果用户确实提供了超时时间，就将其转化为以瞬间计算的值。“+”后面的表达式是对下一个瞬间进行向上舍入的明智方法——其思想是 **timespec_to_jiffies** 可能已经向下舍入了，但是内核必须是上舍入的，因为它必须等够用户请求的瞬间个数。它虽然可以检测 **timespec_to_jiffies**（18357 行）是否是下舍入的，但是下面这种方法更为简单：如果用户提供的超时时间不是 0 就为其增加一个瞬间，并且认为是对它进行了调整。毕竟 Linux 不是一个真正的实时操作系统——当你指定了超时时间时，Linux 只能保证至少等待如此长的时间。

28742：设置当前用户的状态为 **TASK_INTERRUPTIBLE**（请参看第 7 章）。**schedule_timeout**（26577 行）用来让出 CPU；在指定的时间用完以后或者其它事件到达并唤醒进程（比如接收了一个信号量）时，该进程才可以继续运行。

28746：进程希望被信号量唤醒。**sys_rt_sigtimedwait** 再次尝试从进程的等待信号量队列中取出信号量并恢复原来锁定的集合。

28752：此处，该函数仍然不知道信号量是否已经到达了——它可能无需等待就可以得到一个信号量，或者在等待期间可能有另一个信号量到达，也或者该函数一直在等待但是没有信号量到达。

28753：如果信号量到达，该函数就给用户进程传递信息并且返回信号量编号。

28759：否则，虽然进行了等待，但是没有信号量到达。在这种情况下，该函数或者返回**-EAGAIN**（说明用户进程可以再次使用相同的参数尝试），或者返回**-EINTR**（说明其等待过程被由于某些原因而不能传递的信号量中断了）。

内核如何区分实时信号量和非实时信号量

简单的说，答案并不复杂。我几乎掩盖了其中的绝大部分区别，这是有一定原因的：退出语句不多。现在，为了使这一点更加清楚，让我们来看一下系统调用 **sigprocmask** 的两个版本，它允许进程处理自己的锁定信号量的集合——增加，删除，或者简单地对信号量集合进行设置。

sys_sigprocmask

28931：**sys_sigprocmask** 是这个函数的原始版本，这一版本并不知道或者是不关心实时信号量。参数 **how** 指明了要执行的操作；如果 **set** 不为 **NULL**，就是这个操作的操作数；如果 **oset** 是非空的，那么 **oset** 返回的就是原始的锁定集合。

28937：如果 **set** 为空，那么 **how** 的值就没有什么用处了：该操作就没有操作数了，因此该函数不会处理有关的内容。否则，就继续执行该操作。

28939：在新的锁定集合中的拷贝，其中删除了不可锁定的 **SIGKILL** 和 **SIGSTOP**。

28944：为了处理以后将当前锁定集合拷贝回用户空间的需要，在 **old_set** 中存储当前锁定集合的一个备份。由于当前锁定集合在以后的代码中可能会被修改，因此在它改变之前必须对其值进行存储。

28948：当然是忽略无效的操作。

28951：**SIG_BLOCK** 操作符指明 **new_set** 应该解释为要锁定的附加信号量的集合。这些信号量将被加入该锁定集合中。

28954：**SIG_UNBLOCK** 操作符指明 **new_set** 应该解释为要从锁定的信号量的集合移出的信号量集合。这些信号量现在被移出锁定集合。

28957：**SIG_SETMASK** 操作符指明 **new_set** 应该解释为新的锁定集合，简单覆盖该锁定集合原有的值。因此，**sys_sigprocmask** 正是实现这一点的。注意它只设置了 **bloched.set** 数组的最低的元素——这个元素包含低 32 位非实时信号量，这是该函数所关心的内容。

28966：如果调用者已经请求查询锁定的集合的原来的值，执行流程就向前跳到 **set_old** 标号（28970 行）。

28968：如果 **set** 为空，意味着调用者没有请求对锁定集合进行修改，但是调用者可能仍然希望了解锁定集合的当前值。

28972：**oset** 非空（**set** 也可能为非空）。不管哪一种情况，**old_set** 都包含一个原来锁定的集合的备份，在返回之前 **sys_sigprocmask** 会试图将其拷贝回用户空间。

sys_rt_sigprocmask

28612：**sys_rt_sigprocmask** 和 **sys_sigprocmask** 非常类似，但是它也能够处理新的实时信号量。由于这两者之间的相似性，在这里我仅仅介绍一下它们之间有趣的区别。

28638：与如下代码不相类似的是

```
/* how sys_sigprocmask does SIG_BLOCK.*/
    new_set =    *set ;          /* line 28938 */
```

```
    blocked |= new_set ;       /* line 28952 */
```

（作为一个例子采用 **SIG_BLOCK** 的情况），实际代码类似如下代码：

```
/* how sys_rt_sigprocmask does SIG_BLOCK.*/
    new_set =    *set ;          /* line 28625 */
    new_set |= old_set ;       /* line 28639 */
    blocked |= new_set ;       /* line 28648 */
```

我不明白为什么 **sys_rt_sigprocmask** 不使用和 **sys_sigprocmask** 相同的方式实现，而且这样还可以节约一点效率。

中断

中断的名字十分形象，因为它们终止了系统正常的处理过程。在前面第 5 章中你就已经看到了中断的一个例子：提供系统调用基本机制的软件中断。在本章中，我们来了解一下硬件中断。

和系统调用中断一样，硬件中断也可能转化为内核模式运行然后返回。如果用户进程运行时发生了中断，系统就转化为内核模式，并且内核要对中断做出响应。接着，内核将控制返回给用户进程，用户进程能够从当时离开的位置继续运行。

同系统调用中断的另一个区别是硬件中断可能在内核已经在内核模式下运行时发生。这在系统调用中很少发生——通常内核不会麻烦地触发系统调用中断，因为它可以直接调用目标内核函数。如果中断发生时系统处于内核模式，结果就同在用户模式下的机制相一致——唯一的区别是内核自身所特有的执行过程而不是用户进程的执行过程暂时地被中断。

如果内核在一段时期内不希望被中断，那么就可以使用 **cli** 和 **sti** 函数（13105 行和 13104 行是 UP 版本；1216 行和 1229 行是 SMP 版本）屏蔽和开启中断。这些函数根据底层的 x86 指令命名：**cli** 代表“清除中断标志”，**sti** 代表“设置中断标志”。其工作方式和其名称类似：CPU 有一个“中断允许”标志，如果对其置位就允许中断，如果将其清空就禁止中断。因此，你可以使用 **cli** 清空这个标志从而禁止中断，也可以使用 **sti** 设置这个标志从而允许中断。在 UP 代码中，你可以选择调用两个等价的宏**__cli** 和**__sti**——分别见 13105 行和 13104 行。

当然，把内核移植到非 x86 平台上会使用不同的底层指令——在这些体系结构中 **cli** 和 **sti** 函数的实现都不相同。

IRQs

IRQ，或者中断请求，是从硬件设备发往 CPU 的中断信号。作为对 IRQ 的响应，CPU 跳转到某个地址——中断服务例行程序（***ISR***），更普通的情况是调用中断处理程序——内核在前面已经对这些处理程序进行了登记。中断处理程序是内核执行的为中断服务的函数；从中断处理程序中返回就继续执行中断前所在位置的代码。

IRQ 是有编号的，每一个硬件设备在系统中都对应一个 IRQ 号码。例如在 IBM PC 体系结构中，IRQ 0 就关联着一个每秒产生 100 次中断的定时器。把 IRQ 号码和设备关联起来，使得 CPU 可以区分每个中断是哪个设备产生的，从而允许它跳转到正确的中断处理程序。（在某些情况中，在一个系统中一个 IRQ 号可以被多个设备所共用，当然这不是非常普遍的情况。）

Bottom Halves

中断处理程序的下半部分（bottom half）是无须立即执行的部分。在某些中断之后，你甚至可能根本就不需要执行它。

给定的中断处理程序从概念上可以被分为上半部分（top half）和下半部分（bottom half）；在中断发生时上半部分的处理过程立即执行，但是下半部分（如果有的话）却推迟执行。这是通过把上半部分和下半部分处理为独立的函数并对其区别对待实现的。总之，上半部分要决定其相关的下半部分是否需要执行。不能推迟的部分显然不会属于下半部分，但是可以推迟的部分只是可能属于下半部分。

你也许会很奇怪为什么 Linux 会辛苦地把它们区分开——为什么要延迟呢？一个原因是要把中断的总延迟时间最小化。Linux 内核定义了两种类型的中断，快速的和慢速的，这两者之间的一个区别是慢速中断自身还可以被中断，而快速中断则不能。因此，当处理快速中断时，如果有其它中断到达——不管是快速中断还是慢速中断——它们都必须等待。为了尽可能快地处理这些其它的中断，内核就需要尽可能地将处理延迟到下半部分执行。

另外一个原因是，在最低层，当内核执行上半部分时，中断控制芯片将被告知禁止正在服务的这个特殊 IRQ（这和 CPU 级别的中断禁止不同，它把快速中断和慢速中断区别开来）。我们并不希望这种状态会持续地比需要的时间还长，因此只有上半部分中时间最为关键的部分才被处理，但是下半部分中其它的工作就要延迟处理了。

区分上下部分还有一个原因是处理程序的下半部分包含有一些中断所不一定非要处理的操作，只要内核可以在一系列设备中断之后可以从某些地方得到。在这种情况下，执行对于每个中断的下半部分的处理完全是一种浪费，它可以稍稍延迟并在后来只执行一次。

最后一段的一个暗示是值得说明的：没有必要每次中断都调用下半部分。相反，是上半部分（或者也可能是其它代码）简单地标记下半部分，通过设置某一位来指明下半部分必须执行。如果下半部分已经标记过需要执行了，现在又再次标记，那么内核就简单地保持这个标记；当情况允许的时候，内核就对它进行处理。如果在内核有机会运行其下半部分之前给定的设备就已经发生了 100 次中断，那么内核的上半部分就运行 100 次，下半部分运行 1 次。

下半部分在内核中有时候被认为是“软 IRQ”或者“软中断处理程序”，这有助于你理解今后要遇到的一些文件名和术语。

在本节的剩余内容中，我们将保持下半部分概念的抽象。下一节深入介绍定时器中断，包括其下半部分的处理，并展示了下半部分概念的一个有趣的滥用现象——我的意思是一个有趣的变种。

数据结构

同对信号量的处理一样，我们首先介绍一下中断和下半部分使用的重要的数据结构。图 6.1 阐述了这些数据类型之间的关系。

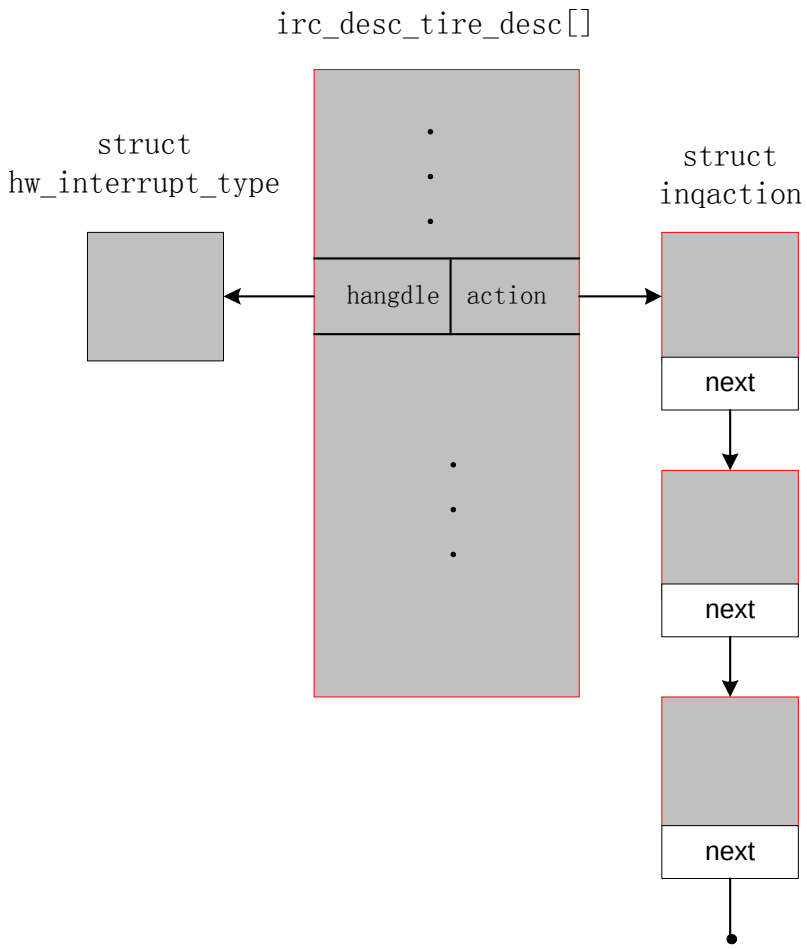

图 6.1 有关中断的数据结构

我们从这里开始，体系结构无关的头文件 linux/interrupt.h 定义了 **struct irqaction** 结构（14844 行），它代表了内核接收到特定 IRQ 之后应该采取的操作（在本章后面的部分中你将看到 **struct irqaction** 结构是如何与 IRQ 关联的）。其成员如下：

- **handler**——指向某一函数的指针，该函数是作为对中断的响应所执行的操作。
- **flags**——从与前面已经介绍过了的 **sa_flags** 相同的集合中提取出来；这个集合从 12108 行开始。该集合中仅仅为此目的而出现的值只有 **SA_INTERRUPT**（使用另外一个中断来中断这个中断也是可以的），**SA_SAMPLE_RANDOM**（考虑到这个中断也是源于物理随机性），和 **SA_SHIRQ**（这个 IRQ 和其它 **struct irqaction** 共享）。
- **mask**——在 x86 或者体系结构无关的代码中不会使用（除非将其设置为 0）；看起来只有在 SPARC64 的移植版本中要跟踪有关软盘的信息时才会使用它。
- **name**——生成中断的硬件设备的名字。由于不止一个硬件可以共享一个 IRQ，这在打印人工阅读程序时就有助于区分它们。
- **dev_id**——标识硬件类型的一个唯一的 ID——Linux 支持的所有硬件设备的每一种类型都有一个由制造厂商定义的在此成员中记录的设备 ID。其所有的可能值都是从一个巨大的集合中抽取出来的，这个集合在本书中没有介绍，因为它包含的内容是十分繁琐的，而且都是重复的——它仅仅是结构上类似于下面一小段代码的巨大宏定义块。

```
#define PCI_DEVICE_ID_S3_868 0x8880
#define PCI_DEVICE_ID_S3_928 0x88b0
#define PCI_DEVICE_ID_S3_864_1 0x88c0
#define PCI_DEVICE_ID_S3_864_2 0x88c2
```

在你看完这些的一部分之后，也就相当于将其完整的看了一下。可能你已经发现了，摘录的这部分内容是从包含针对基于 S3 的 PCI 显卡的设备 ID 的文件中选取的。虽然 **dev_id** 是一个指针，可它并不指向任何内容，但若将其解除参照就会引起错误。能够说明问题的是它的位结构模式。

- **next**——如果 IRQ 是共享的，那么这就是指向队列中下一个 **struct irqaction** 结构的指针。通常情况下，IRQ 不是共享的，因此这个成员就为空。

接下来我们感兴趣的两个数据结构存在于体系结构相关的文件 arch/i386/kernel/irq.h 中。第一个是 **struct hw_interrupt_type** 结构（1673 行），它是一个抽象的中断控制器。这是一系列的指向函数的指针，这些函数处理控制器特有的操作：

- **typename**——赋给控制器的人工可读的名字。
- **startup**——允许从给定的控制器的 IRQ 所产生的事件。
- **shutdown**——禁止从给定的控制器的 IRQ 所产生的事件。
- **handle**——根据提供给该函数的 IRQ 处理唯一的中断。
- **enable** 和 **disable**——这两个函数基本上和 **startup** 和 **shutdown** 相同；存在的差异对于本书中涉及的代码都不很重要。（实际上，对于本书中包含的所有代码来说，**enable**/**disable** 函数对和 **startup**/**shutdown** 函数对都是相同的。）

这个文件中我们感兴趣的另外一个数据结构是 **irq_desc_t**（1698 行），它具有如下成员：

- **status**——一个整数，它的位或者为 0，或者对应从 1685 行到 1689 行定义的集合中抽取出的标志。这些标志的集合代表了 IRQ 的状态——IRQ 是否被禁止了，有关 IRQ 的设备当前是否正被自动检测，等等。
- **handler**——指向 **hw_interrupt_type** 的指针。
- **action**——指向 **irqaction** 结构组成的队列的头。如同前面说明的一样，正常情况下每个 IRQ 只有一个操作，因此链接列表的正常长度是 1（或者 0）。但是，如果 IRQ 被两个或者多个设备所共享，那么这个队列中就有多个操作了。
- **depth**——**irq_desc_t** 的当前用户的个数。主要是用来保证事件正在处理的过程中 IRQ 不会被禁止。

irq_desc_t 是在 **irq_desc** 数组中（733 行）积聚起来的。对于每一个 IRQ 都有一个数组入口，因此数组把每一个 IRQ 映射到和它相关的处理程序和 **irq_desc_t** 中的其它信息上。

最后一个需要说明的数据结构集合从 29094 行开始；这些都与前面所讨论的下半部分有关：

- **bh_mask_count**（29094 行）——跟踪为每个下半部分提出的 **enable**/**disable** 请求嵌套对的数组。这些请求通过调用 **enable_bh**（12575 行）和 **disable_bh**（12568 行）实现。每个禁止请求都增加计数器；每个使能请求都减小计数器。当计数器达到 0 时，所有未完成的禁止语句都已经被使能语句所匹配了，因此下半部分最终被重新使能。
- **bh_mask** 和 **bh_active**（14856 行和 14857 行）——它们共同控制下半部分是否运行。它们两个都有 32 位，而每一个下半部分都占用一位。当一个上半部分（或者一些其它代码）决定其下半部分需要运行时，就通过设置 **bh_active**（12498 行中使用 **mark_bh**）中的一位来标记下半部分。不管是否经过了这样的标记，下半部

分可能会通过清空 **bh_mask** 中的相关位来整个跳过——通过调整 **bh_mask_count** 入口，**enable_bh** 和 **disable_bh** 完成了这个功能。

因此，对 **bh_mask** 和 **bh_active** 进行位 AND 运算就能够表明应该运行哪一个下半部分。特别是如果位与运算的结果是 0，就没有下半部分需要运行。这种技术在内核中多次使用，例如在宏 **get_active_bhs**（12480 行）中就使用了这种技术

- **bh_base**（14858 行）——这是一组简单的指向下半部分函数的指针。
- 未命名的 **enum**——从 14866 行开始的未命名的 **enum** 为内核使用的每一个下半部分指定了一个符号名称。例如，为了把计数器的下半部分标记为活动的，你可以这样的语句：

  mark_bh（TIME_BH）；

  27450 行的确就是这样处理的。

操作和 IRQ

一个经过仔细选择的小型函数集合处理了操作和 IRQ 之间的链接和解除链接。本节就是要讨论这些函数，以及那些从整体上对 IRQ 系统进行初始化的函数。

init_ IRQ

1597： **init_ IRQ** 初始化 IRQ 的处理。

1601： 符号 **CONFIG_X86_ISWS_APIC** 是为 SGI 虚拟工作站以及 SGI 的基于 x86 的工作站流水线而设置的。虽然同样基于 x86 的 CPU，虚拟工作站不能和基于 IBM PC 的体系结构共享很多其它特性——特别是如同你看到的，它们的中断处理有些不同。我们以后将忽略虚拟工作站所特有的代码。

1609： 建立中断描述符表，给 32 项到 95 项（十进制）赋缺省值。在这个过程中使用了 **set_nitr_gate**（6647 行），该函数很快就会介绍到。

1651： 建立 IRQ 2（级联中断）和 IRQ13（为 FPU 使用——请参看 955 行）。和这两个 IRQ 有关的 **irqaction** 结构分别是 **irq2**（979 行）和 **irq13**（974 行）。

init_ISA_irqs

1578： 该函数填充 **irq_desc** 数组，为 ISA 总线类型的机器（也就是所有标准 PC）初始化所有 IRQ。虽然该函数没有声明为 **static** 类型的，也没有使用**__initfunc** 标签标记，但是它只会被 **init_ IRQ** 调用。因此，只有在内核初始化过程中这个函数才是必要的。

1583： 对 **irq_desc** 中的每一个元素，系统为 **status**，**action** 和 **depth** 成员赋与了不会惹人反对的，也不会使人吃惊的的缺省值。

1589： 原来的（在 PCI 之前）IRQ 使用 **i8259A_irq_type**（723 行）处理。

1592： 编号比较高的 IRQ 初始化为 **no_irq_type**（701 行），这是一个必要的空处理程序。后来它们可能会改变——实际上，如果你使用了 PCI 卡，就确实会改变，就象现在的大多数 PC 一样。

set_intr_gate

6647： **set_intr_gate** 在 x86CPU 的中断描述符表（IDT）中建立一个项。在基于 x86 的系统中发生的每一个软件中断和硬件中断都有一个编号，这个编号被 CPU 用作是对这个表的索引。（包括系统调用中断——编号为 0x80——在第 5 章中我们已经介绍过了。）表中相关的项是中断发生时（内核）函数需要跳转到的地址。

setup_x86_irq

1388：**setup_x86_irq** 给指定的 IRQ 增加了一个操作（一个 **struct irqaction** 结构）。例如，在 6088 行使用它来记录定时器的中断。它还可以通过 **request_irq**（1439 行）使用，这在下一节介绍。

1398： Linux 使用了几种物理的随机源——例如中断——把一系列不可预知的值提供给设备/dev/random，这是一个有限却具有很高随机性的数据源，还有/dev/urandom，这是对应/dev/random 的无限的但是随机性较小的对应版本。随机系统作为一个整体在本书中并没有涉及，但是如果你不知道这个概念，这一大部分代码就会显得十分神秘。

1412： 如果现存的操作列表非空，**setup_x86_irq** 必须保证现存的操作和新的操作可以共享这个 IRQ。

1414： 验证这个 IRQ 可以和其上现存的 **struct irqaction** 结构共享。这种测试是十分有效的，它部分是基于我们的一些认识：没有必要遍历执行队列中的所有操作，也没有必要检测它们可能共享的所有情况。除非这两个操作和第一个操作都允许共享 IRQ，否则不会允许第一个操作后的所有操作都进入队列。因此，如果第一个操作可以共享 IRQ，那么队列中的其它操作也就可以共享 IRQ；如果第一个操作不能共享，那么队列中的其它任何操作也都不能共享 IRQ。

1420： IRQ 正在被共享。**setup_x86_irq** 利用 **p** 向前执行操作队列直到末尾，离开时 **p** 指向队列的最后一个元素的 **next** 域。它也会增加 **shared** 标志的值，这将会在 1429 行中被使用。

1427： **p** 现在指向队列中的最后一个元素的 **next** 域，如果要共享 IRQ，或者 **p** 在不共享的情况下指向 **irq_desc[irq].action**——指向队列的头节点的指针。不管怎样，指针现在被设置为新的元素了。

1429： 如果还没有操作和这个 IRQ 关联，**irq_desc[irq]**的其它部分也就还没有设置，在这里就需要对其初始化了。特别要注意 1433 行中为这个 IRQ 调用了 **startup** 函数。

request_irq

1439：**request_irq** 从提供的值中创建一个 **struct irqaction** 结构，并将其加入对应给定的 IRQ 的 **struct irqaction** 列表中。（如果你对 C++和 Java 比较熟悉，可以把它当作是操作的构造函数。）它的实现非常简单明了。

1448： 对一对输入值进行健全性检测。注意没有必要测试 **irq** 是否小于 0，因为它是一个无符号数。

1453： 动态分配新的 **struct irqaction** 结构。为此目的使用的函数 **kmalloc** 在第 8 章中简单介绍。

1458： 填充新的操作并使用 **setup_x86_irq** 将其加入操作列表。

free_irq

1472： **free_irq** 是 **request_irq** 的补数（inverse）。如果 **request_irq** 类似于操作的构造函数，那么这就是操作的析构函数。

1481： 在确保 **irq** 在范围内以后，**free_irq** 找到有关的 **irq_desc** 项并且开始遍历操作列表。

1483： 除非它有正确的设备 ID，否则就忽略这个队列元素。

1487： 把这个元素从队列中分离出来并且释放其所占用的内存。

1489： 如果现在操作队列为空——也就是如果队列中只有唯一一个元素没有被链接——设备就会被关闭。

1495： 如果控制流程执行到这里，就意味着 **free_irq** 处理了整个操作列表而没有发现匹配

的 **dev_id**。如果发现了匹配对象，1493 行的 **goto** 语句就已经跳过了本行。因此，这个试图释放 IRQ 操作的努力是错误的；在这种情况下 **free_irq** 会打印出一条警告信息对当前状况进行描述。

prove_irq_on

1506：**probe_irq_on** 实现了内核 IRQ 自动探测的重要的一部分。阅读 14889 行开始的标题注释就得到了对整个进程的描述。根据描述我们知道这里要作的工作（只）是执行步骤三：暂时使能所有没有定义的 IRQ，以使得 **probe_irq_on** 的调用者可以检测它们。

1514：　对于除 IRQ 0 之外的每一个 IRQ，如果这个 IRQ 还没有与之相关的操作，**probe_irq_on** 会记录下这个 IRQ 正在自动探测的事实并启动关联设备。顺便说明一下，我不认为有任何的原因使这个循环向后执行。

1524：忙等待约十分之一秒时间以允许生成伪中断的设备取消自己。

1530：循环再次遍历所有的 IRQ，这一次要过滤出所有生成伪中断的设备。这个循环每次重复执行都是从 1 开始而不是从 0 开始，这是因为不需要自动检测的 IRQ 都被忽略掉了，而 IRQ0 是从来都不会自动检测的。速度在这里也是一个问题；在十分之一秒的延时之后——这是很长的一段时间，即使是从慢速的 CPU 的观点来看也是如此——一个循环或多或少都是有些不合理的。

1537：如果设备在 1524 行的等待过程中触发了中断，这个中断可能就是伪中断：在此期间系统应该还没有和设备通讯过，因此设备也应该还没有和系统通讯过。因此自动探测位将被清空，处理程序再次关闭。

1544：返回特殊数字 0x12345678，其原因将在下面的讨论中进行说明。

prove_irq_off

1547：**prove_irq_off** 实现了 IRQ 自动探测的另外一部分重要的内容。这里的工作是决定对探测到的哪一个 IRQ 做出响应，并返回其中的一个 IRQ。

1551：检测名字很容易让人误解的参数 **unused** 和 **probe_irq_on** 返回的特殊数字是否相同。调用者假定象下面这样处理：

```
magic = prove_irq_on（）
/* … */
probe_irq_off（magic）;
```

如果偶然使用了其它的方法调用了 **probe_irq_off**（例如，如果由于其它一些逻辑调用者偶尔跳过了对 **probe_irq_on** 的调用），那么提供的参数可能不会包含正确的值。可能更重要的是这个参数给正在编写代码使用这个函数的程序员提供了一些信息：在研究其参数应该是什么的时候，你会发现在调用它的时候一直遵守的规则。这种规则很容易就被过度使用。

通过对紧随的错误消息的严格调整，似乎该函数的早期版本中可能已经在其参数中采用了调用者的地址。如果的确如此，这个测试就具有了第三种目的：把仍然不正确使用这个函数的调用者检测出来。

1557：循环遍历所有的 IRQ，搜寻响应调用者探测的所有设备。这个循环也可以从 1 开始循环，这和前面讨论的 **probe_irq_on** 的原因是相同的。

1560：内核没有试图自动检测这个 IRQ 上的任何内容；它跳到了下一个 IRQ。

1563：**IRQ_INPROGRESS** 标志指明了该 IRQ 的一个中断已经到达。由于 **probe_irq_on** 可能捕获所有的伪中断，假定这是对探测的真实响应。成功地自动检测到 IRQ 的数量因此而增一，同时保存第一次的数字。

1568：　不管是否成功自动探测到 IRQ，自动探测标志都要减少，并且再次结束处理程序。
1573：　如果不止一个 IRQ 被成功地自动探测到，就通过否定的 **irq_found** 来通知调用者。
1575：　返回 **irq_found**——0，或者（可能是经过求反的）第一个成功地自动探测到的 IRQ 号。注意如果发现了设备，则返回值决不会是 0，因为内核不会试图自动检测 IRQ 0。因此当没有自动检测到 IRQ 时，**probe_irq_off** 就返回 0。

硬件中断处理程序和下半部分

x86 系列的实际中断处理程序是微不足道的；在最低的层次上，这是通过反复使用 **BUILD_IRQ** 宏（1886 行）建立了一系列小汇编函数而实现的。**BUILD_ IRQ** 自己被 **BI** 宏调用（866 行），这个宏又顺次被 **BUILD_16_IRQS** 宏（869 行）调用，该宏在 878 行到 895 行的代码中用来建立汇编程序。这一连串的宏调用的目的仅仅是试图减少必须编写的代码数量和复杂度——我们应该使用 256 次对 **BUILD_IRQ** 的调用，而不是 16 次对 **BUILD_16_IRQS** 的调用。

汇编程序和如下代码相类似：

```
IRQ0x00_interrupt:
  push1 0x00-256
  jmp common_interrupt
```

也就是每一次都简单地把它的 IRQ 号（减去 256，原因在 1897 行有论述）压入堆栈并且跳转到正常的中断程序。

正常的中断处理程序是调用 **common_interrupt**，该函数也十分简短。它是使用 **BUILD_COMMON_IRQ** 宏（1871 行）建立的，在为 **do_IRQ** 进行安排之后简单调用 **do_IRQ** 返回给 **from_intr**（233 行）——这是第 5 章中介绍的系统调用的一部分。随后将要介绍的 **do_ IRQ**（1362 行）负责查看中断是否已经被处理了。

在介绍这些代码之前，从总体上观察一下在处理单个中断时这些部分如何组织在一起是很有帮助的：

1. CPU 跳转到 **IRQ0x*NN*_interrupt** 程序（其中的 *NN* 是中断号），它将其唯一的中断号压入堆栈并跳转到 **common_interrupt**。
2. **common_interrupt** 调用 **do_IRQ** 并保证当 **do_IRQ** 返回时控制流程能够转向 **ret_from_intr**。
3. **do_IRQ** 调用中断处理器芯片独有的代码——直接和芯片通讯的代码，如果需要就用它来处理中断。对于 PC 体系结构中流行的 8259A 控制器芯片，处理函数是 **do_8259A_IRQ**，在这里提到它仅仅是为了举一个例子而已。
4. **do_8259A_IRQ** 暂时禁止正在处理的特殊 IRQ，调用 **handle_IRQ_event**，接着重新使能这个 IRQ。
5. **handle_IRQ_event** 为慢速 IRQ 使能中断，或者为处理快速 IRQ 而将这些中断保持在禁止状态。接着遍历一个已经和这个 IRQ 建立联系的函数队列，并依次调用这些函数。由于中断为慢速 IRQ 而使能，这里就是慢速 IRQ 的处理程序可能被其它中断所中断的地方。在执行完队列中的所有函数之后，**handle_IRQ_event** 禁止中断并返回控制器所特有的处理函数，而该函数将返回到 **do_IRQ**。
6. **do_IRQ** 处理所有挂起等待的下半部分，接着返回。如同你已经知道的那样，它要返回到 **ret_from_intr**。第 5 章中介绍了从此之后的处理内容。

do_IRQ

1375：更新内核的一些统计数字并调用与该 IRQ 相关的处理函数。对于一些老式 PC 上的标号较小的 IRQ 来说，其处理程序是 **handle_IRQ_event**。

1383：如果下半部分是激活的，内核现在就使用 **do_bottom_half**（29126 行）对它们进行处理。

do_IRQ_event

1292：**do_IRQ_event** 为 **do_8259A_IRQ**（821 行）负担了大半重要的工作。本书中没有涉及到的其它一些代码也会调用这个函数。

1302：与其描述（12094 行）正好相反，**SA_INTERRUPT** 标志并不是一个 no_op。如果没有设置该标志，在接下来的代码中就允许中断。这是快速中断和慢速中断之间历史上遗留下来的区别，这一点我们已经讨论过了。（处理这两种类型中断的代码通常有很多差异，但是结果是相同的——代码已经被处理得更加出色了。）恰当的说，这个标志似乎是大多数情况下都为慢速设备使用——顾名思义，就是软盘设备。

1305：通过调用每一个的处理函数来遍历执行这个 IRQ 的操作队列（队列头是由调用者提供的）。

1310：这里的中断触发是用来为/dev/random 和/dev/urandom 增加一些随机信息——从大体上看来，大部分中断都是随机发生的。

1312：禁止中断（当条件具备时调用者可以再次允许这些中断）。

do_bottom_half

29126：Linux 代码中有三处调用了 **do_bottom_half**：26707 行，243 行，1384 行。（你会发现，其中的两个是在体系结构特有的文件中的；在非 x86 平台体系结构特有的文件中对应部分也会调用这个函数。）因此，下半部分是用来处理如下三种情况的：

- 当决定随后哪一个进程应该获得 CPU 时。
- 当从系统调用中返回时。
- 在从 **do_IRQ** 中返回之前——也就是说，在每个中断之后。代码中的注释暗示了在内核的未来版本中不一定总在这里运行下半部分。

29130：下半部分的一个理想特性是在某一时刻只能有一个下半部分处于运行状态。这种特性在这里，也就是锁定（locking）在 UP 代码中体现出重要意义的位置之一，得到了强化。首先调用 **soft_trylock**（UP 版本在 12559 行——第 10 章中可以查看所有的 SMP 版本），只有在 **local_bh_count[cpu]**原来为 0 时这个函数才把 **local_bh_count [cpu]**设置为 1 并返回真值。根据 17479 行，对于 UP 来说 **cpu** 总是 0，而且你应该注意到 **softirq_trylock** 自己是不能被中断的，因为在这里中断已经被禁止了。**softirq_trylock** 和对应的 **softirq_endlock**（12561 行），就是仅仅为了以下目的而退出的，而不存在其它原因：协助保证这个下半部分不会被其它下半部分中断（虽然它们可以被上半部分所中断）。

29131：如果成功获得了锁，那么该函数就尝试另一个函数，10736 行的 **hardirq_trylock**。它只报告当前执行进程是否位于 **hardirq_enter/hardirq_exit** 对（10739 行和 10740 行）之间。对于 UP，这两者的定义是和 **irq_enter** 与 **irq_exit**（1810 行和 1811 行）两者的定义相同的；后两个函数在 **handle_IRQ_event** 中使用，当然在其它我们所没有讨论到的地方也对它们有所引用。这些宏协同工作来保证**__cli** 和**__sti** 对能够正确的进行嵌套——由于 CPU 不会嵌套使用这些宏，我们必须保证不会使用**__sti** 处理其它的**__cli**，而这也不是我们所希望的。

29132：没有其它下半部分在运行，而且 **do_bottom_half** 有权使能硬件中断。因此，它就使能硬件中断，运行下半部分，接着再次禁止中断。

29135：释放该函数已经获取的锁并返回。

run_bottom_halves

29110：现在内核能够运行挂起等待的下半部分。

29115：存储当前局部变量 **active** 中活动的——也就是被标记过的——下半部分的集合，并使用 **clear_active_bh** 宏（12481 行）清空全局变量 **bh_active** 中的设置位。对 **bh_active** 中这些位的清除将同时取消对所有下半部分的标记。

现在你就应该可以看到下半部分有时候是批量处理的，这一点在前面已经进行了没有论证的说明。此处中断是被使能的，因此如果在 **run_bottom_halves** 把 **bh_active** 拷贝到 **active** 之前就有中断触发并标记已经标记过的下半部分，那么上半部分就已经运行了两次，然而下半部分只运行了一次。还有，由于这个中断可以被自己中断，在下半部分运行一次的时候上半部分就已经运行了三次，等等。但是随着递归调用数量的增长，这很快就会变得不再可能了。

代码有可能忽略某个下半部分吗？假定中断在最坏的时刻发生：在 29115 行和 29116 行之间——也就是在拷贝 **bh_active** 之后，但是在清空其中的设置位之前。下面是三种可能的情况：

- *新的中断没有标记下半部分*。这种情况显然不会引起什么问题——中断之后处理的下半部分集合和前面的集合相同，因此 **run_bottom_halves** 仍将运行所有它应该运行的下半部分。
- *新的中断标记了一个已经标记过了的下半部分*。这种情况也不会引起问题——**run_bottom_halves** 不管怎样都要运行这个下半部分，而且在新中断返回之后就运行它。
- *新的中断标记了一个前面没有标记过的下半部分*。在这种情况下，当 **run_bottom_halves** 遍历执行所有的下半部分时，**active** 就不再和 **bh_active** 匹配了。然而，由于 **clear_active_bhs** 只会清空 **active** 集合中设置的位，所以 29116 行不会清空 **bh_active** 中新近标记的位。**clear_active_bhs** 使用 **atomic_clear_mask**（10262 行），后者简单的对 **active** 集合中的设置位进行位 AND 运算，而并不处理其余部分。因此，当 **run_bottom_halves** 执行循环时，就不会立刻对新近标记的下半部分进行处理；但是由于它的位仍然在 **bh_active** 中设置，**run_bottom_halves** 就仍然会在最后对它进行处理。更为准确的说法是，通过这种方法跳过的下半部分会在处理随后的一个定时器中断的过程中一起处理，这只是一个瞬间的延迟而已——或者如果有其它中断首先发生了，那么这段时间延迟会更短。因此，迷途的下半部分通常的等待时间不会超过百分之一秒，而且根据定义，下半部分毕竟不是时效性要求非常高的，这种少量延时不会引起任何问题。

29118：同时遍历执行 **bh_base** 数组和 **active** 中的位。如果 **active** 中最低的位被设置了，就调用相关的下半部分；接着循环推进到下一个 **bh_base** 项和 **active** 中的下一位继续执行。

因为这是一个 **do/while** 循环，所以它最少执行一次。部分原因是由于在调用 **do_bottom_half** 之前，调用者都要检测是否有下半部分需要处理，所以这个循环最少要执行一次。在一些情况下这种检测会执行到 **do_bottom_half** 本身中，但是如果没有下半部分需要运行，在调用之前执行测试就能够节省函数调用的开销。不管怎样，我们很容易就可以看出即使没有下半部分需要运行，这个循环也可以正确执行；虽然这会浪费一些时间，但是不这样就会引起错误。

29123：当 **active** 中没有任何位被设置时，循环就终止退出。由于在循环执行的过程中 **active** 是不断移位的，这样就同时测试了其余的位，而没有必要对它们的每一个都进行循环处理。

时间

本节通过观察一个中断的例子——定时器中断——的工作方式来使你能够将中断和下半部分的知识融会贯通起来。

定时器中断函数 **timer_interrupt** 是和 6086 行的 IRQ 0 相关的。此处使用的 **irq0** 变量是在 5937 行定义的。27972 行通过使用 **init_bh**（12484 行）把 **timer_bh** 函数注册为定时器的下半部分。

当触发 IRQ 0 时，**timer_interrupt** 从 CPU 时间戳计数器中读取一些属性值，如果 CPU 中有值（这在本书中没有涉及到的一些代码中使用），就调用 **do_timer_interrupt**（5758 行）。除了其它一些工作之外，它会调用 **do_timer**，这是定时器中断非常有趣的一部分。

do_timer

27446：从我们的出发点来看，这是我们感兴趣的定时器的上半部分。

27448：更新全局变量 **jiffies**，这个值记录了机器启动以来系统时钟跳动的次数。（显然，这实际上记录的是从定时器中断装载以来已经经过的定时器跳动的次数，定时器中断在系统启动的瞬间是不会发生的。）

27449：递增丢失的定时器跳动的数目——也就是那些没有被下半部分处理的定时器的跳动。很快你就会看到有定时器的下半部分是怎样使用这个变量的。

27450：上半部分已经运行了，因此其下半部分被标记为只要可能就运行。

27451：除了要记录从上一次定时器的下半部分运行以来定时器跳动发生的次数之外，我们还要知道有多少这种跳动发生在系统模式下。很快我们会再次看到为什么下半部分需要它；但是在目前，如果进程运行在系统（内核）模式下而不是用户模式下，那么只需要递增 **lost_ticks_system** 的值。

27453：如果定时器队列中有任务在等待，定时器队列的下半部分被标记为准备好运行（我们很快就会对定时器队列进行讨论）。而这就是整个定时器中断。虽然这看起来十分简单，但是这很大程度上是由于主要的工作都适当的延迟到下半部分处理了。

timer_bh

27439：这是定时器的下半部分。它调用函数为进程和内核本身更新有关时间的统计数字，并同时对老式内核定时器和新式内核定时器进行处理。

update_times

27412：这个函数主要是更新统计数字：计算系统的平均负载，更新记录当前时间的全局变量，并更新内核的当前进程使用的 CPU 时间的估计值。

27422：取得从上次下半部分运行以来发生的定时器跳动的数目，并重置计数器。

27427：如果数字非空——正常情况下都是这样——**update_times** 会找出有多少这种跳动发生在系统模式下。

27429：调用 **calc_load**（27135 行，马上就介绍）更新内核有关系统负载要素的估计值。

27430：调用 **calc_wall_time**（27311 行，后面会讨论）更新 **xtime**，它记录了当前的 wall_clock 时间。

27433：调用 **update_process_times**（27382 行），该函数同一个辅助函数 **update_one_process**（27371 行）共同作用，更新内核中有关当前进程已经运行的时间长度的统计。这些统计数字可以使用诸如 time，top，和 ps 之类的普通程序得到。现在你就可以看出，这些统计资料未必一定要正确：如果一个进程能够做到在所有时钟中断触发的时候都不处于运行状态，那么它就可以偷偷的使用大部分 CPU 而且还不会被认为使用了任何 CPU 资源。然而，还是比较难以理解为什么（或者怎样）恶意的进程要试图执行它，对于意图良好的进程来说，统计资料自然要进行平均——它们要为一些自己几乎不能使用的 CPU 定时器跳动负责，但是却不用为那些自己几乎在整个定时器跳动期间都在使用 CPU 的另一种情况负责，但是这些最终都不存在了。

update_wall_time

27313：为每一个必须要处理的跳动调用 **update_wall_time_one_tick**（27271 行）。它更新了全局变量 **xtime**，如果可能，就通过遵守网络时间协议（Network Time Protocol）努力将其和实际时间保持同步。

27318：将 **xtime** 标准化，使得微秒数在 0 到 999,999 的范围内。在极端的情况下可能会丢失多于一秒的定时器跳动，那么 **xtime** 的 **tv_usec** 部分就可能会超出 2 百万，这段代码就可能在标准化 **xtime** 时失败。但是随后的调用可以把 **xtime** 完全标准化。

calc_load

27135：**calc_load** 从定时器的下半部分中调用以更新内核对于当前系统负载的估计值。虽然对于它的介绍有点离题，但是对于每个对这个随处可见的数字是如何计算的感到疑惑的人都会对这个函数很感兴趣，因此它还是值得一看的。

27138：静态变量 **count** 记录了从上次计算平均负载以来已经经过了多少时间。它被初始化为 **LOAD_FREQ**，这在 16164 行中进行宏定义以代表相当于 5 秒时间的定时器间隔。对于这个函数的每一个调用都要递减遗留到下一次计算负载的定时器跳动数量。

27142：当剩余的跳动数小于 0 时，就应该重新计算了。（我倒是希望这个数是小于或者等于 0，而不是仅仅小于 0，但是实际上这两个值差不了多少。）

27143：为了能在另一 5 秒内可以再次触发，重新初始化 **count**，此后，**count_active_tasks**（27119 行）被用来监视系统中当前有多少任务。现在 **count_active_tasks** 的实现可能已经不是什么奇妙的事情了，但是你所有有关它的问题在阅读完第 7 章后都应该得到解答。

27144：在 16169 行定义的 **CALC_LOAD** 宏用来更新 **avenrun** 数组（27116 行）的三个项，它的三个元素分别记录了前面 5 秒、10 秒、15 秒的系统负载。与前一章中说明的一样，内核中尽量避免浮点数运算，因此这些计算都是在固定点进行的。

run_old_timers

27077：内核提供了类似于下半部分的已经不再使用了的技巧，通过这种技巧内核函数可以被登记到表中，和超时时间建立联系，并在指定时间到达时调用。这个函数根据需要调用其它函数。由于现在使用这个函数的唯一目的是为了支持原来的代码，所以我就不仔细地介绍它了。它简单的遍历处理 **timer_table** 数组（27109 行）的列表项，如果定时器已经触发就调用相关函数。

定时器队列

定时器队列最初的背景思想是与上半部分相关的下半部分并不一定非要与中断处理有关。相反的，它也可能是我们所需要周期性处理的任何内容。将一个函数定义为定时器中断处理程序下半部分的一部分能够保证这个函数每秒钟大约被调用 100 次。（如果每秒运行 100 次太频繁了，那么这个函数可以保持一个计数器并每 10 次调用都简单返回 9 次，例如——我们看到 **calc_load** 就是这样处理的。）结果就像是创建了一个进程，它的 **main** 部分在无穷循环中调用这个函数，于是它没有占用通常的和进程相关的开销。

但是，下半部分是一种有限的资源——只存在 32 个，这是因为我们希望 **bh_mask** 和 **bh_active** 各自都匹配一个无符号长整型数。我们可以通过使用系统中类似于实现信号量的方法来扩展下半部分的数量，但是这样仅仅能增加静态可用的下半部分的数量——它并不能使我们能够动态地扩展下半部分列表。

从本质上说，这是定时器队列所提供的内容：一个动态的可增长的下半部分的列表，所有项都和定时器中断有关。这里有一个独立的下半部分以处理这种情况——**TQUEUE_BH**——如同你看到的一样，如果定时器队列中有任务，它就和 **TIMER_BH** 一起被标记。因此，定时器中断有两个下半部分。

定时器队列实际上只是更普通的内核特性——任务队列的一个实例。根据说明文档，任务队列在内核本身中是相当自由的——请参看文件 include/linux/tqueue.h 从 18527 行开始的部分。因此，接下来我们就不再介绍它们了。但是，它们是很重要的内核服务，而且那个简短的文件也很值得一读。

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。

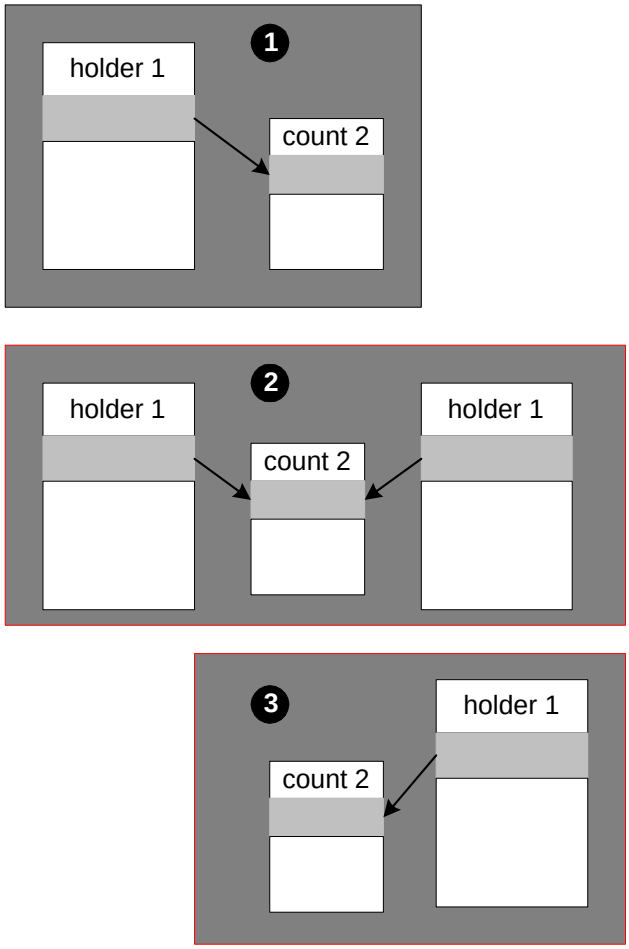

图 7.1 引用计数

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。

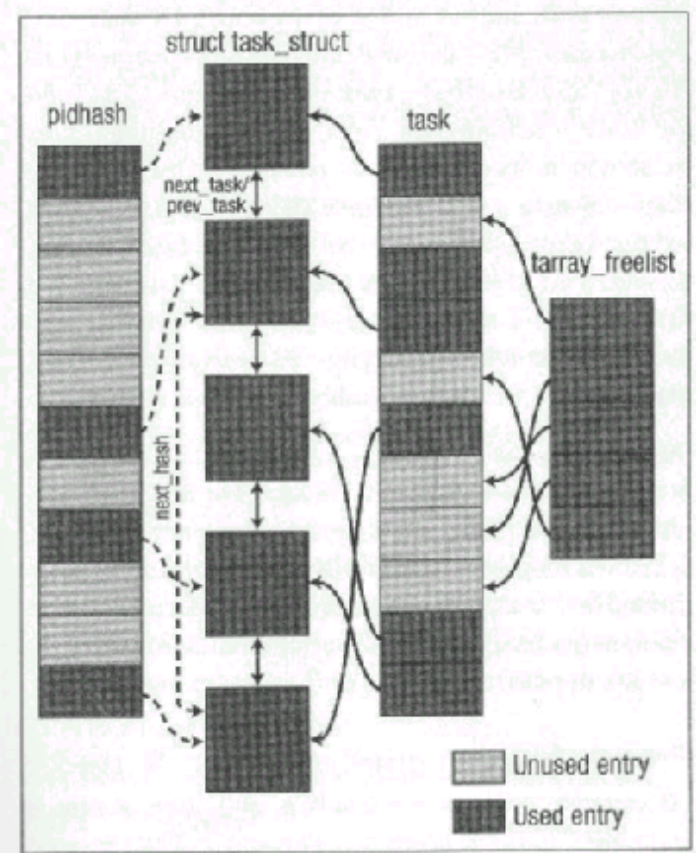

图 7.2 管理任务使用的内核数据结构

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。

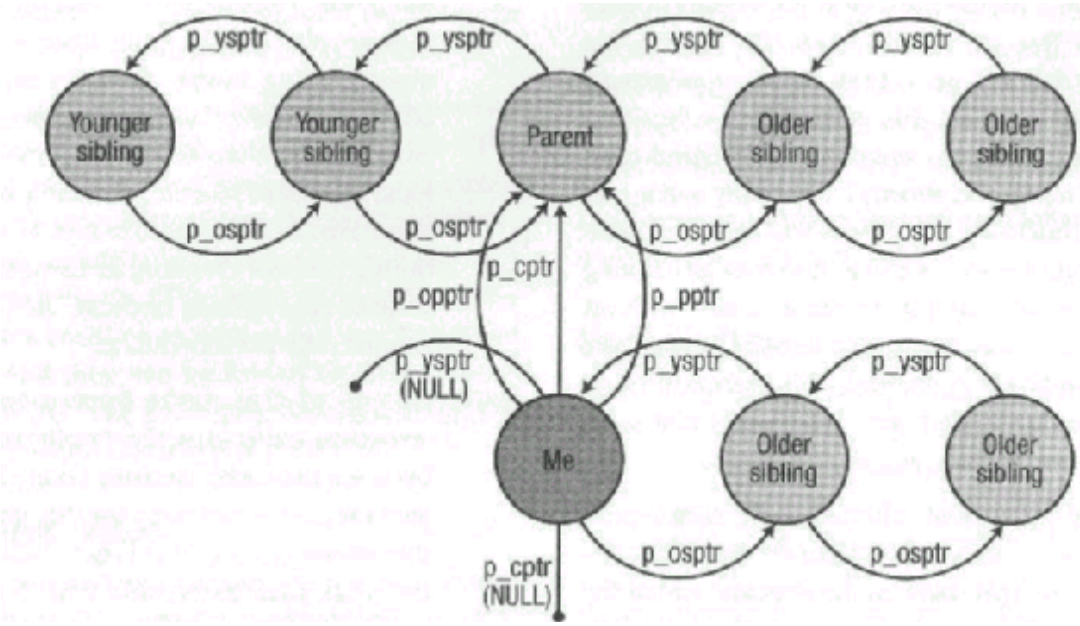

图 7.3　进程图

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。

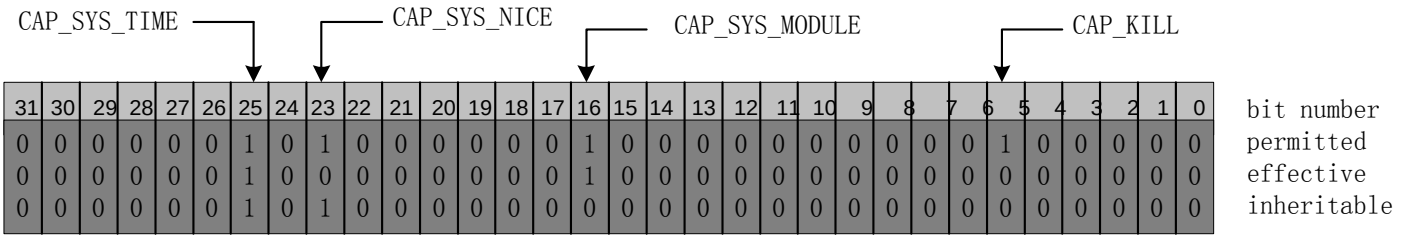

图 7.4 权能集

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。

第 8 章 内存

内存是内核所管理的最重要的资源之一。某进程区别于其它进程的一个特征是两个进程存在于逻辑上相互独立的内存空间（与之相反，线程共享内存）。即使进程都是同一程序的实例，比如，两个 xterm 或两个 Emacse，内核都会为每个进程安排内存空间，使得它们看起来像是在系统之上运行的唯一进程。当一个进程不可能偶然或恶意的修改其它进程的执行空间时，系统的安全性和稳定性就会得到增强。

内核也生存在它自己的内存空间之中，即内核空间（kernel space）。与之对应的是用户空间（user space），它是所有非内核任务所处的内存空间的一个通用术语。

虚拟内存

计算机系统包括不同级别的存储器。图 8-1 说明了这些存储器中最重要的几项，并且以我自己原有的 Linux 机器（Linux box）为例标注了一些参数的估计值。当你从左向右观察该图时，会发现存储器容量越来越大而速度却越来越慢（而且每字节价格也会更低）。尤其令人注意的是，访问速度跨越了 3 个数量级（乘数因子为 1000），而容量竟跨越了超过 8 个数量级（乘数因子为 312500000）。（实际上有时速度的差异是可以被掩盖的，不过这些数字足以很好的说明这一部分讨论的目的。）最大的差距体现在最后两个：RAM 和磁盘上，它们又分别可被称作主存和辅存。

额外附加的存储器空间总是十分诱人的，即使它们也很慢。如果在 RAM 被用完时，通过暂时把不用的代码和数据转移到磁盘上以腾出更多空间的方法来使用磁盘代替 RAM 的话，那将是很好的一件事情。正如读者可能已经知道的，Linux 恰好能够做到这一点，这被称之为虚拟内存（virtual memory）。

虚拟内存是一种对 RAM 和磁盘（或称之为：主存和辅存）进行无缝混合访问的技术。所有这些（虚拟）内存对于应用程序来说就好像它真的存在一样。当然我们知道它并非真的内存，这正是为什么它被称为是“虚拟的”，但是多亏了内核使得应用程序无法分辨出它们的区别。对于应用程序来说，就好像真的有很大数量的 RAM，只不过有时候比较慢而已。

术语“虚拟内存”还有另外一层意思，从严格意义来讲是与前述的第一种意思没有关系的。这里的虚拟内存指的是对进程驻留地址进行欺骗的方法。每个进程都会有这样一种错觉，认为它的地址是从 0 开始并由此连续向上发展的。很明显，这一点同时对所有进程都成立是不可能的，但是在生成代码的时候这个假定（fiction）却能够带来很大方便，这是由于进程不必知道它们是否真正从 0 地址开始驻留，而且它们也不必去关心此事。

这两种意思也不必相关，因为一个操作系统从理论上可以给每个进程分配一个独有的逻辑地址空间而不用混合使用主存和辅存。然而在所有我已经知道的系统中（对这两种虚拟内存的实现方式）要么都采纳要么都不采纳，这一点可能会在开始时令人感到困惑。

为了避免这种意义上的分歧，有人倾向于术语“虚拟内存”代表逻辑地址空间（logical-address-space）的意义，同时使用“分页（paging）”或“交换”表示磁盘作为内存使用（disk-as-memory）的含义。尽管这种严格的区分具有充足的理由，但是我更喜欢普通的用法。除非上下文要求，否则我很少花费精力对它们进行区分。

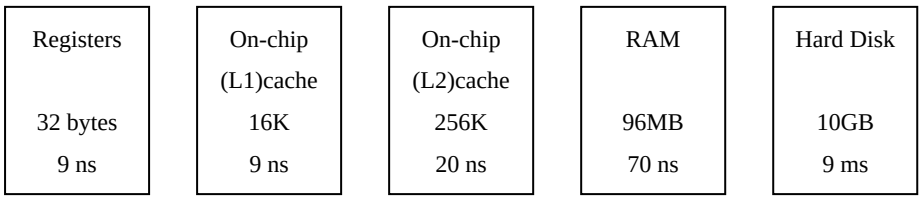

图 8-1 具有速度和容量的存储级别

交换和分页

早期的虚拟内存（VM）系统仅能够把整个应用程序代码和数据，即完整的进程从磁盘上移出或移入磁盘。这种技术被称为交换（swapping），因为它是把一个进程同另一个进程进行了对调。出于这个原因，磁盘上为 VM 所保留的区域通常被称为交换空间（swap space），或简称为交换区（swap），尽管如我们所见，现代的系统已不再使用这种最初意义上的交换技术。与此类似，读者通常会见到的术语是交换设备（swap device）和交换分区（swap partition），它是磁盘分区的同义词，但是被专门作为交换空间使用，以及术语交换文件（swap file），这是一个用于交换的规则的、有固定长度的文件。

交换是很有用的，当然要比根本没有 VM 好的多，但是它也有一定局限性。首先，交换需要把整个进程同时调入内存，所以当运行一个需要比系统所有 RAM 还要大的存储空间的进程时，交换便于事无补了，即使磁盘有大量空间可供补充。

其次，交换可能会很低效。交换就必须把整个进程同时调出，这就意味着为了 2K 的空间你不得不把一个 8MB 的进程整个调出。同样的道理，即使仅仅需要执行被调进的应用程序代码的一小部分，你也必须把整个进程同时调进。

分页（paging）是把系统的内存划分成很小的块，即页面，每个页面可以独立的从磁盘调入或调出磁盘。分页与交换技术相似，但它使用更加细小的粒度（granularity）。分页比交换有更多的登记（book-keeping）开销，这是因为页面数远比进程数要多，然而通过分页可以获得更多的灵活性。而且分页也更快一些，原因之一就是不再需要把整个进程调进调出，而只需要交换必要的页面就足够了。要记住前述的 1000 倍的速度差异，所以我们应该尽可能避免磁盘的 I/O 操作。

传统上特定平台上页面的大小是固定的，比如 x86 平台为 4K，这可以简化分页操作。不过，大多数 CPU 为可变大小的页面提供硬件支持，通常能够达到 4M 或者更大。可变大小页面可以使分页操作执行更快和更有效，不过要以复杂性为代价。标准发行的 Linux 内核不支持可变大小页面，所以我们仍然假定页面大小是 4K。（已经有支持 Cyrix 可变大小页面机制的补丁程序，但它们不是本书中官方发行版本的部分。而且据闻由此获得的性能增益也并不非常显著。）

因为分页可以完成交换所能完成的所有工作，而且更加有效，所以类似于 Linux 一样的现代操作系统已不再使用交换，严格的说是只使用分页技术。但是术语“交换”已得到了广泛使用，以至于实际应用中术语“交换”和“分页”已经几乎可以通用；由于内核使用分页技术，所以本书就遵从这种用法。

Linux 能够交换到一个专用磁盘分区、或一个文件，或是分区和文件的不同组合。Linux 甚至允许在系统运行时增加和移去交换空间，当你暂时需要额外大量的交换空间，或者假如你发现需要额外交换空间而又不想重启系统的时候，这就会很有用了。另外，与一些 Unix 的风格（flavors）不同，Linux 即使没有任何交换空间也能运行得很好。

地址空间

地址空间（address space）是一段表示内存位置的地址范围。地址空间有三种：

- 物理地址空间
- 线性地址空间
- 逻辑地址空间，也被称为虚拟地址空间

（需要指出的是，I/O 地址能够被看作是第四种地址空间，但是本书中对其不作讨论。）

物理地址是一个系统中可用的真实的硬件地址。假如一个系统有 64M 内存，它的合法地址范围是从 0 到 0x4000000（以十六进制表示）。每个地址都对应于所安装的 SIMMs 中的一组晶体管，而且对应于处理器地址总线上的一组特定信号。

分页可以在一个进程的生存期里，把它或它的片段移入或者移出不同的物理内存区域（或不同物理地址）。这正是进程被分配一个逻辑地址空间的原因之一。就任何特定的进程来说，从 0 开始扩展到十六进制地址 0xc0000000 共 3GB 的地址空间是绰绰有余的。即使每个进程有相同的逻辑地址空间，相应进程的物理地址也都是不同的，因此它们不会彼此重叠。

从内核的角度看来，逻辑和物理地址都被划分成页面。因此，就像我们所说的逻辑和物理地址一样，可以称它们为逻辑和物理页面：每个合法的逻辑地址恰好处于一个逻辑页面中，物理地址也是这样的。

与之相反，线性地址通常不被认为是分页的。CPU（实际是下文中的 MMU）会以一种体系结构特有的方式把进程使用的逻辑地址转换成线性地址。在 x86 平台上，这种转换是简单地把虚拟地址与另一地址，即进程的段基址相加；因为每个任务的基址都被设置为 0，所以在这种体系结构中，逻辑地址和线性地址是相同的。得到的线性地址接着被转换成物理地址并与系统的 RAM 直接作用。

内存管理单元

在逻辑地址和物理地址之间相互转换的工作是由内核和硬件内存管理单元（MMU—memory management unit）共同完成的。MMU 是被集成进现代的 CPU 里的，它们都是同一块 CPU 芯片内的一个部分，但是把 MMU 当作一个独立的部分仍然非常有益。内核告诉 MMU 如何为每个进程把某逻辑页面映射到某特定物理页面，而 MMU 在进程提出内存请求时完成实际的转换工作。

当地址转换无法完成时，比如，由于给定的逻辑地址不合法或者由于逻辑页面没有对应的物理页面的时候，MMU 就给内核发出信号。这种情况称为页面错误（page fault），本章后面会对此进行详细论述。

MMU 也负责增强内存保护，比如当一个应用程序试图在它的内存中对一个已标明是只读的页面进行写操作时，MMU 就会通知 OS。

MMU 的主要好处在于速度。缺少 MMU 时为了获得同样的效果，OS 将不得不使用软件为每个进程的每一次内存引用进行校验，这种校验同时包括数据和指令在内，而这可能还包括要用为进程创建其生存所需的虚拟机。（Java 所进行的一些工作与此类似。）这样做的结果将使系统慢得令人无法忍受。但是一个以这种内存访问合法性检查方式集成在计算机硬件里的 MMU 却根本不会使系统变慢。在 MMU 建立起一个进程以后，内核就只是偶尔参与工作，例如在发生页面错误时，而这与全部内存引用数量相比是非常少的。

除此而外，MMU 还可以协助保护内存自身。没有 MMU，内核可能不能够防止一个进程非法侵入它自己的内存空间或者是其它进程的内存空间。但是如何避免内核也会作同样的

操作呢？在 Intel’s 80486 或更新的芯片上（不是 80386），MMU 的内存保护特性也适用于内核进程。

页目录和页表

在 x86 体系结构上，把线性地址（或者逻辑地址——记住在 Linux 上，这二者具有相同的值）解析（resolving）到物理地址分为两个步骤，整个过程如图 8-2 所示。提供给进程的线性地址被分为三个部分：一个页目录索引，一个页表索引和一个偏移量。页目录（page directory）是一个指向页表的指针数组，页表（page table）是一个指向页面的指针数组，因此地址解析就是一个跟踪指针链的过程。一个页目录使你能够确定一个页表，继而得到一个页面，然后页面中的偏移量（offset）能够指出该页面里的一个地址。

为了进行更详细因而也会更准确的描述：给定页目录索引中的页目录项保存着贮存在物理内存上的一个页表地址；给定页表索引中的页表项保存着物理内存上相应物理页面的基地址；然后线性地址的偏移量加到这个物理地址上形成最终物理页面内的目的地址。

其它 CPU 使用三级转换方法，如图 8-3 所示。这在 64 位体系中尤其有用，以 Alpha 为例，其更大的 64 位的地址空间意味着类似于 x86 体系的地址转换将要求大量的页目录、大量页表、大量偏移量，或三者兼有。对于这种情况，Alpha 的设计者们向线性地址模式中引入了另一层次，即 Linux 所称的页面中间目录（page middle directory），它位于页目录和页表之间。

这个方案与以前实际是一样的，只不过多增加了一级。这种三级转换方法同样具有页目录，页目录的每一项包含一个页面中间目录的入口地址，页面中间目录的每一项包含一个页表的入口地址，而页表也同以前一样每一项包含物理内存中一个页面的地址，这个地址再加上偏移量就得到了最终的地址。

而使情况更为复杂的是，通过进一步观察可知，三部分地址模式与两级地址转换是相关联的，而四部分地址模式则与三级地址转换相关联的，这是由于我们通常所说的“级（或层次 levels）”不包括索引到页目录的第一步（我想是因为这一步没有进行转换的缘故）。

令人奇怪的是内核开发者们决定只用其中一种模式来处理问题。绝大部分的内核代码对 MMU 一视同仁，就如同 MMU 都使用三级转换方法（也就是四部分地址模式）一样。在 x86 平台上，通过将页面中间目录定义为 1，页面相关的宏可以把三级分解过程完美地转换到二级分解过程上去。这些宏认为页面中间目录和页目录是几乎可以进行相互替换的等价品，以至于内核的主要代

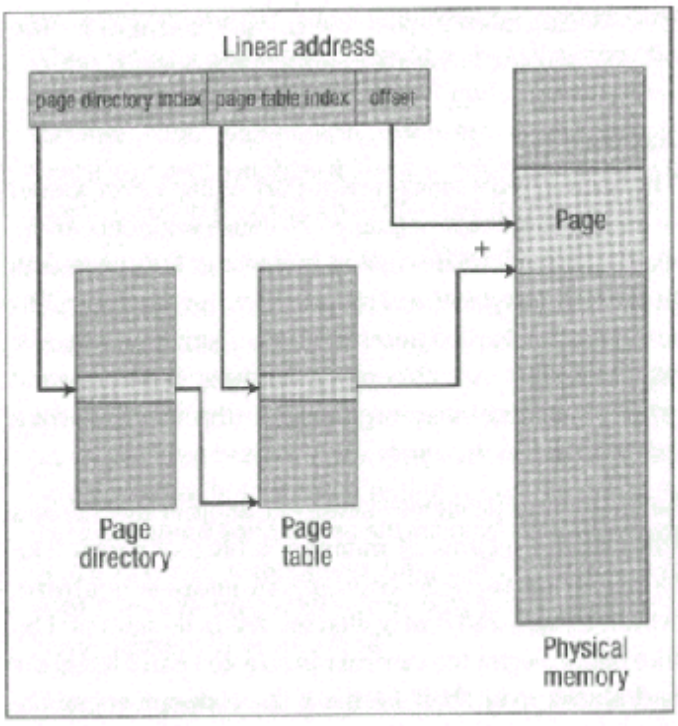

Figure 8.2 Paging on the x86.

在 x86 系统中 …索引，剩下的 12 位用作偏移量，这

用于创建和操 …ge.h（第 10786 行）和 include/asm …数和宏的时候，记住 PGD 通常代表 … PMD 通常代表“页面中间目录项 …”），同样 PTE 也通常代表“页表项 … 例外是存在的，例如下文将要提到 …页表项。非常遗憾的是，由于篇幅 …的一部分进行讨论。

页表项不仅记 …ctions），也就是一组指定该页为可 …的保护位）。随着我们对页面保护信 …存有的标志。

转换后备缓存（Translation Lookaside Buffers：TLBs)

如果简单的执行从线性地址到物理地址的转换过程，在跟踪指针链时将会需要几个内存引用。RAM 虽然不像磁盘那么慢，但是仍然比 CPU 要慢的多，这样就容易形成性能的瓶颈。为了减少这种开销，最近被执行过的地址转换结果将被存储在 MMU 的转换后备缓存（translation lookaside buffers：TLBs）内。除了偶尔会通知 CPU，由于内核的某操作致使 TLBs 无效之外，Linux 不用明确管理 TLBs。

在作用于 TLB 的函数和宏中，我们只研究一下**__flush_tlb**，在 x86 平台上，它是其它大部分函数和宏的基础。

__flush_tlb

10917：CR3（控制寄存器 3）是 x86CPU 寄存器，它保存页目录的基地址。往这个寄存器送入一个值将会使 CPU 认为 TLBs 变成无效，甚至写入与 CR3 已有值相同的值也是这样。

因此，**__flush_tlb** 仅是两条汇编程序指令：它把 CR3 的值保存在临时变量 **tmpreg** 里，然后立刻把 **tmpreg** 的值拷贝回 CR3 中，整个过程就这么简单！

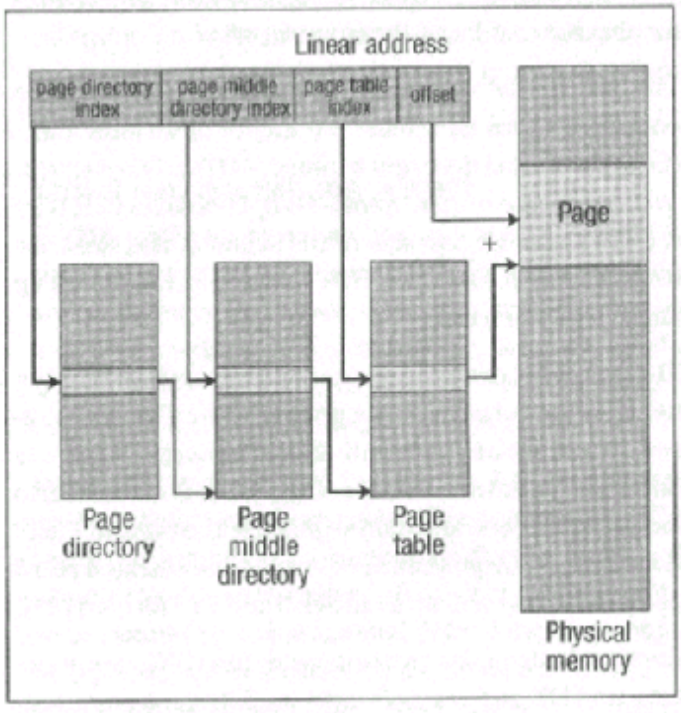

Figure 8.3 The kernel's generic view of paging.

注意 x86 系统也允许使某一个单独的 TLB 项无效，而并不一定非要使全部项，这种方法使用 **invipg** 指令——参见第 10926 行它的使用信息。

段

由于段不是在所有 CPU 中均可用，所以 Linux 内核中与体系结构无关的部分不能对段进行辨识。在不同的 CPU 体系中，段的（如果段在体系中是可用的）处理方式大相径庭，这一点是非常重要的。因此，我们在这个问题上不会花费太多时间，不过 x86 系统上内核使用段的方式还是值得大概讨论一下的。

段可以被简单的看作是定义内存区域的另一种机制，有些类似于页。这两种机制可以重叠：地址总是在页面之内，也可能处于段内。与页不同，段的大小可以变化，甚至在其生存期里能够增长和收缩。与页相同的是，段可以被保护，而且其保护可由硬件实施；当 x86 的段保护和同一地址的页保护发生冲突时，段保护优先。

X86 系统使用一些寄存器和全局描述表（GDT）和局部描述表（LDT）这两种表来对段进行跟踪。描述符（descriptor）是段的描述信息，它是用来保存段的大小和基址以及段的保护信息的 8 字节的对象。GDT 在系统中只有一个，而 Linux 可以为每个任务建立一个 LDT。

接下来我们将简单解释内核是如何使用这些表来建立段的。内核本身拥有分离的代码和数据段，它们被记述在 GDT 的第 2 和第 3 行项里。每个任务也有分离的代码和数据段。当前任务的代码段和数据段不仅在它自己的 LDT 的第 0 和第 1 行项被说明，而且还被记述在 GDT 的第 4 和第 5 行项里。

在 GDT 里，每个任务占两行项，一个用来定位它的 LDT，一个用来定位它的 TSS（前面章节曾简要提及的任务状态段）。因为 x86CPU 限制 GDT 的大小为 8192 个项，而且 Linux 为每个任务占用两行 GDT 项，因此显而易见的是我们不能拥有超过 4096 个任务，这也正是在第 7 章里提到的限制。事实上，任务的最大数目要稍小一点儿，不过仍有 4090 个，这是由于 GDT 的前 12 行项被保留用于其它目的。

富有经验的 x86 程序员可能已经注意到 Linux 所使用的 x86 分段机制是采用最低限度方式的；段的主要使用目的仅是为了避免用户代码出现在内核段中。Linux 更倾向于分页机制。从大的方面来看，对于处理器来说分页或多或少都是相同的，或者说总的事实就是这样，因此内核越是以分页方式工作，它的可移植性就越好。

最后要提及的是，如果读者对于 x86 的分段机制很感兴趣的话，不妨阅读一下 Intel 体系结构下的软件开发手册第 3 卷（Intel Architecture Software Developer’s Manual Volume 3），该书可以从 Intel 站点上免费得到（**developer.intel.com/design/pentiumii/manuals/243192.htm**）。

进程的内存组织

有三个重要的数据结构用于表示进程的内存使用：**struct vm_area_struct**（第 15113 行）、**struct vm_operations_struct**（第 15171 行），和 **struct mm_struct**（第 16270 行）。我们随后将对这三个数据结构进行逐一介绍。

Struct vm_area_struct

内核使用一个或更多的 **struct vm_area_struct** 来跟踪进程使用的内存区域，该结构体通常缩写为 VMAs。每个 VMA 代表进程地址空间的一块单独连续的区间。对于一个给定的进程，两个 VMAs 决不会重叠，一个地址最多被一个 VMA 所覆盖；进程从未访问过的的一个地址将不会在任何一个 VMA 中。

两个 VMA 之间的区别有两个特征：

- 两个 VMA 可以不连续（Two VMAs may be discontiguous）——换句话说，一个 VMA 的末尾不一定是另一个的开头。
- 两个 VMA 的保护模式可以不同（Two VMAs may have different protections）——例如，一个是可写的而另一个可能是不可写的。即使两个这样的 VMA 是连在一起的，它们也必须被分开管理，因为其不同的保护信息。

应注意的一个重点是，一个地址可以被一个 VMA 所覆盖，即使内核并没有分配一个页面来存贮这个地址。VMA 的一个主要应用就是在页面错误时决定如何作出反应。我们可以将 VMAS 看作是一个进程所占用的内存空间以及这些空间的保护模式的总体视图。内核能够反复重新计算从页表而来的 VMA 中的大部分信息，不过那样速度会相当慢。

进程的所有 VMA 是以一个排序的双向链表方式存储的，并且它们使用自己的指针来管理该列表。当一个进程有多于 **avl_min_map_count** 数目（在第 16286 行定义为 32）的 VAM 时，内核也会创建一个 AVL 数来存储它们，此时仍然是使用 VMAs 自己的指针对该树进行管理。AVL 树是一个平衡二叉树结构，因此这种方法在 VMA 数量巨大时查找效率十分高。不过，即使在 AVL 树被创建后，线性列表也会被保留以便内核即使不使用递归也能轻松的遍历一个进程的所有 VMA。

Struct_vm_area_struct 的两个最重要的元素是它的 **vm_start** 和 **vm_end** 成员（分别在第 15115 行和 15116 行），它们定义了 VMA 所覆盖的起止范围，其中 **vm_start** 是 VMA 中的最小地址，而 **vm_end** 是比 VMA 最大地址大一位的地址。在本章后面我们会反复提及这些成员。

注意，**vm_start** 和 **vm_end** 的类型是 **unsigned long**，而不是读者可能会认为的 **void***。由于这个原因，内核在所有表示地址的地方都使用 **unsigned long** 类型，而不用 **void***类型。采用这种方法的部分原因是可以避免因内核对诸如比特一级的地址进行计算操作时引起的编译警告，还可能避免由于它们而偶然引起的间接错误。在引用内核空间的一个数据结构的地址时，内核代码使用指针变量；在对用户空间的地址进行操作时，内核却频繁的使用 **unsigned long**——实际上，几乎只有本章中所涉及的代码才是这样。

这样就给用来编译内核的编译器提出了要求。使用 **unsigned long** 作为地址类型就意味着编译器必须使 **unsigned long** 的类型长度和 **void***的一样，尽管实践中对这一点的要求不是十分严格。对于 x86 寄存器上的 gcc 来说，两种类型很自然的都是 32 位长。在 64 位指针长度的体系中，比如 Alpha，gcc 的 **unsigned long** 类型长度通常也是 64 位。尽管如此，在将来的体系结构上，gcc 的一个端口可能提供与 **void***不同的 **unsigned long** 类型长度，这是需要内核的移植版本开发人员（kernel porters）注意的一点。

还要说明的是，除了 gcc 之外你不需要对编译器的性能有太多担心，因为其它大部分与 gcc 相关的特性都已经包括在代码之中了。假如读者试图用某个其它的编译器来编译内核的话，我想有关 **unsigned long** 和 **void***长度的错误将会占编译错误列表的绝大多数。

Struct vm_operations_struct

一个 VMA 可能代表一段平常的内存区间，就像是 **malloc** 函数所返回的那样。但是它也可以是对应于一个文件、共享内存、交换设备，或是其它特别的对象而建立的一块内存区域；这种对应关系是由本章后面将要涉及的称为 **mmap** 的系统调用所确定的。

我们不想牵扯太多关于 VMA 可以被映射的每一种对象的专门知识，这会使对内核代码的剖析变得凌乱不堪，因为那样就不得不反复决定是否要关闭一个文件、分离共享内存等等令人非常头疼的事情。与此不同，对象类型 **struct vm_operations_struct** 抽象了各种可能提供给被映射对象的操作，比如打开、关闭之类。一个 **struct vm_operations_struct** 结构体就

是一组函数指针，它们之中可能会是 **NULL** 用来表示一个操作对某个被映射对象是不可用的。举例来说，在一个共享内存没有被映射的情况下，把该共享内存对象的页面与磁盘进行同步是没有意义的，表示共享内存操作的 **struct vm_operations_struct** 里的 **sync** 成员就是 **NULL**。

总之，一旦 VMA 映射为一个对象，那么它的 **vm_ops** 成员就会是一个非空的指针，指向一个表示被映射对象所提供操作的 **struct vm_operations_struct** 结构体。对于 VMA 可以映射的每一种对象类型，都有一个该 VMA 可能会在某处指向的静态 **static struct vm _ operations_struct** 结构体。参见第 21809 行这样的一个例子。

Struct mm_struct

一个进程所保留的所有 VMA 都是由 **struct mm_struct** 结构体来管理的。指向这种结构类型的指针在 **struct task_struct** 中，确切的说，它就是后者的 **mm** 成员。这个成员被前一章中所讨论的 **goodness**（第 26388 行）应用，来判断是否两个任务是在同一个线程组中。两个具有相同 **mm** 成员（正如我们所见到的）的任务管理同一块全局内存区域，这也是线程的一个特点。

struct mm_struct 结构体的 **mmap** 成员（第 16271 行）就是前述的 VMA 的链接列表，而它的 **mmap_avl** 成员，如果非空，就是 VMA 的 AVL 树。读者可以浏览 **struct mm_struct** 的定义，会发现它还包括相当多的其它成员，它们中的几个会在本章中涉及到。

VMA 的操作

本小节介绍后面要用到的 **find_vma** 函数，并捎带简介它的同类函数 **find_vma_prev**。这将阐明 VMA 处理操作的一些方面，也为读者将要接触的代码做准备。

find_vma

33460：简单说来，**find_vma** 函数的工作就是找到包含某特定地址的第一个 VMA。更准确的说，它的工作是找到其 **vm_end** 比给定地址大的第一个 VMA，这个地址可能会在该 VMA 之外，因为它可以比 VMA 的 **vm_start** 要小。这个函数返回指向 VMA 的指针，如果没有满足要求的 VMA 就返回 **NULL**。

33468：首先，通过使用 **mm** 的 **mmap_cache** 成员，满足进程最近一次请求的同一 VMA 会被检查，而 **mmap_cache** 正是为此目的而设。我没有亲自测试过，不过这个函数的文档中说高速缓存的命中率可以达到 35%，考虑到高速缓存只由一个 VMA 组成，那么这个数字就相当好了。当然，著名的、被称之为“引用的局部性（locality of reference）”的特性一直在其中提供了很大帮助，这也是软件访问数据（和指令）时的一条原则，即访问最近使用过的数据（和指令）。由于 VMA 包含一块连续的地址区间，引用的局部性就使得所需的地址都在同一个 VMA 中变为可能，而这样的 VMA 就会满足前面的要求。

在修改 VMA 列表的其它几个地方，这个高速缓存的值被设为空，表明对 VMA 列表所做的修改可能会使它失效。至少在一个这种情况中，第 33953 行，使该高速缓存为空不总是必要的；这段代码如果能够再聪明一些的话，就可能从本质上改善高速缓存的命中率。

33471：高速缓存没有命中。假如没有 AVL 树，**find_vma** 只是搜索列表上的所有 VMA，然后返回第一个符合条件的 VMA。回想一下 VMA 的列表是保持顺序的，所以符合条件的 VMA 也就是所有符合条件的 VMA 中地址最小的一个。假如搜索到列表的末

尾都没有一个匹配，**vma** 就被置为 **NULL**，并被返回。

33476：若有大量 VMA，沿树遍历就比沿链表遍历要快；由于 AVL 树是平衡的，这就是一种对数时间操作而不是线性时间操作。

树的迭代遍历并不是十分少见的现象，不过一些特征也并不非常明显。首先注意第 33484 行的赋值；这个操作一直跟踪当前找到的最好节点，当不能找到更好的时，它就会被返回。接着的下一行中的 **if** 语句是一个最优测试，检测 **addr** 是否处于 VMA 中（我们已知的一点是 **addr** 小于 VMA 的 **vm_end**）。因为 VMA 绝对不会彼此覆盖，没有其它 VMA 将是一个较贴近的匹配结果，所以树的遍历可以早些结束。

33492：如果在树的遍历或列表搜索过程中找到一个 VMA，找到的值就被保存在高速缓存里以便下一次查找。

33496：在任何情况下，**vma** 都被返回；它的值或者是 **NULL**，或者是满足查找条件的第一个 VMA。

Find_vma_prev

如前所述，这个函数（从第 33501 行开始）和 **find_vma** 函数是一样的，不过它还会额外的返回一个指向前一个限定的 **addr**（如果有）的 VMA 的指针。这个函数不仅是因为它本身的缘故而令人感兴趣，更主要是由于它的出现会告诉我们一些关于内核程序设计，特别是关于 Linux 内核程序设计的信息。

应用程序员很可能已经在更加通用的 **find_vma_prev** 函数之上写出了 **find_vma** 函数，这只需简单的把指向 VMA 的指针去掉即可，代码如下：

p504.1

应用程序员这样做的原因是具有代表性的应用程序并不太拘于速度因素。这并非纯粹是在为铺张浪费找借口，而是由于 CPU 速度的不断增加使得应用能够更关注于其它方面，我们现在可以出于可维护性的充分理由而提供一个可以到处使用的额外函数调用。

与之相反，一个内核程序员可能不会随便增加多余的函数调用；试着减少几个 CPU 时钟周期会被认为比负责维护某个近乎是副本的函数要更胜一筹。即使没有其它原因，我们也可以说内核开发者所持有的这种态度就是为了让应用程序员能相对自由一些。

为什么这种重复对于源代码相对封闭的操作系统，Linux 而言不那么重要，这里是否有更深层次原因呢？尽管 Linux 内核必须限制它占用的 CPU 时间，Linux 内核的开发工作却不受程序员时间的限制。（明确的说，我必须要指出 Linux 的开发者不必把他们的时间浪费在会议上的，他们也不必被人工制订的时间表所拘束。）正是由于这众多的队伍，众人的智慧，才改变了软件开发的规则。

Linux 内核的源代码对任何人都是公开的，Linus 本人曾说过的一句名言是“…只要眼睛够多，所有的臭虫（程序错误）都是浅薄的[1]”。就算函数 find_vma 和 find_vma_prev 的执行会产生重大差异，在你能想到“重编译”之前，不知什么地方的某个 Linux 内核开发者就已经迅速发现并修复了这个问题。实际上，Linux 内核开发者比它的商业对手动作快得多，所得到的代码运行更快而且错误更少，尽管有时偶然出现的结构会被认为在任何其它环境中都不可维护。

当然，如果没有人负责对这些函数的改进进行维护的话，我认为这也是非常愚蠢的。内核的下一个发布版本就把它们合并了。但是我仍然对此持怀疑态度，而且即使我在这个具体问题上所持的态度并不正确，我依然在总体上保持原有态度。 不同的事还会继续不同，而不同正是 Linux 之所以为 Linux 的一方面。

[1] 原文为：”…given enough eyeballs, all bugs are shallow.”

分页

本章前面对分页已作了概要描述，现在我们进一步来研究 Linux 是如何处理分页的。

页面保护详述

正如早先提及的，页表项不仅保存了一个页面的基地址，还有其它一些标志信息，这些标志指出了该页面上所能进行的操作。现在是仔细研究一下这些标志的时候了。

如果页表项只保存一个页面的基地址，并且页面是页对准的（page-aligned），这个地址的低 12 位（x86 系统），即偏移量部分通常将总是为 0。取代这些位置 0 的作法是把它们编码作为与页面有关的标志，在获取地址时只需简单的把它们屏蔽掉就行了。以下就是这 12 位中的标志位：

- **_PAGE_PRESENT** 位（第 11092 行），若置位，当前页面物理存在于 RAM 中。
- **_PAGE_RW** 位（第 11093 行），置为 0 表示该页面是只读的，置为 1 表示可读可写。因此，没有只写的页面。
- **_PAGE_USER** 位（第 11094 行），置位表示某页面是用户空间页面，清空表示为内核空间页面。
- **_PAGE_WT** 位（第 11095 行），置为 1 表示页面高速缓存管理策略是透写，置为 0 表示管理策略是回写。透写（writethrough）会立刻把写入高速缓存的数据复制（拷贝）到主存储器内，即使保存在高速缓存的数据仍是读访问。与之相反，回写（writeback）具有更高的效率，写入高速缓存的数据仅当其必须为其它数据腾出空间，而必须移出时才被复制到主存储器内。（这是由硬件，而不是 Linux 完成的。）尽管直到本书写作之时，这个标志位在内核中的使用还并不非常普遍，不过这种情形有望很快改变。有时候，Intel 公司的处理器资料中把 WT 位更多的称为 PWT。
- **_PAGE_PCD** 位（第 11096 行），关闭页面高速缓存；本书中的代码不总是使用这个标志位。（缩写“CD”表示“caching disabled”。）如果我们恰好知道一个不经常使用的页面，那么就不必为它设置高速缓存，这可能会更有效率。这个标志位好像对于映射内存的 I/O 设备来说更有用处，尽管我们想确保对表示设备的内存进行的写操作不被高速缓存缓冲，但是取而代之的作法是立刻把数据直接拷贝到设备之中。
- **_PAGE_ACCESSED** 位（第 11097 行），若置位表示该页面最近曾被访问过。Linux 可以设置或清除这个标志，不过通常这是由硬件完成的。因为清除了该标志的页面已很久未被使用过，所以它们会在交换时被优先调出主存。
- **_PAGE_DIRTY** 位（第 11098 行），若置位，表明该页面的内容自从上次该位被清除后已发生改变。这就意味着它是一个内容没有保存的页面，就不能简单的为交换而被删除。当一个页面第一次写入内存时，该标志位由 MMU 或 Linux 设置；当这个页面调出内存时，Linux 要读取它的值。
- **_PAGE_PROTNONE** 位（第 11103 行），是一个以前的页表项没有使用过的标志位，用来跟踪当前页面。

_PAGE_4M 位和**_PAGE_GLOBAL** 位在同一个**#define** 定义块中出现，但是由于它们不像其它标志位那样用于页面级的保护，所以我们在此不予讨论。

随后的文件中，上述这些标志位被组合在一个高级宏内。

写拷贝（copy-on-write）

提高效率的一条路就是偷懒——只做最少量的必要工作，而且只在不得不做的时候才完成。现实生活中这可能是个坏习惯，至少它会导致拖拖拉拉。而在计算机的世界里，它可能更是一种优点。写拷贝（Copy-On-Write）就是 Linux 内核一种通过懒惰来获得效率的方法。其基本思想是把一个页面标记为只读，却把它所含的 VMA 标识为可写。任何对页面的写操作都会与页级保护相冲突，然后触发一个页面错误。页面错误处理程序会注意到这是由页级保护和 VMA 的保护不一致而导致的，然后它就会创建一个该页的可写拷贝作为代替。

写拷贝十分有用。进程经常 **fork** 并立刻 **exec**，这样为 **fork** 而复制的进程页面会造成浪费，因为 **exec** 之后它们会不得不被抛弃。正如读者所见，进程分配大量内存时也使用同样的机制。所有被分配的页面都与一个单独的空白页面相映射，这就是写拷贝的原意。向某页面的第一次写操作会触发页面错误，然后空白页面执行复制。用这种办法，只有页面分配不能再延期时，它才会被分配。

页面错误

到现在为止，本章已几次提到一个页面可以不在 RAM 里的可能性——毕竟，如果页面总是在内存里，虚拟内存就没什么必要了。但是我们还没有详细介绍过当某页面不在 RAM 中会怎样。当处理器试图访问一个当前不在 RAM 中的页面时，MMU 就会产生一次页面错误，而内核会尽力解决它。在进程违反页级保护时，页面错误也会产生，例如进程试图向只读内存区域写入。

因为任何无效内存访问都会导致页面错误，同样的机制支持请求分页。请求分页（Demand paging）的意思是只有在页面被引用的时候才从磁盘上读取它们——即按需分配——这是另一种通过懒惰来获得效率的方法。

特别地，请求分页用于实现被请求页面的可执行化。为了达到这个目的，应用程序第一次被装载时，只有一小部分可执行映象（image）被读入物理内存，然后内核就依靠页面错误来调入需要的（比如，进程首次跳转到一个子例程时）可执行页面。除了一些意外的情况，这样做总是要比把所有部分一次读入要快，这是因为磁盘较慢，而且并不是所有的程序都会用到的。事实上，因为一个大程序运行一次时，大部分功能特性都不会再用到，所以通常根本不需要全部都读入（这一点对大多数中小规模的程序也是成立的）。这对于按需分页（demand-paged）的可执行程序稍有不同——如果你对这种情况进行考虑的话，你就可以知道按需分页还需要二进制处理程序的支持，而且它是一个具有决定意义的部分。

Do_page_fault

6980： **do_page_fault** 是内核函数，产生页面错误时（在第 363 行）被调用。当页面错误产生时，CPU 调整进程的寄存器，当解决页面错误时，进程再从引起错误的指令处开始执行。通过这种方法，在内核使得冲突地内存访问操作完成后，会自动重试该操作。相反，如果页面错误仍然无法解决，内核就通知引起冲突的进程。当页面错误是由内核本身导致的时候，所采用的措施是近似的，但并不完全相同。

6992： 控制寄存器 2（CR2）是 Intel CPU 的寄存器，保存引起页面错误的线性地址。该寄存器内的地址会被直接读入局部变量 **address**。

7004： 函数 **find_vma**（第 33460 行）返回地址范围末尾在 **address** 之后的第一个 VMA。大家知道，这并不能够保证该地址位于 VMA 的范围内，而仅保证该地址比 VMA

的结束地址小，这样它就可能比 VMA 的初始地址还要小。因此这一点要被检查。假如通过判断，则 **address** 在 VMA 之内，控制就会向前跳转到标号 **good_area** 处（第 7023 行）；我们随后就会对这一点进行讨论。

7005： 如果 **find_vma** 返回空值 NULL，那么 **address** 就位于进程的所有 VMA 之后——换句话说就是超出了由进程引用的所有内存。

7009： **vma** 的开头和结尾都确实超过了 **address**；因此 **address** 在 VMA 低端地址以下。但是这并不会失去什么。如果 VMA 是向下扩展的类型——也可以说它是堆栈——这个堆栈可以简单的向下扩展来适应那个地址。

7011： CPU 提供的 **error_code** 的测试位 2。与监控（内核）模式相比，更多是在用户模式发生页面错误时设置此位。如果是在用户模式下，**do_page_fault** 函数会保证给定的地址在为进程建立的堆栈区域内，正如 ESP 寄存器所定义的那样。（例如，在代码溢出了被分配的堆栈矩阵时，就会产生这种情况。）如果是在监控（内核）模式下，就会跳过后一种判断，而简单的假定内核运行正常。

7019： 如果可能，使用 **expand_stack**（行 15480）将扩展到包含新的地址。如果成功，VMA 的 **vm_start** 成员将调整到包含 **address**。

7023： 到达 **good_area** 标记时，就意味着 VMA 包含 **address**，或者说要么它已经包括了 **address**，要么就是堆栈扩展后包括了该地址。

不管那一种方法，包括错误产生原因信息的 **error_code** 最低两位现在都可以被测试了。第 0 位是存在/保护位：0 表示该页不存在；1 表示该页存在，但试图的访问操作与页级保护位冲突。第 1 位是读/写位：0 表示读，1 表示写。

7025： **switch** 条件判断语句对于上述两个测试位所组合出的四种可能情况作出相应处理：

- case 2 或 3——检查包括的 VMA 是否可写。若可写，就是向一个写拷贝页面执行一次写操作；变量 **write** 被增加（设置到 1）以便接下来对 **hand_mm_fault** 的调用能够完成写拷贝过程。
- case 1——这意味着页面错误是由试图从一个存在但不可读的页面中读数据而导致的；这个尝试会被拒绝。
- case 0——表示页面错误是由试图从一个不存在的页面中读数据而导致的。如果涉及的 VMA 保护指出该区间既不可读也不可写，读页面只不过是浪费时间——如果再次尝试，将引起另一个页面错误，这样 **do_page_fault** 函数会以 case 1 的结果告终，即拒绝尝试。否则 **do_page_fault** 函数继续执行并从磁盘上读入页面。

7047： 请求 **handle_mm_fault** 函数（下面讨论）使该页面变为当前页面。如果失败，则发出一个 **SIGBUS** 错误。

7062： 大多数内核函数的清除代码都不太显著。**do_page_fault** 函数是一个例外；我们会比较详尽的研究它的清除代码。下列任何情况发生都会跳到 **bad_area** 标记处：

- 被引用的地址超过了为进程分配的（或保留的）所有内存。
- 被引用的地址位于所有 VMA 之外，而且可能由于比该地址小的 VMA 不是堆栈而无法扩展到这个地址。
- 违反了页面的读/写保护。

7066： 如果用户代码引起以上任何错误，那将发送致命的 **SIGSEGV** 信号——一个分段违例。（注意术语“分段”在这里是历史上的说法而不是字面所表达的意思——对 CPU 来说，从技术角度看它是分页违例，不一定是分段违例。）这个信号通常会像第 6 章中讨论的那样杀死一个进程。

7075： Intel Pentium CPU（以及它的一些兼容产品）具有一个所谓的 f00f 缺陷，它允许任

何进程用非法的 0xf00fc7C8 指令来冻结 CPU。Intel 所提议的弥补工作就是在这里实现的：中断描述表（见第 6 章）的一部分以前是被标识为只读的，因为这样会使非法指令执行时用产生页面错误代替冻结 CPU。在这里，**do_page_fault** 函数检查导致页面错误的地址是否位于 IDT 中由非法指令执行而产生的位置上。如果是这样的，处理器会试着执行“Invalid Opcode”服务中断——CPU 的缺陷会使得正确完成这一步失败，而代码却会通过直接调用 **do_invalid_op** 函数而产生正确的结果。否则，CPU 决不会对 IDT 进行写操作(即使没有标注为只读时也是如此),所以即使第 7080 行的检测失败，非法指令也是根本不会被执行的。

7086： 下列情况发生时，标记 **no_context** 会被执行：

- 在内核（不是用户）模式里到达 **bad_area**，而且 CPU 不执行触发 f00f 缺陷的非法指令。
- 在一个中断中或没有用户环境（用户任务没有处于正在执行状态）时发生的页面错误。
- **Handle_mm_fault** 函数错误并且系统处于内核模式中（我还从未遇到过这种情况）。

这里的任何一种情形都是内核错误（经常由驱动程序所导致），它不是因为任何用户代码而造成的页面错误。如果内核（或驱动程序）事先为这种可能准备了错误处理代码，那么这些错误处理代码一定位于本书讨论范围之外，并在错误发生时可以通过某种特殊技术跳转过去。

7097： 否则，内核试图访问一个坏页面，**do_page_fault** 函数将不知如何处理它。这可能也够被考虑到。内核启动代码检查是否 MMU 写保护工作正常；如果正常，那就不是一个真正的错误，**do_page_fault** 函数就可以简单的返回了。

7109： 内核访问了一个坏页面，并且 **do_page_fault** 函数无法修复这个错误。**do_page_fault** 函数会在第 7129 行显示出一些描述错误的信息，然后中止内核本身。这样整个系统就会被停止，很明显没有任何操作会被执行。不过，如果系统运行到了这一步，内核也别无选择了。

7134： 最后一个标记是 **do_sigbus**，只有当 **handle_mm_fault** 函数无法处理错误时才会执行到这里。这种情况相对简单；大体上是给违例的进程发送一个 **SIGBUS** 错误信号，如果这是在内核模式里发生的就再跳回到 **no_context** 标记处。

Handle_mm_fault

32725：调用者已经检测到了需要一个可用的页面。该页面正是包含 **address** 的页面，这个地址应归入 **vma** 中。**Handle_mm_fault** 函数本身相当简洁，但是它建立在其它几个处理冗长细节问题的函数和宏之上。我们介绍完此函数后将逐一研究那些底层函数。

32732：查找关联的页目录和页面中间目录入口项（如前所述，这两者在 x86 平台上实际是一样的）。

32735：从页面中间目录项得到或定位（如果可能的话）页表。

32737：调用 **handle_pte_fault** 函数把页面读入页表项（page table entry）；如果成功，就调用 **update_mmu _cache** 函数更新 MMU 的高速缓存。控制流程到此为止，一切顺利，**handle_mm_fault** 函数就可以返回一个非零值（1）表示成功了。如果此过程任何一步出错，控制就转向第 32744 行，函数返回 0 值表示失败。

Pgd_offset

11284：这个宏将 **address** 除以 2^{PGDIR_SHIFT}（第 11052 行**#define**d to 32），并对结果向下舍入，然后把最终结果（移位之前的高端 10 位）作为提供的 **struct mm_struct** 的 **pgd** 数组

的一个索引。因此，它的值就是页目录项，相应的页表 **address** 地址就位于该项中。这等价于代码

&((mm)->pgd[(address)>>PGDIR_SHIFT]);

而且可能会更高效。

Pmd_alloc

11454：因为 x86 平台上没有定义页面中间目录，这样就极其简单：它只需返回给定的 **pgd** 指针，并映射为一种不同类型。在其它平台上，该函数与 **pte_alloc** 类似，还要实现更多的工作。

Pte_alloc

11422：**Pte_alloc** 函数有两个参数：一个是指针，指向目标地址所位于的页面中间目录项，另一个是地址本身。如果我们暂时跳过一部分内容，那么对该函数经过变形的逻辑的理解就会更容易，所以让我们看一下随后的若干行代码。

11425：用一种几乎无法理解的方式把 **address** 转换成 PMD 内的一个偏移量。

这一行需要详细进行解释。首先，回忆一下 PMD 中的每项都是一个指针，在 x86 平台上它的长度是 4 个字节（这里的代码是与体系结构相关的，所以我们可以作出这样的假定）。用 C 语言来定义就是，

&pmd[middle_10_bits(address)]

（为清晰起见，我在这里引入了假定的 **pmd** 数组和 **middle_10_bits** 函数）该代码等价于

pmd+middle_10_bits(address)

这又与如下代码指向的地址相同

((char*)pmd)+middle_10_bits(address)*sizeof(pte_t*)

其技巧是在最后的公式中——或者更准确的说是+号后边的部分——最接近于第 11425 行所要计算的实际值。

为了使这一点更为明确，首先可知

4*(PTRS_PER_PTE-1)

就是 4092（第 11059 行 **PTRS_PER_PTE** 被定义为 1024）。用二进制表示，4092 只用占最低 12 位，甚至最后 2 位也用不上。它和只占最低 10 位的 1023 左移 2 位后的值相同。这样就有

(address>>(PAGE_SHIFT-2))

把 address 右移 10 位（第 10790 行 **PAGE_SHIFT** 被定义为 12）。这两个表达式结果再逐位进行与（AND）操作。最终的结果类似于：

((address>>PAGE_SHIFT)&(PTRS_PER_PTE-1))<<2

尽管这仍很复杂，不过它更简单明了：它把 **address** 右移 12 位（为了去掉页面偏移量部分），屏蔽掉除最低 10 位的其它位（去掉页目录索引部分，只保留最低 10 位的页面中间目录索引），接着把结果左移 2 位（相当于乘以 4，即指针长度的字节数 **sizeof(pte_t*)**）。更直接的方法可能会稍慢一些，但在内核里，我们终归是要尽力节省时间的。（虽然更直接的方法看来并非明显偏慢：同样版本内核进行两次移位、两次减法，以及按位与的操作，和进行两次移位、两次按位与的操作，就我的测试看来实际上是一样快。）

不管采用那一种方法，经过计算之后，把 **address** 和 PMD 的基地址相加（在第 11432 行和别的地方执行），就得到了指向与 **address** 初值关联的 PTE 的项指针。

11428：如果 PMD 项不指向任一个页表，函数向前跳到 **getnew** 处分配一个页表。

11435：通过调用 **get_pte_fast**（第 11357 行）尝试从 **pte_quicklist** 中申请一个页表。这个页表是页表的一个高速缓存，其思想是分配页表（它们本身就是独立的页面）慢，而从一系列近期释放的页表中指定一个却会稍快一些。所以，代码经常用 **free_pte_fast**（第 11369 行）来释放页表，这会把它们放在 **pte_quicklist** 里而不是确实把它们消除掉。

11439：**pte_quicklist** 能够提供一个页表页面。页表可以被送入页面中间目录，并且函数返回页表中这个页面的偏移值。

11438：**pte_quicklist** 缓存里没有剩下页面，因此 **pte_alloc** 需要调用 **get_pte_slow** 函数（第 7216 行）来分配一个缓慢页面。该函数用**__get_free_page** 来分配页面，执行过程和一个页面被找到时相似。

11430：如果 PMD 项不是 0，但是是无效的，**pte_alloc** 显示一个警告（通过调用第 7187 行的 **bad_pte**）并放弃尝试。

11432：所期待的正常情况：**pte_alloc** 函数返回一个指向包括 **address** 地址的 PTE 的指针。

Handle_pte_fault

32690：**Handle_pte_fault** 函数试图取回或者创建一个缺少的 PTE。

32702：给定的项与物理内存中的任何一个页面都无关联（32700 行），而且确实没有被设置（32701 行）。这样，**do_no_page**（32633 行）将被调用以创建一个新的页面映射。

32704：页面在内存中不存在，但是它有一个映射，所以它一定在交换空间里。函数 **do_swap_page**（32569 行）将被调用来把该页面读回内存。

32708：页面在内存里，所以情况可能是内核正在处理一个页面保护冲突。**Handle_pte_fault** 首先要用 **pte_mkyoung**（11252 行）来把该页面标识为已被访问。

32713：如果是一个写访问操作而页面又不是可写的，**Handle_pte_fault** 就调用 **do_wp_page** 函数（32401 行）。这个函数完成真正的写拷贝功能，因此我们要简单介绍一下。

32715：这是一次对可写页面的写访问。**Handle_pte_faul**t 设置该页面的“dirty”位，表示在它被丢弃之前必须被复制到交换空间。

32720：所需的页面现在可被调用者使用，所以 **Handle_pte_fault** 函数返回非零值（确定为 1）以示成功。

Update_mmu_cache

11506：在 x86 平台上，**update_mmu_cache** 函数是一个空操作。它是一种所谓的“挂钩（hook）”函数——这种函数要在内核的平台无关部分中适当地点处保证被调用，以便不同的移植版本都能够在必要的情况下对它进行定义。

Do_wp_page

32401：如前所述，真正的写拷贝操作是在这里实现的，所以我们有必要介绍一下。**tsk** 试图写入 **address**，这个地址在给定的 **vma** 里并由所提供的 **page_table** 来控制。

32410：调用**__get_free_page**（15364 行，简单的转向第 34696 行调用**__get_free_pages** 函数）为进程提供一个新页面，此页面是写保护页面的一个新拷贝。注意这里可以允许一个任务转换。有趣的是，这里的代码不检查**__get_free_page** 分配新页面时是否成功——它实际上可能不需要新的页面，因此到必要时才会去进行检查。

32422：增加“次要（minor）”页面错误，这些错误无需访问磁盘就可被满足。

32438：只有两个页面用户存在，其中一个是交换高速缓存（swap cache），它是已被交换出但还未被回收的页面的临时缓冲池。该页面被移出交换高速缓存后（利用 37686 行的函数 **delete_from_swap_cache**），现在它就只有一个用户了。

32445：要么从交换高速缓存里回收该页面，要么它只有一个开始用户。这个页面会被标识为可写和“脏”dirty（因为它已被写过）。

32448：如果已分配了一个新页面，它就没有用了：由于该页面只有一个用户，所以没有必要进行复制。**do_wp_page** 函数释放这个新页面，并返回非零值表示成功。

32454：页面拥有不止一个用户，不能简单的从交换高速缓存里被收回。因此 **do_wp_page** 函数将需要复制一个新页面。如果先前的页面分配失败，现在就是该结果产生作用的时候了，**do_wp_page** 函数将不得不返回错误。

32459：利用 **copy_cow_page**（31814 行）复制页面内容。这通常是调用 **copy_page** 宏（32814 行），它是一个 **memcpy**。

32460：利用 **flush_page_to_ram**（10900 行）使 RAM 新旧页面拷贝同步。像 **update_mmu_cache** 函数一样，在 x86 平台上这是一个空操作。

32463：像以前一样，使得页面可写和“脏”，同时保留从封装的 VMA 而来的其它页面保护（比如可执行）。

32466：对函数 **free_page**（在 15386 行，它只是调用 34633 行的 **free_pages** 函数）的调用而不会真正释放旧的页面，因为该页面拥有多个用户——它只会减少旧页面被引用的次数。由于满足了调用者的请求，**do_wp_page** 就返回非零值表示成功。

页面调出

现在读者已经对交换页面调入有所了解，接下来看一看另一方面，交换页面的调出。

Try_to_swap_out

38863：**try_to_swap_out** 函数是最低一级交换调出函数，它由内核任务 **kswapd**（见 39272 行 **kswapd** 函数）周期性地调用（通过一系列其它函数调用）。这个函数用来写一个页面，该页面是由位于给定任务特定 VMA 中的一个单独页表项来控制的。

38873：如果内存中缺少该页面，它就不能从内存写回到磁盘，这样 **try_to_swap_out** 函数就返回失败。如果给定的地址明显是不合法的（**max_mapnr** 是当前系统中物理内存的页面数目；参见 7546 行），它也会丢弃尝试操作。

38880：如果页面被保留、锁住，或者被一个外设用于直接内存访问时，它就不能被调出。

38885：如果页面最近被访问过，把它调出可能是不明智的，因为引用的局部性可能会使该页面不久将再被引用。把该页面标识成“旧的”，这样将来的再一次尝试就可能把它调出内存——这可能很快就会发生，如果内核不顾一切要这么做的话。但事实是，

页面还没有被调出。

这一行之后的代码注释本身就含有大量信息，所以我们将跳过几段代码而不失完整性。

38965：减少任务的驻留段长度（注意 **vma->vm_mm** 是指向含有 **vma** 的 **struct mm_struct** 的指针）。驻留段长度是物理内存中的任务所占页面数目，而且很明显，这些页面中的一个现在已经不存在了。

38968：因为页面无效，所以 **try_to_swap_out** 函数必须通知所有 TLB 以无效化它们对该页面的引用。TLB 不应该再把地址解析到一个已经不存在了的页面。**try_to_swap_out** 函数接着把这个页面放入交换缓存。

38977：最后，**try_to_swap_out** 函数通过使用 **rw_swap_cache**（35186 行）把旧的页面写回磁盘，写操作是异步的，以便等待磁盘处理时系统也可以作其它工作。

38979：用**__free_page**（34621 行）来释放页面，并返回非零值表示成功。

交换设备

Linux 拥有一个按优先级排序的合法交换设备列表（以及文件，不过为了简单起见，这一部分通常用“设备”来代替这两者）。当需要分配一个交换页面时，Linux 会在仍然拥有空间的优先级最高的交换设备上来分配它。

Linux 也会在所有优先级相同的未满交换设备之间轮转使用，采用的是循环方式，通过这种在多个磁盘上分布分页请求的方法可以提高交换的性能。在等待第一个请求被满足时，另一个请求就可以分派到下一个磁盘上。最快的设置是把交换分区分布在几个相似的磁盘上，并给它们同样的优先级设置；而较慢的磁盘则有稍低一些的优先级。

不过循环也可能造成交换速度的降低。如果同一磁盘上的多个交换设备有同样的优先级，那么磁盘的读/写头将不得不在磁盘上来回的反复访问它们；在这种情况下，臭名卓著的 1000 倍的速度差异就不容忽视了。幸运的是，系统管理员会合理安排优先级以避免这种情况。Linux 继承了 Unix 的传统特性，既能让你陷入绝境，也能使你达成非常良好的目标。最简单的方案是给每个交换设备分配不同的优先级；这会有助于避免最坏的情况，但可能也不会最好。尽管如此，由于该方法简单且不会最坏，如果你不指定优先级设置，它将是缺省设置。

交换设备用 **struct swap_info_struct**（17554 行）结构体类型来表示。在 37834 行定义了这些结构体的一个数组 **swap_info**。好几个文件里的函数都操作和使用 **swap_info** 数组来进行交换管理；很快我们就会对它们进行分析。先来分析一下 **struct swap_info_struct** 的成员，这会使我们能够更清楚的了解这些函数。

- **swap_device**——发生交换的设备号；如果 **struct** 代表一个文件而不是分区，值是 0。
- **swap_file**——**struct** 代表的交换文件或分区
- **swap_map**——对交换空间里每个交换页面的用户数进行计数的数组；为 0 则表示页面空闲。
- **swap_lockmap**——用来跟踪基于磁盘的页面当前是否正被读出或写入磁盘，数组里的每一位对应一个页面。在 I/O 过程中页面将被锁定以防止内核同时对同一页面执行两次 I/O 操作，或者其它类似的愚蠢操作——需要记住的是，一旦有可能，其它进程就会与 I/O 操作相重叠，所以发生这种情况并非难事。
- **lowest_bit** 和 **highest_bit**——跟踪交换设备里第一个和最后一个可用的页面的位置。这可以有助于加快寻找空闲页面的循环。设备的第一个页面是一个不允许用于交换的头部，因此 **lowest_bit** 不会是 0。

- **cluster_next** 和 **cluster_nr**——用来对磁盘上的交换页面进行分组以获得更高的效率。
- **prio**——交换设备的优先级。
- **pages**——设备上可用的页面数目。
- **max**——内核在此设备中所允许的最大页面数目。
- **next**——把 **swap_info** 数组中的所有 **struct** 形成一个单独的链接列表（并保持优先级顺序）。这样，数组就被逻辑排序，而不是物理排序了。**next** 的值就是列表中逻辑指向下一个元素的索引，如果到达列表末尾它就是-1。

swap_list 在 37832 行定义，包括列表头（即 **head** 成员——参见 17627 行 **struct swap_list_t** 的定义）的索引；如果列表为空则此索引为-1。它还包括名字很令人迷惑的 **next** 成员，这个成员能够跟踪我们将要在其上尝试页面分配的下一个交换设备。因此 **next** 是一个迭代指针。如果列表为空或者当前没有交换，它的值就是-1。

Get_swap_page

37879：**get_swap_page** 函数从最高优先级的拥有空间的可用交换设备里获得一个页面；如果找到一个，它就返回一个非零代码描述该项，如果系统没有交换就返回 0。

37885：从上一次停止的地方继续进行迭代。如果列表是空的或没有剩余交换设备，函数即刻返回。

37891：否则的话，有理由确信存在交换空间，**get_swap_page** 函数恰恰需要找到它。这个循环过程一直迭代，直到函数找到一项（很可能的情形）或者扫描了每一个交换设备但没有一个还有剩余空间（不太可能的情形）为止。

37894：利用 **scan_swap_map**（37838 行）扫描当前交换设备的 **swap_map** 以寻找一个空闲单元，如果找到了一项，**lowest_bit** 和 **highest_bit** 成员也会被更新。要返回的 **offset** 是 0 或者是该项。

37897：当前的交换设备能够分配一个页面。**get_swap_page** 函数现在把 **swap_map** 的迭代游标向前推进以便请求能被正确的分布在交换设备上。

如果已经到达交换设备列表的末尾或是下一个交换设备的优先级低于当前设备，迭代过程就会从列表的头部重新开始。这产生两个重要作用：

- 如果较高优先级设备的交换空间又变得可用，**get_swap_page** 就会在下一次迭代时从那个设备开始分配交换。如果孤立的观察这些代码，读者会认为当高优先级设备可用的时候，这个函数仍可以从低优先级设备分配少数页面。然而事实并非如此，在我们对交换页面是如何被释放进行介绍的时候读者就会看到这一点。
- 如果优先级高的交换设备不可用，那么在下一次内核分配一个交换项时，**get_swap_page** 函数将沿列表进行迭代直到它找到当前优先级的第一个设备为止，并试着从那个设备分配交换。因此，在内核转向优先级较低的设备之前，内核会继续考虑优先级较高的设备直至它们全部耗尽。这就是先前讨论过的循环执行过程。

37910：当前设备没有可用的交换空间，或者当前设备是不可写的（这与我们所说的是同样的）。跳到下一个设备，这样如果它已经到达末尾但还未曾循环一整圈时，它就会再从头开始循环。

37916：如果 **get_swap_page** 函数到达列表的末尾而且已经循环了一遍，它就已经考虑了所有交换设备但是没有一个拥有空余的空间。因此，结论是再也没有可用的交换空间了，函数返回 0。

Swap_free

37923：**swap_free** 函数是与 **get_swap_page** 函数相对的，它释放一个单独的交换项。

37939：通过许多简单而又周密的测试后，**swap_free** 函数检查是否正在释放交换页面的设备具有比随后将被考虑的设备更高的优先级。如果是，它就把此作为一个线索以将 **swap_list** 的迭代器重新设置在列表头部。这样对 **get_swap_page** 函数的下一次调用就会从列表头部开始并能够检测到新近被释放的高优先级空间。

37944：假如最新被释放的页面处于 **lowest_bit** 和 **highest_bit** 成员所定义的范围之外，就要相应的对它们进行调整。你可以看到如果 **swap_free** 函数在一个以前已经耗尽了的设备中释放页面，这通常会引起对 **lowest_bit** 或者 **highest_bit** 的调整，但并非都要调整。这会使该区间比所需要的大，交换页面分配也会因此比所需要的要慢。不过这种情况很少发生。无论如何，交换范围都将调整自己以使更多的交换页面能够被分配和释放。

37950：对 **swap_map** 每一元素的使用计数只维护到一个最大值，即 **SWAP_MAP_MAX**（17551 行定义为 32767）。达到这个最大值之后，内核将无法知道真正的计数值有多大；由此它也无法安全的减少该值。否则的话，**swap_free** 函数将减少使用计数并增加空闲页面的总数。

Sys_swapoff

38161：**sys_swapoff** 函数在可能情况下从交换设备列表中移去被指明的交换设备。

38178：搜索 **swap_info_structs** 的列表以查找匹配的项，设置 **p** 指向这个数据项、**type** 指向该数据项的索引，以及 **prev** 指向前一项的索引。如果第一个元素被删除，**prev** 将是 -1。

38195：如果 **sys_swapoff** 函数搜索了整个列表但没有找到匹配项，那显然是给定了一个无效的名字。函数返回错误。

38198：如果 **prev** 是负值，**sys_swapoff** 函数将删除列表的第一个元素；它相应的适当更新 **swap_list.head**。可以证明，这等价于

```
swap_list.head=
  swap_info[swap_list.head].next
```

不过速度更快，因为其中所牵扯的间接转换更少。

38203：如果正被移去的设备是内核进行交换尝试的下一个设备，迭代游标会被重新设置在列表头部。这样下一次分配可能要稍慢一点儿，不过并不显著；无论如何，实际中这样的情况是相当少见的。

38209：由于设备仍在使用中而不能被释放时，它会被恢复到列表的适当地方。如果这是数个拥有同样优先级交换设备之中的一个，它可能不会回到同以前一样的相对位置了——它将是具有同样优先级的设备的第一个而不是最后一个——不过列表仍然是按照优先级进行排序的。

从交换设备列表上删除一个仍有可能被我们又放回原处的设备，这看起来就象是在做无用功——为什么不等到可以确信它可以被删除时再删除它呢？

答案在于经由一系列利用 **swap_list** 的函数调用后，在前面代码行对 **try_to_unuse**（38105 行）的调用能够结束。 如果正被删除的交换设备那时仍在 **swap_list** 里，那么终止这一切的代码将会给系统造成极大的混乱。

38223：若在一个分区上进行交换，**sys_swapoff** 函数将解除对它的引用。

38244：**sys_swapoff** 函数以使所有数据域无效和释放已分配的内存而告终。特别的，这行代码清除 **SWP_USED** 位，这样内核就会在它再次利用该交换设备时知道它已经是不可用的了。接下来，**sys_swapoff** 函数清除 **err** 指示符并返回成功。

Sys_swapon

38300：**sys_swapon** 函数是 **sys_swapoff** 的对应函数，它向系统列表里增加交换设备或交换文件。

38321：找寻未用的一个项。这里有一些微妙之处。读者可能会从 **nr_swapfiles** 的名字推断出它就是交换文件（或者设备）的数目，但是实际它不是。它是曾被使用过的 **swap_info** 的最大索引值，而且从不会被降低。（它记录着这个数组被使用的最高峰值。）因此，把 **swap_info** 中的这许多项循环一遍的结果是，要么发现未用的一个项，要么在最后一次循环增量后让 **p** 指向第 **nr_swapfiles** 项之后。在上述的后一种情况下，若 **nr_swapfiles** 比 **MAX_ SWAPFILES** 小，那么所有用过的项恰好会排在数组的左边，而循环就使得 **p** 指向它们右边的一个空位。这样，**nr_swapfiles** 就会被更新。

有趣的是，即使 **nr_swapfiles** 不是最高峰值而是活动交换设备的计数值，循环也能正确执行。不过若我们改变了 **nr_swapfiles** 的原意，文件里的其它代码就会有问题了。

38328：在 **swap_info** 里找到了一个未用的项；**sys_swapon** 函数开始对其进行填充。这里所提供的一些值将会发生变化。

38338：若 **SWAP_FLAGS_PREFER** 被置位，**swap_flags** 的低端 15 位就被编码为所需的优先级。（这里使用的常量和接着的几行代码在 17510 行进行定义。）否则，就不指定优先级。如前所述，在此情况下的缺省作法是给每一个设备分派一个逐渐降低的优先级，其目的是在无须人工干预时也能得到令人满意的交换性能。

38344：保证内核允许交换的文件或设备可以被打开。

38352：检查提供给 **sys_swapon** 函数的是一个文件还是一个分区。若 **S_ISBLK** 返回为真，它就是一个块设备，即磁盘分区。在此情形下，**sys_swapon** 函数继续确保能够打开该块设备而且内核此时没有同它进行交换。

38375：同样的，若给定的不是分区，**sys_swapon** 函数必须确保它是一个普通文件。若是文件，函数还要确保内核此时没有同该文件进行交换。

38384：如果两项测试均告失败，**sys_swapon** 函数就不会再被请求在磁盘分区或文件上进行交换；它已经拒绝了该尝试。

38396：从交换设备里把头页面读入 **swap_header**；这是一个在 17516 行定义的 **union swap_ header** 联合体类型。

38400：检查一串特征字节序列，该序列记述了交换头部的版本信息，它是由 mkswap 程序给出的。

38412：交换类型 1。此时，该头页面被当作一个大的位映射图，每一位代表设备中剩下的一个可用页面。同其它页面一样，头页面也是 4K 字节，即 32K 比特。由于每一位表示一个页面，设备就可以拥有 32768 个页面，也就是每个设备总计 128MB。（实际上要稍小一些，因为头页面的最后 10 位用于签名，这样我们就不能假定它们对应的 80 个页面也是可用的；另外头页面本身也不能用于交换。）如果实际设备比这个值小，那么头页面中的一些位就不起作用。在 38417 行，函数进入循环来检查哪些页面是可用的，并对它正在创建的 **swap_info_struct** 的 **lowest_bit**、**highest_bit** 以及 **max** 成员进行设置。

注意这个头页面位映射图不会永远被保持——当 **sys_swapon** 函数结束时它就会被

释放。内核利用交换映射表来跟踪正在使用的页面；该头页面位映射图仅被用来设置 **lowest_bit** 和其它 **swap_info_struct** 结构体的成员。

38427：分配交换映射表并把所有使用计数值设置为 0。

38440：交换类型为 2 的交换并没有减轻交换区容量的限制，不过它以一种更自然和有效的方式贮存头部的信息。在此情形之下，**swap_header** 的 **info** 成员就包含了 **sys_swapon** 函数所需的信息。

38451：新的交换头部版本不需要 **sys_swapon** 函数把头页面当作一个位映射图来计算 **lowest_bit**、**highest_bit**，和 **max** 的值——**lowest_bit** 总是 1，另外两个值可以从明确储存在头部的信息在定长时间内计算出来。这要比执行 32768 次位测试的循环快的多也简单的多，而且后者的定义语句甚至比前者要多出两倍以上！尽管如此，这部分以及余下的工作从概念上讲还是与以前十分相似的；**sys_swapon** 函数只不过是从交换头部直接获取了它所需要的大部分信息，而无须在计算它们而已。

读者现在可以看出我刚才撒的一个小谎；版本类型为 2 的交换实际上真正克服了交换区容量的限制。在这个版本中，文件末尾的 80 个页面不会由于交换头部签名而不可利用，因此单独一个设备有可以有 320K 用于交换。不过上限仍然是大概 128MB。

38491：**sys_swapon** 函数忽略读取头部。它把设备交换映射表的第一个元素设置为 **SWAP_MAP_BAD**（17552 行）以避免内核在头页面上进行交换。

38492：分配加锁映射表并清零。

38499：更新可用的交换页面总数，并对此结果显示一个消息。（在 38502 行，从移位计数器里减去 10 以便输出结果是千字节表示，2^{10} 就是 1K。）

38505：在交换设备的逻辑列表中插入新元素，仍遵循优先级排序的顺序。这里的代码从功能上是与 **sys_swapoff** 函数中相应的代码一样的，所以没理由把它们分离开来。一个能代替两者的内嵌函数就能简单的解决问题。

38519：进行清理工作，然后结束。

内存映射 mmap

mmap 是一个重要的系统调用，它允许为不同目的而设置专用的独享内存区域。该内存可能是一个文件或其它特别对象的代理，在这种情形中，内核将保持内存区域和潜在对象的一致，或者该内存可能是为一个应用程序所需要的简单的无格式内存。（应用程序通常不使用 **mmap** 来分配无格式内存区，因为此时 **malloc** 更符合其目的。）

mmap 最普遍的使用方法之一是为内核本身通过内存映射（memory-map）形成一个可执行文件（参见 8323 行的一个例子）。这是关于二进制处理程序如何同分页机制协同工作以提供所需要分页的可执行体，这正如本章早些时候所暗示的。可执行体通过 **mmap** 被映射为进程内存空间中的适当区域，然后 **do_page_fault** 函数调入执行体所需的剩余页面。

被 **mmap** 分配的内存可能被标识为可执行，其中充满了指令代码，随后系统跳入其中开始执行；这正是 Java Just-In-Time（JIT）编译器的工作方式。更简单的说，可执行文件能够被直接映射成一个正在运行的进程的内存空间；这项技术用于动态连接库的执行中。

执行 **mmap** 功能的内核函数是 **do_ mmap**。

do_ mmap

33240：**do_ mmap** 函数具有几个参数；它们共同定义应在内存中映射的文件或设备，并决定将被创建的内存区域的首选地址及其它特性。

33252：**TASK_SIZE** 和在 10867 行定义的 **PAGE_OFFSET** 值相同——即是 0xc0000000 或

3GB。这是用户进程所能拥有的最大内存，在此基础上代码才有意义：显然，如果要求 **do_ mmap** 函数分配大于 3GB 的内存，或者在 **addr** 之后的 3GB 内存空间没有足够的空间，分配请求就必须被放弃。

33275：如果 **file** 为 **NULL**，**do_ mmap** 函数将被请求去执行匿名映射（anonymous mapping）操作，这是一种并不与任何一个文件或其它特别对象连接的映射过程。否则，映射将被关联到一个文件，接着 **do_ mmap** 函数要继续检查为内存区域设置的标志位是否与用户在文件上允许执行的操作相兼容。举例来说，在 33278 行，函数要确保是否内存区可写，因为文件已经被打开并执行写操作了。省略这项判断将可能使文件打开时所作的检查发生混乱。

33307：允许调用程序强调 **do_ mmap** 函数应该或者在要求的地址上提供映射操作，或者根本没有什么也不做。如果提供地址，**do_ mmap** 函数只需保证提供的地址从一个页面的边界开始。否则，它将获得在 **addr** 处或之后的第一个可用地址（通过调用开始于 33432 行的 **get_unmapped_area** 函数），然后就使用这个地址。

33323：创建一个 VMA 并对其进行填写。

33333：如果内存映射着一个可读文件，则内存区域就被设为可读、可写和可执行。（**do_ mmap** 函数可以很快的取消写许可——这只是假定）另外，如果要求共享该内存区域，那么现在就可以满足该请求。

33347：若文件不可写，则内存区域也必须不可写。

33351：在此情形中，没有这样的文件，使得 **do_ mmap** 函数必须与该文件的打开模式和许可权限相一致——就允许函数自由运行。因此，函数把内存区域设为可读、可写和可执行的。

33361：在地址范围建立时，利用 **do_munmap**（很快就会被讨论到）来清除任何旧的内存映射。因为新的 VMA 还没有插入进程列表之中（只有 **do_ mmap** 函数当前知道它的存在），所以新 VMA 不会被此次调用影响。

33406：不会再有错误发生。**do_ mmap** 函数把新 VMA 插入进程的 VMA 列表（或是它的 AVL 树），合并所有新近相连的段片（接下来会对 **merge_segments** 函数进行讨论），更新一些统计数字，并返回新映射的地址。

Merge_segments

33892：**merge_segments** 函数是一个有趣的函数，它把相邻的 VMA 合并成单独的一个大范围的 VMA。换句话说，如果一个 VMA 所覆盖（有意这样设计）范围是从 0x100 到 0x200，而另一个 VMA 的覆盖范围是从 0x200 到 0x300，并且两者保护信息相同，那么 **merge_segments** 函数就会用一个覆盖范围从 0x100 到 0x300 的单独 VMA 来代替它们。（注意函数名中的“segments”并不暗示此时我们采用 CPU 分段机制。）**merge_segments** 函数的参数是结构体 **struct mm_struct**，它包含了我们该兴趣的 VMA 以及可能进行合并的开始地址和终止地址。

33897：**find_vma_prev** 函数将其 **vm_end** 定位在给定的 **start_addr** 之后的第一个 VMA 上——由此，第一个 VMA 可能会包括 **start_addr**。回忆一下 **find_vma_prev** 函数，它也返回一个指向前一个 VMA 的指针 **prevl**（如果第一个 VMA 满足条件则该返回值是 **NULL**）。

33911：进入处理所有覆盖给定区间的 VMA 的循环。在该循环过程中，**merge_segments** 函数将尝试把每一个段片都与其前一个段片进行合并，而前一个段片的值可以通过 **prev** 获得。

33921：绝大部分条件判断都是相对直截了当的，不过最后一个测试就不这么简捷了。它确保 **prev** 和 **mpnt** 是连续的——也就是在 **prev** 的结尾和 **mpnt** 的开头之间没有未被映

射的内存。即使检测结果是一个的 **vm_end** 和另一个的 **vm_start** 相等，这两块区域在这一点上也未必一定相互覆盖——回忆一下，**vm_end** 是要比 VMA 拥有的最后地址还要大一位的。从 33926 行到 33932 行的代码为被映射文件和共享内存坚持了同样的特性：一块区域的末尾要等于下一块的开头。

33937：**merge_segments** 函数找到了可以合并的 VMA。它把 **mpnt** 从 VMA 列表（还可能是 AVL 树）里移出，再将它存入 **prev**。要注意的是即使 VMA 的数目降到了 **MIN_MAP_COUNT** 以下，它都不会拆除 AVL 树。

33948：如果将要消失的 VMA 是一个被内存映射的文件的一部分，**merge_segments** 函数就删除它对该文件的引用。

do_munmap

33689：**do_munmap** 函数明显是 **do_mmap** 函数的反作用函数；它从一个进程的内存空间里废除虚拟内存映射。

33695：如果 **do_munmap** 函数被要求取消映射的地址不是页面对准的，或者地址区域位于进程的内存空间之外，那么很明显它就是无效的，因此请求就会被拒绝。

33699：如果连一个页面也没有被释放，就拒绝尝试。

33707：查找包括给定地址的 VMA。令人奇怪的是，**do_munmap** 函数返回的是 0——而不是错误——如果地址不在任何一个 VMA 之内的话。从某种意义上讲，这是正确无误的；**do_munmap** 函数被要求用来确保一个进程不再对特定内存区域进行映射，如果一开始就没有这种映射的话，那就很容易办到。不过这仍颇为奇怪；在调用者看来这是一个错误而且 **do_munmap** 函数也应该报告这个错误。然而，某些调用程序却希望它如 33361 行的示例那样执行工作。

33717：如果给出的内存区域整个在单独的一个 VMA 中，但又不在该 VMA 的一端，那么移去这段区域就会在封闭的 VMA 里生成一个空洞。内核是不会容忍这个空洞的，因为按照定义，VMA 应该是连续的一段内存。因此在这种情况之下，**do_munmap** 函数就需要创建另一个 VMA，使得空洞的两边各有一个 VMA。尽管如此，如果内核已经为该进程创建了所允许的所有 VMA，那么函数就不能这样做了，所以此时 **do_munmap** 函数不能满足请求。

33730：标识所有与该区域相交迭或在区域里的 VMA 为空闲状态，同时把每一个都放在本地堆栈 **free** 里。顺着这个过程，**do_munmap** 函数会把 VMA 从它们的 AVL 树中删除，如果有的话。

33743：**do_munmap** 函数已经建造了要释放的 VMA 堆栈，现在释放它们。

33748：计算要释放的准确范围，要牢牢记住的是这个范围可能不能以完整的 VMA 来度量。假如为 **min** 和 **max** 的定义适当，这三行可以被写成如下代码：

```
st  = max ( mpnt -> vm_start, addr );
end = min ( mpnt -> vm_end,   addr + len );
```

由此，**st** 是 **do_munmap** 函数实际开始释放区域的开头，**end** 是该区域的结尾。

33765：如果 VMA 是共享映射的一部分，**do_munmap** 函数通过调用 **remove_shared_vm_struct**（33140 行）来断开 **mpnt** 与共享 VMAs 列表的链接。

33759：更新 MMU 数据结构，它对应于这个 VMA 里当前被释放掉的子区域。

33765：调用 **unmap_fixup** 函数来修补映射，我们接下来就会对这个函数进行研究。

33773：**do_munmap** 函数已经释放了该范围内由 VMA 代表的所有映射；最后重要的一步就是要为同一区域释放页表，这是通过调用 **free_pgtables**（33645 行）实现的。

Unmap_fixup

33578：**unmap_fixup** 函数修复给定 VMA 的映射，这可以或者通过对一端进行调整，或者通过在中间制造一个空洞，再或者通过把 VMA 完全删除的方法来完成。

33590：第一种情况比较简单：去掉整个区间的映射。**do_munmap** 函数仅仅需要关闭底下的文件或其它对象即可，如果它们有的话。读者可以看到，这无须把 VMA 本身从 **current->mm** 里移出；它已经被调用者删除了。因为 VMA 的全部范围将被解除映射，没有什么要向后推移的，所以 **unmap_fixup** 函数就此返回。

33599：接下来的两种情况处理把 VMA 从开头到末尾一块区间移去的问题。这也是比较简单的；它们的主要工作是要调整 VMA 的 **vm_start** 或 **vm_end** 成员。

33608：这是四种情况中最有意思的一种——从一个 VMA 的中间移去一块区域，从而会产生一个空洞。函数先开始要复制一份额外生成的 VMA 的本地拷贝，然后通过将 ***extra** 设置为 **NULL** 来通知调用程序该附加 VMA 已被使用。

33611：图 8-4 表示了分裂 VMA 的过程。大部分信息被直接从旧 VMA 复制到了新 VMA，在此之后，**unmap_fixup** 函数对两个 VMA 的范围都作了调整以解决空洞问题。原先的 VMA，**area**，被缩小到了表示低于空洞的子区域，而 **mpnt** 则表示高于空洞的子区域。

33626：把全部新子区域插入 **current->mm**。

33629：在除了第一种的其它情况里，**unmap_fixup** 函数保持了旧的 VMA。它缩小了，但还未消失，因此它将被插回到 VMA 的 **current->mm** 集合中。

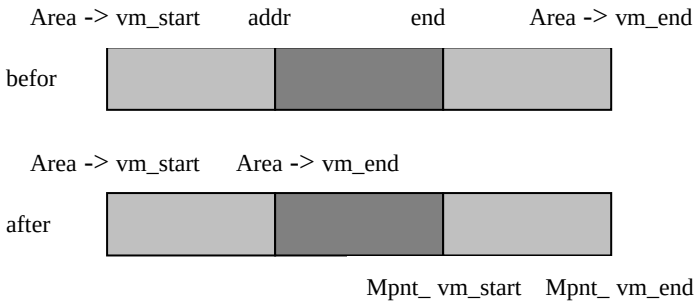

图 8.4 分裂 VMA

用户空间和内核空间

动态内存

用户任务和内核本身都经常需要快速分配内存。C 程序一般使用著名的 **malloc** 和 **free** 函数来完成这项工作；内核也有它自己类似的机制。当然，内核必须至少提供支持 C 语言的 **malloc** 和 **free** 函数的低级操作。

在 Linux 平台上，就像其它的 Unix 变种一样，一个进程的数据区分为两个便于使用的部分，即栈（stack）和堆（heap）。为了避免这两个部分冲突，栈从（准确的是接近）可用地址空间的顶端开始并向下扩展，而堆从紧靠代码段上方开始并向上扩展。虽然可以使用 **mmap** 在堆和栈之间分配内存，但是这部分空间通常是没有使用的内存的空白地带。

即使不去研究有关的内核代码（不过我们还是要继续这项工作），读者也能对这些地址区间所处位置有相当好的了解。下面的短程序显示了几个挑选出来的对象的地址，它们分处于三种不同内存区域之内。由于种种理由，我们不能保证它可以被移植到所有平台上，不过它可以在 Linux 的任何版本下工作，而且也应该可以被移植到你所尝试的大部分其它平台上。

P515-1 代码

在我的系统上，我得到了如下的数字。你的结果可能会稍有不同，除了所使用的编译器标志外，它还取决于你的内核及 gcc 的版本。即使不完全相同，它们也应该与下面结果相当接近。

P515-2 代码

从这里你不难看出，如果使用大概的数字的话，栈从接近 0xC0000000 处开始并向下生长，代码从 0x8000000 处开始，而堆则如前所述从临近代码上部的地方开始并向上扩展。

Brk

系统调用 **brk** 是一个在 C 库函数 **malloc** 和 **free** 底层的原语操作。进程的 **brk** 值是一个位于进程堆空间和它的堆、栈中间未映射区域之间的转折点。从另一个角度看，它就是进程的最高有效堆地址。

堆位于代码段顶端和 **brk** 之间。如果 **brk** 底下的可用自由空间不够满足请求，C 库函数 **malloc** 就抬高 **brk**；如果被释放的空间位于 **brk** 之下，就降低 **brk**。顺便说一句，Linux 是我所知道的唯一的在使用 **free** 函数时真正的减少进程内存空间的 Unix 变体；其它我所经历过的所有 Unix 商业版本实际上都是保留该进程的空间的——显然这是“以防万一”的作法。（其它 Unix 的自由版本可能同 Linux 一样，不过我没有使用过。）另外，对于大量的分配工作，GNU 的 C 库使用 **mmap** 和 **munmap** 系统调用来执行 **malloc** 和 **free**。

代码、数据，以及栈的关系如图 8-5 所示。

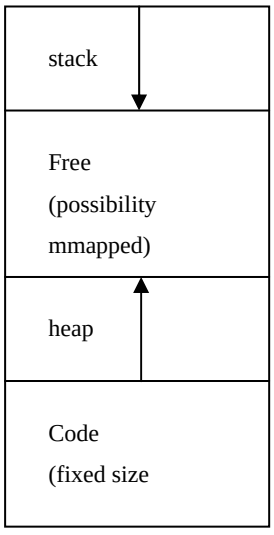

图 8.5 代码、数据和栈

Sys_brk

33155：实现 **brk** 的函数是 **sys_brk**。它可以修改进程的 **brk** 值，还可以返回一个新值。如果无法修改 **brk** 的值，返回的 **brk** 值就等于其原值。

33177：如果 **brk** 的新值位于代码区域之中，它就明显偏低而必须被抛弃。

33179：通过使用宏 **PAGE_ALIGN**（10842 行）把 **brk** 参数向上取整到地址更高的下一个页面。

33180：按页对准进程原有的 **brk** 值。这看起来有些多余，因为如果进程的 **brk** 只是在这里被设置，它就一定是按页排列的。但是在初始化一个进程的时候，进程的 **brk** 可以被设置在别的地方，代码并不会把它按页对准排列。不管进程的 **brk** 在哪里被设置，把它按页对准都可能会快一些；允许内核在这里跳过一次页对准操作，而且由于此处要比别的地方更频繁的对进程的 **brk** 进行设置，它应该不会降低执行效率而且还会少许提高。

33185：**brk** 被降低了，不过还没有进入代码区域，因此尝试被允许。

33192：如果堆的大小有限制，它就要被考虑。图 8-5 清楚的表明，**brk - mm->end_code** 是堆的大小。

33197：如果 **brk** 扩展到了已被一个 VMA 所内存映射的（**mmap**ped）区域，它就是不可利用的，因此这个新 **brk** 值要被舍弃。

33201：最后一项必要的检查是察看是否存在足够的自由页面用于空间分配。

33205：使用 **do_mmap** 函数（33240 行）为新区域分配空间。然后，**sys_brk** 函数更新进程的 **brk** 的位置并返回新值。

Vmalloc 和 vfree

内核编程中一个有趣的方面是并没有像应用程序编程人员通常所想当然的那样能够得到很多服务。就拿 **malloc** 和 **free** 作为例子，它们就是建立在一个内核原语 **brk** 之上的 C 库函数。

假使内核被修订以使其可以和标准 C 库连接，并使用它的函数 **malloc** 和 **free**，那么最终结果将是既笨拙又缓慢——这些函数被要求从用户模式调用，所以内核将不得不切换到用户模式去调用它们，然后它们又不得不掉转回到内核，还必须要对整个过程进行监控，等等。为了避免这一切，内核有许多十分熟悉的函数的自己的版本，它们包括 **malloc** 和 **free** 在内。

的确，内核提供了像 **malloc** 和 **free** 一样的两对独立的函数。第一对是 **kmalloc** 和 **kfree**，管理在内核段内分配的内存——这是真实地址已知的实际和物理内存块。第二对是 **vmalloc** 和 **vfree**，用于对内核使用的虚拟内存进行分配和释放。由 **kmalloc** 返回的内存更适合于类似设备驱动的程序来使用，因为它在物理内存里而且是物理连续的。不过，**kmalloc** 要比 **vmalloc** 所能使用的资源少，因为 **vmalloc** 还可以处理交换空间。

vmalloc 和 **vfree** 的一部分也是通过 **kmalloc** 和 **kfree** 来实现的，因为它们需要一部分不可交换的内存用于登记操作（bookkeeping）。**kmalloc** 和 **kfree** 又依次使用**__get_free_ pages**、**free_pages**，以及其它低级页面操作函数实现的。

在此我不对 **kmalloc** 和 **kfree** 进行解释，不过本书中提供了相关代码以供读者阅读（分别见 37043 和 37058 行）。我将要讨论的是更有意思的函数 **vmalloc** 和 **vfree**。

Vmalloc

38776：**vmalloc** 函数拥有一个参数，即要分配的内存区域的大小。函数返回指向分配区域的指针，如果无法分配就返回 **NULL**。

Vmalloc 可以分配内存的虚拟地址范围是由常量 **VMALLOC_START**（11081 行）

和 **VMALLOC_END**（11084 行）决定的。**VMALLOC_START** 从超过物理内存结束地址 8MB 的地方开始，以便对任何在这一区域错误的内核内存访问进行截获，**VMALLOC_END** 在接近可能的最大 32 位地址 4GB 的地址处。除非你的系统拥有比我的系统多得多的物理内存，否则这就意味着几乎整个 CPU 地址空间都潜在的可为 **Vmalloc** 所用。

38781：**vmalloc** 函数首先把要求的区域大小向上取整到地址更高的下一个页面边界，如果它不在一个页面的边界上的话。（**PAGE_ALIGN** 宏在 10842 行定义。）如果最终范围结果太小（0）或明显过大，则请求会被拒绝。

38784：利用 **get_vm_area** 来为 **size** 大小的块定位一段足够大的内存区域，这个函数接下来会进行介绍。

38788：通过调用 **vmalloc_area_pages**（38701 行）保证能够建立页表映射。

38792：返回被分配的区域。

get_vm_area

38727：**get_vm_area** 函数返回从 **VMALLOC_START** 到 **VMALLOC_END** 的一段自由内存区间。通常这就是 **vmalloc** 函数的工作；它还被用于我未曾提及的其它少数场合。调用程序有责任确保参数 **size** 是一个非零的页面大小的倍数值。

vmalloc 函数采用所谓的首次适应算法（first-fit algorithm），因为它返回一个指向定位区域的指针，该区域是它所能找到的第一个满足请求的区域。除此而外，还有最佳适应算法（best-fit algorithm），该算法选取足够满足需求的最小的一块可用自由区域进行分配，以及最坏适应算法（worst-fit algorithm），该算法总是分配最大的一块可用自由区间。每种分配方式都有优点和缺点，不过首次适应算法在这里对要达到的目的来讲，就已经非常简单、快捷而且足以满足要求了。

38732：分配一个 **struct vm_struct** 来代表新的区域。被分配的区域用一个有序链表，即 **vmlist**（38578 行）来维护，该链表是由 **struct vm_struct**s 构成的。包括 **struct vm_struct** 结构体的头文件被省略以节约空间，不过结构体的定义十分简单：

```
struct vm_struct {
  unsigned long flags;
  void* addr;
  unsigned long size;
  struct vm_struct* next;
};
```

如图 8-6 所示，链表的每一个元素都与单独一块已分配了的内存块相关联。形象的看起来，**get_vm_area** 函数的任务就是在已分配的区域之间找出足够宽的间隔。

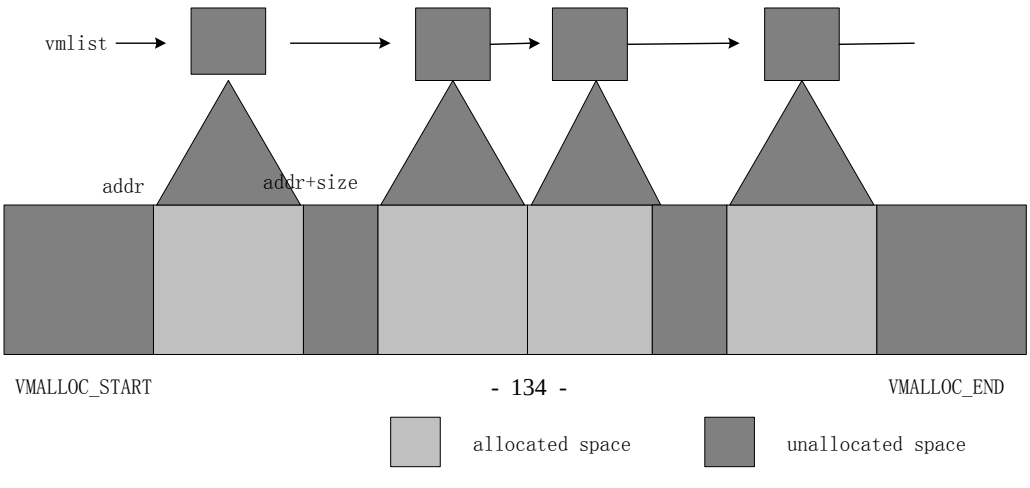

图 8.6 VMLIST 列表

38737：沿着链表进行循环。循环的结果要么是找到一个足够大的自由区间，要么是证明这样的区域不存在。它会先从 **VMALLOC_START** 开始尝试，然后挨个尝试紧随着每块被分配区域之后的地址。

38746：链表为空或者循环发现了一个足够大的新块；无论哪种情形，现在 **addr** 都是最小可用地址。填充新的 **struct vm_struct** 结构体，它将会被返回。

38747：给保留块增加一个页面的大小（x86 平台上是 4K），来捕获内核超出的内存——可能的话还包括下一个更高地址块下方的内存。因为在决定是否当前区域足够大的时候（38738 行）并没有把这些额外的空间算在内，那么保留区域可能会与接下来的一个相重叠，而且内核内存中超出而进入这个“额外”区域的部分也确实可能覆盖到被分配了的内存。事实不是如此吗？

事实不是这样。我们很容易证明 **addr** 总是页对准的，而且我们也已知道 **size** 总是页面大小的倍数。因此，**addr + size** 要小于接下来区域的开始地址，它至少是一整页。当然超出范围多于一页的内存会进入下一个区域，不过超出范围少于一页的内存就不会这样。

因为内核不会为额外内存建立页面映射，所以对它的错误访问将造成不可解决的页面错误（这在 Linux 的现代版本中几乎还未听说过！）。这将会带给内核一次痛苦的中断，不过那要比允许内核悄然无息地破坏自己的数据结构要好一些。至少你可以立刻知道这个折磨人的系统停机，它可以帮助你诊断问题所在；而后一种作法可能在内核已经破坏了你的磁盘之后，才能看出它的危害。

Vfree

38759：**vfree** 函数比 **vmalloc** 简单得多（要是把 **get_vm_area** 加进 **vmalloc** 至少是这样的），不过为了完整起见，我们还是要对 **vfree** 略为讨论。当然 **addr** 是要被释放的已分配区域的开头地址。

38763：在几项简洁而又完善的测试之后，函数沿着 **vmlist** 进行循环，搜索要释放的区域。这个线性查找过程使我想到一件有趣的事，假如采用一个如同 VMA 管理所用的 AVL 树那样的平衡树结构，也将会提高 **vmalloc** 和 **vfree** 函数的性能。

38764：当与 **addr** 相匹配的 **struct vm_struct** 被找到时，**vfree** 函数就把它从链表里分离出去，并释放该结构体和它所关联的页面，然后返回。每个 **struct vm_struct** 不仅记录它的初址还记录区域的大小，这一点对于 **get_vm_area** 是便利的，在这里同样也颇为便利，因此 **vfree** 函数是知道应该释放多大空间的。

38772：如果 **vfree** 函数在链表里找到了匹配项，它在此之前就应该已经返回了，所以没有找到匹配项。这是一个坏事，不过还未糟糕到不可收拾的地步。这样，**vfree** 函数以显示一个警告而结束。

转储内核（Dumping Core）

在一些情况之下，比如一个满是“臭虫”的程序试图去访问自己允许内存空间之外的内存时，进程可以转储内核。进行“转储内核”就是把一个进程的内存空间的映象（随同一些关于应用程序本身和其状态的识别信息一起）写入一个文件以备将来使用诸如 gdb 之类的调试器进行分析的过程（“内核”是一个差不多已经过时的内存术语）。

当然，或许你的代码从来不会犯这样的错误，但是这可能会发生在你隔壁不太聪明的程序员身上，而他可能在某一天会向你询问这件事，因此在此我要对此问题进行一些讨论。

不同的二进制处理程序完成转储内核的方式不同。（第 7 章里论述过二进制处理程序。）最常用的 Linux 二进制格式是 ELF，所以我们来看看 ELF 二进制处理程序是如何进行转储内核的。

Elf_core_dump

8748： **elf_core_dump** 函数由此开始。因为一个进程转储内存是由接受到一个信号而引起的（它也可能发送给自己，例如通过对 **about** 的调用），该信号编号在 **signr** 中被给出。**Signr** 对进程是否或者如何执行转储内存没有影响，但是在调试器里看内存文件的用户却想要知道是哪个信号导致内存转储的，它就像是一个关于出了什么错的提示一样。指向 **struct pt_regs**（11546 行）的 **regs** 参数包含一份对 CPU 寄存器的描述。**regs** 的重要性除了一些其它原因之外还在于它包含了 EIP 寄存器的内容，该寄存器是指令指针，它决定了收到信号时所执行的指令。

8771： 假如进程未通过一些基本检查则立即返回，这些检查中的第一个是确保 **dumpable** 标志被设置。进程的 **dumpable** 标志（16359 行）通常会被设置；它的清除主要是在进程改变其用户或组 ID 的时候。这似乎是一项安全措施。例如我们将不愿意创建一个被设定为 root 的不可读执行程序的可读内存文件——那会使得保证执行体不可读（出于安全考虑）的目的遭到失败。

elf_core_dump 函数此时也会返回，假如内存文件的大小限制使得连一个页面也无法转储，或者如果有其它线程要引用将要转储的内存。转储内核是和退出进程相关的，从用户的角度看来，只要进程任何一个线程还存在，它就没有消亡。

如果进程通过了这些测试，**elf_core_dump** 函数就继续运行并清除 **dumpable** 位以便它不会再次尝试转储进程的内存。（尽管这种情形不能会发生；我认为这只是预防式的编程设计。）

8785： 进入一个循环以对内存文件大小限制之内可以被转储的 VMA 个数进行计数。尽管 **elf_core_dump** 函数把计数植保存在叫做 **segs** 的变量里，它并不表明我们正对本章中所使用过的“内存段”进行计数。不要认为这个变量的名字有其它特别的附加涵义。

由于 **elf_core_dump** 函数在转储 VMA 之前要向内存文件写一些头部信息，而且这些头部的大小没有进行计算，因此输出结果可能会稍微超出内存文件的大小限制。这不难解决：一个简单的策略是在写入头部时递减 **limit**，并把循环计数移动到头部写入代码之后。实际解决方案要更麻烦一些，不过也并不是十分复杂。

8805： ELF 内存文件格式根据正式规范进行定义；第一个部分是描述文件的头部。结构体 **struct elfhdr** 类型（参见 14726 和 14541 行）定义了头部的格式，**elf_core_dump** 函数填写这个类型的一个局部变量 **elf**。

8827： 创建要转储到的文件名，并尝试打开这个文件。通过把 8828 行的**#if 0** 改变为**#if 1**，我们可以让内存文件名包括生成文件的执行程序的名字（或至少是名字的前 16 个字符——参见在 16406 行定义的 **struct task_struct** 的 **comm** 成员）。有的时候这是一个很有用的特性；能够一看到内存文件的名字就可以马上知道是什么应用生成的将是一件很好的事情。不过，这种行为并不标准，而且还有可能破坏已有代码——比如监视器脚本程序，它周期性地检查名为“code”的文件是否存在——所以缺省行为还是为遵守标准惯例而把文件命名为普通的“code”。尽管如此，发现这么一个可以调整的内核参数还是不错的。这个可选项也对 8756 行局部变量 **corefile** 那看似与众不同的定义方式进行了解释。

8853： 设置 **PF_DUMPCORE** 标识（16448 行），发出信号表明该进程正在转储内核。这个标识不在本书所涉及的任何代码中使用，它被用于读者将要了解的审计进程。审计进程（process accounting ）跟踪一个进程的资源使用情况和其它的一些相关信息——包括它是否在退出时转储内核——这些信息原本是用来帮助计算中心计算应向每个资源使用部门或用户收取多少费用的。这些日子都已经离我们远去了，难道我们不应该为此而感到高兴吗？

8855： 写入早先建立的 ELF 内存文件头部。这里要涉及一些隐含的流控制：定义在 8707 行的 **DUMP_WRITE** 宏使得 **elf_core_dump** 函数在写操作失败时关闭文件并返回。

8862： 跟在 ELF 内存文件头部之后的是一系列节点（note）；它们中的每一个都有特殊目的，记录着有关进程的特定信息。我们将逐一对其论述。一个注解（数据类型是 **struct memelfnote**，8666 行）包括一个指向辅助数据（它的 **data** 成员）的指针和该数据的长度（它的 **datasz** 成员）；填写一个注解的大部分工作就是填充辅助数据结构，然后使该注解指向它。

有些信息被存储在若干个注解里。代码中没有对这种重复进行解释，但是其中至少有一部分原因是从 Unix 的变种中拷贝它们的行为方式。保持文件格式和其它平台一致有助于把诸如 gdb 这样的程序移植到 Linux 上来；少许重复要比延迟移植版本的进度和增加诸如此类的关键工具的维护复杂要好得多。

8865： 注解 0 在辅助数据结构体（类型 **struct_elf_prstatus**；参见 14774 行）里记录了进程的继承关系、信号量，以及 CPU 的使用情况。我们需要特别注意 8869 行的 **elf_core_dump**，它存储了引起进程转储内核的信号编号。所以当你（或者是你隔壁那个初级程序员）在一个内存文件上运行 gdb 而它显示”Program terminated with signal 11, Segmentation fault”的时候，你就会知道该信息是从哪里来的了。

8916： 注解 1 在辅助数据结构体 **psinfo**（属于类型 **struct_elf_prpsinfo**；参见 14813 行）里记录了进程的属主、状态，优先级等等信息。8922 行有一个虽然正确，但很不寻常的指向一个文字字符串常量的数组下标；被选择的字符是进程状态的一个记忆码。这与 ps 程序的 STAT 域报告的状态字是一样的（除非下标溢出）。更有意思的是 8945 行，代码把执行体的名字（如前所述，最多 16 个字符）复制进了注解。Gdb 和程序“文件”都用这个字段来报告是哪一个程序生成的内核转储。

8948： 节点 2 记录转储进程的 **struct task_struct**，这明显存储了关于该进程的大量必要信息。因为 **struct task_struct** 内的一些信息是由当调试器检查代码时便不再有效的指针组成的，**elf_core_dump** 函数随后还会分别转储一些指针所指向的信息——最紧要的，如进程的内存空间。

8954： 如果这个系统包含一个 FPU（浮点计算单元），那么就会据此而生成一个注解。否则，8957 行对所要存储的注解数目进行递减。

8968： 对于每个被创建的注解，都有一个描述该注解的头部；而注解本身会紧随其后。注解头是 **struct elf_phdr** 类型；参见 14727 和 14581 行它的定义。

8992： 这是写入进程内存空间的第一步。在这里，函数写入头部信息（又一次是 **phdr**），该头部描述了它将要写入的所有 VMA。

9016： 最后，**elf_core_dump** 函数才真正地写入它先前辛辛苦苦创建好的各个注解（内存文件）。

9022： 在文件里向前跳过 4K 到达下一个边界，内存文件真正的数据是从这里开始的。完成此项操作的 **DUMP_SEEK** 宏在 8710 行定义，像 **DUMP_WRITE** 宏一样，假如搜索失败它也会导致 **elf_core_dump** 函数的返回。

9024：在所有那些准备之后，这里的工作简直有些虎头蛇尾。不过，这才是转储内核的主要

部分：写入进程的每一个 VMA 直至先前计算出并保存在 **segs** 里的上限。接下来是少许收尾工作，然后 **elf_core_dump** 函数就完成了使命。

第 9 章 System Ⅴ IPC

Unix 从开发的早期就提供了管道的机制，管道在同一机器的两个进程间的双向通信方面工作的相当出色。后来，BSD（Berkeley Software Development）的 Unix 版本又提供了通用的套接字 socket，它用来在不同机器的两个进程之间进行通信（或者是同一机器的）。

Unix System V 版本增加了被视为一体的三个机制，现在它们被统称为 System V IPC。像管道一样，这些机制都可以用于同一机器上的进程间通信，不过与管道和套接字不同的是，System V 的 IPC 特性使得同一机器上的许多进程之间都可以互相通信，而不是仅限于两个进程。而且，管道——不是套接字——还有一个更大的限制就是两个通信中的进程必须相关。它们必须有一个建立管道的共同祖先进程——通常情况下，一个进程是另一个的父进程，或者这两个都是为它们建立管道的父进程的子进程。System V IPC 像套接字一样使得进程间通信（IPC）不需要有共同的继承关系，只需要一个经过协商的的协议。

组成 System V IPC 的三个进程间通信机制是：消息队列、信号量和共享内存。

消息队列

System V 的消息队列（message queues）是进程之间互相发送消息的一种异步（asynchronously）方式，在这种情形之下，发送方不必等待接收方检查它的消息——即在发送完消息后，发送方就可以从事其它工作了——而接收方也不必一直等待消息[①]。对消息进行编码和解码是发送者和接受者进程的工作；消息队列的执行并不会给它们特别的帮助。这就形成了一个实现起来相对比较简单的通用机制，尽管是以增加应用程序的复杂度为代价来获得这种简明性的。

这里是一个可能发生在 SMP 机器上的简单的应用情景。运行在一个 CPU 上的调度程序把工作请求发送到一个特定的消息队列上。工作请求可能以各种形式出现：用来破译代码的一组密码、需要进行计算的在不规则图形里的象素范围、在一个原子系统里要更新的一部分空间，或者诸如此类的任务。与此同时，工作者进程在其它 CPU 上运行，只要它们空闲就从消息队列中检索消息，然后再把结果消息发送到另一个消息队列上去。

这种体系结构很容易实现，而且假定选择好了每个消息中被请求工作的粒度，就能极大的提高机器中 CPU 的利用效率。（还要注意的是，因为调度进程可能不用做许多工作，所以调度进程的 CPU 上大部分时间也可以运行一个工作者进程。）以这种方式，消息队列可以被用作是远程过程调用（RPC）的一种低级形式。

新消息总是加在队列的末尾，不过它们并不总是从排头移出；你将能够在本章中看到，消息可以从队列的任何地方被移出。在某个方面，消息队列与语音邮件类似：新消息总是在末尾，不过消息接收方可以从列表的中间接收（以及删除）消息。

消息队列概述

首先对消息队列进行介绍是因为它的实现最简单，不过它仍然体现出了几个所有三种

[①] 原文为：”the receiver doesn’t have to go to sleep if no messages are waiting.”，直译是：如果没有消息正被等待，接收方也不必进入休眠。

System V IPC 机制都具有的共同结构特征。

给进程提供了四种与队列相关的系统调用：

- **msgget**——一个不合时宜的名字：读者可能认为这会得到一个等待的消息。但实际它不会。调用者提供一个消息队列键标（key），如果存在一个队列，**msgget** 就用该键标为它返回一个标识号，如果没有队列，就用它为一个新的消息队列返回一个标识号。因此，**msgget** 所得到的不是一个消息，而是唯一标识一个消息队列的标识号。
- **msgsnd**——向一个消息队列发送一条消息。
- **msgrcv**——从一个消息队列中接受一条消息。
- **msgctl**——在消息队列上执行一组管理操作——检索关于它的限制的信息（比如队列所允许的最大消息数据量）、删除一个队列，等等。

Struct msg

15919：**struct msg** 代表在队列中等待的一个消息。它有如下成员：

- **msg_next**——指向队列中的下一个消息——当然假如这是最后一个消息就为 **NULL**。
- **msg_type**——用户指定类型编码；它的使用在本章讨论消息如何被接收时再进行分析。
- **msg_spot**——指向消息内容的开头。读者后面将看到，为消息分配的空间总是紧靠在 **struct msg** 的上边，因此 **msg_spot** 恰恰指向 **struct msg** 末尾之后的位置。
- **msg_stime**——记录消息被发送的时间。因为消息以先进先出（FIFO）顺序保存，所以队列中的消息拥有的 **msg_stime** 值就是单调非递减的。
- **msg_ts**——记录消息的大小容量（“ts”是“text size”的缩写，尽管消息不一定非要是人们可以读懂的文本）。一条消息的最大容量是 **MSGMAX**，它在 15902 行定义为 4056 字节。推测一下，这应该是 4K（4096 字节）减去一个 **struct msg** 的结果。不过 **b** 只有 20 字节，因此还有另外的 20 字节有待说明。

Struct msqid_ds

15865：msqid_ds 代表一个消息队列。它有如下成员：

- **msg_perm**——说明哪一个进程可以读写该消息队列。
- **msg_first** 和 **msg_last**——指向队列中的第一个和最后一个消息。
- **msg_stime** 和 **msg_rtime**——分别记录消息被发送入队列的最后时间和消息从队列中读出的最后时间。（一项挑战：什么时候队列中最后一条消息的 **msg_stime** 成员不等于队列本身的 **msg_stime** 成员？至少有两个答案，但是你所掌握的信息现在只能得出一个——你将不得不仔细阅读代码以寻求另一个解答。）
- **msg_ctime**——上一次队列改变的时间——它被创建的时间，或是上一次利用 **msgctl** 系统调用设置参数被确信的时间。
- **wwait**——等待写消息队列的进程队列。因为消息发送是异步的，通常进程把一个消息写入消息队列后就可离开。但是，为了避免拒绝服务（denial-of-service）的攻击，队列有一个最大容量——若没有这个限制，一个进程就可以不断的向一个没有读者的队列发送消息，强迫内核为每个消息分配内存直至空间耗尽。因此，当一个队列达到其最大容量时，想要发送消息给该队列的进程必须等待，直到队列中有了空间容纳新的消息，或者发送消息的尝试被立刻拒绝为止（读者将看到，进程能够选择它所希望的执行方式）。**wwait** 队列保留那些决定等待的进程。

- **rwait**——与之类似，消息通常可以从消息队列中立刻读出。但是如果没有正等待被读的消息将怎么办呢？进程再次进行选择：它们要么立刻重获控制（用一个错误代码表示读消息失败）要么进入休眠等待消息到来。
- **msg_cbytes**——当前在队列中的所有消息的总字节总数。
- **msg_qnum**——队列中消息的总数。对于能够进入队列的消息数目没有明确的限制——这也是一个问题，本章随后还要进行解释。
- **msg_qbytes**——队列中允许存储的所有消息的最大字节数；把 **msg_cbytes** 和 **msg_qbytes** 进行比较来确定是否还有空间容纳新消息。**msg_qbytes** 缺省为 **MAGMNB**，尽管这个限制可以被有适当权限的用户动态地增加。

  MAGMNB 在 15904 行定义为 16384。有四个理由说明为什么这个界限被定的这样低。第一，实际上，你通常不需要把太多的信息包括在一个给定的消息中，所以这个界限并不是十分苛刻的。第二，如果消息发送方的速度远远领先于接收方，那么让消息能多包含些信息可能也没有意义——它们还将是接收方要费些时间才能得到的一大块数据。第三，这个每队列 16K 的界限可以与潜在的 128 个队列相乘，总计达 2MB。

  但是采用这个界限的主要原因还是为了避免先前提及的拒绝服务攻击。然而，没有什么能防止应用程序发送长度为零的（也就是空的）消息。**msg_qbytes** 不会被这样的消息影响，而且仍然要给消息头分配内存，因此拒绝服务攻击仍然是可行的。解决这个问题的一个方案是引入一个独立的、对允许进入队列的消息总数进行限制的界限；另一个方案是从 **msg_qbytes** 中减去整个消息长度——包括消息头。再一种解决方法当然是不允许有空消息，但这又将同兼容性相抵触。
- **msg_lspid** 和 **msg_lrpid**——最后消息发送方和最后消息接收方的 PID。

Msgque

20129：消息队列实现中的主要数据结构是 **msgque**，一个指向 **struct msqid_ds** 的指针数组。这些指针有一个是 **MSGMNI**（15900 行定义为 128），它等于 128 个消息队列的最大值。为什么不只是用一个的数组而要用一个指针数组呢？一个原因是为了节省空间：替代一个 128 个 56 字节结构体的数组（7168 字节，7K），**msgque** 是一个 128 个 4 字节指针的数组（512 字节）。在正常情况下，当很少的消息队列投入使用，这能够节约好几千字节的空间。在最坏的情况时，所有的消息队列都被分配了，最大的消耗也只是 512 字节。唯一会引发的真正缺点是附加了一层间接转换，这意味着速度要有少许损失。

主要消息队列数结构之间的关系如图 9.1 所示。

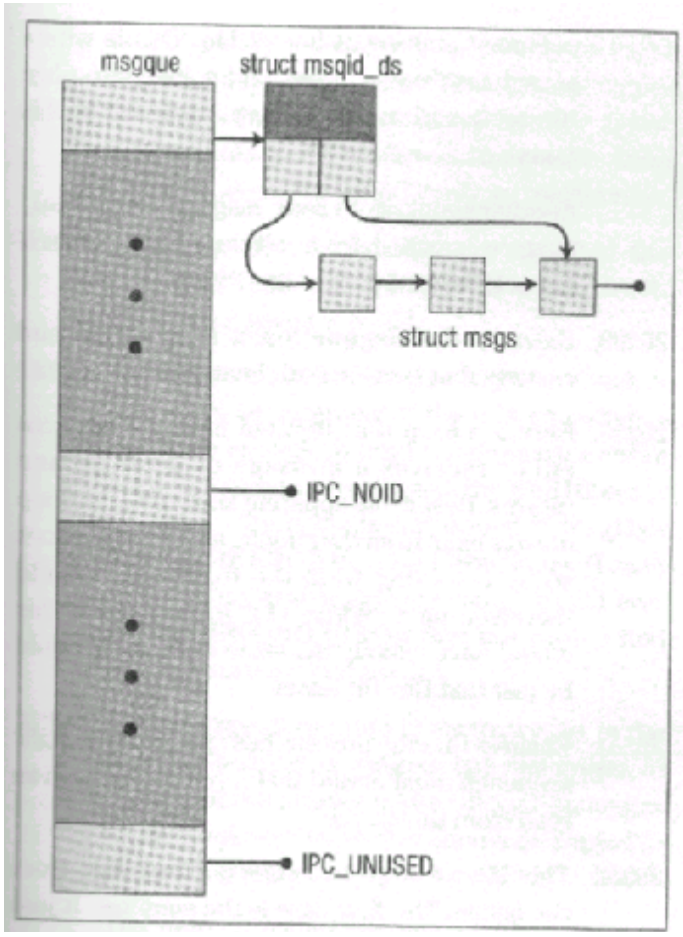

图 9.1 消息队列数据结构

Msg_init

20137：**msg_init** 用于消息队列实现时变量的初始化。它的大部分都是不必要，因为同样的变量已经在函数前面紧挨本段代码的声明中被初始化为同样的值了。

20141：然而这个把 **msgque** 的条目设置为 **IPC_UNUSED** 的循环是必要的。**IPC_UNUSED** 不在本书讨论之列，值为-1（能够更好的被映射为 **void***）；它代表一个没有使用的消息队列。**msgque** 条目可能接纳的其它特殊值是 **IPC_NOID**（也不在本书讨论之列）——这只是暂时的，也就是在消息队列被创建的时候。

Real_msgsnd

20149：**real_msgsnd** 实现 **sys_msgsnd** 的实质内容，即 **msgsnd** 系统调用。这里和内核的约定有一些偏差，该约定要在命名系统调用的“内脏函数[①]”时使用一个“do_”前缀。在 20338 行调用了 **real_msgsnd** 函数，在那里它处于 **lock_kernel/unlock_kernel** 函数对之中。（那两个函数在第 10 章中讨论——基本上，每次只能有一个 CPU 对内核加锁，这与 SMP 机器有关。）这是一种确保 **unlock_kernel** 得到执行的最佳方式——否则，**real_msgsnd** 复杂的流程控制将因需要在它退出时确保调用 **unlock_kernel** 而变得更加复杂。

[①] 原文为：”guts function”。”gut”通常的意思是“内脏”，这里是指本质的东西。

正如读者已经熟悉的，内核大多使用返回代码变量和 **goto** 语句来解决这样的问题。虽然它不能很好的适应每种情况，但是 **sys_msgsnd** 函数的方法更加清晰。例如，当一个函数必须获得多项资源，其中一些只有在以前所有资源请求都成功地被满足时才能提出请求，考虑这样可能引发什么样的后果。简单扩展的解决方法将需要大量函数——就像下边代码段所描述的：

P523 —1

很快，这样的代码将变得臃肿不堪，内核不这样做的原因就在于此。

20158：开始一系列条件判断。假如有了第一个测试，本行三项测试中的第二项就是不必要的——任何不能通过第二项测试的消息同样也不能通过第一项测试。虽然以后这种说法可能会不成立，假如 **MSGMAX** 的界限增加到足够高的话。（事实上，在写作本书时，完全消除 **MSGMAX** 界限的工作已在开展之中了。）

20164：消息队列标识号对两段信息进行编码：与之对应的 **msgque** 元素的索引在低端 7 位，一个序列编号（其作用随后就将讨论到）就位于紧挨这 7 位之前的 16 位里。现在所需要全部的就是数组下标部分。

20166：如果指定的数组下标处没有消息队列，或者正在创建一个，那么就没有消息可以进入队列。

20171：保存在消息队列中的序列编号必须和那个 **msgque** 参数里的编码相匹配。其思想是：在正确的数组下标处有一个消息队列并不代表它就是调用者所需要的消息队列。自从调用者引用一个队列之后，原先处于那个下标的消息队列可能已经被移去了而且在同一下标处创建了一个新的消息队列。16 位序列编号被周期性的增加，所以在同一下标处的新队列将有一个和旧队列不同的序列编号。（除非正好先创建了 65535 个其它的新队列，这是相当不可能的——或者是 131071 个其它的新队列，这就更不可能了。本章随后将对其进行解释，实际情况并非这样简单。）不管怎样，只要序列编号不匹配，**real_msgsnd** 就返回一个 **EIDRM** 错误来指示调用者所需要的消息队列已经被移出了。

20174：确保调用者有写消息队列的权限。类似的一个方法将在第 11 章详细介绍；在这里，可以简单的把它看作是类似于 Unix 文件权限应用的一个方法。

20177：检查如果提供的消息被写入队列，是否会超过队列所允许的最大容量。接下来一行代码再次检查同一件事，这显然是当代码从 2.0 系列的内核版本被转换过来时留下的一个编辑疏漏。在两次检查之间，曾经有过一些能够释放队列中的部分空间的代码。

20180：队列中没有空间。如果在 **msgflg** 里的 **IPC_NOWAIT** 位被设置了，这种情况发生时调用者就不会等待，这样的结果是返回 **EAGAIN** 错误。

20182：进程将要进入休眠状态。**real_msgsnd** 首先检查是否一条消息正在等待该进程。如果存在等待消息的话，就会用进程的休眠被该消息所中断的方式来处理它（进程可能已经休眠，就如随后所示的那样）。

20184：假如没有正在等待进程的信号，进程就进入休眠状态，直到有信号到达或移出队列中的一条消息时它才被唤醒。当进程被唤醒之后，它将再次向读列写入。

20190：为消息队列头（**struct msg**）和消息体分配足够的空间——正如前面所说，消息体将紧接在消息头后面存放。消息头的 **msg_spot** 直接指向该头部之后消息体开始的地方。

20196：从用户空间复制消息体。

20202：再次检查消息队列的合法性。**Msgque** 入口可能已经在 20184 行这个进程休眠时被其它进程修改过了，因此直到通过检查之前不能认为 **msg** 是有效的指针。

即便如此，这里看起来也有一个潜在的缺陷。如果在当前进程执行到这一步之前，该消息队列已被删除而另一个消息队列被设置在同一个数组下标的地方那又将怎样呢？在 UP 机器上是不会发生这种情况的，因为销毁消息队列的函数，**freeque**（20440 行），在销毁它之前将唤醒任何休眠于该队列的进程，而且在 **real_msgsnd** 完成之前 **freeque** 不会继续进行（本章后面将对 **freeque** 进行分析）。然而，在 SMP 机器上，这仍然是一个小小的隐患。

假如发生这种情形，**msgque[id]**将不是 **IPC_UNUSED** 或 **IPC_NOID**，但是 **msq** 指向的内存已经被 **freeque** 释放了，因此在 20203 行将废除无效的指针引用。

20209：填写消息头，将其入队，并更新队列自己相应的统计值（比如消息的总共大小）。注意只要有可能就推迟填写消息头的工作，所以假如在分配和当前阶段之间检测到错误时，这样就不会浪费时间。

20226：唤醒所有等待消息到达这个队列的进程，然后返回 0 以示成功。

Real_msgrcv

20230：同 **real_msgsnd** 函数一样，**real_msgrcv** 函数实现 **msgrcv** 系统调用。**Msgtyp** 参数含义灵活，这可以从在 20248 行开始的标题注释之中看出。**Struct msg** 的 **msg_type** 域在这里发挥作用：在该函数中它要与 **msgtyp** 参数相比较。

另一个与 **real_msgsnd** 相同的地方是 **real_msgrcv** 函数也是从 20349 行的 **lock_kernel** /**unlock_kernel** 函数对内调用的。

20239：从 **msgid** 提取 **msgque** 下标并确保在那个下标所指的空间中有合法的一项。

20262：这个 **if/else** 语句对从队列中选择一个消息。第一种情况最简单：它只需要得到队列中的第一条消息，使得 **nmsg** 或者为 **NULL** 或者指向队列的第一个元素。

20266：**msgtyp** 为正值，并且 **msgflg** 里的 **MSG_EXCEPT** 位（15862 行）被设置。**real_msgrcv** 函数沿着队列搜索第一个类型和 **msgtyp** 不匹配的项。

20272：**msgtyp** 为正值，但是 **MSG_EXCEPT** 位未被设置。**real_msgrcv** 函数沿着队列搜索第一个类型和 **msgtyp** 匹配的项。

20279：**msgtyp** 是负值。**real_msgrcv** 函数用最小的 **msgtyp** 编号来搜索消息，如果最小值比 **msgtyp** 的绝对值还要小的话。注意 20281 行在比较时使用<而不是<=，这样队列中消息的选择就不再有利于第一个消息了。这样的结果不仅令人满意——尽量遵循 FIFO 方式是一个好的策略——而且效率也稍有提高，因为这种方式减轻了赋值工作。如果比较采用<=，每个连接（tie）都将意味一次赋值操作。

20287：此时，如果有消息满足给定的标准，**nmsg** 就指向它。否则，**nmsg** 就是 **NULL**。

20288：即使找到一个合适的消息，它也有可能不被返回。如果调用程序的缓冲没有足够大的空间来容纳整个消息体，调用者通常会得到 **E2BIG** 错误。然而，假如 **msgflg** 的 **MSG_NOERROR** 位（15860 行）被设置，那么这个错误就不会被公布。（我找不出什么理由可以让一个应用程序去设置 **MSG_NOERROR** 标志位，我也找不出任何一个使用它的应用程序。）

20292：如果 **msgsz** 指定了多于消息体的字节数，**real_msgrcv** 函数就把 **msgsz** 减少到消息的实际大小。当程序执行过这里之后，**msgsz** 就是应该被复制到调用者缓冲区的字节数。

虽然此处代码的更加传统的写法有时比较慢，不过平均起来还是要更快一些：

```
if ( msgsz > nmsg -> msg_ts )
  msgsz = nmsg -> msg_ts;
```

20294：把选中的消息从队列中移出。队列是一个单向链表，不是双向链表，所以当不是队列中第一个的消息被移出时，**real_msgrcv** 函数必须先要在队列中进行循环以寻找它的前趋队列节点。

通过将队列转换为双向链接，前趋节点就能在恒定时间里被找到。这个改变将引入空间损耗（需要额外的指针），时间损耗（用来更新附加的指针），以及复杂度的提高（需要增加完成这些工作的代码）。尽管如此，那些代价都是很小的，而在被移出的消息处于队列中部的情况下，它们可以显著地提高速度。

不过实际情况中，大部分应用程序从队列中移出的都是第一个消息。其结果是，额外花费在管理 **msg _prev** 指针（假定我们这样称呼它们）上的时间通常被完全的浪费了。只在从队列中间移出消息时它才会有所补偿，但应用程序又很少这样做。结论是为了提高特殊情形时的速度而降低了普遍情况下的效率——这几乎总是一个坏主意。甚至于确实要移出队列中间节点的应用程序也不会等很长时间，因为通常消息队列很短，典型情况下最多也就是几十个消息而已，而且平均在循环进行到一半时就能找到选择的消息了。

因此，只有当消息队列有成百上千条消息而且应用程序又要移出队列中间的节点时，应用程序才会经历一次明显的速度减慢过程。考虑到这种情况的罕见程度，开发者的决定就是正确的。除此而外，如果一个应用程序真的陷入这种困境，而且它的开发者又不顾一切的需要这额外一点点速度——那么好吧，这就是 Linux，他们可以自己修改内核源代码以满足需要。

20305：处理移去队列中唯一节点的情况。

20308：更新消息队列统计值。

20313：唤醒所有等待写入这个消息队列的进程——也就是所有被 **real_msgsnd** 函数设置为休眠状态的进程。

20314：把消息复制到用户空间并释放队列节点（头部和体部）。

20318：返回正被返回的消息的容量大小——这对可变长度消息来说至关重要，因为应用程序的消息格式可能无法说明消息在哪里结束。

20320：没有符合调用程序标准的消息。接下来发生的操作将取决于调用者：如果调用者设置 **msgflg** 的 **IPC_NOWAIT** 位，那么 **real_msgrcv** 函数可以立刻返回一个失败错误。

20323：否则，调用者宁愿在没有可用的消息时进入休眠状态。如果一个信号正等待调用进程则返回 **EINTR** 错误；否则，调用者进入休眠状态直到一个信号到达或者别的进程写队列为止。

20329：永远不会执行到这里，但是编译器并不知道这一点。所以，这儿有一个假 **return** 语句，只是为了满足 gcc 的要求而已。

Sys_msgget

20412：因为 **sys_msgget** 的流程控制比 **sys_msgsnd** 和 **sys_msgrcv** 的要简单，所以就没有必要把 **sys_msgget** 的所有实质操作转移到一个独立的辅助函数上。尽管它确实有自己的辅助函数，本章随后还将对它进行分析。

20414：跟踪函数所需的返回值的 **ret** 不必初始化成**-EPERM**。**Ret** 会在函数的每一分支路径上被赋值，所以这一行的赋值就是多余的。然而，gcc 的优化器的聪明程度足以发现并消除这种无效赋值，因此这一点是没有意义的。

20418：特殊键值 **IPC_PRIVATE**（未包括在内——它的值是 0）表明调用者需要一个新队列，无论是否有其它具有相同键值的消息队列存在。在这种情况下，通过使用 **newque**

（20370 行）能够立刻创建该队列，随后我们还将对 **newque** 进行详细讨论。

20420：否则，**key** 唯一地标识出了调用者需要使用的消息队列。一般地，开发员选择键值时或多或少带有随机性（或者给用户一种办法来选择一个）而且希望它不会与任何运行中的应用程序的键值发生冲突。

这可能听起来耸人听闻，但是临时文件名也存在同样的问题——你只能期望没有其它应用程序选择了同样的命名方式。实际上，很少出现问题——**key_t** 是 **int** 类型的 **typedef**，所以在 32 位机上有超过 40 亿个可能值，而在 64 位机上超过了 9×10^{18}[①]个！这个键值空间容量的巨大程度有助于降低偶然冲突的机率。而且对于消息队列键值，或者对于文件，即使偶然发生冲突，一个授权方案也能进一步减小问题发生的可能性。

即便如此，难道我们不能做得更好吗？像标准 C 库函数 **tmpnam** 一样的函数可以极大地帮助产生能够保证在系统范围内唯一的临时文件名，但是却没有类似的办法能够产生一个消息队列键值而且保证它的唯一性。

如果对这个问题进行进一步研究的话，这些因素看起来应该是两个不同的问题。应用程序大体上并不关心临时文件的名字是什么，只要它不是正在使用的文件就可以。但是应用程序一般需要提前知道应把消息发送到哪一个队列中。如果一个应用程序动态的选择它的消息队列键值，那么它有时又莫名其妙地需要把被选择的键值告诉其它应用程序。（等价的，它可以发送 **msgid** 来代替键值。）而且，假如被涉及的应用程序已经有办法来像那样进行彼此之间的发送消息，那么它们还要消息队列做什么？因此，这可能不是一个值得解决的问题。如果一个应用程序需要一个非专有（nonprivate）队列的唯一键值，但是它对实际键值是什么并不太关心，那么它就可以通过尝试键值 1 来得到一个（记住 0 就是 **IPC_PRIVATE**）并且可以从那儿逐步尝试直到成功为止——那只需少量的工作，尽管不太可能需要。

无论如何，这一行使用 **findkey**（20354 行，后边讨论）来查找拥有给定键值的存在着的一个消息队列。

20421：如果键值没有被使用，那么 **sys_msgget** 就可以创建它。如果 **IPC_CREAT** 位没有被设置，则将返回 **ENOENT** 错误；否则，**newque** 函数（20370 行）创建队列。

20425：键值已被使用。如果调用者把 **IPC_CREAT** 和 **IPC_EXCL** 位都设置了，那么在这种情况下调用者就希望能够产生一个错误，因此它就得到了一个。（这是为了故意与 **open** 的 **O_CREAT** 和 **O_EXCL** 位恰好能够并列。）

若不加考虑，很难分辨出如原文所写的 **if** 判断和下面的等价形式相比那个更快：

P526—1

两种判断方式都检查是否那两个标志位都被设置了，但是，出于种种原因，你可能会期望任意一个快于对方。然而，结果是 gcc 为这两者产生同样的代码，至少是在优化编译的时候。（如果你对此有兴趣的话，它所选择的方案是把我建议的替代品直接进行翻译的结果——它的转换方案在内核中看起来就好像是同时测试两个标志位的代码。）这是一个相当棒的优化过程，而且也是我过去所没有期望得到的。

20428：否则，使用该健值，调用者将接受具有那个健值的存在着的队列。（这是最普遍的情况。）如果在期望的地方没有消息队列（考虑到 **findkey** 的执行，那应该是决不会出现的情况）或者调用者缺少访问它的权限许可，那么将返回一个错误。

20434：序列编号和 **msgque** 下标被编码在返回值里。这将成为调用者要传递给 **sys_msgsnd**、**sys_msgrcv**，以及 **sys_msgctl** 的 **msgid** 参数。

---

① 原文是：”9 quintillion”，“quintillion” (美、法) 百万的三次方, (英、德)百万的五次方，用科学技术法表示应该为：9.e+18。

这种编码方案有两个重要特征。更明显的一个特征是如何把序列编号部分和数组下标部分分离开来：因为 **id** 是一个索引 **msgque** 的数组下标，它只能具有最大到（但不包括）**msgque** 中含有元素的数目，即 **MSGMNI** 的值。通过把这个值与序列编号相乘，就可以使低位空出来以保存 **id** 了——它很像是一种标准的 **MSGMNI** 算法。这里还需要注意的是返回值永远不会是负值——这一点是非常重要的，因为 C 库执行时可能会把负的返回值当作是一个错误。因为当前值是 128，所以数组下标占据返回值的低端 7 位。序列编号是 16 位，因此只有 **ret** 的低 23 位可以被这次赋值设置成 1，而且所有高位应是 0。特别地，符号位是 0，所以 **ret** 是 0 或正值。

20437：不管 **ret** 被如何计算，它现在都将返回。

Sys_msgctl

20468：**sys_msgctl** 函数无疑是消息队列实现中最大的一个函数。这部分上是因为它要完成许多不同的功能——类似于 **ioctl** 函数，它是一个功能联系松散的函数聚合体。（顺便要说明的是，不要因为此处的混乱而责备 Linux 的开发者们；他们只是想要提供与 System V 那蹩脚的设计相一致的兼容性。）

msqid 参数指定了一个消息队列，**cmd** 指出 **sys_msgctl** 函数应该对它如何操作。很快读者就会看到，需不需要 **buf** 取决于 **cmd**，而且即使当它被使用时它的含义也将随情况的不同而不同。

20477：拒绝明显非法的参数。在不经常出现的，参数无效情况已经是不容置疑时，在调用 **lock_kernel** 函数之前执行本操作能够挽救不必要的内核锁定。（当然，流程控制将不得不相应作出调整——必须跳过 **lock_kernel** 函数）

20481：在 **IPC_INFO** 和 **MSG_INFO** 情况中，调用者需要有关消息队列实现的属性信息。它可能要用这些信息来选择消息容量，比如说，在最大消息容量较大的机器上，调用进程可以提高它自己在每个消息中发送的信息量界限。

所有清晰地在消息队列实现中定义那些缺省界限的常数都是通过 **struct msginfo**（15888 行）对象复制回来的。假如 **cmd** 是 **MSG_INFO** 而不是 **IPC_INFO** 时，还要包括一些额外信息，读者可以在 20495 行看到这一点，不过这两种情况在其它方面是相同的。

注意一下调用程序的缓存 **buf**，它被定义成了指向一种不同类型 **struct msqid_ds** 的指针。不过没有关系。复制是由 **copy_to_user** 函数（13735 行）完成的，它并不关心它的参数的类型，尽管当被要求向一块不可访问的内存写入时该函数也会产生错误。如果调用者提供了一个指向一块足够大空间的指针，**sys_msgctl** 函数将把请求的数据复制到那里；使得类型（或至少是容量）正确是取决于调用程序的。

20505：如果复制成功，**sys_msgctl** 函数返回一个附加的信息段，即 **max_msqid**。注意这种情况完全忽略了 **msqid** 参数。这样做有重要的意义，因为它返回了有关消息队列执行情况的总体信息，而不是某个特别的消息队列的具体信息。不过，就这种情况下是否应该拒绝负的 **msqid** 值仍是一个各人看法不同的问题。不可否认的是，即使没有使用 **msqid** 时也拒绝一个无效的 **msqid** 值一定能够简化代码。

20508：**MSG_STAT** 请求内核对给定消息队列持续作出的统计性信息——它的当前和最大容量、它的最近的读者和写者的 PID，等等。

20512：如果 **msqid** 参数不合法，在给定位指处没有队列存在，或者调用者缺少访问该队列的许可，则返回一个错误。因此，队列上的读许可不仅意味着是对入队消息的读许可，而且也是对关于队列本身“元数据（metadata）”的读许可。

顺便提及一下，要注意命令 **MSG_STAT** 假定 **msqid** 只是 **msgque** 下标，并不包括

序列编号。

20521：调用者通过了测试。**sys_msgctl** 函数把请求的信息复制到一个临时变量中，然后再把临时变量复制回调用者的缓存。

20533：返回“完全的”标识符——序列编号现在已经被编码在其中了（在 20520 行完成）。

20535：还剩下三种情况：**IPC_SET**、**IPC_STAT**，和 **IPC_RMID**。与读者迄今为止所见的那些情况有所不同的是，那些情况都在 **switch** 语句里被完全的处理了，而剩余的这三种在此仅进行部分处理。第一种情况，**IPC_SET** 只要确保用户提供的缓冲区非空，就将它复制到 **tbuf** 里以便后面函数的进一步处理。（注意拷贝操作之后在 20540 行对 **err** 的赋值是不必要的——因为在它使用之前的 20550 行，**err** 将被再次赋值。）

20542：剩余三种情况中的第二种，**IPC_STAT** 仅仅执行一次健全性检查——它的真正工作还在后边的函数体中。最后一种情形，**IPC_RMID** 在这个语句中不工作；它所有的工作都推迟到后边的函数中完成。

20548：这段代码对所有剩余的情况都是共同的，而且大家现在都应该对它比较熟悉了：它从 **msqid** 里提取出数组下标，确保在指定的下标处存在着一个有效的消息队列，并验证序列编号的合法性。

20559：处理 **IPC_STAT** 命令的剩余部分。假如用户有从队列中读出的许可，**sys_msgctl** 函数就把统计信息复制进调用者的缓冲区里。如果你认为这与先前 **MSG_STAT** 的情形非常类似，那你就是对的。这两者之间的唯一不同之处在于：正如读者所见，**MSG_STAT** 期望一个“不完全”的 **msqid**，而 **IPC_STAT** 却期望一个“完全”的 **msqid**（就是说包括序列编号）。

20572：复制统计数据到用户空间。如果按照如下方式重写这三行代码，那么运行速度或许稍快一些：

P527—1

毕竟，对于写入用户空间来说成功要肯定比失败更为普遍。基于同样的原因，**MSG_STAT** 情况下（始于 20530 行）的相应的代码如果被重写成以下形式也可能更快：

P527—2

或者，下边的一个甚至可能更快，因为没有一次多余的赋值操作：

P528—1

然而和直觉相反的是，我对所有这三种修改都作了测试，结果却发现是内核的版本执行起来更快。这必然与 gcc 生成目标代码的方式有关：显然，我的版本中的一条额外跳转要比内核版本的额外赋值所花费的代价高得多。（从 C 源代码来考虑额外的跳转并不直观——你不得不考察 gcc 的汇编输出代码。）回想前边章节所讨论过的，跳转会带来明显的性能损失，这是因为它们会使得 CPU 丧失其内在的并行性所带来的好处。CPU 的设计者们竭尽全力要避免分支造成的性能损失影响，不过很明显，他们并不总是成功的。

最终，对 gcc 优化器的进一步改善可能消除内核版本和我的代码之间的差别。每当两种形式逻辑相同而一个较快时，假如 gcc 能够发现这种等价并为两者生成同样的代码，那将非常令人愉快。不过这个问题是要比看上去难得多的。为了生成最快的代码，gcc 将需要能够猜测哪一次赋值最易发生——另一种情况则涉及了分支。（对 gcc 的最近版本所作的工作已为这样的改进打下了基础。）

20576：在 **IPC_SET** 情形里，调用者需要设置消息队列的某些参数：它的最大容量、属主，和模式（mode）。

20578：为了操纵消息队列的参数，调用者必须拥有该队列或者拥有 **CAP_SYS_ADMIN** 权

能（14092 行）。权能已在第 7 章中讨论过。

20584：把消息队列中最大字节数的界限提高到正常限制以上，这就类似于提高任何其它资源的硬界限一样，因此它也需要与之相同的权能，即 **CAP_SYS_RESOURCE**（14117 行）。资源限制在第 7 章已经讨论过。

20587：调用者应该被允许执行该操作，所以被选择的参数根据调用者提供的 **tbuf** 被设置。

20595：**IPC_RMID** 意味着删除特定的队列——不是队列中的消息，而是队列本身。假如调用者拥有该队列或者有 **CAP_SYS_ADMIN** 权能，这个队列就可以用 **freeque** 函数调用（20440 行）来释放。

20605：**cmd** 最终不是经过验证的命令中的一条，所以调用程序得到 **EINVAL** 错误。在这种情况下，在 20548 行所作的工作原本是可以避免的。假设我们要试图尽早检测无效的 **cmd**，通过删除 **switch** 语句里的 **default** 情况并把下列代码附加到函数第 20546 行的第一个 **switch** 后：

P528—2

这样就会改变函数的行为状态。当调用者提供了一个无效 **cmd** 和一个无效 **msqid** 时，它将得到一个与现在所得的不同的错误——有了这种改变之后，无效的 **cmd** 将先于无效的 **msqid** 而被检查出来。虽然有关 **msgctl** 的文档并没有诺许任何一种行为，但是这样我们就可以自由的来改变它。其结果能够少许提高这种无效 **cmd** 情形下的速度。

然而，要注意这种解决方案很不幸地需要在第一个 **switch** 开关处引入一个空的 **IPC_RMID case**。没有它，函数将错误的把 **IPC_RMID** 也当作一种无效 **cmd** 情况而抛弃掉。这个额外的 **case** 减缓了 **cmd** 合法这种正常条件下的速度——虽然不很严重，但情况的确如此。而且，正如你所知道的，用普遍情形的代价来换取特殊情形时速度的提高从来就不是一个良好的解决办法。因此还是原来的方式更好。

Findkey

20354：**findkey** 函数为 **sys_msgget** 系统调用（调用在第 20420 行）定位具有给定键值的消息队列。

20359：开始对 **msgque** 里所有可能被占据的单元槽进行循环。**max_msqid** 跟踪 **msgque** 里被占据的最大数组元素；在这里使用到了它，并且在很快就要提到的 **newque** 和 **freeque** 里将对它进行维护。若没有 **max_msqid**，这个循环将需要在 **msgque** 的所有 **MSGMNI** 个元素里反复进行，就算是只有前 5 个在使用也要如此。

20360：如果当前数组元素值是 **IPC_NOID**，那么就会在那里创建一个消息队列。这个消息队列可能具有正被搜寻的键值，所以 **findkey** 函数将等待该队列的创建工作完成。（当 20385 行的 **kmalloc** 调用使进程休眠时就会进入这种状态。）

20362：如果该 **msgque** 的项目是未被使用的，那么它明显不具有匹配的键值。

20364：若匹配的键值被找到，相应的数组下标就被返回。

20367：如果循环结束仍未找到匹配的键值，就返回-1 以示失败。

Newque

20370：**newque** 函数定位一个没有使用的 **msgque** 项目，并尝试在那里创建一个新的消息队列。

20376：循环 **msgque** 以查找未用的一项。如果找到了一项，就用 **IPC_NOID** 来标记它，控制随之跳转到 20383 行的 **found** 标记处。

20381：如果循环结束却没有发现未用的项目，**msgque** 就是满的。**Newque** 返回 **ENOSPC**

错误表示表里没有剩余的空间。

20384：分配一个 **struct msqid_ds** 来代表新的队列。

20387：如果分配失败，该 **msgque** 项目被设置回 **IPC_UNUSED** 标志。

20388：一旦发现有 **IPC_NOID** 就激活任何已经休眠的 **findkey**。

20391：初始化新队列。

20404：如果这个队列被建立在 **msgque** 中原来最高的已用单元槽之后，**newque** 就相应的增加 **max_msqid**。

20406：在 **msgque** 里建立新队列。

20408：唤醒每个可能一直在等待该队列初始化完成的 **findkey**。

20409：返回序列编号和 **msgque** 的数组下标。（创建一组宏来处理此处的编码和随后的解码不会有什么损害。）奇怪的是，没有在这里增加序列编号——它要由接下来讨论的 **freeque** 来完成。如果读者考虑一下，这里的决定是有一定道理的。你并不需要每个队列都有一个唯一的序列编号——你只是想让每次 **msgque** 元素被重用时有一个不同的序列编号，以便数组下标和序列编号二者的组合（combination）不可能重复而已。数组下标直到建立在该位置的队列被释放后才能重新使用，所以增加序列编号的操作也可以推迟到那个时候。

为了把这个含义说的更明确一些，一个序列编号是可以被两个 **msgque** 元素同时使用的。

Freeque

20440：我们将以 **freeque** 函数来结束这次内核消息队列实现的讨论，它的作用是删除一个队列并释放相应的 **msgque** 元素项。

20449：如果正在被释放的是最高的被使用项，**freeque** 函数将尽可能地减低 **max_msqid**。循环之后，**max_msqid** 将再次成为被使用的 **msgque** 项的最高下标值，或者在所有元素项都没有使用时变成 0。要注意的是如果 **max_msqid** 是 0，则 **msgque** 要么是空，要么就只有一个元素项。

20452：**msgque** 数组的元素被标识成为未使用，尽管此时 **struct msqid_ds** 还没有被释放（在 **msq** 里，**freeque** 函数仍然有一个指向该 **struct msqid_ds** 的指针）。

20454：假如有某个进程正等待读出或写入这个队列，必须警告它们该队列即将消失。这里的循环唤醒所有那些进程。每个正等着向该队列发送消息的进程将在第 20171 行知道被改变了的序列编号；每个等待从该队列里读取消息的进程也将在第 20254 行进行同样的工作。

20458：调用 **schedule** 函数（26686 行，在第 7 章讨论过）来赋予被唤醒了的进程运行的机会。有趣的是，被唤醒了的进程可能还没有得到 CPU 使用权——当前进程仍然有最大的优先权。假如这种情况发生，新近被唤醒的进程将不会从各自的等待队列中被移出；而 **freeque** 又会注意到这一点并继续重复以图再次唤醒进程。最终，执行 **freeque** 的进程会因耗尽它的时间片而将（CPU）让出给其它进程。在考虑了这一切之后，在调用之前明确设置当前进程的 **SCHED_YIELD** 标志（16202 行）可能是更好的方法，这样可以给其它进程更好的使用 CPU 的机会。

20460：没有被挂起的读者和写者，所以该队列和它的消息可以被安全的释放掉。

信号量

信号量（Semaphores）是一种对资源访问进行保护的方式。信号量在通常概念上的模型是指一种发送信号的标志（名称由此而来），但是我认为更好的象征是一把钥匙（key）。不要把它与我们已经讲过的整数类型的键值（key）搞混了——在这个类比中，我所指的意思是你的前门钥匙。

在最简单的情况下，信号量只是悬挂在一扇锁着的门旁吊钩上单独的一把钥匙。为了穿过这道门，你必须把钥匙从吊钩上拿下来；当你出来时再把钥匙重新放回吊钩之上。如果你到达时钥匙不再那里，你就不得不等待它的拥有者把它放回原处——假如你已决定要通过这道门，就必须这样。而作为另一种选择，如果无法立刻得到钥匙，你也可以因没有耐心等待而就此放弃。

上边描述了某资源每次只能由一个实体（entity）来使用的情形；在这种只有一把钥匙的情况下，信号量可以被看作是一个二元信号量（binary semaphore）。对于每次可以被多个实体占用的资源而言，信号量可被看作是计数信号量（counted semaphores）。这与前边一样，只不过是吊钩上悬挂了更多的钥匙而已。如果资源同时可供四个用户使用（或者假如有四个等价的可用资源，它们基本上是相同的），那么就有四把钥匙。依次可自然的进行类推。

进程使用信号量来协调它们的动作。比如，假设你正在写一个程序而且想保证每次在给定的机器上最多只有该程序的一个实例可运行。这方面的一个好例子是声音文件播放器——可能你不会想让它同时播放多个声音文件，因为其结果将是令人烦恼的一团糟。另一个例子是 X 服务器。当然偶尔也会有充分的理由使得在同一个机器上同时运行多个 X 服务器，但是对于一个 X 服务器来说禁止这样做也是很合理的，至少缺省的做法就是如此。

信号量提供了一种解决这个问题的方案。你的音响播放器、或 X 服务器，或是其它任何程序都可以定义一个信号量，检查该信号量是否在使用，若没有使用则继续运行。如果该信号量已被使用，则表明程序的另一个实例在运行之中——你的程序可以等待信号量被释放（音响播放器可能的行为），只是放弃并退出（X 服务器可能的行为），或者暂时继续其它工作稍候再试信号量。顺便说一句，这样一种信号量的用法由于显而易见的原因而通常被叫做相互排斥（mutual exclusion）；它的通用简称，互斥（mutex）将在内核源代码中反复出现。

锁文件是获得与二元信号量同样效果的更为普遍的一种方式，至少在某种情况下如此。锁文件更易使用，而且锁文件的一些实现可工作在网络上；但是信号量则不行。另一方面，锁文件在超出二元的情况时就不容易使用并推广了。但无论如何，锁文件都超出了本书的范围。

信号量和消息队列二者的代码是如此相似以至于没有必要再讨论 **sem_init**（20695 行）、**findkey**（20706 行）、**sys_semget**（20770 行）、**newary**（20722 行），以及 **freeary**（20978 行）了，因为它们几乎同它们所对应的消息队列部分是一样的。

Struct sem

16983：**struct sem** 结构体代表一个单独的信号量。它有两个成员：

- **semval**——如果是 0 或为正值，**semval** +1 就是仍然挂在这个信号量吊钩上的钥匙数目。若为负值，它的绝对值就比正等待访问它的进程数目大一。缺省的信号量是二元的，但是它们也可以通过使用 **sys_semctl** 变为计数型的；信号量的最大值是 **SEMVMX**（在 16971 行定义为 32767）。
- **Sempid**——存储最后一个操作该信号量的进程的 PID。

Struct semid_ds

16927：**struct semid_ds** 与 **struct msqid_ds** 相对应：它跟踪所有关于单独一个信号量以及在它上面所执行的一系列操作的信息。我们所感兴趣的、有别于 **struct msqid_ds** 中的成员如下所述：

- **sem_base**——指向一个 **struct sem** 数组——换句话说，指向一个信号量数组。如同单独一个 **struct msqid_ds** 可以包含多个消息一样，一个 **struct semid_ds** 也可以包含多个信号量——该数组中信号量总和被具有代表性地称为一个信号量集合（semaphore set）。然而与消息队列不同的是，被一个 **struct semid_ds** 所跟踪的信号量的数目并不在它的生存期里变化。数组的容量大小是固定的。这些数组中一个的最大长度是 **SEMMSL**，它在第 16968 行被定义为 32。数组的实际长度记录在 **struct semid_ds** 的 **sem_nsems** 成员中。
- **sem_pending**——跟踪挂起的信号量操作组的一个队列。信号量操作一有可能就立刻完成，正如读者所预期的那样，所以只有当操作必须等待时，这个队列才会增加节点。此成员与 **struct msqid_ds** 的 **rwit** 和 **wwait** 成员大致等价。
- **sem_pending_last**——跟踪上述同一队列的队尾。它并不直接指向最后一个节点——它指向一个指向最后节点的指针，这将有利于稍微提高后面代码的速度（尽管这给理解增加了难度）。（需要顺便提一下的是，我不知道为什么同样的思想没有被应用于消息队列。）
- **sem_undo**——当各个进程退出时所应该执行操作的一个队列。这将在后续章节中讨论。

Struct sem_queue

16989：**struct sem_queue** 结构体是单个 **struct semid_ds** 之上休眠着的操作队列中的一个节点。它有如下成员：

- **next** 和 **prev**——队列中的下一个和前一个节点。正如 **sem_pending_last** 一样，**prev** 是指向一个指向前面节点的指针的指针。读者将能够在本章后面章节中看到为什么系统要这样做的原因。**prev** 永远不会变成 **NULL**；在退化的情况里，即当队列为空时，**prev** 指向 **next**。
- **sleeper**——当某进程必须等待完成一个信号量操作时使用的等待队列。等待队列在第 2 章中介绍过。
- **undo**——一个将要撤销由 **sops** 所暗示的操作的操作数组——用另一种方式表示它就是 **sops** 的反转。
- **pid**——尝试完成这个队列节点操作的进程的 PID。
- **status**——记录一个休眠进程被唤醒的过程。
- **sma**——向后指向这个结构体 **struct** 所存在的 **sem_pending** 队列的 **struct semid_ds**。
- **sops**——指向这个队列节点所代表的一个或多个操作的一个数组；它永远不为 **NULL**。这个队列节点所描述的工作目的是执行 **sops** 里所有的操作。
- **nsops**——**sops** 数组的长度。
- **alter**——说明是否操作会影响信号量集合里的任何一个信号量。这个问题的回答看起来总是肯定的，但是要记住等待信号量变成 0（即成为可用）并不会影响信号量本身。

Struct sembuf

16939：**struct sembuf** 结构体表示在信号量上执行的单个操作。它有如下成员：

- **sem_num**——是 **struct semid_ds** 的 **sem_base** 数组的数组下标，该数组由这种操作适用的信号量构成。因为 **struct sembuf** 是 **struct sem_queue** 的一部分，而且 **struct sem_queue** 知道它与哪一个 **struct semid_ds** 相关联，这样就从不会出现该操作应使用哪一个 **struct semid_ds** 的信号量数组的疑惑了。在其它情况下，一个 **struct sembuf** 数组与一个索引 **semary** 的下标组成一对，这也蕴含了一个信号量数组。
- **sem_op**——要执行的信号量操作。通常，它的值是-1、0，或 1：-1 表示获得（procure）信号量（从吊钩上取走钥匙），1 表示交出（vacate）信号量（把钥匙重新放回到吊钩上），而 0 表示等待该信号量变成 0。除了这些值以外的值也是有用的，不过它们只是被翻译为获得或交出更多的信号量值而已——也就是说，取走或放回吊钩上更多的钥匙。（这段文字里的“获得”和“交出”可能看起来有点儿怪——无须担心；这只是普通的信号量术语。）
- **sem_flg**——可以修改操作执行方法的一个或多个标志位（在这个 **short** 里的每一个位）。

这些数据结构之间的关系如图 9.2 所示。

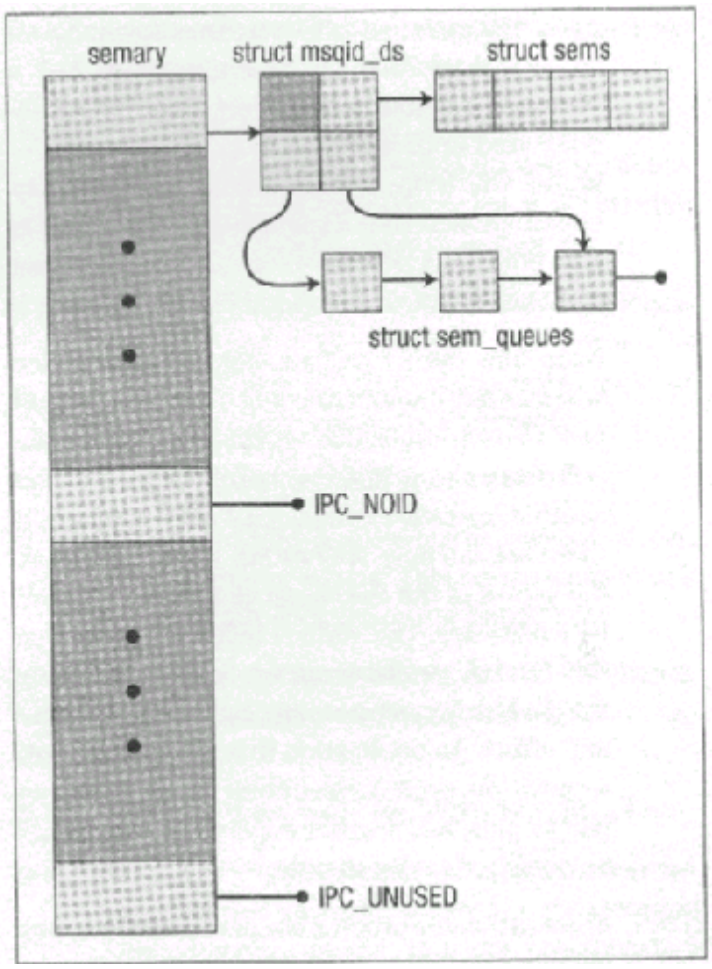

图 9.2 信号量数据结构

Struct sem_undo

17014：**struct sem_undo** 含有足够的撤销单个信号量的操作的信息。当一个进程执行信号量操作同时设置了 **SEM_UNDO** 标志位时就创建一个用来撤销该操作的 **struct sem_undo**。进程的 **struct sem_undo** 列表所包含的所有撤销操作在该进程退出时都会执行。熟悉设计模式（design patterns）的读者可能发现这是命令模式（Command pattern）的一个实例。

这个特性保证不管进程如何退出，都将自动为它执行相应的清理工作——这样一来，就不会意外的让其它进程空等一个永远也不会被释放的信号量了。（除非进程获得信号量后陷入死循环之中，但是避免这个问题不是内核的工作——在这种情况下，目的是要提供给进程正确的工作方法，而不是对其进行人工干预。）

struct sem_undo 有如下成员：

- **proc_next**——指向固有进程 **struct sem_undo** 列表里的下一个 **struct sem_undo**。
- **id_next**——指向与信号量集合相关联的 **struct sem_undo** 列表里的下一个 **struct sem_undo**。你所看到的是正确的，同一个 **struct sem_undo** 确实是同时存在两个不同的列表之中。读者将在本章的后面看到这两者的作用。
- **semid**——标识出这个 **struct sem_undo** 所归属的 **semary** 元素项。
- **semadj**——一个调整器的数组，这些调节将使用在和这个 **struct sem_undo** 相关联的信号量集合中的每个信号量上。这种结构所不知道的信号量在数组中有一个 0——并没有进行调整。

Sys_semop

21244：**sys_semop** 函数实现了 **semop** 系统调用。消息队列代码中没有直接同 **sys_semop** 函数对等的函数——它是 **sys_msgsnd**、**sys_msgrcv**，或者同时是两者的对应函数，这取决于你如何看待它。在任意一种情况下，它的工作都是在一个或多个信号量上完成一种或多种操作。它将自动尝试完成所有操作（即无需中断）。假如无法全部完成，它将不会执行其中的任何一项操作。

21254：同消息队列函数非常类似，这像是在比必要的时机稍微提前一些的时候就锁住内核。加锁也应该可以被推迟到第 21265 行左右再执行。

21255：参数的健全性检查。特别注意 **nsops** 受到 **SEMOPM** 的限制，它是可以被立刻尝试的信号量操作的最大数目。在第 16970 行它被定义为 32。

21261：把请求的操作描述从用户空间复制到一个临时缓冲区，即 **sops** 中。

21265：确保在指定的数组位置存在一项。正如读者所见，同消息队列代码的 **msgque** 对等的是 **semary**（20688 行）。还要注意的是数组下标和序列编号以与消息队列代码相同的方式被打包进了 **semid** 参数。当然这里应用的常量稍有不同——**SEMMNI** 在第 16967 行定义为 128（而巧合的是，**MSGMNI** 也是一样的值）。

21272：开始一个遍历所有特定操作的循环。首先检查在操作中给出的信号量数目是否超出范围，如果是的话就放弃它。但是令人奇怪的是，这里返回的失败信息是 **EFBIG** 错误（意思是“文件太大”）而不是 **EINVAL** 错误（“非法参数”）。尽管这也是符合文档规范的。

21275：记录设置了 **SEM_UNDO** 标志位的操作的数目。**undos** 只是一个标志——重要的是它是否为 0——因此，当条件满足时给它赋值 1（或任何非零值）将产生同样的效果。不过，内核的版本更快一点。而且因为循环重复的循环次数最多是 **SEMOPM** 次，

undos 就不可能被增加多次以至于回到原点再次变为 0。

21277：接下来的几个测试更新两个局部标志：**decrease** 和 **alter**。它们分别用来跟踪集合里的任何操作是否在减少某个信号量的值以及是否在修改一个信号量的值。直到循环结束之后，**alter** 才会在第 21282 行被计算出来——在循环里，它只是跟踪是否有操作增加信号量的值；这个结果与后边 **decrease** 里的信息结合起来最终决定是否发生了改变。

要注意这里的代码没有检查是否组合在一起的操作将彼此抵消——可能一个操作把某个信号量减 1，而另一个操作又会把它加 1。如果这仅有一个操作，那么从某种意义上讲，**decrease** 和 **alter** 标志的值将是很容易引起误解的。内核可以尝试着优化这种情况（并得到实现同样内容的更加精致的版本），不过相比较于所花费的时间和精力，这可能并不值得：一个愚蠢到执行这样一种奇怪得空操作的应用程序就应该这么慢，而一个聪明的应用程序则不应该这么愚蠢。

21285：确保进程具有在信号量上执行特定操作的许可。如果 **alter** 为真，那么修改信号量的进程就需要写许可；否则，它只是在等待一个或多个信号量的值变为 0，这样该进程就只需要读许可。

21291：包括某些撤销操作的一组操作。如果当前进程已经有了在退出时要在这个信号量上执行的一组撤销操作，那么新的撤销操作的数据就应该合并到那一组中去。这个循环查找存在的撤销操作集合，假如有，就使 **un** 指向它，若没有，则 **un** 为 **NULL**。

21295：进程还没有这个信号量集合的一个取消集（undo set），所以需要为它分配一个新的。在读者已经在消息队列代码中熟悉了的一段编码的实践经验之后，为撤销调节（**semadj** 数组）分配的空间将被安排在紧靠 **struct sem_undo** 本身之后，并作为同一分配的一部分。接着就填入 **struct sem_undo**。

21311：在提供的操作集合里没有撤销操作，所以 **un** 被设置为 **NULL**。

21313：调用 **try_atomic_semop**（20838 行，后边讨论）来尝试在单个的槽内执行所有操作。如果有任何引起变化的操作，**un** 即为非空；若失败，就需要利用它来在函数返回之前取消任何已经完成的部分操作。

21315：**try_atomic_semop** 返回 0 表示成功，负值表示错误。无论何种情况，控制流程都向前跳转到第 21359 行。

21321：否则，**try_atomic_semop** 返回一个正值。这表示此刻无法执行所有操作，但是该进程希望等待且稍后再试。一个局部 **struct sem_queue** 将首先被填写。

21328：代表修改信号量操作的节点被放在队列的末尾；代表等待信号量值归 0 的操作的节点位于队列的前边。在本章后边探究 **update_queue** 函数时（20900 行）读者将对这种做法的原因有所了解。

注意在挂起操作的队列中放置了一个局部变量——这很不寻常；这样的数据结构通常具有以堆形式分配（heap-allocated）的节点。在这种情况下这样做是安全的，因为节点在函数返回之前将被从队列中移出；而上下文转换部分将负责剩下的工作。或者进程也可先退出，此时由 **sem_exit**（21379 行）来负责进行收尾工作。

21333：开始一个反复尝试执行这些操作的循环，仅当所有要求的操作都成功完成或者发生一个错误时该循环才会退出。

21336：一直休眠到被一个信号中断或有某断点（point）被再次尝试为止。

21342：如果进程由于此刻具有成功的机会而被 **update_queue** 唤醒，则它重新尝试进行该操作。

21358：把这个进程从等待修改信号量集合的进程队列中移出。

21360：假如这个操作的集合改变了队列，那么某个其它进程所等待的条件可能就已经具备。

Sys_semop 调用 **update_queue** 来寻找并唤醒这样的进程。

Sys_semctl

21013：实现 **semctl** 系统调用的 **sys_semctl** 函数具有与 **sys_msgctl** 相似的许多共同之处。相应的，这里的讨论只涉及那些感兴趣的不同点，比如在 **sys_msgctl** 里没有对应部分的 **sys_semctl** 命令（command）。

21093：**GETVAL**、**GETPID**、**GETNCNT**、**GETZCNT**，以及 **SETVAL** 命令对单个信号量、而不是信号量集合进行操作，所以在这些情形里提供的 **semnum** 参数必须首先进行范围检查。若 **semnum** 在范围之内，**curr** 就指向相应的信号量。

21115：几乎是同样的命令集——**GETVAL**、**GETPID**、**GETNCNT**，以及 **GETZCNT**——涉及了对关于信号量的一段信息进行的阅读和计算。这里就完成那些工作。注意 **sempid** 成员的高位在第 21116 行被屏蔽掉了——通过后面的讨论你将知道这样做的原因。

21121：**GETALL** 命令请求这个信号量集合里所有信号量的值。和许多其它命令一样，此命令的工作并非在一处全部完成；稍后读者将见到其余的命令。

21126：**SETVAL** 命令把信号量的值设置成给定的值——当然，是在规定的限制内。同样地，此时只完成部分工作——主要是范围检查。

21142：**SETALL** 是 **SETVAL** 的一个普遍化结果；它设置集合中所有的信号量值。同 **SETVAL** 类似，在此只完成诸如范围检查一类的准备工作。

21173：**GETALL** 的剩余部分由此开始。

21175：确保进程有读取信号量值的许可。这里的许可检查与第 21112 行的相重复。

21177：把信号量值复制到局部数组 **sem_io** 里，然后再从那里将它们复制到用户空间。

21183：**SETVAL** 的剩余工作由此开始。

21187：因为信号量取得新值，所以任何有记录的为 **semnum** 信号量所进行的取消调节操作都将变为无效。这个循环通过把它们设置成 0 以使它们失去作用。

21189：把信号量的值设置成调用者提供的值，并调用 **update_queue**（20900 行）来唤醒那些等待该条件成立的进程。

21220：**SETALL** 的主要部分由此开始。

21224：所有信号量的值都被设置成了调用者提供的值。

21226：与集合内各个信号量相关的所有取消调节操作都被设置成 0。当信号量被设置为它已经拥有的值时，这并没有什么特别的地方——它也不应该有什么特别之处。如果调用程序需要为除了一个信号量之外的所有信号量都赋予新值，那么就不能通过设置那个信号量为原值的方法来欺骗它。取而代之的做法是，必须要对不应改变其值以外的集合中所有信号量施用 **SETVAL** 命令。

Sem_exit

21379：**sem_exit** 函数在消息队列代码里没有对应函数。它实现进程在退出时所要求自动执行的撤销操作。所以，它在进程退出时调用（23285 行）。

21389：如果进程的 **semsleeping** 成员非空，那么以下二者必有其一成立：要么进程正在某个 **sem_queue** 队列上处于休眠状态，要么它已经从该队列被移出但 **semsleeping** 还未更新。假如是前者，进程将被从休眠队列里移出。

21395：开始遍历当前进程的 **struct sem_undo** 列表。轮流对每个条目进行分析然后在循环的更新部分释放它们。

21397：如果对应于这个撤销结构体的信号量已被释放，就继续循环。**struct semundo** 的 **semid**

域可以被 **freeary** 设置成-1，本章随后将对其进行介绍。

21399：类似地，如果相应的 **semque** 项不再有效，则继续循环。

21406：与从消息队列中间移出一条消息的情形相当类似，这个循环遍历 **sma** 的 **struct semundos** 列表以找寻将被移出的前一个节点。当找到时，**sem_exit** 向前跳转到第 21413 行的 **found** 标记处。

21411：如果在 **sma** 的列表里没有找到撤销结构体，那么就发生了错误。**sem_exit** 显示一条警告消息并停止外层循环。这种反应看来有点过激，因为可能会有更多的撤销结构体能够依照这种处理方式进行释放。不应该因一个烂苹果就糟踏整整一桶苹果。尽管这几乎是“不可能发生”的情形，仅当内核逻辑错误才会导致其发生。我的推测是这样的，若检测到这样一个错误的话，剩下的数据就不再可信了。

21414：在 **sma** 的列表里找到了撤销结构体，**unp** 就指向一个指向其前驱的指针。接着把 **un** 从队列里移出。

21417：执行这个撤销结构体里对所有信号量的调节。

21427：像往常一样，调用 **update_queue** 以免被这个函数所执行的操作满足了唤醒某个休眠进程的条件。

21429：所有的 **struct sem_undo** 都已经处理过了——或者在 21412 行就检测到了错误并结束了循环。不管哪一种结果，当前进程的队列被设置成为 **NULL** 然后函数返回。

Append_to_queue

20805：把 **q** 附加在 **sma** 的 **sem_pending** 队列之后。这里的实现很紧凑；通常类似如下的形式：

P534_1

真正的优点在于内核的实现方式避免了潜在的代价昂贵的分支。通过使得 **sem_pending_last** 成为指向一个指向队列节点的指针的指针，而不仅仅是一个指向队列节点的指针，可能会部分的提高执行的效率。

Prepend_to_queue

20812：把 **q** 附加在 **sma** 的 **sem_pending** 队列之前。由于 **sem_pending** 不是一个指针的指针，这种实现就同前面考虑过的简单的实现一样具有相同的形式。

Remove_from_queue

20823：这是 **struct sem_queue** 队列上的最后一个原语操作，它把一个节点从队列中移出。

20826：通过修改前一队列节点的 **next** 指针，部分解除 **q** 与队列的链接。

20828：如果有下一个节点，还要更新下一个节点的 **prev** 指针；或者假如这已经是队列的最后一个节点，就使用 **sma->sem_pending_last**。要注意的是没有非常清楚的代码被用于移出队列中唯一的节点——假设你还没有发现原因的话，这就值得你花些时间研究一下为什么这种情形下代码也可工作。

20831：把已移出节点的 **prev** 指针设置成 **NULL**，以便第 21350 和 21390 行代码能有效地发现该节点是否仍在队列之内。

Try_atomic_semop

20838：该函数上方的标题注释说明它被用于测试是否给定的操作集能被全部执行。该注释没有说明这些操作是否能够被全部被执行，通常情况下它们是能够被执行的。

20846：开始循环所有通过检查的操作并依次对其进行尝试执行。

20850：**sem_op** 为 0 表示调用程序希望等待 **curr->semval** 变为 0。因此，如果 **curr->semval** 不是 0，调用程序不得不阻塞（block），这意味着操作无法自动被执行（由于这个进程被阻塞时要完成其它工作）。

20853：调用程序的 PID 被暂时保存在 **curr->sempid** 的低 16 位里；从前的 PID 现在被移进高 16 位。

20854：**curr->semval** 依照 **sem_op** 所要求的进行调整——还是临时性的。虽然该操作的结果的范围在随后的代码段中进行了核查，但是不管是在这里还是在其调用者中 **sem_op** 都未进行范围检查。由于 **sem_op** 的值过大或者过小都将造成 **semval** 回绕（wrap around），这样将导致意想不到的结果。

20855：如果这条操作的 **SEM_UNDO** 标志位被设置了，就表示在进程退出时该操作应当被自动取消，相应的撤销结构体会被更新。要注意这里假定 **un** 是非空的——确保这一点是调用程序的责任。

20858：对新的 **semval** 进行范围检查。

20864：循环即将完成，所有操作都将成功完成。如果调用程序只想知道操作能否成功，但此刻并不想执行它们，这些操作就可以马上被取消。否则，操作已经被执行了，所以 **try_atomic_semop** 就继续执行下去并返回成功。

20874：当一个操作把 **semval** 增加得过大时，跳到 **out_of_range** 标记处。函数安排返回 **ERANGE** 错误，并向前跳转到撤销代码。

20878：当进程因为它必须等待信号量归 0 或者操作不能立刻获得信号量而不得不等待信号量时，程序将跳至 **would_block** 标记处。如果这种情形之下调用程序不愿等待，就返回 **EAGAIN** 错误。否则，函数返回 1 表示调用者将需要休眠。

20884：在 **undo** 标记之后的代码取消所有从第 20846 行开始 **for** 循环里所作的工作。

20888：这一行代码显而易见的部分是用来把在第 20853 行暂存的值保存在 **curr->sempid** 的低 16 位中。其隐含的部分是高 16 位（在此假定是 32 位平台）没有必要被设置成 0：C 标准有意给予编译器用 0 或用符号位拷贝来填充空余位的自由。在实际实现中，低级机器指令怎样能最快的工作，编译器就如何工作，结果有时是这些操作的一种，而有时则是另一种。（C 语言标准为什么不拘限于任何一种实现的原因正在于此。）这样的结果时，高位可以是全 0 也可以是全 1，这也正是在第 21116 行里只有低 16 位被屏蔽的原因。

Update_queue

20900：**update_queue** 函数在信号量的值发生改变时被调用。它完成那些此刻可以成功（或者将要失败）的挂起操作，并把它们从挂起队列中移出。

20907：如果这个节点的 **status** 标志已经被前一次 **update_queue** 调用增加过了，那么与该节点相关的进程就还没有机会把它自己从队列中移出。为了提供其它进程机会来执行它的挂起操作并从队列脱离，函数返回。

20910：检查是否此刻能够完成当前一组挂起操作。**q->alter** 是最后一个被通过的参数，所以即将成功的变异（mutating）操作就会自动被取消。这是因为进程将要继续亲自尝试这些操作，而它们是不应该被执行两次的。

20914：假设错误或者成功状态已经能够被判定（对立于需要继续等待），这个节点就被从队列中移出，并且与它关联的进程也会被唤醒。否则，节点留在队列中以便在将来某处被再次尝试。

20917：如果该操作集包括一些变异的操作，标志就被提高以便进程知道唤醒它是由于现在

能够成功了；进程将尝试那些操作并把自己从队列中移出。前边讨论过，第 21342 行要对这个标志进行检查。

20920：函数现在返回，以便多个变异进程不会同时尝试进行它们的那些可能并不互相兼容的改变。回忆一下，非变异的操作位于队列头部，而变异的操作是位于末尾的。其结果是，所有的非变异进程（它们不会彼此干扰）被首先唤醒，然后最多唤醒一个变异进程。

20922：否则，将产生一个错误。该错误代码被保存在 **q->status** 里，接着队列节点被移出。

Count_semncnt

20938：**count_semncnt** 函数从 21117 行被调用来实现 **sys_semctl** 内的 **GETNCNT** 命令。它的工作是记录因等待获得信号量而阻塞的任务数目。

20949：这个循环用于执行在 **sma** 的挂起队列中等待着的每个任务中的每个挂起操作。每当找到一个满足的操作时它就递增 **semncnt**——该操作试图获得特定的以及没有设置 **IPC_NOWAIT** 标志的信号量。

Count_semzcnt

20957：**count_semzcnt** 函数在 21119 行被调用以实现 **sys_semctl** 内的 **GETZCNT** 命令。它除了要对等待信号量归 0 的任务（也就是等待信号量变得可用的任务）进行计数之外，它和 **count_semncnt** 函数几乎一样。因此唯一的区别就在第 20970 行，在那里它使用等于 0 而不是小于 0 来进行测试。

共享内存

共享内存（shared memory）顾名思义就是：一块预留出的内存区域，而且一组进程均可对其进行访问。因为它涉及 IPC 和内存管理两方面的内容，这部分讨论将融合本章及第 8 章以前的材料进行分析。

截至目前为止，共享内存是本章要介绍的三种 IPC 机制里最快的一种，而且也是最简单的一种——对于进程来说，获得共享内存后它和任何其它内存看起来都是一样的。由一个进程对共享内存所作出的改变对所有其它进程都是立即可见的——它们只需通过一个指向共享内存空间的指针来读取，然后就轻松的获得了结果。然而，System V 共享内存没有确保互斥的内置方案：一个进程可以向共享内存中的给定地址写入而同时另一个进程从相同的地址读出，这会导致读者所看到的将是不一致的数据。这个问题在 SMP 机器上非常明显，但是它也会发生在 UP 机器之上——举个例子，假设正当把某个较大的结构写入共享内存空间时写者被转换出了上下文环境，而读者又在写者完成操作之前读取了共享内存的时候。

这样的结果是，使用共享内存的进程必须努力确保读操作与写操作的严格分离（考虑一下，写操作和写操作之间也是如此）。锁和原子操作的相关概念将在下一章详细论述。但是读者已经了解了保证互斥访问共享内存区域的一种方法：使用信号量。这种思想是一旦获得信号量就全速访问内存区域，工作一完成后就立即释放该信号量。

共享内存在一些用到消息队列的情况下也具有同样的帮助作用——一个调度进程可以把工作请求写入共享内存区域的一部分，同时工作者进程可以把结果写入另一部分。这就意味着应用程序要预先为请求和结果空间限制界限，但这样的内存分配和结果写入还是要比使用消息队列快。

对于每个进程来说共享内存区域不必看起来具有相同的地址。如果进程 A 和进程 B 都

在使用同一块共享内存区域，那么 A 可能看到它在一个地址，而 B 则可能会看它在另一个地址。当然，共享内存区域中给定的一个页面将最多被映射为一个物理页面。前一章介绍过的虚拟内存机制只需要为每个进程进行不同的逻辑地址转换即可。

在内核代码中，共享内存区域被称为段（segments），这正是有时会被误用于 VMA 的一个术语。为了预先防止任何混淆，这是一个该术语的非正式用法；它与第 8 章里讨论过的硬件增强的（MMU）段是不同的。为了避免这种说法所可能引起的迷惑，我将继续使用区域（regions）这个术语。

共享内存代码从设计到实现都与消息队列及信号量的代码有一些相似之处。因此，没有必要再介绍 **shm_init**（21482 行）和 **findkey**（21493 行）函数。出于同样的原因，剩下的一些函数和数据结构的讨论也会相应缩短。

Struct shmid_ds

17042：多少有点打破了已经建立的模式，**struct shmid_ds** 不是内核用来跟踪共享内存区域的数据结构。取而代之的是，**struct shmid_ds** 包含这种信息的绝大部分，而剩下的信息则位于下边要介绍的 **struct shmid_kernel** 中。以下是 **struct shmid_ds** 的那些与其对应对象所不同的成员：

- **shm_segsz**——这块共享内存区域的大小尺寸，用字节（不是页面）度量。
- **shm_nattch**——用典型的术语，是指“附属（attached）”到这块区域的任务数目——换句话说，就是使用该共享内存区域的任务数。这个成员是一个参考计数（reference count）。
- **shm_unused**、**shm_unused2** 和 **shm_unused3**——从它们的名字就可推断，这些成员不再用于实现之中；它们的唯一角色看来是为了保持该结构体大小的向后兼容性。

Struct shmid_kernel

17056：**struct shmid_kernel** 用于分离“私有（private）”的共享内存相关信息和“公有（public）”的信息。**struct shmid_ds** 里那些对用户应用程序可见的部分还保留在 **struct shmid_ds** 之内，而关系到内核的私有信息则位于 **struct shmid_kernel** 之内。用户应用程序需要能够通过 **struct shmid_ds** 来进行 **shmctl** 系统调用，所以它的定义必须对它们是可见的，但是内核私有实现的细节就不应该出现在 **struct** 的定义之中。否则，改变内核的执行可能会中断应用程序。**struct shmid_kernel** 具有如下成员；

- **u**——即 **struct shmid_ds**，也就是数据的公共部分。
- **shm_npages**——用页面数表示的共享内存区域的容量。它恰为 **shm_segsz** 成员除以 **PAGE_SIZE**（10791 行）的结果。
- **shm_pages**——用于跟踪这块共享内存区域页面分配的一个“页表”——“页表”在这里加了引号，是因为它不是一个同前一章里一样真正的、硬件支持的页表。不过它完成同样的工作。
- **attaches**——代表各自进程对这块共享内存区域进行映射的 VMA 的一个链表。VMA 在第 8 章里已经介绍过。

Newseg

21511：是与 **newque** 和 **newary** 相对应的函数。它分配并初始化一个 **struct shmid_kernel**，然后把它安置在 **shm_segs** 数组之中。

21537：分配“页表”。它和紧随 **struct shmid_kernel** 之后的对这块内存空间进行分配的另一个 IPC 代码一样，它们都是一个大的分配过程中的一部分。不过，**struct shmid_kernel** 是由 **kmalloc** 分配的（在不可交换的内核内存里），然而“页表”是由 **vmalloc** 分配的（在可交换内存里）。

21546：以把页表填零为起点来初始化所有分配了的元素项。

Sys_shmget

21573：这个函数自然是对应于 **sys_msgget** 和 **sys_semget** 的。唯一新颖的特征是它对进程 **struct mm_struct** 的信号量获取和释放过程。这是一个内核信号量，它与 System V 信号量并不相同——内核信号量将在第 10 章介绍。

Killseg

21610：这个函数对应于 **freeque** 和 **freeary**。它的代码也同那些函数的非常相似，但是有几个特征值得注意。

21616：如果用一个未被占用的 **shm_segs** 元素的索引调用 **killseg** 函数，它就显示一条警告并立刻返回。它的两个对应函数中都不存在相似的代码。

21629：如果元素项的 **shm_pages** 成员是 **NULL**，那么就在某处有一个逻辑错误。**struct shmid_kernel** 要么是没有完全构建好，要么就是已经销毁但还没有被从数组中删除，再或者就是某个类似的看起来“不可能发生”的情况发生了。

21635：释放为页表分配的页面。

21638：如果页表没有映射这个页面，在释放这一项时就无需执行什么操作。

21640：如果页面在物理内存里，则它将被释放回可用页面的缓冲池里，同时递减驻留页面的数目。

21643：否则，页面位于交换空间，它将从那里被释放。

21648：释放页表本身。

Sys_shmctl

21654：这个函数明显是对应于 **sys_msgctl** 和 **sys_semctl** 的，而且和它们有许多共同点。在此只介绍两个共享内存所特有的命令。

21733：**SHM_UNLOCK** 命令是 **SHM_LOCK** 的反作用命令，**case** 在第 21742 行。**SHM_LOCK** 允许拥有足够权能的进程锁住物理内存里的一整块区域，以防止它被交换出去。而 **SHM_UNLOCK** 则对一块加锁区域进行解锁，使得其中的页面再次可以被用于交换。

在这两个 **case** 里的工作看来不甚相似：它只是确定调用者有合适的权能、要被解锁的区域当前是加锁的（或反之亦然），然后设置或者清除适当的模式位。但是这就是所要完成的一切了——其效果会在 **shm_swap**（22172 行）中显现出来。

注意有一个分离的权能用于加锁和解锁共享内存，即 **CAP_IPC_LOCK**（14021 行）。

Insert_attach

21823：这个短小的函数只是把一个 VMA 添加到附属于给定 **struct shmid_kernel** 的 VMA 列表中。注意该 VMA 是添加到列表头部的——顺序并不重要，而且这样处理最为简单。否则的话，**attaches** 的头和尾都将不得不分别被进行跟踪。

Remove_attach

21833：这个函数自然是从附属于给定 **struct shmid_kernel** 的列表中移出一个 VMA。关于此函数的奇怪之处是它并不依赖于它的 **shp** 参数——该参数是一个指针，指向存储在 VMA 列表第一个 VMA 里的 **shp** 的 **attaches** 列表，它位于第 21829 行，而且用于更新列表的过程同样不考虑是否该 VMA 为列表里的第一项（如果它是，相应的 **attaches** 也被更新）。

Sys_shmat

21898：这个函数实现了 **shmat** 系统调用，调用进程借助它可以同一个共享内存区域建立联系。

21923：在一些熟悉的准备工作之后，**sys_shmat** 开始对共享内存区域应出现在调用进程内存空间中的地址进行计算。首先，它要检查调用者传过来的地址。如果它是 **NULL**，而且 **SHM_REMAP** 标志位也未被设置（参见 21959 行），那么请求必须被抛弃——**NULL** 永远不可被读和写。

21929：调用者传入 **NULL** 作为目标地址，这意味着 **sys_shmat** 应该在该进程的内存空间里选择一个地址。**get_unmapped_area** 将提供一个候选的地址（33432 行），顺便需要提及的是该函数在前一章已经讨论过。如果它返回 0（在所有内核支持的平台上都等价于 **NULL**），那么就是无法找到足够大的区域。

21932：如果候选的地址不是恰好在一个页面的边界上，它就会被向上舍入到更高的下一个页面边界，然后用调整过的地址将原先的地址取而代之。**get_unmapped_area** 返回在给定地址上或超过它的第一个可用地址，因此假如上舍入的地址是可用的，它将被采用。

现在解释一下为什么地址要向上舍入而不是向下舍入（那样能够更快和更简单一些）：假如 **sys_shmat** 进行向下舍入而所得地址不可用，那么代码将陷入死循环。下一次调用 **get_unmapped_area** 将从下舍入地址处向上搜索并返回到原先未经舍入的地址处，而它将再次被向下舍入，发现不合适，又传送给 **get_unmapped_area**……要注意在这里使用的是 **SHMLBA**（11777 行）而不是 **PAGE_SIZE**（10791 行）来决定地址的适宜性。不过，正如你所见到的，**SHMLBA** 恰好被定义为 **PAGE_SIZE**，所以效果是相同的。

如果 **SHMLBA** 和 **PAGE_SIZE** 是一样的，那么二者又为什么要兼有呢？答案在于 **SHMLBA** 在绝大多数平台上就是 **PAGE_SIZE**，但并不是在所有平台上都是如此。在 MIPS 上——CPU 具有 4K 的 **PAGE_SIZE**——Linux 把 **SHMLBA** 定义为非常大的 0x40000（256K），其注释说明选择这样大的值是为了遵守基于 MIPS 机器的 SGI 应用程序二进制接口（ABI——Application Binary Interface）。然而，MIPS ABI 的版本 2 和 3 却明确声明了 **SHMLBA** 的值“在符合标准的实现上是允许有所不同的”，所以不清楚为什么内核开发人员认为 256K 的值是必要的。或许该值是非常早期的 ABI 版本所要求的，但是我向回检查 ABI 一直到 1.2 版仍没有发现任何这样的要求。还有，在 SPARC-64 上，**SHMLBA** 是 **PAGE_SIZE** 的两倍；不幸的是，这个区别没有在代码中进行解释。

21936：否则，调用者传送一个建议的地址。如果有必要而且是被允许的，该地址就被向下取整。

21945：确保从被选地址开始的大小为 **len** 的内存块在进程的允许内存空间之内。（**len** 已在几行之前计算出来，21913 行。）当调用者提供候选地址时进行检查明显是必要的，

而且粗看起来当 **sys_shmat** 用 **get_unmapped_area** 来选择一个地址时进行检查也是必要的。 尽管区域的大小已经被传递给它，**get_unmapped_area** 还是要执行一个相似的检查，**struct shmid_ds** 的 **shm_segsz** 成员不必和 **len** 相同——**len** 是 **PAGE_SIZE** 的一个倍数，而 **shm_segsz** 则可以不是。

不过，因为所有被 **get_unmapped_area** 使用的地址都是页对准的，所以传递给它的区域大小是否是页面尺寸的倍数都不会影响它的计算。

21951：正如注释中所说明的，被选区域必须为进程的栈留出一些空间。这个缓冲区间有四个页面——这个数字并没有什么特别之处，只要达到进程有足够的栈空间的目的即可。在上一章中曾提到过如果某任务耗尽了它的栈，它将被杀死。综合考虑起来，让单个系统调用失败可能要比让整个进程被无理的杀死更好一些——进程可以从前者中逐渐恢复，但是后者却不行。

21959：**SHM_REMAP**（17075 行）的主要作用在此体现：如果 **SHM_REMAP** 被设置了而且调用者提供的区域已在使用，那么就没有错误，这是因为 **SHM_REMAP** 用于允许调用者把一块共享内存区域映射到它自己的内存里——比如是一个全局缓冲区。如果这个标志没有被设置，被选的地址就一定不能和进程已经拥有的任何内存相互重叠。

21971：如果调用者缺少使用这块内存区域的许可，系统调用失败。如果 **SHM_RDONLY**（只读）标志被提供，调用者只需要读许可；否则，调用者需要读许可和写许可。

21991：填充新的 VMA。特别注意它的 **vm_ops** 成员被初始化为指向 **shm_vm_ops**（21809 行），就像在第 8 章里讨论过的一样。

22004：增加这块区域的引用计数，以便它不会被过早的销毁。

22005：调用 **shm_map**（21844 行）把共享内存页面映射到进程的内存空间里。如果失败，它就会返回，同时递减引用计数，如果这是第一个和唯一的引用，那么还需要销毁该区域，然后释放 VMA，这样整个工作就结束了。

注意即使这是最后一个引用，VMA 也不必被释放；该区域也必须要用 **SHM_DEST** 标志（17106 行）来进行标记。**SHM_DEST** 可以在由调用者来设置的标志位之中；它也可以在后面 **sys_shmctl** 的 **IPC_RMID** 情况里被设置——参见 21780 行。以这样的方式，一块共享内存区域可以比它所有的附属进程生存更长时间。出于同保留一个检查点（checkpoint）文件会在某些情况下非常有用相类似的原因，这样的处理方式也是有用的：你可能会有一个每晚都要运行几个小时的耗时进程，要把它的处理结果保存在一个即使该进程当前工作完成之后仍然继续存在的共享内存区域。只要通过附属到剩下的共享内存区域，它就能够恰好在下一个晚上从停下的地方重新开始。（当然，由于共享内存区域——不同于文件——在计算机关闭后就会消失，所以这种方案不适用于不能有丢失危险的工作。）

22014：添加到附属于这块区域的 VMA 列表中，然后更新一些关于每区域统计的数据。

22019：返回在调用者空间里真正被选择的共享内存区域地址，然后成功地返回。

Shm_open

22028：**shm_open** 函数像是 **sys_shmat** 的一个简化版本（21898 行）。它把一个给定的 VMA 附加到一个共享内存区域里。提供的 VMA 是从一个已经附属于目标区域的 VMA 复制而来，所以这个 VMA 本身已经被正确填写了；**shm_open** 函数的工作基本上只是要完成附属连结。

正如 **shm_open** 上方的注释所陈述的，这个函数是从 **do_fork**（23953 行）里被调用的，该函数在第 7 章里已经介绍过。更准确的说，这个函数是在 **dup_mmap** 里（23654

行）第 23692 行被调用的。然后，**dup_mmap** 在 **copy_mm** 里（23774 行）的第 23801 行被调用；而 **copy_mm** 又是在 **do_fork** 里的第 24051 行被调用的。

22033：从 VMA 的 **vm_pte** 成员里抽取 **shm_segs** 下标，然后确保该处有一合法项。注意下标无需进行范围检查，这是因为和 **SHM_ID_MADK** 所进行的按位与操作（11757 行）已强迫它合乎范围了。

22040：添加 VMA 到区域里并更新区域的统计数字。

Shm_close

22050：**shm_close** 明显是 **shm_open** 的反作用函数，它把一个 VMA 从它附属的共享内存区域里分离出来。尽管在其它地方内核也可以调用 VMA 的 **close** 操作，但 33821 行看来是能结束调用 **shm_close** 的唯一之处。这是 **exit_mmap** 的一部分（33802 行），而它又是被 **mmput**（23764 行）调用、**mmput** 被**__exit_mm**（23174 行）调用，而 **__exit_mm** 又被 **do_exit**（23267 行）所调用，**do_exit** 函数在第 7 章就已经讨论过。要注意还有其它到达 **shm_close** 的路径，我们很快就会对其中之一进行介绍。

22056：从 VMA 的 **vm_pte** 成员里抽取 **shm_segs** 下标然后把该 VMA 分离出区域。出于和 **shm_open** 同样的原因，下标不用进行范围检查。还要注意的是 **shm_close** 不检查是否在指示的下标处存在一个合法的 **shm_segs** 项。读者已经看到，**remove_attach** 不依赖于它的 **shp** 参数，所以它对此并不关心。然而 **shm_close** 剩下部分将假定其它共享内存代码被正确的使用和执行，所以这种“不可能发生”的情形真的是不可能发生的。

22058：从共享内存区域分离 VMA 然后更新区域的统计数字。

22061：减少该区域的引用计数，如果可能的话还需要将其释放。

Sys_shmdt

22068：与 **sys_shmat** 相反，**sys_shmdt** 函数把一个进程从一块共享内存区域里分离出去。

22074：开始对所有代表进程内存的 VMA 进行循环处理。

22076：如果 VMA 代表一块共享内存区域（这可以通过检查它的 **vm_ops** 成员进行精巧的测试），而且该 VMA 始于目标地址，就应解除该 VMA 的映射。

22079：**do_munmap**（33689 行）调用 **unmap_fixup**（33578 行），它又间接的在 33592 行调用 **shm_close**。**do_munmap** 和 **unmap_fixup** 都在第 8 章里介绍过。

第 10 章 对称多处理（SMP）

在全书的讨论过程中，我一直在忽略 SMP 代码，而倾向于把注意力集中在只涉及一个处理器的相对简单的情况。现在已经到了重新访问读者已经熟悉的一些内容的时候了，不过要从一个新的角度来审视它：当内核必须支持多于一个 CPU 的机器时将发生什么？

在一般情况下，使用多于一个 CPU 来完成工作被称为并行处理（parallel processing），它可以被想象成是一段频谱范围，分布式计算（distributed computing）在其中一端，而对称多处理（SMP—symmetric multiprocessing）在另一端。通常，当你沿着该频谱从分布式计算向 SMP 移动时，系统将变得更加紧密耦合——在 CPU 之间共享更多的资源——而且更加均匀。在一个典型的分布式系统中，每个 CPU 通常都至少拥有它自己的高速缓存和 RAM。每个 CPU 还往往拥有自己的磁盘、图形子系统、声卡，监视器等等。

在极端的情形下，分布式系统经常不外乎就是一组普通的计算机，虽然它们可能具有完全不同的体系结构，但是都共同工作在某个网络之上——它们甚至不需要在同一个 LAN 里。读者可能知道的一些有趣的分布式系统包括：Beowulf，它是对相当传统而又极其强大的分布式系统的一个通用术语称谓；SETI@home，它通过利用上百万台计算机来协助搜寻地外生命的证据，以及 distributed.net，它是类似想法的另一个实现，它主要关注于地球上产生的密码的破解。

SMP 是并行处理的一个特殊情况，系统里所有 CPU 都是相同的。举例来说，SMP 就是你共同支配两块 80486 或两块 Pentium（具有相同的时钟速率）处理器，而不是一块 80486 和一块 Pentium，或者一块 Pentium 和一块 PowerPC。在通常的用法中，SMP 也意味着所有 CPU 都是“在相同处境下的”——那就是它们都在同一个计算机里，通过特殊用途的硬件进行彼此通信。

SMP 系统通常是另一种平常的单一（single）计算机——只不过具有两个或更多的 CPU。因此，SMP 系统除了 CPU 以外每样东西只有一个——一块图形卡、一个声音卡，等等之类。诸如 RAM 和磁盘这样以及类似的资源都是为系统的 CPU 们所共享的。（尽管现在 SMP 系统中每个 CPU 都拥有自己的高存缓存的情况已经变得愈发普遍了。）

分布式配置需要很少的或者甚至不需要来自内核的特殊支持；节点之间的协同是依靠用户空间的应用程序或者诸如网络子系统之类未经修改的内核组件来处理的。但是 SMP 在计算机系统内创建了一个不同的硬件配置，并由此需要特殊用途的内核支持。比如，内核必须确保 CPU 在访问它们的共享资源时要相互合作——这是一个读者在 UP 世界中所不曾遇到的问题。

SMP 的逐渐普及主要是因为通过 SMP 所获得的性能的提高要比购买几台独立的机器再把它们组合在一起更加便宜和简单，而且还因为它与等待下一代 CPU 面世相比要快的多。

非对称多 CPU 的配置没有受到广泛支持，这是因为对称配置情况所需的硬件和软件支持通常较为简单。不过，内核代码中平台无关的部分实际上并不特别关心是否 CPU 是相同的——即，是否配置是真正对称的——尽管它也没有进行任何特殊处理以支持非对称配置。例如，在非对称多处理系统中，调度程序应该更愿意在较快的而不是较慢的 CPU 上运行进程，但是 Linux 内核没有对此进行区别。

谚语说得好，“天下没有白吃的午餐”。对于 SMP，为提高的性能所付出的代价就是内核复杂度的增加和协同开销的增加。CPU 必须安排不互相干涉彼此的工作，但是它们又不能在这种协同上花费太多时间以至于它们显著地耗费额外的 CPU 能力。

代码的 SMP 特定部分由于 UP 机器存在的缘故而被单独编译，所以仅仅因为有了 SMP

寄存器是不会使 UP 寄存器慢下来的。这满足两条久经考验的原理：“为普遍情况进行优化”（UP 机器远比 SMP 机器普遍的多）以及“不为用不着的东西花钱”。

并行程序设计概念及其原语

具有两个 CPU 的 SMP 配置可能是最简单的并行配置，但就算是这最简单的配置也揭开了未知问题的新领域——即使要两块相同的 CPU 在一起协调的工作，时常也都像赶着猫去放牧一样困难。幸运的是，至少 30 年前以来，就在这个项目上作了大量和非常熟悉的研究工作。（考虑到第一台电子数字计算机也只是在 50 年前建造的，那这就是一段令人惊讶的相当长的时间了。）在分析对 SMP 的支持是如何影响内核代码之前，对该支持所基于的若干理论性概念进行一番浏览将能够极大的简化这个问题。

注意：并非所有这些信息都是针对 SMP 内核的。一些要讨论的问题甚至是由 UP 内核上的并行程序设计所引起的，既要支持中断也要处理进程之间的交互。因此即使你对 SMP 问题没有特别的兴趣，这部分的讨论也值得一看。

原子操作

在一个并行的环境里，某些动作必须以一种基本的原子方式（atomically）执行——即不可中断。这种操作必须是不可分割的，就象是原子曾经被认为的那样。

作为一个例子，考虑一下引用计数。如果你想要释放你所控制的一份共享资源并要了解是否还有其它（进程）仍在使用它，你就会减少对该共享资源的计数值并把该值与 0 进行对照测试。一个典型的动作顺序可能如下开始：

1. CPU 把当前计数值（假设是 2）装载进它的一个寄存器里。
2. CPU 在它的寄存器里把这个值递减；现在它是 1。
3. CPU 把新值（1）写回内存里。
4. CPU 推断出：因为该值是 1，某个其它进程仍在使用着共享对象，所以它将不会释放该对象。

对于 UP，应不必在此考虑过多（除了某些情况）。但是对于 SMP 就是另一番景象了：如果另一个 CPU 碰巧同时也在作同样的事情应如何处理呢？最坏的情形可能是这样的：

1. CPU A 把当前计数值（2）装载进它的一个寄存器里。
2. CPU B 把当前计数值（2）装载进它的一个寄存器里。
3. CPU A 在它的寄存器里把这个值递减；现在它是 1。
4. CPU B 在它的寄存器里把这个值递减；现在它是 1。
5. CPU A 把新值（1）写回内存里。
6. CPU B 把新值（1）写回内存里。
7. CPU A 推断出：因为该值是 1，某个其它进程仍在使用着共享对象，所以它将不会释放该对象。
8. CPU B 推断出：因为该值是 1，某个其它进程仍在使用着共享对象，所以它将不会释放该对象。

内存里的引用计数值现在应该是 0，然而它却是 1。两个进程都去掉了它们对该共享对象的引用，但是没有一个能够释放它。

这是一个有趣的失败，因为每个 CPU 都作了它所应该做的事情，尽管这样错误的结果还是发生了。当然这个问题就在于 CPU 没有协调它们的动作行为——右手不知道左手正在

干什么。

你会怎样试图在软件中解决这个问题呢？从任何一个 CPU 的观点来看待它——比如说是 CPU A。需要通知 CPU B 它不应使用引用计数值，由于你想要递减该值，所以不管怎样你最好改变某些 CPU B 所能见到的信息——也就是更新共享内存位置。举例来说，你可以为此目的而开辟出某个内存位置，并且对此达成一致：若任何一个 CPU 正试图减少引用计数它就包含一个 1，如果不是它就为 0。使用方法如下：

1. CPU A 从特殊内存位置出取出该值把它装载进它的一个寄存器里。
2. CPU A 检查它的寄存器里的值并发现它是 0（如果不是，它再次尝试，重复直到该寄存器为 0 为止。）
3. CPU A 把一个 1 写回特殊内存位置。
4. CPU A 访问受保护的引用计数值。
5. CPU A 把一个 0 写回特殊内存位置。

糟糕，令人不安的熟悉情况又出现了。以下所发生的问题仍然无法避免：

1. CPU A 从特殊内存位置出取出该值把它装载进它的一个寄存器里。
2. CPU B 从特殊内存位置出取出该值把它装载进它的一个寄存器里。
3. CPU A 检查它的寄存器里的值并发现它是 0。
4. CPU B 检查它的寄存器里的值并发现它是 0。
5. CPU A 把一个 1 写回特殊内存位置。
6. CPU B 把一个 1 写回特殊内存位置。
7. CPU A 访问受保护的引用计数值。
8. CPU B 访问受保护的引用计数值。
9. CPU A 把一个 0 写回特殊内存位置。
10. CPU B 把一个 0 写回特殊内存位置。

好吧，或许可以再使用一个特殊内存位置来保护被期望保护初始内存位置的那个特殊内存位置……。

面对这一点吧：我们在劫难逃。这种方案只会使问题向后再退一层，而不可能解决它。最后，原子性不可能由软件单独保证——必须要有硬件的特殊帮助。

在 x86 平台上，**lock** 指令正好能够提供这种帮助。（准确地说，**lock** 是一个前缀而非一个单独的指令，不过这种区别和我们的目的没有利害关系。）**lock** 指令用于在随后的指令执行期间锁住内存总线——至少是对目的内存地址。因为 x86 可以在内存里直接减值，而无需明确的先把它读入一个寄存器中，这样对于执行一个减值原子操作来说就是万事俱备了：**lock** 内存总线然后立刻对该内存位置执行 **decl** 操作。

函数 **atomic_dec**（10241 行）正好为 x86 平台完成这样的工作。**LOCK** 宏的 **SMP** 版本在第 10192 行定义并扩展成 **lock** 指令。（在随后的两行定义的 UP 版本完全就是空的——单 CPU 不需要保护自己以防其它 CPU 的干扰，所以锁住内存总线将完全是在浪费时间。）通过把 **LOCK** 宏放在内嵌编译指令的前边，随后的指令就会为 SMP 内核而被锁定。如果 CPU B 在 CPU A 发挥作用时执行了 **atomic_dec** 函数，那么 CPU B 就会自动的等待 CPU A 把锁移开。这样就能够成功了！

这样还只能说是差不多。最初的问题仍然没有被很好的解决。目标不仅是要自动递减引用计数值，而且还要知道结果值是否是 0。现在可以完成原子递减了，可是如果另一个处理器在递减和结果测试之间又“偷偷的”进行了干预，那又怎么办呢？

幸运的是，解决这个部分问题不需要来自 CPU 的特殊目的的帮助。不管加锁还是未锁，x86 的 **decl** 指令总是会在结果为 0 时设置 CPU 的 Zero 标志位，而且这个标志位是 CPU 私有的，所以其它 CPU 的所为是不可能在递减步骤和测试步骤之间影响到这个标志位的。相

应的，**atomic_dec_and_test**（10249 行）如前完成一次加锁的递减，接着依据 CPU 的 Zero 标志位来设置本地变量 c。如果递减之后结果是 0 函数就返回非零值（真）。

如同其它定义在一个文件里的函数一样，**atomic_dec** 和 **atomic_dec_and_test** 都对一个类型为 **atomic_t** 的（10205 行）对象进行操作。就像 **LOCK**，**atomic_t** 对于 UP 和 SMP 也有不同的定义方式——不同之处在于 SMP 情况里引入了 **volatile** 限定词，它指示 gcc 不要对被标记的变量做某种假定（比如，不要假定它可以被安全的保存在一个寄存器里）。

顺便提及一下，读者在这段代码里看到的垃圾代码**__atomic_fool_gcc** 据报告已不再需要了；它曾用于纠正在 gcc 的早期版本下代码生成里的一个故障。

Test-And-Set

经典的并行原语是 test-and-set。test-and-set 操作自动地从一个内存位置读取一个值然后写入一个新值，并把旧值返回。典型的，该位置可以保存 0 或者 1，而且 test-and-set 所写的新值是 1——因此是“设置（set）”。与 test-and-set 对等的是 test-and-clear，它是同样的操作除了写入的是 0 而不是 1。一些 test-and-set 的变体既能写入 1 也可以写入 0，因此 test-and-set 和 test-and-clear 就能够成为一体，只是操作数不同而已。

test-and-set 原语足以实现任何其它并行安全的操作。（实际上，在某些 CPU 上 test-and-set 是唯一被提供的此类原语。）比如，原本 test-and-set 是能够用于前边的例子之中来保护引用计数值的。相似的方法以被尝试——从一个内存位置读取一个值，检查它是否为 0，如果是则写入一个 1，然后继续访问受保护的值。这种尝试的失败并不是因为它在逻辑上是不健全的，而是因为没有可行的方法使其自动完成。假使有了一个原子的 test-and-set，你就可以不通过使用 **lock** 来原子化 **decl** 的方法而顺利通过了。

然而，test-and-set 也有缺点：

- 它是一个低级的原语——在所有与它打交道时，其它原语都必须在它之上被执行。
- 它并不经济——当机器测试该值并发现它已经是 1 了怎么办呢？这个值在内存里不会被搞乱，因为只要用同样的值复写它即可。可事实是它已被设置就意味着其它进程正在访问受保护的对象，所以还不能这样执行。额外需要的逻辑——测试并循环——会浪费 CPU 时钟周期并使得程序变得更大一些（它还会浪费高速缓存里的空间）。

x86 的 **lock** 指令使高级指令更容易执行，但是你也可以在上执行原子 test-and-set 操作。最直接的方式是把 **lock** 和 **btsl** 指令（位 test-and-set）联合起来使用。这种方法要被本章后边介绍的自旋锁（spinlock）所用到。

另一种在 x86 上实现的方法是用它的 **xchg**（exchange）指令，它能够被 x86 自动处理，就好像它的前面有一个 **lock** 指令一样——只要它的一个操作数是在内存里。**xchg** 要比 **lock/btsl** 组合更为普遍，因为它可以一次交换 8、16，或者 32 位而不仅仅是 1 位。除了一个在 arch/i386/kernel/entry.S 里的使用之外，内核对 **xchg** 指令的使用都隐藏在 **xchg** 宏（13052 行）之后，而它又是在函数**__xchg**（13061 行）之上实现的。这样是便于在平台相关的代码里内核代码也可以使用 **xchg** 宏；每种平台都提供它自己对于该宏的等价的实现。

有趣的时，**xchg** 宏是另一个宏，**tas**（test-and-set——13054 行）的基础。然而，内核代码的任何一个地方都没有用到这个宏。

内核有时候使用 **xchg** 宏来完成简单的 test-and-set 操作（尽管不必在锁变得可用之前一直循环，如同第 22770 行），并把它用于其它目的（如同第 27427 行）。

信号量

第 9 章中讨论了信号量的基本概念并演示了它们在进程间通信中的用法。内核为达自己的目的有其特有的信号量实现，它们被特别的称为是“内核信号量”。（在这一章里，未经修饰的名词“信号量”应被理解为是“内核信号量”。）第 9 章里所讨论的基本信号量的概念同样适用于内核信号量：允许一个可访问某资源用户的最大数目（最初悬挂在吊钩上钥匙的特定数目），然后规定每个申请资源者都必须先获得一把钥匙才能使用该资源。

到目前为止，你大概应该已经发现信号量如何能够被建立在 test-and-set 之上并成为二元（“唯一钥匙”）信号量，或者在像 **atomic_dec_and_test** 这样的函数之上成为计数信号量的过程。内核正好就完成着这样的工作：它用整数代表信号量并使用函数 **down**（11644 行）和 **up**（11714 行）以及其它一些函数来递减和递增该整数。读者将看到，用于减少和增加整数的底层代码和 **atomic_dec_and_test** 及其它类似函数所使用的代码是一样的。

作为相关历史事件的提示，第一位规范信号量概念的研究者，Edsger Dijistra 是荷兰人，所以信号量的基础操作就用荷兰语命名为：Proberen 和 Verhogen，常缩写成 P 和 V。这对术语被翻译成“测试（test）”（检查是否还有一把钥匙可用，若是就取走）和“递增（increment）”（把一个钥匙放回到吊钩之上）。那些词首字母正是在前一章中所引入的术语“获得（procure）”和“交出（vacate）”的来源。Linux 内核打破了这个传统，用操作 **down** 和 **up** 的称呼取代了它们。

内核用一个非常简单的类型来代表信号量：定义在 11609 行的 **struct semaphore**。他只有三个成员：

- **count**——跟踪仍然可用的钥匙数目。如果是 0，钥匙就被取完了；如果是负数，钥匙被取完而且还有其它申请者在等待它。另外，如果 **count** 是 0 或负数，那么其它申请者的数目就等于 **count** 的绝对值。

  Sema_init 宏（11637 行）允许 **count** 被初始化为任何值，所以内核信号量可以是二元的（初始化 **count** 为 1）也可以是计数型的（赋予它某个更大的初始值）。所有内核信号量代码都完全支持二元和计数型信号量，前者可作为后者的一个特例。不过在实践中 **count** 总是被初始化为 1，这样内核信号量也总是二元类型的。尽管如此，没有什么能够阻止一个开发者将来增加一个新的计数信号量。

  要顺便提及的是，把 **count** 初始化为正值而且用递减它来表明你需要一个信号量的方法并没有什么神秘之处。你也可以用一个负值（或者是 0）来初始化计数值然后增加它，或者遵循其它的方案。使用正的数字只是内核所采用的办法，而这碰巧和我们头脑中的吊钩上的钥匙模型吻合得相当好。的确，正如你将看到的那样，内核锁采用的是另一种方式工作——它被初始化为负值，并在进程需要它时进行增加。
- **waking**——在 **up** 操作期间及之后被暂时使用；如果 **up** 正在释放信号量则它被设置为 1，否则是 0。
- **wait**——因为要等待这个信号量再次变为可用而不得不被挂起的进程队列。

down

11644：**down** 操作递减信号量计数值。你可能会认为它与概念里的实现一样简单，不过实际上远不是这样简单。

11648：减少信号量计数值——要确保对 SMP 这是自动完成的。对于 SMP 来说（当然也适于 UP），除了被访问的整数是在一个不同类型的 **struct** 之内以外，这同在 **atomic_dec_and_test** 中所完成的工作本质上是相同的。

读者可能会怀疑 **count** 是否会下溢。它不会：进程总是在递减 **count** 之后进入休眠，所以一个给定的进程一次只能获得一个信号量，而且 **int** 具有的负值要比进程的数目多的多。

11652：如果符号位被设置，信号量就是负值。这意味着甚至它在被递减之前就是 0 或者负值了，这样进程无法得到该信号量并因此而应该休眠一直到它变成可用。接下来的几行代码十分巧妙地完成了这一点。如果符号位被设置则执行 **js** 跳转（即若 **decl** 的结果是负的它就跳转），**2f** 标识出跳转的目的地。**2f** 并非十六进制值——它是特殊的 GNU 汇编程序语法：**2** 表示跳转到本地符号“2”，**f** 表示向前搜索这个符号。（**2b** 将表示向后搜索最近的本地符号“2”。）这个本地符号在第 11655 行。

11653：分支转移没有执行，所以进程得到了信号量。虽然看起来不是这样，但是这实际已经到达 **down** 的末尾。稍后将对此进行解释。

11654：**down** 的技巧在于指令**.section** 紧跟在跳转目标的前面，它表示把随后的代码汇编到内核的一个单独的段中——该段被称为**.text.lock**。这个段将在内存中被分配并标识为可执行的。这一点是由跟在段名之后的 **ax** 标志字符串来指定的——注意这个 **ax** 与 x86 的 AX 寄存器无关。

这样的结果是，汇编程序将把 11655 和 11656 行的指令从 **down** 所在的段里转移到可执行内核的一个不同的段里。所以这些行生成的目标代码与其前边的行所生成的代码从物理上不是连续的。这就是为什么说 11653 行是 **down** 的结尾的原因。

11655：当信号量无法得到时跳转到的这一目的行。**Pushl $1b** 并不是要把十六进制值 1b 压入栈中——如果要执行那种工作应该使用 **pushl $0x1b**（也可以写成是不带**$**的）。正确的解释是，这个 **1b** 和前边见到的 **2f** 一样都是 GNU 汇编程序语法——它指向一个指令的地址；在此情形中，它是向后搜索时碰到的第一个本地标识“1”的地址。所以，这条指令是把 11653 行代码的地址压入栈中；这个地址将成为返回地址，以便在随后的跳转操作之后，执行过程还能返回到 **down** 的末尾。

11656：开始跳转到**__down_failed**（不包括在本书之内）。这个函数在栈里保存几个寄存器并调用后边要介绍的**__down**（26932 行）来完成等待信号量的工作。一旦**__down** 返回了，**__down_failed** 就返回到 **down**，而它也随之返回。一直到进程获得了信号量**__down** 才会返回；最终结果就是只要 **down** 返回，进程就得到信号量了，而不管它是立刻还是经过等待后获得的它。

11657：伪汇编程序指令**.previous** 的作用未在正式文档中说明，但是它的意思肯定是还原到以前的段中，结束 11654 行里的伪指令**.section** 的作用效果。

down_interruptible

11664：**down_interruptible** 函数被用于进程想要获得信号量但也愿意在等待它时被信号中断的情况。这个函数与 **down** 的实现非常相似，不过有两个区别将在随后的两段里进行解释。

11666：第一个区别是 **down_interruptible** 函数返回一个 **int** 值来指示是否它获得了信号量或者被一个信号所打断。在前一种情况里返回值（在 **result** 里）是 0，在后一种情况里它是负值。这部分上是由 11675 行代码完成的，如果函数未经等待获得了信号量则该行把 **result** 设置为 0。

11679：第二个区别是 **down_interruptible** 函数跳转到**__down_failed_interruptible**（不包括在本书之内）而不是**__down_failed**。因循**__down_failed** 建立起来的模式，**__down_failed_interruptible** 只是调整几个寄存器并调用将在随后进行研究的**__down_interruptible** 函数（26942 行）。要注意的是 11676 行为**__down_failed_**

interruptible 设置的返回目标跟在 **xorl** 之后，**xorl** 用于在信号量可以被立刻获得的情况中把 **result** 归 0。**down_interruptible** 函数的返回值再被复制进 **result** 中。

down_trylock

11687：除了调用**__down_failed_trylock** 函数（当然还要调用 26961 行的**__down_trylock** 函数，我们将在后面对它进行检查）之外，**down_trylock** 函数和 **down_interruptible** 函数相同。因此，在这里不必对 **down_trylock** 函数进行更多解释。

DOWN_VAR

26900：这是作为**__down** 和**_down_interruptible** 共同代码因子的三个宏中的第一个。它只是声明了几个变量。

DOWN_HEAD

26904：这个宏使任务 **tsk**（被 **DOWN_VAR** 所声明）转移到 **task_state** 给出的状态，然后把 **tsk** 添加到等待信号量的任务队列。最后，它开始一个无限循环，在此循环期间当**__down** 和**__down_interruptible** 准备退出时将使用 **break** 语句结束该循环。

DOWN_TAIL

26926：这个宏完成循环收尾工作，把 **tsk** 设置回 **task_state** 的状态，为再次尝试获得信号量做准备。

26929：循环已经退出；**tsk** 已或者得到了信号量或者被一个信号中断了（仅适于**__down_interruptible**）。无论哪一种方式，任务已准备再次运行而不再等待该信号量了，因此它被转移回 **TASK_RUNNING** 并从信号量的等待队列里被注销。

__down

26932：**__down** 和**__down_interruptible** 遵循以下模式：

1. 用 **DOWN_VAR** 声明所需的本地变量，随后可能还有补充的本地变量声明。
2. 以 **DOWN_HEAD** 开始进入无穷循环。
3. 在循环体内完成函数特定的（function-specific）工作。
4. 重新调度。
5. 以 **DOWN_TAIL** 结束。注意对 **schedule** 的调用（26686 行，在第 7 章里讨论过）可以被移进 **DOWN_TAIL** 宏中。
6. 完成任何函数特定的收尾工作。

我将只对函数特定的步骤（第 3 和第 6 步）进行讨论。

26936：**__down** 的循环体调用 **waking_non_zero**（未包括），它自动检查 **sem->waking** 来判断是否进程正被 **up** 唤醒。如果是这样，它将 **waking** 归零并返回 1（这仍然是同一个原子操作的一部分）；如果不是，它返回 0。因此，它返回的值指示了是否进程获得了信号量。如果它获得了值，循环就退出，接着函数也将返回。否则，进程将继续等待。

顺便要说明的是，观察一下**__down** 尝试获得信号量是在调用 **schedule** 之前。如果信号量的计数值已知为负值时，为什么不用另一种相反的方式来实现它呢？实际上它对于第一遍循环之后的任何一遍重复都是没有影响的，但是去掉一次没有必要的检查可以稍微加快第一遍循环的速度。如果需要为此提出什么特别的理由的话，那

可能就是因为自从信号量第一次被检查之后的几个微秒内它就应该可以被释放（可能是在另一个处理器上），而且额外获取标志要比一次额外调度所付出的代价少得多。因此**__down** 可能还可以在重新调度之前做一次快速检查。

__down_interruptible

26942：**__down_interruptible** 除了允许被信号中断以外，它和**__down** 在本质上是一样的。

26948：所以，当获取信号量时对 **waking_non_zero_interruptible**（未包括）进行调用。如果它没能得到信号量就返回 0，如果得到就返回 1，或者如果它被一个信号所中断就返回**–EINTR**。在第一种情况下，循环继续。

26958：否则，**__down_interruptible** 退出，如果它得到信号量就返回 0（不是 1），或者假如被中断则返回**–EINTR**。

__down_trylock

26961：有时在不能立刻获得信号量的情况下，内核也需要继续运行。所以，**__down_trylock** 不在循环之内。它仅仅调用 **waking_nonzero_trylock**（未包括），该函数夺取信号量，如果失败就递增该信号量的 **count**（因为内核不打算继续等待下去）然后返回。

up

11714：我们已经详尽的分析了内核尝试获得信号量时的情况，也讨论了它失败时的情况。现在是考察另一面的时候了：当释放一个信号量时将发生什么。这一部分相对简单。

11721：原子性地递增信号量的计数值。

11722：如果结果小于等于 0，就有某个进程正在等待被唤醒。up 向前跳转到 11725 行。

11724：**up** 采用了 **down** 里同样的技巧：这一行进入了内核的单独的一段，而不是在 **up** 本身的段内。**up** 的末尾的地址被压入栈然后 **up** 跳转到**__up_wakeup**（未包括）。这里完成如同**__down_failed** 一样的寄存器操作并调用下边要讨论的**__up** 函数。

__up

26877：**__up** 函数负责唤醒所有等待该信号量的进程。

26897：调用 **wake_one_more**（未包括在本书中），该函数检查是否有进程在等待该信号量，如果有，就增加 **waking** 成员来通知它们可以尝试获取它了。

26880：利用 **wake_up** 宏（16612 行），它只是调用**__wake_up** 函数（26829 行）来唤醒所有等待进程。

__wake_up

26829：正如在第 2 章中所讨论的那样，**__wake_up** 函数唤醒所有传递给它的在等待队列上的进程，假如它们处于被 **mode** 所隐含的状态之一的话。当从 **wake_up** 被调用时，函数唤醒所有处于 **TASK_UNINTERRUPTIBLE** 或 **TASK_INTERRUPTIBLE** 状态的进程；当从 **wake_up_interruptible**（16614 行）被调用时，它只唤醒处于 **TASK_INTERRUPTIBLE** 状态的任务。

26842：进程用 **wake_up_process**（26356 行）被唤醒，该函数曾在以前提到过，它将在本章随后进行详细介绍。

现在所感兴趣的是唤醒所有进程后的结果。因为**__wake_up** 唤醒所有队列里的进程，而不仅仅是队列里的第一个，所以它们都要竞争信号量——在 SMP 里，它们可以精确的同

时做这件事。通常，获胜者将首先获得 CPU。这个进程将是拥有最大“goodness”的进程（回忆一下第 7 章中 26338 行对 **goodness** 的讨论）。 这一点意义非常重大，因为拥有更高优先权的进程应该首先被给予继续其工作的机会。（这对于实时进程尤其重要。）

这种方案的不足之处是有发生饥饿（starvation）的危险，这发生在一个进程永远不能得到它赖以继续运行的资源时。这里可能会发生饥饿现象：假如两个进程反复竞争同一个信号量，而第一个进程总是有比第二个更高的优先权，那么第二个进程将永远不会得到 CPU。这种场景同它应该的运行方式存在一定差距——设想一个是实时进程而另一个以 20 的 niceness 运行。我们可以通过只唤醒队列里第一个进程的方法来避免这种饥饿的危险，可是那样又将意味着有时候会耽误从各个方面来说都更有资格的进程对 CPU 的使用。

以前对此没有讨论过，可是 Linux 的调度程序在适当的环境下也能够使得 CPU 的一个进程被彻底饿死。这不完全是一件坏事——只是一种设计决策而已——而且至少应用于通篇内核代码的原则是一致的，这就很好。还要注意的是使用前边讨论过的其它机制，饥饿现象也同样会发生。例如说，test-and-set 原语就是和内核信号量一样的潜在饥饿根源。

无论如何，在实际中，饥饿是非常少见的——它只是一个有趣的理论案例。

Spinlocks

这一章里最后一个重要的并行程序设计原语是自旋锁（spinlock）。自旋锁的思想就是在一个密封的循环里坚持反复尝试夺取一个资源（一把锁）直到成功为止。这通常是通过在类似 test-and-set 操作之上进行循环来实现的——即，旋转（spinning）——一直到获得该锁。

如果这听起来好像是一个二元信号量，那是因为它就是一个二元信号量。自旋锁和二元信号量唯一的概念区别就是你不必循环等待一个信号量——你可以夺取信号量，也可以在不能立刻得到它时放弃申请。因此，自旋锁原本是可以通过在信号量代码外再包裹一层循环来实现的。不过，因为自旋锁是信号量的一个受限特例，它们有更高效的实现方法。

自旋锁变量——其中的一位被测试和设置——总是 **spinlock_t** 类型（12785 行）。只有 **spinlock_t** 的最低位被使用；如果锁可用，则它是 0，如果被取走，则它是 1。在一个声明里，自旋锁被初始化为值 **SPIN_LOCK_UNLOCKED**（12789 行）；它也可以用 **spin_lock_init** 函数（12791 行）来初始化。这两者都把 **spinlock_t** 的 **lock** 成员设置成 0——也就是未锁状态。

注意 12795 行代码简洁地对公平性进行了考虑并最后抛弃了它——公平是饥饿的背面，正如我们前面已经介绍过的（使得一个 CPU 或进程饥饿应被认为是“不公平的”）。

自旋锁的加锁和解锁宏建立在 **spin_lock_string** 和 **sping_unlock_string** 函数之上，所以这一小节只对 **spin_lock_string** 和 **sping_unlock_string** 函数进行详述。其它宏如果有的话只是增加了 IRQ 加锁和解锁。

spin_lock_string

12805：这个宏的代码对于所有自旋锁加锁的宏都是相同的。它也被用于 x86 专用的 **lock_kernel** 和 **unlock_kernel** 版本之中（它们不在本书之列，不过其常规版本则是包括的——参见 10174 和 10182 行）。

12807：尝试测试和设置自旋锁的最低位，这要把内存总线锁住以便对于任何其它对同一个自旋锁的访问来说这个操作都是原子的。

12808：如果成功了，控制流程就继续向下运行；否则，**spin_lock_string** 函数向前跳转到第 12810 行（**btsl** 把这一位的原值放入 CPU 的进位标志位（Carry flag），这正是这里使

用 **jc** 的原因）。同样的技巧我们已经看到过三次了：跳转目标放在内核的单独一段中。

12811：在封闭的循环里不停地检测循环锁的最低位。注意 **btsl** 和 **testb** 以不同方式解释它们第一个操作数——对于 **btsl**，它是一个位状态（bit position），而对于 **testb**，它是一个位屏蔽（bitmask）。因此，12811 行在测试 **spin_lock_string** 曾在 12807 行已经试图设置（但失败了）的同一位，尽管一个使用**$0** 而另一个使用**$1**。

12813：该位被清除了，所以 **spin_lock_string** 应该再次夺取它。函数调转回第 12806 行。这个代码可以只用加上 **lock** 前缀的两条代码加以简化：

```
1: lock ; btsl $0, %0
   jc 1b
```

不过，使用这个简化版本的话，系统性能将明显受到损害，这因为每次循环重复内存总线都要被加锁。内核使用的版本虽然长一些，但是它可以使其它 CPU 运行的更有效，这是由于该版本只有在它有充分理由相信能够获得锁的时候才会锁住内存总线。

spin_unlock_string

12816：并不很重要：只是重新设置了自旋锁的锁定位（lock bit）。

读/写自旋锁

自旋锁的一个特殊情况就是读/写自旋锁。这里的思想是这样的：在某些情况中，我们想要允许某个对象有多个读者，但是当有一个写者正在写入这个对象时，则不允许它再有其它读者或者写者。

遵循基于 **spinlock_t** 的自旋锁的同样模式，读/写自旋锁是用 **rwlock_t**（12853 行）来代表的，它可以在有 **RW_LOCK_UNLOCKED**（12858 行）的声明里被初始化。与 **rwlock_t** 一起工作的最低级的宏是 **read_lock**、**read_unlock**、**write_lock**，以及 **write_unlock**，它们在本小节中进行描述。很明显，那些跟随在这些宏之后并建立在它们之上的宏，自然要在你理解了最初的这四个宏之后在去接触。

正如第 12860 行注释中所声明的，当写锁（write lock）被占有时，**rwlock_t** 的 **lock** 成员是负值。当既没有读者也没有写者时它为 0，当只有读者而没有写者时它是正值——在这种情况下，**lock** 将对读者的数目进行计数。

read_lock

12867：开始于 **rwlock_t** 的 **lock** 成员的自动递增。这是推测性的操作——它可以被撤销。

12868：如果它在增量之后为负，表示某个进程占用了写锁——或者至少是某个进程正试图得到它。**read_lock** 向前跳到第 12870 行（注意，在一个不同的内核段里）。否则，没有写者退出（尽管还有可能有，或者也有可能没有其它读者——这并不重要），所以可以继续执行读锁定（read-locked）代码。

12870：一个写者出现了。**read_lock** 取消第 12867 行增值操作的影响。

12871：循环等待 **rwlock_t** 的 **lock** 变为 0 或正值。

12873：跳回到第 12866 行再次尝试。

read_unlock

12878：不太复杂：只是递减该计数值。

write_lock

12883：表示出有一个进程需要写锁：检测并设置 **lock** 的符号位并保证 **lock** 的值是负的。

12884：如果符号位已经被设置，则另外有进程占有了写锁；**write_lock** 向前跳转到第 12889 行（同以前一样，那是在一个不同的内核段里）。

12885：没有别的进程正试图获得该写锁，可是读者仍可以退出。因为符号位被设置了，读者不能获得读锁，但是 **write_lock** 仍然必须等待正在退出的读者完全离开。它通过检查低端的 31 位中是否任何一位被设置过开始，这可以表示 **lock** 以前曾是正值。如果没有，则 **lock** 在符号位反转之前曾是 0，这意味着没有读者；因而，这对于写者的继续工作是很安全的，所以控制流程就可以继续向下运行了。不过，如果低端 31 位中任何一位被设置过了，也就是说有读者了，这样 **write_lock** 就会向前跳转到第 12888 行等到它们结束。

12888：该进程是仅有的写者，但是有若干读者。**write_lock** 会暂时清除符号位（这个宏稍后将再次操纵它）。有趣的是，对符号位进行这样的胡乱操作并不会影响读者操纵 **lock** 的正确性。考虑作为示例的下列顺序事件：

1. 两个读者增加了 **lock**；**lock** 用十六进制表示现在是 0x00000002。
2. 一个即将成为写者的进程设置了符号位；**lock** 现在是 0x80000002。
3. 读者中的一个离开；**lock** 现在是 0x80000001。
4. 写者看到剩余的位不全部是 0——仍然有读者存在。这样它根本没有写锁，因此它就清除符号位；**lock** 现在是 0x00000001。

这样，读和写可以任何顺序交错尝试操作而不会影响结果的正确程度。

12889：循环等待计数值降到 0——也就是等待所有读者退出。实际上，0 除了表示所有读者已离开之外，它还表示着没有其它进程获得了写锁。

12891：所有读者和写者都结束了操作；**write_lock** 又从头开始，并再次获得写锁。

write_unlock

12896：不太重要：只是重置符号位。

APICs 和 CPU-To-CPU 通信

Intel 多处理规范的核心就是高级可编程中断控制器（Advanced Programmable Interrupt Controllers——APICs）的使用。CPU 通过彼此发送中断来完成它们之间的通信。通过给中断附加动作（actions），不同的 CPU 可以在某种程度上彼此进行控制。每个 CPU 有自己的 APIC（成为那个 CPU 的本地 APIC），并且还有一个 I/O APIC 来处理由 I/O 设备引起的中断。在普通的多处理器系统中，I/O APIC 取代了第 6 章里提到的中断控制器芯片组的作用。

这里有几个示例性的函数来让你了解其工作方式的风格。

smp_send_reschedule

5019： 这个函数只有一行，其作用将在本章随后进行说明，它仅仅是给其 ID 以参数形式给出了的目标 CPU 发送一个中断。函数用 CPU ID 和 **RESCHEDULE_VECTOR** 向量调用 **send_IPI_single** 函数（4937 行）。**RESCHEDULE_VECTOR** 与其它 CPU 中断向量是一起在第 1723 行开始的一个定义块中被定义的。

send_IPI_single

4937： **send_IPI_single** 函数发送一个 IPI——那是 Intel 对处理器间中断（interprocessor interrupt）的称呼——给指定的目的 CPU。在这一行，内核以相当低级的方式与发送 CPU 的本地 APIC 对话。

4949： 得到中断命令寄存器（ICR）高半段的内容——本地 APIC 就是通过这个寄存器进行编程的——不过它的目的信息段要被设置为 **dest**。尽管**__prepare_ICR2**（4885 行）里使用了“2”，CPU 实际上只有一个 ICR 而不是两个。但是它是一个 64 位寄存器，内核更愿意把它看作是两个 32 位寄存器——在内核代码里，“ICR”表示这个寄存器的低端 32 位，所以“ICR2”就表示高端 32 位。我们想要设置的的目的信息段就在高端 32 位，即 ICR2 里。

4950： 把修改过的信息写回 ICR。现在 ICR 知道了目的 CPU。

4953： 调用**__prepare_ICR**（4874 行）来设置我们想要发送给目的 CPU 的中断向量。（注意没有什么措施能够保证目的 CPU 不是当前 CPU——ICR 完全能够发送一个 IPI 给它自己的 CPU。尽管这样，我还是没有找到有任何理由要这样做。）

4957： 通过往 ICR 里写入新的配置来发送中断。

SMP 支持如何影响内核

既然读者已经学习了能够成功支持 SMP 的若干原语，那么就让我们来纵览一下内核的 SMP 支持吧。本章剩余的部分将局限于对分布在内核之中的那些具有代表性的 SMP 代码进行讨论。

对调度的影响

schedule（26686 行）正是内核的调度函数，它已在第 7 章中全面地介绍过了。**schedule** 的 SMP 版本与 UP 的相比有两个主要区别：

- 在 **schedule** 里从第 26780 开始的一段代码要计算某些其它地方所需的信息。
- 在 SMP 和 UP 上都要发生的对**__schedule_tail** 的调用（26638 行）实际上在 UP 上并无作用，因为**__schedule_tail** 完全是为 SMP 所写的代码，所以从实用的角度来说它就是 SMP 所特有的。

schedule

26784：获取当前时间，也就是自从机器开机后时钟流逝的周期数。这很像是检查 **jiffies**，不过是以 CPU 周期而不是以时钟滴答作为计时方法的——显然，这要精确得多。

26785：计算自从 **schedule** 上一次在此 CPU 上进行调度后过去了多长时间，并且为下一次的计算而记录下当前周期计数。（**schedule_data** 是每个 CPU **aligned_data** 数组的一部分，它在 26628 行定义。）

26790：进程的 **avg_slice** 成员（16342 行）记录该进程在其生命周期里占有 CPU 的平均时间。可是这并不是简单的平均——它是加权平均，进程近期的活动远比很久以前的活动权值大。（因为真实计算机的计算是有穷的，“很久以前”的部分在足够远以后，将逐渐趋近于 0。）这将在 **reschedule_idle** 中（26221 行，下文讨论）被用来决定是否把进程调入另一个 CPU 中。因此，在 UP 的情况下它是无需而且也不会被计算的。

26797：记录哪一个 CPU 将运行 **next**（它将在当前的 CPU 上被执行），并引发它的 **has_cpu** 标志位。

26803：如果上下文环境发生了切换，**schedule** 记录失去 CPU 的进程——这将在下文的 **__schedule_tail** 中被使用到。

__schedule_tail

26654：如果失去 CPU 的任务已经改变了状态（这一点在前边的注释里解释过了），它将被标记以便今后的重新调度。

26664：因为内核已经调度出了这个进程，它就不再拥有 CPU 了——这样的事实也将被记录。

reschedule_idle

26221：当已经不在运行队列里的进程被唤醒时，**wake_up_process** 将调用 **reschedule_idle**，进程是作为 **p** 而被传递进 **reschedule_idle** 中的。这个函数试图把新近唤醒的进程在一个不同的 CPU 上进行调度——即一个空闲的 CPU 上。

26225：这个函数的第一部分在 SMP 和 UP 场合中都是适用的。它将使高优先级的进程得到占用 CPU 的机会，同时它也会为那些处于饥饿状态的进程争取同样的机会。如果该进程是实时的或者它的动态优先级确实比当前占有 CPU 进程的动态优先级要高某个量级（强制选定的），该进程就会被标记为重新调度以便它能够争取占用 CPU。

26263：现在来到 SMP 部分，它仅仅适用于在上述测试中失败了的那些进程——虽然这种现象经常发生。**reschedule_idle** 必须确定是否要在另一个 CPU 上尝试运行该进程。

正如在对 **schedule** 的讨论中所提到的那样，一个进程的 **avg_slice** 成员是它对 CPU 使用的加权平均值；因此，它说明了假如该进程继续运行的话是否它可能要控制 CPU 一段相对来说较长的时间。

26264：这个 **if** 条件判断的第二个子句使用 **related** 宏（就在本函数之上的第 26218 行）来测试是否 CPU 都在控制着——或想要控制——内核锁。如果是这样，那么不管它们生存于何处，都将不大可能同时运行，这样把进程发送到另一个 CPU 上将不会全面提高并行的效能。因此，假如这条子句或者前一条子句被满足，函数将不会考虑使进程在另一 CPU 上进行调度并简单的返回。

26267：否则，**reschedule_idle_slow**（接下来讨论）被调用以决定是否进程应当被删除。

reschedule_idle_slow

26157：正如注释中所说明的，**reschedule_idle_slow** 试图找出一个空闲 CPU 来贮存 **p**。这个算法是基于如下观察结果的，即 **task** 数组的前 n 项是系统的空闲进程，机器的 n 个 CPU 中每个都对应一个这样的空闲进程。这些空闲进程当（且仅当）对应 CPU 上没有其它进程需要处理器时才会运行。如果可能，函数通常是用 **hlt** 指令使 CPU 进入低功耗的“睡眠”状态。

因此，如果有空闲 CPU 存在的话，对任务数组的前 n 个进程进行循环是找出一个空闲 CPU 所必须的。**reschedule_idle_slow** 函数只需简单的查询每个空闲进程是否此刻正在运行着；如果是这样，它所在的 CPU 就一定是空闲的，这就为进程 **p** 提供了一个很好的候选地点来运行。

当然，这个被选中的明显空闲的 CPU 完全有可能只是暂时空闲而且必定会被一堆拥有更高优先级的，CPU 绑定的进程所充满，这些进程可能在一纳秒后就会被唤醒并在该 CPU 上运行。所以，这并不是完美的解决方法，可是从统计的角度来说它已经

相当好了——要记住，像这样的选择是很符合调度程序“快餐店式（quick-and-dirty）”的处理方式的。

26180：建立本地变量。**best_cpu** 是此时正在运行的 CPU；它是“最佳”的 CPU，因为 **p** 在其上会避免缓冲区溢出或其它的开销麻烦。**this_cpu** 是运行 **reschedule_idle_slow** 的 CPU。

26182：**idle** 和 **tsk** 将沿 **task** 数组进行遍历，**target_tsk** 将是所找到的最后一个正在运行的空闲进程（或者假如没有空闲进程它就为 **NULL**）。

26183：**i** 从 **smp_num_cpus**（前边被叫作 n）开始并且在每一次循环后都递减。

26189：假如这个空闲进程的 **has_cpu** 标志被设置，它就正在它的 CPU 上运行着（我们将称这样的 CPU 为“目标（target）CPU”）。如果该标志没有被设置，那么目标 CPU 就正被某个其它进程占用着；因而，它也就不是空闲的，这样 **reschedule_idle_slow** 将不会把 **p** 发送到那里。刚刚提及问题的反面在这里出现了：现在仅因为 CPU 不空闲并不能表示它所有的进程都不会死亡而使其空闲下来。可是 **reschedule_idle_slow** 无法知道这种情形，所以它最好还是假定目标 CPU 将要被占用一段时间。无论如何，这都是可能的，就算并非如此，某个其它的进程也将很快会被调度到另一个空闲 CPU 上运行。

26190：不过假如 CPU 目标就是当前 CPU，它就会被跳过。这看来很怪，不过无论怎样这都是“不可能发生”的情况：一个空闲进程的 **counter** 是负值，在第 26226 行的测试将早已阻止这个函数执行到这一步了。

26192：找到一个可用的空闲 CPU；相关的空闲进程被保存在 **target_tsk** 中。

既然已找到了空闲 CPU，为什么现在不中断循环呢？这是因为继续循环可能会发现 **p** 当前所在的处理器也是空闲的，在两个 CPU 都空闲时，维持在当前处理器上运行要比把它送往另一个好一些。

26193：这一步 **reschedule_idle_slow** 检查是否 **p** 所在的处理器空闲。如果刚才找到的空闲 CPU 就是 **p** 所在的，函数将向前跳转到 **send** 标记处（26203 行）来在那个 CPU 上对 **p** 进行调度。

26199：函数已经转向另一个 CPU；它要递减。

26204：如果循环遍历了所有空闲的 CPU，该 CPU 的空闲任务就被标记为重新调度并且 **smp_send_reschedule**（26205 行）会给那个 CPU 发送一个 IPI 以便它可以重新对其进程进行调度。

正如读者所见到的，**reschedule_idle_slow** 是 CPU 之间协调无需在 UP 系统中所进行的工作的典范示例。对于 UP 机器来说，询问进程应占有哪一个 CPU 和询问它是否应拥有系统的唯一的一个 CPU 或根本不应该占有 CPU 是等价的。SMP 机器必须花费一些代价来决定系统中哪一个 CPU 是该进程的最佳栖身之所。当然，换来的速度极大提高使得这些额外的努力还是相当合算的。

release

22951：**release** 中非 SMP 特有的部分在第 7 章中已经介绍过了——在这里，一个僵进程（zombie）将被送往坟墓，而且其 **struct task_struct** 将被释放。

22960：查看是否该进程拥有一个 CPU。（拥有它的 CPU 可能还没有清除这个标志；但是它马上就将执行这个操作。）如果没有，**release** 退出循环并像往常一样接着释放 **struct task_ struct** 结构体。

22966：否则，**release** 等待进程的 **has_cpu** 标志被清除。当它被清除后，**release** 再次进行尝试。这种貌似奇特的情况——某进程正被删除，然而它仍占有 CPU——确实少见，

不过并非不可能。进程可能已经在一个 CPU 上被杀死，而且这个 CPU 还没来得及清除 **has_cpu** 标志，但是它的父进程已经正在从另一个 CPU 对它进行释放了。

smp_local_timer_interrupt

对于 UP 专有的 **update_process_times** 函数（27382 行）来说，这个函数就是它在 SMP 上的对应。该函数能够完成 **update_process_times** 所完成的所有任务——更新进程和内核在 CPU 使用方面的统计值——以及其它的一些操作。与众不同的地方在于拥有这个特性的 SMP 版本并没有被添加到一个 UP 函数中去，而是采用了一个具有同样功能，但却完全分离的功能程序。在浏览了函数之后，我们就能够很容易的知道这是为什么了——它与 UP 版本差别甚大到以至于试图将二者融为一体都将是无意义的。**smp_local_timer_interrupt** 可从两个地方进行调用：

- 从 **smp_apic_timer_interrupt**（5118 行）调用，它用于 SMP 的时钟中断。这是通过使用在第 1856 行定义的 **BUILD_SMP_TIMER_INTERRUPT** 宏于第 919 行建立起来的。
- 从第 5776 行通常的 UP 时钟中断函数里进行调用。只有当在 UP 机器上运行 SMP 内核时此种调用方式才会发生。

smp_local_timer_interrupt

5059： **prof_counter**（4610 行）用于跟踪到更新进程和内核统计值之前内核应该等待多长时间；如果该计数器还没有到达 0，控制流程会有效地跳转到函数的末尾。正如代码中所证明的，**prof_counter** 项目从 1 开始递减计数，除非由根（root）来增加这个值，因此在缺省情况下每次时钟滴答都要完成此项工作。然后，**prof_counter[cpu]** 从 **prof_multiplier[cpu]**处被重新初始化。

明显的这是一个优化的过程：每次时钟滴答都在这个 **if** 语句块里完成所有工作将相当的缓慢，所以我们可能想到以牺牲一些精确度的代价将工作分批完成。因为乘法器是可调的，所以你可以指定你所需要的速度频率来放松对准确度的要求。

然而，关于这段代码我总感到有些困惑：确定无疑的是，当 **prof_multiplier[cpu]**耗尽时，统计值应该被更新，就像 **prof_multiplier[cpu]**的计数流逝一样——既然它们已经如此。（除了 **prof_multiplier[cpu]**本身刚刚被改变时，不过这已经偏离了这里讨论的主题。）与此不同的是，这里代码表现出来的就好像只经过了一次滴答计数。或许其用意是为了以后能把记录下来的滴答数目和 **prof_multiplier[cpu]**在某个地方相乘，不过现在并没有这样实现。

5068： 当时钟中断被触发时假如系统正在用户模式运行，**smp_local_timer_interrupt** 会假定全部滴答都是在用户模式里流逝的；否则，它将假定全部滴答是在系统模式里流逝的。

5073： 用 **irq_enter**（1792 行）来夺取全局 IRQ 锁。这是我们要分批处理这项工作的另一个原因：并不需要在每次时钟滴答时都要得到全局 IRQ 锁，这有可能成为 CPU 之间争夺的一个重要根源，实际中函数是以较低的频度来争取该锁的。因此，函数不经常夺取这个锁，可是一旦它获得了锁，就不会再使其被锁。在此我们又一次以准确度的代价换来了这种效率上的提高。

5074： 不用为保存空闲进程的统计值而操心。这样做只会浪费 CPU 的周期。总之，内核会跟踪系统处于空闲的总共时间，对空闲进程的更多细节进行统计价值不大（比如我

们知道它们总是在系统模式下执行的，所以就没有必要再明确计算它们的系统时间了）。

5075： **update_process_times** 和 **smp_local_timer_interrupt** 在这一点上是一致的：它们都调用 **update_process_times** 来完成对单进程 CPU 使用统计的更新工作。

5077： 减少进程的 **counter**（它的动态优先级），如果它被耗尽就重新调度该进程。

5082： 更新内核的统计数字。如在 **update_process_times** 中一样，用户时间既可以用内核的“最优时间”也可以用常规的用户时间来计算，这要取决于进程的优先级是否低于 **DEF_PRIORITY**。

5094： 重新初始化 CPU 的 **prof_counter** 并释放全局 IRQ 锁。该工作必须要以这种顺序完成，当然——若以相反的方式，则可能在 **prof_counter** 被重新初始化之前发生又一次时钟中断。

lock_kernel 和 unlock_kernel

这两个函数也有专门适应于 x86 平台的版本；但是在这里只介绍通用版本。

lock_kernel

10174：这个函数相当简单，它获得全局内核锁——在任何一对 **lock_kernel/unlock_kernel** 函数里至多可以有一个 CPU。显然这在 UP 机器上是一个空操作（no-op）。

10176：进程的 **lock_depth** 成员初始为–1（参见 24040 行）。在它小于 0 时（若小于 0 则恒为-1），进程不拥有内核锁；当大于或等于 0 时，进程得到内核锁。

这样，单个进程可以调用 **lock_kernel**，然后在运行到 **unlock_kernel** 之前可能又将调用另一个要使用 **lock_kernel** 的函数。在这种情况中，进程将立刻被赋予内核锁——而这正是我们所期望的。

其结果是，一旦增加进程的 **lock_depth** 就会使 **lock_depth** 为 0，那么进程以前就是没有锁的。所以，函数在此情形下获得 **kernel_flag** 自旋锁（3587 行）。

unlock_kernel

10182：同样的，如果丢弃内核锁就会使 **lock_depth** 低于 0 值，进程退出它所进入的最后一对 **lock_kernel/unlock_kernel** 函数。此时，**kernel_flag** 自旋锁一定要被解锁以便其它进程可以给内核加锁。通过测试结果的符号位（即使用“<0”而不是“== -1”）可以使 gcc 生成更高效的代码，除此之外，这还可能有利于内核在面对不配对的 **lock_kernel/unlock_kernel** 时可正确执行（或者不能，这取决于具体情况）。

softirq_trylock

你可能能够回忆起在第 6 章的讨论中，**softirq_trylock** 的作用是保证对于其它程序段来说下半部分代码（bottom half）是原子操作——也就是说，保证在任何特定时段的整个系统范围之内至多只有一个下半部分代码在运行。对于 UP 来说这相当容易：内核只不过需要检查或者还要设置一下标志位就可以了。不过对于 SMP 来说自然没有这样简单。

softirq_trylock

12528：测试并设置（tests-and-sets）**global_bh_count** 的第 0 位。尽管读者可能会从 **global_bh_count** 的名字上得到另外一种看法，实际它总是 0 或者 1 的——这样的考虑是适当的，因为至多运行一个下半部分程序代码。不管怎样，如果 **global_bh_count** 已经是 1 了，那么就已经有一个下半部分代码在运行着，因此控制流程就跳转到函数末尾。

12529：如果还可得到 **global_bh_lock**，那么下半部分代码就能够在这个 CPU 上运行。这种情况与 UP 机器上使用的双锁系统非常类似。

12533：**softirq_trylock** 无法获取 **global_bh_lock**，因此它的工作失败了。

cli 和 sti

正如在第 6 章中解释过的，**cli** 和 **sti** 分别用于禁止和启用中断。对于 UP 这简化为单个 **cli** 或 **sti** 指令。而在 SMP 情况下，这就很不够了，我们不仅需要禁止本地 CPU 还要暂时避免其它 CPU 处理 IRQ。因此对于 SMP，宏就变成了对**__global_cli** 和**__global_sti** 函数的调用。

__global_cli

1220： 把 CPU 的 EFLAGS 寄存器复制到本地变量 **flags** 里。

1221： x86 系统里的中断使能标志在 EFLAGS 寄存器的第 9 位——在第 1205 行解释了 EFLAG_IF_SHIFT 的定义。它被用来检测是否已经禁止了中断，这样就不再需要去禁止它们了。

1223： 禁止这个 CPU 的中断。

1224： 如果该 CPU 没有正在对 IRQ 进行处理，**__global_cli** 就调用 **get_irqlock**（1184 行）来获得全局 IRQ 锁。如果 CPU 已经在对 IRQ 进行处理了，那么正如我们马上要看到的，它已经拥有了该全局 IRQ 锁。
现在本 CPU 已经禁止了中断，而且它也拥有了全局 IRQ 锁，这样任务就完成了。

__global_sti

1233： 如果 CPU 没有正在对 IRQ 进行处理，**__global_sti** 就在**__global_cli** 中通过 **release_irqlock**（10752 行）调用来实现对全局 IRQ 锁的释放工作。如果 CPU 已经在对 IRQ 进行处理了，那么它已经拥有了该全局 IRQ 锁，正如在接下来的部分中将要解释的那样，这个锁将在其它地方被释放掉。

1235： 再次允许在本 CPU 上进行中断。

irq_enter 和 irq_exit

第 6 章中顺便提及了这两个函数的 UP 版本。包含在一对 **irq_enter/irq_exit** 之中的代码段都是原子操作，这不仅对于其它这样的代码区域是原子的，而且对于 **cli/sti** 宏对来说也是如此。

irq_enter

1794： 调用 **hardirq_enter**（10761 行）自动为本 CPU 增加全局 IRQ 计数和本地 IRQ 计数。这个函数记录了 CPU 正在处理一个 IRQ 的情况。

1795： 执行循环直到这个 CPU 得到全局 IRQ 锁为止。这就是为什么我要在前面说明如果 CPU 正在处理 IRQ，那么它就已经获得了全局 IRQ 锁的原因：到这个函数退出时，这两个特性都将被加强。对于内核代码来说，把这两个特性分离出去并没有太大的意义——它可以直接调用 **hardirq_enter**，而且也不用去争夺全局 IRQ 锁。函数只是没有这样作而已。

irq_exit

1802： 这个函数转向 **hardirq_enter** 的相反函数 **hardirq_exit**（10767 行）。顺便要提及的是，对 **irq_enter** 和 **irq_exit** 来说其 **irq** 参数都被忽略了——至少在 x86 平台上如此。

第 11 章 可调内核参数

遵循 Unix 的 BSD 4.4 版本所倡导的风格，Linux 提供 **sysctl** 系统调用以便在系统运行过程中对它所拥有的某些特性进行检查和重新配置，它并不需要你编辑内核的源代码、重新编译，然后重启机器。这是对早期 Unix 版本的一个十分重要的改进，在早期版本里调整系统经常是令人头痛的琐碎事务。Linux 把可以被检查和重新配置的系统特性有机地组织成了几个种类：常规内核参数、虚拟内存参数、网络参数，等等。

同样的特性也可以从一个不同的接口进行访问：/proc 文件系统。（因为它真正的是系统的一个透视区（window）而不只是真实文件的一个容器，所以/proc 是一个“伪的文件系统”，不过那是一个蹩脚的词汇，而且无论如何这个区别在此并不重要。）每种可调内核参数在/proc/sys 下都表现为一个子目录，而每个单独的可调系统参数由某个子目录下的一个文件来代表。这些子目录可能又包含一级子目录，它们仍然含有更多的代表可调系统参数的文件和子目录，等等，但是这种嵌套级数从来都不会很深。

/proc/sys 绕过了通常的 **sysctl** 接口：一个可调内核参数的值可以简单的通过读取相应的文件来得到，通过写入该文件可以设置它的值。普通 Unix 文件系统的许可被应用于这些文件，以便对能够对它们进行读写的用户进行控制。大多数文件对所有用户是可读的但是只对 root（根用户）可写，不过也有例外：比如，/proc/sys/vm 下的文件（虚拟内存参数）只能被 root 来读写。如果不使用/proc/sys，检查和调整系统将需要编写程序并使用必须的参数调用 **sysctl**——虽然不是任务艰巨的劳动，可是也比不上使用/proc/sys 来得方便。

struct ctl_table

18274：这是本章涉及的代码中所使用的一个主要数据结构。**struct ctl_table**s 通常是由数组聚合起来的，每个这样的数组对应于/proc/sys 下某处一个单独目录里的条目。（依我之见，称它为 **struct ctl_table_entry** 可能更好。）**root_table**（30328 行）以及在它之后的数组通过 **struct ctl_table** 的 **child** 指针连结节点而形成了一个数组树（**child** 将在下边的列表中介绍）。注意所有这些都是 **ctl_table** 的数组，它只是为 **struct ctl_table** 进行 **typedef**；18184 行完成这项工作。

图 11.1 示意出了数组树间的关系。这幅图显示了由 **root_table** 形成的树的一小部分以及它所指向的树。

struct ctl_table 具有如下成员：

- **ctl_name**——是唯一标识表项的一个整数——在它所在的数组中是唯一的；这个数字在不同的数组中是可以重用的。数组的任何一项都已经存在这样一个唯一的数字了——就是它的数组下标——可是这个数字不能被用于该目的，因为我们想要维护不同内核发布版本中的二进制兼容性。与某内核版本里一个数组项相关联的可调内核参数可能不会出现在将来的内核版本里，所以假如参数是被它们的数组下标定义的，对数组里废弃项目位置的重新使用将使还没有在新内核版本下编译过的程序变得混乱。随着时间的推移，为了向后兼容而带上的只浪费空间但没有作用的元素项将会使数组变得乱七八糟。相反的，这种方法只会“浪费”整数，而整数资源却无疑是非常丰富的。（另一方面，查找也会更慢，因为一个简单的数组下标还不足以满足这种方法。）

  要注意的是这与有系统调用的情形相当类似：每个系统调用都与一个在系统调

用表里唯一标识它位置的数字相关联。但是在这种情况里使用了一个不同的解决办法，可能由于速度在此并不重要的缘故。

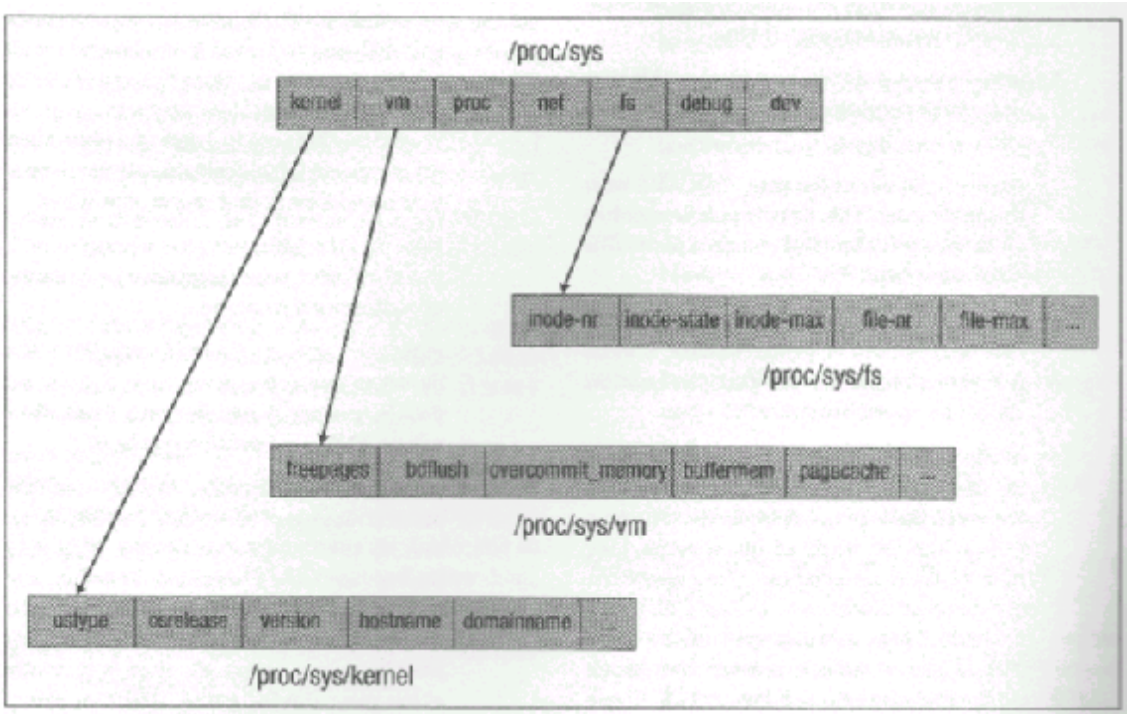

图 11.1 **ctl_table** 树的一部分

struct ctl_table 具有如下成员：

- **ctl_name**——是唯一标识表项的一个整数——在它所在的数组中是唯一的；这个数字在不同的数组中是可以重用的。数组的任何一项都已经存在这样一个唯一的数字了——就是它的数组下标——可是这个数字不能被用于该目的，因为我们想要维护不同内核发布版本中的二进制兼容性。与某内核版本里一个数组项相关联的可调内核参数可能不会出现在将来的内核版本里，所以假如参数是被它们的数组下标定义的，对数组里废弃项目位置的重新使用将使还没有在新内核版本下编译过的程序变得混乱。随着时间的推移，为了向后兼容而带上的只浪费空间但没有作用的元素项将会使数组变得乱七八糟。相反的，这种方法只会“浪费”整数，而整数资源却无疑是非常丰富的。（另一方面，查找也会更慢，因为一个简单的数组下标还不足以满足这种方法。）
  要注意的是这与有系统调用的情形相当类似：每个系统调用都与一个在系统调用表里唯一标识它位置的数字相关联。但是在这种情况里使用了一个不同的解决办法，可能由于速度在此并不重要的缘故。
- **procname**——是用于/proc/sys 下的相应项的一个可供我们阅读的简短文件名。
- **data**——一个指向与此表项关联的数据的指针。它通常指向一个 **int** 或者一个 **char**（当然，指向 **char** 的指针是字符串）。
- **maxlen**——可以读取或者写入 **data** 的最大字节数。如果 **data** 指向一个单精度型的 **int**，举例来说，**maxlen** 就应该是 **sizeof（int）**。
- **mode**——Unix 类型的文件许可位，它对应于这一项的/proc 文件（或目录）。对此的解释需要少量文件系统的内容。就像其它 Unix 的实现一样，Linux 使用三

个三元组，其中每一位都记录一个文件许可（在 **ls -l** 命令产生的列表里它们表现为 r、w，和 x 的三组字母）——参见图 11.2。它们占据 **mode** 的低端 9 位。文件系统把文件的 **mode** 里剩余的位留作它用，比如用来跟踪是否文件是常规文件（第 16 位，当它如此时）、目录（第 15 位）、setuid 或 setgid 执行程序（第 12 和 11 位），等等。不过就本章的目的来说，那些其它位都不是我们所关心的内容。

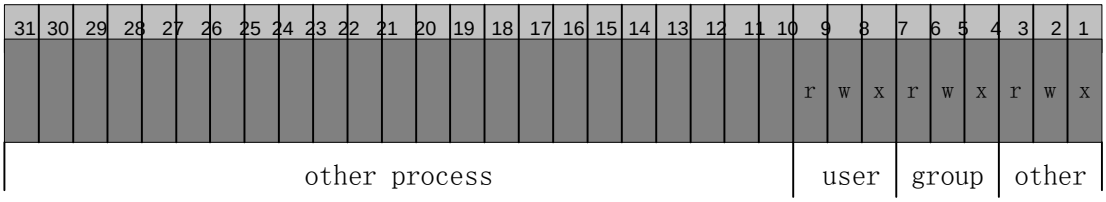

图 11.2 文件的 **mode** 位

这种方式的结果是，读者将经常见到八进制的常数 004、002，和 001 与 **mode** 一起使用——它们分别是在移位 **mode** 后可能得到的适当的三位组中检测读（r）、写（w），和执行（x）位。这种移位和检查工作基本上是在 30544 行的 **test_perm** 里完成的。

注意如果一个表项的 **maxlen** 是 0，那么不管它的 **mode** 是什么，从最终效果上看它都是既不可读也不可写的。

- **child**——如果这是一个目录类型的条目，那么它就是指向子表（child table）的一个指针。在这样的情况下，因为没有数据与此条目相关联，**data** 将是 **NULL**，而 **maxlen** 则将是 0。
- **proc_handler**——指针，指向对 **data** 成员实际进行读取和写入操作的一个函数；它在通过/proc 文件系统读写数据时被使用。以这种方法，任何类型的数据都可以通过 **data** 来进行指向，而且 **proc_handler** 函数会正确的处理对它的工作。**Proc_handler** 通常指向 **proc_dostring** 函数（30820 行）或 **proc_dointvec** 函数（30881 行）；这两个以及其它被普遍适用的函数将在本章后面被讨论。（当然，任何具有适当原型（prototype）的函数都可以使用。）对于目录类型的条目，**proc_handler** 是 **NULL**。
- **strategy**——指针，指向对 **data** 成员实际进行读取和写入操作的另一个函数；它使用在通过 **sysctl** 系统调用进行读写的时候。它通常是 **sysctl_string**（31121 行），不过也可以是 **stringctl_intvec**（31163 行）；这两个函数在本章后面进行讨论。出于种种原因，大多数可调内核参数是通过/proc 接口而不是 **sysctl** 系统调用进行调整的，所以这个指针是 **NULL** 会比非空更为常见。
- **de**——指向 **struct proc_dir_entry** 的一个指针，它在/proc 文件系统代码中使用以追踪文件系统里的文件或目录。如果它非空，**struct ctl_table** 就在/proc 下的某处注册过了。
- **extra** 和 **extra2**——指向在处理这个表元素时所需的任何补充数据。它们当前只用于指定某些整数参数的最小和最大值。

/proc/sys 支持

不是所有实现用于可调内核参数/proc/sys 接口的代码都包括在这本书中——的确，大部分代码并没有包括在内，因为它们基本上属于/proc 文件系统本身。尽管如此，只要你不关心/proc 剩下的部分是如何工作的，就不难理解在 kernel/sysctl.c 里的代码，它们与/proc 文件系统一起工作用来使/proc 下的可调内核参数是可见的。

register_proc_table

30689：**register_proc_table** 函数在/proc/sys 下注册一个 **ctl_table**。注意这里并不要求所提供的表是根一级的节点（即 **ctl_table** 没有双亲）——它本应该是，不过这取决于调用者是否能够进行保证。

这个表被直接建立在 **root** 之下，它应该对应于/proc/sys 或者其下的一个子目录。（在初次调用时，**root** 总是指向 **proc_sys_root** 的，但是在递归调用时它的值改变了。）

30696：开始在 **table** 数组的所有元素中进行循环；在当前元素的 **clt_name** 成员为 0 时循环结束，表示这是数组的末尾。

30698：如果 **ctl_table 的 procname** 元素是 **NULL**，那么即使同一数组的其它元素都可以为用户所见，它也不可以在/proc/sys 下被用户所见。这样的数组元素会被跳过。

30701：如果表项有 **procname**，表明它应该在/proc/sys 下被注册，那么它一定还有一个 **proc_handler**（如果是一个叶子，或文件类型的节点）或者一个 **child**（如果是一个目录类型的节点）。如果它同时缺少这两者，那么系统将显示一条警告，而后循环继续进行。

30711：若表项有一个 **proc_handler**，它被标记成常规文件。

30713：否则，正如可从第 30701 行推断的那样，它一定有一个非空的 **child**，这样该条目将被看作是一个目录。注意并没有禁止 **ctl_table** 同时拥有非空 **proc_handler** 和 **child** 这两者——在这种情形下，所有代码将对其一视同仁。

30715：用给定的名字搜索一个存在的子目录，如果找到就让 **de** 指向它，如果没找到则 **de** 为 **NULL**。为什么对文件不做类似的检查比较难于理解——这可能是我没有领会的文件系统的某个细节问题，答案无疑就在那里。

30723：如果指定的子目录已经不存在了，或者假如 **table** 对应于一个文件而不是一个目录，新的文件或者目录就会通过调用 **create_proc_entry**（未包含在本书中）来创建。

30728：如果表项是一个叶子节点，**register_proc_table** 会告诉文件系统代码使用由 **proc_sys_inode_operations**（30295 行）定义的文件操作。**proc_sys_inode_operations** 只定义了两个操作，读和写（不是搜索、内存映射，或者其它）。这些操作是用 **proc_readsys** 和 **proc_writesys** 函数（30802 和 30808 行）来执行的，在本章的后面章节中将对它们进行介绍。

30731：到了这一行，**de** 就不可能是 **NULL** 了——它或者已经非空或者在第 30723 行被初始化了。

30733：如果增加的条目是目录类型，**register_proc_table** 会被递归调用来增加这一项的所有子孙。这是内核里不多见的一次递归调用。

unregister_proc_table

30739：**unregister_proc_table** 函数删除 **ctl_table** 数组树和/proc 文件系统之间的关联。**ctl_table** 里的条目以及它们下面所有的“子目录”里的条目也将会从/proc/sys 消失。

30743：同第 30396 行一样，这一行开始在给定的表项数组上进行循环。

30744：与/proc/sys 下任意条目都不关联的表项具有一个为 **NULL** 的 **de** 成员；显然这些表项可被忽略。

30748：如果/proc 文件系统认为这是一个目录，但表项是一个叶子（非目录），这两个结构就是不一致的。**unregister_proc_table** 就会显示一条警告并继续循环，而不会移去这一项。

30752：目录被逐层的进行释放——内核中另一次并不多见的递归过程。

30756：在递归调用结束之后，**unregister_proc_table** 检查是否所有子目录和文件都被逐层删除了——如果不是，当前元素就不能被安全的移去，接着要继续循环。

30762：这里就是为什么子目录（以及其中的文件）可能还没有被移去的原因：它们可能当前还正被使用着。如果这个元素正在被使用，循环将继续，这样该元素就不会被移走。

30765：节点通过 **proc_unregister**（本书不进行介绍）从文件系统里被删除，接着用于追踪该节点而分配的内存被释放。

do_rw_proc

30771：**do_rw_proc** 实现 **proc_readsys**（30802 行）和 **proc_writesys**（30806 行）函数的核心部分，这两个函数被/proc 文件系统代码用于对 **ctl_table** 执行读取和写入操作。

30782：确保一个表与/proc/sys 下的这一条目相关联。

30785：注意这一行的第一个测试与第 30782 行的第二个测试是相重复的，这是因为 **table** 是从 **de->data** 初始而来。

30788：确保调用进程有适当的读或写权限。

30795：调用该表项的 **proc_handler** 来完成真正的读操作或写操作。（要注意第 30785 行证实了 **proc_ handler** 成员是非空的。）如前所述，**proc_handler** 成员通常是 **proc_dostring** 或 **proc_ dointvec**（30820 行和 30792 行），在随后的几段中我们将对它们进行讨论。

30799：**do_rw_proc** 返回实际读取或写入的字节数。注意到本地变量 **res** 完全是多余的；它可以被参数 **count** 所替代。

proc_dostring

30820：**proc_dostring** 是供文件系统代码调用以对 C 语言字符串型的内核参数进行读取或写入操作的函数。

注意 **write** 标志表示调用者正在写表元素，不过这主要是涉及从输入缓冲区里进行读取——因此，用来写入的代码是受读控制的。类似的，如果 **write** 未被设置，调用者正从该表项读取，这里主要涉及的是写入给定的缓冲区。

这个函数在第 31085 行还实现了一个存根程序（stub）；这个存根程序在/proc 文件系统被编译出内核时使用。大多数其它函数中的类似存根程序将在这个存根程序之后被介绍。

30835：从输入缓冲区内读取字符直到一个表示结束的 ASCII NUL（0）字节或者发现新的一行，再或者到达了被允许从该输入缓冲区内读出数据的最大值（被 **lenp** 所指定）为止。（为了不引起混淆，牢记 **NULL** 是一个 C 指针常量，而 NUL——只有一个 L——是 ASCII 用于字符数字 0 的术语。）

30842：如果从缓冲区读出的字符数超出了可在表项里存储的限度，该数目会被降低。在循环之前就限制最大输入长度（**lenp**）可能会更高效，因为不管怎样从 **buffer** 里读取

大于 **table->maxlen** 字节的数据是无意义的。实际上，循环可能读出，假设是 1024 字节，然后降低计数到 64，因为表项里只能存储这么多。

30844：该字符串从输入缓冲区里被读出，然后以 NUL 结束。

30847：内核为每个进程所拥有的每个文件维护一个“当前位置”变量；这就是 **struct file** 的 **f_pos** 成员。它是 **tell** 系统调用返回的值并由 **seek** 系统调用进行设置。因此，文件的当前位置是由写入的字节数所推进的。

proc_doutsstring

30871：在获得 **uts_sem** 信号量后（29975 行），**proc_doutsstring** 仅是调用 **proc_dostring**。这个函数被 **kern_table**（30341 行）里的一些条目用来设置 **system_utsname** 结构体的不同部分（20094 行）。

do_proc_dointvec

30881：**proc_dointvec**（30972 行）把它的工作委托给了该函数。**do_proc_dointvec** 读或写一个被 **table** 的 **data** 成员所指向的 **int** 类型数组。要读写的 **int** 类型数目通过 **lenp** 传递；它通常是 1，所以本函数通常只被用于读写单独一个 **int**。

用于 **int** 的值是被 **buffer** 指定的。这些 **int** 是不会被以一个未经加工的 **int** 数组传递的；相反的，它们以 ASCII 文本给出，而这正是用户写入相关/proc 文件的。

30898：在所有要读写的 **int** 中循环。**left** 追踪调用者想要读写 **int** 的剩余数目，而 **vleft** 追踪 **table->data** 里剩余的有效元素数目。在这二者中任何一个到达 0，或它从半途退出时，该循环结束。

注意如果从循环中去掉第 30899 行的 **if** 语句，可以使整个循环的效率稍微提高一些，尽管这样做的结果较难维护。取代的代码如下：

P556—1

这种方式使得并不在循环内改变的 **write** 的值将只需被检查一次，而不必在每次循环重复检查。

30900：向前搜索一个不是空格的字符，它是输入（缓冲区）里下一个数字的开头。

30913：从用户空间把一大块数据复制到本地缓冲区 **buf**，然后以 NUL 结束 **buf**。现在 **buf** 里包含了所有输入缓冲里剩余的 ASCII 文本——或者是它所能容纳的那些文本。

这种方法看起来不很有效率，原因在于它可能读取的超出了它所需要的。然而，因为 **buf** 的容量仅为 20（**TMPBUFLEN**，30885 行），它就不可能读取比它所需多出许多的数据。这里的思想可能是读入稍多一些数据要比检查每个字节以确定是否应该停止读操作所付出的代价要少些。

计划使 **buf** 足够大来包括任何 64 位整数的 ASCII 表示，以便这个函数不仅可以支持 32 位平台还可以支持 64 位平台。的确，它只能满足最大的正 64 位整数，它有 19 个数位（使终结的 NUL 字节是第 20 个字节）。可是要记住这些是有符号的整数，所以最小的 64 位有符号整数，即-9,223,372,032,854,775,808 也应是合法输入。这个数字无法被正确的读取。但是幸运的是，补救方法工作量不大而且也非常明显。

随后读者就能够看到当这个输入出现时代码将如何对其进行处理。

30919：处理打头的减号（-），如果发现一个减号就跳过它并设置一个标志。

30923：确保从 **buffer** 读取的文本（可能是打头的减号之后的部分）至少是以一个数字开始的，这样它才能顺利的转换为一个整数。若没有这次检查，就不可能分辨出第 30925

行调用 **simple_strtoul** 返回的 0 是因为输入就是“0”还是因为函数无法转换任何文本。

30925：把文本转换为一个整数，用 **conv** 参数换算结果。这个换算步骤对于 **proc_dointvec_jiffies** 这样的函数（31077 行）比较有用，它用乘以常数 **HZ** 的简单手段把它的输入从秒转换为一段时间值（jiffies）。然而一般情况里，这个比例因子是 1——即没有换算。

30927：如果还要从缓冲区读取更多的文本，而且下一个要读的字符不是分割参数的（argument-separating）空格，那么整个参数（argument）就无法装进 **buf**。这样的输入是无效的，所以循环提早结束。（一种可以导致函数处于这种状态的方式就是前边所描述的，输入表示的是最小的有符号 64 位整数。）不过，没有错误代码会被返回，因此调用者可能会错误地认为一切正常。当然这也不完全正确：一个错误代码将在第 31070 行被返回，不过这仅当无效参数是在第一次循环重复中被检测到的时候；如果它在后续的循环里被检测到，错误就不会被注意到。

30929：参数被成功的读取。如果有前导的减号，那么现在就对它进行考虑，其它的本地变量被调整转移到下一个参数上，然后这个参数通过指针 **i** 被存储在表项中。

30936：调用者从表项里读取值——由于无需对 ASCII 文本进行语法分析，这就是一种更为简单的情形。输出是由 tab（制表符）分隔的，所以在除了第一次之外的任何一次循环里都把一个 tab 写入临时缓冲区里（在最后一个参数之后也不用写，只需在参数之间即可）。

30938：接着，当前的整数被 **conv** 因子按比例缩减并打印到临时缓冲区里。这段代码同样会受读者前边已经见到的问题的损害：临时缓冲区 **buf** 的大小可能不足以容纳打印到它里边的全部整数值。在这种情况下，实际问题还会因缓冲区的第一个位置可以是一个 tab 制表符而被恶化。这会使 **buf** 的可用部分减少一个字符，进一步还会降低可被正确处理的输入范围。

在这里过大或过小的整数所造成的结果要比在写入情形里严重的多。在那种情形中，代码只要抛弃一些本应接受的输入即可。而在这儿，**sprintf** 会越过 **buf** 的末尾继续写下去。

然而令人惊讶的是，这正是实际工作中可能发生的。在一次典型的执行过程中将有可能发生如下执行过程：从总体上来说，超过 **buf** 的末尾之后还要写入两个额外的字节（一个是因为它可以写入比预期更长的数字，另一个是 tab 制表符）。在栈里 **p** 通常是紧跟在 **buf** 之后的，所以超出 **buf** 末尾写入的部分将会覆盖 **p**。可是由于 **p** 没有先被重新初始化时它是不会再被使用的，因此暂时覆盖它的值并没有危害。

这是一个看似有趣的事故，但是仅仅通过使 **buf** 稍微大一些就能够成为一个更好的解决方式，这样便于代码为正确的而不是错误的前提（reason）而工作。依照原样，对于 gcc 的代码生成器进行完全合法的很小的修改就能够揭示出潜在的缺陷。

30939：把当前 **int** 的文本型表示复制进输出缓冲区里——或者和它所能容纳的相等的文本——接着更新本地变量使其转移到表项的下一个数组元素。

30949：如果调用者刚才在读取，输出就被新的一行结束。**if** 条件语句也保证循环不会在其第一遍执行而且还有空间来写入新行时就结束。注意输出缓冲区不是用 ASCII NUL 字节（读者可能会这样猜测）来结束的，因为它无需如此：调用者能够利用 **lenp** 被写入新值来减少返回字符串的长度。

30954：如果调用者正向表项里写入数值，则略过从输入缓冲区读取的最后参数之后所有的空格。

30967：更新文件的当前位置和 **lenp**，然后返回 0 表示成功。

proc_dointvec_minmax

30978：**proc_dointvec_minmax** 函数类似于 **do_proc_dointvec**，区别是这个函数还把表项的 **extra1** 和 **extra2** 成员作为可以写入该表项的限制值数组来处理。**extra1** 里的值是最小限度，而 **extra2** 里的值则是最大限度。另一点区别是 **proc_dointvec_minmax** 不使用 **conv** 参数。

因为这两个函数颇为相似，所以这一段里只介绍其不同之处。

31033：最大的区别在于：当写入时，超过被 **min** 和 **max**（在 **extra1** 和 **extra2** 数组上循环得到）所定义的范围之外的值将悄无声息的被略过。这段代码的目的明显是要使 **min** 和 **max** 伴随着 **val** 一起继续。当一个数值从输入缓冲里被读取时，它应该被下一个 **min** 和 **max** 来检查，然后才能决定被接受或被忽略。可是，这并非是实际所发生的那样。

假设从 **buffer** 而来的当前值已经进行了语法分析并存入里 **val**，它小于最小值；为了更具体一些，再假设已是第三遍循环，以便 **min** 和 **max** 分别指向对应数组中的第三个元素。然后 **val** 将用 **min** 来检查并发现它超出了范围（太小），接着循环还要继续。可是 **min** 会作为检查的副作用被更新，而 **max** 则没有。现在，**min** 指向它对应数组的第四个元素了，可是 **max** 仍然指向它的数组的第三个元素。这两者不再同步，而且它们还将保存这种状态，这样在下一个从 **buffer** 里读取的值被检验时采用的就是错误的界限。下列代码是最简单的一种修补程序：

P558—1

正如读者将要在本章后边看到的，现在的 Linux 源代码永远不会暴露出这个缺陷。（未来发行的版本情况将有所不同，尽管还未曾明确写出。）

sysctl 系统调用

用于可调内核参数的另一个接口是 **sysctl** 系统调用，以及相关函数。我不很喜欢这个接口。为什么不呢？对于大部分实际工作目的来说，使用 **sysctl**——不过这种方法比修改源代码的旧方法来调整内核能够获得更大的性能提高——只会比访问/proc 文件更为笨拙。通过 **sysctl** 来进行读写需要 C 程序（或相似的东西），而/proc 却很容易通过外壳（shell）命令（或等价的通过命令解释程序脚本）来进行访问。

另一方面，如果你正在 C 环境下工作，调用 **sysctl** 就比打开文件、读取并/或写入，以及再关闭它要方便的多，所以 **sysctl** 在今后也有它的用武之地。与此同时，还是让我们来看一看它的实现吧。

do_sysctl

30471：**do_sysctl** 实现 **sys_sysctl**（30504 行），即 **sysctl** 系统调用的主要内容。注意 **sys_sysctl** 还在第 31275 行出现过——那个版本只是在 **sysctl** 系统调用被编译出内核时所使用的一个简单的存根程序（stub）函数。

如果 **oldval** 非空就用 **oldval** 返回内核参数原有的值，而它的新值在 **newval** 非空时从 **newval** 来进行设置。**oldlenp** 和 **newlen** 分别标识出有多少字节应被写入 **oldval** 和从 **newval** 读出，这是在相应的指针不是 **NULL** 的时候；当指针为 **NULL** 的时候，它们将被忽略。

要注意这里的不对称性：函数对旧值的长度使用指针，而对新的长度不使用指针。这是因为旧的长度既是输入参数也是输出参数；它的输入值是可以通过 **oldval** 返回

的最大字节数，而它的输出值是实际返回的字节数。与之相反，新的长度只是一个输入参数。

30482：如果调用者需要旧的内核参数值，从 **oldlenp** 来对 **old_len** 进行设置。

30490：开始遍历表树的循环列表。（参见本章随后对 **register_sysctl_table** 的讨论。）

30493：使用 **parse_table**（30560 行，在下一段里讨论）来定位可调内核参数，然后读和/或写它的值。

30495：如果 **parse_table** 分配了所有环境信息，它就被释放。很难准确地说出这个环境信息表示着什么。它不被本书所讨论的任何代码使用——据我所知，它目前甚至没有被内核里的任何代码所使用。

30497：**ENOTDIR** 错误表示没有在这一棵表树中找到指定的内核参数——它可能在另一棵还没有查找过的表树中。否则，**error** 将为某个其它的错误代码，或者是代表成功的 0；无论如何，函数应该返回了。

30499：用 **DLIST_NEXT** 宏（本书对此不做介绍）来增加循环控制变量的值（loop iterator）。

30501：返回 **ENOTDIR** 错误，报告出指定的内核参数在任何一个表里都未找到。

parse_table

30560：**parse_table** 用于在表树里查找一个条目，其方法类同于在一个目录树里解析出一个完全合格的文件名的方法。其思想如下：沿着一个 **int** 数组（数组 **name**）进行查找，并在一个 **ctl_table** 数组里搜索每个 **int**。当找到一个匹配项时，它对应的子孙表就被递归查阅（如果匹配项是目录类型的条目），或者该条目被读和/或写（如果它是文件类型的条目）。

30566：多少有些令人惊讶的是，这一行就开始了对整型数组 **name** 内所有元素的循环。习惯上的方法原本是把从这一行到第 30597 行所有代码用一个 **for** 循环包括起来，它的开始是这样的：

```
for ( ; nlen ; ++name , --nlen , table = table -> child )
```

（这个循环还需要删除第 30567 和 30568 行代码，并用一个语句来替代从 30587 直到 30590 行的代码。）推测起来，可能是实际使用的版本可以生成更好的目标代码吧。

30570：开始循环所有的表项，查找与当前 **name** 匹配的一项；当表已被遍历结束（**table->ctl_name** 为 0 了）或者指定的表项已被找到并处理时本循环结束。

30572：把 **name** 数组的当前项读入 **n** 里，以便它可以与当前表项的 **ctl_name** 进行检查。因为 **name** 在内层循环中没有变化，这个读取操作可以放在循环外边（也就是移至 30569 行）以提高一点速度。

30574：核查是否当前 **ctl_table** 的名字与被找到的名字相匹配，或是否它有特殊的“通配符（wildcard）”值，即 **CTL_ANY**（17761 行）。后者的使用目的还不清楚，因为现在并没有内核源代码的任何一处使用过 **CTL_ANY**。它可能用于将来的方案中——我也不认为它是过去版本的一个遗留物，因为 **CTL_ANY** 在 2.0 内核里也没有被用到，而且整个 **sysctl** 接口也只向后兼容到 2.0 以前的开发树版本。

30576：如果这个表元素有一个孩子，它就是一个“目录”。

30577：遵循标准 Unix 行为，检查目录的 x（可执行）位来判断是否当前进程可被允许对它进行访问。注意到这与文件系统所实现的工作非常类似，虽然这并不是（/proc）文件系统接口。这样可以使这两种接口在施用于可调内核参数时能够得到一致的结果——如果一个用户有通过一种接口来修改某个内核参数的权限而通过另一种却没有该权限，那么将是非常不可思议的。

30579：如果表项有一个策略（strategy）函数，它可能需要覆盖允许该进程进入目录的授权。

这个策略函数将被访问，如果它返回一个非零值，整个查找就被中止。

30587：进入目录。本行有效的继续外层循环，并转移到该名字的下一部分。

30592：这个表节点是一个叶子节点，因此内核参数就被找到了。注意这并不打扰对 **name** 数组是否已到其最后元素的检查（也就是现在 **nlen** 是否为 1），虽然可以证明如果不是那样就会有某类型错误产生。不管哪一种情况，**do_sysctl_strategy**（30603 行）都要负责对当前表元素进行读和/或写操作。

30598：**name** 数组非空，可是它的元素在叶子节点被找到之前均已用完。**parse_table** 就返回 **ENOTDIR** 错误，来表示查找指定节点失败。顺便提及一点，注意前一行里的分号是多余的。

do_sysctl_strategy

30603：**do_sysctl_strategy** 在单独一个 **ctl_table** 里读和/或写数据。计划使用该表元素里的 **strategy** 成员，如果存在的话，来完成读/写工作。如果表元素没有它自己的 **strategy** 例程，某些通用的读/写代码将被替代使用。不过读者将要看到，它并不完全按照计划工作。

30610：如果 **oldval** 非空，调用者将读取旧值，这样 r 位就会在 **op** 里被设置。类似的，如果 **newval** 非空则 w 位被设置。接着，第 30614 行核查许可，如果当前进程缺少所需的授权就返回 **EPERM** 错误。

30617：如果表项有它自己的 **strategy** 例程，这个例程就要处理读/写请求。如果它返回负数——一个错误——这个错误就被传送给调用者。如果返回的是正数，0（成功）就会被传送给调用者。如果是 0，**strategy** 例程就拒绝由它自己来处理请求，取而代之的将是缺省行为。（读者可以设想只返回 0 的 **strategy** 例程，如果它完成一些其它诸如收集被调用次数的统计数据这样的工作，它仍然是有用处的。）

30630：这里是通用读取代码开始的地方。注意 **get_user**（13254 行）的返回值不被检查。（类似的缺陷发生在第 9537 和 31186 行。）

30632：确保不会有多于与该表项的 **maxlen** 成员所指定的数值相等的数据被返回。

30634：通过 **oldval** 从表里复制所要求的数据，再将真正被写的数据总量存储在 **oldlenp** 中。

30642：类似于 **oldlenp**，要确保写入表项的数据不能多于它的 **maxlen** 成员所允许的值。注意如果 **copy_from_user** 在中途的第 30644 行检测到一个错误，**tabel->data** 可能会在仅仅被部分更新的情况下就结束。

30648：返回 0 表示成功。以下三种情况都可以达到这一点：

- 调用者对这个表项既不读也不写。
- 调用者尝试读和/或写这个表项，而且所有步骤都被成功执行。
- 表项没有关联的数据，或者因为它的 **maxlen** 是 0，所以它是只读的。

三种情形中的第一种有点儿奇怪，而最后一种则更令人奇怪。第一种情况有些奇特是因为调用 **sysctl** 却要求它对指定的表项既不读也不写，这并没有多少意义，所以可以正当的把它当作一个错误来处理。尽管如此，它要与其它系统调用的内核实现保持基本一致，那就是把一个无操作请求不看作是一个错误。比如说，在第 8 章中介绍的 **sys_brk**（33155 行）在由调用者指定的新 **brk** 值与旧值相同时并不产生一个错误信号。

第三种情况要比第一种奇怪一些，因为它可能真的反映着一个错误。例如，调用代码尝试写入一个 **maxlen** 是 0 的参数，而且由于系统调用返回成功值而认为该尝试已被完成。看起来事情好像不是这样，因为不管怎样为 0 的 **maxlen** 都会使该条目失效，不过还真的存在一个 **maxlen** 为 0 的表项——参见第 30380 行。最终，一切都归结为

sysctl 是怎样在文档中描述的，但是 man 帮助程序中却对此没有任何记载。我仍然认为 **do_sysctl_strategy** 在这种情况下应该返回一个 **EPERM** 错误。

register_sysctl_table

30651：把一个新的根已经被给出的 **ctl_table** 树插入到其它树所形成的循环链表里。

30655：分配一个 **struct ctl_table_header** 用来管理新树的信息。

30659：把新的首部（跟踪 **ctl_table**s 数组形成的新树）插入到首部组成的链表里。

30666：调用 **register_proc_table**（30689 行，本章前边讨论过）把新的表树注册在/proc/sys 目录下。如果没有内核在没有/proc 文件系统支持的情况下进行编译时，则这一行将被编译到内核以外。

30688：新分配的首部被返回给调用程序，以便调用程序能够在以后通过把该首部转递给 **unregister_sysctl_table**（30672 行）来删除相应的树。

unregister_sysctl_table

30672：如前所述，这个简单函数只是把一个 **ctl_table** 的树从内核里这样的树所组成的循环链表里删除。如果内核是在支持/proc 的情况下编译的，它也用于从/proc 文件系统里删除相应的数据。

回顾一下第 30490 和 30500 行，读者不难发现 **root_table_header**（30256 行）——对应于 **root_table** 的列表节点——是在遍历树的循环链表时被用作头和尾节点的。读者现在能够明白实际上在 **unregister_sysctl_table** 函数里没有什么可以避免 **root_table_header** 被从表头列表里删除——它只是还没有这样做而已。

sysctl_string

31121：**sysctl_string** 是 **ctl_table** 的策略例程之一。回忆一下，策略例程可以从第 30618 行（在 **do_sysctl_strategy** 里）被调用来有选择的覆盖一个表项的缺省读/写代码。（策略例程也可以从第 30580 行被调用，不过该例程却从不会被调用。）

31127：如果该表没有相关数据，或者如果可访问部分的长度是 0，则返回 **ENOTDIR** 错误。这与 **do_sysctl_strategy** 的做法是不一致的，在同样的情况里它返回的是成功。

31138：当前字符串的值被复制到用户空间，然后结果以 NUL 来结束（这意味着比由 **lenp** 指定值多一个字节的数据可能会被复制——依据文档记录，这可能是一个缺陷）。因为当前值已经是 NUL 结束的，这四行代码可以被简化为两行：

```
if ( copy_to_user ( oldval , table -> data , len + 1 ) )
  return –EFAULT ;
```

这种改变的正确性部分上依赖于当写入 **table->data** 时代码剩余部分所遵循的三个特征：

- 代码剩余部分不能把多于 **table->maxlen** 的 **char** 数据复制进 **table->data** 里。（这也使得第 31136 行的测试变得没有必要。即使还需要该测试，那也只用检查>，而不用检查>=了。）
- 然后 **table->data** 以 NUL 来结束，如果必要就复写最后一个拷贝进来的字节，以便包括 NUL 在内的总长度不大于 **table->maxlen**。
- **table->maxlen** 永不发生变化。

因为所有三个特征都有效，所以在第 31138 行 **len** 将总是严格小于 **table->maxlen**，而且结束 NUL 字节一定会在 **table->data[len+1]**或之前的位置出现。

31146：与前边的情况类似，从用户空间中复制新值，而且结果以 NUL 来结束。不过在这种情形下，不从用户空间复制 NUL 字节是一种正确的做法，因为把它从用户空间复制进来要比仅仅在 data 的适当字节安排一个 NUL 效率低。而且以这种方式，即使输入不是 NUL 结束的，**table->data** 也要如此。当然，从 **newval** 读出的字符串可能已经是 NUL 结束的，在那种情况里第 31154 行的赋值就是多余的。这还是另一种情况，直接完成工作比检查需要是否执行它还要快。

31156：返回 0 表示成功。相反，返回的值应该为正数，以便 30618 行代码认为结果是成功的。而又相反，调用代码认为 **sysctl_string** 想让缺省处理发生，然后它就继续从用户空间再次复制多余的数据。

sysctl_intvec

31163：**sysctl_intvec** 是在 kernel/sysctl.c 里定义的另一个策略例程。它确保假如调用程序正在写入表项，所有被写的 **int** 都应位于某个最小和最大值之间。（顺便提及一下，**sysctl_intvec** 在这个文件里只被使用了一次——在第 30414 行——尽管它被广泛的用于本书所没有包括的内核的其它代码之中。）

31170：如果新的欲写数据总量不符合一个 **int** 大小的边界，它就无效，所以尝试被抛弃。

31173：假如表项没有指定一组最大或最小值，输入的值就永远不可能超出范围，这样调用程序里的普通写代码（**do_sysctl_strategy**，30603 行）就足够好了。因此在这种情况里 **sysctl_intvec** 返回 0。

31184：进行循环以确保所有来自输入数组的值都位于适当范围之内。

31186：这行代码不检查 **get_user** 的返回值——没有迫切的需要去这样做。如果当不能读取一个输入内存位置时，**sysctl_intvec** 返回 0（成功），那么当它试图读取整个数组时 **do_sysctl_strategy** 就会注意到这个问题。作为另一选择，假如 **get_user** 无法读取内存位置，无用信息（garbage）可能在 **value** 里结束并且数值可能会不正确的被抛弃。在此情况里，调用程序将得到一个 **EINVAL** 错误而不是 **EFAULT** 错误，这只是一个小缺陷（bug）。

31187：注意这一行不会被折磨第 31033 行相似代码的缺陷所困扰，该行中在最小值和最大值之上进行的并行循环会产生不同步的情况。

这一行代码能够避免位于 31033 行的缺陷被暴露出来。正如实际中所进行的，**sysctl_intvec** 和 **proc_dointvec_minmax** 都总是与同一个 **ctl_table** 条目相关联的。因此，在调用处理例程（handler routine）**proc_dointvec_minmax** 之前，任何超出允许范围之外的数值将会被策略例程 **sysctl_intvec** 截获。所以，我们知道——在给定内核里所有的 **ctl_tables** 最新定义的情况下——**proc_dointvec_minmax** 将永远不会遇到超出界限的数值，而那里是唯一可以触发该缺陷的数值种类。某个调用程序或许可以注册一个使用 **proc_dointvec_minmax** 但没有策略例程的 **ctl_table**，但是尽管这样，在 **proc_dointvec_ minmax** 里的这个缺陷迟早会造成一定损害。

31193：返回 0 表示成功。这里不像在第 31156 行那样是一个错误，因为 **sysctl_intvec** 并不向 **table->data** 里写入。从用户空间读出的值只是被读进一个临时变量里作范围检查，然后就被删除；**do_sysctl_strategy** 将完成那项工作，并只向 **table->data** 里进行写入。

附录 A　Linux 2.4

内核的开发人员们并没有因为我在写这本关于 Linux 的书而暂停他们的工作。（我是多么希望他们这样做……。）说真的，内核的发展进展神速，就在这本书即将出版之际，它的最新稳定版本，2.4.0，也应该面世了。尽管在写作本书时要将对内核的成千种修改都一一涵盖到是不可能的，但是这篇附录却总结了发生在本书所涉及内核部分的最令人感兴趣的那些改变。这些修改中的绝大部分已经在内核的 2.3.12 版本里实现了，它们被包括在附赠的 CD-ROM 上。

作者十分感谢 Joe Pranevich 的论文，”Wonderful World of Linux 2.4”（**http://linuxtoday.Com/stories/8191.html**），在准备这篇附录时它提供了无价的帮助。

更少的“惊扰（stampedes[1]）”

请读者回忆一下第 2 章里，函数**__wake_up**（26829 行）唤醒等待在等待队列上的所有进程。可以考虑一个诸如 Apache 一样的 Web 服务器，它试图通过在同一端口派生出许多进程监听连接申请以缩短响应时间（正好与等待连接请求到达，并且与只通过 **fork** 派生一个进程来响应的方式相反）。所有这些进程都在一个等待队列里，等待连接请求到达，而当请求确实到达的时候，它们都会被唤醒。它们当中只有一个能够为请求提供服务，所以剩下的又都将返回休眠状态。

这种“惊扰”现象会浪费 CPU 的时钟周期——只唤醒一个等待进程会更好，因为不管怎样只有一个能够运行。因此，一个新的任务状态位 **TASK_EXCLUSIVE** 就被引进；在一次调用**__wake_up** 里，最多有一个设置了 **TASK_EXCLUSIVE** 位的任务会被唤醒。**TASK_EXCLUSIVE** 位是附加在其他任务状态位上的——它并不代表一个新的任务状态，它只是为了方便而保存在任务状态信息组里的信息而已。

现在**__wake_up** 要检查它正在唤醒的进程是否设置了 **TASK_EXCLUSIVE** 位，并且在它唤醒该进程之后便不再唤醒其他进程（通过使用 **break** 中断循环）。**TASK_EXCLUSIVE** 位现在只被用在有关网络的等待队列上，而且它在那种环境中效果也的确很好。对于大多数其他等待队列来说，你需要所有进程对等待着的资源都有机会进行占用，这样可以避免饥饿现象。不过等待在同一个端口的服务程序通常与 Apache 境遇相同：所有等待的任务都是一样的，而且重要的是只要它们中有一个能够处理请求就可以，即使每次都是同一个任务（也无所谓）。

再见吧，Java

正如第 7 章里所预言的那样，Java 二进制处理程序（binary handler）已经从内核里消失了，它是因被杂项二进制处理程序所替代而失效的。二元处理程序的常规用法没有变化，Java 执行体仍然可被杂项二进制处理程序的适当配置所完全支持。

[1] “stampede”原意是指：动物受到惊吓后纷纷逃窜。作者使用该词说明：在等待队列中休眠的进程同时被唤醒的情况。

ELF 权能位

已有非官方的修补程序可以用于（增强）ELF 执行体中权能的存储。它们还没有成为正式内核版本的一部分，部分原因在于是否 ELF 文件头是保存这些信息的恰当位置的问题还处于讨论之中。不过，这些修补程序也有可能已经成为正式版本的一部分了。

调度程序提速

已被高度优化了的 **schedule** 函数（26686 行）又被作了进一步优化。大多数修改只是在 **system_call**（171 行）系列调用的基础上进行了重新组织——也就是说，通过允许普通情况直接通过以及把遍布在函数里的大部分 **if** 语句体都分散开来的方式提高代码的（运行）速度。举例来说，第 26706 和 26707 行，如果又要运行的代码它们就运行底下的下半部分（bottom halves），现在采用的是这种形式：

P563—1

这样一来，如果下半部分程序必须运行，控制流程就会跳转到新的 **handle_bh** 标记处，然后在运行完下半部分之后再跳转回去。原有方式在无需运行下半部分时的正常情况下也要产生一个分支转移，因为在那种情况下所产生的代码不得不跳过对 **handle_bottom_halves** 的调用。在新的版本里，正常情况只需直接通过，不会产生任何分支。

更多的进程

Linux 2.4 几乎消除了对同时可以运行的进程数目的固有限制。唯一剩下来的硬编码（hard-coded）限制就是 PID 的最大数目了（不要忘记 PID 可被共享）。这个改进使得 Linux 能够更好的适合于高端（high-end）服务器应用程序，包括经常需要同时运行大量进程的 Web 服务。

在第 8 章中说明过，原有 4090 个任务的最高限度是受可以同时保存在全局描述表（GDT）里的任务状态段（TSS）和局部描述表（LDT）的最大数目所影响的：Linux 2.2 在 GDT 里为每个进程存储一个 TSS 和一个 LDT，而且 GDT 总共允许有最多 8192 个条目。Linux 2.4 通过存储 TSS 和 LDT 本身而不是把它们存储在 GDT 里的方法回避了这个限制。现在，GDT 只保持每 CPU 一个 TSS 和一个 LDT，而且把这些条目设置为在每个 CPU 上的当前运行任务所需要的信息。

虽然用于追踪每个 CPU 的空闲进程的 **init_tasks** 数组还存在，但是 **task** 数组（26150 行）现在却没有了。

日益进步的 SMP 支持

每一个锁都是一个 SMP 机器可以被有效地转化为 UP 机器的地方：正等待一个锁的 CPU 对系统的整体性能没有任何贡献。因此，改善 SMP 性能的主要方法就是通过减小锁的作用域或者完全消除它们来增加并行程序的运行机会。

Linux 2.0 只有一个单独的全局内核锁，所以每次只有一个 CPU 可以在内核里执行。Linux 2.2 把这个全局内核锁使用的大部分地方都用更小的、子系统专用的锁来替代了，其

中的一些我们在第 10 章中已经见到过。Linux 2.4 继续了这个趋势，它把可能减小其作用域的锁都作了进一步的分割。比如现在每个等待队列就有一个锁，而不是所有的等待队列才有唯一的一个锁。

附录 B　GNU 通用公共许可证

序言

大多数软件许可证都有意剥夺你的共享和修改软件的自由。相对之下，GNU 通用公共许可证力图保证你的共享和修改自由软件的自由。——保证自由软件对所有用户都是自由的。GPL 适用于大多数自由软件基金会的软件，以及由愿意遵守该许可证规定的作者所开发的软件。（自由软件基金会的其它一些软件受 GNU 库通用许可证的保护）。你也可以将它应用到你的程序中。

当我们谈到自由软件（free software）时，我们指的是自由而不是价格。我们的 GNU 通用公共许可证目的是要保证你有发布自由软件的自由（如果你愿意，你也可以对此项服务收取一定的费用）；保证你能收到源程序或者在你需要时能得到它；保证你能修改软件或将它的一部分用于新的自由软件；而且还保证你知道你能做这些事情。

为了保护你的权利，我们需要作出规定：禁止任何人不承认你的权利，或者要求你放弃这些权利。如果你自行发布了软件的副本或者对其进行了修改，那么这些规定就转化为你的责任。

例如，如果你发布这样一个程序的副本，不管是收费的还是免费的，你必须将你具有的一切权利给予你的接受者；你必须保证他们能收到或得到源程序；并且将这些条款给他们看，使他们知道他们有这样的权利。

我们采取两项措施来保护你的权利。

（1）给软件以版权保护。

（2）给你提供许可证。它给你复制，发布和修改这些软件的法律许可。

同样，为了保护每个作者和我们自己，我们需要清楚地让每个人明白，自由软件没有担保（no warranty）。如果由于其他某个人修改了软件，并继续加以传播。我们需要它的接受者明白：他们所得到的并不是原来的自由软件。由其他人引入的任何问题，不应损害原作者的声誉。

最后，任何自由软件都不断受到软件专利的威胁。我们希望能够避免这样的风险，自由软件的再发布者以个人名义获得专利许可证。事实上，将软件变为私有。为防止这一点，我们必须明确：任何专利必须以允许每个人自由使用为前提，否则就不准许有专利。

下面是有关复制，发布和修改的确切的条款和条件。

复制，发布和修改的条款和条件

此许可证适用于任何包含版权所有者声明的程序和其它作品，版权所有者在声明中明确说明程序和作品可以在本 GPL 条款的约束下发布。下面提到的“程序”指的是任何这样的

程序或作品。而“基于程序的作品”指的是程序或者任何受版权法约束的衍生作品：也就是说包含程序或程序的一部分的作品。可以是原封不动的，也可以是经过修改的和／或翻译成其它语言的（程序）。在下文中，翻译包含在修改的条款中。每个许可证接受人（licensee）用“你”来称呼。

许可证条款不适用于复制，发布和修改以外的活动。这些活动超出这些条款的范围。运行程序的活动不受条款的限制。仅当程序的输出构成基于程序作品的内容时，这一条款才适用（如果只运行程序就无关）。是否普遍适用取决于程序具体用来做什么。

1. 只要你在每一副本上明显和恰当地给出版权声明和不承担担保的声明，保持此许可证的声明和没有担保的声明完整无损，并和程序一起给每个其它的程序接受者一份许可证的副本，你就可以用任何媒体复制和发布你收到的原始的程序的源代码。

你可以为转让副本的实际行动收取一定费用。你也有权选择提供担保以换取一定的费用。

2. 你可以修改程序的一个或几个副本或程序的任何部分，以此形成基于程序的作品。只要你同时满足下面的所有条件，你就可以按前面第一款的要求复制和发布这一经过修改的程序或作品。

a） 你必须在修改的文件中附有明确的说明：你修改了这一文件及具体的修改日期。

b） 你必须使你发布或出版的作品（它包含程序的全部或一部分，或包含由程序的全部或部分衍生的作品）允许第三方作为整体按许可证条款免费使用。

c） 如果修改的程序在运行时以交互方式读取命令，你必须使它在开始进入常规的交互使用方式时打印或显示声明：包括适当的版权声明和没有担保的声明（或者你提供担保的声明）；用户可以按此许可证条款重新发布程序的说明；并告诉用户如何看到这一许可证的副本。（例外的情况：如果程序自身以交互方式工作，但是它并不像通常情况一样打印这样的声明，你的基于程序的作品也就不用打印声明）。

这些要求适用于修改了的作品的整体。如果能够确定作品的一部分并非程序的衍生产品，可以合理地认为这部分是独立的，是不同的作品。当你将它作为独立作品发布时，它不受此许可证和它的条款的约束。但是当你将这部分作为基于程序的作品的一部分发布时，作为整体它将受到许可证条款约束。准予其他许可证持有人的使用范围扩大到整个产品。也就是每个部分，不管它是谁写的。

因此，本条款的意图不在于索取权利；或剥夺全部由你写成的作品的权利。而是履行权利来控制基于程序的集体作品或衍生作品的发布。

此外，将与程序无关的作品和该程序或基于程序的作品一起放在存贮介质或发布媒体的同一卷上，并不导致将其它作品置于此许可证的约束范围之内。

3. 你可以以目标码或可执行形式复制或发布程序（或符合第 2 款的基于程序的作品)，只要你遵守前面的第 1，2 款，并同时满足下列 3 条中的 1 条。

a）在通常用作软件交换的媒体上，和目标码一起附有机器可读的完整的源码。这些源码的发布应符合上面第 1，2 款的要求。或者

b）在通常用作软件交换的媒体上，和目标码一起，附有给第三方提供相应的机器可读的源码的书面报价。有效期不少于 3 年，费用不超过完成源程序发布的实际成本。源码的发布应符合上面的第 1，2 款的要求。或者

c）和目标码一起，附有你收到的发布源码的报价信息。（这一条款只适用于非商业性发布，而且你只收到程序的目标码或可执行代码和按 b）款要求提供的报价）。

作品的源码指的是对作品进行修改最优先择取的形式。对可执行的作品讲，完整的源码包括：所有模块的所有源程序，加上有关的接口的定义，加上控制可执行作品的安装和编译的 script。作为特殊例外，发布的源码不必包含任何常规发布的供可执行代码在上面运行的

操作系统的主要组成部分（如编译程序，内核等）。除非这些组成部分和可执行作品结合在一起。

如果采用提供对指定地点的访问和复制的方式发布可执行码或目标码，那么，提供对同一地点的访问和复制源码可以算作源码的发布，即使第三方不强求与目标码一起复制源码。

4. 除非你明确按许可证提出的要求去做，否则你不能复制、修改、转发许可证和发布程序。任何试图用其它方式复制、修改、转发许可证和发布程序是无效的。而且将自动结束许可证赋予你的权利。然而，对那些从你那里按许可证条款得到副本和权利的人们，只要他们继续全面履行条款，许可证赋予他们的权利仍然有效。

5. 你没有在许可证上签字，因而你没有必要一定要接受这一许可证。然而，没有任何其它东西赋予你修改和发布程序及其衍生作品的权利。如果你不接受许可证，这些行为是法律禁止的。因此，如果你修改或发布程序（或任何基于程序的作品），你就表明你接受这一许可证以及它的所有有关复制，发布和修改程序或基于程序的作品的条款和条件。

6. 每当你重新发布程序（或任何基于程序的作品）时，接受者自动从原始许可证颁发者那里接到受这些条款和条件支配的复制，发布或修改程序的许可证。你不可以对接受者履行这里赋予他们的权利强加其它限制。你也没有强求第三方履行许可证条款的义务。

7. 如果由于法院判决或违反专利的指控或任何其它原因（不限于专利问题）的结果，强加于你的条件（不管是法院判决，协议或其它）和许可证的条件有冲突。他们也不能用许可证条款为你开脱。在你不能同时满足本许可证规定的义务及其它相关的义务时，作为结果，你可以根本不发布程序。例如，如果某一专利许可证不允许所有那些直接或间接从你那里接受副本的人们在不付专利费的情况下重新发布程序，唯一能同时满足两方面要求的办法是停止发布程序。

如果本条款的任何部分在特定的环境下无效或无法实施，就使用条款的其余部分。并将条款作为整体用于其它环境。

本条款的目的不在于引诱你侵犯专利或其它财产权的要求，或争论这种要求的有效性。本条款的主要目的在于保护自由软件发布系统的完整性。它是通过通用公共许可证的应用来实现的。许多人坚持应用这一系统，已经为通过这一系统发布大量自由软件作出慷慨的供献。作者／捐献者有权决定他／她是否通过任何其它系统发布软件。许可证持有人不能强制这种选择。

本节的目的在于明确说明许可证其余部分可能产生的结果。

8. 如果由于专利或者由于有版权的接口问题使程序在某些国家的发布和使用受到限制，将此程序置于许可证约束下的原始版权拥有者可以增加限止发布地区的条款，将这些国家明确排除在外。并在这些国家以外的地区发布程序。在这种情况下，许可证包含的限止条款和许可证正文一样有效。

9. 自由软件基金会可能随时出版通用公共许可证的修改版或新版。新版和当前的版本在原则上保持一致，但在提到新问题时或有关事项时，在细节上可能存在一定差别。

每一版本都有不同的版本号。如果程序指定适用于它的许可证版本号以及“任何更新的版本”。你有权选择遵循指定的版本或自由软件基金会以后出版的新版本，如果程序未指定许可证版本，你可选择自由软件基金会已经出版的任何版本。

10. 如果你愿意将程序的一部分结合到其它自由程序中，而它们的发布条件不同。写信给作者，要求准予使用。如果是自由软件基金会加以版权保护的软件，写信给自由软件基金会。我们有时会作为例外的情况处理。我们的决定受两个主要目标的指导。这两个主要目标是：我们的自由软件的衍生作品继续保持自由状态。以及从整体上促进软件的共享和重复利用。

没有担保

11. 由于程序准予免费使用，在适用法准许的范围内，对程序没有担保。除非另有书面说明，版权所有者和／或其它提供程序的人们“一样”不提供任何类型的担保。不论是明确的，还是隐含的。包括但不限于隐含的适销和适合特定用途的保证。全部的风险，如程序的质量和性能问题都由你来承担。如果程序出现缺陷，你承担所有必要的服务，修复和改正的费用。

12. 除非适用法或书面协议的要求，在任何情况下，任何版权所有者或任何按许可证条款修改和发布程序的人们都不对你的损失负有任何责任。包括由于使用或不能使用程序引起的任何一般的、特殊的、偶然发生的或重大的损失（包括但不限于数据的损失，或者数据变得不精确，或者你或第三方的持续的损失，或者程序不能和其它程序协调运行等）。即使版权所有者和其他人提到这种损失的可能性也不例外。

如何将这些条款用到你的新程序

如果你开发了新程序，而且你需要它得到公众最大限度的利用。要做到这一点的最好办法是将它变为自由软件。使得每个人都能在遵守条款的基础上对它进行修改和重新发布。

为了做到这一点，给程序附上下列声明。最安全的方式是将它放在每个源程序的开头，以便最有效地传递拒绝担保的信息。每个文件至少应有“版权所有”行以及在什么地方能看到声明全文的说明。

<用一行空间给出程序的名称和它用来做什么的简单说明>

版权所有（C） 19XX <作者姓名>

这一程序是自由软件，你可以遵照自由软件基金会出版的 GNU 通用公共许可证条款来修改和重新发布这一程序。或者用许可证的第二版，或者（根据你的选择）用任何更新的版本。

发布这一程序的目的是希望它有用，但没有任何担保。甚至没有适合特定目的的隐含的担保。更详细的情况请参阅 GNU 通用公共许可证。

你应该已经和程序一起收到一份 GNU 通用公共许可证的副本。如果还没有，写信给：

The Free Software Foundation, Inc., 59 Temple Place, Suite, 330, Boston, MA02111-1307, USA

还应加上如何和你保持联系的信息。

如果程序以交互方式进行工作，当它开始进入交互方式工作时，使它输出类似下面的简短声明：

Gnomovision 第 69 版， 版权所有（C） 19XX， 作者姓名，

Gnomovision 绝对没有担保。 要知道详细情况，请输入‘show w’。

这是自由软件，欢迎你遵守一定的条件重新发布它，要知道详细情况，

请输入‘show c’。

假设的命令‘show w’和‘show c’应显示通用公共许可证的相应条款。当然，你使用的命令名称可以不同于‘show w’和‘show c’。根据你的程序的具体情况，也可以用菜单或鼠标选项来显示这些条款。

如果需要，你应该取得你的上司（如果你是程序员）或你的学校签署放弃程序版权的声明。下面只是一个例子，你应该改变相应的名称：

Yoyodyne 公司以此方式放弃 James Hacker
所写的 Gnomovision 程序的全部版权利益。
<Ty coon 签名>，1989.4.1
Ty coon 付总裁

这一许可证不允许你将程序并入专用程序。如果你的程序是一个子程序库。你可能会认为用库的方式和专用应用程序连接更有用。如果这是你想做的事，使用 GNU 库通用公共许可证代替本许可证。