THE LINUX PROGRAMMING INTERFACE

A Linux and UNIX® System Programming Handbook

MICHAEL KERRISK

no starch press

THE LINUX PROGRAMMING INTERFACE

A Linux and UNIX System Programming Handbook

MICHAEL KERRISK

no starch press

San Francisco

Translated by: Kevin

目录

前言

主题

本书描述 Linux 编程接口——Linux（UNIX 操作系统的一种免费实现）提供的系统调用、库函数、和其它底层接口。这些接口被直接或间接地使用在 Linux 上运行的每个程序中。它们允许应用程序完成各种任务：如文件 I/O、创建删除文件和目录、创建新进程、执行程序、设置定时器、本机进程和线程间通信、通过网络连接的不同机器进程间通信等等。这些底层接口有时候也叫做系统编程接口。

尽管本书关注于 Linux，但我也非常注意标准和可移植性问题，清晰地区分了 Linux 特有的接口、多数 UNIX 实现共有的特性、以及 POSIX 和 Single UNIX Specification 标准定义的特性。因此本书也提供了 UNIX/POSIX 编程接口的详尽描述，能够适用于编写 UNIX 系统应用或跨平台应用的程序员。

目标读者

本书主要面向以下读者：

- 为 Linux、UNIX、或者其它遵循 POSIX 的系统开发应用的程序员和软件设计师；
- 在 Linux、UNIX、或其它操作系统之间移植应用的程序员；
- Linux 或 UNIX 系统编程课程的教师和高年级学生；
- 希望深入理解 Linux/UNIX 编程接口，以及系统软件是如何实现的系统管理员和“高级用户”。

我假设你拥有一定的编程经验，但不要求系统编程经验。我还假设你了解 C 编程语言，并且知道如何使用 shell 和常用的 Linux 或 UNIX 命令。如果你是 Linux/UNIX 的新手，你会发现第 2 章非常有用，我们以程序员的视角来讲述 Linux 和 UNIX 的基础概念。

Linux 和 UNIX

本书原本可以纯粹地讲解标准 UNIX（也就是 POSIX）系统编程，因为 UNIX 和 Linux 的大多数特性都是相同的。不过虽然编写可移植程序是很好的目标，理解 Linux 对标准 UNIX 编程接口的扩展也是非常重要的。理由之一是 Linux 非常流行；其二是有时候为了性能、或使用标准 UNIX 没有的功能，我们不得不使用非标准的扩展（所有 UNIX 实现都提供类似的非标准扩展）。

因此本书在适用于标准 UNIX 的程序员时，还提供了 Linux 特定编程特性的详细描述。这些特性包括：

- epoll，获得文件 I/O 事件通知的机制；
- inotify，监控文件和目录改变的机制；
- 能力，授予进程一组超级用户能力的机制；
- 扩展属性；
- i-node 标志；
- clone()系统调用；
- /proc 文件系统
- Linux 对文件 I/O、信号、定时器、线程、共享库、进程间通信、和 socket 的特殊实现细节。

使用和组织

你至少可以按两种方式来使用本书：

- 作为 Linux/UNIX 编程接口的介绍手册。你可以从头到尾阅读本书。后续章节建立在之前章节的基础之上，我尽量避免依赖后续章节的情况。
- 作为 Linux/UNIX 编程接口的索引参考手册。详细的索引和频繁的交叉引用，允许你随机地阅读任何主题。

我把本书分为以下几部分：

1. 背景和概念：UNIX、C 和 Linux 的历史；UNIX 标准简介（第 1 章）；以程

序员的视角介绍 Linux 和 UNIX 的基本概念（第 2 章）；Linux 和 UNIX 系统编程的基本概念（第 3 章）。

2. 系统编程接口的基础特性：文件 I/O（第 4 章和第 5 章）；进程（第 6 章）；内存分配（第 7 章）；用户和组（第 8 章）；进程凭证（第 9 章）；定时器（第 10 章）；系统限制和选项（第 11 章）；获取系统和进程信息（第 12 章）。
3. 系统编程接口的高级特性：文件 I/O 缓冲（第 13 章）；文件系统（第 14 章）；文件属性（第 15 章）；扩展属性（第 16 章）；访问控制列表（第 17 章）；目录和链接（第 18 章）；监控文件事件（第 19 章）；信号（第 20 章到第 22 章）；定时器（第 23 章）。
4. 进程、程序、和线程：进程创建、进程结束、监控子进程、执行程序（第 24 章到第 28 章）；POSIX 线程（第 29 章到第 33 章）。
5. 进程和程序的高级主题：进程组、会话、任务控制（第 34 章）；进程优先级和调度（第 35 章）；进程资源（第 36 章）；daemon（第 37 章）；编写安全的特权程序（第 38 章）；能力（第 39 章）；登录会计（第 40 章）；共享库（第 41 章到第 42 章）。
6. 进程间通信（IPC）：IPC 简介（第 43 章）；管道和 FIFO（第 44 章）；System V IPC——消息队列、信号量、共享内存（第 45 章到第 48 章）；内存映射（第 49 章）；虚拟内存操作（第 50 章）；POSIX IPC——消息队列、信号量、共享内存（第 51 章到第 54 章）；文件锁（第 55 章）。
7. Socket 和网络编程：IPC 和 socket 网络编程（第 56 章到第 61 章）。
8. 高级 I/O 主题：终端（第 62 章）；可选 I/O 模型（第 63 章）；伪终端（第 64 章）。

例子程序

我用短小但完整的例子程序来阐述多数接口的使用方法，这些例子都被设计为很容易就能从命令行体验，来查看不同的系统调用和库函数如何工作。所以本书包含大量的示例代码——大概 15000 行 C 代码和 shell 会话日志。

尽管阅读和试验例子程序是不错的起点，掌握本书讨论的概念最有效的方法是编写代码，按你的想法修改例子程序，或者编写新程序都可以。

本书的所有源代码都可以在网站上下载。源代码包含许多书中没有的程序。这些程序的目的和细节在注释中都有相关描述。我提供了 Makefile 编译这些程序，以及一个 README 文件，给出了例子程序更多的细节信息。

源代码采用 GNU Affero 通用公共授权版本 3，可以自由分发和修改。源代码中也包含一份该协议的拷贝。

练习

多数章节都以一组练习结束，其中一些是要你按不同方式来试验例子程序，另外一些是该章讨论过的概念相关的问题，还有就是要求你来编写代码以巩固你对本书的理解。你可以在附录 F 找到部分练习的解答。

标准和可移植性

贯穿整本书，我都对可移植性问题特别地关注。你会发现很多相关标准的引用，特别是 POSIX.1-2001 和 Single UNIX 规范版本 3（SUSv3）标准。同时你还将看到这些标准最新修订的细节改变，也就是 POSIX.1-2008 和 SUSv4 标准。（由于 SUSv3 是更大的修订版本，也是本书编写时最广泛有效的 UNIX 标准，本书讨论的标准大多是 SUSv3，并标注出 SUSv4 不同的地方。除非我明确地提到，你可以假设我们对 SUSv3 规范的描述也适用于 SUSv4）。

对于那些不是标准的特性，我会指出在不同 UNIX 实现间的差别。我还会突出那些 Linux 特定的特性，以及 Linux 与其它 UNIX 对系统调用和库函数实现上的细小差别。当某个特性我没有明确指出是 Linux 专有时，你也通常可以假设它在多数或所有 UNIX 上都有实现。

本书大多数例子程序我都在 Solaris、FreeBSD、Mac OS X、Tru64 UNIX、和 HP-UX 上测试通过（除了那些 Linux 独有的特性）。为了提高代码在这些系统上的可移植性，本书网站上提供的某些例子程序有一些额外的代码。

Linux 内核和 C 库版本

本书主要关注 Linux 2.6.x 系列，这是本书写作时最广泛使用的内核版本。Linux 2.4 的某些细节也会提到，我也会指出 Linux 2.4 和 2.6 的区别。当 Linux 2.6.x 系列出现了新特性时（例如 2.6.34），我也会特别指出相应的内核版本号。

至于 C 库，本书则主要关注于 GNU C 库（glibc）版本 2。当然，glibc 2.x 系列版本存在差异时，我也会特别指出。

在本书即将印刷时，Linux 内核刚刚发布了 2.6.35 版本，glibc 则已经发布 2.12 版本。本书完全适用于这两个软件版本。Linux 内核和 glibc 将来接口的变化，会在本书的网站上列出。

其它语言使用编程接口

尽管例子程序用 C 语言编写，你也可以在其它编程语言中使用本书讨论的接口——例如编译型语言 C++、Pascal、Modula、Ada、FORTRAN、D；解释型语言 Perl、Python、Ruby 等。（Java 则需要采用一种不同的方式 JNI）。不同的语言要获取必要的常量定义和函数声明，需要使用不同的技术（C++除外），另外传递函数参数时可能也需要一点额外的工作。此外就没有太大的区别了，核心概念其实都是一样的。因此即使你使用其它的编程语言，你也会发现本书提供的信息是适用的。

关于作者

（略）

致谢

（略）

许可

电子工程学会和开放组织非常友好地许可我引用 IEEE Std 1003.1, 2004 版本，以及信息技术标准——可移植操作系统接口（POSIX），开放组织基本规范 Issue6。完整的标准可以在 http://www.unix.org/version3/online.html 上在线查阅。

网站和例子程序源代码

你可以在 http://www.man7.org/tlpi 上找到关于本书更多的信息，包括勘误表和例子程序的源代码。

反馈

我非常欢迎代码 bug 报告、代码改进建议、以及代码可移植性的提高。同样我也欢迎本书的 bug 报告和改进建议。由于 Linux 编程接口总是在变化，我也非常高兴能获得关于本书将来版本的改进意见，包括新特性和变化特性。

Michael Timothy Kerrisk

Munich, Germany and Christchurch, New Zealand

August 2010

mtk@man7.org

第 1 章 历史和标准

Linux 是 UNIX 操作系统家族的成员之一。在计算机的术语里，UNIX 已经拥有很悠久的历史。第 1 章的前半部分简述 UNIX 的历史。我们首先描述 UNIX 系统和 C 编程语言的起源，然后讲述导致 Linux 发展成为今天这个样子的两个关键因素：GNU 项目和 Linux 内核的开发。

UNIX 系统最显著的特点之一是它的开发不是被一个厂商或组织控制。相反许多商业和非商业组织都为 UNIX 的发展做出了贡献。UNIX 也因此增加了许多革新的特性，但同时也导致 UNIX 各个实现之间的分歧越来越大，编写一个能运行于所有 UNIX 实现的应用也变得非常困难。于是产生了 UNIX 的标准化运动，我们将在本章后半部分进行讨论。

1.1 UNIX 和 C 简史

第一个 UNIX 由贝尔实验室（电话公司 AT&T 的一个部门）的 Ken Thompson 在 1969 年开发完成（Linus Torvalds 也正是在这一年出生）。这个 UNIX 是用汇编为 Digital PDP-7 微计算机编写。UNIX 这个名字和 MULTICS（Multiplexed Information and Computing Service）有关，后者是 AT&T 与麻省理工学院（MIT）和通用电子之前合作开发的操作系统项目。（由于该项目最初的失败，没有能够开发出一个有用的系统，当时 AT&T 已经退出项目）。Thompson 的新操作系统从 MULTICS 中借用了一些设计，包括树型结构文件系统、对命令解释执行采用独立的程序（shell）、以及把文件当作无结构的字节流。

在 1970 年， UNIX 使用汇编语言为新的 Digital PDP-11 微计算机重新编写，这个 PDP-11 的遗留痕迹至今仍然可以在多数 UNIX 实现中找到，包括 Linux。

不久之后，Dennis Ritchie，Thompson 在贝尔实验室的一个同事，设计和实现了 C 编程语言。这是一个进化的过程，C 起源于更早的解释语言 B，最初由 Thompson 实现了 B 语言，并从一个更早的语言 BCPL 中借鉴了许多想法。到 1973 年，C 已经成熟到 UNIX 内核几乎可以全部使用其重写。UNIX 也因此成为最早使用高级语言编写的操作系统，使其迁移到其它硬件体系架构成为可能的重要因素。

C 语言的这个起源，解释了 C 和 C++成为今天最广泛的系统编程语言的原因。之前广泛使用的语言都是为其它目的而设计的：FORTRAN 为工程师和科学家完成数学任务；COBOL 为商业系统处理面向记录的数据流。C 填补了一个空白，和 FORTRAN、COBOL 不一样的是，C 语言是几个人为了一个目标而设计的：开发一个高级语言来实现 UNIX 内核和相关的软件。和 UNIX 操作系统本身一样，C 由专业的程序员为自身所设计。所产生的语言是小巧、高效、强大、简洁、模块化、注重实效、和一致的。

UNIX 第一至第六版

在 1969 年到 1979 年间，UNIX 发布了一系列版本。本质上就是 AT&T 对 UNIX 开发进展的一个快照。UNIX 最初的六个版本发布时间如下：

- 第一版，1971 年 11 月：此时 UNIX 还运行在 PDP-11 上，已经拥有一个 FORTRAN 编译器，和许多今天依然在使用的工具，包括 ar, cat, chmod, chown, cp, dc, ed, find, ln, ls, mail, mkdir, mv, rm, sh, su, who。
- 第二版，1972 年 6 月：UNIX 安装在 AT&T 内部的 10 台机器上。
- 第三版，1973 年 2 月：这个版本包含一个 C 编译器和管道的最初实现。
- 第四版，1973 年 11 月：第一个几乎全部用 C 编写的版本。
- 第五版，1974 年 6 月：此时 UNIX 已经安装在超过 50 个系统中。
- 第六版，1975 年 5 月：这是第一个在 AT&T 范围外广泛使用的版本。

在这些版本发布的过程中，UNIX 的使用和声望得到了扩展，首先在 AT&T 内部，随后在外部。Communications of the ACM 杂志发表的一篇关于 UNIX 的论文也为此做出了巨大贡献。

当时 AT&T 正在接受美国电话系统对其垄断的政府制裁。AT&T 与美国政府的协议禁止其销售软件，这也意味着 AT&T 不能把 UNIX 作为产品销售。相反，从 1974 年的第五版开始，特别是第六版，AT&T 授权大学免费使用 UNIX。针对大学的 UNIX 发布版包含文档和内核源代码（当时大约 10000 行）。

AT&T 对大学发布 UNIX 极大地促进了 UNIX 的使用和流行，到 1977 年 UNIX

已经运行在 500 个地方，包括 125 所美国大学和其它一些国家。当时的商业操作系统非常昂贵，而 UNIX 为大学提供了一个交互式多用户的操作系统，即便宜又强大。同时 UNIX 还给大学计算机科学研究提供 UNIX 操作系统的源代码，他们可以修改并提供给学生学习和体验。很多学生学习了 UNIX 之后，就成为了 UNIX 的布道者。其它则加入或组建自己的公司，销售运行着 UNIX 操作系统的计算机工作站。

BSD 和 System V 的诞生

1979 年 1 月 UNIX 发布了第七版，改进了系统的可靠性，提供了一个增强的文件系统。这个发布版还包含一些新的工具，包括：awk, make, sed, tar, uucp, Bourne shell, 和 FORTRAN 77 编译器。第七版的发布对于 UNIX 来说具有重要意义，因为从这一刻起，UNIX 产生了两个重要的变种：BSD 和 System V，它们的起源我们马上就会简要地描述。

Thompson 在 1975/1976 学年回到自己的母校，加州大学伯克利分校担任客座教授。在那里他和几个毕业生为 UNIX 增加了许多新特性。（其中一个学生 Bill Joy，随后与别人一起组建了 Sun Microsystems，成为 UNIX 工作站市场早期参与者）。Berkeley 开发了许多新的工具和特性，包括 C shell、vi 编辑器、改进的文件系统（Berkeley Fast File System）、sendmail、Pascal 编译器、新的 Digital VAX 体系架构下的虚拟内存管理等。

在 Berkeley Software Distribution（BSD）的授权许可下，这个版本的 UNIX，包括它的源代码，被广泛地发布出去。1979 年发布了第一个完整发行版 3BSD（更早的 Berkeley-BSD 和 2BSD，只是增加 Berkeley 开发的新工具，而不是完整的 UNIX 发行版）。

到 1983，加州大学伯克利的计算机系统研究组织（Computer Systems Research Group）发布了 4.2BSD。这是一个重大的发行版，因为它包含了完整的 TCP/IP 实现，包括 socket 应用编程接口（API）和许多网络工具。4.2BSD 和它的前任 4.1BSD 被广泛发布于全世界的许多大学。它们也构成了 Sun 公司的 UNIX 变种，SunOS（1983 首次发布）的基础。其它重要的 BSD 发布包括 1986 年的 4.3BSD，以及

1993 年的最终发布版：4.4BSD。

与此同时，US 反托拉斯诉讼强制 AT&T 解散（法律诉讼起于 1970 年代中期，1982 年解散生效），由于在电话系统中不再垄断，公司被允许运营 UNIX。结果就是 1981 年 System III 的诞生。AT&T 的 UNIX 支持组（USG）负责开发 System III，它雇佣了数百名开发者来增强 UNIX，和开发 UNIX 应用（著名的有 document preparation package 和软件开发工具）。随后在 1983 年发布了 System V(5)的第一个版本，一系列的小发布版后最终是 1989 年的 System V 发布版 4（SVR4），到这时 System V 已经吸收了 BSD 的许多特性，包括网络基础设施。System V 授权给许多商业厂商，这些厂商使用 System V 作为自己 UNIX 实现的基础。

因此到 1980 年代末，除了各种 BSD 发布版在大学广泛使用，UNIX 还在许多硬件上拥有各种商业实现：包括 Sun 的 SunOS 及随后的 Solaris、Digital 的 Ultrix 和 OSF/1（经过一系列的改名和收购之后，成为了今天的 HP Tru64 UNIX）、IBM 的 AIX、Hewlett-Packard（HP）的 HP-UX、NeXT 的 NeXTStep、Apple Macintosh 的 A/UX、Microsoft 和 SCO 为 Intel x86-32 体系架构开发的 XENIX。（本书将 Linux 的 x86-32 实现统一称为 Linux/x86-32）。这种状况和当时典型的私有硬件/操作系统的方式完全不同，后者通常是厂商只生产一个或少数私有计算机芯片体系架构，然后在上面销售自己的私有操作系统。多数厂商系统的这种私有属性，意味着购买受限于一个厂商。切换到另一种私有操作系统和硬件平台会非常昂贵，因为需要迁移现有应用并进行相关的重新训练。这个因素再加上各个厂商便宜的单用户 UNIX 工作站，使得可移植的 UNIX 系统对商业应用非常具有吸引力。

1.2 Linux 简史

Linux 这个术语通常引用基于 Linux 内核的完整的类 UNIX 操作系统。不过这是错误的叫法，因为典型商业 Linux 发行版的许多关键组件，都起源于另一个项目，这个项目比 Linux 要早好几年。

1.2.1 GNU 项目

Richard Stallman 是一个天才程序员，曾工作于 MIT，他在 1984 年开始考虑实现一个"Free" UNIX。Stallman 对"free"的观点是精神上的自由，并且定义在法律层面上，而不仅仅是免费（参考 http://www.gnu.org/philosophy/free-sw.html）。无论如何，Stallman 倡导的自由也就意味着软件（如操作系统）应该免费或非常便宜。

Stallman 大大影响了厂商对私有操作系统系统附加的限制。这些限制意味着购买计算机软件通常不包含源代码，而且通常不能对该软件进行复制、修改、和分发。Stallman 指出这种形式鼓励程序员互相竞争并且保密自己的工作，而不是互相合作和共享成果。

于是 Stallman 创建了 GNU 项目（GNU's not UNIX），目标是开发一个完整、自由、类 UNIX 的系统，包含一个内核和所有相关的软件包，并且鼓励其它人参与该项目。到 1985 年，Stallman 成立了自由软件基金会（FSF），这是一个旨在支持 GNU 项目以及其它自由软件开发的非赢利组织。

GNU 项目的一个重要成果就是 GNU General Public License(GPL)的产生，这也是 Stallman 对自由软件精神的具体化。Linux 发行版的多数软件，包括内核都按 GPL（或者类似的许可）授权。GPL 授权的软件必须使源代码自由可用，而且允许按 GPL 许可自由地重新发布。GPL 授权的软件允许自由地修改，但是修改后的软件必须同样遵循 GPL 许可。如果修改后的软件以可执行方式发布，作者必须同时允许以不超过发布的代价获得修改过的源代码。GPL 第一版发布于 1989 年，目前的版本 3 发布于 2007 年。版本 2 发布于 1991 年，目前使用最广泛，也是 Linux 内核采用的授权。

GNU 项目最初并没有开发出一个可用的 UNIX 内核，但确实创建了许多其它程序。由于这些程序设计成在类 UNIX 操作系统中运行，它们可以也确实被用在现有的 UNIX 实现中，有些还迁移到其它操作系统。GNU 项目最著名的程序有 Emacs 文本编辑器、GCC（最早是 GNU C 编译器，不过现在重新命名为 GNU 编译器集合，包含 C、C++和其它语言的编译器）、Bash shell、和 glibc（GNU C 库）。

在 1990 年代初期，GNU 项目已经拥有了一个几乎完整的系统，除了一个关键的组成：可用的 UNIX 内核。GNU 项目开始规划一个野心勃勃的内核设计，被称为 GNU/HURD，基于 Mach 微内核。不过 HURD 远远达不到可发布的程度。（在本书写作之时，HURD 的工作仍在继续，目前只能运行在 x86-32 体系架构下）。

万事俱备，只欠东风。GNU 项目已经创建了完整 UNIX 系统所需的一切，只差一个最重要的内核了。

1.2.2 Linux 内核

Linus Torvalds 在 1991 年还是芬兰郝尔辛基大学的一名学生，当时他想为自己的 Intel 80386 PC 编写一个操作系统。在 Linus 的课程学习过程中，他接触了 Minix，由 Andrew Tanenbaum 在 1985 年左右开发的类 UNIX 操作系统内核，后者是荷兰某大学的教授。Tanenbaum 创造了 Minix，并提供完整的源代码，用作大学操作系统设计课程的教学工具使用。Minix 内核可以在 386 系统中构建和运行，但是由于主要目的是教学工具，Minix 设计成很大程度上独立于硬件体系架构，因此不能完全发挥 386 处理器的能力。

于是 Torvalds 启动了自己的项目，开始为 386 创建一个高效、全功能的 UNIX 内核。几个月之后，Torvalds 开发了一个基本的内核，允许自己编译和运行许多 GNU 程序。然后在 1991 年 10 月 5 日，Trovalds 开始在网上请求其它程序员的帮助，发出了下面这段被广泛引用的声明，他在 comp.os.minix Usenet 新闻组上发布了自己内核的 0.02 版：

你是否怀念 minix-1.1 版时的日子？那时人们干劲十足，自己编写设备驱动程序。你是否手头正缺少一个很好的项目，并且非常渴望为符合自己的需要动手修改一个操作系统？当几乎所有的程序都能在 Minix 上运行时，你是否感到非常失望？不再有为了调通一个巧妙的程序而整夜不睡觉的夜猫子？那么本消息（邮件、公告）可能正是为你而发布的:-)。

正如我一个月前所提到的，我正在开发一个用于 AT-386 微机类似于 Minix 的操作系统。它目前已经达到了可用的程度(当然，能不能用还依赖于你的具体要求)，而且我很高兴把源代码拿出来广泛发布。目前它的版本是 0.02(加上已经编制好的(很小的)补丁程序，就是 0.03)，但是我已经在它上面成功地运行了 bash/gcc/gnu-make/gnu-sed/压缩程序等。

该小巧项目的源程序可以在 nic.funet.fi(128.214.6.100)上/pub/OS/Linux 目录中找到。该目录中含有一些 README 文件以及几个在 Linux 下运行的二进制执行程序(bash, update 和 gcc，你还能要求什么呢:-)。提供了完整的内核源代码，而且没有使用 minix 的代码。库文件的源代码仅是部分免费的，所以目前不能给出。照内核现在的样子，系统已经可以进行编译，并且已经可以运行。二进制执行程序（bash 和 gcc）的源代码可以在同一个地方的/pub/gnu 目录中找到。

当心！警告！注意！这些源代码仍然需要 minix-386 系统来进行编译（需要 gcc-1.40，1.37.1 可能也能用，但没有试过），并且如果你想运行它的话还需要 minix 来进行设置，所以对没有 minix 的人来说，它至今它还不是一个独立的系统，不过我正在朝这方面努力着。你还需要有些骇客的本事来设置它，所以对那些希望一个 minix-386 取代品的人来说，就不用考虑 Linux 了。它目前主要是供对操作系统感兴趣的骇客使用的，并且有能使用 minix 的 386 机器。该系统需要一个 AT 兼容硬盘（IDE 硬盘当然更好）以及 EGA/VGA 显示卡，如果你还感兴趣的话，就使用 ftp 下载 README/RELNOTES 文件看看，并且/或者给我 EMAIL 告之其它信息。

我能够（当然，几乎是）听到你问自己“为什么？”，Hurd 将在近年（或者两年、或者下个月，谁知道）内推出，而且我已经有了 minix。这是一个骇客为骇客们写的程序，在开发过程中我已经得到了快乐，而某些人可能也乐意阅读它，甚至为自己的需要而修改它。它仍然很小，足以理解、使用和修改，我正期望你可能有的任何建议和说明。 我也对为 minix 系统编写过工具软件/库函数的任何人的反馈信息感兴趣。如果你的软件是可以自由发布的(在版权下甚至公共域内)，那么我很希望得到你们的消息，这样我就可以将它们加入到 Linux 系统中。现在我正使用着 Earl Chews 的 stdio（Earl，谢谢你的很好而又能使用的系统），很欢迎这种类似的软件。你的版权当然会保留着，如果你乐意我使用你的代码，就请告知。

Linus

按照传统 UNIX 克隆采用的 X 字母结尾命名惯例，这个内核最终命名为 Linux。最初 Linux 采用更加受限制的授权，不过 Torvalds 很快就将 Linux 许可更换为 GNU GPL 协议。

Linus 的请求帮助得到热烈影响。很多程序员加入 Linux 的开发，添加了许多

特性，例如增强的文件系统、网络支持、设备驱动、和多处理器支持等。到 1994 年 3 月，开发者们发布了 1.0 版本，1995 年 3 月发布了 Linux 1.2，1996 年 6 月发布了 Linux 2.0，1999 年 1 月发布了 Linux 2.2，2001 年 1 月发布了 Linux 2.4。2001 年 11 月开始内核 2.5 的开发，到 2003 年 12 月发布了 Linux 2.6。

BSD

值得一提的是 1990 年代前期，另一个免费的 UNIX 也已经能够用于 x86-32 体系架构。Bill 和 Lynne Jolitz 对一个已经很成熟的 BSD 系统向 x86-32 做了迁移，名叫 386/BSD。迁移基于 BSD Net/2 发布版（1991 年 6 月），是 4.3BSD 的一个版本，把所有 AT&T 私有的源代码都替换或移除掉。Jolitz 夫妇把 Net/2 迁移到 x86-32，并重写了缺失的代码，在 1992 年 2 月发布了 386/BSD 的第一个版本（V0.0）。

在经历了最初短暂的成功和流行之后，386/BSD 的工作由于各种原因而停滞。随着大量 patch 逐渐积压得不到处理，两个开发团队应运而生，分别创建了自己基于 386/BSD 的发布版：NetBSD，强调在各种硬件之间保持可移植性；FreeBSD，强调性能，也是现代 BSD 中最流行的一个。NetBSD 的第一个发布版是 1993 年 4 月的 0.8；FreeBSD 的首张 CD-ROM（版本 1.0）发布于 1993 年 12 月。另外还有一个 OpenBSD，派生自 NetBSD 项目，在 1996 年发布了最初的 2.0 版本，OpenBSD 特别强调安全性。到 2003 年中期，一个新的 DragonFly BSD 又从 FreeBSD 4.x 分离而出。DragonFly BSD 采用了不同于 FreeBSD 5.x 的方式，特别为对称多处理器（SMP）体系架构设计。

如果不提到 UNIX 系统实验室（USL，负责开发和销售 UNIX 的 AT&T 子公司）和伯克利之间的诉讼，那我们对于 BSD 的讨论就不是完整的。在 1992 年初，合并成立了伯克利软件设计公司（BSDi，今天是 Wind River 的一部分），开始发布一个商业支持的 BSD UNIX：BSD/OS，基于 Net/2 发行版和 Jolitz 夫妇的 386/BSD 增强功能。BSDi 以 995 美元发布二进制和源代码，并且建议潜在客户使用他们的电话号码 1-800-ITS-UNIX。

1992 年 4 月，USL 向 BSDi 正式提出诉讼，试图阻止 BSDi 销售包含 USL 私有源代码和商业秘密的产品。USL 同时还要求 BSDi 停止使用迷惑性的电话号码。

这个官司最终扩大为要求加州大学赔偿。法院最后判决同意了 USL 的两个主张，并驳回了其它请求。接着马上加州大学向 USL 提出反诉讼，声称 USL 未经许可在 System V 中使用了 BSD 代码。

官司正在悬而未决的时候，Novell 收购了 USL，其 CEO（Ray Noorda）开始公开声明自己希望双方在市场上而不是法院里竞争。诉讼最终得以在 1994 年 1 月终结，加州大学必须移除 Net/2 发布版 18000 个文件中的 3 个，并对其它少数文件做一些很小的修改，另外还要对大约 70 个文件增加 USL 版本声明，而且这些文件不能够再次发布。这个修改后的系统在 1994 年 6 月发布为 4.4BSD-Lite（加州大学发布的最后一个版本是 1995 年 6 月的 4.4BSD-Lite 版本 2）。从这时开始，法律条款要求 BSDi、FreeBSD、NetBSD 用修改后的 4.4BSD-Lite 源代码替换 Net/2。尽管这导致 BSD 派生开发的一定延迟，但也使这些系统通过三年的开发，从加州大学计算机系统研究组织发布 Net/2 后重新同步到一起。

Linux 内核版本号

和多数自由软件项目一样，Linux 采用尽早发布、经常发布的模型，因此新的内核修订频繁更新（有时候几乎每天）。随着 Linux 用户群的增长，对发布模型进行了一定的修改，以减少对现有用户的影响。具体来说，从 Linux 1.0 发布之后，内核开发者就采用了固定的内核版本命名规范，每个发布版本统一命名为 x.y.z：其中 x 表示主版本号；y 表示在该主版本号下的副版本号；而 z 则是副版本号下的修订版本号（通常是很小的改进和 bug 修复）。

在这样一种模型下，通常会有两个内核版本总是处在开发过程中：一个是稳定版，用于生产系统，其主版本号为偶数；另一个是开发版，相对来说不稳定一些，主版本号一般是下一个奇数。理论上（实践中并不总是）所有新特性都只应该添加在当前开发版内核中，而稳定版的修订系列严格限制为很小的改进和 bug 修复。当内核开发者认为开发版本适合发布时，就会成为新的稳定版，并赋予一个偶数版本号。例如 2.3.z 开发内核最终形成了 2.4 稳定内核版本。

2.6 内核发布之后，开发模型发生了变化，主要目的是解决稳定版内核发布时间间隔太长导致的问题和挫折（Linux 2.4.0 和 2.6.0 之间差不多有三年时间）。

关于改善开发模型的谈论时不时都有进行，但是核心细节基本保持如下：

- 不再有稳定和开发版的明确区分。每个新的 2.6.z 发布都可以包含新特性，而且都经历增加新特性，然后通过几个候选发布版达到稳定的生命周期。当候选版本足够稳定时，就发布为内核 2.6.z 版本。发布周期大约三个月。
- 有时候稳定的 2.6.z 发布版需要小的 patch 来修复 bug 或安全性问题。如果这些修复有足够高的优先级，而且这些 patch 也足够简单到不可能出错，那么不需要等待下一个 2.6.z 发布版，可以直接创建一个 2.6.z.r 发布版，这里的 r 序列号表示 2.6.z 内核的副修订版本。
- 额外的责任被转移到发行版厂商，来确保发行版内核的稳定性。

后面章节有时候遇到特殊的 API 时，会提及具体的内核版本（例如新的或修改的系统调用）。不过在 2.6.z 系列内核之前，多数内核变更都发生在奇数开发版中，我们通常会注明这个变化是在下一个稳定版中产生的，因为多数应用开发者都是使用稳定版内核而不是开发版内核。许多情况下，手册页则会精确地标注某个特性是在哪个开发版出现或修改的。

对于 2.6.z 系列内核出现的变化，我们会标注具体的内核版本号。当我们说某个特性是内核 2.6 的新特性时，如果不带 z 修订号，就表示这个特性是在 2.5 开发内核中实现的，首次出现在稳定内核版本 2.6.0。

移植到其它硬件体系架构

在 Linux 最初的开发阶段，高效地实现 Intel 80386 是主要目标，而不是与其它处理器体系架构的可移植性。但是随着 Linux 越来越流行，开始向其它处理器体系架构进行移植，最开始是 Digital Alpha 芯片。Linux 能够支持的硬件体系架构非常多，而且还在不断增长。包括但不限于：x86-64、Motorola/IBM PowerPC 和 PowerPC64、Sun SPARC 和 SPARC64（UltraSPARC）、MIPS、ARM（Acorn）、IBM z 系列（以前的 System/390）、Intel IA-64（Itanium）、Hitachi SuperH、HP PA-RISC、和 Motorola 68000。

Linux 发行版

准确地说，Linux 这个术语只是指 Linus Torvalds 和其它开发者开发的内核。但是通常我们说的 Linux 则包括内核，加上大量其它软件（工具和库），它们一起组成了完整的操作系统。在 Linux 最早期的时代，用户需要自己组合所有这些软件，创建文件系统，正确地存放和配置文件系统中的所有软件。这需要大量时间和专业知识。结果就是 Linux 发行版市场的兴起，发行版自动化处理大多数安装过程，创建文件系统并安装内核和其它必需的软件。

最早的发行版出现于 1992 年，包含了 MCC Interim Linux（Manchester Computing Centre，UK）、TAMU（Texas A&M 大学）、和 SLS（SoftLanding Linux 系统）。现存最老的商业发行版是 1993 年出现的 Slackware；非商业的 Debian 发行版大约也在那时候出现，随后是 SUSE 和红帽。当前非常流行的 Ubuntu 发行版于 2004 年发布。今天许多发行版公司都雇佣了大量程序员，继续为自由软件项目做出贡献，或者发起新的项目。

1.3 标准化

1980 年代后期，众多的 UNIX 实现也带来一个问题。某些 UNIX 实现基于 BSD，其它则基于 System V，某些特性则同时来自这两个变种。此外每个商业厂商都为自己的 UNIX 实现增加了额外的特性。结果就是从一个 UNIX 实现向另一个移植软件变得非常困难。这种状况为 C 编程语言和 UNIX 系统的标准化施加了积极的压力，标准化可以使应用在平台间移植就得非常简单。我们来看一看相关的标准。

1.3.1 C 编程语言

在 1980 年代早期，C 已经存在了 10 年之久，而且在多数 UNIX 系统和其它操作系统中都已经被实现。各种不同实现之间存在许多细小的差别，部分原因是 C 语言某些方面如何工作，并没有在事实上的标准中（Kernighan 和 Ritchie 在 1978 年出版的 C 编程语言一书）详细描述（书中老式的 C 语法有时候也称为传统 C 或者 K&R C）。此外，1985 年产生的 C++突出了 C 语言中缺乏的某些不影响兼容

性的改进或增强，比如函数原型、结构体赋值、类型限定符（const 和 volatile）、枚举类型、和 void 关键字。

这些因素驱动了 C 语言的标准化，最终在 1989 年通过了美国国家标准协会（ANSI）的 C 标准（X3.159-1989），随后又在 1990 年被采纳为国际标准组织（ISO）标准（ISO/IEC 9899:1990）。除了定义 C 语言的语法和语义，该标准还描述了标准 C 库，包括 stdio 函数、字符串处理函数、数学函数、各种头文件等等。这个版本的 C 被称为 C89 或 ISO C90，Kernighan 和 Ritchie 的 C 编程语言第二版（1988）对标准做了完整描述。

ISO 在 1999 年接受了 C 标准的新修订（ISO/IEC 9899:1999；参考 http://www.open-std.org/jtc1/sc22/wg14/www/standards）。这个标准通常称为 C99，对 C 语言和标准库做了一定的修改。包括增加 long long 和 bool 数据类型、C++ 风格注释（//）、受限指针、以及变量长度数组。（在本书写作的时候，还在对 C 标准进行进一步的修订，非正式地命名为 C1X。新标准有望在 2011 年获得批准）。

C 标准与操作系统实现完全无关；也就是说并没有绑定于 UNIX 系统。这表示使用纯标准库编写的 C 程序应该可以在任何计算机和操作系统之间移植。

1.3.2 第一个 POSIX 标准

POSIX 术语（可移植操作系统接口）表示了一组标准，由电子电气工程协会（IEEE）组织开发，特别是其下属的可移植应用标准委员会（PASC，http://www.pasc.org/）。PASC 标准的目标是在源代码层面上提高应用的可移植性。

POSIX 标准对于我们来说关系最紧密的是第一个 POSIX 标准，称为 POSIX.1（或者全称 POSIX 1003.1），以及随后的 POSIX.2 标准。

POSIX.1 和 POSIX.2

POSIX.1 在 1988 年成为 IEEE 标准，然后在 1990 年经过很小的修订，被采纳为 ISO 标准（ISO/IEC 9945-1:1990）。（原始的 POSIX 标准没有在线提供，但可以在 IEEE 的网站 http://www.ieee.org/ 上购买）。

POSIX.1 定义 API 提供明确的服务，而且遵循该标准的操作系统必须提供该

API。这样的操作系统才可以获得 POSIX.1 依从的证明。

POSIX.1 基于 UNIX 系统调用和 C 库函数 API，但是并没有要求特定实现一定要与这个接口绑定。这意味着这些接口可以被任何操作系统实现，不必非得是 UNIX 操作系统。实际上有些厂商已经增加了 API 到自己私有的操作系统中，获得了依从 POSIX.1 的证明，同时又大体上保持底层操作系统不变。

原始 POSIX.1 标准的很多扩展也很重要。1993 年通过的 IEEE POSIX 1003.1b（POSIX.1b，正式名称是 POSIX.4 或者 POSIX 1003.4），包含了对基础 POSIX 标准的许多实时扩展。1995 年通过的 IEEE POSIX 1003.1c（POSIX.1c），定义了 POSIX 线程。1996 年通过了 POSIX.1 标准的修订版（ISO/IEC 9945-1:1996），核心内容保持不变，但整合了实时与线程扩展。IEEE POSIX 1003.1g（POSIX.1g）定义了网络 API，包括 socket；1999 年通过的 IEEE POSIX 1003.1d（POSIX.1d）和 2000 年通过的 POSIX.1j，定义了额外的实时扩展。

另外一个相关的标准，POSIX.2（1992，ISO/IEC 9945-2:1993）标准化了 shell 和许多 UNIX 实用工具，包括 C 编译器的命令行接口。

FIPS 151-1 和 FIPS 151-2

FIPS 是联邦信息处理标准的简称，是 US 政府为采购计算机系统而制定的一组标准。1989 年公布了 FIPS 151-1。这个标准基于 1988 年的 IEEE POSIX.1 标准和 ANSI C 标准草案。FIPS 151-1 和 POSIX.1（1988）的主要区别是 FIPS 标准强制要求某些 POSIX.1 指定可选的特性。因为 US 政府是主要的计算机系统采购商，多数计算机厂商都确保自己的 UNIX 系统遵循 FIPS 151-1 版本的 POSIX.1 标准。

FIPS 151-2 对应于 1990 年 POSIX.1 的 ISO 版本，其它则保持不变。现在已经过时的 FIPS 151-2 在 2000 年 2 月取消标准。

1.3.3 X/Open 公司和开放组织

X/Open 公司是国际计算机厂商组成的集团，采纳或改编现有标准来产生综合的开放系统标准。它创建了 X/Open 可移植指南，基于 POSIX 标准的一系列可移植指南。这个指南的首个重要发布版是 1989 年的 Issue 3（XPG3），随后 1992

年发布了 XPG4，并在 1994 年重新修订，生成了 XPG4 的版本 2，这个标准同时整合了 AT&T System V 接口定义 Issue 3 的重要部分，我们在 1.3.7 节会再加描述。这个修订版本也被称为 Spec 1170，其中 1170 指的是标准定义的接口数量（函数、头文件、和命令）。

当 Novell 在 1993 年初获得了 AT&T 的 UNIX 系统业务后（后来又自己丢失了这块业务），把 UNIX 商标的权利转移给了 X/Open（转移的计划发布于 1993 年，但法律要求延迟到 1994 初才完成）。XPG4 版本 2 也因此重新包装为 Single UNIX Specification（SUS 或 SUSv1），有时候也叫 UNIX 95。包括 XPG4 版本 2、X/Open Curses Issue 4 版本 2 规范、和 X/Open 网络服务（XNS）Issue 4 规范。Single UNIX 规范的版本 2（SUSv2，http://www.unix.org/version2/online.html）发布于 1997 年，实现并通过验证这个规范就可以称为 UNIX 98。（这个标准有时候也被称为 XPG5）。

到 1996 年，X/Open 与开放软件基金会合并组成了开放组织。几乎所有与 UNIX 系统有关联的公司或组织现在都是开放组织的成员，继续开发 API 标准。

1.3.4 SUSv3 和 POSIX.1-2001

从 1999 年开始，IEEE、开放组织、和 ISO/IEC Joint 技术委员会就 Austin 公共标准修订组织（CSRG，http://www.opengroup.org/austin/）进行合作，目标是修订和巩固 POSIX 标准和 Single UNIX 规范。（Austin 组织由于 1998 年 9 月在德克萨斯洲的奥斯丁举行开幕式而得名）。结果在 2001 年 12 月批准了 POSIX 1003.1-2001，有时候直接称为 POSIX.1-2001（随后被采纳为 ISO 标准 ISO/IEC 9945:2002）。

POSIX 1003.1-2001 替代了 SUSv2、POSIX.1、POSIX.2、和其它早期 POSIX 标准草案。这个标准也被称为 Single UNIX 规范版本 3，本书后面通常使用 SUSv3 来引用它。

SUSv3 基本规范大概有 3700 页，分成以下四个部分：

- 基本定义（XBD）：这部分包含定义、术语、概念、和头文件内容规范。一共提供了 84 个头文件规范。
- 系统接口（XSH）：这部分的开头描述了许多有用的背景信息。中间大部分内容包含许多函数的规范（实现为系统调用或库函数）。这部分总共包

含了 1123 个系统接口）。

- Shell 和实用工具（XCU）：这部分规范了 shell 的操作和许多 UNIX 命令。总共规定了 160 个实用工具。
- Rationale（XRAT）：这部分包含与前面几个部分相关联的文本信息和阐述。

此外 SUSv3 还包含 X/Open CURSES Issue 4 版本 2（XCURSES）规范，规定了 curses 屏幕处理 API 相关的 372 个函数和 3 个头文件。

SUSv3 总共规定了 1742 个接口。相比较 POSIX.1-1990（包含 FIPS 151-2）才规定了 199 个接口，而 POSIX.2-1992 则规定了 130 个实用工具。

SUSv3 可以在 http://www.unix.org/version3/online.html 上找到。实现并通过 SUSv3 验证的系统则称为 UNIX 03。

原始的 SUSv3 批准之后，经过了一些小的变化和改进。结果就是 Technical Corrigendum Number 1 的出现，这些改进最后在 2003 年被整合到 SUSv3 修订版，而 Technical Corrigendum Number 2 的改进则被整合到 2004 修订版。

POSIX 依从、XSI 依从、和 XSI 扩展

历史上 SUS（和 XPG）标准与相应的 POSIX 标准存在差异，并组织为 POSIX 的功能超集。除了规定额外的接口，SUS 标准还强制要求实现许多 POSIX 可选的接口和行为。

这种差异在 POSIX 1003.1-2001 中更为微妙，它同时是 IEEE 和开放组织技术标准（也是早期 POSIX 和 SUS 标准的合并）。这个文档定义了两个级别的依从：

- POSIX 依从：定义了依从实现必须提供的接口基准。允许实现提供其它可选接口。
- X/Open 系统接口（XSI）依从：要依从于 XSI，实现必须符合所有 POSIX 依从的要求，同时还必须提供许多 POSIX 可选的接口和行为。实现必须达到这个级别的依从，才能从开放组织获得 UNIX 03 商标。

XSI 依从要求的额外接口和行为合称为 XSI 扩展。它要求支持的特性包括：线

程、mmap()和 munmap()、dlopen API、资源限制、伪终端、System V IPC、syslog API、poll()、和登录会计。

在后面章节中，当我们说 SUSv3 依从时，指的是 XSI 依从。

未规定和软规定

有时候我们会谈到某个接口在 SUSv3 中“未规定”或“软规定”

对于未规定的接口，意思是虽然可能在背景注解或 rationale 文本中提到过，但在正式标准中根本没有定义。

对于软规定的接口，则指的是虽然接口包含在标准中，但其重要细节未明确规定（通常是由于委员会成员因现有实现的差异而无法达成一致）。

当使用未规定或软规定的接口时，我们很难保证能够迁移到其它 UNIX 实现。无论如何，少数情况下这种接口在不同实现间还是比较一致的，这时我们会明确地标注这一点。

遗留特性

有时候我们会提到 SUSv3 标记某个特性是遗留的。这个术语表示这个特性只是为了兼容老的应用而保留，应该避免在新应用中使用。在许多情况下，都有其它 API 提供等价的功能。

1.3.5 SUSv4 和 POSIX.1-2008

2008 年 Austin 组织完成了 POSIX.1 和 Single UNIX 规范的修订。和之前版本的标准一样，它也包含基本规范和 XSI 扩展。我们把这个修订版称为 SUSv4。

SUSv4 的变化比 SUSv3 要少很多。最重要的改变如下：

- SUSv4 为一些函数增加了新的规范。在本书中涉及的新规范函数包括：dirfd(), fdopendir(), fexecve(), futimens(), mkdtemp(), psignal(), strsignal(), utimensat()。其它一些文件相关的函数（例如 18.11 节描述的 openat()）是现有函数（如 open()）的类似物，区别是它们根据文件描述符来解释相对路径，而不是根据进程的当前工作目录来解释相对路径。

- 有些 SUSv3 规定为可选的函数在 SUSv4 中成为强制要求。例如 SUSv3 中的很多 XSI 扩展函数现在成为 SUSv4 的基本标准。这些函数包括 dlopen API（42.1 节），实时信号 API（22.8 节），POSIX 信号量 API（第 53 章），和 POSIX 定时器 API（23.6 节）。
- SUSv3 的某些函数被标记为过时。包括 asctime(), ctime(), ftw(), gettimeofday(), getitimer(), setitimer(), siginterrupt()。
- 某些 SUSv3 标记为过时的函数从 SUSv4 中移除。包括 gethostbyname(), gethostbyaddr(), vfork()。
- SUSv3 规范的一些细节在 SUSv4 中进行了修改。例如许多函数被添加到异步信号安全函数列表（表 21-1）。

在本书的后面部分，我们会在相关主题被讨论时标注 SUSv4 的变化。

1.3.6 UNIX 标准时间线

图 1-1 总结了前面章节描述的各种标准之间的关系，并按年代顺序排列了所有标准。在这个图中，实线表示标准之间直接继承；而虚线表示某个标准影响了另一个标准，并被整合到另一个标准中，或者推迟为其它标准。

网络标准的情况比较复杂，网络的标准化开始于 1980 年代末，由 POSIX 1003.12 委员会标准化 socket API、X/Open 传输接口（XTI）API（基于 System V 传输层接口的另一个网络编程 API）、以及许多相关的 API。这个标准酝酿了很多年，也就是 POSIX 1003.12 重命名为 POSIX 1003.1G 期间。最终批准于 2000 年。

在开发 POSIX 1003.1g 的同时，X/Open 也在开发自己的 X/Open 网络规范（XNS）。该规范的首个版本 XNS Issue 4 是首个 Single UNIX 规范的一部分。后面还有 XNS Issue 5，属于 SUSv2 的一部分。XNS Issue 5 和当前的 POSIX.1g（6.6）草案本质上是一样的。再后面是 XNS Issue 5.2，与 XNS Issue 5 和 POSIX.1g 草案不一样的地方是标记 XTI API 为过时的，并包含了因特网协议版本 6（IPv6），后者大约在 1990 年代中期设计。XNS Issue 5.2 组成了 SUSv3 网络部分的基础，现在已经被废弃。相同的原因，POSIX.1g 也很快被废除标准资格。

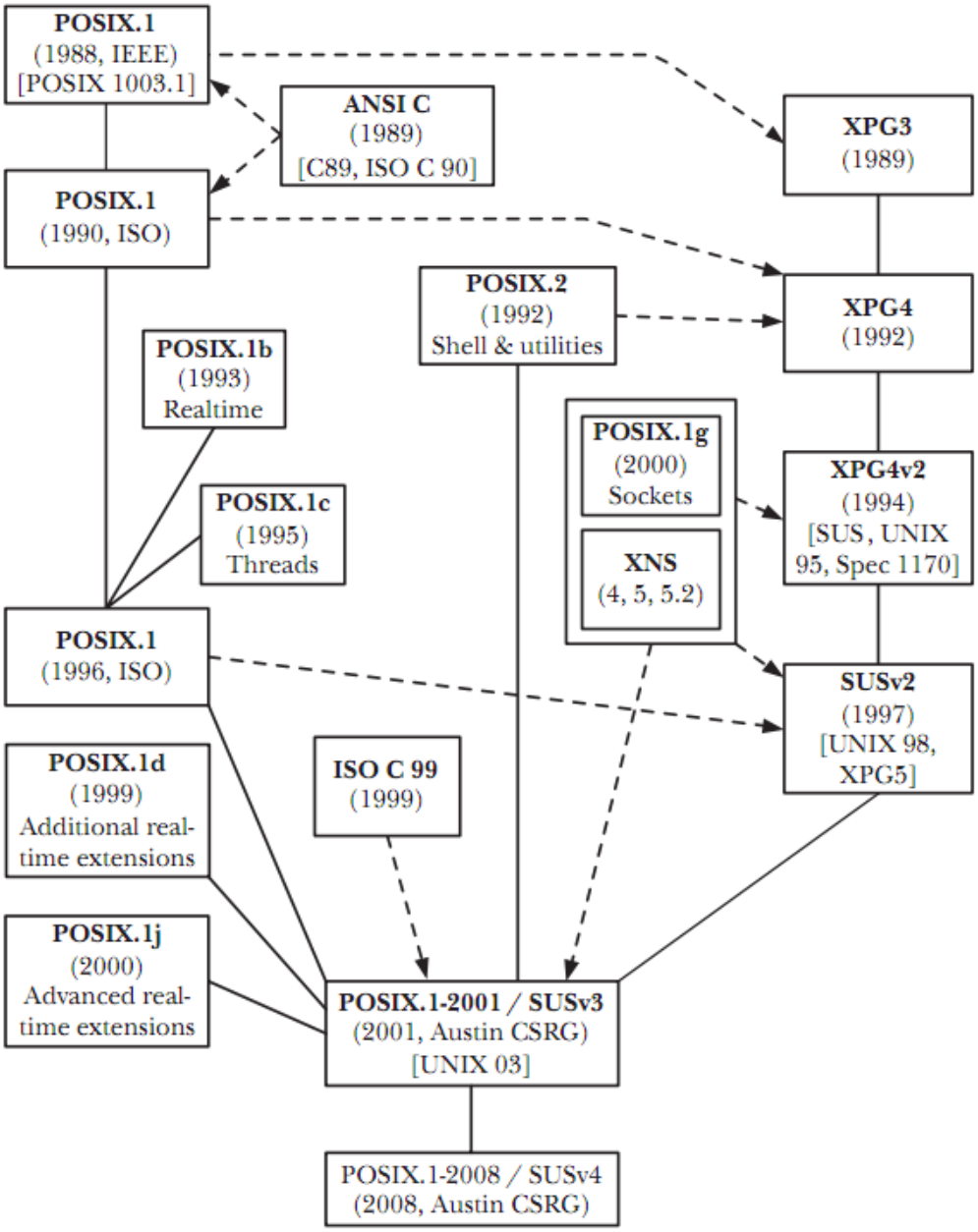

图 1-1：各种 UNIX 和 C 标准之间的关系

1.3.7 实现标准

除了上面独立或多方组织产生的标准，有时候我们会提到两个实现标准，由最终的 BSD 发布版（4.4BSD）和 AT&T System V 发布版 4（SVR4）定义。后一个实现标准由 AT&T 的 System V 接口定义（SVID）出版而正式化。1989 年 AT&T 出版了 SVID Issue 3，定义了 UNIX 实现要通过 System V 版本 4 验证，必须提供的接口。（SVID 可以在 http://www.sco.com/developers/devspecs/上在线查看）。

1.3.8 Linux、标准、和 Linux 标准基础

Linux（内核、glibc、和工具）开发致力于遵循各种 UNIX 标准，特别是 POSIX 和 Single UNIX 规范。但是在本书写作时，还没有哪个 Linux 发行版获得开放组织的“UNIX”标志。主要的问题是时间和代价。每个厂商的发行版都需要经历依从测试来获得这个认证，而且每个发行版的新版本也需要重新测试。无论如何，Linux 能够在 UNIX 市场上如此成功，得益于 Linux 对各种标准事实上的接近依从。

对于多数商业 UNIX 实现来说，相同的公司开发和发行操作系统。而 Linux 则不同，各个发行版的实现是分开的，而且由多个组织（包括商业和非商业）处理 Linux 发行版。

Linus Torvalds 并没有为某个特定的 Linux 发行版单独贡献，也没有认可某个 Linux 发行版。再加上其它个体也在负责 Linux 的开发，情况就更加复杂了。许多 Linux 内核和其它自由软件项目的开发者，都受雇于不同的 Linux 发行版公司，或者工作于对 Linux 非常感兴趣的公司（如 IBM 和 HP）。这些公司都能够影响 Linux 的发展方向，但又无法控制 Linux 的发展。当然 Linux 内核和 GNU 项目也有许多贡献者是自愿工作的。

由于存在多个 Linux 发行版，而且内核实现并不能控制发行版的内容，因此没有“标准”的商业 Linux 这回事。每个 Linux 发行商的内核通常都基于内核主版本的某个快照，并应用许多 patch 而形成。

这些 patch 一般或多或少都是由于商业需求而提供，目的是提高市场竞争力。有些情况下这些 patch 后来也被内核采纳。实际上有些新内核特性最初就是由发行公司开发，在成为内核主版本的一部分之前，已经先出现在他们的发行版中。例如 Reiserfs 日志文件系统版本 3 先是某些 Linux 发行版的一部分，然后才被 2.4 主内核采纳。

上面这些描述的要点是，不同 Linux 发行版公司提供的系统存在差异（大多数的差异都很小）。从更小的范围来讲，这种实现间的分裂和 UNIX 早期发生的分裂是一样的。Linux 标准基础（LSB）已经在努力，希望能够保证各个 Linux 发行版之间的可移植性。为了达到这个目标，LSB

（http://www.linux-foundation.org/en/LSB）开发和推广了一组 Linux 系统的标准，目的在于确保二进制应用（编译过的程序）可以在任何 LSB 依从的系统中运行。

1.4 小结

UNIX 系统最早于 1969 年在 Digital PDP-7 微计算机中由贝尔实验室的 Ken Thompson 实现。UNIX 操作系统和它的双关语名字一样，从早期的 MULTICS 系统中吸收了许多想法。1973 年 UNIX 被移植到 PDP-11 微计算机中，并用 C 重新编写，C 语言由贝尔实验室的 Dennis Ritchie 设计和实现。由于法律阻止销售 UNIX，AT&T 转而向大学发布了完整的系统。这个发布包括源代码，在大学迅速流行起来，因为它提供了便宜的操作系统，并且可以让计算机科学院和学生学习和修改其源代码。

加州大学伯克利分校在 UNIX 系统的开发中扮演了关键角色。在那里 Ken Thompson 和一些毕业生扩展了 UNIX 操作系统。1979 年伯克利发布了自己的 UNIX 系统 BSD。这个发布版在学院广泛普及，并成为几个商业实现的基础。

同时 AT&T 垄断的解体，允许公司开始销售 UNIX 系统。这就产生了另一个主要的 UNIX 变种：System V，同样也成为几个商业实现的基础。

两个不同的因素促成了 GNU/Linux 的开发，其中一个因素是 GNU 项目，由 Richard Stallman 成立。到 1980 年代末，GNU 项目已经创建了一个几乎完整的自由 UNIX 实现。缺少的只是可以工作的内核。1991 年，Linus Torvalds 受到 Andrew Tanenbaum 编写的 Minix 内核的启发，为 Intel x86-32 体系架构创建了一个可以工作的 UNIX 内核。Torvalds 邀请其它程序员加入，来改进这个内核。许多程序员积极响应，于是 Linux 被扩展和移植到大量硬件体系架构下。

不同 UNIX 和 C 实现在 1980 年代末存在的可移植性问题，直接促成了标准化进程。1989 年 C 语言标准化（C89），1999 年进一步修订标准（C99）。对操作系统接口的首个标准化尝试产生了 POSIX.1，并于 1988 年批准为 IEEE 标准，1990 年批准为 ISO 标准。在整个 1990 年代，草拟了许多标准，包括各种版本的 Single UNIX 规范。2001 年 POSIX 1003.1-2001 和 SUSv3 结合的标准得到批准。这个标准巩固和扩展了许多早期的 POSIX 标准和早期的 Single UNIX 规范。2008 年完成了

一个不那么广泛应用的标准修订，结合了 POSIX 1003.1-2008 和 SUSv4 标准。

和多数商业 UNIX 实现不同，Linux 的实现和发行是分离的。因此没有单一的“官方”Linux 发行版。每个 Linux 发行商都提供当前稳定版内核的某个快照，并增加许多 patch。LSB 开发和促进了一组 Linux 系统标准，目的是确保二进制应用在不同 Linux 发行版之间的可移植性，这样编译后的程序就可以在相同硬件的任何 LSB 依从系统中运行。

更多信息

（略）

第 2 章 基础概念

本章介绍 Linux 系统编程相关的许多概念。目标是那些主要工作于其它操作系统，或者对 Linux 和其它 UNIX 实现只有有限经验的读者。

2.1 操作系统的核心：内核

操作系统这个术语通常表示两个不同的意思：

- 表示整个软件包系统，是管理计算机资源的中心软件，包含所有标准软件工具，如命令行解释器、图形用户界面、文件工具、和编辑器。
- 狭义的含义则指管理和分配计算机资源（如 CPU、RAM、和设备）的核心软件。

内核这个术语通常则代表第二种意思，本书所说的操作系统也是这种意思。

尽管没有内核也可以在计算机中运行程序，但内核能够极大地简化编写和使用其它程序，并增强程序员的能力和灵活性。内核通过提供软件分层来管理有限的计算机资源。

内核执行的任务

内核主要执行以下任务：

- 进程调度：计算机只有一个或少数中央处理单元（CPU）来执行程序指令。和其它 UNIX 系统一样，Linux 是抢先式多任务操作系统，多任务表示多个进程（正在运行的程序）可以同时在内存中，而且每个都可以使用 CPU。抢先式表示由内核进程调度器支配哪个进程获得 CPU，以及确定进程使用 CPU 的时间。
- 内存管理：虽然计算机内存容量在近十来年变得非常庞大，但软件的体积也相应地快速增长，因此物理内存（RAM）仍然是一种有限的资源，内核必须以公平和有效的方式使多个进程间共享物理内存。和多数现代

操作系统一样，Linux 采用了虚拟内存管理机制（6.4 节），这个技术有两个主要的优点：

- 进程与其它进程以及内核隔离，因此一个进程不能读取和修改另一个进程以及内核的内存。
- 内存中只保留某个进程的部分，因此降低了每个进程的内存需求，允许更多进程同时存在于 RAM 中。这也提高了 CPU 利用率，因为增强了这样一种可能性，任何时候至少有一个进程可以让 CPU 执行。

- 文件系统管理：内核提供文件系统，允许创建、读取、更新、删除文件等等操作。
- 创建和终止进程：内核可以装载新程序到内存中，为其提供运行所需的相关资源（CPU、内存、文件访问等）。每个正在运行的程序就是一个进程。一旦某个进程完成执行，内核确保它使用的资源被释放，并可以提供给接下来的程序使用。
- 设备访问：计算机系统中附加的设备（鼠标、显示器、键盘、磁盘和磁带设备等等）允许计算机与外界进行交流，提供输入和输出功能。内核为程序提供标准化和简化的接口访问设备，同时为多个进程使用设备进行仲裁。
- 网络：内核代表用户进程传输和接收网络信号（包）。这个任务包括将网络包路由至目标系统。
- 提供系统调用应用编程接口（API）：进程可以向内核请求执行不同的任务，使用内核入口也就是系统调用。Linux 系统调用 API 是本书的主要主题。3.1 节详细描述了进程执行系统调用时的步骤。

除了上面这些特性，多用户操作系统（如 Linux）通常还给用户提供虚拟私有计算机的抽象；每个用户都可以登录到系统中，并与其它用户大体上独立操作。例如每个用户有自己的磁盘存储空间（home 目录）。此外用户还可以运行程序，每个程序都能获得共享的 CPU，并在自己的虚拟地址空间中操作，这些程序还可以独立的访问设备和通过网络传输信息。内核解决潜在的硬件资源访问冲突，因

此用户和进程通常感觉不到冲突的存在。

内核模式和用户模式

现代处理器体系架构通常允许 CPU 至少在两种不同模式下操作：用户模式和内核模式（有时候也称为超级模式）。通过硬件指令就可以在不同模式间切换。相应地虚拟内存也被划分为用户空间和内核空间等区域。当运行在用户模式中时，CPU 只能访问标记为用户空间的内存；试图访问内核空间内存会导致硬件异常。当运行在内核模式中时，CPU 可以同时访问用户和内核空间内存。

有些操作只有进程处于内核模式时才能执行。例如执行 halt 指令来停止系统、访问内存管理硬件、发起设备 I/O 操作等。通过把操作系统放在内核空间中，操作系统实现可以确保用户进程无法访问内核的指令和数据结构，或者阻止用户进程执行有害操作。

进程 VS 内核对系统的视角

在每天的许多编程工作中，我们习惯于按面向进程的方式来思考。但是考虑到本书后面讲解的许多主题，调整我们的视角，从内核的角度来观察会非常有帮助。为了使对比更加明显，我们首先考虑进程视角，然后是内核视角。

一个运行系统通常有许多进程。对于每个进程，很多事情都在异步发生。执行进程并不知道自己什么时候 CPU 时间用完，其它进程被调度获得 CPU，以及自己何时再次被调度，也不知道发生的顺序如何。信号递送和进程间通信事件由内核仲裁，对进程来说可能在任何时间发生。许多事情对进程是透明的。进程不知道自己在 RAM 中的位置，也不知道自己哪部分内存空间在内存中或是在交换区域（磁盘的保留区域，用来补充计算机的 RAM）。类似地，进程也不知道自己访问的文件被存放于磁盘驱动器的位置；进程只是简单地通过名字来引用文件。进程的操作相互独立，不能直接与其它进程通信。进程自己也不能创建新进程，甚至无法终止自己。最后进程也不能直接与计算机的输入输出设备交互。

相比之下，运行系统的内核则知道和控制了所有一切。内核为系统中所有运行进程提供协助。内核决定哪个进程获得 CPU 访问权，什么时候获得，使用多

长时间。内核维护一组进程数据结构，包含所有运行进程的所有信息，并根据进程创建、状态变化、进程终止来更新这些数据结构。内核维护所有底层的文件数据结构，允许程序使用文件名访问文件，并转换为磁盘中的物理位置。内核同时还维护每个进程虚拟内存到物理内存映射，以及到磁盘交换区域映射的数据结构。进程间的所有通信都通过内核提供的机制来完成。根据进程的请求，内核创建新进程或结束现有进程。最后内核（特别是设备驱动）执行所有与输入输出设备的交互，为用户进程传递信息。

本书后面我们讲到“进程可以创建另一个进程”、“进程可以创建管道”、“进程可以向文件写入数据”、“进程可以通过调用 exit()终止”，请记住内核仲裁所有这些动作，这些句子只不过是“进程可以请求内核创建另一个进程”的简称。

2.2 Shell

Shell 是特殊的程序，它读取用户输入的命令，并执行适当的程序来响应这些命令。Shell 有时候也被称为命令解释器。

login shell 表示用户首次登录时，为运行 shell 而创建的那个进程。

虽然在某些操作系统中命令解释器是内核的部分，但在 UNIX 系统中，shell 实际上是用户进程。存在许多不同的 shell，相同计算机的不同用户可以同时使用不同的 shell。比较重要的几个 shell 如下：

- Bourne shell（sh）：这是被广泛使用的最古老的 shell，由 Steven Bourne 编写。它是 UNIX 第 7 版的标准 shell。Bourne shell 包含许多其它所有 shell 拥有的特性：I/O 重定向、管道、文件名自动生成、变量、环境变量操作、命令替换、后台命令执行、和函数。所有后来的 UNIX 实现都包含 Bourne shell，同时也提供其它某些 shell。
- C shell（csh）：这个 shell 由加州大学伯克利分校的 Bill Joy 编写。名字的来源是这个 shell 和 C 编程语言有许多相似的流控制。C shell 提供 Bourne shell 没有的几个有用的交互特性，包括命令历史、命令行编辑、任务控制、和别名。C shell 和 Bourne shell 不保持向后兼容。尽管 BSD 的标准交

互 shell 是 C shell，shell 脚本（马上讲到）通常都是按 Bourne shell 编写，这样才能在所有 UNIX 实现中保持可移植。

- Korn shell（ksh）：这个 shell 由 AT&T 贝尔实验室的 David Korn 编写，是 Bourne shell 的继承者。与 Bourne shell 保持向后兼容的同时，增加了与 C shell 类似的交互特性。
- Bourne again shell（bash）：这个 shell 是 GNU 项目对 Bourne shell 的重新实现。提供了类似于 C shell 和 Korn shell 的交互特性。bash shell 理论上的作者是 Brian Fox 和 Chet Ramey。Bash 可能是 Linux 系统使用最广泛的 shell（Linux 中 Bourne shell 是由 bash 提供的尽可能相似的模拟）。

shell 不仅仅为交互用户设计，也可以解释 shell 脚本，后者是包含 shell 命令的文本文件。为了实现这个目的，每个 shell 都有类似于编程语言的机制：变量、循环和条件控制语句、I/O 命令、和函数。

每个 shell 都执行类似的任务，只在语法上存在区别。不管我们讲哪个特定 shell 的操作，我们通常都只说“shell”，所有 shell 都按这种方式进行操作。本书的多数例子都需要使用 bash，但是除非特别提到，读者可以假设这些例子可以在其它 Bourne shell 中同样工作。

2.3 用户和组

系统的每个用户都有唯一标识，用户可能属于某个或几个组。

用户

系统的每个用户都有唯一的逻辑名（用户名）和相应的用户 ID（UID 数字）。对于每个用户，系统的 password 文件（/etc/passwd）都有一行对其进行定义，还包含以下额外信息：

- 组 ID：数字的组 ID，用户加入的第一个组。
- home 目录：用户登录后的初始目录。

- 登录 shell：用来解释用户命令的 shell 名称。

这个密码记录可能还包含用户的密码，以加密形式存储。但是由于安全原因，通常密码会存放在单独的 shadow 密码文件中，只对超级用户可读。

组

从管理的角度来讲（特别是控制文件和其它系统资源的访问），把用户组织为组是非常有用的。例如工作于同一个项目的团队成员，需要共享相同的一组文件，就可以把所有成员添加到同一个组。在早期 UNIX 实现中，用户只能加入一个组。BSD 允许用户同时加入多个组，这个想法被其它 UNIX 实现和 POSIX.1-1990 标准接受。每个组由系统组文件（/etc/group）一个单独的行定义，主要包括以下信息：

- 组名：组的唯一名称。
- 组 ID（GID）：与该组相关联的 ID 数值。
- 用户列表：逗号分隔的用户登录名列表，这些用户都属于这个组（没有在这里标识的用户也可以在自己的密码文件记录中添加该组）。

超级用户

超级用户拥有系统的特别权限。超级用户的用户 ID 是 0，通常登录名是 root。在典型的 UNIX 系统中，超级用户可以绕过系统的所有权限检查。例如超级用户可以访问系统的任何文件，无论文件的权限如何设置；也可以向系统中的任何用户进程发送信号。系统管理员使用超级用户执行许多管理性的任务。

2.4 单一目录层次、目录、链接、和文件

内核维护一个单一层次的目录结构，来组织系统中的所有文件。（这和 Microsoft Windows 明显不同，后者的每个磁盘分区都有自己的目录层次）。层次的最底部是 root 目录，名为“/”（斜线）。所有文件和目录都是 root 目录直接或

间接的子目录。图 2-1 显示了这种文件结构的一个例子：

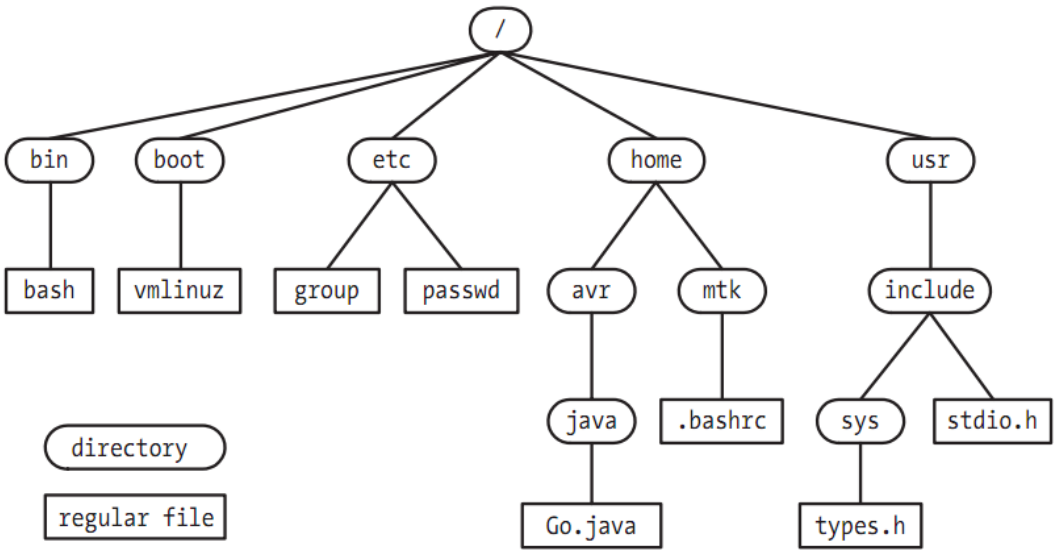

图 2-1：Linux 单一目录层次的子集

文件类型

目录是一种特殊的文件，它的内容是文件名加上相应文件索引的表格。这种文件名+引用的关联就称为链接，而文件可以有多个链接，因此在相同或不同的目录下，可以有文件的多个名字。

目录可以同时包含文件和其它目录的链接。目录之间的链接组成了图 2-1 所示的目录层次。

每个目录都至少包含两项：“.”（点），链接到目录本身；“..”（点点），链接到父目录，也就是层次中上面那个目录。每个目录（除了 root）都有父目录。对于 root 来说，“..”链接到 root 目录本身（因此“/..”等同于“/”）。

符号链接

和普通链接一样，符号链接也提供名字到文件的映射。但是普通链接是在目录列表中的文件名-指针项，而符号链接则是特殊的文件，它的内容是另一个文件的名字。（换句话说，符号链接有文件名-指针项，指针引用的文件内容是另一个文件的名字）。后一个文件通常称为符号链接的目标，通常也称符号链接“指向”或“引用”目标文件。当在系统调用中指定路径时，多数情况下内核会自动

“解引用”（跟随）路径中的每个符号链接，使用实际的文件名替换该符号链接指针。如果符号链接的目标本身也是一个符号链接，那么这个过程可能产生递归。（内核强制解引用的数量限制，以避免环形符号链接）。如果符号链接引用的文件不存在，就称为 dangling 链接。

通常把普通链接和符号链接分别称为硬链接和软链接。为什么要使用两种类型的链接？我们在后面第 18 章会做出解释。

文件名

在多数 Linux 文件系统中，文件名最多可以有 255 个字符长度。文件名可以包含任何字符，除了斜线（/）和 null 字符（\0）。但是只使用字母和数字，以及“.”（点）、“_”（下划线）、“-”（连字符）是明智的。这 65 个字符集[-._a-zA-Z0-9]在 SUSv3 中被称为可移植文件名字符集。

我们应该避免使用不可移植的文件名字符，因为这些字符在 shell、正则表达式、或其它上下文中可能有特殊含义。如果一个文件名包含特殊含义的字符，那么这些字符就必须被转义。通常是在前面加上反斜线（\）来表示这些字符不要按特殊含义来解析。在无法使用转义机制的情况下，这个文件名就是不可用的。

我们应该避免以连字符（-）来开始一个文件名，因为这样的文件名可能会被 shell 错误地解析为命令行参数。

路径名

路径名是以可选的“/”开始，包含一系列以“/”分隔的文件名的字符串。除掉最后那个文件名，这串字符就标识了一个目录（或者一个指向目录的符号链接）。路径的最后那个文件名可以是任何文件，也可以是目录。在最后一个“/”之前的所有部分有时候称为路径的目录部分，紧跟最后那个“/”的名字就称为文件，或路径的 base 部分。

路径名以从左向右的顺序读取；每个文件名都存在于路径名之前那部分所标识的目录。字符“..”可以用在路径名的任何位置，来引用当前位置路径的父目录。

路径名描述了一个文件在单一目录层次架构中的具体位置，可以是绝对或相对路径：

- 绝对路径：开始于“/”，指定了相对于根目录的位置。例如图 2-1 中的绝对路径：/home/mtk/.bashrc、/usr/include、和/（根目录的路径名）。
- 相对路径：指定相对于进程当前工作目录（下面会介绍）的文件位置，和绝对路径的区别在于不以“/”开始。在图 2-1 中，相对于目录 usr，文件 types.h 的相对路径就是 include/sys/types.h；相对于目录 avr，文件.bashrc 则可以使用相对路径../mtk/.bashrc 来引用。

当前工作目录

每个进程都有一个当前工作目录（有时候称为进程的工作目录或当前目录）。这是进程在单一层次目录架构中的“当前位置”，从这个目录开始解析所有的相对路径。

进程继承父进程的当前工作目录。login shell 设置自己的当前工作目录为用户密码文件项的 home 目录。可以使用“cd”命令修改 shell 的当前工作目录。

文件所有权和权限

每个文件都关联到一个用户 ID 和组 ID，定义了该文件的所有权，和文件所属于的组。文件所有权用来确定对于不同用户的访问权限。

要访问一个文件，系统把用户划分为三种类型：文件所有者（user）、文件组 ID 相匹配的用户（group）、和其它所有用户（other）。每种类型的用户都有三个权限位可以设置（总共 9 个权限位）：读权限允许读取文件内容；写权限允许修改文件内容；执行权限则允许执行该文件，这个文件要么是程序，要么是某种解释器可以处理的脚本（通常但不一定总是 shell）。

这些权限也可以为目录设置，不过含义稍微有些不同：读权限允许列出目录的内容（也就是文件名）；写权限允许修改目录的内容（添加、移除、和修改文件名）；执行权限（有时候称为查找权限）允许访问目录中的文件（还要取决于文件本身的权限设置）。

2.5 文件 I/O 模型

UNIX 的 I/O 模型的一个显著特点就是通用 I/O 概念。这意味着相同的一组系统调用（open(), read(), write(), close()等等）可以执行所有文件类型的 I/O 操作，包括设备（内核把应用 I/O 请求转化为适当的文件系统或设备驱动操作，来执行目标文件或设备的 I/O 操作）。因此采用这些系统调用的程序可以工作于任何文件类型。

内核本质上只提供一种文件类型，顺序字节流，如果是磁盘文件（磁盘或磁带设备），则可以通过 lseek()系统调用进行随机访问。

许多应用和库把换行字符（ASCII 码 10，有时候也称为 linefeed）解释为一行文本的终结符并开始下一行。UNIX 系统没有文件结束字符（end-of-file）；读取文件无返回数据时表示到达文件末尾。

文件描述符

I/O 系统调用通过文件描述符来引用已打开的文件，通常是一个很小的非负整数。文件描述符一般通过调用 open()获得，传入路径参数指定要对哪个文件执行 I/O 操作。

通常进程由 shell 启动时会继承三个已经打开的文件描述符：描述符 0 是标准输入，进程把它那里获得输入；描述符 1 是标准输出，进程向它写入输出数据；描述符 2 是标准错误，进程向它写入错误信息，并通知异常或错误情况。在交互式 shell 或程序中，这三个描述符通常都连接到终端。在 stdio 库中，这三个描述符对应于文件流 stdin, stdout, stderr。

stdio 库

C 程序通常采用标准 C 库中的 I/O 函数执行文件 I/O。这一组函数称为 stdio 库，包括 fopen(), fclose(), scanf(), printf(), fgets(), fputs()等等。stdio 函数在 I/O 系统调用（open(), close(), read(), write()等等）之上。

2.6 程序

程序通常有两种存在形式。第一个是源代码，使用编程语言编写（如 C），是人类可读的一系列程序语句。程序要被执行，源代码必须转化为第二种形式：二进制机器语言指令，这样计算机才能理解。（这和脚本形成对比，后者是包含许多命令的文本文件，直接由 shell 或其它命令解释器等程序处理）。程序的这两种含义通常认为是同义的，因为编译和链接最终会将源代码转化为语义相同的二进制机器代码。

过滤器

过滤器通常指的是那些从 stdin 读取输入，执行一些转化后，再将结果写入 stdout 的程序。例如 cat, grep, tr, sort, wc, sed 和 awk。

命令行参数

在 C 语言中，程序可以访问命令行参数，即程序运行时提供的命令行。要访问命令行参数，程序的 main()函数必须如下定义：

```
int main(int argc, char *argv[])
```

argc 变量包含命令行参数的总数，单个的参数由 argv 数组的字符串指针引用。其中第一个字符串 argv[0]，标识了程序本身的名字。

2.7 进程

最简单地说，进程就是执行中的程序。当程序被执行时，内核装载程序代码到虚拟内存中，为程序变量分配空间，并设置内核数据结构来记录该进程的许多信息（例如进程 ID、终止状态、用户 ID、和组 ID 等）。

从内核的视角来看，进程是内核必须为其共享许多计算机资源的实体。由于资源是有限的（如内存），内核一开始只分配一定的资源给进程，然后在进程的生命周期过程中，根据进程的需要和整个系统的负载情况，来调整这些分配。当

进程终止时，进程使用的所有资源都会被回收，并提供给其它进程重新使用。其它一些资源（如 CPU 和网络带宽），还必须在所有进程中公平地共享。

进程内存布局

进程逻辑上划分为以下部分，称为段（segment）：

- 文本（Text）：程序的指令。
- 数据（Data：程序使用的静态变量。
- 堆（Heap）：程序可以动态分配额外内存的一个区域。
- 堆栈（Stack）：随着函数调用和返回自动扩展和缩小的一小段内存，为本地变量和函数调用链接信息分配存储空间。

进程创建和程序执行

进程可以使用 fork()系统调用创建新的进程。调用 fork()的进程称为父进程，新创建的进程就是子进程。内核通过复制父进程来创建子进程。子进程获得父进程的数据、堆栈、和堆的拷贝，并且随后可以进行修改，而不影响父进程。（程序的文本，存放于只读内存区域，由父子进程共享）。

调用 fork()后，子进程要么执行父进程代码中的另一组函数；或者更常见的是使用 execve()系统调用装载和执行一个全新的程序。execve()系统调用销毁现有的文本、数据、堆栈、和堆段，并根据新程序代码的新段进行替换。

有几个 C 库函数基于 execve()系统调用实现，每个都提供稍微不同的接口，但是功能是一样的。所有这些函数都以相同的 exec 字符串开头，区别在哪里目前并不重要，我们使用 exec()来引用所有这些函数。不过要明确一点，Linux 中并没有名为 exec()的函数。

通常我们使用动词 exec 来描述 execve()和相关库函数执行的操作。

进程 ID 和父进程 ID

每个进程都有唯一的整数类型的进程标识符（PID）。每个进程同时还有一个父进程标识符（PPID），标识了创建自己的那个进程。

进程终止和终止状态

进程可以按两种方式终止：使用_exit()系统调用（或者相关的 exit()库函数）自己请求终止；或者被信号 kill 而终止。前一种情况进程会产生一个终止状态，一个很小的非负整数，父进程可以使用 wait()系统调用来检查这个值。如果调用_exit()，进程可以显式地指定自己的终止状态。如果进程被信号杀掉，终止状态根据引起进程终止的信号类型来决定。(有时候我们把传递给_exit()的参数称为进程的退出状态，以区别于终止状态，后者要么是传递给_exit()的值，要么是信号 kill 进程产生的值)。

习惯上终止状态 0 表示进程成功退出。非 0 状态表示发生了某种错误。多数 shell 都可以通过$?变量来获得最后执行程序的终止状态。

进程用户和组标识符（凭证）

每个进程都有一组相关的用户 ID（UID）和组 ID（GID）。包括：

- 实际用户 ID 和实际组 ID：标识进程所属的用户和组。新进程继承父进程的实际用户 ID 和实际组 ID。login shell 从系统密码文件相应的域获得实际用户 ID 和实际组 ID。
- 有效用户 ID 和有效组 ID：这两个 ID（再加上下面的附加组 ID）用来确定进程访问受保护资源时的权限，如文件和进程间通信对象。通常进程的有效 ID 和相应的实际 ID 相同。修改有效 ID 是允许进程获得其它用户和组的权限的一种机制，马上我们就会讲到。
- 附加组 ID：这些 ID 标识进程属于的额外的组。新进程继承父进程的附加组 ID。login shell 从系统组文件中获取自己的附加组 ID。

特权进程

在 UNIX 系统中特权进程的有效用户 ID 是 0（超级用户）。这样的进程可以绕过内核实施的权限限制。相反非特权（或无特权）则是其它用户的进程。这种进程的有效用户 ID 非 0，并且受内核的权限规则控制。

特权进程创建的进程也拥有特权，例如由 root 用户启动的 login shell。另一种使进程拥有特权的方法是通过设置用户 ID 机制，允许进程使用程序文件拥有者的身份执行该进程。

能力

从内核 2.2 开始，Linux 对特权进行了划分。每种特权操作都与特定的能力相关联，只有进程拥有相应的能力，才能执行该特权操作。超级用户进程（有效用户 ID 等于 0）的所有能力都被启用。

赋予进程一组能力子集，可以使其执行某些超级用户才允许的操作，同时又防止其执行其它特权操作。

第 30 章详细讨论了能力，在本书的后面部分，当提到特定操作只能由特权进程执行时，我们通常会标识出相应的能力。能力的名字以前缀 CAP_开始，例如 CAP_KILL。

init 进程

系统启动时，内核会创建一个特殊的 init 进程，它是所有进程的父进程，通常是/sbin/init 程序文件。系统中的所有进程都是 init 或其后代创建的（通过 fork()）。init 进程的 ID 总是 1，并且以超级用户权限运行。init 进程不能被 kill（超级用户也不行），只有系统关机时它才会终止。init 的主要任务是创建和监控运行系统需要的所有进程（更多细节请参考 init(8)手册页）。

Daemon 进程

daemon 是一种特殊用途的进程，daemon 的创建和处理与其它进程相同，但是有以下区别：

- 长期运行，daemon 进程通常在系统引导时启动，一直运行到系统关机。
- 后台运行，没有控制终端，不能读取输入也无法进行输出。

daemon 的典型例子是 syslogd，为系统记录日志信息；以及 httpd，通过 HTTP 提供 web 网页服务。

环境列表

每个进程都有一个环境列表，是进程的用户空间内存中维护的一组环境变量。这个列表的每个元素都包含一个名字和相应的值。当通过 fork()创建新进程时，继承父进程的环境。因此环境提供了一种父进程向子进程传递信息的机制。当进程使用 exec()替换原有程序时，新的程序要么继承老程序的环境，要么使用 exec()调用指定的新环境参数。

环境变量在多数 shell 中都是通过 export 命令来创建（C shell 使用 setenv 命令），例子如下：

```
$ export MYVAR='Hello world'
```

C 程序可以使用一个 external 变量（char **environ）来访问环境，还有许多库函数允许进程获得和修改环境中的值。

环境变量有许多用途。例如 shell 定义和使用了大量变量，可以被 shell 执行的脚本和程序访问。包括变量 HOME（指定了用户登录目录的路径）、变量 PATH（指定了一组目录，shell 执行用户输入的命令时会在里面查找相应的程序）。

资源限制

每个进程都要消耗资源，例如打开的文件、内存、CPU 时间。进程可以使用 setrlimit()系统调用设置自己消耗各种资源的上限。每个资源限制都有两个关联的值：软限制，限制了进程可以消耗的资源数量；硬限制，是软限制可以调整的上限。非特权进程可以把软限制设为 0 到相应的硬限制，但是只能降低硬限制。

当新进程创建时，会继承父进程的资源限制设置。

shell 的资源限制可以使用 ulimit 命令进行调整（C shell 使用 limit）。这些限制值会被 shell 执行命令创建的子进程继承。

2.8 内存映射

使用 mmap()系统调用可以在调用进程的虚拟地址空间中创建新的内存映射。内存映射有以下两种类型：

- 文件映射把文件区域映射到调用进程的虚拟内存中。一旦映射完成，就可以通过相应内存区域来访问文件内容。当需要时会自动从文件装载到内存页面中。
- 匿名映射则没有相应的文件。相反所有映射的页面都初始化为 0。

一个进程映射的内存可以和另一个进程共享。可能是两个进程同时映射一个文件的相同区域，或者子进程继承父进程的映射。

当两个或多个进程共享相同的页面时，每个进程都可能看到其它进程对页面内容的修改，具体则取决于映射是私有还是共享的。当映射是私有的时，对映射内容的修改对于其它进程是不可见的，也不会修改到底层的文件。当映射是共享的时，对映射内容的修改对于其它共享该映射的进程是可见的，而且会更新底层的文件。

使用内存映射有许多目的，包括装载可执行文件来初始化进程的文本段、分配新的内存（置 0）、文件 I/O(内存映射 I/O)、和进程间通信（通过共享映射）。

2.9 静态和共享库

对象库是已编译对象代码的文件，包含一组可被应用程序调用的函数（通常是逻辑相关的一组函数）。把一组函数的代码放在一个单独的对象库中，简化了程序创建和维护的工作。现代 UNIX 系统提供两种对象库：静态库和共享库。

静态库

静态库（有时候称为 archive）是早期 UNIX 系统唯一支持的库类型。静态库本质上是结构化的已编译对象模块。要使用静态库中的函数，我们在构建程序时

使用链接命令来指定该库。链接器为应用程序引用的所有函数找到相应的静态库模块，然后从静态库中提取出所需的对象模块，并复制到最终的可执行文件中。我们称这样的程序是静态链接的。

每个静态链接的程序都从库中复制了需要的对象模块，这种方式导致了一些缺点。其中之一就是不同可执行文件中的对象代码重复浪费了磁盘空间。当使用相同静态库的多个程序一起执行时，也浪费了内存空间；每个程序都会有相同的函数拷贝在内存中。此外如果库函数需要修改，那么在重新编译该函数并添加到静态库中后，所有使用该函数的应用都必须重新与库进行链接。

共享库

共享库是为了解决静态库的问题而设计的。

如果程序链接到共享库，那么就不会复制对象模块到可执行文件中，相反链接器会在可执行文件中插入一条记录，表示运行时需要使用这个共享库。当可执行文件装载到内存时，程序调用动态链接器确保所有需要的共享库都能够找到并装载到内存中，然后执行动态链接或 resolve 到相应的函数定义。在运行时，只有一份共享库需要保存在内存中，所有运行程序都使用这份拷贝。

共享库只包含唯一的已编译函数，可以节省磁盘空间。同时可以极大地确保程序能够轻松地使用更新版本的函数。只需要重新构建共享库，现有程序在下次运行时就可以自动使用到最新的函数定义。

2.10 进程间通信和同步

Linux 系统运行着许多进程，许多是相互独立进行操作的。但某些进程则需要合作才能完成自己的任务。这些进程需要能够与其它进程进行通信，并同步各自的动作。

进程间通信的一个方法是通过读取和写入相关信息到磁盘文件中。但是对于许多应用来说，这样做太慢也不够灵活。

因此 Linux 和所有现代 UNIX 实现一样，提供一组丰富的进程间通信机制，包括以下这些：

- 信号，用来指示发生了某个事件。
- 管道（shell 用户熟知的“|”操作符）和 FIFO，用来在进程间传输数据。
- socket，用来在进程间传输数据，既可以在同一计算机中，也可以在通过网络连接的不同计算机中进行通信。
- 文件锁，允许进程锁住文件的某个区域，阻止其它进程读取和更新该区域的文件内容。
- 消息队列，用来在不同进程间交换消息（数据包）。
- 信号量，用来同步进程间的动作。
- 共享内存，允许两个或多个进程共享一块内存。当一个进程修改共享内存的内容时，所有进程都可以立即看到这个修改。

UNIX 系统的 IPC 机制数量繁多，有些功能存在重叠，部分原因是各种 UNIX 系统变种不同发展，以及各种标准的要求导致。例如 FIFO 和 UNIX 域 socket 本质上执行相同的功能，都允许相同系统的不相关进程之间交换数据。现代 UNIX 系统拥有这两种机制，因为 FIFO 来自 System V，而 socket 来自 BSD。

2.11 信号

尽管我们在上一节把信号列为 IPC 机制之一，信号通常还在许多其它情况下被使用。值得我们进一步详加讨论。

信号通常被描述为“软件中断”。信号的到来通知进程发生了某些事件或者异常条件。信号的种类非常多，每个都标识了不同的事件或异常条件。每个信号类型都由一个整数标识，并使用符号名 SIGxxxx 来定义。

信号可以由内核发送给进程，也可以是其它进程发送（需要适当的权限），甚至可以自己给自己发送信号。例如当发生以下情况时，内核会给进程发送信号：

- 用户用键盘输入中断字符（通常是 Control-C）。
- 进程的某个子进程终止。
- 进程设置的定时器（alarm 时钟）过期。
- 进程试图访问非法内存地址。

在 shell 中，kill 命令可以向进程发送信号。kill()系统调用则为程序提供相同的功能。

当进程接收到一个信号时，它可以根据不同的信号类型，采取以下动作：

- 进程忽略信号
- 进程被信号 kill
- 进程暂时挂起，稍后在收到特别的信号后再继续。

对于多数信号类型，除了接受默认的信号动作，程序可以选择忽略信号，或者创建一个信号处理器。信号处理器是由程序员定义的函数，当信号到来时会被自动调用。这个函数可以根据信号产生的条件执行适当的动作。

从信号产生到被递送至进程，这段时间称信号是“未决”的。通常未决信号会尽快在进程下次被调度时递送至进程;或者如果进程正在运行,则会立即递送。但是通过添加信号到进程的信号掩码中，也可以阻塞该信号。如果信号产生时被阻塞，就会一直保持未决状态，直到被解除阻塞（从信号掩码中移除）。

2.12 线程

在现代 UNIX 系统中，每个进程都可以有多个执行线程。你可以把线程想象成共享相同虚拟内存，以及其它许多属性的进程。每个线程都执行同一个程序代码文件，并且共享相同的数据区域和堆。但是每个线程拥有自己的堆栈，里面存放本地变量和函数调用链接信息。

线程可以通过全局对象来互相通信。线程 API 提供了条件变量和 mutex，主要是用来允许线程通信和动作同步，特别是保护共享变量的访问。线程也可以使用 2.10 节描述的 IPC 机制进行通信和同步。

使用的线程的主要优点是多个线程间共享数据非常容易（通过全局变量）;以及某些算法使用多线程实现更加自然。此外多线程应用还可以明显地利用并行处理和多核硬件的能力。

2.13 进程组和 shell 工作控制

shell 执行的每个程序都会启动一个新的进程。例如 shell 创建三个进程来执行下面这个管道命令（按文件大小排序显示当前工作目录下的文件列表）：

```
$ ls -l | sort -k5n | less
```

所有主流 shell，除了 Bourne shell，都提供 job 控制的交互特性，允许用户同时执行和操作多个命令或管道。在 job 控制的 shell 中，管道中的所有进程都置于一个新进程组或 job 中。（shell 命令行只包含一条命令时，新的进程组只包含一个进程）。该进程组中的每个进程都拥有相同的整数值进程组标识符，这个值和进程组中的进程组领导者的进程 ID 相同。

内核允许对进程组的所有成员进行许多操作，例如递送信号。job 控制 shell 使用这个特性允许用户挂起或继续管道中的所有进程，下一节我们会描述。

2.14 会话、控制终端、和控制进程

会话是进程组(job)的一个集合。会话中的所有进程拥有相同的会话标识符，会话领导者是创建会话的那个进程，会话 ID 就是它的进程 ID。

会话主要用于 job 控制 shell。job 控制 shell 创建的所有进程组都属于相同会话，shell 就是会话领导者。

会话通常会有一个关联的控制终端。当会话领导者进程第一次打开终端设备时建立控制终端。如果是交互式 shell 创建的会话，那就是用户登录时的终端。一个终端只能作为一个会话的控制终端。

会话领导者打开控制终端之后，自己也就成为这个终端的控制进程。如果终端连接断开（例如关闭了终端窗口），控制进程会收到一个 SIGHUP 信号。

在任何时候，会话中的一个进程组是前台进程组（前台 job），它可以从终端读取输入和写入输出。如果用户在控制终端中按下中断字符（通常是 Ctrl-C）或者挂起字符（通常是 Ctrl-Z），终端设备就会发送一个 kill 或挂起信号到前台进程组。会话可以有任意数量的后台进程组（后台 job），在命令后面加上“&”字符可以创建后台进程组。

job 控制 Shell 提供一组 job 相关的命令，包括列出所有 job、向 job 发送信号、把 job 在前后台之间切换。

2.15 伪终端

伪终端是连接在一起的一对虚拟设备，称为 master（主）和 slave（从）。这对设备提供 IPC 通道，允许在两个设备间双向传输数据。

伪终端的关键是 slave 设备提供了类似终端的接口，这样就可以把一个面向终端的程序连接到 slave 设备，然后使用另一个程序连接到 master 设备，来驱动这个面向终端的程序。由驱动程序写入的输出经过终端驱动正常的输入处理（例如在默认模式下，回车被映射到换行），然后作为输入传递给连接到 slave 设备的那个面向终端的程序。面向终端的程序向 slave 设备写入的所有东西都会作为输入传递给驱动程序（也需要经过正常的终端输出处理）。换句话说，驱动程序按终端的惯例为用户处理相关的功能。

伪终端可以用在各种应用中，最显著的是实现 X Window 系统登录的终端窗口，以及提供网络登录服务，例如 telnet 和 ssh。

2.16 日期和时间

进程一般会关心两种时间类型：

- 实际时间，一般从某个标准时间点开始计量（日历时间）；或者从某个固定点开始，通常是进程启动时（逝去时间或墙上时钟时间）。在 UNIX 系统中，日历时间是从 1970 年 1 月 1 日 0 点（Universal Coordinated Time，简称 UTC）开始按秒计量，再以英国格林威治经线按时区进行调整。这个时间与 UNIX 系统的诞生比较接近，被称为 Epoch。
- 进程时间，也称为 CPU 时间，是进程从启动开始总共使用的 CPU 时间。CPU 时间又进一步划分为系统 CPU 时间、内核模式代码执行时间（执行系统调用和内核代表进程执行其它服务）、以及用户模式代码执行时间的用户 CPU 时间（例如执行普通程序代码）。

time 命令可以显示管道中进程执行所花费的实际时间、系统 CPU 时间、和用户 CPU 时间。

2.17 客户端-服务器体系架构

在本书的一些地方，我们会讨论客户端-服务器应用的设计和实现：

客户端-服务器应用分为两个组件：

- 客户端，通过发送消息请求服务器执行某种服务。
- 服务器，接收客户端请求，执行适当的动作，并发送反馈信息给客户端。

有时候，客户端和服务器可能需要进行请求和返回的扩展对话。

一般客户端应用与用户交互，而服务器应用则提供某些共享资源的访问。通常会有许多客户端进程与一个或少数几个服务器进程通信。

客户端和服务器可以同时在同一台主机，也可以通过网络存在于不同机器上。客户端和服务器之间的互相通信，需要使用 2.10 节讨论的 IPC 机制。

服务器可以实现许多服务，例如：

- 提供数据库或其它共享信息资源的访问。
- 提供跨网络的远程文件访问。
- 封装某些业务逻辑。
- 提供共享硬件资源（如打印机）的访问。
- web 页面服务。

把服务封装在一个服务器中有许多好处，例如：

- 高效，由服务器管理资源并提供服务，比在每台计算机中提供相同资源要便宜而且高效。
- 可控、协同、和安全，通过把资源（特别是信息资源）控制在单一位置，服务器可以控制资源的协同访问（如两个客户端不能同时更新相同的信

息块），也可以使资源仅对选定客户端可用，提高安全性。

- 在多样环境中操作，在网络环境下，存在许多各不相同的客户端，服务器可以运行在不同的硬件和操作系统平台中。

2.18 实时

实时应用是那些必须及时响应输入的应用。最常见的输入是外部传感器或特殊的输入设备，输出则是控制某些外部硬件。常见的需要实时响应的应用有：自动化组装流水线、银行 ATM、以及飞机导航系统。

尽管许多实时应用要求快速响应输入，但实时定义的关键是应用必须确保能够在最后期限之前响应输入。

要提供实时响应，特别是要求短时间内响应，要求底层操作系统提供支持。多数操作系统都不能够原生地提供实时支持，因为实时响应的需求和多用户共享时间的需求互相冲突。虽然 UNIX 变种有提供实时特性，传统的 UNIX 系统并不是实时操作系统。Linux 的实时变种也有，而且目前内核也正在向完全原生支持实时应用的方向发展。

POSIX.1b 定义了一组 POSIX.1 扩展来支持实时应用。包括异步 I/O、共享内存、内存映射文件、内存锁、实时时钟和定时器、可选调度策略、实时信号、消息队列、和信号量等。尽管标准没有严格限定实时，多数 UNIX 实现现在都支持上面的部分或全部特性（在本书写作之时，Linux 已经支持我们讨论的所有这些 POSIX.1b 特性）。

2.19 /proc 文件系统

和某些其它 UNIX 实现一样，Linux 也提供一个/proc 文件系统，挂载在/proc 目录下，它包含许多目录和文件。

/proc 是虚拟的文件系统，它以文件系统的文件和目录的方式，提供内核数

据结构的访问接口。这样就可以轻松地查看或修改许多系统属性。另外有一些/proc/PID 形式的目录（PID 是进程 ID），允许我们查看系统每个运行进程的信息。

/proc 文件系统的内容一般是人类可读的文本形式，可以被 shell 脚本解析处理。程序可以简单地 open 和 read，也可以 write 需要的文件。多数情况下，程序必须拥有特权才能修改/proc 目录下的文件内容。

在我们讨论许多 Linux 编程接口的时候，我们会同时描述相关的/proc 文件。12.1 节提供了/proc 文件系统的更多信息。没有任何标准对/proc 文件系统进行了定义，因此我们对其的讨论是特定于 Linux 的。

2.20 小结

在这一章，我们查看了许多 Linux 系统编程相关的基础概念。理解这些概念能够为读者提供 Linux 或 UNIX 的一定经验，使读者拥有足够的背景知识来开始学习系统编程。

第 3 章 系统编程概念

第 4 章 文件 I/O：统一的 I/O 模型

第 5 章 文件 I/O：更多细节

第 6 章 进程

第 7 章 内存分配

第 8 章 用户和组

第 9 章 进程凭证

第 10 章 时间

第 11 章 系统限制和选项

第 12 章 系统和进程信息

第 13 章 文件 I/O 缓冲

第 14 章 文件系统

第 15 章 文件属性

第 16 章 扩展属性

第 17 章 访问控制列表

第 18 章 目录和链接

第 19 章 监控文件事件

第 20 章 信号：基础概念

第 21 章 信号：信号处理器

第 22 章 信号：高级特性

第 23 章 定时器和睡眠

第 24 章 进程创建

第 25 章 进程结束

第 26 章 监控子进程

第 27 章 程序执行

第 28 章 进程创建和程序执行的更多细节

第 29 章 线程：介绍

第 30 章 线程：同步

第 31 章 线程：线程安全和线程存储

第 32 章 线程：线程取消

第 33 章 线程：更多细节

第 34 章 进程组、会话和任务控制

第 35 章 进程优先级和调度

第 36 章 进程资源

第 37 章 Daemon

第 38 章 编写安全的特权程序

第 39 章 能力

第 40 章 登录会计

第 41 章 共享库基础

第 42 章 共享库高级特性

第 43 章 进程间通信简介

第 44 章 管道和 FIFO

第 45 章 System V IPC 介绍

第 46 章 System V 消息队列

第 47 章 System V 信号量

第 48 章 System V 共享内存

第 49 章 内存映射

第 50 章 虚拟内存操作

第 51 章 POSIX IPC 介绍

第 52 章 POSIX 消息队列

第 53 章 POSIX 信号量

第 54 章 POSIX 共享内存

第 55 章 文件锁

第 56 章 Sockets：介绍

第 57 章 Sockets：UNIX Domain

第 58 章 Sockets：TCP/IP 网络基础

第 59 章 Sockets：Internet Domain

第 60 章 Sockets：服务器设计

第 61 章 Sockets：高级主题

第 62 章 终端

第 63 章 可选 I/O 模型

第 64 章 伪终端

附录 A：跟踪系统调用

附录 B：解析命令行参数

附录 C：转换 NULL 指针

附录 D：内核配置

附录 E：更多信息来源

附录 F：部分习题解答

参考书目

索引