Robert Love

Third Edition

Linux Kernel Development

A thorough guide to the design and implementation of the Linux kernel

Developer's Library

Linux Kernel Development

Third Edition

Robert Love

Translated by: Kevin

目录

序

随着 Linux 内核和使用内核的应用得到越来越广泛的使用，希望参与开发和维护 Linux 的系统软件开发者的数量也在不断增加。这些工程师中有的单纯是个人兴趣，有些任职于 Linux 公司，有些为硬件厂商工作，也有一些是个人开发项目相关。

但是所有人都面临同样的一个问题：内核学习曲线越来越长、越来越陡峭。内核系统非常庞大，而且复杂度与日俱增。随着这几年的发展，当前内核开发团队的成员对内核内部的知识更加深入和广泛，这也进一步扩大了他们和新手之间的差距。

我相信 Linux 内核源码的这种可达性衰退已经成为影响内核质量的一个问题，并且随着时间的发展会越来越严重。所有关心 Linux 发展的人都希望为内核贡献的开发者数量能够得到增长。

解决这个问题的一个办法是保持代码的清晰：合理的接口、统一的层次架构、“只做一件事并且做好”等等。这也正是 Linus Torvalds 所采取的办法。

我个人建议的办法是增加代码注释：读者在阅读代码时可以据此来理解代码编写者的真实意图。（找出意图和实现之间分歧的过程被称为调试，如果意图不明确就非常难以调试）。

但是代码注释也不能提供主要子系统的更高层次的意图，以及开发者是如何理解它的。要理解这种意图，就只有相关的书籍才能做到了。

Robert Love 的这本书给我们展示了富有经验的开发者对于内核的核心认识，内核子系统提供的各种服务、它们是如何提供这些服务。所有这些知识能够满足许多不同的人：好奇者、应用开发者、想要体会内核设计的人等等。

本书还能帮助有抱负的内核开发者步入下一个层次：修改内核来完成明确的既定目标。我非常鼓励有抱负的开发者亲自动手。理解内核的最佳途径就是修改它。修改内核要求开发者对内核的理解，要站在比阅读代码更高的层次。认真的内核开发者会加入邮件列表、并与其他开发者积极沟通。这种沟通是内核贡献者互相学习和保持一致的主要动力。Robert 很好的阐述了内核的结构和相关文化。

从 Robert 的书中你能学习并得到享受，你会决定进入下一个层次，成为内核开发社区的一份子。我们通过人们的贡献来评价和衡量一个人，当你为 Linux 做出贡献时，即使你完成的工作很小，但是能够立即让数百万人获益。这是特权和责任能够得到的最大享受。

Andrew Morton

前言

当我最早考虑将我对 Linux 内核的经验转换为一本书时，我始终都怀着一丝忧虑。怎样才能使我的书成为该主题最好的书籍？如果我不能做特别的、最好的工作，那我就不会感兴趣。

最终我意识到我可以对内核这一主题提供独到的方法。我的工作是 hacking 内核、我的嗜好是 hacking 内核、我也热爱 hacking 内核。这么多年以来，我积累了大量有趣的轶闻和内情。以我的经验可以写一本如何 hack 内核（同样重要的是如何防止 hack 内核）的书。首先也是最重要的，这是一本关于 Linux 内核设计和实现的书。这本书使用的方式与它的竞争者不同，我会提供足够的信息使你能更好地完成工作。我是一个实用主义的工程师，这也是一本注重实践的书。本书应该很有趣、阅读很轻松、并且非常有用。

我希望读者通过阅读本书，能够加深对 Linux 内核规则的理解。我的目标是让你（没有阅读过本书和内核源代码的人），能迅速入门并开始编写有用的、正确的、清晰的内核代码。当然，你也可以仅仅把本书当作娱乐使用。

本书第一版出版至今，已经好几年过去了。第三版比之前两版提供了相当多的内容：修订、更新、和许多全新的章节。第三版吸收并增加了第二版之后内核的所有变更。更重要的是，Linux 内核社区决定短期内不继续进行 2.7 版本内核的开发，而是计划继续开发和稳定 2.6 系列。这个决定有许多影响，但是与本书相关的则是继续使用 2.6 版本的内核。由于 Linux 内核非常成熟，即使是内核快照也能保持相当长时间的稳定。本书可以说是内核的权威文档，同时兼顾内核的历史和未来。

如何使用本书

开发内核代码不需要天才和魔法，甚至 Unix 黑客标志性的大胡子也不需要。内核尽管也有一些自己的有趣的规则，但它与其它任何一个大型软件并无太大区别。当然你需要掌握许多细节（和任何大型项目一样），但区别只是数量，而不是性质。

要学习内核，使用源码是必须的。Linux 系统源码的开放可用性，是一份珍贵的礼物。但是仅仅阅读源码是远远不够的，你还需要深入进去并修改一些代码、找出 bug 并修复它、改进你使用的硬件驱动、增加新功能。即使是很小的事情，你都要找出并完成它。只有当你真正编写了内核代码，才能真正理解内核的所有。

内核版本

本书基于 Linux 内核 2.6 系列版本，不涉及更老的内核，除非与历史相关。例如只要对于我们的讨论是有帮助的，我们也会讨论某个内核子系统在 2.4 版本中的实现方式。具体来说，本书第三版使用了 Linux 内核 2.6.34。尽管内核一直在不断的前进，要保持更新确实很难，但我的目标是使本书能够同时适用于老的和新的内核开发者和用户。

尽管本书讨论 2.6.34 内核，但我花了很大力气确保书中的材料也能适用于 2.6.32 内核。后者是许多 Linux 发布版为“企业”内核所做的剪裁，在许多年内都将在产品系统中使用，也会有持续动态的开发支持（2.6.9，2.6.18 和 2.6.27 都是类似的长期发布版）。

本书读者

本书的目标读者是对理解 Linux 内核感兴趣的开发者和用户。它不是对内核源码一行一行的注释，也不是驱动开发的指南，或内核 API 的参考手册。相反本书的目标是提供 Linux 内核设计和实现的足够的信息，使程序员能开始开发内核代码。内核开发非常有趣，报酬也不错，而我希望本书能尽快把读者引入这个世界。本书既讨论理论又兼顾实用，所以应该同时适用于学院派和实践派的读者。我一直认为只有掌握了理论，才能更好地进行实践，所以我试图在本书中平衡二者。我希望无论你理解 Linux 内核的动机是什么，本书都能满足你的需求，为你解释清楚内核的设计和实现。

因此本书同时讲解了核心内核子系统的使用，和它们的设计与实现。我认为这一点很重要，值得我们稍加讨论。以第 8 章“Bottom Halves 和 Deferring Work”

为例，该章讲解了一个被称为 bottom halves 的设备驱动组件。在那一章中，我既讨论了内核 bottom-half 机制的设计和实现（核心内核开发者和学院派应该会感兴趣），也讨论了如何使用导出的接口来实现你自己的 bottom half（设备驱动开发者和黑客会感兴趣）。核心内核开发者当然需要理解内核的内部工作原理，而他们一般都对接口的使用非常熟悉。同时设备驱动开发者也能从理解接口背后的实现得到很多好处。

这和学习某个库的 API 与学习库的实现是一个道理。表面看来，应用程序员只需要理解 API（将接口视作黑箱通常是足够的）；库开发者只需要关心库的设计和实现。但我认为双方都应该花费一些时间来学习另一方面。那些更好地理解了底层操作系统的应用程序员，能够更好地使用操作系统的接口。同样，库开发者也不应该与应用如何使用库完全脱离。因此，我同时讨论了内核子系统的设计和使用，希望本书能够对双方都有用。

我假设读者了解 C 编程语言，熟悉 Linux 系统。有一些操作系统设计和计算机科学相关的理论对学习本书会有帮助，我也会尽可能多地解释相关的概念。如果我没有，附录的参考书目列出了一些操作系统设计方面的卓越书籍。

本书也适用于操作系统设计课程的大学教材，可以作为理论介绍教材的补充材料。本书可以在大学高年级课程或研究生课程中使用。

第三版鸣谢

（略）

关于作者

Robert Love 是一个开源程序员，演讲者，和作者，他使用和贡献 Linux 已经超过 15 年。Robert 目前是 Google 的高级软件工程师，是 Android 移动平台内核团队的成员之一。在去 Google 之前，他是 Novell Linux 桌面的首席架构师。在 Novell 之前，他是 MontaVista 软件和 Ximian 的内核工程师。

Robert 的内核项目包括抢占式内核、进程调度器、内核事件层、inotify、VM 增加、和几个设备驱动。

Robert 进行了无数演讲，撰写了许多关于 Linux 内核的文章。他也是 Linux Journal 的编辑之一。他编写的其它几本书包括 Linux System Programming 和 Linux in a Nutshell。

Robert 获得了福罗里达大学的数学和计算机科学学位。现居住于波士顿。

第 1 章 Linux 内核介绍

本章介绍 Linux 内核和 Linux 操作系统，将它们跟 Unix 的历史结合在一起。今天，Unix 指的是一组操作系统，它们实现了类似的应用编程接口（API），并且遵循相同的设计原则。但是 Unix 也是一个特定的操作系统，距今已有 40 多年。要理解 Linux，我们必须首先讨论最初的 Unix 系统。

Unix 的历史

经过四十多年的使用，计算机科学家仍然赞誉 Unix 操作系统是当今最强大最优美的系统。从 1969 年 Unix 诞生以来，Dennis Ritchie 和 Ken Thompson 的作品已经成为一个传奇，Unix 的设计经受住了时间的考验。

Unix 起源于 Multics，后者是 Bell 实验室参与的一个失败的多用户操作系统项目。Multics 项目终止以后，Bell 实验室计算机科学研究中心的成员发现自己没有可用的交互式操作系统。于是在 1969 年夏，Bell 实验室的程序员们设计出了一个文件系统，最终演化成了 Unix。为了测试该系统的设计，Thompson 在一台 PDP-7 机器上实现了这个新系统。在 1971 年，Unix 成功移植到 PDP-11；然后在 1973 年，又用 C 语言重写了 Unix 操作系统。在当时这是史无前例的一步，但也正是如此，才为将来的可移植性铺平了道路。第一个在 Bell 实验室之外得到广泛使用的 Unix 操作系统是 Unix System 第六版，经常被称为 V6。

许多公司将 Unix 移植到新机器上。这些移植在增强 Unix 功能的同时，也导致了 Unix 操作系统的几个不同变种。1977 年 Bell 实验室组合这些变种到一起，发布了 Unix System III；1982 年 AT&T 发布了 System V。（System IV 是一个内部开发版本）。

Unix 设计的简洁性，加上它又以源码的方式发布，吸引了许多外部组织的进一步开发。其中最有影响的莫过于加利福尼亚大学伯克利分校。伯克利的 Unix 变种被称为 Berkeley Software Distributions，或者 BSD。伯克利在 1977 年第一次发布 1BSD，在 Bell 实验室的 Unix 上增加了补丁和额外的软件包。随后 1978 年的 2BSD 仍然是以这种形式发布，增加了 csh 和 vi 等实用工具，这些工具一直持

续使用到今天。第一个独立的 Berkeley Unix 是 1979 年发布的 3BSD，它在已经非常丰富的特性集中增加了虚拟机功能。随后发布的是 4 系列的 BSD，4.0BSD、4.1BSD、4.2BSD、4.3BSD。这些版本增加了任务控制、动态页面、和 TCP/IP。1994 年伯克利发布了最后一个官方版本的 Berkeley Unix，重写了 VM 子系统，是为 4.4BSD。今天我们要感谢 BSD 宽容的许可，正因为此才有了后来的 Darwin、FreeBSD、NetBSD、和 OpenBSD 系统。

1980 至 1990 年间，多个工作站和服务器公司发布了自己商业版本的 Unix。这些系统都是基于 AT&T 或 Berkeley 的 Unix 发布版，但是支持自己特定硬件体系架构的高端特性。比较有名的有 Digital 的 Tru64、Hewlett Packard 的 HP-UX、IBM 的 AIX、Sequent 的 DYNIX/ptx、SGI 的 IRIX、以及 Sun 的 Solaris & SunOS。

Unix 系统最初优雅的设计，加上多年来的改革和发展，已经成为了功能强大、健壮、和稳定的操作系统。Unix 拥有许多优秀的特性。首先 Unix 非常简单，比起某些操作系统动辄几千个系统调用和不清晰的设计目标，Unix 只实现了大约几百个系统调用，而且拥有非常直接和基本的设计。第二、在 Unix 中，“一切都是文件”（好吧，也不是所有，但大多数都被表示为文件。Sockets 是明显的一个例外。最新的一些系统例如 Bell 实验室的 Unix 继承者 Plan9，几乎系统的所有方面都实现为文件）。这样就可以把所有操作数据和设备简化为一组核心系统调用：open(), read(), write(), lseek(), close()。第三、Unix 内核和相关的系统工具用 C 语言编写，这赋予了 Unix 极大的可移植性，使其适用于各种硬件体系架构，并且能够对许多开发者可用。第四、Unix 拥有非常快速的进程创建和独特的 fork()系统调用。最后、Unix 提供一组简单但却健壮的进程间通讯（IPC）原语，与快速进程创建结合在一起，允许创建简单的程序“做一件事并做到最好”。这些单一目的的程序可以被串连在一起，来完成复杂的任务。Unix 系统也因此获得了清晰的层次架构，在策略和机制之间建立了给力的分离。

今天 Unix 是一个支持抢占式多任务、多线程、虚拟机、动态页面、动态装载共享库、和 TCP/IP 网络的现代操作系统。各种 Unix 变种可以运行在大到数百个处理器的机器、小至嵌入式设备中。尽管 Unix 已经不再是一个研究项目，Unix 系统仍然在推动操作系统设计的发展，同时又保持实践性和通用操作系统的特点。

Unix 的成功归因于简单性和优雅的设计。它今天的强大起源于 Dennis Ritchie、Ken Thompson 和其它早期开发者所做出的英明决策：赋予了 Unix 不断进化却又不需妥协自己的能力。

Linus 和 Linux 介绍

Linus Torvalds 在 1991 年为一台 Intel 80386 微处理器的计算机开发了 Linux 操作系统的第一个版本，当时 80386 是一款很新而且很高级的处理器。Linus 当时是赫尔辛基大学的一名学生，正苦恼于缺乏强大而免费的 Unix 系统。当时主流的个人计算机 OS 是 Microsoft DOS，对于 Torvalds 来说只能玩波斯王子而已。Linus 也用过 Minix，一个廉价的为教学目的而产生的 Unix，但是由于 Minix 的许可和其作者所做的设计决策，导致 Linus 无法修改和发布 Minix 系统的源码，这让他感到很气馁。

面对如此困局，Linus 做了任何一个正常的大学生都会做的事情：他决定编写自己的操作系统。Linus 从编写一个简单的终端模拟器开始，他用来连接学校的大型 Unix 系统。随着大学课程的进行，他的终端模拟器得到了不断的发展和改进。不久以后，Linus 手上就有了一个粗糙但羽翼丰满的 Unix。他在 1991 年底在因特网上发布了第一个早期版本。

早期 Linux 发布版很快地获得了许多用户，但是比起最初的成功，更重要的是 Linux 很快地吸引了大批开发者——增加、修改、改进代码的黑客。而 Linux 许可的条款也使其很快就演变成为一个许多人合作开发的项目。

今天 Linux 已经是一个羽翼丰满的操作系统，可以支持 Alpha、ARM、PowerPC、SPARC、x86-64 以及许多其它体系架构。Linux 运行在小到手表，大到整间房子的超级计算机集群的不同系统中。Linux 为最小的消费电子产品和大型数据中心提供动力。今天 Linux 的商业利益也非常强劲。新的 Linux 公司如 RedHat、以及现有的计算机大企业如 IBM，都为嵌入式、移动、桌面、和服务器等领域提供基于 Linux 的解决方案。

Linux 是一个类 Unix 的系统，但它不是 Unix。也就是说尽管 Linux 借鉴了 Unix 很多的思想，也实现了 Unix API（由 POSIX 和 Single Unix Specification 定义），但

它不像其它 Unix 系统一样直接继承自 Unix 的源代码。当需要时 Linux 也会偏离 Unix 的路线，但它并没有抛弃 Unix 的设计目标，也没有破坏标准化的应用接口。

Linux 最有趣的特点之一是：它不是一个商业产品，相反，它是一个基于互联网的合作开发项目。虽然 Linus 仍然是 Linux 内核的创建者和维护者，内核的持续发展却是通过一个松散组织的开发者团队来实现。任何人都可以为 Linux 贡献。Linux 内核与系统的大部分都是免费或开放源码软件。具体来说，Linux 内核采用 GNU 通用公共许可证（GPL）版本 2.0。因此你可以自由地下载源代码，并进行任何你想要的修改。唯一需要注意的是，如果要发布你的更改，你必须提供你享受到的所有权利，包括源代码的可用性。

Linux 对很多人来说意味着很多东西。Linux 系统的基础是内核、C 库、工具链、基本的系统工具（如登录过程和 Shell 等）。Linux 系统也可以包含一个现代的 X Window 系统，包括一个全功能的桌面环境，比如 GNOME。Linux 系统拥有成千上万的免费和商业应用。在这本书中，我说的 Linux 通常指的是 Linux 内核。如果存在混淆，我会明确地指出是指一个完整的 Linux 系统，还是指 Linux 内核。严格来说，Linux 这个词仅仅表示 Linux 内核。

操作系统和内核概述

由于持续不断增长的特性，以及某些现代商业操作系统病态的设计，操作系统的概念已经很难精确定义。许多用户认为他们在屏幕上看到就是操作系统。技术上来讲，操作系统是负责实现基本使用和管理的那部分系统，本书也采纳这个观点。这包括内核和设备驱动、启动引导器、命令行 shell 或其它用户界面、以及基本的文件和系统工具。操作系统只是你需要的那些东西——不是网页浏览器或音乐播放器。系统这个术语则表示操作系统和所有运行在其上的应用程序。

当然本书的主题是内核，而用户界面是操作系统的最外层部分，内核则处于最内层。它是系统的内部核心，为系统的所有其它部分提供核心服务，管理硬件，并分配系统资源。内核有时候也被称为操作系统的 supervisor、core 或 internals。内核的典型组成包括服务中断请求的中断处理器、多个进程共享处理器时间的调度器、管理进程地址空间的内存管理子系统、以及网络和进程间通讯等系统服务。

在拥有受保护内存管理单元的现代系统中，内核一般都处于高级系统状态，区别于普通的用户应用。这包括一个受保护的内存空间，和硬件的完全访问权。这个系统状态和内存空间合起来称为内核空间。反之，用户应用则在用户空间中执行。它们只能看到机器可用资源的一个子集，只能执行特定的系统函数，只能访问内核分配给它们的硬件和内存，否则将被拒绝。当执行内核代码时，系统将以内核模式运行在内核空间中；当运行一个普通的进程时，系统就以用户模式运行在用户空间中。

应用程序通过系统调用（如图 1.1）与内核进行交互。应用通常调用系统库提供的函数（例如 C 库），而库则依赖于系统调用接口指示内核来完成应用请求的任务。有一些库还提供很多系统调用中没有的特性，因此在很多功能的实现中调用内核只是其中的一部分。例如我们熟悉的 printf()函数，它提供数据的缓存和格式化输出，它进行了很多的工作，但只是最后一步才调用了 write()把数据写入到控制台。反过来，有一些库调用则与内核系统调用有一对一的关系。例如 open()库函数基本上只是调用 open()系统调用。还有其它一些 C 库函数，如 strcpy()，则完全不使用内核。当应用执行系统调用时，我们就说内核正在代表应用程序执行（kernel is executing on behalf of the application）。此时应用就称为正在内核空间中执行系统调用（executing a system call in kernel-space），而内核则运行在进程上下文中。这种关系（应用通过系统调用接口进入内核）是应用完成工作的基本模式。

内核也管理系统的硬件。几乎 Linux 支持的所有体系架构和系统都提供中断的概念。当硬件需要与系统交互时，它就会向处理器发出一个中断，进而中断内核。中断用一个数字来标识，内核通过这个数字来执行特定的中断处理器，来处理和响应中断。例如当你打字时，键盘控制器就会产生一个中断，通知系统键盘缓冲区有了新的数据。内核拿到这个中断数值后，就可以执行正确的中断处理器。中断处理器处理键盘数据，并且通知键盘可以继续接收数据。为了同步，内核可以禁止中断（所有中断或某个特定的中断）。在许多操作系统中，包括 Linux，中断处理器并不运行在进程上下文中。相反，它们运行在特殊的中断上下文，与任何进程都无关。这个特殊的上下文存在的目的是让中断处理器能够迅速地响应中

断，然后退出。

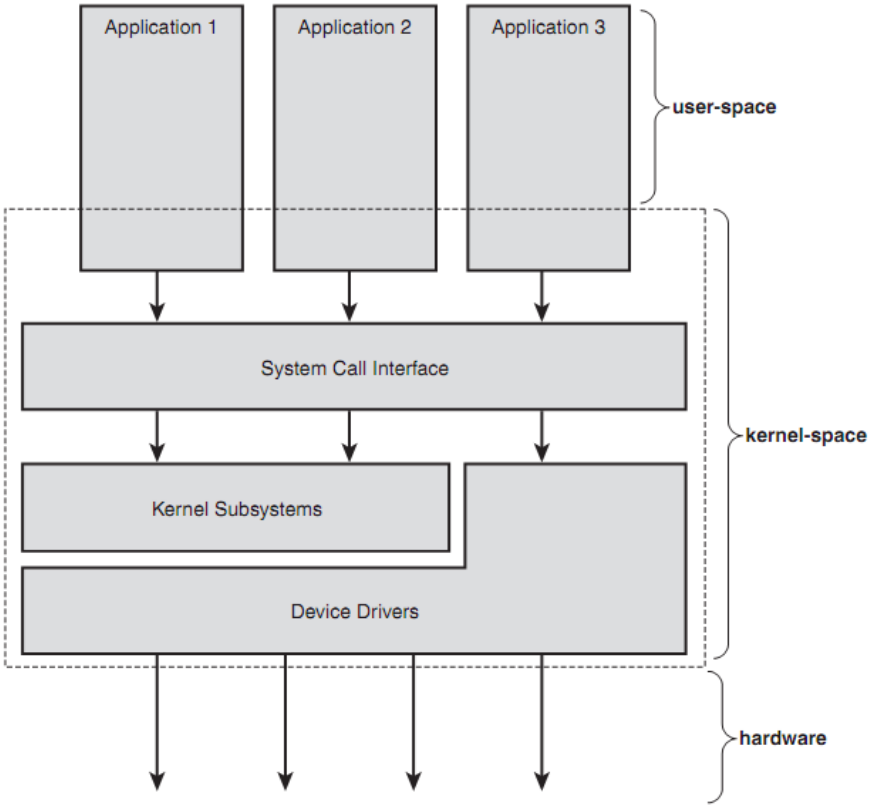

图 1.1 应用、内核和硬件之间的关系

上面描述的这些上下文表示了内核的活动性。实际上在 Linux 中，我们可以把每个进程在任意时刻所做的事情概括为以下三种情况：

- 在用户空间，进程正在执行用户代码。
- 在内核空间的进程上下文，代表特定进程执行。
- 有内核空间的中断上下文，不与任何进程关联，正在处理中断。

这份列表囊括了所有可能的情况，即使是最边角的情况也能适用。例如空闲时，内核正在进程上下文中执行 idle 进程。

Linux 内核对比经典的 Unix 内核

由于共同的祖先和相同的 API，现代 Unix 内核都共享许多设计特性。（请看参考书目中我最喜欢的关于经典 Unix 内核设计的书籍）。通常 Unix 内核都是单

一静态的二进制文件，只有极少数例外。也就是说内核是一个单一的、大型的、可执行的镜像，并且运行在单一的地址空间中。Unix 系统通常需要分页内存管理单元（MMU）的支持。MMU 允许系统执行内存保护，为每个进程提供唯一的虚拟地址空间。Linux 历史上也需要 MMU，但特定版本实际上可以脱离 MMU 运行。这是一个很好的特性，允许 Linux 运行在非常小的嵌入式无 MMU 的系统中。不过今天的实际情况是，即使最简单的嵌入式系统也都拥有很多高级特性，包括内存管理单元。因此在本书，我们只关注基于 MMU 的系统。

单内核 VS 微内核设计

我们可以把内核设计分为两个主要的阵营：单内核和微内核（还有一个第三阵营 exokernel，主要用在研究系统中）。

单内核是两个之中更为简单的设计，直到 1980 年代之前所有内核都是如此设计的。单内核完全实现为单一进程，并运行在单一的地址空间中。因此这种内核在硬盘中一般也是单一的静态二进制文件。所有的内核服务都存在并运行于一个大的内核地址空间中。内核内部的通讯非常简单，因为所有的东西都以内核模式运行在相同的地址空间中：内核可以像用户空间应用一样直接调用函数。单内核的优点在于简单和性能。大多数 Unix 系统都设计为单内核。

相反，微内核就不是实现为大的单一进程。内核的功能被分解为许多独立的进程，通常称为 server。理想情况下，只有那些需要特权的 Server 才会运行在特权模式，其它 server 则运行在用户空间。所有 server 都被分开在不同的地址空间，因此直接函数调用是不可能的。微内核通过消息传递来进行交互：消息传递是系统内建的一种进程间通讯进制，不同的 server 通过 IPC 机制发送消息来调用其它 server 的“服务”。不同 server 的分离防止了一个 server 出错导致其它 server 崩溃的情况。同样，系统的模块化也允许 server 的自由替换。

由于 IPC 机制比简单的函数调用开销更大，也由于经常需要在内核空间和用户空间之间互相切换，消息传递会有一定的延迟，吞吐量也不如单内核。因此所有实际的微内核系统现在都把多数或全部 server 放在内核空间中，以消除频繁的上下文切换开销，并允许直接函数调用。Windows NT 内核（Windows XP、Vista

和 Win 7 的基础)和 Mach(Mac OS X 的基础)都是微内核的例子。无论是 Windows NT 还是 Mac OS X，都没有把它们任何的微内核 server 放在用户空间，完全放弃了微内核的主要设计目标。

Linux 是单内核的，也就是说 Linux 内核运行在单一的地址空间，完全在内核模式下执行。但是 Linux 借用了微内核很多优秀的设计原则：拥有模块化设计、抢占自己的能力（称为内核抢占）、支持内核线程、拥有动态加载不同模块的能力（内核模块）。因此 Linux 没有微内核设计的性能问题：所有东西都运行在内核模式，使用直接函数调用而不是消息传递。但是 Linux 却又是模块化的、线程的、内核本身也可以被调度。实用主义再次获胜。

Linus 和其它内核开发者开发 Linux 内核时，会决定怎样最好地发展 Linux 而又不忽视它的 Unix 根源（更重要的是 Unix API）。由于 Linux 不是基于某个 Unix 变种，Linus 和内核团队可以对任何问题选择最佳解决方案（或者是发明新的方案）。Linux 内核和经典 Unix 系统的显著区别主要有：

- Linux 支持动态装载内核模块。尽管 Linux 是单内核，它可以在需要时动态地装载和卸载内核代码。
- Linux 有对称多处理器（SMP）支持。尽管多数商业 Unix 变种现在也支持 SMP，但多数传统 Unix 实现不支持。
- Linux 内核是抢占的。不像传统的 Unix 变种，Linux 内核可以抢占正在内核中执行的任务。商业的 Unix 实现，如 Solaris 和 IRIX 也有抢占式内核，但多数 Unix 内核都不支持抢占。
- Linux 对线程的实现方式很特别：它不区分线程和普通的进程。对于内核来说，所有进程都是一样的——线程只不过是共享资源的进程而已。
- Linux 以 Device Class 来提供面向对象的设备模型，支持热插拔事件，以及用户空间的设备文件系统（sysfs）。
- Linux 忽略了 Unix 那些被认为设计不良的特性（如 STREAMS），和那些不能够被清晰地实现的标准。
- Linux 是免费且自由的，完全体现了 free 这个单词的各种意义。Linux 实

现的特性集是由 Linux 开放的开发模型自由决定的。如果一个特性没有优点，那 Linux 开发者就不会去实现它。反过来，Linux 对于变更采取了非常优秀的态度：改变必须解决一个特定的现实问题、拥有清晰的设计和稳定的实现。因此现代 Unix 变种的某些花哨不实用的特性，如分页式内核内存，没有被 Linux 采纳。

虽然有这些区别，但 Linux 仍然是一个继承自 Unix 的操作系统。

Linux 内核版本

Linux 内核有两种版本：稳定版和开发版。稳定版内核是产品级的发布，适用于大规模部署。稳定版内核的更新一般都只是提供 bug 修复，以及新的设备驱动。相反，开发版内核则会有快速的变化，任何东西都可能改变。随着开发者不停地尝试新方案，内核代码会以非常激烈的方式改变。

Linux 内核用简单的命名规则来区分稳定版和开发版（如图 1.2）。Linux 内核版本使用三或四个数字，加点区分来表示。第一个数是主版本号，第二个是次版本号，第三个是修订版本号。可选的第四个数字表示稳定版本号。次版本号同时也用来区分内核是稳定的还是开发版；偶数值代表稳定版，而奇数值表示开发版。例如内核版本 2.6.30.1 就是稳定版。前面两个数字说明这是 2.6 系列的内核。

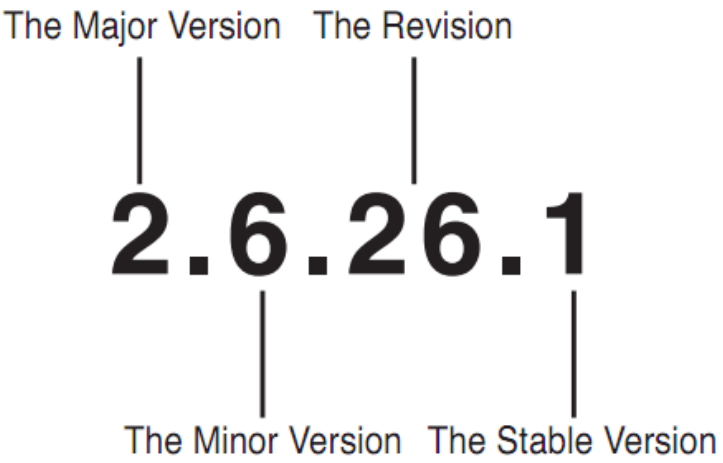

图 1.2 内核版本命名约定

内核的开发过程有一系列的阶段。最开始内核开发者会开发各种新特性，混乱也随之而至。一段时间之后，内核渐趋成熟，于是就宣布特性冻结。从这时候开始，Linus 不再接受新特性。但是还可以继续针对现有特性进行开发。当 Linus 认为内核接近于稳定时，他就会实行代码冻结。随后就只有 bug 修复才会被接受。不久之后（幸运的话），Linus 就会发布新的内核稳定系列的第一个版本。例如内核 1.3 开发版系列稳定为 2.0 版本，2.5 系列稳定为 2.6 版本。

针对每个系列，Linus 都会定期发布新的内核，每个版本都使用新的修订版本号。例如 2.6 内核系列的第一个版本是 2.6.0，下一个则是 2.6.1。这些修订版包括 bug 修复、新的驱动、和新的特性，但是两个修订版（如 2.6.3 和 2.6.4）之间的区别一般都很小。

内核的开发一直这样进行着，直到 2004 年内核开发者大会，到会的内核开发者决定延长 2.6 内核系列，并推后 2.7 开发版。原因是 2.6 内核已被广泛使用、非常稳定、足够成熟，新的特性暂时并不需要。目前看来这是非常明智的，随后几年内 2.6 都会是成熟且胜任的内核。在本书写作的时候，2.7 开发版系列仍然没有摆上桌面。相反每个 2.6 修订版的开发周期变得更长，每次发布也只是很小的开发系列。Andrew Morton（Linus 的副司令）repurposed 他的 2.6-mm 树（一旦内存管理相关的测试发生改变），修改为通用的测试床。这些改变因此也注入了 2.6-mm，当成熟以后，也就进入了 2.6 的开发系列。在最近的几年里，每个 2.6 系列的发布（例如 2.6.29）都需要几个月，而且与前一版本相比都有较大的改变。这种“微型开发系列”被证明是非常成功的，既保持了很高的稳定性，又能适时地引入新特性，而且在短时间内都不会再次改变。实际上内核开发者都认为这种新的发布过程会持续很久。

为了补偿发布频率的降低，内核开发者引入了前面提到的稳定发布版。这种发布（2.6.32.8 中的 8）通常包含严重 Bug 修复、开发内核的反向迁移（如 2.6.33）。有了这种方式，前面的发布就可以继续关注于稳定性。

Linux 内核开发社区

当你开始为 Linux 内核开发代码时，你就已经成为全球内核开发社区的一员。

社区的主要论坛是 Linux 内核邮件列表（通常简称为 lkml）。订阅信息在 http://vger.kernel.org 上。注意这是一个高流量的列表，每天数百条信息，其它阅读者（包括所有核心开发者和 Linus）没有时间和心情处理无意义的事情。但是这个列表是开发过程中一份无价的帮助信息，你可以找到测试人员，接受审查，也可以提出问题。

后面章节会概述内核的开发过程，以及成功参与内核开发社区的详细指导。在此期间，有事没事泡一泡 Linux 内核邮件列表（请保持沉默），是对学习本书最好的一种补充。

开始之前

本书是关于 Linux 内核的：目标、满足这些目标的设计、以及这些设计的实现。本书采用实际的方法，平衡了理论和实践。我的目标是向你展示权威人士对于 Linux 内核设计和实现的理解。同时结合一些个人轶闻和内核 hacking 的技巧，确保本书能让你取得进步，无论你是要开发内核代码、新的设备驱动、还是想更深入地理解 Linux 操作系统。

当你阅读本书时，你需要能够访问 Linux 系统和内核源码。理想的情况是，你已经是一个 Linux 用户，而且已经鼓捣过内核源码，但是需要一些帮助来帮你统筹全局。当然你可能从未用过 Linux，只是想满足好奇心而学习内核的设计。但是如果你想编写自己的内核代码，使用 Linux 内核源码就是必须的。幸运的是内核源码是免费的，赶快使用它吧！

最后，have fun!

第 2 章 内核入门

在这一章，我们介绍 Linux 内核的一些基本知识：去哪里获取源码；怎样编译；怎样安装新内核。然后我们讲解内核和用户空间程序的区别，以及内核使用的常见编程构造。尽管内核在许多方面都是独特的，但归根到底它与任何大型软件项目都区别不大。

获取内核源码

Linux 内核官方网站 http://www.kernel.org，可以获取 Linux 内核源码的完整 tar 包（由 tar 命令创建），也可以获取增量 patch。

除非你有特殊的理由，需要工作于 Linux 源码的旧版本，否则你应该使用最新的代码。kernel.org 的库可以获取最新源码，以及其它主要内核开发者提交的附加 patch。

使用 Git

在过去的几年中，由 Linus 领导的内核黑客们，开始使用新的版本控制系统来管理 Linux 内核源码。Linus 亲自操刀编写了这个被称为 Git 的版本控制系统，它的核心理念是速度。不像传统的 CVS 等系统，Git 是分布式的，因此它的使用和流程对于许多开发者都是不熟悉的。我强烈推荐你使用 Git 下载和管理 Linux 内核源码。

你可以使用 Git 来获取 Linus 推送的最新版本源码树的一份拷贝：

$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6.git

签出之后，你可以随时更新至 Linus 的最新源码树：

```
$ git pull
```

使用这两个命令，你就可以获取并随后更新官方的内核源码。如果要提交和管理你自己的修改，请参考第 20 章“Patches、Hacking 和 Community”。Git 的完整讨论超出了本书的范围，许多在线资源提供了卓越的指导。

安装内核源码

内核 tar 包同时以 GNU zip(gzip)和 bzip2 格式发布。Bzip2 是默认和首选的格式，因为它的压缩率大大好于 gzip。Linux 内核 bzip2 格式的 tar 包命名为 linux-x.y.z.tar.bz2，其中 x.y.z 是该内核源码的版本号。下载内核之后，解压是非常容易的。如果你的 tar 包以 bzip2 格式压缩，运行：

```
$ tar xvjf linux-x.y.z.tar.bz2
```

如果它是以 GNU zip 压缩，则运行：

```
$ tar xvzf linux-x.y.z.tar.gz
```

源码会被解压到 linux-x.y.z 目录下。如果你使用 git 来获取和管理内核源码，就不需要下载 tar 包。只需要运行前面所述的 git clone 命令，git 会下载并取出最新的源码。

源码安装和使用路径

内核源码一般安装在/usr/src/linux 目录下，你不应该使用这个源码树来开发，因为你的 C 库是使用这个内核版本来编译和链接的。此外你也不应该使用 root 权限来修改内核——相反应该在你的 home 目录工作，仅仅使用 root 来安装新内核。即使是安装新内核时，/usr/src/linux 也不应该去改动。

使用 Patch

在整个内核社区里，patch 是互相沟通的交际语。你的代码变更会以 patch 来发布，你也会接收其他人的 patch。增量 patch 为内核树迁移到下一个内核树提供了一种简单的方法。你不需要下载内核源码的每个超大 tar 包，直接应用增量 patch 就可以迁到下一个版本。这节省了所有人的带宽和时间。要应用增量 patch，只需要在内核源码树中运行：

```
$ patch -p1 < ../patch-x.y.z
```

通常特定内核版本的 patch 只能应用于前一个版本的内核。

本章后面会更加深入地讨论 patch 的生成和应用。

内核源码树

内核源码树被分成了许多目录，每个目录还包含了许多子目录。表 2.1 列出了源码树的主目录以及相关描述：

表 2.1 内核源码树的主目录

目录	描述
arch	体系架构相关的源码
block	块 I/O 层
crypto	Crypto API
Documentation	内核源码的文档
drivers	设备驱动
firmware	设备固件
fs	VFS 和独立的文件系统
include	内核头文件
init	内核启动和初始化
ipc	进程间通讯相关的代码
kernel	核心子系统，如调度器
lib	Helper 例程
mm	内存管理子系统和 VM
net	网络子系统
samples	示例，演示代码
scripts	构建内核的脚本
security	Linux 安全模块
sound	声音子系统
usr	早期用户空间代码（称为 initramfs）
tools	开发 Linux 有用的工具

virt	虚拟化基础架构

源码树根目录下还有一些文件也值得注意。COPYING 是内核的许可（GNU GPL v2）。CREDITS 是为内核提交过一定数量代码的开发者列表。MAINTAINERS 列出了内核中子系统和驱动的维护者。Makefile 是内核的构建文件。

构建内核

构建内核非常简单，比编译和安装其它系统级的组件（如 glibc）要容易许多。2.6 内核系列引入了新的配置和构建系统，使得构建工作更加容易，这个改进也受到了极大的欢迎。

配置内核

由于 Linux 源代码是开放的，你可以在编译前进行配置和自定义剪裁。实际上你可以把内核编译成只有特定特性，只支持你需要的设备。构建内核之前必须首先对内核进行配置。由于内核提供了无数特性并支持许多硬件，有很多东西需要配置。我们使用配置选项来控制内核的配置，前缀是 CONFIG，形式如 CONFIG_FEATURE。例如对称多处理器（SMP）由配置选项 CONFIG_SMP 来控制，如果设置了这个选项，SMP 就是可用的；如果未设置则禁用 SMP。配置选项既用来确定哪些文件需要构建，也用作预处理器命令来操纵代码。

控制构建过程的配置选项是布尔或 tristate。布尔选项可以是 yes 和 no，内核特性通常都是布尔型配置选项（如 CONFIG_PREEMPT）。tristate 选项的值可以是 yes、no 和 module。module 表示配置选项被设置，但是被编译为模块（也就是独立的可动态加载的对象）。tristate 选项的 yes 则明确地表示将代码编译进内核主镜像而不作为模块。驱动通常是 tristate 选项。

配置选项也可以是字符串或整数。这些选项不控制构建过程，而是指定一些内核源码可以访问的预处理宏定义的值。例如配置选项可以指定静态分配数组的大小。

厂商内核，例如 Canonical 为 Ubuntu 或 Red Hat 为 Fedora 提供的内核，都已经预编译为发行版的一部分。这些内核一般都启用了必须的内核特性，并且将几乎所有驱动编译为模块。这样可以提供很好的基础内核，并以独立的模块支持大量的硬件。不管这种做法的好坏，作为一个内核黑客，你需要编译自己的内核，学习并包含你自己的模块。

幸运的是内核提供了多个工具来帮助配置。最简单的工具是基于文本的命令行实用工具：

```
$ make config
```

这个工具依次检查所有选项，一次一个，并询问用户选择 yes、no、或 module。这样做下来需要花费大量时间，所以除非你是按小时收费，否则应该使用基于 ncurses 的图形工具：

```
$ make menuconfig
```

或者基于 gtk+的图形工具：

```
$ make gconfig
```

这些工具把许多配置选项划分为不同的类别，例如“预处理器类型和特性”，你可以选择不同的类别，查看内核的选项，并且修改它们的值。

下面命令创建当前体系架构下的默认配置：

```
$ make defconfig
```

尽管这些默认值看上去有一点武断（谣传 i386 架构下使用的就是 Linus 的配置），但是如果你从未配置过内核，它提供了一个很好的起点。你可以先运行这个命令，然后再回过头来确认你的硬件需要的配置选项是否启用。

配置选项保存在内核源码根目录下的.config 文件中。你会发现直接编辑这个文件要容易许多（多数内核开发者都是这样干的），查找并修改某个配置选项的值是非常简单的。完成修改配置文件之后，或者为新内核树使用已有配置文件时，你应该验证并更新配置：

```
$ make oldconfig
```

在构建内核之前你应该先运行这个命令。

配置选项 CONFIG_IKCONFIG_PROC 会把完整的内核配置文件压缩存储为/proc/config.gz。这使得构建新内核时克隆当前配置非常简单。如果你的当前内核启用了这个选项，你就可以从/proc 复制所有配置，并用来构建新内核：

```
$ zcat /proc/config.gz > .config
$ make oldconfig
```

在配置完成之后，使用一个命令就可以构建内核：

```
$ make
```

2.6 版本和之前的内核不同，在构建内核之前不再需要运行 make dep，内核可以自动维护依赖关系。你也不需要指定特定的构建类型，例如 bzImage，或者单独构建模块。默认的 Makefile 规则能够处理所有事情。

最小化构建噪音

有一个技巧可以最小化构建的噪音，但仍然可以查看到警告和错误信息，那就是重定向 make 的输出：

```
$ make > ../detritus
```

如果你需要查看构建输出，可以阅读这个文件。由于警告和错误会输出到标准错误，通常你并不需要重定向到文件。实际上我自己会这样做：

```
$ make > /dev/null
```

这样会把所有没用的信息输出到那个无尽且邪恶的/dev/null。

产生多个构建作业

make 程序有一个特性，可以把构建过程拆分为多个并行的*作业*。每个任务都独立且并行地运行，在多处理器系统中可以大大加速构建过程。它还能提高处理器利用率，因为构建大型源码树会花费大量时间等待 I/O（处理器空闲等待 I/O 请求完成）。

make 默认只产生一个作业，因为 Makefile 经常会包含错误的依赖信息。如果依赖关系存在错误，多个作业可能互相影响，导致构建出错。内核的 Makefile 有正确的依赖信息，因此产生多个作业并不会导致错误。要多个作业同时构建内核，使用下面命令：

```
$ make –jn
```

这里 n 表示要产生的作业数量。实践中通常每个处理器产生一到两个作业。例如在一个 16 核机器中，你可能会这样做：

```
$ make -j32 > /dev/null
```

使用诸如 distcc 或 ccache 等优秀工具也可以大量地提高内核的构建速度。

安装新内核

内核构建完成后，你需要安装它。安装方法是体系架构相关的——也与 boot loader 有关——参考你使用的启动引导器的文档，确定内核镜像的存放位置，以及怎样设置为启动。你需要保留至少一个已知可用的内核，以防新内核出现问题！

举个例子，x86 使用 grub 作为启动引导，你需要把/arch/i386/boot/bzImage 复制到/boot，并给它起个形如 vmlinuz-version 的名字。然后编辑/boot/grub/grub.conf 文件，为新内核增加一个新的启动入口。使用 LILO 的系统则需要编辑/etc/lilo.conf，并重新运行 lilo。

幸运的是安装模块是自动的，而且与体系架构无关。root 用户可以运行：

```
% make modules_install
```

这样就把所有已编译的模块安装到/lib/modules 下的相应路径。

构建过程还会在内核源码树根目录中创建 System.map 文件。它包含符号查找表，映射内核符号到它们的起始地址。调试时可以使用它将内存地址转换为函数和变量名。

内核的独特之处

Linux 内核相比于普通的用户空间应用，有一些独特的地方。尽管这些区别并没有使内核代码的开发难度高于用户空间代码的开发，但它们确实使得内核代码非常不同。

这些特征使内核成为一头性质不同的野兽。一些平常规则是弯曲的，其它则是全新的。尽管有些区别是明显的（我们都知道内核可以做任何事情），其它一些就不那么明显。内核最重要的区别主要有：

- 内核不能访问 C 库和标准 C 头文件。
- 内核使用 GNU C 编码。
- 内核缺乏用户空间那样的内存保护。
- 内核执行浮点操作很困难。
- 内核只有很小的单进程固定大小的堆栈。
- 内核有异步中断、抢占式、支持 SMP，因此必须时刻注意同步和并行。
- 可移植性是非常重要的。

让我们简短地一个一个来看一下这些问题，因为所有内核开发者都必须记住上面的区别。

没有 libc 和标准头文件

和用户空间应用不一样，内核不与标准 C 库链接（也不链接其它库）。这样做的原因有很多，而且还存在“鸡生蛋还是蛋生鸡”的问题。但最主要的原因是速度和大小。完整的 C 库对于内核来说太大，而且也太低效。

不要因此而烦躁：内核已经实现了许多常用的 libc 函数。例如常见的字符串操作函数就在 lib/string.c 中实现。只需要包含头文件<linux/string.h>就可以使用这些字符串函数。

头文件

在本书中，当我说头文件时，我指的是内核源码树中的头文件。内核源码不能包含外部的头文件，就像它不能使用外部库一样。

基本的头文件都存放在内核源码树根目录下的 include/目录下。例如头文件<linux/inotify.h>在源码树中就是 include/linux/inotify.h。

有一些体系架构相关的头文件被放在 arch/<architecture>/include/asm 目录下。例如编译 x86 体系架构，你的体系架构相关的头文件就在 arch/x86/include/asm 目录中。包含这些头文件的源码只需要使用 asm/前缀。例如<asm/ioctl.h>。

在缺失的 C 库函数中，最熟悉的莫过于 printf()。内核确实没有调用 printf()，但它提供了 printk()函数，和 printf()是非常类似的。printk()函数把格式化的字符串复制到内核日志缓冲区，一般是供 syslog 程序来读取。使用方法如下：

```
printk("Hello world! A string '%s' and an integer '%d'\n", str, i);
```

printf()和 printk()的一个差别是 printk()允许你指定优先级标志。这个标志被 syslog 用来确定如何显示内核消息。下面是优先级的一个例子：

```
printk(KERN_ERR "this is an error!\n");
```

注意 KERN_ERR 和打印消息之间没有逗号。这是有意设计的，优先级标志是一个预定义的字符串常量，编译时会与打印消息连接在一起。本书从头到尾一直都在使用 printk()。

GNU C

和其它 Unix 内核一样，Linux 内核也是用 C 语言编写的，不过 Linux 内核没有严格按照 ANSI C 来编码。相反在适当的时候，内核开发者会使用 gcc 的各种 C 语言扩展特性（GNU 编译器家族中的 C 编译器，被用来编译内核，以及 Linux 系统中使用 C 语言编写的任何东西）。

内核开发者同时使用了 ISO C99 和 GNU C 对 C 语言的扩展。这样就使得 Linux 内核选择了 gcc 作为编译器，虽然最近 Intel C 编译器也支持 gcc 的大部分特性，也可以编译 Linux 内核。内核支持的最老版本是 gcc 3.2，推荐使用 gcc 4.4 或更新版本。内核使用的 ISO C99 扩展没有什么特别之处，由于 C99 是官方对 C 语言的修订，现在慢慢开始也被其它代码使用。内核代码相比 ANSI C 更让人不熟悉的地方是 GNU C 提供的扩展。让我们来看看其中一些比较有趣的扩展；这也是内核与你所熟悉的其它项目的主要区别所在。

inline 函数

C99 和 GNU C 都支持 inline 函数。inline 函数正如其名，会在每个函数调用点插入内联的代码。这样就消除了函数调用和返回带来的开销（寄存器保存和还原），也可能带来更好的优化效果，因为编译器可以把调用方和被调用函数放在一起优化。inline 函数的缺点（没有免费的午餐）是增加了代码尺寸，因为函数的内容被复制到所有被调用的地方，这样也就增加了内存消耗和指令 cache 印迹。内核开发者对所有时间至上的小函数使用 inline。大型函数，特别是那些被多次使用，或者不是特别强烈要求时间的函数，使用 inline 是有问题的。

inline 函数的定义使用关键字 static 和 inline 来声明，例如：

```
static inline void wolf(unsigned long tail_size)
```

函数声明必须放在使用的前面，否则编译器就不能使函数 inline。通常的实践是把 inline 函数放在头文件中。由于它们标记为 static，因此不会被导出。如果 inline 函数只在一个文件中使用，就可以直接把它放在文件的顶部。

在 Linux 内核中，优先使用 inline 函数而不是复杂的宏定义，因为这样拥有类型安全和可读性高等优点。

inline 汇编

gcc C 编译器允许普通 C 函数内嵌汇编指令。当然这个特性只有特定于某个体系架构那部分内核才会使用。

asm()编译器命令用来 inline 汇编代码。例如下面 inline 汇编指令执行 x86 处

理器的 rdtsc 指令，返回时间戳（tsc）寄存器的值：

```
unsigned int low, high;
asm volatile("rdtsc" : "=a" (low), "=d" (high));
/* low and high now contain the lower and upper
    32-bits of the 64-bit tsc */
```

Linux 内核由 C 和汇编混合编写而成，底层体系架构相关及要求速度的代码使用汇编。但内核代码的大部分都是用 C 编写的。

分支注解

gcc C 编译器有一个内建的指令，可以优化条件分支为非常有可能和不太可能。编译器使用这个指示对分支进行适当的优化。内核把它封装为非常容易使用的宏 likely()和 unlikely()。

例如下面的 if 语句：

```
if (error) {
    /* ... */
}
```

下面代码标记这个分支为不太可能发生：

```
/* we predict 'error' is nearly always zero ... */
if (unlikely(error)) {
    /* ... */
}
```

反过来我们也可以标记为很可能发生：

```
/* we predict 'success' is nearly always nonzero ... */
if (likely(success)) {
    /* ... */
}
```

只有在分支的某个方向占据绝对优势，或者你希望优化某个特定情况而无视另一种情况时，才应该使用分支注解指令。这一点非常重要：当分支被正确标注时可以提高性能，但分支标注错误时则会有性能损失。正如上面例子所示，unlikely()和 likely()的常见使用就是错误条件。内核更多是使用 unlikely()，因为 if 语句通常都指示特殊情况。

没有内存保护

当用户空间应用试图访问非法内存时，内核会捕获这个错误，并发送 SIGSEGV 信号来杀掉进程。但是如果内核试图访问非法内存，结果就很难控制了。（毕竟，谁来负责照看内核呢）。内核中的内存违规会导致 oops，这是一个主要的内核错误。不用说我们都应该避免访问非法内存，例如解引用 NULL 指针，只是在内核中非法内存的结果更严重而已。

此外内核内存是不分页的。因此你每消耗一个字节内存，物理内存就会少一个字节。下次你要为内核增加一个特性时，请牢记这一点。

不能轻松地使用浮点

用户空间进程使用浮点指令时，内核会负责管理整数到浮点模式的切换。内核使用浮点指令时需要做的事情与体系架构相关，通常内核需要捕获一个 trap，然后开始从整数向浮点模式转换。

和用户空间不一样，内核对浮点缺乏无缝的支持，因为它不能轻松地 trap 自己。在内核中使用浮点数需要手动保存和还原浮点寄存器，以及完成其它一些事情。最简单的办法是：不要使用浮点数！内核只在极少数情况下使用了浮点数。

固定大小的小堆栈

用户空间应用可以直接在堆栈中静态分配许多变量，使用大型的结构体和数千元素的数组。这是合法的，因为用户空间拥有大型的堆栈，并且可以动态增长。（老的不那么先进的操作系统，例如 DOS 上的开发者，可能会回想起当年用户空间也只有固定大小堆栈的日子）。

内核堆栈既不大，也不是动态的；它固定大小而且非常小。内核堆栈的具体大小和体系架构相关。在 x86 上，堆栈大小可以在编译时进行配置，可以设为 4KB 或 8KB。历史上内核堆栈大小是两个页，通常也意味着在 32 位机器上是 8KB，64 位机器上是 16KB。不管怎么样，它的大小是绝对固定的。每个进程都会拥有

自己的堆栈。

内核堆栈将在后续章节中更为深入地讨论。

同步和并行

内核很容易受到竞争条件的影响。和单线程的用户空间应用不一样，内核的很多属性允许并行访问共享资源，因此需要同步来防止竞争。特别是：

- Linux 是抢占式多任务操作系统。内核的进程调度器负责调度所有进程。内核必须在这些任务中进行同步。
- Linux 支持对称多处理器（SMP）。因此如果没有适当的保护，在两个或多个处理器中同时执行的内核代码就会访问相同的资源。
- 中断对于当前正在执行的代码是异步发生的。因此如果没有适当的保护，中断就会在访问资源的中途发生，然后中断处理器也可能访问相同的这个资源。
- Linux 内核是抢占式的。因此如果没有适当的保护，内核代码会被访问相同资源的不同代码所抢占。

解决竞争条件的典型方法包括自旋锁和信号量。后续章节会提供同步和并行的深入讨论。

可移植性非常重要

用户空间应用可能不需要考虑可移植性，但 Linux 是一个可移植的操作系统，必须保持统一。这意味着体系架构无关的 C 代码必须能够在不同的系统上编译和运行，体系架构相关的代码则必须被正确地隔离在内核源码树的系统相关目录下。

这里有许多规则对我们大有帮助，如保持自然的大小端、支持 64 位、不假定字大小和页大小等等。后面章节会更加深入地讨论可移植性。

小结

诚然，内核拥有独特的性质。内核有自己的规则，并管理整个系统。但是 Linux 内核的复杂度和入门的难度，和其它大型软件项目并没有质的差别。迈向 Linux 内核开发之路最重要的一步就是意识到内核没有什么可怕之处。不熟悉？没问题；难以克服？不可能！

本章和上一章提供了本书后续章节的基础材料。在后面的每一章，我们讲解特定的内核概念或子系统。在这个过程中，我们要求你阅读和修改内核源码。只有实际地阅读和试验代码，你才能真正地理解它。而且内核源码是自由且免费的——你应该好好利用它！

第 3 章 进程管理

本章介绍进程的概念，进程是 Unix 操作系统中最基本的抽象之一。我们对进程和相关的概念（如线程）进行了定义，然后讨论 Linux 内核如何管理每个进程：内核怎样枚举进程、怎样创建进程、进程最终如何死亡。由于运行用户应用是我们使用操作系统的主要用途，因此进程管理是所有操作系统内核的核心部分，包括 Linux。

进程

进程是正在执行中的程序（存储在某种介质中的目标代码）。但是进程又不仅仅只是执行中的程序代码（Unix 中通常称为 text 段）。进程还包含一组资源：如打开的文件、未决信号、内部内核数据、处理器状态、内存地址空间、一个或多个内存映射、一个或多个执行线程、以及包含全局变量的 data 段。因此进程实际上是正在运行的程序代码的生存状态。内核必须高效而透明地管理所有这些细节。

执行线程（通常简称为线程），是进程中的活动对象。每个线程都有自己单独的程序计数器、进程堆栈、和一组处理器寄存器。内核对单个线程而不是进程进行调度。在传统的 Unix 系统中，每个进程只包含一个线程。不过在现代系统中，多线程程序已经很常见。后面你将会看到，Linux 对线程的实现非常特殊：它不区分线程和进程。对 Linux 来说，线程只不过是一种特殊的进程。

在现代操作系统中，进程提供两个虚拟化：虚拟化的处理器和虚拟内存。虚拟处理器给予进程自己独占整个系统的假象，虽然实际上已经数百个进程共享该处理器。第 4 章“进程调度”讨论虚拟处理器。虚拟内存允许进程就像独占系统所有内存那样分配和管理自己的内存。第 12 章“内存管理”会讨论虚拟内存。有趣的是，线程共享虚拟内存的抽象，但却会拥有自己的虚拟处理器。

程序本身并不是进程：进程是活动的程序及相关的资源。实际上，可以存在执行相同程序的多个进程。而且多个进程还可以共享各种资源，如打开的文件和地址空间等。

进程从被创建那一刻开始自己的生命，在 Linux 中就是使用 fork()系统调用，通过复制一个现有进程来创建新的进程。调用 fork()的进程称为父进程，而新创建的进程就称为子进程。父进程 fork()之后继续执行，子进程也从相同位置开始执行：也就是 fork()返回的位置。fork()系统调用从内核中返回两次：父进程和子进程各返回一次。

通常 fork()之后都需要执行一个新的不同的程序。exec()函数家族创建新的地址空间并装载新的程序。在当前 Linux 内核，fork()实际上是通过 clone()系统调用来实现的，接下来我们会对其进行讨论。

最后，程序通过 exit()系统调用退出。这个函数结束进程，并释放所有资源。父进程可以通过 wait4()[1] 系统调用来询问已终止子进程的状态，这样就允许进程等待特定进程的终止。当进程退出时，会处于特殊的 zombie（僵尸）状态，表示进程已终止，直到父进程调用 wait()或 waitpid()才会结束僵尸状态。

注：进程的另一个名字是任务（Task）。Linux 内核内部就是按任务来引用进程。在本书中，我交互使用这两个术语，不过当我说任务时，通常我是从内核的角度来指代进程。

进程描述符和 Task 结构体