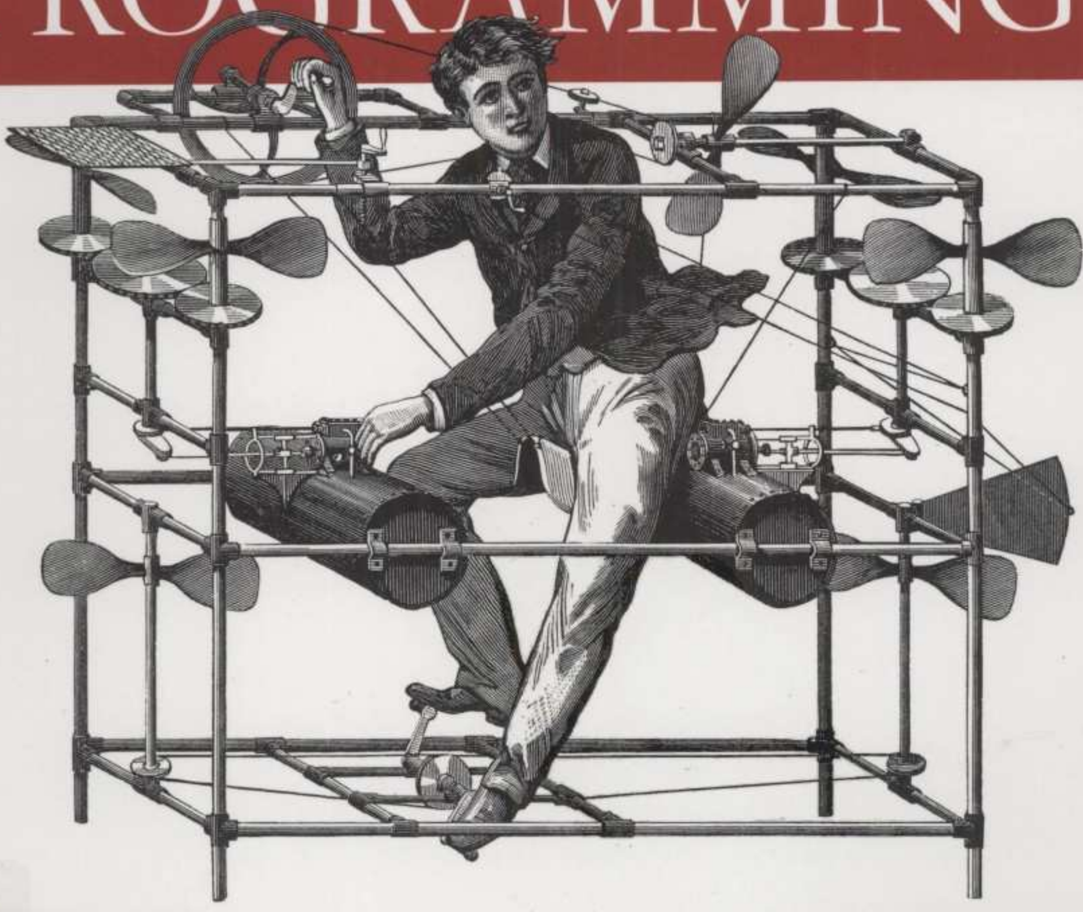
LINUX系统编程

LINUX SYSTEM PROGRAMMING



O'REILLY®

南大學 出版社

ROBERT LOVE 著

O'REILLY TAIWAN公司 编译

LINUX
系统编程

Robert Love 著
*O'Reilly Taiwan*公司 编译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo

O'Reilly Media, Inc. 授权东南大学出版社出版

南京 东南大学出版社

直接与内核及C程序库对话

LINUX系统编程

在某些时刻，几乎[illegible]所处操作系统中的系统调用和程序库打交道。本书主要讨[illegible]一代码位于底层，并且直接跟内核及核心系统程序库对话[illegible]使用标准接口包括使用Linux独有的高级接口时，在功能和性能之间如[illegible]

该书同样也是一本内行人士编写灵活高效代码的学习指南。作为内核黑客和本书的作者，Robert Love不仅阐释了系统接口应该如何工作，还介绍了它们实际上是如何工作的，以及怎样安全有效地使用它们。《Linux系统编程》包含了帮助你在任何层面编写更佳代码的实用技巧。

本书主题包括：

- 读写文件以及其他文件I/O操作，包括Linux内核如何实现和管理文件I/O，内存映射与优化技术
- 进程管理的系统调用，包括实时进程
- 文件与目录——创建、移动、复制、删除和管理
- 内存管理——内存分配接口，管理内存，以及优化内存访问
- 信号及其在Unix系统中的角色，以及基本和高级信号接口
- 时间、休眠和时钟管理，从基础开始讲述，并且涵盖POSIX时钟和高精度计时器

拥有《Linux系统编程》，你将从理论和应用的角度深入了解Linux，可以最大限度地利用系统的潜能。

Robert Love很早就成为一位Linux用户和黑客。他一直并且充满激情地活跃在Linux内核与GNOME桌面社区之中。他近来为Linux内核作出的贡献包括内核事件层方面的工作以及inotify，GNOME相关的贡献包括Beagle、GNOME卷管理器、网络管理器以及Project Utopia等。目前，Robert供职于Google开源软件办公室。

O'REILLY®
www.oreilly.com

责任编辑：张烨
封面设计：Karen Montgomery 张健

O'Reilly Media, Inc.授权东南大学出版社出版

ISBN 978-7-5641-1519-7
9 787564 115197
定价：56.00元

O'Reilly Media, Inc. 介绍

为了满足读者对网络和软件技术知识的迫切需求，世界著名计算机图书出版机构O'Reilly Media, Inc. 授权东南大学出版社，翻译出版一批该公司久负盛名的英文经典技术专著。

O'Reilly Media, Inc. 是世界上在 Unix、X、Internet 和其他开放系统图书领域具有领导地位的出版公司，同时也是联机出版的先锋。

从最畅销的《The Whole Internet User's Guide & Catalog》（被纽约公共图书馆评为20世纪最重要的50本书之一）到GNN（最早的Internet门户和商业网站），再到WebSite（第一个桌面PC的Web服务器软件），O'Reilly Media, Inc. 一直处于Internet发展的最前沿。

许多书店的反馈表明，O'Reilly Media, Inc. 是最稳定的计算机图书出版商 —— 每一本书都一版再版。与大多数计算机图书出版商相比，O'Reilly Media, Inc. 具有深厚的计算机专业背景，这使得 O'Reilly Media, Inc. 形成了一个与其他出版商迥然不同的出版方针。O'Reilly Media, Inc. 所有的编辑人员以前都是程序员，或者是顶尖级的技术专家。O'Reilly Media, Inc. 还有许多固定的作者群体 —— 他们本身是相关领域的技术专家、咨询专家，而现在编写著作，O'Reilly Media, Inc. 依靠他们及时地推出图书。因为 O'Reilly Media, Inc. 紧密地与计算机业界联系着，所以 O'Reilly Media, Inc. 知道市场上真正需要什么图书。

作者简介

Robert Love 是一位资深的 Linux 使用者和计算机迷。他积极 —— 并且热心 —— 参与 Linux 内核以及 GNOME desktop 社区的各项计划。他近来对 Linux 内核的贡献包括 kernel event layer 与 inotify 子系统。GNOME 方面的贡献包括 Beagle、GNOME Volume Manager、NetworkManager 以及 Project Utopia。目前，Robert 在 Google 的 Open Source Program Office 工作。

Robert 还是《Linux Kernel Development Second Edition》（Novell Press）的作者以及《Linux in a Nutshell Fifth Edition》（O'Reilly）的共同作者。他也是《Linux Journal》的撰稿编辑，曾撰写过许多文章，而且受邀到世界各地发表演讲。

Robert 于佛罗里达大学取得他的数学文学士以及计算机科学理学士学位。他来自南佛罗里达州，现在住在波士顿。

封面设计

本书的封面是一个在飞行器里的人。在莱特兄弟于 1903 年做出第一个比空气还重的载人器之前，世界各地的人们都想要乘坐简单但精心制作的机器来飞行。

2 或 3 世纪，据说中国的诸葛亮发明了可以在空中飞行的孔明灯（天灯），这是热气球的始祖。5 或 6 世纪左右，据称中国有许多人将自己绑在大型风筝上飞行于空中。

还有一种说法，中国所创造的旋转玩具（竹蜻蜓）是直升机的雏型，它的外观可能启发了达芬奇设计出直升机的原型，以作为人类飞行的解决方案。达芬奇还研究鸟类，并且设计出了降落伞。1845 年，他设计出了扑翼机（ornithopter），这是一个可以载人的抖动双翼的飞行器。尽管他未曾制造过它，但是扑翼机与鸟相似的结构影响了整个世纪的飞行器的设计。

封面所描绘的飞行器相较于 James Means 于 1893 所设计的滑翔机模型（没有螺旋桨）更是精心杰作。Means 之后为他的滑翔机出版了说明书，书中提到"Mt. Willard 山顶，靠近 Crawford House, N.H.，找到了一个很好的地方，可以实验这些机器。"

但这样的实验往往很危险。19 世纪晚期，Otto Lilienthal 制造出了单翼机（monoplane）、双翼飞机（biplane）以及滑翔机（glider）。他首先证明，载人飞行器可以飞行至伸手可及的距离之内，他获得了“飞行测试之父”的昵称，因为他进行了2000 多次滑翔机的飞行测试，有时可以飞行超过 1000 英尺的距离。在一次紧急着陆折断他的脊椎骨后，他于 1896 年辞世。

飞行器也被称为机械鸟（mechanical bird）以及飞艇（airship），并偶尔会获得更加丰富多彩的名字，例如人造信天翁（Artificial Albatross）。飞行器的热潮仍持续着，因为航空发烧友们仍在制造早期的飞行器。

图书在版编目（CIP）数据

Linux 系统编程 /（美）洛夫（Love, R.）著；O'Reilly Taiwan 公司编译 .—南京：东南大学出版社，2009.7

书名原文：Linux System Pogramming

ISBN 978-7-5641-1519-7

Ⅰ.L… Ⅱ.①洛… ②O… Ⅲ.Linux 操作系统－程序设计 Ⅳ.TP316.89

中国版本图书馆 CIP 数据核字（2009）第 084166 号

江苏省版权局著作权合同登记

图字：10-2008-357 号

Linux 系统编程（中文版）

出版发行：东南大学出版社

地　　址：南京四牌楼 2 号　　邮编：210096

出 版 人：江　汉

网　　址：http://press.seu.edu.cn

电子邮件：press@seu.edu.cn

印　　刷：扬中市印刷有限公司

开　　本：787 毫米 × 980 毫米　16 开本

印　　张：24.75 印张

字　　数：412 千字

版　　次：2009 年 7 月第 1 版

印　　次：2009 年 7 月第 1 次印刷

书　　号：ISBN 978-7-5641-1519-7

印　　数：1~3000 册

定　　价：56.00 元（册）

本社图书若有印装质量问题，请直接与读者服务部联系。电话（传真）：025-83792328

目录

序

每当 Linux 内核开发者不高兴的时候总是喜欢丢出一句话："用户空间只是内核的一个试验负载（test load）。"

内核开发者之所以会这么说，目的是于任何用户空间程序代码执行失败的时候撇清所有责任。当所发生的问题绝对不是内核的过错时，他们所关心的是用户空间开发者应该去修正他们自己的程序代码。

为了证明这通常不是内核的过错，3 年多前，一位具领导地位的 Linux 内核开发者曾在挤满人的会议室里，以"Why User Space Sucks"（为何用户空间程序很糟糕）为题发表演讲，他举出实例说明我们每个人每天使用了哪些可怕的用户空间程序代码。其他内核开发者则以自己创建的工具来展示差劲的用户空间程序如何滥用硬件，并耗尽无预警的笔记本电脑电池的电量。

尽管用户空间程序对嘲笑它的内核开发者而言可能只是一个"试验负载"，不过这些内核开发者也是每天都得依靠这些用户空间程序。如果没有用户空间程序可用，所有的内核充其量就只能在屏幕上交替输出 ABABAB 样式的信息。

而今，Linux 已经成为有史以来最灵活、最强大的操作系统，随处都可以看到它的踪迹，不仅最小型的手机和嵌入式装置运行它，全世界前 500 台速度最快的超级计算机中也有70%以上的在运行它。其他操作系统从未曾有过这么好的规模，也不会遭受各种硬件和环境的挑战。

如同内核一样，在 Linux 的用户空间上运行的程序也得运作在各种平台上，以人们所依赖的应用程序和公用程序提供给全世界使用。

Robert Love 自找麻烦地想要教读者关于Linux 系统上所有的系统调用，本书于焉诞生。本书将以用户空间的观点让读者全面了解 Linux 内核的运作原理，以及如何利用此系统的强大功能。

本书将告诉你如何创建可以在各种 Linux 发行版本以及硬件平台上运行的程序代码。本书也会让你了解 Linux 如何运作以及如何善用它的灵活性。

最后，也是最好的一件事，本书会教你如何让自己所编写的程序不会太差劲。

—— Greg Kroah-Hartman

前言

本书所要探讨的是系统编程 —— 特别是Linux系统编程。系统编程就是编写系统软件，也就是位于底层的程序代码，它可以直接跟内核与核心系统链接库交互。换句话说，本书的主题就是 Linux 系统调用以及其他的低端函数，例如 C 链接库所定义的。

尽管已经有许多书在探讨 Unix 的系统编程，但很少有只将焦点放在 Linux 上的，就算有，也很少有探讨最新 Linux 版本以及 Linux 独有的高级接口的。此外，本书的读者将得益于我的特殊经验：我曾为 Linux 编写过许多程序代码（包括内核以及系统软件的）。事实上，我曾经实现过本书所探讨的某些系统调用以及其他功能。因此，本书揭露了许多内幕知识，不仅会描述系统接口应该如何运作，也会说明它们实际的运作方式，以及你（编程者）如何以最有效的方式来使用它们。所以本书是 Linux 系统编程的教材，也是 Linux 系统调用的参考手册，亦是编写更聪明、更快的程序代码的权威指南。本书内容有趣且容易理解，不管你是否需要每日编写系统层的程序代码，本书都会教你让你可以编写出较理想程序代码的诀窍。

本书的读者

本书假设读者已经熟悉 C 编程以及 Linux 编程环境，不需要精通这些主题，但至少要了解它们。如果你尚未读过任何有关 C 编程语言的书籍，例如 Brian W. Kernighan 和 Dennis M. Ritchie 的经典之作《*The C Programming Language*》（Prentice Hall 出版，本书就是大家所熟悉的 K&R），我强烈建议你阅读这本书。也许你不习惯使用 Unix 文本编辑器，Emacs 和 *vim* 是最常见且备受推崇的编辑器，你可以择一使用。你还应该熟悉 *gcc*、*gdb*、*make* 等工具的基本用法。市面上可以找到许多与 Linux 编程工具及实践有关的书籍，本书结尾的参考文献列举了几本有用的参考书。

关于 Unix 或 Linux 系统编程，我已经对读者所应具备的知识作了以上假设。本书将从基础开始谈起，并蜿蜒前进到高端接口及优化诀窍。希望所有层次的读者都会觉得这是一本值得阅读以及能够学到新东西的书。在本书的写作过程中，我确实是这么想的。

我也不是没有假设本书读者的类型以及阅读本书的动机。本书适用于那些希望能够在底层写出好程序的工程师，但是那些想在自己安身之处寻找更稳固立足点的较高层的编程人员也可以在本书中找到许多对他们有用的资料。

不管你的动机是什么，祝你阅读愉快。

本书的内容

本书分成 10 章、1 个附录以及 1 个参考文献。

第一章 *介绍与基本概念*

本章可作为导读，它概述了 Linux、系统编程、内核、C 链接库以及 C 编译器。即使是高级用户也应该阅读这一章 —— 相信我。

第二章 *文件 I/O*

本章将介绍文件（Unix 环境中最重要的抽象概念）以及文件 I/O（Linux 编程的基本模式），并且说明如何进行文件的读写以及其他基本的文件 I/O 操作，最后还会探讨 Linux 内核的实现和管理文件的方式。

第三章 *缓冲式 I/O*

本章将探讨基本文件 I/O 接口的问题 —— 缓冲区大小的管理，以及介绍一般的缓冲式 I/O（尤其是标准 I/O）作为解决方案。

第四章 *高级文件 I/O*

本章将以高级的 I/O 接口、内存映射以及优化技术来结束本书对 I/O 的探讨。本章还会附带探讨如何避免搜索磁盘以及 Linux 内核的 I/O 调度程序扮演何种角色。

第五章 *进程管理*

本章将介绍进程（Unix 中第二重要的抽象概念）以及基本进程管理的一系列系统调用，包括令人尊敬的 *fork*。

第六章 *高级进程管理*

本章将继续探讨高级进程管理，包括实时进程。

第七章 *文件和目录管理*

本章将探讨创建、移动、复制、删除以及其他用来管理文件和目录的功能。

第八章 *内存管理*

本章将说明内存管理。首先会介绍 Unix 的内存概念，例如进程地址空间以及页面调度（paging），接着会探讨从内核取得内存空间以及将内存空间释回内核的接口。最后会探讨进阶的内存相关接口。

第九章 *信号*

本章将说明信号。首先会探讨信号以及它们在 Unix 系统中所扮演的角色。然后会说明信号接口，先探讨基本部分，再探讨高级部分。

第十章 *时间*

本章将探讨时间、修眠以及时钟管理。本章会从基本的接口开始谈起，一直到 POSIX 时钟以及高精度定时器。

附录 *GCC 对 C 语言的扩展*

本附录会回顾 *gcc* 和 GNU C 所提供的许多优化功能，例如constant、pure 和 inline 等用来标记函数的属性。

本书最后提供了一份参考书目，列出了对系统编程有帮助以及论述本书未提到的必备知识的相关书籍。

本书所涵盖的版本

Linux 系统接口可被定义成Linux 内核（操作系统的中心组件）、GNU C 链接库（*glibc*）以及 GNU C 编译器（*gcc* —— 现在被正式称为 GNU Compiler Collection，但是我们只关心 C 编译器）三者所提供的应用程序二进制接口（application binary interface）以及应用程序编程接口（application programming interface）。本书内容涵盖 Linux 内核 2.6.22 版、*glibc* 2.5 版以及 *gcc* 4.2 版三者所定义的系统接口。本书对系统接口所做的说明应该可以向下兼容于较旧的版本（不含新的接口）以及向上兼容于较新的版本。

如果任何进化中的操作系统是一个移动的目标，Linux 就是一只饥渴的猎豹。Linux 的进展是以日数（而非年数）来度量的，它频繁地发布内核与其他组件并且在不断变化中。没有哪一本书可以永久捕获这只充满活力的怪兽。

尽管如此，系统编程所定义的编程环境却不再改变。内核开发者尽可能不去改变系统调用，*glibc* 开发者高度重视向上（forward）和向下兼容（backward compatibility），而且 Linux 工具链（toolchain）可以产生跨版本（特别是 C 语言的）的兼容程序代码。因此，尽管 Linux 不断在变化，Linux 系统编程仍能保持稳定。我想说的事情很简单：不要担心系统接口有任何变化，购买本书就对了！

本书的排版惯例

下面是本书的排版惯例：

斜体字（*Italic*）

　　用于强调、新术语、URL、外来词汇、Unix 命令与实用程序、文件名、目录名以及路径名称。

等宽字（`Constant Width`）

　　用于指出头文件、变量、属性、函数、类型、参数、对象、宏以及其他编程结构。

等宽斜体字（*`Constant Width Italic`*）

　　用于指出以用户提供的值替换的文本（例如路径名称组件）。

> 此图标代表诀窍、建议或一般注意事项。

本书大部分的程序将以简洁、实用的程序代码片段来呈现，例如：

```
while (1) {
        int ret;

        ret = fork ();
        if (ret == -1)
                perror ("fork");
}
```

本书煞费苦心地提供简洁而实用的程序代码片段。你不会看到特殊的头文件、没有节制的宏和难以辨认的快捷方式，你不会看到庞大的程序，你所看到的是许多简单的范例程序。本书的范例不仅描述充分、完全可用，而且小而明确，我希望这些范例在读者首次

阅读本书时能提供一个有用的指导，而且在之后多次阅读本书时仍然能成为很好的参考资料。

本书所有范例几乎都是各自独立的。这意味着你可以轻易地将它们复制到你的文本编辑器中，将它们使用在实际的用途上。除非另有提及，否则所有的程序代码在编译时应该不需要用到任何特殊的编译器标志（只有在少数的情况下，才需要连接特殊的链接库）。建议使用如下的命令编译源文件：

```
$ gcc -Wall -Wextra -O2 -g -o snippet snippet.c
```

这条命令会将源文件 *snippet.c* 编译成可执行的二进制文件 *snippet*，启动许多警告检查，进行显著但合理的优化以及调试。本书的程序代码使用这条命令进行编译应该会产生错误或警告 —— 当然，你可能首先需要为 snippet（程序代码片段）建立一个骨架程序。

介绍新函数时，本书会采用常见的 Unix manpage 格式并以黑体字强调，如下所示：

```
#include <fcntl.h>

int posix_fadvise (int fd, off_t pos, off_t len, int advice);
```

所需要的头文件以及任何必要的定义都会放在开头，后面跟着完整的调用原型。

使用范例程序

本书的宗旨是协助你完成工作。一般而言，你可以将本书的程序代码用于你的程序中，或是在文档里提及而无需取得我们的同意，除非你想大幅引用。举例来说，使用本书范例的几个程序代码片段无需经过我们的同意，但如果打算将 O'Reilly 书籍里的范例程序制作成光盘销售或散布则需要授权；引用本书内文或程序代码片段来回答问题不需要授权，但如果在你的产品说明书里大量引用本书范例则需要知会我们。

如果你引用本书（内文或范例程序），我们会感谢你注明出处，但没要求你必须得这么做。如果你愿意，请注明书名、作者、出版公司以及 ISBN。例如，" *Linux System Programming* by Robert Love. Copyright 2007 O'Reilly Media, Inc., 978-0-596-00958-8"。

建议和问题

O'Reilly公司是世界级的计算机信息出版公司。我们永远乐意听到读者对出版品的意见，包括如何让本书可以更好的建议、指正本书的错误或是读者建议本书往后改版时应该再加进来的其他主题。以下是本公司的联络资料：

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

100035 北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室
奥莱利技术咨询（北京）有限公司

我们为本书建立了网页，列出勘误、样例和任何附加的信息。你可以访问：

http://www.oreilly.com/catalog/9780596009588 （原版书）
http://www.oreilly.com.cn/book.php?bn=978-7-5641-1519-7（中文版）

关于本书的评论或者技术问题，发送电子邮件给：

bookquestions@oreilly.com
info@mail.oreilly.com.cn

获得关于我们的图书、会议、资源中心以及 O'Reilly Network 的信息，可访问网站：

http://www.oreilly.com
http://www.oreilly.com.cn

致谢

本书是在许多人的协助之下完成的。尽管可能挂一漏万，不过我真诚地感谢那些一路上鼓励、指导并支持我们的友人们。

Andy Oram 是一位不平凡的编辑人物，没有他的努力我是无法完成这项艰难的工作的。难得的是，Andy 夫妇熟知如何用英语写诗的技术。

Brian Jepson 曾做过本书的编辑，尽管编辑已经换人，他所做的努力仍继续在本书中余波荡漾着。

本书有幸能够找到学有专精的人士作为技术审阅者，没有他们的努力本书将不是你现在看到的样子。这些技术审阅者是 Robert Day、Jim Lieb、Chris Rivera、Joey Shaw 以及 Alain Williams。如果还有任何错误，我应该负最大的责任。

Rachel Head 善尽了她作为文稿编辑（copyeditor）的责任。在她的校稿之下，本书的文稿点缀着红色的墨水 —— 相信读者一定会感激她所做的努力。基于各种理由，我想要感谢 Paul Amici、Mikey Babbitt、Keith Barbag、Jacob Berkman、Dave Camp、Chris DiBona、Larry Ewing、Nat Friedman、Albert Gator、Dustin Hall、Joyce Hawkins、Miguel de Icaza、Jimmy Krehl、Greg Kroah-Hartman、Doris Love、Jonathan Love、Linda Love、Tim O'Reilly、Aaron Matthews、John McCain、Randy O'Dowd、Salvatore Ribaudo 及他的家庭、Chris Rivera、Joey Shaw、Sarah Stewart、Peter Teichman、Linus Torvalds、Jon Trowbridge、Jeremy VanDoren 及他的家庭、Luis Villa、Steve Weisberg 及他的家庭以及 Helen Whisnant。最后要感谢我的父母 Bob 和 Elaine。

—— Robert Love

于波士顿

第一章
介绍与基本概念

这是一本关于系统编程（*system programming*）的书，这是一门关于编写系统软件（system software）的艺术。系统软件位于低层（译注 1），直接跟内核（kernel）及核心系统链接库（core system library）交互。系统软件包括你的 shell、你的文本编辑器、你的编译器和调试器、你的核心实用程序（core utility）以及系统守护进程（system daemon）。这些组件全都是基于内核和 C 链接库的系统软件。许多其他的软件（例如高级GUI 应用程序）大多位于高层，只会偶尔进入低层。有些程序设计者（programmer）会将所有的时间全部投入系统软件的编写上；有些程序设计者则只花部分的时间在此工作上。然而，不论是哪一种程序设计者，都可以从了解系统编程获得好处。不管系统编程是否为程序设计者生活的目标，或者仅是高级概念的基础，都是我们所编写软件的中心所在。

本书专门讨论*Linux* 上的系统编程。Linux 是一个现代的 Unix-like 系统，它是由 Linus Torvalds 以及世界各地组织松散的黑客社群从无到有编写而成的。尽管 Linux 的目标和思想借用自 Unix，但是 Linux 并非 Unix。Linux 的设计完全是按照自己的想法，它的理想与 Unix 背道而驰，与 Unix 一致的只有使用的方式。一般来说，Linux 系统编程的核心部分（core）同任何其他 Unix 系统一样。然而基础之外，Linux 在自己特有的部分有很优异的表现 —— 与传统的 Unix 系统相比较。Linux 里充斥着额外的系统调用、不一样的表现以及新的功能。

译注 1：软件可分成两大类：应用软件和系统软件。应用软件由“为用户而设计的”程序所构成，系统软件由“与计算机硬件交互的”或“提供基础功能的”程序所构成，应用软件的执行必须通过系统软件所提供的基础功能。这就是所谓的软件栈：应用软件位于高层，而系统软件位于低层。

系统编程

就传统来说，所有的Unix编程皆为系统层的编程。以往，Unix系统中并未包括许多层次较高的抽象层。即使是在开发环境（例如X Window System）中的编程，双眼所见的也全都是Unix系统的核心API。因此，大体上来说，本书是一本探讨Linux编程的书。但请注意，本书并不会提到Linux编程环境——找不到*make*的相关指南。本书所要探讨的是一个现代Linux机器提供给我们的系统编程API。

系统编程常被拿来跟应用程序编程作比较。尽管系统层和应用层的编程在某些方面有所不同，但是在其他方面并非如此。与应用程序员截然不同之处在于，系统程序员必须对他们所操作的硬件和操作系统了如指掌。当然，它们所使用的链接库与所进行的调用也不一样。这两者实际上并没有很多互换性，这取决于应用程序被编写在栈中哪一层，但是一般来说，从应用编程转变到系统编程并不难（反之亦然）。尽管应用程序位于栈中的极高层，与最低层的系统程序相去甚远，但是系统编程的知识非常重要。好的实践（good practice）同时适用于任何形式的编程。

过去这几年呈现出一个趋势，应用编程正远离系统层的编程，并通过Web软件（例如JavaScript或PHP）或者托管程序代码（译注2）（例如C#或Java）朝非常高水平的方向发展。事实上，仍需要有人编写JavaScript interpreter（解释器）以及C# runtime（运行时环境），这本身就是系统编程。此外，编写PHP或Java程序的开发者仍可得益于系统编程的知识，因为若能了解其内部的运作原理，你就能够写出较好的程序代码，无论程序代码是在栈中的哪个层次编写的。

虽然应用编程有这样的趋势，但仍有许多的Unix和Linux程序代码是在系统层编写的。其中大部分是用C语言写成，而且主要存在于C链接库和内核所提供的接口上。Apache、*bash*、*cp*、Emacs、*init*、*gcc*、*gdb*、*glibc*、*ls*、*mv*、*vim*以及X皆属于传统的系统编程。这些应用程序都不会很快消失。

系统编程的范畴通常包含内核开发，或者至少包括设备驱动程序的编写。但是本书，如同探讨系统编程的多数书籍一样，并不关心内核开发之事。相反，本书把重点聚焦于用户空间的系统层的编程，也就是内核之上的每一件事（尽管对内核内部的了解对本书的阅读有莫大的帮助）。同样地，网络编程（socket之类）也不在本书讨论之列。设备驱动程序的编写以及网络编程都是大而广泛的题目，最好是以专著来探讨。

什么是系统层的接口，在Linux中如何编写系统层的应用程序？内核和C链接库提供了哪些东西？如何编写出理想的程序代码，Linux提供了哪些工具？相较于其他的Unix

译注2：托管程序代码（managed code）意指程序代码的执行并非通过操作系统，而是通过虚拟机（virtual machine）。然而此发展并未预示系统编程的死亡。

变体，Linux 提供了哪些系统调用？这一切是如何运作的？这些问题都是本书所要探讨的中心议题。

Linux 的系统编程有三个基石：系统调用、C 链接库以及 C 编译器。这三者中的每一个都值得加以介绍。

系统调用

系统编程从系统调用（*system call*）开始。系统调用（常简写为 *syscalls*）就是函数调用（function invocation），从用户空间（你的文本编辑器、游戏程序等）进入内核（系统的基础组件），以便向操作系统请求特定的服务或资源。系统调用的范围从常见的 `read()` 和 `write()` 到少见的 `get_thread_area()` 和 `set_tid_address()` 都涵盖在内。

Linux 所实现的系统调用远少于多数其他操作系统的内核。举例来说，Linux的 i386 架构的系统调用大约有 300 个，而 Microsoft Windows 之上据说有上千个。就 Linux 内核而言，每种机器架构（例如 Alpha、i386 或 PowerPC）都会实现可供自己使用的系统调用。因此，可在某种架构上使用的系统调用可能不同于另一个架构上所使用的。尽管如此，系统调用中有绝大部分（超过90%）是所有架构都会实现的。本书所要探讨的就是这些每种架构都提供的通用接口。

调用(invoking)系统调用

直接链接用户空间应用程序（user-space application）与内核空间（kernel space）是不可能的。基于安全性和可靠性的理由，用户空间应用程序不得执行内核程序代码或者操作内核数据。相反，内核必须提供一种机制，让用户空间应用程序得以“通知”内核它想调用（invoke）的系统调用。然后应用程序可以通过这个定义明确的机制进入内核空间，但是仅能执行内核允许执行的程序代码。实际的机制因架构而异。例如，在 i386 上，用户空间应用程序会使用 `0x80` 这个值来执行软件中断指令 `int`。这个指令会切换到内核空间，在这个受保护的内核领域中，内核会执行软件中断处理器（software interrupt handler）—— 而中断向量 `0x80` 的处理器是什么？难道没有其他的系统调用处理器！

应用程序可经机器寄存器（*machine register*）通知内核它想执行哪个系统调用以及执行时使用哪些参数。系统调用以从 0 开始的数字来指定。在 i386 架构上，如果要调用（invoke）编号为 5 的系统调用（正好是 `open()`），用户空间应用程序在下达 `int` 指令之前应该将数字 `5` 填入寄存器 `eax`。

参数的传递以类似的方式处理。例如，在 i386 上，系使用寄存器来存放每一个可能的参数 —— 寄存器 `ebx`、`ecx`、`edx`、`esi` 和 `edi` 依次用来存放前 5 个参数。一个系统

调用很少会用到 5 个以上的参数，它可以利用一个寄存器指向用户空间中用来存放所有参数的缓冲区。当然，多数系统调用只会用到一对参数。

其他架构处理系统调用调用（system call invocation）的方式则有所不同，然而精神是一样的。身为一个系统编程者，你通常不需要知道内核处理系统调用（invocation）的调用细节。该细节已被编码进架构的标准调用规则（standard calling convention）中，由编译器和 C 链接库自动处理。

C 链接库

C 链接库（*libc*）是 Unix 应用程序的中心组件。即使你是以其他语言进行编程，C 链接库也几乎会派得上用场，通过较高层次链接库的包装，C 链接库可以提供核心服务（core service）以及简化系统调用调用。在现代的 Linux 系统上，C 链接库提供自 *GNU libc*，简写为 *glibc*（念成 *gee-lib-see*，比较少念成 *glib-see*）。

除了实现标准的 C 链接库，*glibc* 还提供了系统调用的封装程序、对线程的支持以及基本的应用程序工具。

C 编译器

在 Linux 中，标准的 C 编译器提供自 *GNU Compiler Collection*（*gcc*）。最初的 *gcc* 指的是 GNU 版的 *cc*（C Compiler）。因此，*gcc* 代表的是 *GNU C Compiler*。随着时间的推进，其所支持的语言越来越多，因此 *gcc* 成为了 GNU 编译器族的通称。然而，*gcc* 也是一个用来调用 C 编译器的二进制文件。在本书中，当我谈到 *gcc* 的时候，我通常是指 *gcc* 程序，除非特别声明。

Unix 系统 —— 包括 Linux，所使用的编译器与系统编程有密切的关系，因为编译器会协助实现 C 标准（参见“C 语言标准”）以及系统 ABI（参见“API 与 ABI”），本章稍后会加以说明。

API 与 ABI

程序员自然会想确保自己的程序能够在自己承诺过要支持的所有系统上运行，不论是现在或是将来。他们希望自己为特定 Linux 发行包所写的程序能够安全地在其他 Linux 发行包以及所支持的其他 Linux 架构和较新（或较旧）的 Linux 版本上运行。

在系统层次上，有两套独立的定义和描述会对可移植性造成影响。一个是应用编程接口（*application programming interface*，简称 API），而另一个是应用程序二进制接口

(*application binary interface*，简称 ABI)。它们可用来定义和描述不同计算机软件之间的接口。

API

API 所定义的接口让一段软件可以在源码层与另一段软件沟通。通过一组标准接口(通常是函数)所提供的抽象层，一段软件(通常是，但未必是较高层者)可以调用另一段软件(通常是较低层者)。举例来说，一个 API 可通过一组函数(用来提供描绘文字所需要的一切功能)把在屏幕上描绘文字的概念抽象化。API 接口仅定义接口，而实际提供 API 功能的软件则被称为 API 的实现 (*implementation*)。

API 常被称为 “contract”(契约)。这并不正确，至少就法律的意义来说，因为 API 并非 two-way agreement (双方所签订的契约)。API 的用户(通常是较高层的软件)对 API 以及它的实现并未有任何的贡献。他可能照原样来使用 API，或者根本就不使用。API 的作用只在确保两段软件按照 API 行事以及具备源码兼容性(*source compatibility*)，也就是说 API 的用户可以在包含 API 实现的情况下成功完成编译。

一个实际的例子是 API 定义自 C 标准，实现自标准 C 链接库。此 API 定义了一套基础的函数，例如字符串操纵例程。

本书将通篇讨论现有的各种 API，例如第三章所探讨的标准 I/O 链接库。本章稍后会探讨对 Linux 系统编程而言最重要的 API。

ABI

API 定义的是源码接口，而 ABI 定义的是特定架构上两段或多段软件之间的低层二进制接口。ABI定义了一个应用程序如何与自己交互、一个应用程序如何与内核交互以及一个应用程序如何与链接库交互。ABI 可确保二进制码兼容性 (*binary compatibility*)，这让一段目标码 (object code) 可以直接使用在具有相同 ABI 的任何系统上而不需要重新编译。

ABI 所关心的问题包括：调用规则 (calling convention)、字节顺序、寄存器使用、系统调用调用、链接、链接库行为以及二进制目标码的格式。例如，调用规则的定义应该包括：函数被调用的方式，将参数传递给函数的方式，哪些寄存器将被保留而哪些将被破坏，以及调用者 (caller) 如何取得返回值。

尽管有人多次尝试为特定架构上的多种操作系统(尤其是 i386 上的 Unix 系统)定义单一 ABI，但是都不是非常成功。于是操作系统(包括 Linux)倾向于定义自己认为合

适的 ABI。ABI 与架构密切相关，绝大多数的 ABI 所涉及的都是机器特有的概念，例如特殊的寄存器或组合指令。因此，每一种机器在 Linux 上都有自己的 ABI。事实上，我们倾向于使用机器的名称（例如 *alpha* 和 *x86-64*）来指称特定的 ABI。

系统程序员应该对 ABI 有所认识，但是通常不需要加以背诵。ABI 是通过工具链（*toolchain*）—— 编译器、链接器等来实施的。然而对 ABI 的了解却能让你设计出较理想的程序，而且这也是编写汇编代码或修改工具链（毕竟这也是系统编程）需要具备的能力。

用于 Linux 上特定架构的 ABI 可以在 Internet 中找到，而且都由该架构的工具链和内核实现。

标准

Unix 系统编程是一门古老的艺术。Unix编程的基础已经十几年未变动过。然而Unix 系统却是静不下来的野兽，不仅行为有所变动，而且加入了新的功能。为了从混乱中理出头绪，标准组织遂将这些系统接口编纂成正式的标准。尽管有众多这样的标准存在，但是从技术上来说，Linux 并未正式遵守其中任何标准。Linux 将目标锁定于两个最重要和最盛行的标准：POSIX 和 Single UNIX Specification（SUS）。

除了其他项目，POSIX 和 SUS 描述了 Unix-like 操作系统接口的 C API。事实上，它们针对 Unix 系统的兼容性定义出了系统编程的方法，或者至少是它的一个共同子集。

POSIX 和 SUS 的历史

20 世纪 80 年代中期，电气及电子工程师学会（Institute of Electrical and Electronics Engineers，常简写成 IEEE）率先致力于 Unix 系统上系统层接口的标准化。自由软件活动的开山鼻祖 Richard Stallman 提议将此标准命名为 *POSIX*（念成 *pahz-icks*），这就是现在的可移植操作系统接口（*Portable Operating System Interface*）标准。

这项努力的第一个成果发表于 1988 年，也就是 IEEE Std 1003.1-1988（POSIX 1988）。1990 年，IEEE 将 POSIX 标准修订为 IEEE Std 1003.1-1990（POSIX 1990）。可选用的实时和线程支持分别被定义于 IEEE Std 1003.1b-1993（POSIX 1993 或 POSIX.1b）和 IEEE Std 1003.1c-1995（POSIX 1995 或 POSIX.1c）中。2001 年，这两个可选用的标准连同基础标准 POSIX 1990 被合并成单一标准：IEEE Std 1003.1-2001（POSIX 2001）。最后一次修订发表于 2004 年 4 月，也就是 IEEE Std 1003.1-2004。以上这些 POSIX 标准被简称为 POSIX.1，并以 2004 年的修订版作为最新版本。

20 世纪 80 年代末期到 90 年代初期，Unix 系统厂商卷入了一场“Unix 大战”，每个厂商都努力想让自己的 Unix 变体成为 Unix 操作系统的标准。几个主要的 Unix 厂商成立了 *The Open Group*，这是一个由 *Open Software Foundation*（*OSF*）和 *X/Open* 合并而成的产业协会（industry consortium）。The Open Group 提供了认证、白皮书以及兼容性测试。90 年代初期，Unix 大战正如火如荼地进行中，The Open Group 发布了 Single UNIX Specification（常简写成 SUS）。SUS 旋即受到欢迎，主要是由于它是免费的，而 POSIX 标准却需要花不少钱。今日，SUS 已经包含了最新的 POSIX 标准。

SUS 标准的第一版发表于 1994 年，符合 SUSv1 标准的系统可以获得 UNIX 95 标章。SUS 标准的第二版发表于 1997 年，符合的系统可以获得 UNIX 98 标章。第三版也就是最新版的 SUS 标准（SUSv3）发表于 2002 年，符合的系统可以获得 UNIX 03 标章。本书将通篇说明符合 POSIX 标准的系统调用以及其他接口。之所以只说明 POSIX 而不说明 SUS，是因为后者涵盖前者。

C 语言标准

Dennis Ritchie 和 Brian Kernighan 的名著《*The C Programming Language*》（Prentice Hall 出版）自从 1978 年出版之后，多年来被视为非正式的 C 规范说明书。这个版本的 C 后来被称为 *K&R C*。C 已经迅速地取代了 BASIC 和其他程序语言成为微电脑编程的通用语言。因此，为了规范当时这个颇为流行的语言，1983 年，美国国家标准学会(American National Standards Institute，简称 ANSI）成立了一个委员会负责开发 C 的正式版本，它纳入了来自各家厂商以及新 C++ 语言的功能和所做的改良。标准化的过程是漫长而辛苦的，至 1989 年总算完成了 *ANSI C* 标准。1990 年国际标准化组织(International Organization for Standardization，简称 ISO）将 ANSI C 标准小修之后通过了 *ISO C90* 标准。

1995 年，ISO 对 C 语言进行改版，发布了 *ISO C95*（但是很少被实现）。之后就是 1999 年的大改版，发布了 *ISO C99*，这个版本加入了许多新功能，包括 inline 函数、新的数据类型、可变长数组、C++ 风格的注释以及新的链接库函数。

Linux 与这些标准

如稍早所述，Linux 的目标锁定在 POSIX 和 SUS 的兼容性。它提供了 SUSv3 和 POSIX.1 所规范的接口，包括可选用的实时（POSIX.1b）和线程（POSIX.1c）支持。更重要的是，Linux 试图提供符合 POSIX 及 SUS 要求的性能。一般而言，与标准不符的地方皆被视为缺陷。Linux 被认为符合 POSIX.1 与 SUSv3 标准，但是它并未正式通过 POSIX

或 SUS 的认证(特别是 Linux 的每个版本以及每个修订版)，所以我不能正式地说 Linux 与 POSIX 或 SUS 兼容。

就程序语言方面的标准而言，Linux 有不错的进展。*gcc* C 编译器支持 ISO C99 标准。此外，*gcc* 还提供了自己对 C 语言所做的扩展。这些扩充统称为 *GNU C*，相关细节可参考本书附录。

Linux 的向上兼容性（forward compatibility）历史并没有发生什么重大的事件（注 1），尽管近来它有很不错的进展。标准所规范的接口，例如标准 C 链接库，显然会始终维持源码兼容（source compatible），而且至少需要维持 *glibc* 主要版本之间的二进制码兼容性（binary compatibility）。因为 C 已经标准化，*gcc* 将始终能够正确编译合法的 C 程序，然而 *gcc* 特有的扩展却可能被废弃以及被新的 *gcc* 版本移除。最重要的是，Linux 会确保系统调用的稳定性。一旦 Linux 内核的稳定版本实现了一个系统调用，它就会固定下来不再变动。

在众多 Linux 发行包中，*Linux Standard Base*（简称 LSB）对大部分的 Linux 系统进行了标准化。LSB 是在 *Linux Foundation*（之前的 *Free Standards Group*）的赞助之下由若干 Linux 厂商所参与的联合计划，目的是提供一个二进制码标准，让目标码不必修改就能够在兼容的系统上运行。多数 Linux 厂商都会提供某种程度的 LSB 兼容性。

本书与这些标准

本书刻意避免对任何标准提供任何口惠而实不至的服务。一般的 Unix 系统编程书籍必定会停下来详细说明某个接口在各标准中的行为有何差异，某个系统调用在各个标准中是否被实现，并以类似的页面填充膨胀的内容。然而，本书只会针对现代 Linux 系统 —— 包括最新版的 Linux 内核（2.6）、*gcc* C 编译器（4.2）以及 C 链接库（2.5），来探讨系统编程。

系统接口会普遍固定下来 —— 例如 Linux 系统开发者会大费周章地避免破坏这些系统调用接口，而且会提供某种程度的源码和二进制码的兼容性。此做法可以让我们深入 Linux 系统接口的细节，而不用去关心其与各种其他 Unix 系统和标准的兼容性。这种将焦点放在 Linux 上的做法让本书得以深入探讨 Linux 特有的先进接口，也使得读者可以和未来接轨。本书利用 Linux 最内部的知识，尤其是 *gcc* 与内核等组件的实现和性能，提供了知情人士的看法，而且随处可见老手的最佳做法以及优化诀窍。

注 1： 有经验的 Linux 用户可能还记得从 *a.out* 切换至 ELF、从 *libc5* 切换至 *glibc*、*gcc* 的变革等。还好，这些日子都已经过去了。

Linux 编程的概念

本节将对 Linux 系统所提供的服务做一个简明的概述。所有的 Unix 系统，包括 Linux，会提供一组共同的抽象概念以及接口。的确，这种通用性定义了 Unix。文件和进程、用于管理 pipe 和 socket 的接口等抽象概念就是 Unix 的基础组件。

本节假设你已经熟悉 Linux 的操作环境：你可以在 shell 中游走、使用基本的命令以及编译简单的 C 程序。本节的内容并非 Linux 或其编程环境的概述，而是 Linux 系统编程的基本知识。

文件以及文件系统

文件是 Linux 中最基本且最重要的抽象概念。Linux 遵循了“一切皆文件”的哲学（尽管做得不如其他系统，例如 Plan9（注 2）。因此，许多操作是通过对文件的读写进行的，即使所操作的对象并非你平日所使用的文件。

文件必须先被打开才可以被访问。文件可以被打开以备读取、写入或是读写。你可以通过一个独一无二的描述符（descriptor）来引用一个已打开的文件，此描述符让我们可以从与已打开文件相关的元数据映射回相应文件。在 Linux 内核内部，此描述符的操作是通过一个称为文件描述符（*file descriptor*，简称 *fd*）的整数（这是 C 的 `int` 数据类型）来进行。*fd* 由用户空间程序所共享，而用户空间程序可以直接使用 *fd* 来访问文件。Linux 系统编程多半就是以 *fd* 来进行文件的打开、操纵、关闭以及其他工作。

常规文件

一般人所谓的文件就是 Linux 所称的常规文件（*regular file*）。常规文件内含数据的字节，而且被组织成一个称为字节流（byte stream）的线性数组（linear array）。在 Linux 中，不会再为文件指定任何格式。这些字节可以具有任何值，而且可以通过任何方式被组织在一起。从系统层来看，除了字节流之外，Linux 不会再对文件套上任何结构。有些操作系统，例如 VMS，会提供高度结构化的文件来支持像记录（*record*）这样的概念。Linux 则不会这么做。

你可以对一个文件内的任何字节进行读或写的操作。这些操作始于一个特定的字节，在文件中这是一个概念上的位置（location）。此位置又称为文件位置（*file position*）或文

注 2： Plan9 是一个诞生自 Bell Labs 的操作系统，它常被称为 Unix 的后继者。它展现了若干创新的概念，而且也是“一切皆文件”哲学的拥护者。

件偏移量（*file offset*）。文件位置是内核配给已打开文件的元数据的基本要素。当一个文件首次被打开，文件位置是零。通常，当文件中的字节被读或写时，会逐字节地递增文件位置。你也可以手动设定文件位置的值，即使此值超过了文件末尾的位置。若直接写入一个字节到文件末尾之后的位置，则会使得中间的字节被填补上零。尽管有可能以此方式将字节写到文件末尾之后的位置，但是却不可能将字节写到文件开头之前的位置。因为这种做法听起来很可笑，而且的确没有什么用处。文件位置从零起算，而且不可以是负值。将一个字节写进一个文件的中间位置，会覆盖之前位于该偏移量的字节。因此，将字节写入文件的中间位置是不可能扩大文件的。多数的文件写入操作发生在文件末尾处。文件位置的最大值受限于用来储存它的 C 数据类型的大小，就现代的 Linux 而言此大小为 64 位。

一个文件的大小以字节为度量单位，这又被称为文件的长度（*length*）。换言之，长度就是组成文件的线性数组中字节的数目。你可经一个称为 *truncation*（截短）的操作来变更一个文件的长度。一个文件可以被截短成比原来尺寸还短的新尺寸，这是有字节从文件末尾被移走的结果。令人困惑的是此操作本身的名称，一个文件竟然还可以“被截短”（truncated）成比原来尺寸还长的新尺寸。在此状况下，新的字节（附加在文件末尾）会被填入零。一个文件可能是空的（长度为零），因此未包含任何有效的字节。文件长度的最大值，如同文件位置的最大值，仅受限于 Linux 内核用来管理文件的 C 数据类型的大小。然而，有些文件系统可能会强行加上自己的限制，所以长度的最大值可能会再小一点。

一个文件可以被不同的进程甚至是相同的进程多次打开。每一个被打开的文件实体都会被赋予一个独一无二的 *fd*，进程之间可以共享它们的 *fd*，这让单一描述符可以供一个以上的进程使用。内核并不会对同时进行的文件访问强加任何限制。可以有多个进程同时自由读写相同的文件。此类同时访问的结果取决于个别操作的先后顺序，通常难以预期。用户空间程序通常需要居间协调这些进程，确保这些同时进行的文件访问可以达到同步。

尽管文件的访问通常是通过文件名称（*filename*），但是文件与此类名称并没有直接的关联性。事实上，文件的引用是通过 *inode*（*information node*，信息节点），它会被赋予一个独一无二的数值。此值又称为 *inode number*（信息节点编号），通常会被简写成 *i-number* 或者 *ino*。inode 用来储存与某个文件有关的元数据，例如文件的时间戳、拥有者、类型、长度以及文件内容的摆放位置 —— 然而就是没有文件名！ inode 既是一个实际的对象（被放在磁盘上有 Unix 风格的文件系统中），也是一个概念性实体（由 Linux 内核中的一个数据结构来表示）。

目录与链接

通过文件的 i-number 来访问文件很麻烦（而且具有潜在的安全漏洞），所以用户空间程序通常会使用名称（而不会使用 i-number）来打开文件。目录（*directory*）可用来提供访问文件时所使用的名称。目录的功能就是把人类可阅读的名称映射至 i-number。将名称和 inode 配成对就称为链接（*link*）。此映射关系 —— 一个简单的表格、哈希表或其他数据结构 —— 在磁盘上的实际形式是由支持指定文件系统的内核程序代码来实现和管理的。因此，目录看起来就像一般文件，不同之处在于目录中只包含名称与 inode 之间的映射关系。内核会直接使用此映射关系进行“名称至 inode”的解析工作。

当一个用户空间应用程序要求打开指定的文件名称时，内核会打开包含文件名称的目录并且搜索指定的名称。内核可以从文件名称找到 i-number，再从 i-number 找到 inode。而 inode 中则包含与相应文件有关的元数据，例如相应文件的数据在磁盘上的位置。

起初，磁盘上只会有一个目录，也就是根目录（*root directory*）。此目录通常会被标记成“/”路径。但是大家都知道，一个系统上通常会出现许多目录。内核如何知道自己所要找的文件名称位于哪个目录？

如稍早所述，目录看起来非常像常规文件。的确，它们甚至会被配发 inode。因此，目录里的链接可以只指向其他目录的 inode。这意味着，目录可以嵌套在其他目录之中形成目录的层次结构。这让所有的 Unix 用户可以使用路径名称（*pathname*）来指称文件 —— 例如 */home/blackbeard/landscaping.txt*。

当内核被要求打开这样的路径名称时，它会走过路径名称中每个目录项（*directory entry*，内核内部称之为 *dentry*）以便寻找下一项的 inode。在前面的例子中，内核会从“/”开始，先取得 *home* 的 inode，从那里取得 *blackbeard* 的 inode，再取得 *landscaping.txt* 的 inode。此操作称为目录或路径名称的解析。Linux 内核还会使用一个缓存区，称为 *dentry cache*，来储存目录解析的结果，以便以后能替 temporal locality 现象提供快捷的解析（注 3）。

一个以根目录起头的路径名称被称为完全限定路径名称（*fully qualified pathname*）或绝对路径名称（*absolute pathname*）。有些路径名称并非完全限定的；相反，它们是相对于其他目录的（例如 *todo/plunder*）。这些路径名称被称为相对路径名称（*relative pathname*）。当所提供的是相对路径名称时，内核会从当前工作目录（*current working directory*）开始进行路径名称的解析工作。内核可以从当前工作目录查找 *todo* 目录，从那里开始，内核可以取得 *plunder* 的 inode。

注 3： *temporal locality*（时间定域性）是指特定资源被访问后，在某段时间内再被访问的概率很高。计算机中有许多资源会呈现 temporal locality 现象。

尽管目录看起来像普通文件，但是内核并不允许你像对常规文件那样打开并操纵它们。事实上，它们的操纵必须通过一组特别的系统调用。这组系统调用让你可以新增和移除链接，总之只有这两项操作是你可以感觉到的。如果用户空间程序不经过内核就可以直接操纵目录，那么很容易会因为一个简单的错误就毁坏文件系统。

硬链接

就概念而言，目前并不存在任何机制可以避免多个名称被解析成同一个 inode。的确，这是允许的。当多个链接将不同的名称映射至同一个 inode 时，我们称这些链接为硬链接（*hard link*）。

硬链接让复杂的文件系统结构可以拥有多个指向相同数据的路径名称。硬链接可以位于相同的目录中，或者位于两个或多个不同的目录中。在这两种状况下，内核都会将路径名称解析成正确的 inode。例如，*/home/bluebeard/map.txt* 和 */home/blackbeard/treasure.txt* 可以硬链接至某个指向特定数据块的 inode。

删除一个文件包括了将它的链接从目录结构中移除，方法很简单，只要将它的名称和 inode 的映射关系从目录中移除即可，这又称为 unlink 操作。然而，因为 Linux 支持硬链接，所以在每次 unlink 操作之后，文件系统不会马上破坏 inode 与它的相关数据。如果文件系统中存在另一个硬链接会怎样？在所有链接被移除之前，为了确保文件不会遭到破坏，每个 inode 都会包含一个“链接计数”（*link count*）字段来记录文件系统中有多少链接指向它。当一个路径名称被移除链接时，链接计数就会被递减 1；只有当它变为 0 时，inode 与它的相关数据才会从文件系统中被实际移除。

符号链接

硬链接无法跨越文件系统，因为 inode number 如果位于 inode 所隶属的文件系统之外便毫无意义。为了让链接可以跨越文件系统，Unix 系统还实现了符号链接（*symbolic link*，常简写成 *symlink*）。

symlink 看起来像常规文件。symlink 具有自己的 inode 以及数据块（其中包含了链接至文件的完整路径名称）。这意味着 symlink 可以指向任何地方，包括不同文件系统上的文件和目录，甚至是尚不存在的文件和目录。symlink 若指向尚不存在的文件称为“不完整的链接”（*broken link*）。

相较于硬链接，symlink 会招致更多的开销，因为 symlink 的解析包括了两个文件（符号链接本身以及所链接的文件）的解析。硬链接不会招致额外的开销——多次访问链接至文件系统的某个文件并没有什么差别，一次只解析一个链接。symlink 的开销其实很小，但是仍然会被视为一项缺点。

symlink 透明的程度不如硬链接。硬链接的使用完全透明（译注 3）；事实上，它需要努力找出一个被链接一次以上的文件！另一方面，symlink 却需要使用特殊的系统调用。symlink 缺乏透明度常被视为一项优点，相较于文件系统内部的链接，symlink 的行为比较像快捷方式（*shortcut*）。

特殊文件

特殊文件就是被表示成文件的内核对象。多年来，Unix 支持若干种特殊文件。Linux 支持四种：块设备文件、字符设备文件、命名管道（named pipe）以及 Unix domain socket。特殊文件是一种让特定的抽象概念可以融入文件系统的方法，符合“一切皆文件”的设计原则。Linux 为创建特殊文件提供了一个系统调用。

在 Unix 系统中，设备的访问是通过设备文件，它的行为和样子看起来就像放在文件系统中的一般文件。设备文件可以被打开来进行读和写，这让用户空间程序可以在系统上访问和操纵（实体或虚拟）设备。Unix设备通常分成两类：字符设备（*character device*）以及块设备（*block device*）。每一种设备类型都拥有自己专属的设备文件。

字符设备可被当成线性字节队列来访问。设备驱动程序会把字节一个接着一个地放入队列（queue），而用户空间程序则会按照字节被放入队列的顺序来读取它们。键盘就是字符设备的一个实例。举例来说，如果用户键入“peg”，则应用程序就会从键盘设备上依次读到 *p*、*e* 和 *g* 等字符。当不再有任何字符可供读取时，设备会返回 end-of-file（EOF）字符。如果遗漏了一个字符或以任何其他顺序来读取它们，都将没有多大的意义。字符设备的访问可通过字符设备文件来进行。

相对而言，块设备可被当成字节数组来访问。设备驱动程序会映射可搜索设备（seekable device）上的字节，而且用户空间程序可以以任何顺序自由访问数组中任何有效的字节——它可以读取字节 12，然后读取字节 7，接着再读取字节 12。块设备通常就是存储设备。硬盘、软盘、光驱以及闪存等全都是块设备，它们都通过块设备文件被访问。

命名导管（*named pipe*，通常称为 *FIFO*，也就是“先进先出”的意思）是一种进程间通信（*interprocess communication*、简称 IPC）机制，它通过文件描述符提供了一个通信管道，你可以通过一个特殊文件来访问它。正规管道（regular pipe）是指将一个程序的输出“用管道传输至”（pipe to）另一个程序的输入的方法，它们是由系统调用在内存中创建的，所以不会存在于任何文件系统中。命名管道的行为很像正规管道，但是它的访问是经一个称为 *FIFO* 的特殊文件来进行的。无关的进程可通过访问该文件来与命名管道通信。

译注 3：它仿佛不存在。

socket 是特殊文件的最后一种类型，它是 IPC 的一种高级形式，让不同的进程得以彼此通信，这些彼此通信的进程不仅可以位于相同的机器上，也可以位于不同的机器上。事实上，socket 的通信形式是网络和 Internet 程序设计的基础。socket 有多种变化形式，其中包括 Unix domain socket，这是 socket 用于机器内部通信的一种形式。尽管 socket 通过 Internet 通信时需要使用主机名称和端口号来识别通信的目标，但 Unix domain socket 却是使用文件系统中的一个特殊文件来进行通信，通常这个特殊文件就称为 socket 文件。

文件系统与命名空间

如同所有的 Unix 系统一样，Linux 为文件和目录提供了一个全局和统一的命名空间（*namespace*）。有些操作系统将磁盘和磁盘驱动器分成两个不同的命名空间 —— 例如，一个位于软盘上的文件可能要经路径名称 *A:\plank.jpg* 来访问，然而硬盘则位于 *C:*。在 Unix 中，同一个位于软盘上的文件可能要通过路径名称 */media/floppy/plank.jpg* 来访问，或通过 *home/captain/stuff/plank.jpg* 来访问，不论文件来自哪个媒体。也就是说，在 Unix 上，命名空间是统一的。

一个文件系统（*filesystem*）就是在正式和有效的层次中一群文件和目录所构成的集合。文件系统可被单独地加入文件和目录的全局命名空间或是从全局命名空间移除。这两项操作称为挂载（*mounting*）和卸载（*unmounting*）。每个文件系统可以被挂载到命名空间中某个位置上，这就是所谓的挂载点（*mount point*）。然后文件系统的根目录可通过此挂载点被访问。举例来说，一片 CD 可以被挂载到 */media/cdrom*，这让 CD 上的文件系统根目录可以通过该挂载点被访问。第一个文件系统会被挂载到命名空间的root 节点，也就是“/”，这被称为根文件系统（*root filesystem*）。Linux 系统通常会具有一个根文件系统。其他的文件系统则可被自由挂载到其他挂载点。

文件系统通常物理存在着（也就是被存储在磁盘上），尽管 Linux 也支持仅存在于内存中的虚拟文件系统（*virtual filesystem*）以及跨越网络存在于远程机器上的网络文件系统（*network filesystem*）。物理文件系统位于块存储设备上，例如光盘、软盘、小型闪存卡（compact flash card）或硬盘。此类设备有些是可分割的，也就是可以划分成多个文件系统，这些文件系统每个都可被独立操纵。Linux 支持广泛的文件系统 —— 当然一般用户有时会希望用到任何文件系统，包括媒体特有的文件系统（例如 ISO9660）、网络文件系统（*NFS*）、原生文件系统（*ext3*）、来自其他 Unix 系统的文件系统（*XFS*），甚至是来自非 Unix 系统的文件系统（*FAT*）。

块设备上最小的可寻址单元就是扇区（*sector*）。扇区是设备的物理特性。扇区的大小通常是 2 的幂次方，512 B 是相当常见的尺寸。块设备无法移动或访问尺寸小于一个扇区的数据单元，所有的 I/O 都是以一个或多个扇区来进行的。

同样地，块（*block*）是文件系统上最小的逻辑可寻址单元。块是文件系统的一个抽象概念，文件系统上并不存在这样的物理媒体。块的大小通常是 2 的幂次方，而且是扇区大小的倍数。块一般都大于扇区，但是必须小于页面尺寸（注 4）（内存管理单元——这是一个硬件组件，所能寻址的最小单位）。常见的块大小有 512B、1 KB 以及 4 KB。

以往，Unix 系统仅使用一个共享的命名空间，系统上所有用户以及所有进程都可见。Linux 采用新的做法并且支持每个进程的命名空间（*per-process namespaces*），这让每个进程可以针对系统的文件和目录层次选择自己的视图（注 5）。默认情况下，每个进程将继承其父进程的命名空间，但是进程可以使用自己所拥有的一组挂载点以及唯一的根目录来创建自己的命名空间。

进程

如果说文件是 Unix 系统中最基本的抽象概念，那么进程就是第二基本的抽象概念。进程（process）是执行中的目标码（object code）：正在运行的程序。但是它不仅仅是目标码，进程由数据、资源、状态以及一个虚拟的计算机组成。

进程的生命始于可执行目标码（executable object code），这是采用内核所了解的可执行格式（Linux 中最常见的格式为 ELF）的机器可运行代码（machine-runnable code）。可执行格式包括元数据（metadata）以及多个代码和数据段（*sections* of code and data）。这些段是会被加载至内存的线性块的目标码的线性块（linear chunk）。区段中所有数据的待遇都一样，它们具有相同的访问权限，而且一般用于相似的用途。

最重要和最常用的段是 *text* 段、*data* 段以及 *bss* 段。text 段中包含可执行代码以及只读数据（例如常变量），而且通常会被标示成只读及可执行。data 段中包含已初始化的数据（例如定义过初始值的 C 变量），而且通常会被标示成可擦写。bss 段中包含尚未初始化的全局数据。因为 C 标准规定 C 变量的默认值基本上都为零，所以不需要将零存入磁盘上的目标码。事实上，目标码只会在 bss 段中列出未初始化的变量，当该段加载进内存时，内核会为此段映射 *zero page*（一个填满零的页面）。一般认为 bss 段就是基于此目的的优化手段。bbs 的名称是一个历史遗迹，代表 *block started by symbol* 或 *block storage segment*。ELF 可执行文件中常用的其他段还有 *absolute* 段（内含不可重定位的符号）以及 *undefined* 段（内含所有未定义的符号）。

进程还会涉及由内核仲裁和管理的各种系统资源。进程通常只会通过系统调用来请求和操纵资源（包括定时器、未决信号、已打开的文件、网络连接、硬件以及 IPC 机制）。

注 4：这是一个人为的内核限制，将来可能会有所改动。

注 5：Bell Labs 的 Plan9 是首先采用此做法的操作系统。

在内核内部，一个进程所配得的资源会随着进程的相关数据及统计值一同被存入进程的进程描述符（*process descriptor*）。

进程是一个虚拟的抽象对象。Linux 内核支持抢占式多任务处理(preemptive multitasking)以及虚拟内存(virtual memory)，并为进程提供虚拟化的处理器以及内存的虚拟化视图。从进程的角度来看，仿佛它独自控制整个系统。也就是说，尽管该进程可能与许多其他进程一同参与调度，但在运行的时候好像是它独自掌控整个系统一样。事实上，内核会无间隙且透明地抢占和重新调度进程（preempts and reschedules processes），与正在运行的所有进程共享系统的处理器。进程绝不会察觉有何异样。同样地，每个进程会被分配单一的线性地址空间，仿佛它独自控制系统中所有内存。通过虚拟内存和页面调度(paging) 机制，内核允许多个进程并存于系统中，让每个进程各自运作在不同的地址空间中。内核通过现代处理器所提供的硬件支持来管理此虚拟化功能，让操作系统得以同时管理多个无关进程的状态。

线程

每个进程由一个或多个执行的线程（*threads of execution*、通常只简写为 *threads*）所构成。线程就是进程内最小的执行单位，该抽象对象负责执行程序代码以及维护进程的运行状态。

多数进程仅由单一线程构成，它们被称为单线程（*single-threaded*），而那些包含多个线程的进程被称为多线程（*multithreaded*）。传统上，Unix 程序都是单线程的，这是因为 Unix 历来着重简单、快速的进程创建时间以及稳固的 IPC 机制，这些都会降低对线程的需求。

线程由堆栈（*stack*，用来存放它的局部变量，如同无线程系统上的进程堆栈）、处理器的状态以及目标码中的当前位置（通常存放在处理器的指令指针中）组成。进程的其余部分多半由所有线程所共享。

就内部而言，Linux 内核仅为线程实现了一个视图：它们就像一般的进程那样共享某些资源（最重要的是地址空间）。在用户空间中，Linux 是以符合 POSIX 1003.1c（称为 *pthread*）的方式来实现线程。当前的 Linux 线程实现(这是 *glibc* 的一部分)称为 *Native POSIX Threading Library*（简称 NPTL）。

进程的层次结构

每个进程可通过一个具唯一性的正整数被标识，这称为进程标识符（*process ID*，简称 pid）。第一个进程的 pid 是 1，之后每个进程都会收到一个独一无二的新 pid。在 Linux 中，所有进程会呈现出一个层次结构，这称为进程树（*process tree*）。进程树的根节点是第一个进程，也就是 *init* 进程，通常就是 *init*（8） 程序。可通过 `fork()` 系统调用创建新进程。此系统调用会替进行调用的进程（calling process）创建一个副本。原本的进

程称为父进程（*parent*），新的进程称为子进程（*child*）。第一个进程除外，每个进程皆有父进程。如果父进程在其子进程之前先终止了，内核会将该子进程重新指派给 init 进程。

当一个进程终止时，它并不会立即从系统中被移除。事实上，内核会把该进程的一部分保留在内存中，这让该进程的父进程可以打听到它的终止状态。这被称为等候（*waiting on*）已终止的进程。一旦父进程等到其已终止的子进程，子进程会被完全移除。一个已被终止的进程，如果没有父进程在等候它，则被称为僵尸进程（*zombie*）。

用户与组

Linux 中的授权机制由用户（*user*）和组（*group*）提供。每个用户会被分配一个独一无二的正整数，称为用户标识符（*user ID*，简称 uid）。每个进程恰好会被关联到一个 uid，用来识别运行进程的用户，这被称为进程的真实用户标识符（*real uid* ）。在 Linux 内核内部，用户ID就代表用户。然而，用户本身在引用自己和其他用户时所使用的是用户名称（*username*），而不是数值形式的 uid。用户名称与相应的 uid 都被存储在 */etc/passwd* 文件中，而链接库例程则会将用户所提供的用户名称映射到相应的 uid。

登录系统（login）时，用户必须给 *login*(1) 程序提供用户名称与通行密码。如果提供的是有效的用户名称与正确的通行密码，则 *login*(1) 程序会衍生（spawn）用户的 *login shell*（至于要执行哪一个 shell，也是在 */etc/passwd* 文件中指定的），并且让 shell 的 uid 等于该用户的 uid。子进程的 uid 继承自其父进程。

用户ID会被分配给一个名为 *root* 的特殊用户。root 用户具有特权，可以对系统做几乎所有事。例如，只有 root 用户可以变更一个进程的 uid。因此，*login*(1) 程序应该由 root 用户来运行。

除了 real uid，每个进程还具有一个有效用户标识符（*effective uid*）、一个被保存用户标识符（*saved uid*）以及一个文件系统用户标识符（*filesystem uid*）。虽然 real uid 总是代表启动进程，不过effective uid 却可以根据各种规则加以变更，这让一个进程可以不同用户的使用权限来执行。saved uid 用来保存原本的 effective uid，它的值可用于判断用户可以切换到哪些effective uid 值。filesystem uid 通常等于effective uid，可用于验证文件系统访问。

每个用户可能隶属一个或多个组，包括 */etc/passwd* 文件中所列出的主要或登录组（*primary or login group*）、以及于 */etc/group* 文件中追加的组（*supplemental group*）。每个进程因此还会被关联到一个相应的组标识符（*group ID*，简称 gid），而且有一个真实组标识符（*real gid*）、一个有效组标识符（*effective gid*）、一个被保存组标识符（*saved gid*）以及一个文件系统组标识符（*filesystem gid*）。进程一般会被关联到某个用户的登录组，而不是后来追加的组。

安全系统会在符合特定条件的情况下允许进程进行某些操作。以往，Unix 都是以非黑即白的方式来作这项决定：只有 uid 为 0 的进程才可以访问，其他进程则不行。近来，Linux 将此安全系统代换成能力更佳的系统。新的系统不再只是进行二进制检查(binary check)，它让内核可以根据粒度更细的设定进行访问控制。

使用权限

如同以往的 Unix 一样，Linux 中也有标准的文件权限与安全机制。

每个文件会被关联到一个拥有用户、一个拥有组以及一组权限位（permission bit)。权限位描述了拥有用户、拥有组以及其他任何人是否有权力读取、写入与执行该文件；这三种用户权限设定各需要用到 3 位，所以总共需要用到 9 位。这些拥有者以及权限位的设定就存储在文件的 inode 中。

就常规文件而言，权限位的概念很容易理解：它们可用来指定打开文件以便读取的能力、打开文件以便写入的能力、执行文件的能力。特殊文件的读取和写入的权限如同常规文件，但是读取和写入的实际意义因不同特殊文件而异。执行的权限则会被特殊文件忽略。就目录而言，读取权限让目录的内容可以被列出，写入权限让新的链接可以被加进目录，而执行权限让目录可以被进入以及用在路径名称中。表 1-1 完整地列出了这9个权限位、它们的八进制值（常用来表示这些位的值)、它们的文本值（如同 *ls* 所显示的方式）以及它们所代表的意义。

表 1-1：权限位与它们的值

位	八进制值	文字值	相应的权限
8	400	`r--------`	拥有的用户可读取
7	200	`-w-------`	拥有的用户可写入
6	100	`--x------`	拥有的用户可执行
5	040	`---r-----`	拥有的组可读取
4	020	`----w----`	拥有的组可写入
3	010	`-----x---`	拥有的组可执行
2	004	`------r--`	其他任何人可读取
1	002	`-------w-`	其他任何人可写入
0	001	`--------x`	其他任何人可执行

除了以往 Unix 所采用的权限机制，Linux 还支持访问控制列表（access control list，简称 ACL)。ACL 让你可以进行更详细、更严格的权限和安全控制，但是必须付出复杂度提高和磁盘存储空间增加的代价。

信号

信号（signal）是一种单向异步通知机制。信号可以从内核送往一个进程，从一个进程送往另一个进程，或者从一个进程送给它自己。信号通常用来通知某个进程发生了某个事件，例如 segmentation fault（分段错误）或者用户按下了 Ctrl+C 键。

Linux 内核大约实现了 30 多种信号（实际的数目因架构而异）。每种信号可以使用数值常量以及文字名称来表示。例如，`SIGHUP` 可用来通知发生了终端机挂起（terminal hangup）的事件，在 i386 架构上此信号可用 `1` 这个值来表示。

除了 `SIGKILL`（通常用于终止进程）和 `SIGSTOP`（通常用于中止进程），进程可以控制当自己收到一个信号时该如何处理。它们可以采用默认的处理方式：可能是终止进程、终止进程并产生 coredump、中止进程或者什么都不做，这取决于所收到的信号。另外，进程可以明确地选择忽略或处理所收到信号。忽略的信号会被直接丢掉。处理的信号会导致用户所提供的信号处理器（*signal handler*）函数被执行。一收到信号，程序就会跳至此函数执行，当控制权从信号处理器交回给程序后，便会从先前被中断的指令处继续执行。

进程间通信

允许进程之间交换信息并通知对方事件的发生是操作系统最重要的工作之一。Linux 内核实现了以往 Unix 所采用的 IPC 机制——包括 System V 与 POSIX 所定义以及标准化的机制——并且实现了若干自己的机制。

Linux 所支持的 IPC 机制包括：管道、命名管道、信号量（semaphore）、消息队列、共享内存以及快速用户空间互斥体（futex，译注 4）。

头文件

Linux 系统编程会用到一些头文件。内核本身以及 *glibc* 都提供了系统编程会用到的头文件。这些头文件包括标准 C 所需要的部分（例如 `<string.h>`）以及 Unix 通常会用到的部分（例如 `<unistd.h>`）。

错误处理

不用说，检查以及处理错误也具有至高无上的重要性。在系统编程中，错误是通过函数的返回值来表示并通过特殊变量 `errno` 来描述。*glibc* 会以透明的方式为链接库和系统调用提供 `errno` 支持。本书所提到的接口绝大多数都会使用此机制来沟通所发生的错误。

译注 4：futex 代表“fast user-space mutex”。

函数会通过一个特殊的返回值来通知调用者发生错误的事实，这个返回值通常是 -1(实际会使用哪个值因函数而异)。此错误值只用来警告调用者有错误发生了，并没有提供任何足以洞察错误为何发生的信息。errno 变量可用来找出错误发生的原因。

此变量定义于 <errno.h>，如下所示：

```
extern int errno;
```

它的值只有在设定过 errno 的函数发生了错误（通常是返回 -1）时才有意义，因为在函数成功执行期间修改此变量的内容是完全合法的。

你可以直接读或写 errno 变量，它是一个可修改的左值（modifiable lvalue)。errno 的值会映射到相应错误的文字描述。预处理指令 #define 还可以用来对应数值形式的 errno 值。举例来说，预处理指令可以将 EACCESS 定义成 1，表示 "permission denied"（拒绝不符合权限的操作）。表 1-2 列出了标准的定义以及相应错误的描述。

表 1-2：错误与相应的描述

预处理指令的定义	描述
E2BIG	Argument list too long（参数列表太长）
EACCESS	Permission denied（拒绝不符合权限的操作）
EAGAIN	Try again（再试一次）
EBADF	Bad file number（文件数目错误）
EBUSY	Device or resource busy（设备或系统资源忙碌中）
ECHILD	No child processes（无子进程）
EDOM	Math argument outside of domain of function（数学参数超出函数的定义域）
EEXIST	File already exists（文件已存在）
EFAULT	Bad address（错误的地址）
EFBIG	File too large（文件太大）
EINTR	System call was interrupted（系统调用被禁止）
EINVAL	Invalid argument（无效的参数）
EIO	I/O error（I/O 错误）
EISDIR	Is a directory（是一个目录）
EMFILE	Too many open files（打开的文件太多）
EMLINK	Too many links（太多的链接）
ENFILE	File table overflow（文件表溢出）
ENODEV	No such device（无此设备）
ENOENT	No such file or directory（无此文件或目录）
ENOEXEC	Exec format error（exec 格式错误）

表 1-2：错误与相应的描述（续表）

预处理指令的定义	描述
ENOMEM	Out of memory（内存溢出）
ENOSPC	No space left on device（磁盘空间不足）
ENOTDIR	Not a directory（不是一个目录）
ENOTTY	Inappropriate I/O control operation（不恰当的 I/O 控制操作）
ENXIO	No such device or address（无此设备或地址）
EPERM	Operation not permitted（操作不许可）
EPIPE	Broken pipe（管道断掉）
ERANGE	Result too large（结果太大）
EROFS	Read-only filesystem（只读文件系统）
ESPIPE	Invalid seek（不合法的寻找）
ESRCH	No such process（无此进程）
ETXTBSY	Text file busy（文本文件忙）
EXDEV	Improper link（不适用的链接）

C 链接库提供了可以把 errno 的值转译成相应文字描述的若干函数。这只在进行错误报告之类的工作时需要，错误的检查和处理可以直接使用预处理指令的定义以及 errno。

此类函数中第一个要介绍的是 perror()：

```
#include <stdio.h>

void perror (const char *str);
```

此函数会将函数发生错误的原因输出至 *stderr*（标准错误）：先输出 str 所指向的字符串，接着输出冒号，最后输出错误描述字符串（其内容取决于 errno 的值）。str 所指向的字符串一般会设定成发生错误的函数的名称，例如：

```
if (close (fd) == -1)
        perror ("close");
```

C 链接库还提供了 strerror() 和 strerror_r() 等函数，它们的原型分别是：

```
#include <string.h>

char * strerror (int errnum);
```

与：

```
#include <string.h>

int strerror_r (int errnum, char *buf, size_t len);
```

前一个 strerror() 函数会根据参数 errnum 的值找到错误描述字符串并返回指向

该字符串的指针。应用程序不得直接修改此字符串，不过可以通过随后的 perror() 和 strerror() 调用予以修改。此函数不具线程安全性。

strerror_r() 函数具线程安全性。它的功能如同 strerror()，但是会把找到错误描述字符串填入 buf 所指向的长度为 len 的缓冲区。如果 strerror_r() 的调用成功会返回 0，否则会返回 -1。好笑的是，它会在自己发生错误时设定 errno。

对某些函数而言，整个系列的返回类型都是合法的返回值。因此，调用此类函数之前必须将 errno 的值设为零，并于事后进行检查（此类函数只会在实际发生错误时返回非零值的 error）。例如：

```
errno = 0;
arg = strtoul (buf, NULL, 0);
if (errno)
        perror ("strtoul");
```

检查 errno 时常犯的错误就是忘记任何链接库或系统调用都可以改变它。例如，下面这段有缺陷的程序代码：

```
if (fsync (fd) == -1) {
        fprintf (stderr, "fsync failed!\n");
        if (errno == EIO)
                fprintf (stderr, "I/O error on %d!\n", fd);
}
```

要让 errno 的值可以跨函数调用，你必须保存它：

```
if (fsync (fd) == -1) {
        int err = errno;
        fprintf (stderr, "fsync failed: %s\n", strerror (errno));
        if (err == EIO) {
                /* 如果错误与 I/O 有关，弃船 */
                fprintf (stderr, "I/O error on %d!\n", fd);
                exit (EXIT_FAILURE);
        }
}
```

在单线程的程序中，errno 是全局变量，正如本节稍早所述。然而，在多线程程序中，errno 会按每个线程来存储，因此具有线程安全性。

向系统编程迈进

这一章介绍了 Linux 系统编程的基础知识，并以程序设计者的观点概述了 Linux 系统。下一章将讨论基本的文件 I/O（输入／输出）。当然，这包括了文件的读写操作，然而因为 Linux 将许多接口实现成文件，所以文件 I/O 比文件重要多了。

“开场白”已经远远落在我们后面，逐渐消失在地平线中，迈向系统编程的时间已到。前进吧！

第二章

文件 I/O

本章会说明读写文件的基本原理。此类操作是 Unix 系统的基础功能。第三章将说明标准 C 链接库所提供的标准 I/O 功能，并于第四章说明更高级和特殊的 I/O 接口。最后以第七章所介绍的文件和目录操作为 I/O 的探讨画下完美的句点。

一个文件必须先打开，才可以对它进行读取操作。内核会替每个进程维护一份已打开文件列表，称为文件表（*file table*）。此表是通过非负整数被索引，而此非负整数称为文件描述符（*file descriptor*，常简写为 *fd*）。该表中每个条目内含与已打开文件相关的信息，包括一个指针（指向文件 inode 在内存中的副本）以及相关的元数据（例如文件位置与访问模式）。用户空间与内核空间都会把文件描述符当成独一无二的“每个进程识别符号”(per-process cookies，译注 1)。打开一个文件便会返回一个“文件描述符”，而随后的操作（读取、写入等）则会以“文件描述符”作为主要的参数。

默认情况下，子进程会取得其父进程的文件表。已打开文件列表与它们的访问模式、当前的文件位置等全都一样，但是一个进程所做的改变（例如，子进程关闭了一个文件）不会影响其他进程的文件表。然而，正如你将会在第五章看到的，子进程与父进程可以共享父进程的文件表（如同线程那样）。

文件描述符会被表示成 C 的 `int` 数据类型，不再使用以往 Unix 都在使用的特殊数据类型 —— 例如 `fd_t`。每个 Linux 进程所能打开的文件数目都有上限。文件描述符从 `0` 开始起算，可以一直增长到比上限值少 1 的数目。默认情况下，上限值为 1,024，但是最高可设定到 1,048,576。因为负值并非有效的文件描述符，所以 `-1` 通常代表函数发生了错误，否则会返回有效的文件描述符。

译注 1：“per-process”是指“每个进程”的意思，所以“per-process cookies”是指“每个进程都会拥有的识别符号”。

除非进程显式地予以关闭，否则每个进程按照惯例至少会打开三个文件描述符（fd）：0、1 和 2。fd 0 是 *standard in*（标准输入，常简写为 *stdin*），fd 1 是 *standard out*（标准输出，常简写为 *stdout*），而 fd 2 是 *standard error*（标准错误，常简写为 *stderr*）。实际使用时并不会直接使用这些整数值，而是使用 C 链接库通过预处理指令所定义的 STDIN_FILENO、STDOUT_FILENO 和 STDERR_FILENO。

请注意，文件描述符所能引用的不仅是常规文件。它们还可用于访问设备文件和管道、目录和 futex、FIFO 以及 socket —— 遵奉“一切皆文件”的哲学，任何东西，只要可以对它进行读或写操作，几乎就能通过文件描述符来对它进行访问。

打开文件

访问文件的最基本方法就是通过 read() 和 write() 系统调用。然而，访问文件之前，必须先通过 open() 或 creat() 系统调用打开该文件。一旦文件使用完毕，应该使用 close() 系统调用予以关闭。

open() 系统调用

使用 open() 系统调用打开文件之后，就会取得一个文件描述符：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open (const char *name, int flags);
int open (const char *name, int flags, mode_t mode);
```

open() 系统调用用来将路径名称 name 所指定的文件映射至一个文件描述符，并于映射成功后返回该文件描述符。文件位置会被设为零，而且会根据 flags 所指定的标志来打开该文件以便进行访问。

open() 的标志

flags 参数必须是 O_RDONLY、O_WRONLY 或 O_RDWR 其中之一。这些标志值可分别让所打开的文件用于读取、写入、读写操作。

举例来说，下面的程序代码会打开 */home/kidd/madagascar* 来进行读取操作：

```
int fd;

fd = open ("/home/kidd/madagascar", O_RDONLY);
if (fd == -1)
        /* 错误 */
```

一个为写入操作所打开的文件，你无法对它进行读取操作，反之亦然。进行 open() 系统调用的进程必须具备足够的权限方能进行所要求的操作。

设定 flags 参数时，可将下面所列的值以 bitwise-OR（按位 OR 逻辑运算）的方式组合在一起，以便调整打开要求的行为：

O_APPEND

以附加模式（*append mode*）打开文件。也就是在每次进行写入操作之前，文件位置将被更新成指向文件结尾。即使进程上次进行写入操作之后有另一个进程也对文件进行了写入操作，也会因此变更文件位置（参见本章稍后的“附加模式”这一节）。

O_ASYNC

如果所指定的文件变成可读取或可写入的，则产生一个信号（默认为 SIGIO）。此标志可用于终端机以及 sockets，但不可用于常规文件。

O_CREAT

如果 name 所指定的文件尚不存在，则内核将会创建它；如果该文件已经存在，除非同时指定了 O_EXCL，否则此标志将毫无作用。

O_DIRECT

打开文件以便进行直接 I/O（参见本章稍后的“直接 I/O”这一节）。

O_DIRECTORY

如果 name 不是一个目录，则 open() 调用会失败。此标志于内部供 opendir() 链接库调用所用。

O_EXCL

指定 O_CREAT 时，如果 name 所指定的文件已经存在，此标志将导致 open() 调用失败。这可用于避免文件的创建出现竞争条件（race condition）。

O_LARGEFILE

所指定的文件将采用 64 位的偏移量打开，以便打开尺寸大于 2GB 的文件。在 64 位架构上这是默认值。

O_NOCTTY

如果 name 所指定的是一个终端设备（例如 */dev/tty*），它不会变成进程的控制终端，即使进程当前并不具有控制终端。此标志并不常被用到。

O_NOFOLLOW

如果 name 是一个符号链接，则 open() 调用将会失败。一般情况下，链接会被解析，而且会打开目标文件。如果所指定的路径中有其他元素是链接，则 open() 调用仍旧会执行成功。举例来说，当 name 是 */etc/ship/plank.txt* 时，如果 *plank.txt* 是一个符号链接，则 open() 调用将会失败。然而，如果 *etc* 或 *ship* 是符号链接，则只要 *plank.txt* 不是符号链接，open() 调用就会成功执行。

O_NONBLOCK

如果可能，将以非阻挡模式（nonblocking mode）打开文件。这使得 open() 调用以及任何其他操作都不会让进程在进行 I/O 时受到阻挡（进入休眠状态）。此行为只能用于定义 FIFO。

O_SYNC

打开文件以便进行同步 I/O。直到数据被实际写入磁盘之前都不会完成写入操作，一般的读取操作已经是同步的方式了，所以此标志并不会影响读取操作。POSIX 额外定义了 O_DSYNC 和 O_RSYNC，在 Linux 上，这两个标志与 O_SYNC 是同义词（参见本章稍后的“O_SYNC 标志”这一节）。

O_TRUNC

如果文件存在，是常规文件、而且所指定的标志允许进行写入操作，则文件将被截短成零长度。对 FIFO 或终端设备使用 O_TRUNC 会被忽略。其他类型文件的用法并未被定义。同时指定 O_RDONLY 与 O_TRUNC 的用法也未被定义，因为你需要对文件进行写入操作以便截短它。

举例来说，下面的程序代码会打开文件 */home/teach/pearl* 以便进行写入操作。如果文件已经存在，则将它截短成零长度。因为并未指定 O_CREAT 标志，如果文件尚不存在，则 open() 调用会失败：

```
int fd;

fd = open ("/home/teach/pearl", O_WRONLY | O_TRUNC);
if (fd == -1)
        /* 错误 */
```

新文件的拥有者

判断哪个用户拥有新文件非常简单：文件拥有者的 uid 就是创建文件的进程的有效 uid。

判断拥有组则比较复杂。默认的做法就是将文件的拥有组设定为创建文件的进程的有效 gid。这是 System V 的行为（大部分 Linux 系统的行为模型），也是标准 Linux 的做法（*modus operandi*）。

然而麻烦的是BSD 却定义了自己的行为：文件的拥有组会被设定为上层目录的 gid。你可以通过挂载选项（注 1）让 Linux 采用此行为 —— 如果文件的上层目录设定了组 ID（setgid）位，则这也是 Linux 默认的行为。尽管大部分 Linux 系统采用的是 System V 的行为（新文件的拥有组就是创建进程的 gid），由于可能会采用 BSD 的行为（新文件的拥有组就是上层目录的 gid），这意味着，在乎此事的程序代码需要通过 chown() 系统调用手动设定拥有组（参见第七章）。

幸好，一般并不需要在乎文件的拥有组。

新文件的使用权限

前面所列出的两种 open() 系统调用格式都是正确的。除非你是在创建文件，否则 mode 参数会被忽略；如果指定 O_CREAT 标志，则需要使用 mode 参数。如果你在使用 O_CREAT 时忘了提供 mode 参数，则其结果未定义，而且通常会很麻烦 —— 所以不要忘了！

当一个文件被创建时，mode 参数可用来为新创建的文件设定使用权限。以这个特殊的方式打开文件时并不会检查 mode 参数所设定的使用权限，所以你可以进行与使用权限抵触的操作，例如打开文件以进行写入操作时，却替文件设定只读的使用权限。

参数 mode 就是人们熟悉的 Unix 使用权限位设定值，例如 0644 这个八进制值（拥有者可以进行读写操作，其他任何人只能进行读取操作）。就技术而言，POSIX 允许实际的设定值因实现而异，这让 Unix 系统得以安排自己所期望的权限位。为了弥补 mode 参数中位的摆放位置的不可移植性，POSIX 引进了如下可以用 binary-OR（二元 OR 逻辑运算）组合在一起的常量来供 mode 参数所使用：

S_IRWXU

拥有者具有读取、写入和执行的权限。

S_IRUSR

拥有者具有读取的权限。

S_IWUSR

拥有者具有写入的权限。

S_IXUSR

拥有者具有执行的权限。

S_IRWXG

组具有读取、写入和执行的权限。

注 1：挂载选项 bsdgroups 或 sysvgroups。

S_IRGRP

组具有读取的权限。

S_IWGRP

组具有写入的权限。

S_IXGRP

组具有执行的权限。

S_IRWXO

其他所有人具有读取、写入和执行的权限。

S_IROTH

其他所有人具有读取的权限。

S_IWOTH

其他所有人具有写入的权限。

S_IXOTH

其他所有人具有执行的权限。

磁盘实际的权限位设定取决于“mode 参数”与“用户的文件创建掩码（*umask*）的补码”进行 binary-AND（二元 AND 逻辑运算）的结果。简而言之，umask 中的位用来关闭 open() 系统调用的 mode 参数中相应的位。因此，值为 022 的 umask 会让值为 0666 的 mode 参数变成 0644（0666 & ~022 的结果）。身为一个系统编程人员，设定使用权限时，你通常不需要考虑 umask 的问题 —— umask 的存在是让用户可以在他的程序创建新文件时限制其使用权限。

举一个例子，下面的程序代码会打开 file 参数所指定的文件来进行写入操作。如果文件不存在，假设 umask 的值为 022，则所创建的文件具有 0644 的权限设定值（即使 mode 参数的值为 0664）。如果文件已经存在，该文件就会被截短成零长度：

```
int fd;

fd = open (file, O_WRONLY | O_CREAT | O_TRUNC,
           S_IWUSR | S_IRUSR | S_IWGRP | S_IRGRP | S_IROTH);
if (fd == -1)
        /* 错误 */
```

creat() 函数

由于 O_WRONLY | O_CREAT | O_TRUNC 的组合太常见了，所以有一个系统调用专门提供此行为：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int creat (const char *name, mode_t mode);
```

没错，函数的名字漏掉了一个“*e*”。Unix 的创造者 Thompson 曾经开玩笑地说，设计 Unix 时遗漏字母是他最大的遗憾。

下面是典型的 creat() 调用：

```
int fd;

fd = creat (file, 0644);
if (fd == -1)
        /* 错误 */
```

其行为与下面的程序代码是同样的：

```
int fd;

fd = open (file, O_WRONLY | O_CREAT | O_TRUNC, 0644);
if (fd == -1)
        /* 错误 */
```

在多数 Linux 架构中（注 2），creat() 是一个系统调用，尽管在用户空间中实现它很容易：

```
int creat (const char *name, int mode)
{
        return open (name, O_WRONLY | O_CREAT | O_TRUNC, mode);
}
```

这种重复的现象是历史遗迹，因为以前的 open() 只有两个参数。如今，保留 creat() 系统调用的目的只是为了兼容性。新的架构可以在 *glibc* 中自行实现 creat()。

返回值与错误代码

open() 与 creat() 会在执行成功时返回一个文件描述符。执行失败时，会返回 -1 并将 errno 设定为适当的错误值（第一章讨论过 errno 并列出了可能的错误值）。处理文件打开时所发生的错误并不困难，因为撤销打开的行为只需要采取少许的步骤或者根本不需要有任何作为。典型的反应就是提示用户使用不同的文件名或者直接终止程序。

注 2： 如先前所述，系统调用的定义因架构而异。因此，尽管 i386 有一个 creat() 系统调用，但是 Alpha 并没有。当然，你可以在任何架构上使用 creat()，但是你所使用的可能是库函数而不是架构本身的系统调用。

以 read() 进行读取操作

知道如何打开文件之后，现在让我们来看看如何读取文件，下一节则会研究写入操作。

读取操作所使用的最基本且最常见的机制就是 POSIX.1 所定义的 read() 系统调用：

```
#include <unistd.h>

ssize_t read (int fd, void *buf, size_t len);
```

每次调用 read() 就会从 fd参数所引用文件的当前文件位置读取 len 个字节到 buf。执行成功后会返回写入 buf 的字节数目，执行失败时会则返回 -1 并设定 errno。文件位置会前进到从 fd 所读取的字节数目。如果 fd 所代表的对象不具有查找位置的能力（例如这是一个字符设备文件），则每次只会从“当前”位置读取数据。

基本的用法很简单。下面的范例会从文件描述符 fd 读取字节到 word。所读进的字节数目等于 unsigned long 数据类型的大小，在 32 位的 Linux 系统上这是4个字节的大小，在 64 位系统上这是 8 个字节的大小。返回时，nr 等于所读进的字节数目，如果发生错误则等于 -1：

```
unsigned long word;
ssize_t nr;

/* 从 'fd' 读取字节到 ' word' */
nr = read (fd, &word, sizeof (unsigned long));
if (nr == -1)
        /* 错误 */
```

这个简单的实现有两个问题：read() 调用可能在读完 len 个字节之前就返回了，因此可能会产生程序代码没有检查到以及无法处理的错误。遗憾的是，此类程序代码非常常见。让我们来看看如何改进它。

返回值

read() 返回小于 len 的非零正数并不算错误。发生此情况的原因可能是：可供读取的字节数目小于 len，系统调用可能收到信号而中断，管道可能坏了（如果 fd 是管道）等。

使用 read() 时，返回值为 0 的可能性是另一个应该考虑的因素。read() 系统调用会返回 0 以指示到达了文件末端（*end-of-file*，常简写为 EOF），在此情况下，当然没有字节可供读取。EOF 并不算错误（因此不会以 -1 作为返回值），这表示文件位置已经超过了文件中最后一个有效偏移值，因此没有任何其他字节可供读取。然而，如果你

以 read() 调用读取 len 个字节，但是可供读取的字节尚未出现，则此调用将受到阻挡（进入休眠状态），直到出现可供读取的字节（假设文件描述符被开启时并未进入非阻挡模式，参见本章稍后的“非阻挡式读取操作”一节）。注意，这不同于返回 EOF，也就是说，“no data available”（无数据可用）与“end of data”（到达数据末端）并不相同。就 EOF 的情况而言，这是指到达了文件末端；就受阻挡的情况而言，读取操作正在等待更多的数据 —— 就像对一个 socket 或一个设备文件进行读取操作的情况。

有些错误是可恢复的。举例来说，如果在进行 read() 调用时，却在读取任何字节之前因收到信号而中断，它便会返回 -1（返回 0 可能会与 EOF 混淆），而且 errno 会被设为 EINTR。在此情况下，你可以再次进行 read() 调用。

的确，read() 调用的结果有许多种可能性：

- 此调用的返回值等于 len。所读取的 len 个字节会被存放到 buf 中。这是预期的结果。
- 此调用的返回值小于 len，但是大于零。所读取的字节会被存放到 buf 中。发生此情况的原因可能是：读取操作进行期间因收到信号而中断，读取操作进行期间发生了一个错误，可供读取的字节数目大于零但小于 len，或者读取到 len 个字节之前先到达了文件末端。重新进行 read() 调用（在 buf 和 len 的值根据前一次的进度做过更新的情况下）可把剩余的字节读进剩余的缓冲区或指出问题的原因。
- 此调用的返回值为 0。这表示 EOF（到达了文件末端）。没有数据可读。
- 此调用被阻挡，因为可供读取的数据目前尚未出现。在非阻挡模式中并不会发生此情况。
- 此调用的返回值为 -1，而且 errno 被设为 EINTR。这表示在任何字节被读取之前收到了一个信号。你可以再次进行此调用。
- 此调用的返回值为 -1，而且 errno 被设为 EAGAIN。这表示读取操作被阻挡，因为可供读取的数据目前尚未出现，而且稍后应该再次进行此调用。此情况只发生在非阻挡模式中。
- 此调用的返回值为 -1，而且 errno 被设为 EINTR 或 EAGAIN 以外的其他值。这表示一个更严重的错误。

读取所有的字节

这些可能性意味着，如果你想处理所有错误并实际读取到 len 个字节（至少读取到 EOF），那么简单使用前面所列的 read() 并不合适。事实上，你需要一个循环以及若干条件语句：

```
ssize_t ret;

while (len != 0 && (ret = read (fd, buf, len)) != 0) {
        if (ret == -1) {
                if (errno == EINTR)
                        continue;
                perror ("read");
                break;
        }

        len -= ret;
        buf += ret;
}
```

这段程序代码可以处理5种可能会碰到的情况。程序的循环会从 fd 的当前文件位置读取 len 个字节到 buf。它会继续读取完 len 个字节或直到抵达 EOF。如果所读取的字节数目大于零，但是小于 len，则 len 的值会减去已读取的字节数目，buf 的值会加上已读取的字节数目，并且再次进行此调用。如果此调用返回 -1 而且 errno 等于 EINTR，则不需要更新参数就可以再次进行此调用。如果此调用返回 -1 而且 errno 被设为任何其他值，则会调用 perror() 以便将相关说明输出至标准错误而且会终止循环。

部分读取不仅合法而且也很普遍。无数的缺陷源自程序设计者不能妥善检查并处理供应量不足的读取要求。不要将这种可能性加入列表！

非阻挡式读取操作

有时，程序设计者并不想让对 read() 的调用在尚无数据可用时受到阻挡。他们宁可让调用立即返回，指出尚无数据可用。这称为非阻挡式 I/O，它让应用程序可以在不受阻挡的情况下对多个文件进行 I/O 操作，但是会因此错过另一个文件中的可用数据。

因此，值得检查一个额外的 errno 值：EAGAIN。正如稍早所讨论的，如果所指定的文件描述符是以非阻挡模式打开的（如果调用 open() 时指定了 O_NONBLOCK，参见"open() 的标志"）而且可供读取的数据尚未出现，则 read() 调用会返回 -1 并将 errno 设定为 EAGAIN 来不被阻挡（也就是说不会进入休眠状态）。进行非阻挡式读取操作时你必须检查 EAGAIN，否则你就会冒着"严重错误"与"只是缺乏数据"相混淆的风险。举例来说，你可能会使用如下的程序代码：

```
char buf[BUFSIZ];
ssize_t nr;

start:
nr = read (fd, buf, BUFSIZ);
if (nr == -1) {
```

```
        if (errno == EINTR)
                goto start; /* 哦! 嘘!  */
        if (errno == EAGAIN)
                /* 稍后重提交 */
        else
                /* 错误 */
}
```

这个例子中处理 EAGAIN 的方法如果使用 goto start，则实际没有多大意义——这样不如不使用非阻断式 I/O。这么做根本没有节省时间，而且会因为循环而不断引入更多开销。

其他错误值

还有其他错误码用来表示编程错误或低级问题（例如 EIO）。read() 执行失败时可能会设定的 errno 值包括：

EBADF

所指定的文件描述符无效或者未打开以备读取。

EFAULT

buf 所提供的指针并非指向进行调用进程的地址空间范围内。

EINVAL

文件描述符被映射到一个不允许读取操作的对象。

EIO

发生了一个低级的 I/O 错误。

read()的字节数量限制

size_t 与 ssize_t 是 POSIX 所规定的数据类型。size_t 数据类型所存储的值用于度量以字节为单位的数目。ssize_t 数据类型则是 size_t 有正负号的版本（负值代表错误）。在 32 位的系统上，size_t 与 ssize_t 背后的 C 数据类型通常分别是 unsigned int 与 int。因为这两种数据类型通常会一起使用，所以范围较小的 ssize_t 会对 size_t 的范围造成限制。

size_t 的最大值是 SIZE_MAX，ssize_t 的最大值是 SSIZE_MAX。如果 len 的值大于 SSIZE_MAX，则 read() 调用的结果未定义。在多数 Linux 系统上，SSIZE_MAX 就是 LONG_MAX，在 32 位的系统上，其值为 *0x7fffffff*。对一次读取操作而言，这是个相当大的数目，不过有件事你必须铭记在心。如果你把前面的读取循环用于读进大量的数据，你可能会想进行下面的操作：

```
if (len > SSIZE_MAX)
        len = SSIZE_MAX;
```

调用 read() 时，如果将 len 设为 0，则此调用会立即返回 0。

以 write() 进行写入操作

进行写入操作时，write() 是最基本和最常见的系统调用。write() 是 read() 的反向操作，而且也定义在 POSIX.1 中：

```
#include <unistd.h>

ssize_t write (int fd, const void *buf, size_t count);
```

进行 write() 调用，就会从 buf 开始将 count 个字节写入 fd 所指定文件的当前文件位置。如果文件背后的对象不支持查找位置的功能（例如字符设备），则总是会从头（head）写起。

执行成功时，会返回所写入的字节数目，并以同样的方式更新文件位置。发生错误时，会返回 -1 并且为 errno 设定适当的值。进行 write() 调用有可能会返回 0 这个值，但是这个返回值并没有任何意义，它仅意味着被写入了 0 个字节。

如同 read()，write() 也有最基本的用法：

```
const char *buf = "My ship is solid!";
ssize_t nr;

/* 将 'buf' 里的字符串写入 'fd' */
nr = write (fd, buf, strlen (buf));
if (nr == -1)
        /* 错误 */
```

但同样地，和 read() 一样，这种用法也不太正确。调用者还需要检查部分写入的可能性：

```
unsigned long word = 1720;
size_t count;
ssize_t nr;

count = sizeof (word);
nr = write (fd, &word, count);
if (nr == -1)
        /* 错误，检查 errno */
else if (nr != count)
        /* 可能发生错误，但是未设定 'errno' */
```

部分写入

相比较于返回部分读取结果的 read() 系统调用，write() 系统调用不太可能会返回部分写入的结果。此外，write() 系统调用也不会遇到 EOF 的情况。就常规文件而言，write() 保证能够完成所要求的整个写入操作，除非有错误发生。

因此，就常规文件而言，你并不需要以循环来进行写入操作。然而，就其他文件类型来说（例如，sockets），可能需要使用循环才有办法保证实际写出所要求的所有字节。使用循环的另一个好处是第二次调用 write() 所返回的错误将可显示是何原因让第一次调用仅写入部分字节（但是，同样地，此情况并不常见）。下面是一个例子：

```
ssize_t ret, nr;

while (len != 0 && (ret = write (fd, buf, len)) != 0) {
        if (ret == -1) {
                if (errno == EINTR)
                        continue;
                perror ("write");
                break;
        }

        len -= ret;
        buf += ret;
}
```

附加模式

如果以附加模式打开 fd（使用O_APPEND标志），写入的动作并不会发生在文件描述符的当前文件位置，而会发生在当前文件的末端位置。

举例来说，假设有两个进程对同一个文件进行写入操作。不使用附加模式时，如果第一个进程将字节写入文件的末端，然后第二个进程进行同样的操作，则第一个进程的文件位置将不再指向文件的末端，而会指向文件的末端位置减去第二个进程所写入的字节数目后的位置。这就意味着，在没有明确采用任何同步化机制的情况下，不可以有多个进程对相同的文件进行附加操作，因为这将会造成竞争条件。

采用附加模式可避免此问题。它可以确保文件位置总是被设定为文件末端，所有写入操作总是附加到文件末端，即使存在多个写入者。你可以把前面的每个写入请求想成是会对文件位置进行不可分割的更新，因此文件位置会被更新成刚才所写入的字节的末端。这与下一个 write() 调用无关，因为文件位置的更新是自动进行的，但是如果因为某些奇怪的理由而让下一个调用是 read()，那么可能就无法置身事外了。

附加模式对某些任务（例如更新日志文件）而言可能意义重大，但是对许多其他任务而言可能意义不大。

非阻挡式写入操作

如果以非阻挡模式打开fd(使用O_NONBLOCK标志)，而写入请求被阻挡，则 write() 系统调用会返回 -1 且 errno 会被设为 EAGAIN。稍后应该再送出写入请求。一般来说，常规文件并不会发生此情况。

其他错误代码

其他值得注意的 errno 值包括：

EBADF
: 所指定的文件描述符无效或者未打开以备写入。

EFAULT
: buf 所提供的指针并非指向进行调用进程的地址空间范围内。

EFBIG
: 写入操作所产生的文件大小超过了每个进程文件大小上限或内部所实施的限制。

EINVAL
: 所指定的文件描述符被映射到一个不允许写入操作的对象。

EIO
: 发生了一个低级的 I/O 错误。

ENOSPC
: 文件系统于背后所指定的文件描述符没有足够的空间可用。

EPIPE
: 所指定的文件描述符关联到读取端已经关闭的 pipe 或 socket。此时，进程还会收到 SIGPIPE 信号。SIGPIPE 信号的默认行为是终止收到信号的进程。因此，只有在进程被明确要求去忽略、阻挡或处理 SIGPIPE 信号的情况下，进程才会收到此 errno 值。

write()的字节数量限制

如果 count 参数的值大于 SSIZE_MAX，则 write() 调用的结果未定义。

调用 write() 时，如果将 count 设为 0，则此调用会立即返回 0 这个值。

write() 的行为

从 `write()` 调用返回时，内核已经将缓冲区所提供的数据复制到内核的缓冲区，但是无法保证数据已经写出到其预定的目的地。的确，写入调用返回的速度实在太快了，可能没有时间完成该项工作。处理器和硬盘之间的性能差异使得此类令人头痛的行为显而易见。

事实上，如果用户空间应用程序发出 `write()` 系统调用，Linux 内核会先进行若干检查，接着将数据复制进缓冲区。稍后，内核会在后台收集所有"脏"（有数据写入）缓冲区（内容跟相应磁盘块不同的所有缓冲区），将它们安排成最佳顺序，接着写进磁盘（此过程称为"写回"，*writeback*）。这让写入调用的执行可以快如闪电，几乎立即返回。这也让内核可以将写入操作延后到较空闲的时段再进行，并且是多笔写入操作会整批一起进行。

延后进行的写入操作并不会改变 POSIX 的语义。举例来说，数据刚写入缓冲区而尚未写回磁盘，此时如果发出读取请求，此请求可从缓冲区得到满足，而且不会因此而读取到磁盘上的旧数据。此行为会实际提高性能，因为读取请求可从内存中的缓存区得到满足，而不必从磁盘。当读取和写入请求如预期般交替出现时，结果也和预期一样 —— 也就是说，数据被写回磁盘之前系统不会崩溃！即使应用程序相信写入请求已经成功完成了，但事实上数据尚未写回磁盘。

延后写入的另一个问题是无法安排写入顺序。尽管应用程序可能会安排写入请求的顺序，好让它们能够按照特定的顺序写回磁盘，内核会以它认为合适的方式重新安排写入请求的顺序，主要是基于性能的考虑。除非系统崩溃，否则这通常不是一个问题，因为所有缓冲区最后都会写回磁盘，所以一切都很好。即使如此，绝大多数的应用程序实际上并不关心写入请求的顺序。

延迟写入必须探讨的最后一个问题是汇报 I/O 错误。写回磁盘期间可能会发生任何无法向发出写入请求的进程汇报的 I/O 错误 —— 例如磁盘驱动器故障。的确，缓冲区与这些进程毫无关系。假如有多个进程"弄脏"（将数据写入）单一缓冲区，而这些进程可能在数据写入缓冲区之后并且在数据写回磁盘之前先结束了。此外，你如何在事后与写入失败的进程通信？

内核会试图尽量降低延后写入的风险。为了确保数据可以被及时写出，内核为缓冲区设立了一个时间上限（*maximum buffer age*），而且会在时间超过上限之前写出所有"脏"（有数据写入）缓冲区。用户可通过 */proc/sys/vm/dirty_expire_centiseconds* 来设定此值，此值以厘秒（centisecond，百分之一秒）为单位。

你还可以针对特定文件缓冲区的写回操作甚至所有的写入操作进行同步。这些主题会在下一节“同步化 I/O”中加以讨论。

本章稍后会在“内核内部”一节深入探讨 Linux 内核的缓冲区写回子系统（buffer writeback subsystem）。

同步化 I/O

尽管 I/O 的同步化是一个重要的主题，但是也不应该回避延后写入的相关问题。写入操作的缓冲机制大大提升了系统的性能，因此任何操作系统都应该经缓冲机制来实现延后写入功能。然而，有些时候应用程序可能会想控制何时可以把数据写回磁盘。在此情况下，Linux 内核提供了若干选项，让你可以进行同步化操作。

fsync() 与 fdatasync()

确保数据能够写回磁盘的最简单方法就是使用 POSIX.1b 所定义的 `fsync()` 系统调用：

```
#include <unistd.h>

int fsync (int fd);
```

调用 `fsync()` 可确保文件描述符 `fd` 所映射的文件中所有脏数据会被写回磁盘。文件描述符 `fd` 必须被打开以备写入。此调用会写回数据和元数据，例如文件创建的时间戳以及 inode 中所包含的其他属性。此调用会一直等到硬盘汇报数据与元数据已经写回磁盘才返回。

在硬盘具有写入缓存区（write cache）的情况下，`fsync()` 不可能知道数据是否真的已经位于磁盘上。尽管硬盘汇报数据已经写回，但是数据实际上仍在硬盘的写入缓存区。所幸的是，硬盘缓存区里的数据应该会被立即交付给磁盘。

Linux 还提供了 `fdatasync()` 系统调用：

```
#include <unistd.h>

int fdatasync (int fd);
```

此系统调用和 `fsync()`的行为一样，但是它只会刷新数据（flush data）。此调用无法保证元数据与磁盘是同步的，因此速度可能会更快。通常这样就够了。

这两个函数的用法都一样，很简单：

```
int ret;

ret = fsync (fd);
if (ret == -1)
        /* 错误 */
```

这两个函数均无法保证包含文件的目录项如果有任何变动都能够与磁盘同步化。这意味着，如果一个文件的链接最近有所变动，则尽管文件的数据已经成功写回磁盘，但是相关的目录项或许还没有，这可能会导致无法取得文件。为了确保目录项的任何变动也能够写回磁盘，必须针对目录本身（打开目录以便取得它的文件描述符）调用 fsync()。

返回值与错误代码

执行成功时，这两个调用都会返回 0；执行失败时，这两个调用都会返回 -1 并且将 errno 的值设定为下面的其中一个：

EBADF
: 所指定的文件描述符无效或者未打开以备写入。

EINVAL
: 所指定的文件描述符被映射到一个不支持同步化机制的对象。

EIO
: 同步化期间发生了一个低级的 I/O 错误。这呈现了一个真正的 I/O 错误，而且通常是在能够捕获到此类错误的地方。

当前，调用 fsync() 或许会失败，因为背后的文件系统可能并未实现 fsync()，即使它实现了 fdatasync()。如果 fsync()返回 EINVAL，有此疑虑的应用程序可能会想尝试 fdatasync()。例如：

```
if (fsync (fd) == -1) {
        /*
         * 我们偏好 fsync()，但是如果 fsync() 失败了，
         * 让我们试试 fdatasync()。
         */
        if (errno == EINVAL) {
                if (fdatasync (fd) == -1)
                        perror ("fdatasync");
        } else
                perror ("fsync");
}
```

因为 POSIX 需要 fsync()，而 fdatasync() 只是个选项，所以任何一般的 Linux 文件系统通常应该实现 fsync() 系统调用。然而，旧的文件类型（或许没有元数据需要同步）或者奇怪的文件系统可能只会实现 fdatasync()。

sync()

虽然不是最理想的，但是应用范围较广的 sync() 系统调用可让所有缓冲区与磁盘同步化：

```
#include <unistd.h>

void sync (void);
```

此函数不需要参数，也没有返回值。它总是能成功并且返回，所有的缓冲区（包括数据和元数据）保证都会写回磁盘（注 3）。

标准并没有要求 sync() 必须等到所有缓冲区全都被清空到磁盘才返回，标准只要求该调用启动将所有缓冲区交付给磁盘的进程。基于这个原因，标准通常建议进行多次同步化，以确保所有数据都已交付给磁盘。然而，Linux 会一直等到所有缓冲区都被交付为止。因此，调用一次 sync()就够了。

sync() 的唯一应用就是实现 *sync*(8) 实用程序。应用程序应该使用 fsync() 与 fdatasync() 将必要的文件描述符的数据提交给磁盘。注意，在忙碌的系统中，sync() 可能需要花几分钟的时间才会完成。

O_SYNC 标志

将 O_SYNC 标志传递给 open() 表示文件的所有 I/O 都需要同步化：

```
int fd;

fd = open (file, O_WRONLY | O_SYNC);
if (fd == -1) {
        perror ("open");
        return -1;
}
```

读取请求通常需要同步化，否则将无法得知缓冲区中供读取的数据是否正确。然而，如先前所述，write() 调用通常不需要同步。从此调用返回与数据是否已提交给磁盘并无关系。O_SYNC 标志可用来建立此关系，确保 write() 调用会执行 I/O 的同步化。

O_SYNC 的功能是：每次 write() 调用之后以及从调用返回之前隐式（implicit）调用 fsync()。然而，Linux 内核所实现的 O_SYNC 的效率更高一点。

注 3： 如同之前所警告的：硬盘可能会欺骗内核缓冲区已经写回磁盘，然而它们实际仍位于磁盘的缓存区。

O_SYNC 会让写入操作的用户时间与内核时间(分别代表在用户空间与内核空间中所花的时间)略微变差。此外，这取决于被写入文件的大小，O_SYNC 可能会由于此过程所导致的所有 I/O 等待时间（等待 I/O 完成所花的时间）而让总耗费时间（total elapsed time）增加一个或两个数量级。这会增加巨大的成本，所以除非不得已，否则最好不要使用同步化 I/O。

通常，应用程序需要使用 fsync() 或 fdatasync() 来保证写入操作已经将数据写回磁盘。相比较于 O_SYNC，这么做的成本较低，因为它们被调用的机会比较少（例如只在某些关键操作完成之后）。

O_DSYNC 与 O_RSYNC

POSIX 定义了另外两个与同步化 I/O 相关的 open()标志：O_DSYNC 与 O_RSYNC。在 Linux 中，这两个标志被定义成 O_SYNC 的同义词，它们的行为也一样。

O_DSYNC 标志指定每次写入操作之后只有一般数据需要同步化，不包括元数据。你可以把它想成这会导致每次写入请求之后隐式调用 fdatasync()。因为 O_SYNC 提供了有利的保证，所以即使没有显式支持 O_DSYNC，也不会有功能丧失，在 O_SYNC 所提供的有利规定下，仅会有潜在的性能损失。

O_RSYNC 标志指定读取请求以及写入请求皆需要同步化。它必须从 O_SYNC 或 O_DSYNC 择一使用。如稍早所述，读取操作已经被同步化了 —— 它们不会返回，除非它们能够将数据提供给用户。O_RSYNC 标志规定读取操作的任何副作用也需要同步化。这意味着，读取操作所造成的元数据变动也必须在从调用返回之前写入磁盘。具体而言，这项规定最有可能的应用是从 read() 调用返回之前必须更新 inode 位于磁盘上的副本文件的访问时间。Linux 将 O_RSYNC 定义成和 O_SYNC 一样，虽然这么做的意义不大（O_RSYNC 与 O_SYNC 之间并不存在 O_SYNC 与 O_DSYNC 之间的关系）。在 Linux 中，目前并没有办法获得 O_RSYNC 的行为，开发者所能获得的最接近的行为是在每次 read() 调用之后调用 fdatasync()。但是需要用到此行为的场合并不多。

直接 I/O

Linux 内核，如同任何现代的操作系统内核一样，为设备与应用程序之间的缓存机制、缓冲机制以及 I/O 管理实现了一个复杂的架构层（参见本章结尾的“内核内部”一节）。性能高的应用程序可能会想绕过复杂的层次结构来进行自己的 I/O 管理。然而，实现自己的 I/O 系统通常并不值得，事实上，相比较于应用程序层所提供的工具，操作系统层

所提供的工具更有可能达到较佳的性能。然而，数据库系统通常更喜欢使用自己的缓存机制，并希望尽量减少操作系统在场的机会。

提供 O_DIRECT 标志给 open() 可指示内核尽量减少 I/O 管理在场的机会。提供此标志时，会绕过页面缓存（page cache）直接启动用户空间缓冲区与设备之间的 I/O。所有 I/O 将会同步，直到操作完成后才会返回。

执行直接 I/O 时，请求的长度、缓冲区的调整以及文件的偏移量都必须是底层设备的扇区大小（通常就是 512 字节）的整数倍。就 Linux 内核 2.6 之前的版本而言，此项规定更为严格：在 2.4 版中，一切都必须调整成文件系统逻辑块的大小（通常是 4 KB）。为了维持兼容性，应用程序应该将其调整成较大（而且可能较为不便）的逻辑块大小。

关闭文件

当程序用完文件描述符之后，它可以经 close() 系统调用取消文件描述符与相关联文件的映射关系：

```
#include <unistd.h>

int close (int fd);
```

调用 close() 可以取消已打开文件描述符 fd 的映射关系，以及让进程与相关联文件分离。然后所指定的文件描述符不再有效，于是内核可以释放它，让它再次成为后续 open() 或 creat() 调用的返回值。close() 调用会在执行成功时返回 0，以及在发生错误时返回 -1 并且为 errno 设定适当的值。它的用法很简单：

```
if (close (fd) == -1)
        perror ("close");
```

请注意，若文件已被刷新到磁盘，关闭文件的动作将毫无作用。如果应用程序想在关闭文件之前将文件提交给磁盘，那么它需要利用稍早在“同步化 I/O”一节中所讨论到的其中一个同步化选项。

但是文件的关闭会产生若干副作用。当引用到文件的最后一个已打开的文件描述符被关闭时，内核内部表示该文件的数据结构会被释放。当此数据结构被释放时，与该文件相关联的 inode 在内存中的副本就会被移除。如果内存中已无该 inode 的副本，该 inode 也会从内存中被释放（内核为了性能的原因将该 inode 放在缓存中，但是已无此需要）。如果你从磁盘解除一个文件的链接，但是在解除该文件的链接之前它仍旧维持打开的状态，则它实际上并不会被移除，除非它被关闭并且它的 inode 已从内存中被移除。因此，调用 close() 可能还会造成一个已解除链接的文件最后被实际从磁盘移除的情况。

错误值

常见的错误就是没有检查 close() 的返回值。这可能会导致错过关键错误的情况，因为与延后操作有关的错误稍晚才会出现，而 close() 会予以汇报。

失败时的 errno 值有几种可能性。除了 EBADF（给定的文件描述符无效），最重要的错误值是 EIO（指出可能发生了与实际关闭动作无关的低级 I/O 错误）。无论汇报的是何种错误，只要文件描述符是有效的，它总是会被关闭，而且相应的数据结构也会被释放。

尽管 POSIX 允许，但是 close() 绝对不会返回 EINTR。Linux 内核开发者对此有更深入的了解 —— 这不是一个聪明的实现。

使用 lseek() 查找文件位置

一般而言，I/O 是通过一个文件线性进行的，这意味着，所有需要查找文件位置的读取和写入操作会导致文件位置的变更。然而，某些应用程序需要在文件中来回跳跃。于是提供了 lseek() 系统调用，通过它可将一个文件描述符的文件位置设定为指定的值。除了改变文件的位置，它不会执行其他动作，也不会启动任何 I/O：

```
#include <sys/types.h>
#include <unistd.h>

off_t lseek (int fd, off_t pos, int origin);
```

lseek() 的行为取决于 origin 参数，此参数值可以是下面的其中之一：

SEEK_CUR

fd 的当前文件位置会被设定为它的当前值加上 pos（其值可以是负数、零或正数）。pos 的值为零时，返回的是当前的文件位置值。

SEEK_END

fd 的当前文件位置会被设定为文件的当前长度加上 pos（其值可以是负数、零或正数）。pos 的值为零时，偏移值（offset）会被设定为文件的末端。

SEEK_SET

fd 的当前文件位置会被设定为 pos。pos 的值为零时，偏移值会被设定为文件的开头。

此调用执行成功时，会返回新的文件位置；执行错误时，会返回 -1 并且将 errno 设定为适当值。

例如，你可以将 fd 的文件位置设定为 1825：

```
off_t ret;

ret = lseek (fd, (off_t) 1825, SEEK_SET);
if (ret == (off_t) -1)
        /* 错误 */
```

也可以将 fd 的文件位置设为文件的末端：

```
off_t ret;

ret = lseek (fd, 0, SEEK_END);
if (ret == (off_t) -1)
        /* 错误 */
```

因为 lseek() 会返回变更后的文件位置，所以可以通过将 SEEK_CUR 设为零的方式，使用它来找到当前的文件位置：

```
int pos;

pos = lseek (fd, 0, SEEK_CUR);
if (pos == (off_t) -1)
        /* 错误 */
else
        / 'pos' 是 fd 的当前位置 */
```

到目前为止，lseek() 最常见的用法就是找到文件的开头、找到文件的末端或者判定一个文件描述符的当前文件位置。

查找文件末端之后的位置

你可以要求 lseek() 将文件指针推进到文件末端之后的位置。例如，下面的程序代码会将指针从 fd 所映射的文件的末端推进 1,688 个字节：

```
int ret;

ret = lseek (fd, (off_t) 1688, SEEK_END);
if (ret == (off_t) -1)
        /* 错误 */
```

就其本身而言，将文件指针推进到文件末端之后的位置事实上什么也没做 —— 对刚创建的文件位置进行读取操作将返回 EOF。然而，如果随后对此位置进行写入操作，则会在文件的旧长度与新长度之间创建新的空间，而且会以零来填补它。

这个用零填补的空间称为空洞（*hole*）。在 Unix 风格的文件系统中，空洞并不占用任何物理磁盘空间。这意味着，文件系统上所有文件的尺寸加起来可能会超过磁盘的实际大

小。具有空洞的文件称为稀疏文件（*sparse file*）。稀疏文件可以节省非常可观的空间并增强性能，因为操作空洞并不需要启动任何实体的 I/O。

读取一个文件的空洞部分时会返回适当数量的二进制零值。

错误值

发生错误时，`lseek()` 会返回 `-1` 并且将 `errno` 设定为下面的其中一个值：

`EBADF`
: 指定的文件描述符并未引用一个已打开的文件描述符。

`EINVAL`
: 为 `origin` 指定的值并非 `SEEK_SET`、`SEEK_CUR`、`SEEK_END` 其中之一，或者由此找到的文件位置为负数。事实上，使用 `EINVAL` 来表示这两种错误并不合适。前者几乎就是一个编译时的编程错误，而后者则代表比较诡异的运行时的逻辑错误。

`EOVERFLOW`
: 所产生的文件偏移值无法使用 `off_t` 来表示。这只会发生在 32 位架构上。然而文件位置仍会被更新，此错误只是表示文件位置无法被返回而已。

`ESPIPE`
: 所指定的文件描述符被关联到一个无法查找位置的对象，例如管道（pipe）、FIFO 或 socket。

限制

文件位置的最大值受限于 `off_t` 数据类型本身的大小。多数机器架构会将此定义为 C 的 `long` 数据类型，而在 Linux 上这通常是字长（*word size*，通常就是机器的一般用途寄存器的大小）。然而，内核在内部会使用 C 的 `long long` 数据类型来存储偏移值。这并不会对 64 位机器构成问题，但是这意味着，32 位机器在查找文件位置时可能会产生 `EOVERFLOW` 错误。

针对特定位置的读取与写入

关于 `lseek()` 的使用，Linux 提供了 `read()` 和 `write()` 系统调用的两种变体，它们会以文件位置为参数分别进行读取与写入。操作完成之后，它们并不会改变文件的位置。

读取的形式称为 pread()：

```
#define _XOPEN_SOURCE 500

#include <unistd.h>

ssize_t pread (int fd, void *buf, size_t count, off_t pos);
```

此调用会从文件描述符 fd 的文件位置 pos 读取 count 个字节到 buf。

写入的形式称为 pwrite()：

```
#define _XOPEN_SOURCE 500

#include <unistd.h>

ssize_t pwrite (int fd, const void *buf, size_t count, off_t pos);
```

此调用会将 count 个字节从 buf 写入文件描述符 fd 的文件位置 pos。

这两个调用的行为与它们的 non-p（非针对特定位置的）版本几乎相同，除了它们会完全忽略当前的文件位置。它们不使用当前的位置，而使用 pos 所提供的值。此外，它们不会改变文件的位置。换言之，任何掺杂其中的 read() 和 write() 调用都可能会破坏针对特定位置的调用（positional call）所完成的工作。

这两个针对特定位置的调用只能用于可查找位置的文件描述符（seekable file descriptor）。它们所提供的语义类似于在调用 read() 或 write() 之前先调用 lseek()，但是有三点不同。第一点，这两个调用更容易使用，特别是在进行需要技巧的操作时，例如要向后或随机移动文件。第二点，它们在完成工作后不会改变文件指针。最后且最重要的一点，它们可以避免使用 lseek() 时可能造成的竞争条件。如果有多个线程共享文件描述符，当第一个线程调用 lseek() 之后，在它进行读取或写入操作之前，同一个程序中的另一个线程可能会改变文件的位置。使用 pread() 和 pwrite() 可避免此类竞争条件。

错误值

执行成功时，这两个调用会分别返回所读取或写入的字节数目。当 pread() 的返回值为零时代表 EOF；当 pwrite() 的返回值为 0 时表示该调用并未写入任何字节。发生错误时，这两个调用都会返回 -1 并且将 errno 设定为适当值。就 pread() 而言，此值可以是 read() 或 lseek() 的任何有效 errno 值；就 pwrite() 而言，此值可以是 write() 或 lseek() 的任何有效 errno 值。

截短文件

Linux 提供了两个系统调用用来截短一个文件的长度，它们是各项 POSIX 标准（在不同程度上）所定义和规定的系统调用。它们是：

```
#include <unistd.h>
#include <sys/types.h>

int ftruncate (int fd, off_t len);
```

与：

```
#include <unistd.h>
#include <sys/types.h>

int truncate (const char *path, off_t len);
```

这两个系统调用会将指定的文件截短成 len 所指定的长度。ftruncate() 系统调用可用于处理 fd 所指定的文件描述符，而且该文件描述符必须被打开以备写入。truncate() 系统调用可用于操作 path 所指定的文件，而且该文件必须是可写入的。执行成功时，这两个调用都会返回 0；发生错误时，它们都会返回 -1 并且会将 errno 设定为适当的值。

这两个系统调用的最常见用法就是将一个文件的大小删减成比当前长度短。成功返回后，文件的新长度为 len。先前存在于 len 与旧长度之间的数据会被丢弃，所以再也无法被读取请求所访问。

这两个函数可用于将一个文件"截短"成较大的尺寸，这类似于"查找文件末端之后的位置"一节所提到的查找操作与写入操作的组合。所扩增的字节会被填入零。

这两项操作都不会改变当前文件的位置。

举例来说，假设长度为 74 个字节的文件 *pirate.txt* 有下面的内容：

```
Edward Teach was a notorious English pirate.
He was nicknamed Blackbeard.
```

从同一个目录运行下面的程序：

```
#include <unistd.h>
#include <stdio.h>

int main()
{
        int ret;
```

```
        ret = truncate ("./pirate.txt", 45);
        if (ret == -1) {
                perror ("truncate");
                return -1;
        }

        return 0;
}
```

结果会产生长度为 45 个字节的有如下内容的 *pirate.txt* 文件：

```
Edward Teach was a notorious English pirate.
```

多任务式 I/O

应用程序通常需要服务一个以上的文件描述符，反复地与键盘输入（*stdin*）、进程间通信以及若干文件进行 I/O。现代的事件驱动(event-driven)图形用户界面(graphical user interface，简称 GUI）应用程序可通过其 mainloop（主循环）来处理数以百计的待解决事件（注 4）。

不借助线程，基本上就是分开服务每个文件描述符，单一进程无法在同一时间服务一个以上的文件描述符。只要每个所要服务的文件描述符随时都可以被读取或写入，就能够把多个文件描述符处理得很好。但是一遇到尚未准备好的文件描述符（举例来说，如果进行 `read()` 系统调用时数据尚未就绪)，进程将受到阻挡，不再能够服务其他的文件描述符。进程可能只会受阻几秒钟的时间，而让应用程序的效率变低并困扰用户。然而，如果该文件描述符之上一直都没有出现可用的数据，则进程就会永远受到阻挡。因为各个文件描述符的 I/O 往往都是相互关联的，例如管道，所以极有可能会出现一个文件描述符是否就绪取决于另一个文件描述符是否被服务过的情况。尤其是网络应用程序，可能会同时打开许多 socket，这可能会导致不少问题。

假设被一个与进程间通信有关的文件描述符阻挡时 *stdin* 有数据待处理。IPC 文件描述符返回数据之前，应用程序并不知道键盘输入有数据待处理 —— 但是，如果受阻挡的操作从不返回呢？

本章稍早所讨论的非阻挡式 I/O 可作为此问题的一个解决方案。使用非阻挡式 I/O，应用程序可以发出能够返回错误信息的 I/O 请求而不会受到阻挡。然而，此解决方案的效率并不高，这有两个原因。首先，进程需要不断以任意顺序发出 I/O 请求，一直等到其

注 4： 关于 mainloop，写过 GUI 应用程序的人应该很熟悉 —— 例如，GNOME 应用程序需要用到其基础链接库 *GLib* 所提供的 mainloop（主循环）。通过 mainloop，我们可以查看多个事件以及响应单一断点。

所打开的文件描述符中已经出现可以进行 I/O 者。这是一个差劲的程序设计。其次，如果程序可以休眠，效率将可提升不少，因为可以释放处理器给其他任务使用，每当有一个或多个文件描述符已就绪可进行 I/O 时，则会唤醒休眠中的任务。

接着让我们来探讨多任务式（*multiplexed*）I/O。

多任务式 I/O 让一个应用程序可以同时服务多个文件描述符，以及在其中有任何一个就绪可进行读取或写入时收到通知而不会受到阻挡。多任务式 I/O 因此成为了应用程序的支点，它的运作方式类似下面这样：

1. 多任务式 I/O：当这些文件描述符中有任何一个就绪可进行 I/O，请通知我。
2. 休眠，直到有一个或多个文件描述符准备妥当。
3. 唤醒：什么准备就绪了？
4. 处理所有就绪可进行 I/O 的文件描述符而不会受到阻挡。
5. 回到步骤 1，从头开始。

Linux 提供了三种多任务式 I/O 解决方案：*select*、*poll* 以及 *epoll* 接口。本章将说明前面两种接口，最后一种接口（属于高级 Linux 解决方案）则会在第四章加以说明。

select()

`select()` 系统调用提供了一个可用于实现多任务式同步 I/O 的机制：

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int select (int n,
            fd_set *readfds,
            fd_set *writefds,
            fd_set *exceptfds,
            struct timeval *timeout);

FD_CLR(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);
FD_SET(int fd, fd_set *set);
FD_ZERO(fd_set *set);
```

调用 `select()` 将受到阻挡，必须等到指定的文件描述符就绪可进行 I/O，或者等到一个可指定的时间限额过去。

此调用所监视的文件描述符会被划分成三个部分，每个部分用于等待不同的事件。列于 `readfds` 分组中的文件描述符用于查看是否有数据可供读取（也就是说，读取操作是

否可以完成而不会受到阻挡)。列于 writefds 分组中的文件描述符用于查看写入操作是否可以完成而不会受到阻挡。最后，位于 exceptfds 分组的文件描述符用于查看是否有异常发生，或者是否有紧急数据（out-of-band data，译注 2）可用（这些状态仅应用于 socket)。这些分组有可能是 NULL，在此情况下 select() 将无法监视相应的事件。

从此调用成功返回后，每个分组只会包含就绪可进行 I/O 的文件描述符。举例来说，如果有两个文件描述符，它们的值分别是 7 和 9，会被放在 readfds 分组。从此调用返回时，如果 7 仍然存在于该分组，代表该文件描述符可供读取而不会受到阻挡。如果 9 已不存在于该分组中，那么对它进行读取操作时或许会受到阻挡（之所以说或许，是因为数据可能会在此调用完成之后变成可供读取的。在此情况下，随后调用 select() 将返回此文件描述符（注 5)）。

第一个参数 n 等于任何分组中最高编号的文件描述符的值加 1。因此，select() 的调用者负责检查指定的文件描述符是否为最高值，以及将该值加 1 作为第一个参数。

timeout 参数是一个指向 timeval 结构的指针，该结构的定义如下所示：

```
#include <sys/time.h>

struct timeval {
        long tv_sec;         /* seconds */
        long tv_usec;        /* microseconds */
};
```

如果此参数不是 NULL，则 select() 调用将在 tv_sec 秒与 tv_usec 微秒之后返回，即使尚无任何文件描述符就绪可进行 I/O。作为代价，此结构的状态跨系统调用会变成未定义的，因此每次调用之前必须重新予以初始化。事实上，Linux 当前的版本会自动修改此参数，将它的值设定为剩余的时间。因此，如果 timeout 被设定为 5 秒，而且在某个文件描述符变为可用之前已经过去 3 秒的时间，于是从此调用返回时 tv.tv_sec 的值将是 2。

如果 timeout 的这两个值都被设为零，则此调用会立即返回，报告此调用进行的时候是否有任何事件等待处理，但是不会等待其后的任何事件。

这些文件描述符分组并不会被直接操控，而是通过一些辅助宏（helper macro）来进行

译注 2： “out-o-band data”在 TCP 中称为“urgent data”，所以可翻译成“紧急数据”。

注 5： 这是因为 select() 与 poll() 是电平触发（level-triggered）而非边缘触发（edge-triggered)。至于 epoll()（将于第四章讨论），则可以在这两种模式中操作。边缘触发操作比较简单，但是一不小心可能会漏掉 I/O 事件。

管理。这让 Unix 系统可以使用它们所希望的方式来实现这些分组。然而，多数系统会将这些分组实现成简单的位数组。FD_ZERO 可从特定的分组移除所有的文件描述符。你应该在每次调用 select() 之前调用此宏：

```
fd_set writefds;

FD_ZERO(&writefds);
```

FD_SET 可用来将一个文件描述符加入指定的分组，而 FD_CLR 可用来从所指定的分组移除一个文件描述符：

```
FD_SET(fd, &writefds);    /* 将 fd 加入分组 */
FD_CLR(fd, &writefds);    /* 哎呀，从分组移除 fd */
```

设计良好的程序代码应该永远都不必使用 FD_CLR，如果有需要，也是很少用到。

FD_ISSET 可用来测试某个文件描述符是否为特定分组的一部分。如果此文件描述符在该分组中，则返回非零的整数；否则返回 0。从 select() 调用返回后，可使用 FD_ISSET 来测试特定文件描述符是否已就绪而不会受到阻挡：

```
if (FD_ISSET(fd, &readfds))
        /* fd 可供读取不会受到阻挡! */
```

因为文件描述符是静态创建的，所以会对文件描述符的最大数目造成限制，而文件描述符本身也有最高值的限制，这两种限制可通过 FD_SETSIZE 来设定。在 Linux 中，此值为 1,024。本章稍后会讨论此限制所造成的后果。

返回值与错误代码

执行成功时，select() 会返回这三个分组中就绪可进行 I/O 的文件描述符的数目。如果指定了 timeout 参数，则返回值将会是 0。发生错误时，此调用会返回 -1 并且将 errno 设定为下面的其中一个值：

EBADF

　　分组中提供了一个无效的文件描述符。

EINTR

　　等待时捕捉到一个信号，你可以再次调用 select()。

EINVAL

　　参数 n 是负数或者给定 timeout 是无效值。

ENOMEM

　　内存不足以完成这项请求。

select() 范例

让我们来看一个范例程序，此范例平凡无奇但是功能齐全，足以说明 select() 的用法。此范例会等候 *stdin* 的输入数据 5 秒的时间。它只会检查单一文件描述符，因此它所进行的并非多任务式 I/O，但是这样可明确表示此系统调用的用法：

```
#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

#define TIMEOUT 5        /* select 的等待时间，以秒为单位 */
#define BUF_LEN 1024     /* 读取缓冲区，以字节为单位 */

int main (void)
{
        struct timeval tv;
        fd_set readfds;
        int ret;

        /* 等候 stdin 的输入数据。*/
        FD_ZERO(&readfds);
        FD_SET(STDIN_FILENO, &readfds);

        /* 等候5秒的时间。 */
        tv.tv_sec = TIMEOUT;
        tv.tv_usec = 0;



        /* 好了，开始提供服务！ */
        ret = select (STDIN_FILENO + 1,
                      &readfds,
                      NULL,
                      NULL,
                      &tv);
        if (ret == -1) {
                perror ("select");
                return 1;
        } else if (!ret) {
                printf ("%d seconds elapsed.\n", TIMEOUT);
                return 0;
        }

        /*
         * 我们的文件描述符可供读取了吗?
         * (肯定可以，因为它是我们所提供的唯一 fd，
         * 而且此调用会返回非零值，但是我们想幽自己一默。)
         */
        if (FD_ISSET(STDIN_FILENO, &readfds)) {
                char buf[BUF_LEN+1];
                int len;
```

```
                /* 保证不会遭到阻挡 */
                len = read (STDIN_FILENO, buf, BUF_LEN);
                if (len == -1) {
                        perror ("read");
                        return 1;
                }

                if (len) {
                        buf[len] = '\0';
                        printf ("read: %s\n", buf);
                }

                return 0;
        }

        fprintf (stderr, "This should not happen!\n");
        return 1;
}
```

使用 select() 具可移植性的休眠机制

相比较于亚秒级分辨率(subsecond-resolution)休眠机制，select() 以往就是各 Unix 系统上比较容易实施的休眠机制。需要具可移植性的休眠机制时，往往会使用 select()，方法是提供非 NULL 值的 timeout 参数，但是要将三个分组的参数全都设为 NULL：

```
struct timeval tv;

tv.tv_sec = 0;
tv.tv_usec = 500;

/* 休眠 500 微秒 */
select (0, NULL, NULL, NULL, &tv);
```

当然，Linux 也会为高分辨率休眠机制提供接口。我们将在第十章探讨这些接口。

pselect()

select() 系统调用是 4.2BSD 首先推出的，广受欢迎，但是 POSIX 却在 POSIX 1003.1g-2000 以及之后的 POSIX 1003.1-2001 中定义了自己的解决方案 pselect()：

```
#define _XOPEN_SOURCE 600
#include <sys/select.h>

int pselect (int n,
             fd_set *readfds,
             fd_set *writefds,
             fd_set *exceptfds,
             const struct timespec *timeout,
             const sigset_t *sigmask);
```

```
FD_CLR(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);
FD_SET(int fd, fd_set *set);
FD_ZERO(fd_set *set);
```

pselect() 与 select() 之间有三点不同：

1. pselect() 为它的 timeout 参数使用的是 timespec 结构而不是 timeval 结构。timespec 结构采用的是秒和纳秒（nanosecond，即十亿分之一秒）级设定值，而不是秒和微秒（microsecond，即百万分之一秒）级设定值，理论上而言可以提供较佳的 timeout 分辨率。然而事实上，这两种调用都只能可靠地提供微级的分辨率。

2. 调用 pselect() 的结果并不会改变 timeout 参数的值。因此，后续的调用并不需要重新对此参数进行初始化。

3. select() 系统调用并没有 sigmask 参数。就信号的处理而言，当此参数被设定为 NULL 时，pselect() 的行为和 select() 一样。

timespec 结构的定义如下所示：

```
#include <sys/time.h>

struct timespec {
        long tv_sec;            /* seconds */
        long tv_nsec;           /* nanoseconds */
};
```

之所以会将 pselect() 加入 Unix 工具箱，背后的主要动力是增加了 sigmask 参数，其目的是解决等待文件描述符与信号处理（第九章将有深入的探讨）之间的竞争条件。假设信号处理程序会设定一个全局标志，而且进程在调用 select() 之前会检查此标志。现在，假设信号是在检查之后以及调用之前抵达。这样应用程序可能无限期受到阻挡，而无法对所设定的标志作出响应。pselect() 调用可以解决这个问题，其做法是为调用 pselect() 的应用程序提供一组受阻挡的信号。受阻挡的信号不会被处理，除非它们已不再受阻挡。一旦从 pselect() 返回，内核会恢复旧的信号掩码。详细内容参见第九章。

直到 2.6.16 版内核，pselect() 的 Linux 实现才并非是系统调用，而是 *glibc* 所提供的 select() 的简单包裹函数。此包裹函数可尽量避免（但是无法完全消除）产生竞争条件的风险。如果能够引进真正的系统调用，将可完全避免竞争条件。

尽管 pselect() 做了一些改进，但是多数应用程序，无论是出于习惯还是基于可移植性，仍继续使用 select()。

poll()

poll() 系统调用是 System V 的多任务式 I/O 解决方案。它解决了 select() 的若干缺点，然而 select() 仍旧经常被用到（同样地，多半是出于习惯或者是基于可移植性）：

```
#include <sys/poll.h>

int poll (struct pollfd *fds, unsigned int nfds, int timeout);
```

不同于使用效率低下的三个基于位掩码的文件描述符分组，poll() 使用的是 nfds 个由 fds 指向的 pollfd 结构所组成的单一数组。pollfd 结构的定义如下所示：

```
#include <sys/poll.h>

struct pollfd {
        int fd;           /* 文件描述符 */
        short events;     /* 所要查看的事件 */
        short revents;    /* 返回所目击的事件 */
};
```

每个 pollfd 结构可用于指定一个要查看的文件描述符。你可以传递多个 pollfd 结构，指示 poll() 查看多个文件描述符。每个 pollfd 结构的 events 字段是该文件描述符所要查看事件的位掩码，用户可以设定此字段。而 revents 字段则是该文件描述符所目击事件的位掩码，内核会在返回时设定此字段。events 字段中所要求的事件可能会从 revents 字段返回。有效的事件如下所示：

POLLIN
: 有数据可供读取。

POLLRDNORM
: 有一般数据可供读取。

POLLRDBAND
: 有优先数据可供读取。

POLLPRI
: 有紧急数据可供读取。

POLLOUT
: 写入操作将不受阻挡。

POLLWRNORM
: 写入一般数据将不受阻挡。

POLLWRBAND
: 写入优先数据将不受阻挡。

POLLMSG

有 SIGPOLL 消息可用。

此外，revents 字段可能会返回下列事件：

POLLER

所指定的文件描述符发生错误。

POLLHUP

所指定的文件描述符发生挂起（hung up）事件。

POLLNVAL

所指定的文件描述符无效。

这些事件在 events 字段中毫无意义，因为如果适用，它们总是会被返回。与 select() 不同的是，使用 poll() 时你无须显式要求它报告异常的情况。

POLLIN | POLLPRI 等效于 select() 的读取事件，而 POLLOUT | POLLWRBAND 等效于 select() 的写入事件。POLLIN 等效于 POLLRDNORM | POLLRDBAND，而 POLLOUT 等效于 POLLWRNORM。

举例来说，如果要查看一个文件描述符的可读性和可写性，我们会将 events 字段设定为 POLLIN | POLLOUT。从调用返回后，我们会在与该文件描述符相应的结构中检查 revents 字段。如果设定了 POLLIN，代表该文件描述符可供读取而不会受到阻挡。如果设定了 POLLOUT，代表该文件描述符可供写入而不会受到阻挡。这两个标志并非互相排斥：同时设定时，意味着该文件描述符可供读写而不会受到阻挡。

timeout 参数用来指定等不到有任何就绪的 I/O 而要返回之前所需等待的时间，以毫秒为单位。此参数若为负值，表示无限等待下去；此参数若为 0，表示立即返回以及列出已就绪的可进行 I/O 的任何文件描述符，但是不会等待任何进一步的事件。在此情况下，poll() 的行为正如其名，探询一次便立即返回。

返回值与错误代码

执行成功时，poll() 会返回有目击事件文件描述符（也就是其结构具有非零的 revents 字段）的数目；如果在有任何事件发生之前此调用因逾时而返回，则会返回 0。执行失败时，会返回 -1 并且将 errno 设定为下面的其中一个值：

EBADF

在一个或多个结构中指定了无效的文件描述符。

EFAULT

　　指向 fds 的指针指向了进程地址空间以外的地方。

EINTR

　　所请求的事件发生之前收到了一个信号。你可以重新进行此调用。

EINVAL

　　nfds 参数的设定值超过了 RLIMIT_NOFILE 的限制。

ENOMEM

　　内存不足以完成这项请求。

poll() 的范例

让我们来看一个范例程序,此程序将使用 poll() 同时检查由读取 *stdin* 以及写入 *stdout* 是否会受到阻挡:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/poll.h>

#define TIMEOUT 5        /* poll 的等待时间, 以秒为单位 */

int main (void)
{
        struct pollfd fds[2];
        int ret;

        /* 查看 stdin 的输入 */
        fds[0].fd = STDIN_FILENO;
        fds[0].events = POLLIN;

        /* 查看 stdout 是否可供写入(通常可以) */
        fds[1].fd = STDOUT_FILENO;
        fds[1].events = POLLOUT;

        /* 都设定好了, 开始提供服务! */
        ret = poll (fds, 2, TIMEOUT * 1000);
        if (ret == -1) {
                perror ("poll");
                return 1;
        }

        if (!ret) {
                printf ("%d seconds elapsed.\n", TIMEOUT);
                return 0;
        }

        if (fds[0].revents & POLLIN)
                printf ("stdin is readable\n");
```

```
        if (fds[1].revents & POLLOUT)
                printf ("stdout is writable\n");

        return 0;
}
```

运行此程序，我们会得到如下的预期结果：

```
$ ./poll
stdout is writable
```

再运行一次此程序，不过这次将一个文件重定向至标准输入，于是我们会看到如下的结果：

```
$ ./poll < ode_to_my_parrot.txt
stdin is readable
stdout is writable
```

如果我们在实际的应用程序中使用 poll()，我们将不需要在每次调用时重新设定 pollfd 结构。你可以反复传递相同的结构，如果有需要，内核会负责把 revents 字段清零。

ppoll()

如同 pselect()，Linux 也提供了 poll() 的表亲 ppoll()。然而与 pselect() 不同的是，ppoll() 是 Linux 特有的接口：

```
#define _GNU_SOURCE
#include <sys/poll.h>

int ppoll (struct pollfd *fds,
           nfds_t nfds,
           const struct timespec *timeout,
           const sigset_t *sigmask);
```

如同 pselect()，timeout 参数也是采用秒和纳秒级的设定值，而 sigmask 参数则可用来指定所要等待的一组信号。

比较 poll() 与 select()

尽管 poll() 与 select() 所进行的是相同的工作，不过 poll() 优于 select()，这主要是因为：

- poll() 并不需要用户计算并传递作为参数的最高编号的文件描述符的值加1。
- poll() 的效率优于采用最大值的文件描述符的做法。例如，你以 select() 来查看单一文件描述符的最大值为 900 —— 内核必须为每个分组检查 900 个位的设

定状态。

- select() 的文件描述符分组的大小是固定的，这导致两难的权衡：如果太小，select() 可查看的文件描述符数目就会大受限制；如果太大，就会没有效率。大型位掩码的处理很没效率，特别是在不知道位掩码设定的位是否太稀疏时（注6），使用 poll() 则可创建一个大小刚好的数组。你只要查看一个条目，而且只需传递单一结构。
- 使用 select() 会在返回时重新构建文件描述符分组，所以每个随后的调用必须重新初始化它们。poll() 系统调用则把输入（events 字段）与输出（revents 字段）分离，并允许重复使用数组而不需要变更。
- 返回后，select() 的 timeout 参数会变成未定义的。所以具可移植性的程序代码需要重新对它进行初始化。然而，若使用 pselect() 则不会遇到此问题。

但是 select() 系统调用却具有以下优点：

- select() 较具可移植性，因为有些 Unix 系统并不支持 poll()。
- select() 提供了较佳的 timeout 分辨率：可以精确到微秒。尽管 ppoll() 与 pselect() 理论上可以提供纳秒级分辨率，但实际上它们甚至无法可靠地提供微秒级分辨率。

我们将在第四章探讨优于 poll() 与 select() 的 *epoll* 接口，这是一个 Linux 特有的多任务式 I/O 解决方案。

内核内部

本节将探讨 Linux 内核实现 I/O 的方式，聚焦于内核的三个主要子系统：虚拟文件系统（*virtual filesystem*，简称 VFS）、页面缓存（*page cache*）以及页面写回（*page writeback*）。总的来说，这些子系统可以协助我们进行无缝、高效和优化的 I/O。

> 第四章将探讨第四个子系统，I/O 调度程序（*scheduler*）。

注 6：如果位掩码通常设定的位不多（sparsely populated），你可以检查构成掩码的每个字是否为零，只有在该项操作返回假值时才需要检查每个位。然而，如果位掩码中设定的位很多（densely populated），该项操作就是在浪费时间。

虚拟文件系统

虚拟文件系统（VFS），有时也称为虚拟文件切换系统（*virtual file switch*），是一个抽象的机制，它让 Linux 内核在调用文件系统函数以及操作文件系统数据的时候，不需要知道甚至关心所使用的是何种文件系统。

VFS 这种抽象机制是通过一个公共文件模型（*common file model*）来实现的，公共文件模型是 Linux 中所有文件系统的基础。经函数指针以及各种面向对象的实践（注7），公共文件模型为 Linux 内核的文件系统提供了一个必须遵循的框架。这让 VFS 可以全面进行文件系统的请求。此框架提供了挂钩来支持读取、创建链接、同步化等功能。然后每种文件系统可以注册其特有的函数，以便处理其所能进行的操作。

此做法让各种文件系统之间有了一定程度的共同性。例如，VFS 会用到的 inode、superblock 以及 directory entry（目录项）等对象。然而一个非源自 Unix 的文件系统可能会欠缺 Unix-like 的概念（例如 inode），必须予以克服。当然，它们的确克服了：Linux 支持 FAT 和 NTFS 之类的文件系统而不会有问题。

VFS 的好处很多。单一系统调用可以读取任何存储媒介上的任何操作系统，单一实用程序可以从一个文件系统复制到任何其他文件系统。所有文件系统均支持相同的概念、相同的接口以及相同的调用。这一切都能够运作 —— 而且运作得很好。

当一个应用程序进行 `read()` 系统调用时，它会经历一个有趣的旅程。C 链接库所提供的系统调用定义在编译时会被转换成适当的 trap 语句。当一个用户空间进程陷入（trapped into）内核后，控制权会通过系统调用处理程序交给 `read()` 系统调用，于是内核可以找出支持（back）所指定的文件描述符的是何种对象。然后内核会调用与背后对象相应的读取函数。对文件系统而言，此函数是文件系统程序代码的一部分，它会做它该做的事（例如，实际从文件系统读取数据）并且将数据返回给用户空间的 `read()` 调用。于是会从系统调用处理程序返回，而系统调用处理程序会将数据复制回用户空间，接着从用户空间的 `read()` 系统调用返回，然后进程会继续执行。

对系统编程人员而言，VFS 的相关细节很重要。编程人员并不需要担心文件系统的类型或者文件位于哪种存储媒介之上。使用通用的系统调用，如 `read()`、`write()` 等，就可以操作位于任何支持的文件系统以及任何支持的存储媒介上的文件。

注7：　没错，使用 C 语言。

页面缓存

页面缓存（page cache）是一块内存存储区，用来临时存放最近从磁盘文件系统上访问的数据。磁盘的访问速度相当缓慢，尤其是相对于今日的处理器速度而言。将所请求的数据暂存在内存，让内核可以从内存来满足之后对相同数据所提出的访问请求，避免反复访问磁盘。

页面缓存利用了时间局部性（*temporal locality*）的概念，这是一种引用局部性（*locality of reference*），是指一个资源被访问后，在不久的将来再次被访问的概率很高。使用内存来缓存第一次访问到的数据是值得的，因为这可以避免日后代价昂贵的磁盘访问。

查找文件系统数据时，内核会先从页面缓存开始。只有当它不存在于缓存中时，内核才会调用内存子系统从磁盘读取数据。因此，首次读取任何数据时，该数据会从磁盘传送至页面缓存并从缓存返回应用程序。如果之后再次读取该数据，该数据会直接从缓存返回。所有操作都会自动通过页面缓存进行，以确保缓存区的数据是相关的而且始终有效。

Linux 的页面缓存区的大小是可以变动的。当 I/O 操作将越来越多的数据存入内存时，页面缓存区会变得越来越大，进而消耗任何可用的内存。如果页面缓存区最后耗尽了所有可用的内存，而且出现了分配更多内存的要求，则页面缓存区会遭到修剪（*prune*），释放其最少被用到页面，以便挪出空间给“真正的”内存使用。修剪动作会无间隙地自动进行。由于大小可以变动，因此缓存区允许 Linux 使用系统中所有内存并缓存尽可能多的数据。

然而，相比较于修剪页面缓存区中经常被用到的部分，而它们很可能在下一个读取请求被重新读进内存，把很少用到的数据块交换（*swap*）到磁盘会更有意义（交换功能让内核可以将数据存储在磁盘上，因此可以使用比系统的 RAM 还大的内存空间（memory footprint））。Linux 内核实现了试探性算法以便让数据的交换与页面缓存（以及保留在内存中的其他数据）达到平衡。这些试探性算法可决定是否以换出数据到磁盘取代修剪页面缓存，特别是如果被换出的数据并未在使用中。

交换与缓存的平衡可经*/proc/sys/vm/swappiness* 来调整。此虚拟文件的值的范围是 0 ~ 100，默认值是 60。此值越高，意味着越偏向将页面缓存数据保存在内存中且越容易进行交换；此值越低，意味着越偏向修剪页面缓存区而不进行交换。

顺序局部性（*sequential locality*）是引用局部性的另一种形式，也就说数据往往是顺序地被引用的。利用此原则，内核还实现了页面缓存区的预读（*readahead*）功能。预读是指随着每个读取请求将额外的数据从磁盘读进页面缓存区的行为 —— 实际上，只预先读进一点点数据。当内核从磁盘读进一团数据时，还会顺便读进随后的一个或两个数据

团。一次性顺序读进大型的数据团会更有效率，因为这通常可免掉寻找磁盘的需要。此外，这样内核也可以在进程读取第一个数据团的时候满足其预读的请求。如果进程接着为随后的数据团提交了一个新的读取请求（这经常发生），则内核可交出之前先读进的数据，而不必送出磁盘的 I/O 请求。

如同页面缓存，内核也是以动态的方式来管理预读功能。如果内核注意到进程始终如一地使用以预读功能所读进的数据，则内核会扩大先读范围（readahead window），因而会预先读进越来越多的数据。预读范围可小如 16 KB ，也可大到 128 KB。反过来说，如果内核注意到预读无法提供任何有用的数据，也就是应用程序正在寻找文件而且并非进行顺序读取，则内核可以完全停用预读功能。

页面缓存的存在是透明的。系统程序设计者一般无法优化自己的程序代码，以对页面缓存存在的事实能够有更好的利用 —— 除了，或许他们无法在用户空间中实现出这样的缓存功能。一般情况下，有效率的程序代码所需要的就是善用页面缓存。另一方面，利用预读功能是有可能的。顺序文件 I/O 总是优于随机访问，虽然它并非总是可行。

页面写回

正如稍早在 "write() 的行为" 一节所提到的，内核的延后写入功能是通过缓冲区完成的。当一个进程送出写入请求时，数据会被复制到一个缓冲区，该缓冲区会被标记成"已被改变"(*dirty*)，这意味着内存中的副本比磁盘上的副本新。接着就会从写入请求返回。如果有另一个写入请求针对相同文件的同一个数据团，则缓冲区会被新数据所改变。如果写入请求是针对相同文件的其他地方，则会产生新的缓冲区。

最后被改变的缓冲区需要提交给磁盘，让内存中的数据可以和磁盘上的文件同步。这就是所谓的"写回"(writeback)。两种情况下都会这么做：

- 若可用内存空间已经缩减到低于可设定的阈值，则被改变的缓冲区会被写回磁盘，所以数据已经出清到磁盘的缓冲区可以被移除，释放内存空间。
- 若被改变的缓冲区的保存时间已经超过可设定的阈值，则该缓冲区会被写回磁盘。这样可避免数据永久维持在被改变的状态。

写回操作是通过一群名为 *pdflush*（大概是 *page dirty flush* 的缩写，但是谁知道呢）的内核线程进行的。当这两种情况有一个符合时，pdflush 线程会被唤醒，开始将被改变的缓冲区提交给磁盘，直到这两种情况均不符合。

同一时间可能会有多个 pdflush 线程实际进行写回操作。可以利用并行运算以及实现拥塞避免（*congestion avoidance*）机制来完成。拥塞避免机制可用于避免因为等待数据写

入块设备而造成这些写回操作淤塞。如果这些被改变的缓冲区均来自不同的块设备，则各个 pdflush 线程将可充分使用每个块设备。这改正了之前内核版本的缺失：pdflush 线程的前身（*bdflush*，单一线程）将所有时间花在等待单一块设备上，而其他块设备却在一旁闲置着。在现代机器上，Linux 内核有大量的磁盘设备可用。

缓冲区在内核内部会被表示成 `buffer_head` 数据结构。此数据结构中记录了与缓冲区有关的各种元数据，例如缓冲区是否已经被改变。其中还包含了一个指针，用于指向实际的数据。此数据就放在页面缓存区，于是缓冲区子系统与页面缓存可以被整合在一起。

在 Linux 内核早期的版本（2.4 版之前）中，缓冲区子系统与页面缓存区是分开的，因此会同时用到页面缓存以及缓冲区缓存。这意味着数据可能同时存在于缓冲区缓存（一个被改变的缓冲区）以及页面缓存（被缓存的数据）中。当然，要让两个独立的缓存达到同步化必须费一些工夫。从 2.4 版 Linux 内核开始统一使用页面缓存是一个受欢迎的改进。

Linux 中的延后写入功能以及缓冲区子系统提供了快速写入的能力，不过代价是电力中断时会有数据漏失的风险。为了避免此风险，有此疑虑以及关键的应用程序可以采用同步化 I/O （本章稍早讨论过）。

结束语

本章介绍了 Linux 系统编程的基础知识：文件 I/O。Linux 之类的系统会尽可能将一切表示成文件，所以知道如何打开、读取、写入以及关闭文件非常重要。这些都是典型的 Unix 操作，并被包含在许多标准中。

下一章将介绍缓冲式 I/O 以及标准 C 链接库的标准 I/O 接口。标准 C 链接库不仅仅是方便使用，在用户空间中，缓冲式 I/O 为性能提供了重大的改进。

第三章

缓冲式 I/O

第一章曾提到块（*block*）这个文件系统的抽象概念，它是 I/O（输入/输出）的通用语（*lingua franca*）—— 所有的磁盘操作都会按照块来进行。因此，若所送出的 I/O 要求，其对齐块（block-aligned）边界为实际块大小的整数倍，则可以优化 I/O 的性能。

读取操作需要进行的系统调用次数增加时会导致性能变差，例如，读取一个字节 1,024 次，而不是一次读取 1,024 个字节。块边界若非实际块大小的整数倍，则就算所进行的一系列操作采用的是大于一个实际块的尺寸，也只能获得次优的性能。举例来说，如果实际块大小为 1 千字节（kilobyte，译注 1），则操作大小为 1,130 字节的块时，其速度仍可能比操作大小为 1,024 字节的块时还慢。

用户缓冲式 I/O

必须对常规文件发出许多小型 I/O 请求的程序往往会执行用户缓冲式（*user-buffered*）I/O。这是指在用户空间中进行缓冲，可能是由应用程序手动完成，或是在链接库中自动完成，而不是由内核完成。如第二章所述，基于性能的考虑，内核内部会替延后写入、联合相邻 I/O 请求以及提前读取等操作实现缓冲机制。尽管方式不同，用户缓冲式 I/O 的目的也是提高性能。

下面是一个用户空间程序 *dd* 的执行范例：

```
dd bs=1 count=2097152 if=/dev/zero of=pirate
```

由于指定了 `bs=1` 自变量，这道命令会将 2,097,152 个大小为 1 字节的块（共 2 兆字节，译注 2）从设备 */dev/zero*（这是一个可以源源不断提供 0 值的虚拟设备）复制到

译注 1：1 千字节（kilobyte）指的是 1,024 个字节而不是 1,000 个字节。

译注 2：2 兆字节（megabyte）指的是 2,097,152 个字节而不是 2,000,000 个字节。

文件 *pirate*。也就是说，数据的复制大约经过 2 百万次的读写操作（一次一个字节）。

下面的例子仍旧是进行 2 兆字节的复制，不过这次使用大小为 1,024 字节的块：

```
dd bs=1024 count=2048 if=/dev/zero of=pirate
```

上面这条命令同样是复制 2 兆字节到相同的文件，但这次只进行了 2,048 次的读写操作。这对性能有非常大的改进，如表 3-1 所示。表 3-1 中记录了以大小不同的块执行 *dd* 命令所耗费的时间：实际时间（总执行时间）、用户时间（进程在用户空间中所花费的时间）以及系统时间（在内核空间替进程执行系统调用所花费的时间）。

表 3-1：块大小对性能的影响

块大小	实际时间	用户时间	系统时间
1 字节	18.707 秒	1.118 秒	17.549 秒
1,024 字节	0.025 秒	0.002 秒	0.023 秒
1,130 字节	0.035 秒	0.002 秒	0.027 秒

相比较于使用大小为 1 字节的块，使用大小为 1,024 字节的块的性能会有非常大的改进。然而，表 3-1 同时也说明了使用较大的块尺寸时（这意味着需要用到的系统调用会比较少），如果操作并非以磁盘块大小的倍数执行，会使得性能变差。尽管需要用到的调用比较少，但是 1,130 字节的请求最后会产生非对齐的请求，因此效率不如 1,024 字节的请求。

要得利于此性能提升的特性，需要事先具备实际块大小之类的知识。如表 3-1 所示，实际块大小最可能是 1,024、1,024 的整数倍或 1,024 的因子。例如，*/dev/zero* 的实际块大小是 4,096 个字节。

块大小

事实上，实际块大小通常是 512、1,024、2,048 或 4,096 个字节。

如表 3-1 所示，性能要获得最大的提升，只需要操作的数据块是实际块大小的整数倍或是实际块大小的因子。那是因为块是内核和硬件的通用语。所以，采用实际块大小或是可以对齐实际块的值，保证能够进行对齐块的 I/O 请求以及避免内核在内部进行非必要的工作。

为了知道特定设备的实际块大小，只需要使用 `stat()` 系统调用（第七章会说明）或 *stat*(1) 命令。然而，事实上，你通常不需要知道实际的块大小。

替你的 I/O 操作挑选块大小的主要目的是不要选到奇怪的大小，例如 1,130。为你的操作使用此类大小的块，会在首次送出请求之后产生非对齐的 I/O。然而，选用实际块大小的整数倍或因子就可以避免非对齐的请求。只要你所选用的大小保持与块对齐，性能就不会变差。较大的尺寸只会导致系统调用变少的结果。

因此，进行 I/O 时最简单的选择就是使用大尺寸的缓冲区，而且其尺寸是实际块大小的倍数。例如 4,096 字节和 8,192 字节就可以运作得很好。

当然，问题在于程序很少会用到块。程序一般只处理字段、行以及字符，而不会处理块之类的对象。如稍早所述，要改变这一情况，程序可以使用用户缓冲式 I/O。当数据被写出时，它会被存放在位于程序地址空间的缓冲区。当缓冲区到达特定大小（缓冲区尺寸，*buffer size*）整个缓冲区的内容会被一次写出到磁盘。同样地，读取数据时，应该使用大小为缓冲区尺寸的对齐块的大块。当应用程序发出大小奇特的（odd-sized）读取请求时，缓冲区里的块会被一块一块地交出来。最后，当缓冲区变空时，另一个大型的对齐块的大块会被读进缓冲区。如果缓冲区的大小是正确的，则可以让性能获得显著的提升。

你可以在自己的程序中实现用户缓冲机制。的确，许多重要的应用程序都会这么做。然而，绝大多数的程序会使用广受欢迎的标准 *I/O* 链接库（标准 C 链接库的一部分）所提供的稳定且有效的用户缓冲方案。

标准 I/O

标准 C 链接库所提供的标准 I/O 链接库（常简写为 *stdio*）会提供一个与平台无关的用户缓冲方案。标准 I/O 链接库简单易用并且功能强大。

不同于 FORTRAN 之类的程序语言，C 语言不会包含任何内置支持或关键字（keyword）来提供比流程控制、算术等更高级的功能 —— 当然也不会支持 I/O。随着 C 程序语言的发展，用户开发出了数个标准链接库来提供字符串操作、数学例程、时间与日期以及 I/O 之类的基础功能。随着时间的推进，这些例程趋于成熟，并于 1989 年成为 ANSI C 标准（C89），它们最终正式成为标准的 C 链接库。虽然 C95 与 C99 加入了若干新接口，但标准 I/O 链接库自从于 1989 年创建以来就未曾改变过。

本章其余的部分将讨论用户缓冲式 I/O，因为它涉及文件 I/O 并且实现于标准 C 链接库 —— 也就是经标准 C 链接库打开、关闭、读取以及写入文件。开发者必须仔细权衡应用程序的需求和行为才有办法决定应用程序将使用标准 I/O、自制的用户缓冲方案或直接进行系统调用。

C 标准总是将一些细节留给每一个实现，而且实现往往会加入额外的功能。如同本书其余部分，本章将说明现代 Linux 系统中 *glibc* 所实现的接口和行为。值得注意的是 Linux 源自基础标准。

文件指针

标准 I/O 例程不会直接操作文件描述符，而会使用它们自己的标识符，称为“文件指针”（*file pointer*）。在C 链接库之内，文件指针会映射至一个文件描述符。文件指针会被表示成一个指向 typedef 所定义的 `FILE` 结构的指针（定义于 `<stdio.h>`）。

在标准 I/O 用语中，一个已打开的文件称为“流”（*stream*）。流可能会被打开以供读取（输入流）、写入（输出流）或读写（输入／输出流）。

打开文件

要打开文件以备通过 `fopen()` 读取或写入：

```
#include <stdio.h>

FILE * fopen (const char *path, const char *mode);
```

此函数会根据指定的模式打开文件 `path`，并将它关联到一个新的流。

模式

`mode` 参数用来描述如何打开所指定的文件。它的值可能是下面的其中一个字符串：

`r`
: 打开文件以备读取。流位于文件开端。

`r+`
: 打开文件以备读取和写入。流位于文件开端。

`w`
: 打开文件以备写入。如果文件已存在，它会被截短成零长度；如果文件不存在，则会被创建。流位于文件开端。

`w+`
: 打开文件以备读取和写入。如果文件已存在，它会被截短成零长度；如果文件不存在，则会被创建。流位于文件开端。

a

打开文件以备写入并且进入附加模式。如果文件不存在，则会创建。流位于文件末端。所写入的数据会被附加到文件末端。

a+

打开文件以备读取和写入并且进入附加模式。如果文件不存在，则会创建。流位于文件末端。所写入的数据会被附加到文件末端。

> mode 参数中还可以包含字符 b，虽然在 Linux 上总是会忽略此值。有些操作系统会以不同方式来处理文本文件和二进制文件，而 b 模式可要求 fopen()以二进制模式打开文件。如同与 POSIX 兼容的所有系统一样，Linux 则会以相同的方式来处理文本文件和二进制文件。

执行成功时，fopen() 会返回一个有效的 FILE 指针；执行失败时，它会返回 NULL 并且将 errno 设定为适当的值。

例如，下面的程序代码会打开 */etc/manifest* 以备读取并将它关联到 stream：

```
FILE *stream;

stream = fopen ("/etc/manifest", "r");
if (!stream)
        /* 错误 */
```

经文件描述符打开流

函数 fdopen() 可将一个已打开的文件描述符（fd）转换成流：

```
#include <stdio.h>

FILE * fdopen (int fd, const char *mode);
```

mode 参数的可能值和 fopen()一样，而且必须兼容于最初打开文件描述符时所使用的模式。你可以指定 w 和 w+ 等模式，但是不会造成截短的结果。流位于相关联文件描述符的文件位置上。然而，这么做也是合法的。

文件描述符被转换成流后，不应该再直接以文件描述符来进行 I/O。请注意，文件描述符并未重复，而只是被关联到一个新的流。关闭流的同时也会关闭文件描述符。

执行成功时，fdopen() 会返回一个有效的文件指针；执行失败时，它会返回 NULL。

例如，下面的程序代码会由 open() 系统调用打开 */home/kidd/map.txt*，并使用支持的（backing）文件描述符创建相关联的流：

```
FILE *stream;
int fd;

fd = open ("/home/kidd/map.txt", O_RDONLY);
if (fd == -1)
        /* 错误 */

stream = fdopen (fd, "r");
if (!stream)
        /* 错误 */
```

关闭流

fclose() 函数可用于关闭指定的流：

```
#include <stdio.h>

int fclose (FILE *stream);
```

任何经缓冲而尚未写入磁盘的文件首先会被刷新（flush）至磁盘。执行成功时，fclose() 会返回 0；执行失败时，它会返回 EOF 并且将 errno 设定为适当的值。

关闭所有流

fcloseall() 函数可用于关闭关联到当前进程的所有流，包括标准输入、标准输出以及标准错误：

```
#define _GNU_SOURCE

#include <stdio.h>

int fcloseall (void);
```

关闭之前，所有流会被刷新至磁盘。此函数通常会返回 0，它是 Linux 特有的功能。

从流中读取

标准 C 链接库实现了数个函数（从常见的到罕见的都有），可用于读取一个已打开的流。本节将介绍三种最常见的读取操作：一次读取一个字符、一次读取一整行以及读取二进制数据。为了从流中读取，必须以适当的模式打开一个输入流，也就是 w 或 a 除外的任何有效的模式。

一次读取一个字符

很多时候，理想的 I/O 操作方式就是一次读取一个字符。函数 fgetc() 可用于从一个流中读取一个字符：

```
#include <stdio.h>

int fgetc (FILE *stream);
```

此函数会从 stream 读取下一个字符，把它从 unsigned char 转型成 int 并返回。转型的目的是提供足够的空间给 end-of-file 或错误的通知：返回值为 EOF 就是这种情况。fgetc() 的返回值必须被存放在 int 变量中。将它存放在 char 变量中是一种常见但危险的错误。

下面的范例程序代码会从 stream 读取一个字符，检查错误，然后以 char 数据类型输出结果：

```
int c;

c = fgetc (stream);
if (c == EOF)
        /* 错误 */
else
        printf ("c=%c\n", (char) c);
```

由 stream 所指向的流必须被打开以备读取。

将字符放回

标准 I/O 提供了一个函数来将一个字符推回流，让你可以“偷看一下”流，如果它不是你想要的字符，你可以把它还回去：

```
#include <stdio.h>

int ungetc (int c, FILE *stream);
```

调用此函数就可以把 c（会被转型成 unsigned char）推回 stream。执行成功时会返回 c；执行失败时会返回 EOF。随后读取 stream 会返回 c。如果有多个字符被推回，则它们会以相反的顺序被返回 —— 也就是最后被推回的字符会首先被返回。POSIX 规定只有单个推回操作可以保证执行成功而不会干扰到读取请求。有些实现只允许单个推回操作；Linux 则允许无限多个推回操作，只要有内存可供使用。当然，单个推回操作通常会执行成功。

调用 ungetc() 之后，如果在发出读取请求之前调用了一个查找函数（参见“查找流”一节），将使得被推回的所有字符被丢弃。此情况会出现在一个进程的各个线程之间，因为这些线程会共享缓冲区。

读取一整行

函数 fgets() 可用于从所指定的流读取一个字符串：

```
#include <stdio.h>

char * fgets (char *str, int size, FILE *stream);
```

此函数会从 stream 读取 size-1 个字节并且将结果存入 str。读进这些字节后，缓冲区会被存进一个 null 字符（\0）。读到一个 EOF 或一个 newline 字符后，便会停止读取的动作。如果读到一个 newline 字符，则 \n 会被存入 str。

执行成功时会返回 str；执行失败时会返回 NULL。

例如：

```
char buf[LINE_MAX];

if (!fgets (buf, LINE_MAX, stream))
        /* 错误 */
```

POSIX 将 LINE_MAX 定义于 <limits.h>：它是 POSIX 行操作接口所能处理的输入行的最大尺寸。Linux 的 C 链接库并无此类限制 —— 输入行可以是任何尺寸。具可移植性的程序可以使用 LINE_MAX 以保安全，在 Linux 上它会被设定为相对较大的值。Linux 特有的程序并不需要担心输入行的尺寸限制。

读取任意字符串

尽管 fgets() 所进行的基于行的读取操作是有用的，但是也经常造成困扰。有时候，开发者可能会想使用 newline 以外的定界符（delimiter）；有时候，开发者可能不想使用任何定界符 —— 而且很少有开发者会想将定界符存入缓冲区！事实上，在所返回的缓冲区中存入 newline 字符极少会出现正确的结果。

将 fgets() 换成 fgetc() 并不难。例如，下面这段程序代码可以从 stream 将 n-1 个字节读进 str，然后附加一个 \0 字符：

```
char *s;
int c;

s = str;
while (--n > 0 && (c = fgetc (stream)) != EOF)
```

```
        *s++ = c;
*s = '\0';
```

这段程序代码还可以被扩展成遇到 d 所指定的定界符（在此例中，它不可以是一个 null 字符）时停止读取：

```
char *s;
int c = 0;

s = str;
while (--n > 0 && (c = fgetc (stream)) != EOF && (*s++ = c) != d)
        ;

if (c == d)
        *--s = '\0';
else
        *s = '\0';
```

将 d 设定成 \n 可提供与 fgets() 相似的行为，会减去存入缓冲区的 newline 字符。

视 fgets() 的实现而定，此变化可能会让程序的运行速度变慢，因为它会重复调用 fgetc()。然而，此问题不同于前面的 *dd* 范例所呈现的问题！尽管这段程序代码会带来额外的函数调用开销，但是不会导致以 bs=1 执行 *dd* 所带来的系统调用的开销以及非对齐 I/O 的沉重负担。后者会造成比较大的问题。

读取二进制数据

对于某些应用程序而言，只是读取个别的字符或输入行是不够的。有时候，开发者会想读写复杂的二进制数据，例如 C 的结构。为此，标准 I/O 链接库提供了 fread()：

```
#include <stdio.h>

size_t fread (void *buf, size_t size, size_t nr, FILE *stream);
```

fread() 可从 stream 将数据读取到 buf 所指向的缓冲区，所读取的字节数目由 size*nr 来决定（数据中共有 nr 个元素，每个元素具有 size 个字节）。文件指针会前进实际所读取到字节数。

执行成功时，fread() 会返回所读取到的元素数目（而不是字节数目）。执行失败时，它会返回一个小于 nr 的值，指出发生了错误或 EOF（读取到了文件末端）。可惜，若不使用 ferror() 和 feof()（参见“错误与 EOF”），则无法知道所发生的是这两种情况的哪一种。

因为变量大小、对齐、补白以及字节顺序的不同，一个应用程序所写入的二进制数据未必可以被不同的应用程序所读取，或者未必可以被不同机器上的相同应用程序所读取。

对齐问题

所有机器架构都会有数据对齐的需求。程序设计者往往会把内存想成一个由字节所构成的数组。然而，我们的处理器对内存进行读写操作时并不会使用字节大小的块（byte-sized chunk）。事实上，处理器会使用特定的粒度（granularity），例如以 2、4、8 或 16 字节的块来访问内存。由于每个进程的地址空间均从地址 0 开始，进程必须从其值为粒度整数倍的地址开始访问。

因此，C 的变量必须存储至或读取自对齐的地址。一般而言，变量会自然对齐，这是指对齐相应的 C 数据类型的大小。例如，一个 32 位大小的整数会对齐一个 4 字节大小的边界。换言之，一个 int 值将被存储在可以被 4 整除的内存地址上。

访问未对齐的数据将付出各种代价，这取决于机器的架构。有些处理器可以访问非对齐的数据，但是会在性能上付出很大的代价。有些处理器无法访问未对齐的数据，试图这么做会造成硬件异常事件。更糟的是，有些处理器会默默丢弃低位以迫使地址被对齐，这一定会产生令人意想不到的行为。

一般情况下，编译器会自然对齐所有数据，程序设计者并不知道数据是否对齐。处理数据结构、手动进行内存管理、存储二进制数据到磁盘以及通过网络通信均会把对齐问题带到台面上来。因此，系统程序设计者应该要熟悉此类议题！

第八章会更深入地探讨对齐议题。

fread() 最简单的例子就是从指定的流读进只有一个元素的线性字节：

```
char buf[64];
size_t nr;

nr = fread (buf, sizeof(buf), 1, stream);
if (nr == 0)
        /* error */
```

介绍 fread() 的反向操作 fwrite() 时，我们会看到更复杂的例子。

写入一个流

和读取操作一样，标准 C 链接库定义了许多函数用于写入一个已打开的流。本节将介绍三种最常见的写入操作：写入单一字符、写入一个字符串以及写入二进制数据。要写入至一个流，必须以适当的模式打开一个输出流，也就是 r 除外的任何有效的模式。

写入单一字符

fgetc() 的反向操作是 fputc()：

```
#include <stdio.h>

int fputc (int c, FILE *stream);
```

fputc() 函数会将参数 c（转型成 unsigned char）写入 stream 所指向的流。执行成功时，它会返回 c；执行失败时，它会返回 EOF 并且为 errno 设定适当的值。

fputc() 的用法很简单：

```
if (fputc ('p', stream) == EOF)
        /* 错误 */
```

此例会将字符 p 写入 stream（此流必须被打开以备写入）。

写入一个字符串

函数 fputs() 可用于将一整个字符串写入所指定的流：

```
#include <stdio.h>

int fputs (const char *str, FILE *stream);
```

fputs() 会将参数 str 所指向的以 null 分隔的字符串全都写入参数 stream 所指向的流。执行成功时，fputs() 会返回一个非负值；执行失败时，它会返回 EOF。

下面的例子会以附加模式打开文件以备写入，接着将指定的字符串写入与文件相关联的流，然后关闭流：

```
FILE *stream;

stream = fopen ("journal.txt", "a");
if (!stream)
        /* 错误 */

if (fputs ("The ship is made of wood.\n", stream) == EOF)
        /* 错误 */

if (fclose (stream) == EOF)
        /* 错误 */
```

写入二进制数据

当程序需要写入复杂的数据结构时，使用个别的字符以及输入行是无法妥善处理的。要直接存储二进制数据，例如 C 变量，可以使用标准 I/O 链接库所提供的 fwrite()：

```
#include <stdio.h>

size_t fwrite (void *buf,
               size_t size,
               size_t nr,
               FILE *stream);
```

fwrite() 可将参数 buf 所指向的数据写入参数 stream 所指向的流，所写入的字节数目由 size*nr 来决定（数据中共有 nr 个元素，每个元素有 size 个字节）。文件指针会前进实际所写入的字节数目。

执行成功时，fwrite() 会返回所写入的元素数目（而不是字节数目）；执行失败时，它会返回一个小于 nr 的值，指出发生了错误。

使用缓冲式 I/O 的简单程序

现在让我们来看一个例子 —— 事实上这是一个完整的程序，它整合进了本章迄今为止所介绍的许多接口。此程序首先会定义 struct pirate，接着以该类型声明两个变量。此程序会初始化其中一个变量，随后将它写入磁盘（也就是经输出流写入文件 *data*）。然后此程序会经不同的流从 *data* 文件将数据直接读回 struct pirate 的另一个实例。最后此程序会将该结构的内容写入标准输出：

```
#include <stdio.h>

int main (void)
{
        FILE *in, *out;
        struct pirate {
                char          name[100]; /* 真实姓名 */
                unsigned long booty;     /* 以英镑为单位 */
                unsigned int  beard_len; /* 以英寸为单位 */
        } p, blackbeard = { "Edward Teach", 950, 48 };

        out = fopen ("data", "w");
        if (!out) {
                perror ("fopen");
                return 1;
        }

        if (!fwrite (&blackbeard, sizeof (struct pirate), 1, out)) {
                perror ("fwrite");
```

```
                return 1;
        }

        if (fclose (out)) {
                perror ("fclose");
                return 1;
        }

        in = fopen ("data", "r");
        if (!in) {
                perror ("fopen");
                return 1;
        }

        if (!fread (&p, sizeof (struct pirate), 1, in)) {
                perror ("fread");
                return 1;
        }

        if (fclose (in)) {
                perror ("fclose");
                return 1;
        }

        printf ("name=\"%s\" booty=%lu beard_len=%u\n",
                p.name, p.booty, p.beard_len);

        return 0;
}
```

当然，其输出的是一开始所设定的值：

```
name="Edward Teach" booty=950 beard_len=48
```

此外，请牢记，因为变量大小、对齐等差异，一个应用程序所写入的二进制数据未必可以被其他应用程序读取。也就是说，不同的应用程序甚至是不同机器上的相同应用程序，未必能够正确读回 `fwrite()` 所写入的数据。在我们的例子中，应该考虑 `unsigned long` 是否会有所不同，或者补白的数量是否会有种种变化。只要你是在使用特定 ABI 的特定类型机器上运行应用程序，这些事情保证都会维持不变。

查找一个流

操纵当前的流位置往往会很有用。或许是应用程序正在读取一个基于记录的复杂文件，需要来回跳跃；亦或许是流需要被重新设成文件位置零。无论是何种情况，标准 I/O 链接库都提供了一系列功能相当于 `lseek()` 系统调用的接口。例如，`fseek()` 函数（最常见的标准 I/O 查找接口）会根据 `offset` 与 `whence` 来操纵 `stream` 的文件位置：

```
#include <stdio.h>

int fseek (FILE *stream, long offset, int whence);
```

如果 whence 的值被设为 SEEK_SET，则文件位置会被设成 offset；如果 whence 的值被设为 SEEK_CUR，则文件位置会被设成当前位置加上 offset；如果 whence 的值被设为 SEEK_END，则文件位置会被设成文件末端加上 offset。

执行成功时，fseek() 会返回 0，清除 EOF 指示器并取消 ungetc() 所造成的影响（如果有的话）；发生错误时，它会返回 -1 并且将 errno 设定为适当的值。最常见的错误是无效的流（EBADF）以及无效的 whence 参数（EINVAL）。

此外，标准 I/O 链接库还提供了 fsetpos()：

```
#include <stdio.h>

int fsetpos (FILE *stream, fpos_t *pos);
```

此函数会将 stream 的流位置设成 pos。它的功能如同将 whence 参数设为 SEEK_SET 的 fseek()。执行成功时，它会返回 0；否则，它会返回 -1 并且将 errno 设定成适当的值。之所以提供此函数（以及它的反向操作 fgetpos()，稍后会提到），只是因为其他（非 Unix 的）平台具有复杂的数据类型可以表示流位置。在其他平台上，如果 C 的 long 数据类型不够使用，此函数是将流位置设成任意值的唯一方法。如果想让你的 Linux 应用程序移植到所有可能的平台上，就不应该让它们使用此接口，尽管它们可以这么做。

标准 I/O 链接库还提供了 rewind() 以作为一个捷径：

```
#include <stdio.h>

void rewind (FILE *stream);
```

如下的调用：

```
rewind (stream);
```

会将位置重新设定成流的开头。除了还会清除错误指示器，它的功能如同：

```
fseek (stream, 0, SEEK_SET);
```

注意，rewind() 没有返回值，因此无法直接传达错误的情况。想确定发生错误的调用者，调用此函数之前应该先清除 errno，以便事后检查此变量是否为非零值。例如：

```
errno = 0;
rewind (stream);
if (errno)
        /* 错误 */
```

取得当前流位置

不同于 lseek()，fseek() 并不会返回更新过的位置。为此，另外提供了一个接口。ftell() 函数可用于返回 stream 的当前流位置：

```
#include <stdio.h>

long ftell (FILE *stream);
```

发生错误时，它会返回 -1 并且将 errno 设定成适当的值。

此外，标准 I/O 链接库还提供了 fgetpos()：

```
#include <stdioh.h>

int fgetpos (FILE *stream, fpos_t *pos);
```

执行成功时，fgetpos() 会返回 0 并且将 stream 的当前流位置放于 pos；执行失败时，它会返回 -1 并且将 errno 设定成适当的值。如同 fsetpos()，之所以会提供 fgetpos()，只是因为非 Linux 平台具有复杂的文件位置数据类型。

刷新一个流

标准 I/O 链接库提供了一个接口，可用于将用户缓冲区写出到内核，以确保经write() 写入至流的所有数据可以被刷新。fflush() 函数提供了此功能：

```
#include <stdio.h>

int fflush (FILE *stream);
```

调用此函数时，stream 所指向的流中的任何未被写至内核的数据会被刷新至内核。如果 stream 为 NULL，则进程中所打开的所有输入流均会被刷新。执行成功时，fflush() 会返回 0；执行失败时，它会返回 EOF 并且将 errno 设定成适当的值。

要了解 fflush() 所造成的影响，你必须了解C 链接库所维护的缓冲区与内核的缓冲机制有何差异。本章所介绍的所有调用用到的由 C 链接库负责维护的缓冲区位于用户空间而非内核空间。这就是性能可以改善的原因 —— 你待在用户空间，因此运行的是用户的程序代码，不需要进行系统调用。只有在必须访问磁盘或其他存储媒介时才需要进行系统调用。

fflush() 仅会将用户缓冲式数据写出至内核缓冲区。如果不使用用户缓冲机制而直接使用 write()，效果是一样的。这并不保证数据会被实际提交给任何存储媒介 —— 对于这种需求可以使用 fsync() 之类的系统调用（参见第二章的“同步化 I/O”）。最有

可能的做法是，你会想在调用 fflush() 之后紧接着调用 fsync()：也就是先将用户缓冲区写出至内核，再将内核的缓冲区写出至磁盘。

错误与 EOF

有些标准 I/O 接口，例如 fread()，无法有效地将执行失败的情况传达给调用者，因为它们没有提供任何机制用于区分错误与 EOF。对于这些调用以及其他场合，这可以检查特定流的状态，以确定它到底是遇到了错误还是抵达了文件末端。为此，标准 I/O 链接库提供了两个接口。函数 ferror() 用于测试 stream 之上是否设定了错误指示器：

```
include <stdio.h>

int ferror (FILE *stream);
```

错误指示器由其他的标准 I/O 接口所设定，用于反映错误状态。如果设定了指示器，则此函数会返回一个非零值，否则返回 0。

函数 feof() 用于测试 stream 之上是否设定了 EOF 指示器：

```
include <stdio.h>

int feof (FILE *stream);
```

EOF 指示器由其他的标准 I/O 接口所设定，用于反映抵达了文件的末端。如果设定了指示器，则此函数会返回一个非零值，否则返回 0。

函数 clearerr() 用于清除 stream 之上的错误与 EOF 指示器：

```
#include <stdio.h>

void clearerr (FILE *stream);
```

此函数没有返回值，而且也不会失败（无法知道所提供的流是否有效）。你应该在检查错误与 EOF 指示器之后调用 clearerr()，因为你无法在事后挽回被清除的指示器。例如：

```
/* 'f' 是个有效的流 */

if (ferror (f))
        printf ("Error on f!\n");

if (feof (f))
        printf ("EOF on f!\n");

clearerr (f);
```

取得相应的文件描述符

有时，取得支持指定流的文件描述符会很有用。例如，要对流进行系统调用时，如果不存在相关联的标准 I/O 函数，则可以通过它的文件描述符。要取得支持（backing）流的文件描述符，可使用 fileno()：

```
#include <stdio.h>

int fileno (FILE *stream);
```

执行成功时，fileno() 会返回与 stream 相关联的文件描述符；执行失败时，它会返回 -1。这只会发生在指定了无效的流时，在此情况下，fileno() 会把 errno 设定成 EBADF。

标准 I/O 调用与系统调用混用时通常不会看到警告信息。使用 fileno() 时，程序设计者一定要谨慎为之，以确保所进行的是适当的行为。尤其是在操纵支持的文件描述符前先刷新流可能是明智之举。你绝对不应该将实际的 I/O 操作混合在一起。

控制与缓冲机制

标准的 I/O 链接库实现了三种用户缓冲机制，并且提供了一个接口让开发者可以控制缓冲区的类型和大小。不同类型的用户缓冲机制用于不同的目的，适合不同的情况。下面是三种选项：

未经缓冲（*Unbuffered*）

  未执行用户缓冲机制。数据会被直接提交给内核。由于这是执行用户缓冲机制的对立面，所以不常被用到。标准错误在默认情况下是未经缓冲的。

经行缓冲（*Line-buffered*）

  所执行的缓冲机制以行为基础。每遇到一个 newline 字符，缓冲区就会被提交给内核。行缓冲机制适用于输出至屏幕的流。因此，终端机会默认使用此缓冲机制（标准输出在默认情况下是经行缓冲的）。

经块缓冲（*Block-buffered*）

  所执行的缓冲机制以块为基础。本章一开始所探讨的就是这种类型的缓冲机制，适用于文件。默认情况下，与文件相关的所有流都是经块缓冲的。标准 I/O 会使用 *full buffering*（全缓冲）这个术语来代表 block buffering（块缓冲）。

通常，默认的缓冲类型就是正确和理想的缓冲机制。然而标准 I/O 链接库提供了一个接口，用于控制所使用的缓冲类型：

```
#include <stdio.h>

int setvbuf (FILE *stream, char *buf, int mode, size_t size);
```

setvbuf() 函数会将 stream 的缓冲类型设定成 mode，其值必须是下面的其中一个：

_IONBF

　　未经缓冲

_IOLBF

　　经行缓冲

_IOFBF

　　经块缓冲

_IONBF（在此情况下会忽略 buf 与 size）除外，buf 会指向一个大小为 size 字节的缓冲区，而标准 I/O 将以此作为特定流的缓冲区。如果 buf 的值为 NULL，则 *glibc* 会自动分配一个缓冲区。

打开流之后必须调用 setvbuf() 函数，而且必须在对流执行任何其他操作之前进行。执行成功时，setvbuf() 会返回 0；否则，它会返回非零值。

如果提供了缓冲区，关闭流的当时它必须存在。一个常见的错误就是把缓冲区声明成一个自动变量，而在流被关闭之前已经离开自动变量的有效范围。尤其是小心不要把缓冲区设定为 main() 的局部变量，然后又未显式关闭流。例如：

```
#include <stdio.h>

int main (void)
{
        char buf[BUFSIZ];

        /* 以 BUFSIZ 大小的缓冲区将 stdout 设定成经块缓冲 */
        setvbuf (stdout, buf, _IOFBF, BUFSIZ);
        printf ("Arrr!\n");

        return 0;
}
```

修正此缺陷的方法：在离开有效范围之前显式关闭流，或者将 buf 设定为一个全局变量。

一般来说，开发者并不需要担心流上的缓冲机制。标准错误除外，终端机是经行缓冲的，这是可以理解的。文件是经块缓冲的，这也是可以理解的。块缓冲机制的默认缓冲区大小为 BUFSIZ（定义于 <stdio.h>），而且它通常是一个理想的选择（典型块大小的许多倍）。

线程安全

线程就是单一进程中的多道执行流程。将其概念化的一个方式就是将它们视为共享地址空间的多个进程。线程随时都可以运行，而且可以改写共享的数据，除非数据的访问需要注意同步问题或将它设置为 *thread-local*（线程局部变量，也就是为每个线程准备一个副本）。支持线程的操作系统会提供锁定机制（用于确保相互排斥的程序结构）以确保线程不会彼此踩到对方的脚。标准 I/O 会使用这些机制。尽管如此，它们通常不够使用。例如，有时你会想锁定一群调用，将临界区（*critical region*、执行时不会被另一个线程所干扰的程序代码段）从一个 I/O 操作扩展到多个。在其他情况下，你可能想彻底取消锁定机制以提高效率（注 1）。本节中，我们将讨论如何完成这两件事。

标准 I/O 函数本来就具备线程安全（*thread-safe*）。在内部，它们会关联到一个锁（lock）以及一个锁定计数（lock count），而且每个已打开流会被关联到一个拥有线程（owning thread）。任何线程在发出任何 I/O 请求之前必须先取得"锁"，成为拥有线程。两个或两个以上的线程操作相同的流时不能干扰标准 I/O 操作，因此在单一函数调用的执行环境内，标准 I/O 操作是不可分割的。

当然，实际使用时，许多应用程序所需要的原子性（atomicity）大于个别函数调用的层次。举例来说，如果有多个线程发出写入请求，虽然个别的写入操作不会相互干扰以及产生混乱的输出，但是应用程序可能会希望所有写入请求都能够完成而不会中断。为了允许这么做，标准 I/O 链接库提供了一系列的函数，可用于个别操作与流相关的锁定。

手动文件锁定

函数 `flockfile()` 会等到 `stream` 不再被锁定后取得流的锁，递增锁定计数，成为流的拥有线程，然后返回：

```
#include <stdio.h>

void flockfile (FILE *stream);
```

函数 `funlockfile()` 用于递减与 `stream` 相关联的锁定计数：

```
#include <stdio.h>

void funlockfile (FILE *stream);
```

如果锁定计数变为零，当前线程会放弃流的所有权，另一个线程则可以取得该流的锁。

注 1：　一般情况下，取消锁定机制会导致各种问题。但有些程序可能会显式实现自己的线程用法，将所有 I/O 委派给单一线程。在此情况下，可以免除锁定机制的开销。

这些调用可以嵌套。也就是说，单一线程可以多次调用 flockfile()，但是直到进程进行相同次数的 funlockfile() 调用，流才会被解除锁定状态。

ftrylockfile() 函数是 flockfile() 的不受阻挡的版本：

```
#include <stdio.h>

int ftrylockfile (FILE *stream);
```

如果 stream 目前已被锁定，则 ftrylockfile() 什么事也不会做，而会立即返回一个非零值。如果 stream 目前未被锁定，则它会取得流的锁，递增锁定计数，成为 stream 的拥有者，然后返回 0。

让我们来看一个例子：

```
flockfile (stream);

fputs ("List of treasure:\n", stream);
fputs ("    (1) 500 gold coins\n", stream);
fputs ("    (2) Wonderfully ornate dishware\n", stream);

funlockfile (stream);
```

虽然个别的 fputs() 操作绝不可能造成竞争条件 —— 例如，最后的结果不会有“List of treasure”与其他文字交叉（interleaving）在一起的现象，但是对这个相同的流而言，若它们来自另一个线程的另一个标准 I/O 操作，则可能会出现两个 fputs() 调用交叉的现象。在理想的情况下，一个应用程序的设计不会让多个线程提交 I/O 给相同的流。然而如果你的应用程序需要这么做，那么你需要范围大于单一函数的不可分割区，在此情况下，flockfile() 以及相关函数可以节省你的时间。

不会锁定的流操作

之所以要对流进行手动锁定，还有第二个理由。粒度较细和较精确的锁定机制只有应用程序的设计者能够提供，这么做或许可以尽量减少锁定机制的开销以及改善性能。为达到这个目的，Linux 提供了一系列的函数，它们是常见的标准 I/O 接口的表亲，但是不会使用任何锁定机制。它们实际上是标准 I/O 函数的不会锁定的版本：

```
#define _GNU_SOURCE

#include <stdio.h>

int fgetc_unlocked (FILE *stream);
char *fgets_unlocked (char *str, int size, FILE *stream);
size_t fread_unlocked (void *buf, size_t size, size_t nr,
                       FILE *stream);
```

```
int fputc_unlocked (int c, FILE *stream);
int fputs_unlocked (const char *str, FILE *stream);
size_t fwrite_unlocked (void *buf, size_t size, size_t nr,
                        FILE *stream);
int fflush_unlocked (FILE *stream);
int feof_unlocked (FILE *stream);
int ferror_unlocked (FILE *stream);
int fileno_unlocked (FILE *stream);
void clearerr_unlocked (FILE *stream);
```

这些函数的行为如同它们的会锁定的表亲一样，但是它们不会检查或取得 stream 的锁。如果需要锁定，则程序设计者必须负责手动取得并释放流的锁。

虽然 POSIX 替标准 I/O 函数定义了一些不会锁定的变体，但是上面所列的函数都不是 POSIX 定义的。它们全都是 Linux 特有的函数，虽然各种其他的 Unix 系统支持其子集。

标准 I/O 的缺陷

由于标准 I/O 被广泛采用，一些专家指出了其中的缺陷。有些函数，例如 fgets()，有时会不太合用；有些函数，例如 gets()，太过可怕，应该排除在标准之外。

标准 I/O 最大的问题在于性能受到两次复制的影响。读取数据时，标准 I/O 会对内核进行 read() 系统调用，从内核复制数据到标准 I/O 缓冲区。如果有一个应用程序接着通过标准 I/O（例如使用 fgetc()）送出一个读取请求，则数据又会被复制一次，这次会从标准 I/O 缓冲区被复制到所提供的缓冲区。写入请求则逆向运作：数据从所提供的缓冲区被复制到标准 I/O 缓冲区，而后通过 write() 系统调用从标准 I/O 缓冲区被复制到内核。

另一种实现可避免两次复制，方法是由每个读取请求返回一个指针指向标准 I/O 缓冲区。于是可以直接读取标准 I/O 缓冲区里面的数据，而不需要额外的复制动作。万一应用程序想要它自己本地缓冲区里的数据（或许是写入数据），应用程序通常可以手动进行复制。这个实现提供了一个接口，允许应用程序在完成读取缓冲区中特定数据块的操作时送出通知。

写入操作稍微复杂一点，但是仍然可以避免两次复制的情况。送出写入请求时，此实现会记录指针。最后，当准备刷新数据到内核时，此实现会扫描其所存储的指针列表，以便写出数据。这项工作可通过 writev() 利用分散—聚集（scatter-gather）I/O 来完成，因此仅进行一次系统调用（我们将在下一章探讨分散—聚集 I/O）。

而高度优化的用户缓冲链接库可用于解决两次复制问题的实现，类似于我们刚才所做的讨论。此外，有些开发者选择实现自己的用户缓冲解决方案。尽管有这些替代方案，标准 I/O 仍广受欢迎。

结束语

标准 I/O 是一个用户缓冲链接库，作为标准 C 链接库的一部分被提供。如果不考虑它的几个缺点，可以说它是一种有效而且受欢迎的解决方案。许多 C 程序设计者实际上只知道标准 I/O。当然，对于终端 I/O，基于行的缓冲机制最为理想，标准 I/O 是唯一的选择。有谁曾经直接使用 `write()` 指向标准输出?

当下面有任何一项为真时，就可以使用标准 I/O —— 一般来说还包含用户缓冲机制：

- 你可以想象到多次系统调用的问题，而你想降低调用的次数来尽量减少开销。
- 性能至关重要，而你想确保所有 I/O 均在对齐块的边界上使用块大小的数据块。
- 你的访问方式以字符或行为基础，而你希望接口可以轻易使用这种访问方式，不需要进行任何不相干的系统调用。
- 相比较于低级的 Linux 系统调用，你更喜欢使用较高级的接口。

然而，当你直接使用 Linux 的系统调用时却存在最大的机动性。下一章，我们将着眼于 I/O 的高级形式以及相关的系统调用。

第四章

高级文件 I/O

在第二章中，我们看过了 Linux 的基本 I/O 系统调用。这些调用不仅是文件 I/O 的基础，也是 Linux 上所有通信的基础。在第三章中，我们看过了基本 I/O 系统调用之上是如何常常需要用到用户空间缓冲机制，而且还介绍了一个用户空间缓冲方案，也就是 C 的标准 I/O 链接库。在本章，我们将探讨 Linux 所提供的更高级的 I/O 系统调用：

分散 — 聚集 I/O（*scatter-gather I/O*）

: 让单一调用可以同时对多个缓冲区进行数据的读取或写入操作。可用于将各个数据结构的字段聚集成单次 I/O 操作。

事件轮询接口（*event poll interface*）

: 用于改善第二章所提到的 `poll()` 及 `select()` 系统调用。若单一程序中必须轮询上百个文件描述符，则可以使用事件轮询接口。

内存映射 I/O（*memory-mapped I/O*）

: 将一个文件映射至内存，让文件的 I/O 可以通过简单的内存操纵来进行。可用于某

[illegible]

可在不使用线程的情况下[illegible]

异步 I/O（*asynchron*[illegible]

: 允许进程送出 I/O [illegible]
量的 I/O 工作量。

本章结束时将讨论性能的问[illegible]

分散 — 聚集 I/O

分散 — 聚集（*scatter-gather*）*I/O* 是一种进行输入和输出的方法。通过此方法，单一系统调用可以将缓冲区向量（a vector of buffers）写入单一数据流，或者将单一数据流读取到缓冲区向量。这个类型的 I/O 之所以会有此名称，是因为数据会被分散至（*scattered into*）或聚集自（*gathered from*）特定的缓冲区向量。这种方式的输入和输出又称为向量（*vectored*）*I/O*。相比较之下，第二章的标准读取和写入系统调用所提供的是线性（*linear*）*I/O*。

比较于线性 I/O 的做法，分散 — 聚集 I/O 有几项优点：

更自然的 I/O 操作

如果你的数据原本就是分段的（segmented），例如头文件中预定义的字段，向量 I/O 可对其进行直观操作。

效率

单次向量 I/O 操作可以取代多次线性 I/O 操作。

性能

比较于线性 I/O 的实现，向量 I/O 的实现除了可以减少系统调用的次数，还可以经内部的优化提供性能的改善。

原子性

不同于多次线性 I/O 操作，一个进程可以执行单次向量 I/O 操作，不会有与另一个进程的操作交叉在一起的风险。

更自然的 I/O 操作与原子性其实不使用分散 — 聚集 I/O 机制也能办到。一个进程可以在进行写入操作之前把分开的向量连接成单一缓冲区，并在进行读取操作之后把所返回的缓冲区分解成多个向量，也就是一个用户空间应用程序可以手动执行分散和聚集的操作。然而，这种解决方案实现起来既没效率也不好玩。

readv() 与 writev()

POSIX 1003.1-2001 定义了而且 Linux 实现了一对采用分散 — 聚集 I/O 机制的系统调用。Linux 的实现品满足了前一节所列的各项目标。

`readv()` 函数会把 `count` 个段从文件描述符 `fd` 读取进 `iov` 所描述的缓冲区：

```
#include <sys/uio.h>

ssize_t readv (int fd,
               const struct iovec *iov,
               int count);
```

writev() 函数最多会把 count 个段从 iov 所描述的缓冲区写入至文件描述符 fd：

```
#include <sys/uio.h>

ssize_t writev (int fd,
                const struct iovec *iov,
                int count);
```

尽管 readv() 和 writev() 函数的行为分别如同 read() 和 write()，但是 readv() 和 writev() 会同时读取或写入多个缓冲区。

每个 iovec 结构可用来描述一个独立的缓冲区，这又称为一个段（*segment*）：

```
#include <sys/uio.h>

struct iovec {
        void *iov_base;     /* pointer to start of buffer */
        size_t iov_len;     /* size of buffer in bytes */
};
```

一组段又称为向量（*vector*）。向量里的每个段描述了内存中应该读取或写入缓冲区的地址和长度。readv() 函数在着手处理下一个缓冲区之前，会先填满当前长度 iov_len 个字节的缓冲区。writev() 函数在着手处理下一个缓冲区之前，通常会先把 iov_len 个字节全部写出。这两个函数总是会依次操作各个段，从 iov[0] 开始，然后是 iov[1] 等等，一直到 iov[count-1]。

返回值

执行成功时，readv() 和 writev() 会分别返回所读取或写入的字节数目。此数目应该是 count 个 iov_len 值的总和。发生错误时，系统调用会返回 -1 并且将 errno 设定成适当的值。这两个系统调用可以体验 read() 和 write() 系统调用可能遇到的任何错误，而且会在收到此类错误时设定相同的 errno 代码。此外，标准还定义了两种其他的错误状态。

首先，因为返回值的类型为 ssize_t，如果 count 个 iov_len 值的总和大于 SSIZE_MAX，则数据将不会被传送，此时会返回 -1 并且将 errno 设定成 EINVAL。

其次，POSIX 规定 count 必须大于零以及小于或等于 IOV_MAX，此项限制定义于 <limits.h>。在 Linux 中，IOV_MAX 目前是 1024。如果 count 为 0，这两个系统调用会返回 0（注 1）。如果 count 大于 IOV_MAX，则数据将不会被传送，此时会返回 -1 并且将 errno 设定成 EINVAL。

注 1： 注意，如果 count 为 0，其他的 Unix 系统会把 errno 设定成 EINVAL。这是相关标准明确允许的：如果该值为 0、或者系统可用其他（非错误的）方式来处理值为 0 的情况，errno 可以被设成 EINVAL。

优化 count

向量 I/O 操作期间，Linux 内核必须分配内部的数据结构以表示每个区段。一般情况下，这种分配是动态进行，而且以 count 参数的大小为基础。然而，如果 count 的值够小，不需要动态分配段，则可以进行优化，于是 Linux 内核会在它所使用的堆栈上为段创建一个小型的数组，因此性能多少会有一点改善。此优化阈值（threshold）目前是 8，所以如果 count 的值小于或等于 8，则向量 I/O 操作会在进程的内核堆栈中以非常有内存效率的方式进行。

最有可能的是你并未在指定的向量 I/O 操作中选择需要同时传送多少个段。然而，如果你是一个随机应变的人，考虑使用较小的值，那么你可以选择 8 或更小的值，这样性能必定会有所改善。

writev() 范例程序

让我们来看一个简单的范例程序，它会写出了一个具有三个段的向量，每个段包含了一个长度不同的字符串。这个完整的程序足以示范 writev() 的用法，它虽然简单但具有实用性：

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <sys/uio.h>

int main ()
{
        struct iovec iov[3];
        ssize_t nr;
        int fd, i;

        char *buf[] = {
                "The term buccaneer comes from the word boucan.\n",
                "A boucan is a wooden frame used for cooking meat.\n",
                "Buccaneer is the West Indies name for a pirate.\n" };

        fd = open ("buccaneer.txt", O_WRONLY | O_CREAT | O_TRUNC);
        if (fd == -1) {
                perror ("open");
                return 1;
        }

        /* 填入三个 iovec 结构 */
        for (i = 0; i < 3; i++) {
```

```
                iov[i].iov_base = buf[i];
                iov[i].iov_len = strlen (buf[i]);
        }

        /* 以单次调用将它们全部写出 */
        nr = writev (fd, iov, 3);
        if (nr == -1) {
                perror ("writev");
                return 1;
        }
        printf ("wrote %d bytes\n", nr);

        if (close (fd)) {
                perror ("close");
                return 1;
        }

        return 0;
}
```

运行此程序会产生预期的结果：

```
$ ./writev
wrote 148 bytes
```

读取此程序所写出的文件：

```
$ cat buccaneer.txt
The term buccaneer comes from the word boucan.
A boucan is a wooden frame used for cooking meat.
Buccaneer is the West Indies name for a pirate.
```

readv() 范例程序

现在让我们来看一个范例程序，它会使用 readv() 系统调用来读取前一节利用向量 I/O 所产生的文本文件。同样，这是一个完整的程序，它虽然简单但具有实用性：

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/uio.h>

int main ()
{
        char foo[48], bar[51], baz[49];
        struct iovec iov[3];
        ssize_t nr;
        int fd, i;

        fd = open ("buccaneer.txt", O_RDONLY);
        if (fd == -1) {
```

```
                perror ("open");
                return 1;
        }

        /* 设定我们的 iovec 结构 */
        iov[0].iov_base = foo;
        iov[0].iov_len = sizeof (foo);
        iov[1].iov_base = bar;
        iov[1].iov_len = sizeof (bar);
        iov[2].iov_base = baz;
        iov[2].iov_len = sizeof (baz);

        /* 以单次调用将数据读进这些结构 */
        nr = readv (fd, iov, 3);
        if (nr == -1) {
                perror ("readv");
                return 1;
        }

        for (i = 0; i < 3; i++)
                printf ("%d: %s", i, (char *) iov[i].iov_base);

        if (close (fd)) {
                perror ("close");
                return 1;
        }

        return 0;
}
```

运行前一节的程序之后再运行此程序会产生如下的结果：

```
$ ./readv
0: The term buccaneer comes from the word boucan.
1: A boucan is a wooden frame used for cooking meat.
2: Buccaneer is the West Indies name for a pirate.
```

实现

readv() 和 writev() 的简单实现可以在用户空间中以一个简单的循环来完成，它看起来会像下面这个样子：

```
#include <unistd.h>
#include <sys/uio.h>

ssize_t naive_writev (int fd, const struct iovec *iov, int count)
{
        ssize_t ret = 0;
        int i;

        for (i = 0; i < count; i++) {
```

```
                ssize_t nr;

                nr = write (fd, iov[i].iov_base, iov[i].iov_len);
                if (nr == -1) {
                        ret = -1;
                        break;
                }
                ret += nr;
        }

        return ret;
}
```

幸好，这不是 Linux 的实现品：Linux 将 readv() 与 writev() 实现成系统调用，并在内部执行分散 — 聚集 I/O。事实上，Linux 内核内部所有 I/O 均采用向量的方式，尽管 read() 和 write() 被实现成向量 I/O，但是向量中只有一个段。

事件轮询接口

察觉 poll() 与 select() 的限制后，2.6 版的 Linux 内核（注 2）引进了事件轮询（*event poll*，常简写为 epoll）机制。尽管比 poll() 与 select() 复杂，但 epoll 不仅解决了这两个接口的性能问题，而且还提供了若干新的功能。

对 poll() 与 select()（第二章讨论过）的每次调用需要一份所要查看的文件描述符的完整列表。然后内核必须处理列表中的每个文件描述符。当这份列表变大时 —— 它可能包含成百上千个文件描述符，每次调用所要处理的列表会变成可伸缩性（scalability）瓶颈。

epoll 避开这个问题的方法就是让事件监视器的注册（monitor registration）与实际的事件监视工作（actual monitoring）脱钩。一个系统调用用于初始化 epoll 的上下文（context），另一个系统调用用于将所要查看的文件描述符加入上下文或从上下文中移除文件描述符，第三个系统调用则实际执行事件等待。

创建一个新的 epoll 实例

epoll_create() 可用于创建一个 epoll 上下文：

```
#include <sys/epoll.h>

int epoll_create (int size)
```

执行成功时，epoll_create() 会创建一个新的 epoll 实例，并且返回一个与实例相

注 2：　epoll 是在编号 2.5.44 的开发版内核中引入的，而接口则是在 2.5.66 版定稿的。

关联的文件描述符。此文件描述符与真实的文件没有关系，它只是一个可供随后的调用使用 epoll 设备（facility）的句柄（handle）。size 参数用于提示内核要监视的文件描述符数目，此值并非最大数目。此值设定得当将可获得较佳的性能，但是不需要设定确切的数目。发生错误时，此调用会返回 -1 并且将 errno 设定成下面的其中一个值：

EINVAL

size 参数不是一个正数。

ENFILE

系统已经到达所能打开文件数目的上限。

ENOMEM

可用内存不足以完成此项操作。

典型的调用方式如下：

```
int epfd;

epfd = epoll_create (100);  /* 计划查看 ~100 fds */
if (epfd < 0)
        perror ("epoll_create");
```

轮询完成之后应该调用 close() 来销毁 epoll_create() 所返回的文件描述符。

控制 epoll

epoll_ctl() 系统调用可将文件描述符加入指定的 epoll 上下文，或者从指定的 epoll 上下文中移除文件描述符：

```
#include <sys/epoll.h>

int epoll_ctl (int epfd,
               int op,
               int fd,
               struct epoll_event *event);
```

头文件 <sys/epoll.h> 将 epoll_event 结构定义成：

```
struct epoll_event {
        __u32 events;  /* 事件 */
        union {
                void *ptr;
                int fd;
                __u32 u32;
                __u64 u64;
        } data;
};
```

一个成功的 epoll_ctl() 调用可以控制与文件描述符epfd 相关联的 epoll 实例。参数 op 用于指定如何操作与 fd 相关联的文件。event 参数用于进一步描述操作的行为。

op 参数的有效值如下所示：

EPOLL_CTL_ADD

根据 event 所定义的事件，将特定文件（关联于文件描述符 fd）上的一个监视器加入 epoll 实例（关联于 epfd）。

EPOLL_CTL_DEL

从 epoll 实例（关联于 epfd）移除特定文件（关联于文件描述符 fd）上的一个事件监视器。

EPOLL_CTL_MOD

以 event 所指定的新事件来修改 fd 上的一个现有事件监视器。

epoll_event 结构中的 events 字段用于列出在给定文件描述符上要监视哪些事件。多个事件值可以使用 bitwise-OR（按位 OR 逻辑运算）的方式组合在一起。下面是有效的事件值：

EPOLLERR

文件出现了一个错误状态。此事件通常会被监视，即使它未被指定。

EPOLLET

为文件的监视器启用边缘触发行为（参见“边缘触发事件与电平触发事件”一节）。默认行为是电平触发。

EPOLLHUP

文件出现了一个挂起（hangup）状态。此事件通常会被监视，即使它未被指定。

EPOLLIN

文件可供读取而且不会遭到阻挡。

EPOLLONESHOT

一个事件产生与被读取之后，自动停止监视相应的文件。欲重新启用监视功能，必须经 EPOLL_CTL_MOD 指定一个新的事件掩码。

EPOLLOUT

文件可供写入而且不会遭到阻挡。

EPOLLPRI

有紧急（out-of-band）数据可供读取。

epoll_event 结构中的 data 字段供用户私人使用。收到请求的事件后，此字段的内容会返回给用户。普遍的做法是将 event.data.fd 设定成 fd，让用户可以轻易找出是哪个文件描述符导致此事件。

执行成功时，epoll_ctl() 会返回 0；执行失败时，它会返回 -1 并且将 errno 设定成下面的其中一个值：

EBADF

op 为 EPOLL_CTL_ADD，但是 fd 已经被关联到 epfd。

EINVAL

epfd 不是一个 epoll 实例，epfd 如同 fd 或者 op 无效。

ENOENT

op 为 EPOLL_CTL_MOD 或 EPOLL_CTL_DEL，但是 fd 并未被关联到 epfd。

ENOMEM

内存不足以处理此请求。

EPERM

fd 不支持 epoll。

举例来说，如果想将特定文件（关联于 fd）上一个新的事件监视器加入 epoll 实例 epfd，可以这么做：

```
struct epoll_event event;
int ret;

event.data.fd = fd; /* 稍后将 fd 返回给我们 */
event.events = EPOLLIN | EPOLLOUT;

ret = epoll_ctl (epfd, EPOLL_CTL_ADD, fd, &event);
if (ret)
        perror ("epoll_ctl");
```

要修改 epoll 实例 epfd 中特定文件（关联于 fd）上一个现有的事件监视器，可以这么做：

```
struct epoll_event event;
int ret;

event.data.fd = fd; /* 稍后将 fd 返回给我们 */
event.events = EPOLLIN;

ret = epoll_ctl (epfd, EPOLL_CTL_MOD, fd, &event);
if (ret)
        perror ("epoll_ctl");
```

反过来说，要从 epoll 实例 epfd 中移除特定文件（关联于 fd）上一个现有的事件监视器，可以这么做：

```
struct epoll_event event;
int ret;

ret = epoll_ctl (epfd, EPOLL_CTL_DEL, fd, &event);
if (ret)
        perror ("epoll_ctl");
```

注意，当 op 为 EPOLL_CTL_DEL 时，event 参数可以是 NULL，因为没有提供事件掩码。然而在 2.6.9 版本之前的内核会错误地检查此参数是否为非 NULL 值。若要移植到这些版本较旧的内核，你应该传入一个有效的非 NULL 指针，它不会有所变动。内核

```
                    struct epoll_event *events,
                    int maxevents,
                    int timeout);
```

epoll_wait()最多可以为与特定 epoll 实例相关联的文件上的特定事件等待 timeout 毫秒的时间。执行成功时，events 会指向包含 epoll_event 结构（用于描述每个事件，最多可以包含 maxevents 个事件）的内存，返回值为事件的数目；发生错误时，返回值为 -1 并且会把 errno 设定成下面的其中一个值：

EBADF

epfd 是一个无效的文件描述符。

EFAULT

进程并未对 events 所指向的内存进行写入操作。

EINTR

系统调用完成之前被信号中断。

EINVAL

epfd 是一个无效的 epoll 实例，或者 maxevents 等于或小于 0。

如果 timeout 为 0，该调用会立即返回，即使尚未出现可用事件，在此情况下会返回 0；如果 timeout 为 -1，则该调用会等到事件可用才返回。

当此调用返回时,epoll_event 结构的 events 字段用于描述发生了哪些事件,data 字段包含了调用 epoll_ctl() 之前用户对它所做的设定。

下面是一个完整的 epoll_wait() 范例:

```
#define MAX_EVENTS    64

struct epoll_event *events;
int nr_events, i, epfd;

events = malloc (sizeof (struct epoll_event) * MAX_EVENTS);
if (!events) {
        perror ("malloc");
        return 1;
}

nr_events = epoll_wait (epfd, events, MAX_EVENTS, -1);
if (nr_events < 0) {
        perror ("epoll_wait");
        free (events);
        return 1;
}

for (i = 0; i < nr_events; i++) {
        printf ("event=%ld on fd=%d\n",
                events[i].events,
                events[i].data.fd);

        /*
         * 我们现在可以根据每个 events[i].events 来操作
         * events[i].data.fd 而不会遭到阻挡。
         */
}

free (events);
```

我们将在第八章探讨 malloc() 与 free() 函数。

边缘触发事件与电平触发事件

如果 epoll_ctl() 的 event 参数的 events 字段值被设定成 EPOLLET,则 fd 的事件监视器就是边缘触发的(*edge-triggered*),这是相对于电平触发的(*level-triggered*)而言的。

下面是生产者与消费者之间通过一个 Unix 管道通信时所发生的事件:

1. 生产者将 1 KB 的数据写入一个管道。
2. 消费者对管道调用 epoll_wait(),等待管道送来数据并因此而可供读取。

使用电平触发的事件监视器时，此调用在步骤 2 时会立即返回，这显示管道已经可供读取。使用边缘触发的事件监视器时，此调用会等到步骤 1 发生之后才返回。也就是说，即使管道在 epoll_wait() 被调用时已经可供读取，epoll_wait() 也不会返回，除非有数据被写入管道。

电平触发是默认行为。这也是 poll() 与 select() 的行为以及多数开发者所预期的行为。边缘触发的行为需要使用不同的程序设计方法，通常需要运用非阻挡 I/O 并且仔细检查 EAGAIN。

> 此处所用到的术语来自电气工程。电平触发中断（level-triggered interrupt）发生在信号线到达所指定的电平（低电平或高电平）时。边缘触发中断（edge-triggered interrupt）发生在信号线经历所指定的暂态（降缘或升缘）时。当你对事件（信号线的电平）目前的状态感兴趣时，可以使用电平触发中断；当你对事件本身（信号线的某个暂态）感兴趣时，可以使用边缘触发中断。

将文件映射至内存

标准文件 I/O 还有另一个选项，内核提供了一个接口，让应用程序可以将一个文件映射至（map into）内存，这意味着内存上的一个地址与文件中的一个字之间存在一对一的对应关系。于是程序设计者可以直接通过内存来访问文件，如同将数据存放在内存的任何其他块——甚至可以让对特定内存区域所进行的写入操作自动映射回磁盘上的文件。

POSIX.1 定义了（而且 Linux 实现了）mmap() 系统调用，以便把特定对象映射至内存。本节将探讨 mmap()，因为它与把文件映射至内存时所进行的 I/O 有关。第八章还可以看到 mmap() 的其他应用。

mmap()

调用 mmap() 可要求内核将文件描述符 fd 所代表的对象映射至内存。参数 len 代表对象中有多少个字节将被映射至内存，参数 offset 代表从对象中的何处开始映射，参数 addr 用于指向内存中所要使用的起始地址，参数 prot 用于指示访问权限，参数 flags 可用于指定额外的行为：

```
#include <sys/mman.h>

void * mmap (void *addr,
             size_t len,
             int prot,
             int flags,
             int fd,
             off_t offset);
```

addr 参数可用于建议内核，内存中何处是映射文件的理想位置。这只是一个提示，多数用户会传入 0 。此调用会返回内存中开始进行此项映射的实际地址。

prot 参数用于描述关于此项映射你所期望的内存保护方式。此参数的值可以是 PROT_NONE，代表此项映射无法被访问（这么做没有多大意义），也可以是下列各值经 bitwise OR（按位 OR 逻辑运算）的结果：

PROT_READ
: 页面可以被读取。

PROT_WRITE
: 页面可以被写入。

PROT_EXEC
: 页面可以被执行。

你所期望的内存保护方式不可以与文件的打开模式相冲突。举例来说，如果程序所打开的是只读文件，则不得将 prot 指定成 PROT_WRITE。

保护标志、架构以及安全

尽管 POSIX 定义了 4 个保护位（read、write、exec 以及 none），但有些架构仅支持这些保护标志的子集。这是常见的情况，例如，一个不会区分读取和写入行动的处理器。在该情况下，处理器可能仅具备一个“读取”标志。在这些系统上，PROT_READ 意味着 PROT_EXEC。直到现在，x86 架构还是一个这样的系统。

当然，依靠这种行为是不具可移植性的。具可移植性的程序如果打算执行映射中的程序代码，通常应该被设定成 PROT_EXEC。

相反的情况是缓冲区溢出攻击（buffer overflow attack）会流行的一个原因：即使特定的映射没有指定执行权限，但是处理器却会允许执行。

最近的 x86 处理器引进了 *NX*（no-execute）位，所以映射被指定为可读取不代表可执行。在这些较新的系统上，指定 PROT_READ 不再意味着 PROT_EXEC 也成立。

flags 参数用于描述映射的类型以及它有哪些行为。此参数的值也可以是下列各值经 bitwise OR（按位 OR 逻辑运算）的结果：

MAP_FIXED

要求 mmap() 将 addr 视为必要条件，而非仅是一个提示。如果内核无法将映射放在指定的地址上，则此调用会执行失败。如果地址和长度参数与现有的映射重叠，则重叠的页面会被新的映射抛弃和取代。因为使用这个选项需要充分了解进程地址空间，然而这是不具可移植性的，所以不鼓励使用此选项。

MAP_PRIVATE

不与其他进程共享此映射。文件的映射采用的是 copy-on-write（写入时才复制）操作方式，因而此进程在内存上所做的任何改变并不会反映在实际的文件中或其他进程的映射中。

MAP_SHARED

与映射相同文件的所有其他进程共享此映射。写入此映射等效于写入相应的文件。读取此映射将反映出其他进程所做的改变。

你必须指定 MAP_SHARED 或 MAP_PRIVATE，但是不可以同时指定。其他更高级的标志将在第八章讨论。

当你映射一个文件描述符时，文件的引用计数器（reference count）会递增。所以，映射文件之后你可以关闭文件描述符，而你的进程将仍然能够访问该文件。当你解除文件的映射（unmap the file）或者进程终止时，文件的引用计数器会递减。

下面的程序代码片段会从第一个字节开始，把 len 个字节由以 fd 支持的文件映射至一个只读的映射中：

```
void *p;

p = mmap (0, len, PROT_READ, MAP_SHARED, fd, 0);
if (p == MAP_FAILED)
        perror ("mmap");
```

从图 4-1 可以看出，将一个文件映射至一个进程的地址空间时，mmap()的参数对映射所造成的影响。

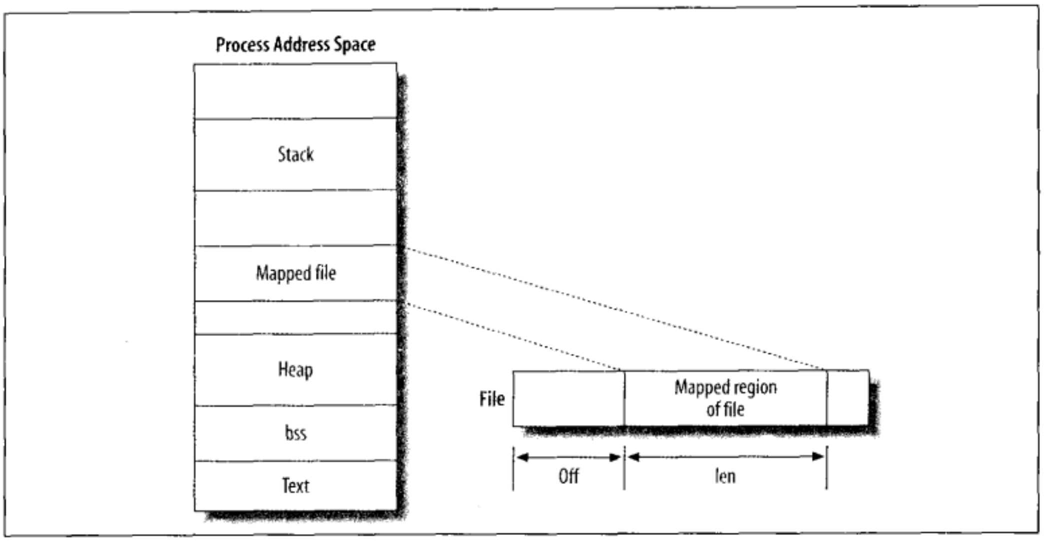


图 4-1:将一个文件映射至一个进程的地址空间

页面的大小

页面（*page*）是内存能够具有独特权限和行为的最小单位。因此，页面是内存映射的构成要素，也是进程地址空间的构成要素。

mmap() 系统调用可运作在页面之上。addr 与 offset 参数的值必须对齐一个页面大小的边界（page-sized boundary），也就是说它们必须是页面大小的整数倍。

因此，映射是页面的整数倍。如果调用者所提供的 len 参数并未对齐一个页面的边界——或许是因为文件的大小并非页面大小的倍数，则映射中不足一个页面大小的部分会用到整个页面。这个页面中的额外字节，也就是最后一个有效字节到映射末端之间，会被填入零。对该内存区域所进行的任何读取操作都将依次返回零。对该内存区域所进行的任何写入操作都不会影响支持文件（backing file），即使它在映射时指定了 MAP_SHARED。只有原本的 len 个字节会被写回文件。

sysconf()

sysconf() 是取得页面大小的标准 POSIX 方法，可用于取得系统特有的信息：

```
#include <unistd.h>

long sysconf (int name);
```

调用 sysconf() 可返回 name 的配置项（configuration item）的值，如果 name 为无效值则会返回 -1。发生错误时，此调用会将 errno 设定成 EINVAL。因为对某些项

（例如 limits，返回 -1 代表无限制）而言，-1 可能是有效值，调用之前先清除 errno 并于调用之后检查其值可能是明智之举。

POSIX 将 _SC_PAGESIZE（以及它的别名 _SC_PAGE_SIZE）定义成一个页面的大小（以字节为单位）。因此页面的大小可以轻易取得：

```
long page_size = sysconf (_SC_PAGESIZE);
```

getpagesize()

Linux 还提供了 getpagesize() 函数：

```
#include <unistd.h>

int getpagesize (void);
```

调用 getpagesize() 同样也会返回一个页面的大小（以字节为单位）。此函数的用法甚至比 sysconf() 还简单：

```
int page_size = getpagesize ();
```

并非所有 Unix 系统都支持此函数，它已从 POSIX 标准的 1003.1-2001 修订版中被移除，此处之所以会提到它是基于完整性。

PAGE_SIZE

页面的大小也被静态地存储在 PAGE_SIZE 宏（定义于 <asm/page.h>）中。因此取得页面大小的第三种方式为：

```
int page_size = PAGE_SIZE;
```

然而不同于前两种选项，此方式所取得的是编译时（而非运行时）系统的页面大小。有些架构支持使用不同页面大小的多种机器类型，而有些机器类型本身就支持多种页面大小！特定架构的二进制文件（binary）应该可以在其所支持的任何机器类型上运行，也就是说，你应该可以“构建一次，到处运行”（build it once and run it everywhere）。把页面大小写死将失去此可能性，因此你应该在运行时决定此值。因为 addr 与 offset 通常是 0，所以要符合此要求并不会太困难。

此外，未来的内核版本可能不会将此宏导出至用户空间。之所以会在本章提到它是因为它常出现在 Unix 程序代码中，但是你不应该将它使用在自己的程序里。sysconf() 是最可行的做法。

返回值与错误代码

执行成功时，mmap() 会返回映射的位置；执行失败时，它会返回 MAP_FAILED 并且将 errno 设定成适当的值。mmap() 绝对不会返回 0。

errno 的可能值如下所示：

EACCESS
: 所指定的文件描述符并非常规文件，或者它在打开时所采用的访问模式与 prot 或 flags 抵触。

EAGAIN
: 文件已经被文件锁定机制锁住了。

EBADF
: 指定了无效的文件描述符。

EINVAL
: addr、len 或 off 参数中有一个或多个是无效的。

ENFILE
: 所打开的文件已经达到了系统范围（system-wide）的限制。

ENODEV
: 要进行映射的文件所在的文件系统不支持内存映射功能。

ENOMEM
: 进程所具有的内存不足。

EOVERFLOW
: addr+len 的结果超过了地址空间的大小。

EPERM
: 虽然指定了 PROT_EXEC，但是文件系统在挂载时指定了 noexec 选项。

相关的信号

有两种信号与所映射的区域相关：

SIGBUS
: 当一个进程试图访问一个内存映射中不再有效的区域时便会产生此信号——例如，可能是因为文件在映射之后被截短了。

SIGSEGV
: 当一个进程试图写入一个以只读方式映射的区域时便会产生此信号。

munmap()

Linux 所提供的 munmap() 系统调用可用于移除 mmap() 所创建的一个内存映射：

```
#include <sys/mman.h>

int munmap (void *addr, size_t len);
```

调用 munmap() 可移除任何的内存映射。参数 addr（它的值必须与页面对齐）用来指定所要移除的映射所包含的页面始于进程地址空间中哪个位置。参数 len 用来指定要从映射中移除多少字节。一旦移除映射，先前与其相关联的内存区域不再有效，之后对其所进行的访问操作将导致 SIGSEGV 信号。

传入 munmap() 的参数一般来自先前所调用的 mmap() 的返回值以及它的 len 参数。

执行成功时，munmap() 会返回 0；执行失败时，它会返回 -1 并且将 errno 设定为适当值。唯一标准的 errno 值是 EINVAL，表示指定了一个或多个无效的参数。

下面的程序代码片段会取消"所包含的"页面位于 [addr,addr+len] 区间的内存映射：

```
if (munmap (addr, len) == -1)
        perror ("munmap");
```

内存映射范例

下面的简单范例程序会使用 mmap() 将用户所指定的一个文件输出至标准输出：

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>

int main (int argc, char *argv[])
{
        struct stat sb;
        off_t len;
        char *p;
        int fd;

        if (argc < 2) {
                fprintf (stderr, "usage: %s <file>\n", argv[0]);
                return 1;
        }

        fd = open (argv[1], O_RDONLY);
        if (fd == -1) {
                perror ("open");
```

```
                return 1;
        }

        if (fstat (fd, &sb) == -1) {
                perror ("fstat");
                return 1;
        }

        if (!S_ISREG (sb.st_mode)) {
                fprintf (stderr, "%s is not a file\n", argv[1]);
                return 1;
        }

        p = mmap (0, sb.st_size, PROT_READ, MAP_SHARED, fd, 0);
        if (p == MAP_FAILED) {
                perror ("mmap");
                return 1;
        }

        if (close (fd) == -1) {
                perror ("close");
                return 1;
        }

        for (len = 0; len < sb.st_size; len++)
                putchar (p[len]);

        if (munmap (p, sb.st_size) == -1) {
                perror ("munmap");
                return 1;
        }

        return 0;
}
```

此范例中，你唯一不熟悉的系统调用应该是 `fstat()`（第七章会有说明）。此处你只需要知道 `fstat()` 会返回所指定文件的相关信息。映射此文件前，`S_ISREG()` 宏可用于检查此信息，以便判断所指定的文件是否为常规文件（相对于设备文件或目录而言）。非常规文件被映射时的行为取决于其支持设备（backing device）。有些设备文件可用于 `mmap` 的操作，其他非常规文件则不可用于 `mmap` 的操作，而且会将 `errno` 设定成 `EACCESS`。

此范例程序其余部分应该很简单。此程序的执行需要一个文件名作为参数。此程序会打开该文件，确定它是一个常规文件，映射它，关闭它，将文件逐字节地输出至标准输出，然后从内存解除文件的映射。

mmap() 的优点

相比较于标准的 read() 和 write() 系统调用，经mmap() 来操作文件有以下优点：

- 对内存映射文件进行读取和写入操作，可避免使用 read() 或 write() 系统调用时所产生的无关副本(数据必须被复制到一个用户空间缓冲区，并从用户空间缓冲区复制回来)。
- 除了任何可能的页面失误(page fault)，对内存映射文件进行读取和写入操作不会产生任何系统调用或操作环境切换的开销。进行简单的内存操作即可。
- 当有多个进程将同一个对象映射至内存时，数据由这些进程所共享。对类型为 read-only 以及 shared 的可写入映射而言，所共享的是它们的全部；对类型为 private 的可写入映射而言，所共享的是它们尚未被“写入时复制”(copy-on-write，常简写成 COW) 的页面。
- 映射的查找仅涉及很少的指针操作。不需要使用 lseek() 系统调用。

基于这些原因，对许多应用程序而言，mmap() 是一个明智的选择。

mmap() 的缺点

使用 mmap() 时，有几点请铭记在心：

- 内存映射往往是页面大小的整数倍。因此，支持文件的大小与页面大小的整数倍的差值就浪费掉了，成为闲置空间。以小文件来说，映射中会有很大的比例被浪费掉。例如，使用 4 KB 大小的页面，一个 7 字节大小的文件，其内存映射会浪费掉 4,089 字节。
- 内存映射必须适合放入进程的地址空间。使用 32-bit 的地址空间时，数目非常大的各种大小的映射 (various-sized mappings) 将导致碎裂的地址空间，从而很难找到可用的大型的连续区域。当然，此问题极少出现在使用 64-bit 地址空间的情况下。
- 创建和维护内存映射以及内核内部相关的数据结构是要付出代价的。此开销通常会与所省去的双重副本开销相抵消，尤其是对较大型及被频繁访问的文件而言。

因此，当所映射的是大型文件 (因此所浪费的空间占整个映射相当小的比例) 或所映射文件的大小是页面大小的整数倍 (因此没有浪费任何空间)，将可从 mmap() 的优点获得最大的好处。

调整映射的大小

Linux 特有的 mremap() 系统调用可用于扩大或缩小所指定的映射的大小：

```
#define _GNU_SOURCE

#include <unistd.h>
#include <sys/mman.h>

void * mremap (void *addr, size_t old_size,
               size_t new_size, unsigned long flags);
```

mremap() 可将位于 [addr,addr+old_size) 区间的映射扩大或缩小成新的大小 new_size。在同一时间内，内核可能会移动此映射，这取决于进程地址空间中的可用空间以及 flags 的值。

> 在[addr,addr+old_size) 中，左边的“[”符号用于指定此区间的开始地址（映射包含该地址），右边的“)”符号用于指定此区间的结束地址（映射不含该地址）。这个惯例就称为区间标记法（*interval notation*）。

flags 参数可以是 0 或 MREMAP_MAYMOVE（指出如果有需要，内核可以自由移动映射，以便进行所请求的调整大小的操作）。进行扩大操作时，如果内核可以移动映射，则比较可能成功。

返回值与错误代码

执行成功时，mremap() 会返回一个指针，指向刚才调整过大小的内存映射；执行失败时，它会返回 MAP_FAILED 并且将 errno 设定成下面的其中一个值：

EAGAIN

该内存区域已被锁住，无法调整其大小。

EFAULT

所指定的范围内有些页面在进程的地址空间中并非有效页面，或是在重新映射所指定的页面时有问题发生。

EINVAL

指定了一个无效参数。

ENOMEM

所指定的范围若不移动则无法扩大（但是并未指定 MREMAP_MAYMOVE），或是进程的地址空间中可用空间不足。

MS_SYNC

指定同步化操作应该以同步的方式进行。msync() 将不会返回，除非所有页面都已写回磁盘。

你必须指定 MS_ASYNC 或 MS_SYNC，但不可同时指定。

它的用法很简单：

```
if (msync (addr, len, MS_ASYNC) == -1)
        perror ("msync");
```

此范例会以异步的方式（或十倍快的方式）将文件位于 [addr,addr+len) 的映射刷新至磁盘。

返回值与错误代码

执行成功时，msync() 会返回 0；执行失败时，它会返回 -1 并且把 errno 设定成适当的值。下面是有效的 errno 值：

EINVAL

flags 参数同时设定了 MS_SYNC 与 MS_ASYNC，或设定了这三个有效标志以外的位，或 addr 并未与页面的大小对齐。

ENOMEM

文件并未（或者有部分未）映射到所指定的内存区域。注意，当所要同步化的内存区域只有部分未映射时，Linux 仍能对该区域内任何有效的映射进行同步化操作，但是 Linux 将返回 ENOMEM，正如 POSIX 的规定。

在 Linux 内核 2.4.19 之前的版本中，msync() 会返回 EFAULT 而不是 ENOMEM。

对映射的用法提供建议

Linux 提供了一个系统调用，名为 madvise()，让进程可以为内核提供建议，指出它们打算如何使用映射。然后内核可以优化其行为以便利用映射的预定用法。尽管 Linux 内核会动态调整其行为，而且在没有明确建议的情况下通常也会提供理想的性能，但是[illegible]

调用 madvise() 可建议内核如何操作与特定内存映射（从 addr 开始，共 len 个字节）有关的页面：

```
#include <sys/mman.h>

int madvise (void *addr,
             size_t len,
             int advice);
```

如果 `len` 为 `0`，则内核会把该建议应用在从 `addr` 开始的整个映射。参数 `advice` 用于描述该建议，它可以被设定成下面的其中一个值：

`MADV_NORMAL`

应用程序并未对此内存范围提供任何建议。它应该按照一般的方式来处理。

`MADV_RANDOM`

应用程序打算以随机（非顺序）的方式来访问所指定范围中的页面。

`MADV_SEQUENTIAL`

应用程序打算以顺序的方式

在不久的将来，应用程序并不打算访问所指定范围中的页面。

内核修改其行为以响应此建议的实际方式与特定实现有关：POSIX 只规定了建议的意义，并未规定任何可能的结果。当前的 2.6 版内核响应建议（`advice` 值）的方式如下所示：

`MADV_NORMAL`

内核的行为一如往常，执行适当数量的预读操作。

`MADV_RANDOM`

内核会禁用预读功能，在每次进行实际的读取操作

预读

当 Linux 内核从磁盘读出文件时，它会执行一个称为预读（*readahead*）的优化操作。

……即返回所请求的数据。因为磁盘具有磁道缓冲区（基本上，硬盘会在内部执行自己的……预读操作），而且磁盘上的文件通常以顺序方式去存储，所以此项优化操作成本并不……高。

进行预读操作通常是有利的，但是优化结果是否理想则取决于执行预读的数量是否适……当以顺序方式访问文件时，可得利于较大的预读窗口（readahead window）。然而，……以随机方式访问文件时，可能会发现预读是一项不值得的开销。

如第二章的"内核内部"一节所述，内核会动态调整预读窗口的大小，以响应该窗口的命中率（hit rate）。命中率高，意味着可得利于较大的预读窗口；命中率低，建议缩小预读窗口。`madvise()` 系统调用让应用程序可以马上影响预读窗口的大小。

返回值与错误代码

执行成功时，`madvise()` 会返回 `0`；执行失败时，它会返回 `-1` 并且把 `errno` 设定成适当的值。下面是有效的错误代码：

`EAGAIN`
: 有个内部的内核资源（可能是内存）无法使用。进程可以再试一次。

`EBADF`
: 所指定的内存区域存在，但是并未映射到文件。

`EINVAL`
: 参数 `len` 为负值，或者 `addr` 并未与页面大小对齐，或者设定了无效的 `advice` 参数，或者页面已被锁定或以 `MADV_DONTNEED` 的方式共享。

`EIO`
: 指定了 `MADV_WILLNEED`，但是发生了内部的 I/O 错误。

ENOMEM

所指定的内存区域在此进程的地址空间中并非有效的映射，或者指定了 MADV_WILLNEED，但是在所指定的区域中没有足够的内存供页面使用。

对一般文件 I/O 的用法提供建议

前一小节，我们研究了如何对内存映射的使用提供建议。这一节，我们将研究如何建议内核有关一般文件 I/O 的使用方式。Linux 为这个建议机制提供了两个接口：posix_fadvise() 与 readahead()。

posix_fadvise() 系统调用

第一个建议接口正如其名，是通过 POSIX 1003.1-2003 标准化：

```
#include <fcntl.h>

int posix_fadvise (int fd,
                   off_t offset,
                   off_t len,
                   int advice);
```

调用 posix_fadvise() 可以把我们对文件描述符 fd 位于 [offset,offset+len) 范围内的建议提供给内核。如果 len 为 0，建议将会应用至 [offset,length of file] 范围的内存。常见的用法是将 len 与 offset 指定为 0，则建议会应用至整个文件。

可用的 advice 选项与 madvise() 的类似。参数 advice 用于描述该建议，它可以被设定成下面的其中一个值：

POSIX_FADV_NORMAL

应用程序对所指定的文件范围并未提供任何建议。它应该按照一般的方式来处理。

POSIX_FADV_RANDOM

应用程序打算以随机（非顺序）的方式来访问所指定范围中的页面。

POSIX_FADV_SEQUENTIAL

应用程序打算以顺序的方式（从低地址到高地址）来访问所指定范围中的页面。

POSIX_FADV_WILLNEED

在不久的将来，应用程序打算访问所指定范围中的页面。

POSIX_FADV_NOREUSE

在不久的将来，应用程序打算访问所指定范围中的页面，但是只进行一次。

POSIX_FADV_DONTNEED

在不久的将来，应用程序并不打算访问所指定范围中的页面。

如同 madvise()，对所提供建议的实际响应与特定实现有关 —— 甚至不同版本的 Linux 内核的反应也会有所不同。下面是当前内核版本所作的响应：

POSIX_FADV_NORMAL

内核的行为一如往常，执行适当数量的预读操作。

POSIX_FADV_RANDOM

内核会禁用预读功能，在每次进行实际的读取操作时只会读进最少量的数据。

POSIX_FADV_SEQUENTIALP

内核会执行积极的预读操作，将预读窗口（readahead window）的大小加倍。

POSIX_FADV_WILLNEED

内核会发起预读操作，开始将所指定的页面读取至内存。

POSIX_FADV_NOREUSE

当前的行为如同 POSIX_FADV_WILLNEED 的，未来的内核版本会进行额外的优化，以利用“只进行一次”的行为。在 madvise() 中并无相应的建议。

POSIX_FADV_DONTNEED

内核会在所指定的范围中从页面缓存区（page cache）收回被缓存的任何数据。注意，此建议不同于其他建议，它的行为也不同于 madvise() 中相应的建议。

下面的范例程序代码会建议内核以随机（非顺序）的方式访问文件描述符 fd 所代表的整个文件：

```
int ret;

ret = posix_fadvise (fd, 0, 0, POSIX_FADV_RANDOM);
if (ret == -1)
        perror ("posix_fadvise");
```

返回值与错误代码

执行成功时，posix_fadvise() 会返回 0；执行失败时，它会返回 -1 并且将 errno 设定成下面的其中一个值：

EBADF

指定了无效的文件描述符。

EINVAL

指定了无效的建议，或者所指定的文件描述符引用了一个管道（pipe），或者所指定的建议无法应用至所指定的文件。

readahead() 系统调用

posix_fadvise() 系统调用是 2.6 版的 Linux 内核新引进的。在此之前，readahead() 系统调用可用于提供与 POSIX_FADV_WILLNEED 建议一致的行为。与 posix_fadvise() 不同的是，readahead() 是 Linux 特有的接口：

```
#include <fcntl.h>

ssize_t readahead (int fd,
                   off64_t offset,
                   size_t count);
```

调用 readahead() 可以把文件描述符 fd 于 [offset,offset+count) 范围内的数据填入页面缓存区。

返回值与错误代码

执行成功时，readahead() 会返回 0；执行失败时，它会返回 -1 并且将 errno 设定成下面的其中一个值：

EBADF

指定了无效的文件描述符。

EINVAL

所指定的文件描述符并未映射至一个支持预读功能的文件。

建议是廉价的

有些常见的应用程序工作量轻易就能受益于对内核的善意建议。此类建议对 I/O 负荷的减轻有很大的帮助。由于硬盘速度如此之慢，而现代的处理器如此之快，每一个小协助与好建议都会有莫大的帮助。

读取文件的一个数据块之前，进程可以提供 POSIX_FADV_WILLNEED 建议，以指示内核将文件读取到页面缓存区。I/O 将在后台以异步的方式进行。当应用程序最后访问文件时，I/O 操作可以顺利完成而不会遭到阻挡。

反过来说，读取或写入许多数据后 —— 例如有连续的视频流被送往磁盘，进程可以提供 POSIX_FADV_DONTNEED 提示，以指示内核从页面缓存区收回文件的特定数据块。

一个大型的流操作会把数据不断填入页面缓存区。如果应用程序不打算再次访问这些数据，这意味着页面缓存区将充满多余的数据，而且会牺牲其他可能有用的数据。因此，对视频流应用程序而言，明智的做法是定期请求内核从缓存区中逐出流数据。

想读取一整个文件的进程可以提供 POSIX_FADV_SEQUENTIAL 建议，以指示内核执行积极的预读操作。反过来说，一个进程若知道它将要随机访问一个文件，来回查找数据，则可以提供 POSIX_FADV_RANDOM 建议，以指示内核预读操作只不过是毫无价值的开销。

同步化、同步及异步操作

Unix 系统大量地使用"同步化"(synchronized)、"异步化"(nonsynchronized)、"同步"(synchronous)以及"异步"(asynchronous)等令人混淆的术语 —— 就英文而言，"synchronous"与"synchronized"的差异并不大！

同步（*synchronous*）写入操作会等到所写入的数据（至少）被存入内核的缓冲缓存区(buffer cache）后才返回。同步读取操作会等到所读取的数据被存入应用程序所提供的用户空间缓冲区（user-space buffer）后才返回。另一方面，异步（asynchronous）写入操作会在数据离开用户空间之前先返回；异步读取操作会在有数据可供读取之前先返回。也就是说，这些操作只是先被排入队列待稍后再进行。当然，在此情况下，必须存在某个机制以便判断这些操作是否真的完成了以及实际的结果。

相较于同步操作，同步化（*synchronized*）操作有较多的限制，但是更为安全。同步化写入操作会将数据刷新至磁盘，以确保磁盘上的数据与相应的内核缓冲区始终是同步的。同步化读取操作总是会返回数据的最新副本，而且可假设它来自磁盘。

总之，"同步"与"异步"等术语用于指出 I/O 操作在返回之前是否需要等待某个事件(例如数据的存储)，而"同步化"与"异步化"等术语则用于指定必须发生何种事件（例如将数据写入磁盘)。

一般情况下，Unix 的写入操作是同步进行但异步化的，而读取操作则是同步进行且同步化的（注 3)。对写入操作而言，这些特点的每个组合都是可能的，如表 4-1 所示。

注 3: 读取操作从技术上来说也是异步化的，就像写入操作那样，但是内核会确保页面缓存区包含最新的数据。也就是说，页面缓存区的数据总是跟磁盘上的数据一样或是比磁盘上的数据还新。因此，其行为实际上始终是同步化的。

表 4-1：写入操作的同步状态

	同步化	异步化
同步	写入操作会等到数据被刷新至磁盘后才返回。如果文件打开时指定了 O_SYNC，便会表现此行为	写入操作会等到数据被存入内核缓冲区后才返回。这是通常的行为模式
异步	写入操作会在写入请求被排入队列后返回。等到该写入操作执行时，数据保证会被写入磁盘	写入操作会在写入请求被排入队列后返回。等到该写入操作实际执行时，数据保证至少会被存入内核缓冲区

读取操作始终是同步化的，因为读取旧数据毫无意义。然而如表 4-2 所示，此类操作可以是同步的，也可以是异步的。

表 4-2：读取操作的同步状态

	同步化
同步	读取操作会在最新的数据被存入所提供的缓冲区后返回（这是通常的行为模式）
异步	读取操作会在读取请求被排入队列后返回，但是当读取操作实际执行时会返回最新的数据

在第二章中，我们讨论过如何对写入操作进行同步化（通过 O_SYNC 标志），以及如何确保所有的 I/O 在特定的点上已被同步化（通过 fsync() 及相关的函数）。现在，让我们来看看如何进行异步的读取与写入操作。

异步 I/O

执行异步 I/O 需要内核在非常低的层次有所支持。幸好 Linux 实现了 POSIX 1003.1-2003 所定义的 *aio* 接口。*aio* 链接库提供了一系列函数以提交异步 I/O 并在它完成时可以收到通知：

```
#include <aio.h>

/* asynchronous I/O control block */
struct aiocb {
        int aio_filedes;              /* file descriptor */
        int aio_lio_opcode;          /* operation to perform */
        int aio_reqprio;             /* request priority offset */
        volatile void *aio_buf;      /* pointer to buffer */
        size_t aio_nbytes;           /* length of operation */
        struct sigevent aio_sigevent; /* signal number and value */
```

```
        /* internal, private members follow... */
};

int aio_read (struct aiocb *aiocbp);
int aio_write (struct aiocb *aiocbp);
int aio_error (const struct aiocb *aiocbp);
int aio_return (struct aiocb *aiocbp);
int aio_cancel (int fd, struct aiocb *aiocbp);
int aio_fsync (int op, struct aiocb *aiocbp);
int aio_suspend (const struct aiocb * const cblist[],
                 int n,
                 const struct timespec *timeout);
```

基于线程的异步 I/O

Linux 仅会对以 O_DIRECT 标志打开的文件支持 *aio*。要对并非以 O_DIRECT 标志打开的常规文件执行异步，我们必须向内寻求自己的解决方案。没有内核的支持，我们只得期望能够近似异步 I/O，让结果类似真实的情况。

首先，让我们来看看为何应用程序开发者想要执行异步 I/O：

- 执行 I/O 操作时不想遭到阻挡
- 将 I/O 排入队列、提交 I/O 给内核、接收操作完成通知等动作分开

第一点关系到性能。如果 I/O 操作不会遭阻挡，则 I/O 的开销就会为零，而进程也不需要成为 I/O-bound（输入/输出密集型）。第二点关系到过程（procedure），它只是处理 I/O 的另一种方法。

完成以上目标的最常见方法就是使用线程（调度事宜在第五章和第六章有详细的探讨）。此方法包括以下程序设计任务：

1. 创建一个"工作者线程"（worker thread）池以便处理所有 I/O。
2. 实现一组接口以便将 I/O 操作放入一个工作队列。
3. 这些接口必须返回独一无二的 I/O 描述符以便识别相应的 I/O 操作。在每个工作者线程中，从队列的头部取出（grab）I/O请求并且提交它们，等待它们完成 I/O 操作。
4. 完成 I/O 操作后，将操作的结果（返回值、错误代码、所读取的任何数据）放入一个结果队列中。
5. 实现一组接口以便从结果队列取回状态信息，使用最初所返回的 I/O 描述符来识别每项操作。

这样便能提供类似 POSIX 的 *aio* 接口的行为，但是在线程管理方面会有比较大的开销。

I/O 调度程序与 I/O 性能

在现代系统中，磁盘与系统其他组件之间的相对性能差距相当大 —— 并且不断扩大。磁盘性能的瓶颈出现在将读 / 写头（read/write head）从磁盘的一个部分移往另一个部分的过程，这就是众所周知的查找（*seek*）操作。在许多操作皆以若干处理器周期（每个处理周期很可能只有 1/3 纳秒）衡量的世界中，单一磁盘查找操作平均消耗的时间可能就会超过 8 毫秒 —— 尽管这仍是一个小数目，但却是单一处理器周期（*processor cycle*）的 500 万倍以上！

由于磁盘驱动器与系统其他组件之间的性能相差悬殊，若按照发出 I/O 请求的顺序将 I/O 请求传送给磁盘，效率将会非常差。因此现代操作系统内核会实现 *I/O* 调度程序（*scheduler*），通过调整服务 I/O 请求的顺序以及时间，尽量减少磁盘查找的次数和大小。I/O 调度程序会尽其所能地降低磁盘访问对系统性能所造成的影响。

磁盘寻址

要了解 I/O 调度程序的作用，需要具备一些背景知识。硬盘采用基于几何（geometry-based）的磁柱（cylinder）、磁头（head）和扇区（sector）寻址法，或称为 *CHS* 寻址法，来寻址其数据。一块硬盘由多个盘片（*platter*）组成，每个盘片由一个磁盘（disk）、磁轴（spindle）以及读 / 写头组成。你可以将每个盘片想成是一片光盘（或唱片），而硬盘中的一组盘片可想成是一叠光盘。如同光盘，每个盘片可划分成多个圆环状的磁道（*track*）。每个磁道又可划分成整数个扇区。

为了定位硬盘上特定的数据单元，磁盘驱动器的逻辑需要三种信息：磁柱、磁头以及扇区的值。磁柱值用于指定数据被摆在哪个磁道上。如果你将一个盘片放在另一个盘片之上，则所指定的磁道就会通过每个盘片形成一个磁柱。换言之，一个磁柱由每个磁盘上与轴心等距离的磁道组成。磁头值用于指出数据位于哪个读 / 写头（因此也指出了数据位于哪个盘片）。查找的范围现在缩小到了单一盘片上的单一磁道。扇区值用于指出数据位于该磁道上的哪个扇区。现在数据的查找工作完成了：硬盘知道要到哪个盘片、哪个磁道以及哪个扇区找数据。它可以把正确盘片的读 / 写头放在正确的磁道上，以便读取或写入到所需要的扇区。

值得庆幸的是，现代的硬盘并不强迫计算机使用磁柱、磁头和扇区来跟它沟通。现代的硬盘会将独一无二的块编号（*block number*，也称为“物理块”或者“设备块”）映射至每个 cylinder/head/sector 的三重组合 —— 有效地将一个块映射至一个特定的扇区。于是现代操作系统可以使用这些块编号来寻址硬盘 —— 此过程称为逻辑块寻址（*logical* *block addressing*，常简写为 *LBA*）—— 而硬盘内部会将块编号转换成正确的 CHS 地址（注 4）。块与 CHS 的映射关系往往是顺序的，但是不保证一定是这样：硬盘上的物理块 n 往往就物理地毗邻逻辑块 n + 1。这种顺序映射关系很重要，我们很快就会看到。

另一方面，文件系统仅存在于软件中。其所操作的是自己的数据单元，称为逻辑块（*logic block*，有时称为"文件系统块"，或者仅称为"块"）。逻辑块的大小必须是物理块大小的整数倍。换言之，文件系统的逻辑块会映射至硬盘上的一个或多个物理块。

调度程序的生命周期

I/O 调度程序会执行两项基本操作：合并与排序。合并是取得两个或多个邻近的 I/O 请求并且将它们组合成单一请求的过程。假设有两个请求，一个请求用于读取磁盘块（disk block）5，另一个请求用于读取磁盘块 6 ～ 7。这些请求可以合并成读取磁盘块 5 ～ 7 的单一请求。尽管 I/O 的总数据量不变，但是 I/O 操作的次数却少了一半。

排序是这两项操作中较为重要的一个。排序是按照递增的块顺序安排未决（pending）I/O 请求的过程。例如，现在有三个针对块 52、109 和 7 的 I/O 操作请求，I/O 调度程序会把这些请求安排成 7、52 和 109 的顺序。如果后来又送出了针对块 81 的请求，则 I/O 调度程序就会把它插在块 52 和 109 之间。于是 I/O 调度程序会按照未决请求在队列中的顺序（先是 7，然后是 52，然后是 81，最后是 109）派送请求给磁盘。

这样可以尽量减少磁头的移动。避免盲目地移动磁头 —— 从这里到那里并且反向移动，查找整个磁盘，应该以平顺、线性的方式移动磁头。因为查找是磁盘 I/O 中代价最高的操作，所以这样做性能可以得到改善。

对读取操作的帮助

每个读取请求必须返回最新的数据。因此，如果被请求的数据并未出现在页面缓存区中，读取进程必须遭到阻挡，直到可从磁盘读取数据为止 —— 这可能是一个冗长的操作。我们将这种性能影响称为读取延迟（*read latency*）。

一个典型的应用程序可能会在短时间内发起数个读取请求。因为每个请求是单独同步化的，所以后面的请求是否能完成取决于前面的请求完成与否。现在假设要读取一个目录中的每个文件。应用程序会打开第一个文件，读取它的一个数据块，等待数据的到来，

注 4：　此块编号的绝对大小（absolute size）的限额多年来主要负责总硬盘大小（total drive sizes）上的各种限额。

读取另一个数据块等等，直到读完整个文件。然后应用程序会对下一个文件重复以上操作。这些请求会变成序列化的（serialized）：除非完成当前的请求，否则无法送出随后的请求。

这与写入请求形成鲜明的对比，写入请求（默认为异步化的状态）不需要发起任何磁盘 I/O 直到未来的某个时间。因此，从用户空间应用程序的观点来看，写入请求流不会妨碍磁盘的性能。写入流的行为只会让读取流的问题加剧：因为写入流会把持着内核和磁盘的关注不放。此现象称为"写入饿死读取"（*writes-starving-reads*）问题。

如果一个 I/O 调度程序总是以插入的顺序来存储新的请求，结果可能会饿死针对遥远块的请求。承前例，如果在 50 秒内又连续送出针对各个块的新请求，则针对块 109 的请求可能远永都不会被服务到。因为读取延迟至关重要，所以这个行为严重伤害了系统性能。因此，I/O 调度程序采用了一个可以避免饿死的机制。

一个简单的方法 —— 例如 2.4 版 Linux 内核的 I/O 调度程序，称为 *Linus Elevator*（莱纳斯电梯）（注 5），所采取的做法就是在队列中出现待得很久的请求时，便停止插入排序（insertion-sorting）的动作。公平对待每个请求对整体性能将有所帮助，就读取请求而言，这将可改善延迟时间。问题是这种探试性机制（heuristic）似乎把事情想得太简单了。有鉴于此，2.6 版 Linux 内核淘汰了 Linus Elevator 并且公布了数种新的 I/O 调度程序。

Deadline I/O 调度程序

Deadline（期限）I/O 调度程序可解决 2.4 版内核的 I/O 调度程序以及传统电梯算法的一般问题。Linus Elevator 维护了一份经排序的未决 I/O 请求列表。位于队列头部的 I/O 请求是下一个要被服务的对象。Deadline I/O 调度程序保留了此队列，但是为了让问题有所改善，额外引进了两个队列：读取请求 *FIFO* 队列以及写入请求 *FIFO* 队列。这两个队列中的元素皆按照提交时间排序过（有利于先入先出的操作）。读取请求 FIFO 队列正如其名，其中只包含读取请求。同样地，写入请求 FIFO 队列中只包含写入请求。FIFO 队列中每条请求都会被赋予一个到期时间（expixation time）。读取请求 FIFO 队列具有 500 毫秒的到期时间，写入请求 FIFO 队列具有 5 秒的到期时间。

当一个新的 I/O 请求被提交时，它会被插入排序（insertion-sorted）至标准队列，而且会被放在相应（读取请求或写入请求）的 FIFO 队列的尾端。一般而言，送给硬盘的 I/O 请求会来自已排序标准队列的头部。若能尽量减少查找，将可获得最好的整体性能，因为标准队列按块编号排序过（如同 Linus Elevator）。

注 5：　没错，有个 I/O 调度程序以人名来命名。I/O 调度程序有时称为电梯（*elevator*）算法，因为此类算法解决问题的方式类似于让电梯运作平稳的做法。

当有一个 FIFO 队列的头部出现已到期的项目时，I/O 调度程序会停止从标准队列派送 I/O 请求，而会开始服务来自该 FIFO 队列的请求 —— 所服务的是位于 FIFO 队列头部的请求，再加上一批额外的请求。I/O 调度程序只需要检查位于队列头部的请求，因为这些都是持续了最久的请求。

这样，Deadline I/O 调度程序可以对 I/O 请求实施软性期限（soft deadline）。尽管无法保证 I/O 请求会在它时间届满之前被服务到，不过 I/O 调度程序一般可以在 I/O 请求接近到期时间的时候服务它。因此，Deadline I/O 调度程序会在没有任何一个请求会挨饿太久的情况下持续提供一个不错的整体性能。因为读取请求具有较短的到期时间，所以写入饿死读取的问题会被最小化。

Anticipatory I/O 调度程序

尽管 Deadline I/O 调度程序已经不错了，但是并不完美。这与读取依赖性（read dependency）有关。使用 Deadline I/O 调度程序时，如果一系列读取请求中的第一个读取请求的到期时间已到或快要到了，它会马上获得服务，然后 I/O 调度程序会回头服务来自已排序队列（sorted queue）的 I/O 请求 —— 到目前为止一切顺利。但假设应用程序取得数据后接着又送出另一个读取请求，那会如何？最后它也会到达到期时间，于是 I/O 调度程序会把它提交给磁盘，而磁盘会立即进行查找以便处理该请求，接着必须查找回来以继续处理来自已排序队列的请求。这种来回查找的现象可能会持续一段时间，因为许多应用程序都会呈现此行为。尽管延迟时间被保持在最少，但是整体性能并不是很好，因为有读取请求不断涌进，于是磁盘必须继续来回进行查找，以便处理它们。如果磁盘可以停顿片刻去等待另一个读取请求，而不立即去服务已排序的队列，则性能可以获得改善。但不幸的是，到了应用程序被调度的时候，它又提交了下一个无关的读取请求，于是 I/O 调度程序又换档（shifted gears）了。

问题又是源自读取依赖 —— 只有在从前一个读取请求返回时才可以送出新的读取请求，但等到应用程序取得读取数据，被调度以备运行并提交它的下一个读取请求，I/O 调度程序却已经走开并着手服务其他请求。结果每个读取请求等于浪费了两次查找时间：磁盘为一个读取请求进行查找以便服务它，然后又往回查找。如果 I/O 调度程序有办法知道 —— 或预测（*anticipate*）—— 另一个读取请求很快会被提交给磁盘的同一个部分，就不会浪费时间来回查找，它预期可以等到下一个读取请求。为了节省大量的查找时间，浪费几毫秒的等待时间绝对是值得的。

这就是 Anticipatory（预测）I/O 调度程序的运作方式。它一开始是 Deadline（期限）I/O 调度程序，但是具备额外的预测机制。若有一个读取请求被提交，则 Anticipatory I/O 调度程序会在期限内对它提供服务，一如往常。然而，与 Deadline I/O 调度程序不

同的是，Anticipatory I/O 调度程序接着就什么事也不做地等待至多 6 毫秒的时间。应用程序有机会在这 6 毫秒的时间内送出另一个针对文件系统中同一部分的读取请求。如果是这样的话，该请求会立即获得服务，而且 Anticipatory I/O 调度程序会等到更多请求。如果 6 毫秒过去了并没有出现读取请求，则 Anticipatory I/O 调度程序会判断它猜错了，而且会回去做它之前该做的事（也就是去服务已排序的标准队列）。如果能够有适当数量的请求被预测成功，则可以节省大量的时间 —— 每成功一次等于两次代价高的查找操作。因为多数读取请求都是彼此依赖的，所以预测算法所付出的代价是值得的。

CFQ I/O 调度程序

Complete Fair Queuing（完全公平队列，简称 CFQ）I/O 调度程序可达成相同的目标，尽管它采用了不同的方法（注 6）。使用 CFQ 时，每个进程都会被指定一个独立的队列，而每个队列都会被指定一个时间片（timeslice）。I/O 调度程序会以轮转的方式（round-robin fashion）扫描每个队列，服务队列里的请求，直到该队列的时间片用完或该队列已无未决请求。就后者而言，CFQ I/O 调度程序将闲置一段时间（默认为 10 毫秒）等待队列里出现新的请求。如果预期得到回报，则 I/O 调度程序可以免掉查找的动作，如果没有，则等待是徒劳的，于是调度程序会移往下一个进程的队列。

在每个进程的队列里，同步化请求（例如读取请求）的优先级高于异步化请求。使用这种方式，CFQ 会让读取请求优先，于是可避免写入饿死读取问题。因为每个进程均设置了队列，所以CFQ I/O 调度程序会公平对待所有进程，同时仍然能够提供不错的整体性能。

CFQ I/O 调度程序非常适合大部分的工作量，这使得它成为电梯的第一选择。

Noop I/O 调度程序

Noop（无运算）I/O 调度程序是最基本的电梯。它不会排序，只会进行基本的合并，可用于无需排序 I/O 请求的特殊设备。

选择并设定你的 I/O 调度程序

默认 I/O 调度程序可以在开机时经 *iosched* 的内核命令行参数 `elevator=` 来选择。有效的选项包括 *as*、*cfq*、*deadline* 以及 *noop*。你还可以在内核运行时经 */sys/block/device/queue/scheduler* 来为特定的设备选择 I/O 调度程序。读取此文件可返回相应设备当前所使用的 I/O调度程序，对此文件写入一个有效的选项便可以对相应的设备

注 6：接下来所探讨的 CFQ I/O 调度程序正如它当前的实现。以往的实现并未使用时间片（timeslice）或是探试性预测（anticipation heuristic），而是以类似的方式运作。

设定 I/O 调度程序。例如，要将设备 *hda* 设定成使用 CFQ I/O 调度程序，可以这么做：

```
# echo cfq > /sys/block/hda/queue/scheduler
```

此外，目录 */sys/block/device/queue/iosched* 中所包含的文件让管理者可以取得并设定与 I/O 调度程序有关的可调整值。要操作哪些选项取决于当前所使用的 I/O 调度程序，要变更任何设定值则必须具备 root 特权。

一个好的程序设计者并不需要了解底层的 I/O 子系统。尽管如此，对这个子系统有所认识将有利于程序的优化。

优化 I/O 性能

因为相对于系统中其他组件而言，磁盘 I/O 的速度是如此的缓慢，然而 I/O 却是现代计算机技术一个重要的方面，所以 I/O 性能的最大化至关重要。

尽量减少 I/O 操作（将许多规模较小的操作凝聚成少数规模较大的操作），执行对齐块大小的 I/O 或者使用用户缓冲机制（参见第三章），以及利用高级 I/O 技术 —— 例如向量式 I/O、特定位置的 I/O（参见第二章）、异步 I/O，这些都是进行系统程序设计时必须加以考虑的重要步骤。

然而，I/O 密集型关键应用程序则可以利用额外的技巧来最大化性能（maximize performance）。尽管 Linux 内核，正如先前所提到的，会利用高级的 I/O 调度程序来尽量减少可怕的磁盘查找操作，不过用户空间应用程序也可以朝相同的目标，使用类似的方式以进一步改善性能。

用户空间的 I/O 调度

I/O 密集型应用程序会发出大量的 I/O 请求，而且需要花时间排序及合并其未决的 I/O 请求，进行与 Linux I/O 调度程序一样的工作（注 7）。

如果你知道 I/O 调度程序会逐块地排序请求、最小化查找（minimizing seeks）以及让磁盘的读写头平稳地以线性的方式移动，那么你可能会觉得奇怪，为何相同的工作要做两遍？现在假设有一个应用程序提交了大量未排序的 I/O 请求。这些请求将以一般随机的顺序抵达 I/O 调度程序的队列。I/O 调度程序会履行它的责任，在把这些请求送往磁盘之前，对它们进行排序及合并 —— 但是在这些请求开始提交给磁盘的同时，应用程

注 7：　此处所探讨的技术应该只能使用于 I/O 密集型关键应用程序。对没有任何东西可排序的应用程序（因为它并未送出许多 I/O 请求）进行 I/O 请求的排序是愚蠢且不必要的。

序仍会产生 I/O 并且提交它们。I/O调度程序一段时间内只能排序一小部分的请求 ——例如，有少数来自此应用程序，而其余多数是未决请求。尽管应用程序的每批请求都经过排序，但是整个队列以及之后的任何请求都不是这个已排序结果的一部分。

因此，如果一个应用程序产生了许多请求 —— 特别是如果这些请求是用于访问遍及于磁盘的数据的话，而且这个应用程序又能在提交这些请求之前先排序它们，则可确保它们按照所期望的顺序抵达 I/O 调度程序。

然而，用户空间应用程序所能访问的信息与内核不同。在 I/O 调度程序内部的最底层，这些请求已经根据物理磁盘块被指定。排序它们并不难，但是在用户空间中，这些请求会根据文件和偏移量被指定。用户空间应用程序必须探索适合的信息，并且对文件系统的布局（layout）作出有根据的猜测。

由于目的是替针对特定文件的一串 I/O 请求确定最有利于查找的顺序（most seek-friendly ordering），用户空间应用程序有以下三种排序方式可供选择：

- 完整路径
- inode 编号
- 文件的物理磁盘块

至于要采用哪一种排序方式，你得作出取舍。下面让我们来看看每一种排序方式。

按路径排序

按路径名称排序是最简单（但最没效率）且接近按块排序（block-wise sort）的一种方式。由于多数文件系统使用的是布局算法（layout algorithm），所以每个目录中的文件在磁盘上往往是相邻的（因而这些目录共享同一个父目录）。相同目录中的文件在同一时间被创建的可能性对此一特性只有增强的作用。

因此，按路径排序大致接近于文件在磁盘上的物理位置。相较于位于文件系统中完全不同部分的两个文件，位于同一目录中的两个文件有更好的机会被放在彼此靠近的位置上。此做法的缺点是并未考虑碎片（fragmentation）问题：文件系统中的碎片越多，按路径排序就越没用处。即使忽略碎片问题，按路径排序也只会接近实际的按块排序的顺序。就优点而言，按路径排序至少比较适用于所有操作系统。无论文件的布局采用何种方式，就 temporal locality（时间局部性）而言，按路径排序至少有轻微的准确度，这也是一个容易完成的排序工作。

按 inode 排序

inode 是 Unix 的概念，内含与个别文件有关的元数据。尽管一个文件的数据可能会耗用多个物理磁盘块，但是每个文件只会有一个 inode（内含文件的大小、使用权限、拥有者等信息）。我们将在第七章深入探讨 inode。现在，你只需要知道两件事：每个文件都会有一个相应的 inode，而且每一个 inode 都会被赋予独一无二的编号。

假设存在以下关系：

```
文件 i 的 inode 编号 < 文件 j 的 inode 编号
```

则按 inode 排序会优于按路径排序。这意味着，在一般情况下：

```
文件 i 的物理块 < 文件 j 的物理块
```

以上关系适用于 Unix 风格的文件系统，例如 *ext2* 和 *ext3*。文件系统若不使用实际的 inode，则任何事情都可能发生，但是 inode 编号（无论它映射至何处）依然是一个不错的一阶近似值（first-order approximation）。

为了取得 inode 编号，可以使用 `stat()` 系统调用（相关细节参见第七章）。由于每个 I/O 请求所涉及的文件都会关联到一个 inode，所以这些请求可按照 inode 编号以从小到大的方式排序。

下面是一个简单的程序，它会输出特定文件的 inode 编号：

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

/*
 * get_inode —— 返回与特定文件描述符相关联的文件的 inode,
 * 执行失败时返回 -1
 */
int get_inode (int fd)
{
        struct stat buf;
        int ret;

        ret = fstat (fd, &buf);
        if (ret < 0) {
                perror ("fstat");
                return -1;
        }

        return buf.st_ino;
```

```
}

int main (int argc, char *argv[])
{
        int fd, inode;

        if (argc < 2) {
                fprintf (stderr, "usage: %s <file>\n", argv[0]);
                return 1;
        }

        fd = open (argv[1], O_RDONLY);
        if (fd < 0) {
                perror ("open");
                return 1;
        }

        inode = get_inode (fd);
        printf ("%d\n", inode);

        return 0;
}
```

将 get_inode() 函数稍微修改一下就能应用在你的程序中。

按 inode 编号排序有几个优点：inode 编号容易取得、容易排序，而且很接近物理文件布局。它的主要缺点是：碎片问题会降低接近的程度，这种接近只是一种猜测，而且对非 Unix 文件系统而言这种接近较不准确。尽管如此，在用户空间对 I/O 请求进行调度时，这是最常被用到的方法。

按物理块排序

设计你自己的电梯算法时，最好的办法当然是按物理磁盘块排序（sort by physical disk block）。正如稍早所做的讨论，每个文件会被划分成数个逻辑块，而逻辑块是一个文件系统的最小配置单元。一个逻辑块的大小与文件系统有关，每个逻辑块会映射至单一的物理块。因此，我们可以在一个文件中找出逻辑块的数目，确定它们映射至哪些物理块，并根据此结果进行排序。

内核提供了一个方法让我们可以从一个文件的逻辑块数目取得物理磁盘块。此工作可通过 ioctl() 系统调用的 FIBMAP 命令（相关细节参见第七章）来完成：

```
ret = ioctl (fd, FIBMAP, &block);
if (ret < 0)
        perror ("ioctl");
```

其中，fd 是特定文件的文件描述符，而 block 就是逻辑块（我们想取得它的物理块）。执行成功时，返回并将 block 替换成物理块编号。所传入的逻辑块具有 zero-indexed

（索引从零开始）和 file-relative 的特性。也就是说，如果一个文件由 8 个逻辑块组成，则有效值为 0 ～ 7。

要找出逻辑块与物理块的映射关系需要两个步骤。首先，我们必须确定所指定文件中有多少块。此工作可通过 `stat()` 系统调用来完成。其次，我们必须针对每个逻辑块送出 `ioctl()` 请求，以便找到相应的物理块。

下面是一个简单的程序，执行时只需要在命令行上传递一个文件名称给它即可：

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <linux/fs.h>

/*
 * get_block —— 替与特定 fd 相关联的文件返回
 *映射至 logical_block 的物理块
 */
int get_block (int fd, int logical_block)
{
        int ret;

        ret = ioctl (fd, FIBMAP, &logical_block);
        if (ret < 0) {
                perror ("ioctl");
                return -1;
        }

        return logical_block;
}

/*
 * get_nr_blocks —— 返回与 fd 相关联的文件所耗用的逻辑块的数目
 */
int get_nr_blocks (int fd)
{
        struct stat buf;
        int ret;

        ret = fstat (fd, &buf);
        if (ret < 0) {
                perror ("fstat");
                return -1;
        }

        return buf.st_blocks;
}
```

```
/*
 * print_blocks —— 替与 fd 相关联的文件所耗用的每个逻辑块
 * 在标准输出中输出如下信息：
 * "(logical block, physical block)"
 */
void print_blocks (int fd)
{
        int nr_blocks, i;

        nr_blocks = get_nr_blocks (fd);
        if (nr_blocks < 0) {
                fprintf (stderr, "get_nr_blocks failed!\n");
                return;
        }

        if (nr_blocks == 0) {
                printf ("no allocated blocks\n");
                return;
        } else if (nr_blocks == 1)
                printf ("1 block\n\n");
        else
                printf ("%d blocks\n\n", nr_blocks);

        for (i = 0; i < nr_blocks; i++) {
                int phys_block;

                phys_block = get_block (fd, i);
                if (phys_block < 0) {
                        fprintf (stderr, "get_block failed!\n");
                        return;
                }
                if (!phys_block)
                        continue;

                printf ("(%u, %u) ", i, phys_block);
        }

        putchar ('\n');
}

int main (int argc, char *argv[])
{
        int fd;

        if (argc < 2) {
                fprintf (stderr, "usage: %s <file>\n", argv[0]);
                return 1;
        }

        fd = open (argv[1], O_RDONLY);
        if (fd < 0) {
                perror ("open");
                return 1;
```

```
        }

        print_blocks (fd);

        return 0;
}
```

因为文件往往是连续的，所以很难按照每个逻辑块（per-logical-block basis）来排序我们的 I/O 请求，比较有意义的做法是按照所指定文件的第一个逻辑块的位置来排序。于是就不需要用到 get_nr_blocks()，而且我们的应用程序会根据：

```
get_block (fd, 0);
```

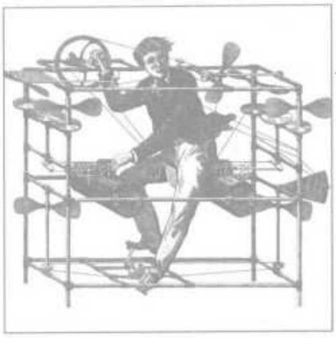
的返回值来进行排序。FIBMAP 的缺点是需要 CAP_SYS_RAWIO 的能力 —— 也就是说，需要 root 特权。因此，不具有 root 特权的应用程序无法使用这个方法。此外，尽管 FIBMAP 命令已经标准化，但是它的实现则留给文件系统自己来完成。尽管一些常见的文件系统（例如 *ext2* 和 *ext3*）都会支持它，不过一些比较少见的文件系统可能不支持它。如果 FIBMAP 未受支持，ioctl() 调用将返回 EINVAL。

这个方法的一个优点是它会返回特定文件所位于的实际物理磁盘块，也就是你实际想排序的东西。即使你只是根据一个块的位置来排序针对单一文件的所有 I/O（内核的 I/O 调度程序是以逐块的方式排序每个请求），此方法仍非常接近理想的顺序。然而，因为这个方法需要用到 root 特权，所以对许多人来说似乎有点不切实际。

结束语

在连续这三章的课程中，我们触及了 Linux 中文件 I/O 的所有方面的内容。第二章中，我们了解了文件 I/O 的基础功能 —— 这实际上是 Unix 设计的基本知识，以及相应的系统调用，例如 read()、write()、open() 以及 close()。第三章中，我们探讨了用户空间缓冲机制以及相应的标准 C 链接库的实现。本章中，我们探讨了高级 I/O 的不同层面，从功能较强但也更复杂的 I/O 系统调用，到优化技术以及把磁盘查找（disk seek）对性能的影响纳入考虑。

下面两章我们将探讨进程管理的议题：进程的创建、销毁以及管理。请继续读下去！



第五章

进程管理

正如第一章所述，除了文件，进程（process）也是 Unix 系统中最基础的抽象概念。因为目标码会进入执行（execution）的状态—— 活跃（active）、活着（alive）、运行程序（running program），所以进程不止是汇编语言，它们由数据、资源、状态以及一个虚拟化的计算机组成。

本章中，我们将看到进程的基本功能，从创建到终止。自从最早期的 Unix 问世以来，它的基本功能几乎就未变过。本章的主题 —— 进程管理 —— 便闪烁着历史悠久的 Unix 原始设计的前瞻性思维的光芒。Unix 行走于一条很少人走过的道路，把创建新进程的步骤与加载新二进制映像的步骤分开。尽管这两项任务多半是一前一后依次完成的，不过分开进行之后，这两项任务的试验和发展可获得很大的自由度。这条很少人走过的道路让 Unix 存活至今，尽管多数操作系统均使用单一系统调用来启动一个新程序，但是 Unix 却需要用到两个：fork 以及 exec。在我们说明这两个系统调用之前，让我们先深入研究进程本身。

进程 ID

每个进程都可以被表示成一个独一无二的标识符，也就是进程 ID（*process ID*，常简写成 *pid*）。在任何单一时间点上，可以保证 pid 是独一无二的。也就是说，尽管在时间 `t0` 上只能有一个进程有 pid 770（如果所有进程中有任何一个进程具有此值），但是无法保证在时间 `t1` 上不会存在一个不同的进程具有 pid 770。然而，基本上多数程序会假设内核不会轻易发出相同的进程 ID —— 这个假设正如你稍后将看到的，相当安全。

空闲进程（*idle process*）—— 没有其他可运行的进程时内核会“运行”（run）此进程 —— 具有 pid 0。系统启动后内核所执行的第一个进程称为初始化进程（*init process*），

每个进程由一个用户（*user*）以及一个组（*group*）所拥有。此所有权（ownership）可用于控制资源的访问权限。对内核而言，用户和组只是整数值。通过文件 */etc/passwd* 和 */etc/group*，这些整数值会被映射至 Unix 用户熟悉的名称（人类可阅读的字符串），例如 *root* 用户以及 *wheel* 组（一般来说，Linux 内核对人类可阅读的字符串并不感兴趣，它偏好以整数来识别对象）。每个子进程会继承其父进程的用户和组的所有权。

每个进程也是进程组（*process group*）的一部分，进程组仅用于表示该进程与其他进程的关系，切勿与上述的用户 / 组的概念搞混了。因为具有同一父进程的关系，子进程一般会属于同一进程组。此外，当 shell 启动一个管道（例如，当用户键入 *ls | less*）时，管道中所有命令均会进入相同的进程组。通过进程组的概念，你可以轻易传送信号给整

子进程取得信息。从用户的角度看，进程组与工作（*job*）密切相关。

pid_t

从程序设计的角度看，进程 ID 会被表示成 pid_t 类型（定义于头文件 *<sys/types.h>* 中）。其背后所使用的 C 类型与架构有关，而且并未定义在任何 C 标准中。然而，在 Linux 之上，pid_t 通常是对 int 这个 C 类型的一个类型定义（typedef）。

取得进程 ID 以及父进程 ID

getpid() 系统调用用于返回进行调用的进程（invoking process）的进程 ID：

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpid (void);
```

getppid() 系统调用用于返回进行调用的进程的父进程的进程 ID：

```
#include <sys/types.h>
#include <unistd.h>

pid_t getppid (void);
```

这两个函数都不会返回错误信息。因此，很容易使用：

```
printf ("My pid=%d\n", getpid());
printf ("Parent's pid=%d\n", getppid());
```

如何知道 pid_t 是一个有符号整数？好问题！答案很简单，就是我们也不知道。即使我们可以很有把握地假设 pid_t 在 Linux 之上是一个 int，但是这种猜测违反了使用抽象类型的目的并且伤害了可移植性。不幸的是，正如 C 语言的所有类型定义，并不

存在一个简单的方式可用来输出 pid_t 的值 —— 这是抽象类型的一部分，而且在技术上需要一个我们所缺少的 pid_to_int() 函数。然而，为了 printf() 的用途而将这些值视为整数是很常见的。

运行一个新进程

Unix 中，将一个程序映像（program image）加载至内存并加以执行的步骤与创建一个新进程的步骤是分开的。有一个系统调用（实际上是一系列调用中的一个）可用于将二进制程序（binary program）加载至内存，取代地址空间先前的内容，以及开始新程序的执行。这称为执行（*executing*）一个新程序，而此功能由 *exec* 系列调用（family of calls）提供。

另有一个系统调用可用于创建一个新进程，而这个新进程最初几乎就是其父进程的副本。通常，这个新进程会立即执行一个新程序。创建一个新进程的步骤称为派生（*forking*），而此功能提供自 fork() 系统调用。因此，如果要在一个新进程中执行一个新的程序映像，则需要两个步骤：首先进行派生步骤以创建一个新进程，然后进行执行步骤以把一个新映射加载至该进程。下面我们会先说明 exec 系列调用，然后说明 fork()。

exec 系列调用

并不存单一的 exec 函数，事实上，那是单一系统调用上所建立的 exec 系列函数。让我们先来看看其中最简单的一个，execl()：

```
#include <unistd.h>

int execl (const char *path,
           const char *arg,
           ...);
```

通过将 path 参数所指向的程序加载至内存，execl() 可以把当前进程的映像替换成新的映像。arg 参数是该程序的第一个参数，而省略号（...）用来表示数目不定的参数 —— execl() 是一个 *variadic*（参数数目不定的）函数，这意味着后面有可能会跟着额外的参数，一个接着一个。此参数列表必须以 NULL 结束。

例如，下面的程序代码会以 */bin/vi* 取代当前所执行的程序：

```
int ret;

ret = execl ("/bin/vi", "vi", NULL);
if (ret == -1)
        perror ("execl");
```

注意，按照 Unix 的惯例，我们会以“vi”作为程序的第一个参数。当 shell 派生／执行(forks/execs) 进程时，会以路径的最后一个元素“vi”作为第一个参数，所以一个程序可以检查它的第一个参数 argv[0]，以便找出它的二进制映像的名称。在许多情况下，尽管用户所看到的是名称不同的系统实用程序，但它们其实是将单一程序硬链接至不同名称的结果。该程序可以使用第一个参数来决定自己的行为。

再举一个例子，如果你想要编辑文件 */home/kidd/hooks.txt*，你可以执行如下的程序代码：

```
int ret;

ret = execl ("/bin/vi", "vi", "/home/kidd/hooks.txt", NULL);
if (ret == -1)
        perror ("execl");
```

正常情况下，execl() 不会返回。执行成功时，execl() 会以跳跃至新程序的入口点作为结束，而刚才所执行的程序代码会从进程的地址空间中消失。然而，发生错误时，execl() 会返回 -1 并且设定 errno 以指出所发生的问题。稍后我们会看到可设定哪些值 errno 。

一次成功的 execl() 调用不仅会改变地址空间和进程映像，也会改变进程的某些属性：

- 任何未决信号会遗失。
- 进程所捕捉到的任何信号(参见第九章)会回到它们的默认行为，因为它们的信号处理程序已消失于进程的地址空间中。
- 内存的任何锁定（参见第八章）会被丢弃。
- 多数线程属性会回到它们的默认值。
- 多数进程统计数据会被重置。
- 与进程的内存有关的任何东西，包括所映射的任何文件，会被丢弃。
- 单独存在于用户空间的任何东西，包括 C 链接库的功能，例如 atexit() 的行为，会被丢弃。

然而，进程的许多特性并不会改变。例如，进程 ID、父进程 ID、优先级以及具所有权的用户和组都维持不变。

一般情况下，已打开的文件可以跨 exec 被继承。这意味着新执行的程序可以全面访问原进程中已打开的所有文件，假设它知道相应的文件描述符的值。然而，这通常不是期望的行为。通常的做法是在进行 exec 之前先关闭文件，尽管也能通过 fcntl() 指示内核自动完成此事。

其余成员

除了 execl()，exec 系列调用还有其他五个成员：

```
#include <unistd.h>

int execlp (const char *file,
            const char *arg,
            ...);

int execle (const char *path,
            const char *arg,
            ...,
            char * const envp[]);

int execv (const char *path, char *const argv[]);

int execvp (const char *file, char *const argv[]);

int execve (const char *filename,
            char *const argv[],
            char *const envp[]);
```

此处所用到的助记符很简单。l 和 v 用于界定参数是提供自一个列表（*l*ist）还是一个数组（*v*ector）。而 p 表示用户的完整路径（*p*ath）可用于搜索指定的文件。使用 p 助记符表示的调用时可以只指定文件名，只要该文件就位于用户的路径中。最后，e 表示新的进程还提供了一个新的环境。奇怪的是，尽管不是为了技术上的理由而要刻意遗漏，exec 系列调用中并没有任何成员会同时搜索路径并取得一个新的环境。这或许是因为，p 助记符表示的调用是给 shell 使用的，而 shell 所执行的进程通常会从 shell 继承它们的环境。

在 exec 系列调用中，使用数组的成员与使用列表的成员并没有什么不同，差别只是前者所传入的是一个数组，而后者所传入的是一个列表。数组的使用让参数可以等到运行时再决定。如同长度不定的参数列表，数组必须以 NULL 作为结束。

如同我们先前所举的例子，下面的程序代码片段会以 execvp() 来执行 *vi*：

```
const char *args[] = { "vi", "/home/kidd/hooks.txt", NULL };
int ret;

ret = execvp ("vi", args);
if (ret == -1)
        perror ("execvp");
```

假设 */bin* 位于用户的路径中，则这个例子的运作方式类似于前一个例子。

Linux 中，exec 系列调用中只有一个成员是系统调用，其余成员则是 C 链接库所提供的系统调用包裹函数。因为 variadic（参数数目不定的）系统调用难以实现，而且因为

用户路径的概念仅存在于用户空间，所以这个唯一的系统调用就是 execve()。这个系统调用的原型与用户调用一致。

错误值

执行成功时，exec 系列调用不会返回；执行失败时，它们会返回 -1 并且将 errno 设定成下面的其中一个值：

E2BIG
: 参数列表（arg）或环境（envp）的字节总数太大。

EACCESS
: 进程无权搜索 path 的组成元素；path 不是一个常规文件；目标文件未被标记成可执行；path 或 file 位于以 noexec 挂载的文件系统上。

EFAULT
: 使用了无效的指针。

EIO
: 发生了低级 I/O 错误（这是相当糟糕的情况）。

EISDIR
: path 的最终元素（final component）或解释器（interpreter）是一个目录。

ELOOP
: 解析 path 时系统遇到太多符号链接。

EMFILE
: 进行调用的进程所打开的文件数目已到达它的上限。

ENFILE
: 所打开的文件数目已到达系统范围的上限。

ENOENT
: path 或 file 的目标文件不存在，或者所需要的共享链接库不存在。

ENOEXEC
: path 或 file 的目标文件是一个无效的二进制文件或是用于不同的机器架构。

ENOMEM
: 内核内存不足以执行一个新的程序。

ENOTDIR
: path 中有一个非最终元素（nonfinal component）不是目录。

EPERM

path 或 file 位于以 nosuid 挂载的文件系统上，但用户不是 root，而且 path 或 file 设定了 suid 或 sgid 位。

ETXTBSY

path 或 file 的目标文件被另一个进程打开以备写入（open for writing）。

fork() 系统调用

一个进程可以通过 fork() 系统调用创建一个新的进程来运行与当前相同的映像：

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork (void);
```

一次成功的 fork() 调用会创建一个新进程，此进程的各方面与进行调用的进程几乎一致。这两个进程会继续运行，而且 fork() 会返回，好像什么事都没发生一样。

新的进程称为原进程的子进程（child），而原进程则称为父进程（parent）。在子进程中，成功的 fork() 调用会返回 0；在父进程中，fork() 会返回子进程的 pid。子进程的各方面和父进程几乎一致，除了：

- 子进程的 pid 与父进程的不同。
- 子进程的 parent pid 会被设定成其父进程的 pid。
- 子进程中的资源统计数据会被重置为零。
- 任何未决信号会被清除，而且不会被子进程继承（参见第九章）。
- 所取得的任何文件锁定不会被子进程继承。

发生错误时，子进程不会被创建，fork() 会返回 -1 并且将 errno 设定成适当的值。errno 的可能值有两个，但是具有三种可能的意义：

EAGAIN

内核无法分配某些资源，例如一个新的 pid，或者已到达 RLIMIT_NPROC 资源限制（resource limit，常简写为 rlimit，参见第六章）。

ENOMEM

内核内存不足以完成此请求。

它的用法很简单：

```
pid_t pid;

pid = fork ();
if (pid > 0)
        printf ("I am the parent of pid=%d!\n", pid);
else if (!pid)
        printf ("I am the baby!\n");
else if (pid == -1)
        perror ("fork");
```

`fork()` 的最常见用法就是创建一个新的进程，然后将一个新的二进制映像加载进来——你可以将此想成是一个 shell 为用户运行一个新的程序，或者是一个进程产生了一个辅助程序。首先进程会派生一个新的进程，然后子进程会执行一个新的二进制映像。这个“fork 加上 exec”的组合既常见又简单。下面的例子会派生一个新的进程以执行二进制文件 */bin/windlass*：

```
pid_t pid;

pid = fork ();
if (pid == -1)
        perror ("fork");

/* 子进程…… */
if (!pid) {
        const char *args[] = { "windlass", NULL };
        int ret;

        ret = execv ("/bin/windlass", args);
        if (ret == -1) {
                perror ("execv");
                exit (EXIT_FAILURE);
        }
}
```

父进程会继续运行并没有改变，除此之外它现在还有一个新的子进程。调用 `execv()` 之后会变成子进程运行 */bin/windlass* 程序。

写入时才复制

在早期的 Unix 系统中，进程的派生很简单。`fork()` 被调用时，内核会替内部的所有数据结构创建副本，复制进程的页表（page table）项，以及将父进程的地址空间逐页复制（page-by-page copy）到子进程的新地址空间。但是这种逐页复制的做法挺费时的（至少以内核的立场来看）。

现代 Unix 系统的表现较理想。现代的 Unix 系统，例如 Linux，不再整批复制父进程的地址空间，而是采用写入时才复制（*copy-on-write*，常简写成 COW）页面。

写入时才复制是一种延迟优化策略，目的是减轻复制资源的开销。它的前提很简单：如果同一个资源有许多消费者（consumer）要读取它们自己的副本，则不需要复制该资源的副本。事实上，每个消费者会得到一个指向相同资源的指针。只要没有消费者企图修改属于自己的副本，则排他访问（exclusive access）的假象依然存在，而且可以避免复制副本的开销。如果有一个消费者企图修改属于自己的副本，则在此情况下，会自动复制该资源的副本，而且此副本会被提供给进行修改的消费者。消费者的能力不足以在其他消费者仍在共享原本未修改资源的同时修改属于自己的副本。因此才会出现“写入时才复制”的策略。

此策略的主要好处是，如果没有消费者企图修改属于自己的副本，则不需要建立任何副本。延迟算法（lazy algorithm）的一般好处 —— 将代价昂贵的动作推延到最后一刻进行 —— 也适用此说法。

在虚拟内存的特例中，写入时才复制的实现是以每一页（per-page）为基础的。因此，只要进程没有修改它所有的地址空间，则整个地址空间就不需要副本。fork（派生）动作完成之后，父进程和子进程会认为它们自己拥有唯一的地址空间，然而事实上它们是共享父进程的原始页面 —— 也可能被其他的父进程或子进程等所共享！

内核的实现方式很简单。在内核的页面相关数据结构中，页面会被标注成 read-only（仅供读取）以及 copy-on-write（写入时才复制）。如果进程企图修改一个页面或是发生了页面失误，则内核会通过自动复制页面的副本来处理页面失误。在此情况下，页面的 copy-on-write 属性会被清除，而且页面不再提供共享。

因为现代机器架构在它们的内存管理单元（memory management unit，常简写成 MMU）中为“写入时才复制”策略提供了硬件层的支持，所以实现起来既简单又容易。

进行 fork 动作时，“写入时才复制”的好处更多。因为进行 fork 动作之后，有很大的比例会立即进行 exec 动作，所以将父进程的地址空间复制到子进程的地址空间通常完全是在浪费时间：如果子进程立即执行了一个新的二进制映像，它先前的地址空间就会被抹除。在此情况下便可使用“写入时才复制”优化策略。

vfork()

在写入时才复制页面的策略出现之前，Unix 设计者所关注的是，如果后面会立即进行 exec 动作，fork 动作期间浪费时间的地址空间复制。因此，BSD 开发者在 3.0BSD 中公开了 `vfork()` 系统调用：

```
#include <sys/types.h>
#include <unistd.h>

pid_t vfork (void);
```

一次成功的 vfork() 调用具有跟 fork() 一样的行为，但是子进程必须立即发出一个成功的 exec 调用或是调用 _exit()（下一节会讨论）以便结束执行。vfork() 调用会通过暂停父进程直到子进程终止或者执行一个新的二进制映像的方式来避免地址空间和页表的复制。在过渡期间，父进程和子进程共享（但是不使用写入时才复制策略）它们的地址空间和页表项。事实上，vfork() 执行期间只会复制内核内部的数据结构。因此，子进程不得修改地址空间中任何内存的内容。

vfork() 系统调用是一个历史遗迹，从来就不应该在 Linux 上实现，但是必须指出的是，就算使用写入时才复制策略，vfork() 的速度仍然比 fork() 快，因为不需要复制页表项（注 1）。尽管如此，写入时才复制页面策略的出现，削弱了为 fork() 提供替换方案的任何论据。的确，直到 2.2.0 版的 Linux 内核，vfork() 仍旧只是一个内含 fork() 的包裹函数。由于对 vfork() 的需求弱于对 fork() 的需求，所以这种 vfork() 的实现是可行的。

严格来说，所有的 vfork() 实现几乎都有瑕疵：考虑 exec 调用失败的情况！在此情况下，父进程将无限期暂停，除非子进程想通了该怎么办或者直到子进程结束执行。

终止一个进程

POSIX 和 C89 定义了一个标准函数用于终止当前进程：

```
#include <stdlib.h>

void exit (int status);
```

exit() 调用会执行一些基本的 shutdown（关闭）步骤，然后指示内核终止进程。此函数无法返回错误 —— 事实上，它绝不会返回任何值。因此，任何指令若跟在 exit() 调用之后，则不会有任何作用。

其中，status 参数用于表示进程的结束状态。其他的程序（以及 shell 上的用户）可以检查此值。特殊的是，status & 0377 会被返回给父进程。本章稍后会查看此返回值。

注 1：尽管目前不是 2.6 版 Linux 内核的一部分，不过你可以在 Linux Kernel Mailing List（*lkml*）上看到一个用于实现写入时才复制共享页表项的补丁文件（patch）。使用 vfork() 时，对内核加入此补丁不会有任何好处。

EXIT_SUCCESS 与 EXIT_FAILURE 被定义成可移植的方式来表示成功和失败的状态。在 Linux 上，0 通常表示成功；非零值，例如 1 或 -1，则表示失败。

因此，要表示成功的结束，就这么简单：

```
exit (EXIT_SUCCESS);
```

终止进程之前，C 链接库会依次执行下面的 shutdown 步骤：

1. 按照与注册相反的顺序调用以 atexit() 或 on_exit() 注册的任何函数（本章稍后会讨论这些函数）。
2. 刷新（flush）所有已打开的标准 I/O 流（参见第三章）。
3. 移除任何由 tmpfile() 函数所创建的临时文件。

这些步骤完成了进程在用户空间中所需进行的所有工作，接着 exit() 会调用系统调用 _exit()，让内核可以处理终止进程的其余工作：

```
#include <unistd.h>

void _exit (int status);
```

当一个进程退出时，内核会清理它为进程创建的所有资源中不再使用的部分。这包括但不仅限于已分配的内存、已打开的文件以及 System V semaphores。完成清理工作后，内核会销毁进程并通知父进程其子进程已死亡的事实。

应用程序可以直接调用 _exit()，但是此类行为通常没有任何意义：多数应用程序需要进行由完全结束所提供的某个清理工作，例如刷新 *stdout* 流。然而请注意，派生动作之后，vfork() 的用户应该调用 _exit() 而不是调用 exit()。

ISO C99 标准中加入了 _Exit() 函数，它的行为如同 _exit()：

```
#include <stdlib.h>

void _Exit (int status);
```

终止进程的其他方式

结束一个程序的典型方式并非通过显式的系统调用，而是从程序的结尾离开。就 C 语言的情况来说，这发生在从 main() 函数返回时。然而，这种从结尾离开（falling off the end）的做法仍会调用一个系统调用：编译器会在自己的 shutdown code（关闭程序代码）之后自动插入一个 _exit()。经由 exit() 或从 main() 返回值，显式返回

结束状态是一个好习惯。shell 可以使用此结束值来判断命令的执行是成功还是失败。注意，执行成功时会使用 exit(0)，或由 main() 返回 0 值。

一个进程也可以被一个信号终止，如果该信号的默认行为是终止进程。此类信号包括 SIGTERM 和 SIGKILL（参见第 九 章）。

结束一个程序的执行最后一个方式是被内核“杀掉”。内核可以在进程执行了非法指令、造成了分段违例（segmentation riolation）、用完内存等时候“杀掉”（kill）它。

atexit()

POSIX 1003.1-2001 所定义以及 Linux 所实现的 atexit() 链接库调用，可用于注册于进程终止时所调用的函数：

```
#include <stdlib.h>

int atexit (void (*function)(void));
```

一次成功的 atexit() 调用会注册指定的函数，从而在正常进程终止期间得以运行，也就是说，经 exit() 或从 main() 返回来终止一个进程。如果进程调用了一个 exec 函数，则已注册函数列表会被清除（因为这些函数已经不在新进程的地址空间中）。如果一个进程通过信号终止，则不会调用所注册的函数。

此处所指定的函数不需要参数也没有返回值。它的原型具有如下的形式：

```
void my_function (void);
```

这些函数的调用顺序与注册的顺序相反，也就是说，这些函数会被存放在一个堆栈里，而且以后进先出（LIFO）的顺序被取出。所注册的函数不得调用 exit()，否则它们会不断地被递归。如果一个函数需要早一点终止进程，它应该调用 _exit()。然而我们不建议这么做，因为这样可能会有重要的函数没有被执行到。

POSIX 标准规定 atexit() 至少要支持 ATEXIT_MAX 个注册函数，此值必须至少是 32。实际的最大值可经 sysconf() 与 _SC_ATEXIT_MAX 参数值来取得：

```
long atexit_max;

atexit_max = sysconf (_SC_ATEXIT_MAX);
printf ("atexit_max=%ld\n", atexit_max);
```

执行成功时，atexit() 会返回 0；发生错误时，它会返回 -1。

下面是一个简单的例子：

```
#include <stdio.h>
#include <stdlib.h>

void out (void)
{
        printf ("atexit() succeeded!\n");
}

int main (void)
{
        if (atexit (out))
                fprintf(stderr, "atexit() failed!\n");

        return 0;
}
```

on_exit()

SunOS 4 自己定义了与 atexit() 等效的函数，而且 Linux 的 *glibc* 也支持这个函数：

```
#include <stdlib.h>

int on_exit (void (*function)(int , void *), void *arg);
```

此函数的工作方式如同 atexit()，但是所注册函数的原型有所不同：

```
void my_function (int status, void *arg);
```

status 参数值会传递给 exit() 或是从 main() 返回。arg 参数是第二个参数，会传递给 on_exit()。要注意的是，当函数最后被调用时，必须确保 arg 所指向的内存是有效的。

Solaris 的最新版本不再支持此函数。你应该改用与标准兼容的 atexit() 函数。

SIGCHLD

当一个进程终止时，内核会传送 SIGCHLD 信号给它的父进程。默认情况下，此信号会被忽略，父进程因此不会采取任何行动。然而，进程可经 signal() 或 sigaction() 系统调用选择处理此信号。这两个信号以及其余的信号会在第九章加以说明。

SIGCHLD 信号随时都可以被产生和派送，因为子进程的终止与它的父进程之间是不同步的。但是很多时候，父进程会想多了解有关其子进程终止的事情，甚至会显式等待此事件的发生。在此情况下，可以使用下一节所讨论的系统调用。

等待已终止的子进程

通过信号收到通知并没有什么不好，但是许多父进程在它们的其中一个子进程终止时，会想取得更多信息 —— 例如，子进程的返回值。

若一个子进程在终止时整体消失，则正如预期的那样，父进程将无法取回任何信息。因此，Unix 的原设计者决定，若子进程的死亡时间先于它的父进程，则内核应该让子进程进入一个特殊的进程状态。进入此状态的进程名为僵尸进程（*zombie*）。一旦进程进入此状态，则只会保留最小的骨架：一些可能会包含有用数据的基本内核数据结构。一个处于此状态的进程会等待它的父进程来打听它的状态（此过程名为等待僵尸进程）。只有在父进程取得所保留的关于已终止子进程的信息时，进程才会正式结束并且不再存在，即使是一个僵尸进程。

Linux 内核提供了若干接口用于取得关于已终止子进程的信息。这些接口中最简单的就是 POSIX 所定义的 `wait()`：

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait (int *status);
```

调用 `wait()` 会返回一个已终止子进程的 `pid`，发生错误时会返回 `-1`。如果没有子进程被终止，则此调用会受到阻挡，直到有子进程被终止。如果有进程已经被终止，则此调用会立即返回。因此，调用 `wait()` 会响应一个子进程的死亡消息 —— 例如收到一个 `SIGCHLD` 信号，所以总是会返回而不会受到阻挡。

发生错误时，`errno` 会有两个可能值：

`ECHILD`
: calling process（进行调用的进程）并不具有任何子进程。

`EINTR`
: 等待的时候收到了一个信号，让此调用提前返回。

如果不是 `NULL` 值，则 `status` 指针会包含关于子进程的额外信息。因为 POSIX 允许实现在 `status` 中定义它们认为合适的位，所以标准提供了一系列宏作为解析参数之用：

```
#include <sys/wait.h>

int WIFEXITED (status);
int WIFSIGNALED (status);
int WIFSTOPPED (status);
int WIFCONTINUED (status);
```

```
int WEXITSTATUS (status);
int WTERMSIG (status);
int WSTOPSIG (status);
int WCOREDUMP (status);
```

前两个宏的其中一个可能会返回真值（非零值），这取决于进程被终止的方式。如果进程是正常终止，则第一个宏 WIFEXITED 会返回真值 —— 也就是说，如果进程调用了 _exit()。在此情况下，宏 WEXITSTATUS 会提供传递给 _exit() 的低 8 位。

如果是信号导致进程的终止（第九章会对信号做更进一步的讨论），WIFSIGNALED 会返回真值。在此情况下，WTERMSIG 会返回导致进程终止的信号数目，而且如果进程以 core dump 来响应所收到的信号，则 WCOREDUMP 会返回真值。POSIX 并未定义 WCOREDUMP，尽管有许多 Unix 系统（包括 Linux）支持它。

如果进程被停止或继续，则 WIFSTOPPED 和 WIFCONTINUED 会分别返回真值，而且可通过 ptrace() 系统调用追踪当前的状态。这些状态一般只能应用在实现调试程序（debugger）的时候，虽然与 waitpid()（稍后会加以说明）一起使用的时候，它们还可以用于实现作业控制（job control）。一般而言， wait() 只用于沟通关于进程终止的信息。如果 WIFSTOPPED 为真， WSTOPSIG 会提供导致进程中止的信号数目。POSIX 并未定义 WIFCONTINUED，虽然未来的标准会替 waitpid() 定义此宏。从 2.6.10 版 Linux 内核起，Linux 也开始为 wait() 定义此宏。

下面的范例程序会使用 wait() 来判断它的子进程发生了什么事：

```
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>

int main (void)
{
        int status;
        pid_t pid;

        if (!fork ())
                return 1;

        pid = wait (&status);
        if (pid == -1)
                perror ("wait");

        printf ("pid=%d\n", pid);

        if (WIFEXITED (status))
                printf ("Normal termination with exit status=%d\n",
                        WEXITSTATUS (status));
```

```
        if (WIFSIGNALED (status))
                printf ("Killed by signal=%d%s\n",
                        WTERMSIG (status),
                        WCOREDUMP (status) ? " (dumped core)" : "");

        if (WIFSTOPPED (status))
                printf ("Stopped by signal=%d\n",
                        WSTOPSIG (status));

        if (WIFCONTINUED (status))
                printf ("Continued\n");

        return 0;
}
```

此程序会派生一个子进程，而此子进程会立即结束。然后父进程会执行 wait() 系统调用来判断其子进程的状态。此进程会输出子进程的 pid 以及它是如何死亡的。因为此例中子进程的终止是因为从 main() 返回，所以我们将看到如下的输出：

```
$ ./wait
pid=8529
Normal termination with exit status=1
```

如果不考虑让子进程返回，而是让它调用 abort()（注 2）来送出 SIGABRT 信号给自己，我们将看到如下的输出：

```
$ ./wait
pid=8678
Killed by signal=6
```

等待特定的进程

观察子进程的行为很重要。然而一个进程往往具有多个子进程，而且不想等待所有子进程，但会希望等待特定的子进程。解决方案之一就是对 wait() 进行多次调用，并观察每次调用的返回值。但是这相当麻烦 —— 如果你后来想检查另一个已终止的进程的状态呢？父进程必须保存 wait() 的所有输出，以备日后所需。

如果你知道要等待进程的 pid，你可以使用 waitpid() 系统调用：

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t waitpid (pid_t pid, int *status, int options);
```

注 2： 定义于头文件 <stdlib.h> 中。

相较于 wait()，waitpid() 系统调用是一个功能更强大的版本。waitpid() 所提供的额外信息让你可以进行微调（fine-tuning）。

pid 参数用于指定要等待的是哪个进程或哪些进程。它的值可分成四大部分：

< -1
: 等待其进程组 ID（process group ID）等于此值的绝对值的任何子进程。例如，传入 -500 表示所等待的是进程组 ID 为 500 的任何进程。

-1
: 等待任何子进程。此刻的行为如同 wait()。

0
: 等待与 calling process（进行调用的进程）均属于同一个进程组的任何子进程。

> 0
: 等待其 pid 等于所指定值的任何子进程。例如，传入 500 表示所等待的是 pid 为 500 的子进程。

status 参数的行为如同 wait() 的唯一参数，而且可以使用前面所介绍的宏来操作它。

options 参数可以是零个或多个以下选项的二元 OR 逻辑运算：

WNOHANG
: 不要阻挡，如果没有相符的子进程已经终止（或停止，或继续），则立即返回。

WUNTRACED
: 如果设定了，则设定 WIFSTOPPED，即使 calling process 并未追踪子进程。此标志让我们可以实现更一般化的作业控制，如同在 shell 中。

WCONTINUED
: 如果设定了，则设定 WIFCONTINUED，即使 calling process 并未追踪子进程。如同 WUNTRACED，此标志可用于实现一个 shell。

执行成功时，waitpid() 会返回状态已改变进程的 pid。如果指定了 WNOHANG 而且所指定的子进程尚未变更状态，则 waitpid() 会返回 0。发生错误时，此调用会返回 -1 并且将 errno 设定成下面的其中一个值：

ECHILD
: pid 参数所指定的进程不存在，或者不是 calling process 的子进程。

EINTR
: 未指定 WNOHANG 选项，而且在等待的时候收到了一个信号。

EINVAL

指定了无效的 options 参数。

举例来说，假设你的程序想取得特定子进程（pid 1742）的返回值，但是如果子进程尚未终止，则立即返回。你可以使用如下的程序代码来完成此事：

```
int status;
pid_t pid;

pid = waitpid (1742, &status, WNOHANG);
if (pid == -1)
        perror ("waitpid");
else {
        printf ("pid=%d\n", pid);

        if (WIFEXITED (status))
                printf ("Normal termination with exit status=%d\n",
                        WEXITSTATUS (status));

        if (WIFSIGNALED (status))
                printf ("Killed by signal=%d%s\n",
                        WTERMSIG (status),
                        WCOREDUMP (status) ? " (dumped core)" : "");
}
```

最后，请注意如下的 wait() 用法：

```
wait (&status);
```

等效于如下的 waitpid() 用法：

```
waitpid (-1, &status, 0);
```

以更灵活的方式等待

对于在“等待子进程”（waiting-for-children）功能性上需要更大灵活度的应用程序，可以使用 POSIX 的 XSI 扩展所定义（以及 Linux 所提供）的 waitid()：

```
#include <sys/wait.h>

int waitid (idtype_t idtype,
            id_t id,
            siginfo_t *infop,
            int options);
```

如同 wait() 和 waitpid()，waitid() 可用于等待并取得关于一个子进程的状态变更（终止、停止、继续）信息。waitid() 提供的选项比较多，但是复杂度较高。

如同 waitpid()，waitid() 允许开发者指定所要等待的是哪些子进程。然而

waitid() 并非使用一个参数来完成此任务，而是使用两个参数。idtype 与 id 参数用于指定所要等待的是哪些子进程，其所达成的目标如同 waitpid() 的 pid 参数所达成的。idtype 可以是下面的其中一个值：

P_PID
: 所要等待的是其进程 ID 与 id 相符的子进程。

P_GID
: 所要等待的是其进程组 ID 与 id 相符的子进程。

P_ALL
: 等待所有子进程，id 会被忽略。

id 参数使用了罕见的 id_t 数据类型，这代表它是一个通用的识别号码。如果未来的实现中加入新的 idtype 值，而且推测预定义的数据类型需要较大的空间来保存新创建的标识符，则可以使用此函数。此类型保证可以提供足够大的空间来保存任何 pid_t。在Linux 中，开发者可能会把它当成 pid_t 来使用 —— 例如，直接传入 pid_t 值或数值常量。然而，卖弄学问的程序设计者却会对它进行类型转换。

options 参数可以是一个或多个如下值的二元 OR 逻辑运算：

WEXITED
: 此调用将等待已终止的子进程（由 id 和 idtype 指定）。

WSTOPPED
: 此调用将等待已停止执行（为了响应所收到的一个信号）的子进程。

WCONTINUED
: 此调用将等待已继续执行（为了响应所收到的一个信号）的子进程。

WNOHANG
: 此调用不会受到阻挡，如果没有相符的子进程已经终止（或停止，或继续），则立即返回。

WNOWAIT
: 此调用不会将相符的进程从僵尸状态移除。因为将来可能需要等待该进程。

成功等到一个子进程时，waitid() 会将值填入 infop 参数，此值必须指向一个有效的 siginfo_t 类型。siginfo_t 结构的设置方式因实现而异（注 3），有些字段的值在 waitid() 调用之后才会有效。也就是说，成功的 waitid() 调用会将值填入以下字段：

注 3： 没错，在 Linux 上，siginfo_t 结构非常复杂。它的定义可以在 */usr/include/ bits/ siginfo.h* 找到。我们将在第九章详细探讨此结构。

si_pid
子进程的 pid。

si_uid
子进程的 uid。

si_code
此字段的可能值包括 CLD_EXITED、CLD_KILLED、CLD_STOPPED 以及 CLD_CONTINUED，分别代表子进程的终止、因信号而死亡、因信号而停止以及因信号而继续。

si_signo
设成 SIGCHLD。

si_status
如果 si_code 字段的值为 CLD_EXITED，则此字段的值就是子进程的结束码；否则，此字段的值就是送给子进程导致状态改变的信号的数目。

执行成功时，waitid() 会返回 0；发生错误时，waitid() 会返 -1 并且将 errno 设定成下面的其中一个值：

ECHLD
id 与 idtype 所指定的进程并不存在。

EINTR
options 未设定 WNOHANG，而且有一个信号导致执行停止。

EINVAL
options 参数或 id 参数与 idtype 参数的组合指定了无效的值。

waitid() 函数额外提供了 wait() 与 waitpid() 中找不到的有用语义。特别是由 siginfo_t 结构所能取回的信息非常有价值。然而，如果不需要此类信息，那么使用较简单的函数会更有意义，因为支持这些函数的系统比较广泛，所以可以被移植到更多非 Linux 的系统中。

BSD 要使用：wait3() 与 wait4()

waitpid() 源自 AT&T 的 System V Release 4，而 BSD 则走自己的路，它提供了两个函数，可用于等待一个进程改变状态：

```
#include <sys/types.h>
#include <sys/time.h>
#include <sys/resource.h>
```

```
#include <sys/wait.h>

pid_t wait3 (int *status,
             int options,
             struct rusage *rusage);

pid_t wait4 (pid_t pid,
             int *status,
             int options,
             struct rusage *rusage);
```

助记符 3 及 4 代表这两个函数分别是 wait() 的具有 3 个参数的版本及具有 4 个参数的版本。

如果不使用 rusage 参数，此函数的行为类似于 waitpid()。如下的 wait3() 调用：

```
pid = wait3 (status, options, NULL);
```

与如下的 waitpid() 调用等效：

```
pid = waitpid (-1, status, options);
```

而如下的 wait4() 调用：

```
pid = wait4 (pid, status, options, NULL);
```

与如下的 waitpid() 调用等效：

```
pid = waitpid (pid, status, options);
```

也就是说，wait3() 会等待任何子进程变更状态，而 wait4() 会等待 pid 参数所指定的子进程变更状态。options 参数的行为如同 waitpid() 的。

如稍早所述，这两个调用与 waitpid() 最大的不同在于 rusage 参数。如果此参数的值不是 NULL，则函数会填入一个指向 rusage 结构的指针。此结构提供了子进程的资源用量（resource usage）信息：

```
#include <sys/resource.h>

struct rusage {
        struct timeval ru_utime; /* user time consumed */
        struct timeval ru_stime; /* system time consumed */
        long ru_maxrss;    /* maximum resident set size */
        long ru_ixrss;     /* shared memory size */
        long ru_idrss;     /* unshared data size */
        long ru_isrss;     /* unshared stack size */
        long ru_minflt;    /* page reclaims */
        long ru_majflt;    /* page faults */
        long ru_nswap;     /* swap operations */
        long ru_inblock;   /* block input operations */
```

```
        long ru_oublock;  /* block output operations */
        long ru_msgsnd;   /* messages sent */
        long ru_msgrcv;   /* messages received */
        long ru_nsignals; /* signals received */
        long ru_nvcsw;    /* voluntary context switches */
        long ru_nivcsw;   /* involuntary context switches */
};
```

下一章我们将进一步探讨资源用量信息。

执行成功时，这两个函数会返回已改变状态进程的 pid；执行失败时，它们会返回 -1 并且将 errno 设定成适当的值，其值如同 waitpid() 所返回的错误值。

因为 POSIX 并未定义 wait3() 与 wait4()（注 4），所以最好仅在资源用量信息至关重要时才使用它们。尽管未被 POSIX 标准化，然而几乎每一种 Unix 系统均支持这两个调用。

启动并等待一个新进程

ANSI C 与 POSIX 定义了一个接口，可用于产生一个新进程并等待它的终止 —— 你可以将此视为同步进程的创建。如果一个进程产生了一个子进程并且立即等待它的终止，便可以使用此接口：

```
#define _XOPEN_SOURCE    /* if we want WEXITSTATUS, etc. */
#include <stdlib.h>

int system (const char *command);
```

system() 函数之所以会如此命名是因为同步进程调用又被称为“*shelling out to the system*”。system() 常用于运行一个简单的实用程序或 shell script，通常这么做的目的只是取得它的返回值。

system() 可用于调用 command 参数所提供的命令（包括任何额外的参数）。command 参数会被放在 */bin/sh -c* 参数之后，因此参数会被整体提供给 shell。

执行成功时，返回值就是命令的返回状态，如同 wait() 所提供的状态。因此，可通过 WEXITSTATUS 取得所执行命令的结束码。如果 */bin/sh* 本身的调用失败，则 WEXITSTATUS 所提供的值如同 exit(127) 所返回的值。因为被调用的命令也可能会返回 127，所以没有可靠的方法可用来检查这是否为 shell 本身所返回的错误状态。发生错误时，此调用会返回 -1。

注 4：　尽管 wait3() 曾被纳入最初的 Single UNIX Specification，但是自那以后就被移除了。

如果 command 的值为 NULL，而且有 */bin/sh* 可用，则 system() 会返回一个非零值，否则会返回 0。

命令执行期间，SIGCHLD 会被阻挡，而 SIGINT 和 SIGQUIT 会被忽略。忽略 SIGINT 和 SIGQUIT 会有几个问题，特别是，如果 system() 于一个循环中被调用的话。如果在循环中调用 system()，你应该确保程序会妥善检查子进程的结束状态。例如：

```
do {
        int ret;

        ret = system ("pidof rudderd");
        if (WIFSIGNALED (ret) &&
            (WTERMSIG (ret) == SIGINT ||
             WTERMSIG (ret) == SIGQUIT))
                break; /* or otherwise handle */
} while (1);
```

system() 函数可以使用 fork()、一个 exec 系列函数以及 waitpid() 来实现。这是一个很有用的练习，你应该自己试着做一做，因为这必须将本章所提到的许多概念连接在一起。然而基于完整性，此处提供了一个简单的实现：

```
/*
 * my_system —— 同步产生并等待命令
 * "/bin/sh -c <cmd>"
 *
 *  发生任何错误时返回 -1，执行成功时返回所启动进程的结束码。
 *  不阻挡或忽略任何信号
 */
int my_system (const char *cmd)
{
        int status;
        pid_t pid;

        pid = fork ();
        if (pid == -1)
                return -1;
        else if (pid == 0) {
                const char *argv[4];

                argv[0] = "sh";
                argv[1] = "-c";
                argv[2] = cmd;
                argv[3] = NULL;
                execv ("/bin/sh", argv);

                exit (-1);
        }

        if (waitpid (pid, &status, 0) == -1)
                return -1;
```

```
        else if (WIFEXITED (status))
                return WEXITSTATUS (status);

        return -1;
}
```

请注意，不同于正式的 system()，此范例程序不会阻挡或禁用任何信号。此行为是好是坏，取决于你的程序的情况，但是至少不阻挡 SIGINT 通常是聪明的做法，因为这让被调用的命令可以被一般用户所预期的方式给中断。一个好的实现会加入额外的指针作为参数，当它非 NULL 时，可区分出当前所发生的是何种错误。例如，它可以加入 fork_failed 和 shell_failed。

僵尸进程

如同稍早所讨论的，一个已经终止的进程如果尚未被其父进程检查过状态，则称为“僵尸进程”（zombie process）。僵尸进程会继续消耗系统资源，尽管它只会占用比例很小的系统资源，但是足以维护进程原本的骨架。有这些资源被保留着，想检查其子进程状态的父进程才有办法取得与这些进程的生和死有关的信息。一旦父进程这么做之后，内核便会清除僵尸进程。

然而，任何长时间使用 Unix 的人肯定都曾看过无所事事的僵尸进程。这些进程，常称为幽灵进程（*ghost*），肇因于不负责任的父进程。如果你的应用程序派生了一个子进程，你的应用程序就有责任（除非它很短命，正如你马上会看到的）等待其子进程，即使它仅仅是丢弃所收集到的信息。否则，你的进程的所有子进程将成为无所事事的幽灵进程，挤满系统的进程列表，并让用户对你的应用程序的草率实现感到不满。

然而，如果发生父进程在子进程之前死亡（终止），或者如果在它死亡之前有机会等待它的僵尸子进程呢？每当有一个进程终止时，Linux 内核会把它的所有子进程重新指派给 init 进程（pid 值为 1 的进程）。这样可确保每个进程都会有一个直接的父进程。于是，init 进程会周期性地等待它的所有子进程，这样可确保僵尸进程不会停留太久的时间 —— 不会成为幽灵进程！因此，如果一个父进程在子进程之前死亡，或者没有在结束之前等待它的子进程，那么这些子进程最后会被重新指派给 init 进程而由它负责等待，这让它们全部可以结束。此做法被认为是一个好的实践，有了这个保障，意味着短命的进程不用太担心它的所有子进程。

用户与组

如本章稍早所述以及第一章所做的讨论，进程会被关联到用户与组。用户与组ID的值可分别用 C 数据类型 uid_t 与 gid_t 来表示。这种将数值形式的标识符映射至人类可

变更有效的用户或组 ID

Linux 为当前执行的进程的有效用户与组 ID 的设定提供了两个 POSIX 所规定的函数：

```
#include <sys/types.h>
#include <unistd.h>

int seteuid (uid_t euid);
int setegid (gid_t egid);
```

seteuid() 可用于将有效用户 ID 的值设定成 euid。root 用户可以替 euid 提供任何值，非 root 用户只可以将有效用户 ID 的值设定成真实的或被保存的用户 ID。执行成功时，seteuid() 会返回 0；执行失败时，它会返回 -1 并且把 errno 设定成 EPERM，这表示当前进程的拥有者并非 root，而且 euid 既不等于真实用户 ID 也不等于被保存的用户 ID。

请注意，在用户不是 root 的情况下，seteuid() 的行为如同 setuid()。因此标准的做法以及比较好的主意总是使用 seteuid()，除非你的进程企图以 root 的身份执行，那么此时使用 setuid() 会更有意义。

前面的讨论同样适用于组 —— 仅需将 seteuid() 替换成 setegid()以及将 euid 替换成 egid。

以 BSD 的手法变更用户和组 ID

BSD 针对用户与组ID的设定提供了它自己的接口。基于兼容性，Linux 也提供了这些接口：

```
#include <sys/types.h>
#include <unistd.h>

int setreuid (uid_t ruid, uid_t euid);
int setregid (gid_t rgid, gid_t egid);
```

setreuid() 可用于将一个进程的真实的以及有效的用户ID分别设定成 ruid 与 euid。若将任何参数值设定成 -1，则相应的用户 ID 会维持不变。非 root 进程只能将有效用户 ID 设定成真实的或被保存的用户 ID，以及将真实用户 ID 设定成有效用户 ID。如果真实用户 ID 有所变动，或者如果有效用户 ID 的值被变更成不等于之前真实用户 ID 的值，则被保存的用户 ID 的值会变更成新的有效用户 ID。至少，那就是 Linux 以及多数其他 Unix 系统响应此类变动的方式。POSIX 并未定义此行为。

执行成功时，setreuid() 会返回 0；执行失败时，它会返回 -1 并且将 errno 设定成 EPERM，这表示当前进程的拥有者并非 root，而且 euid 既不等于真实用户 ID 也不等于被保存的用户 ID，或者 ruid 不等于有效用户 ID。

前面的讨论同样适用于组 —— 仅需将 setreuid() 替换成 setregid()、将 ruid 替换成 rgid以及将 euid 替换成 egid。

以 HP-UX 的手法变更用户和组 ID

你可能会觉得情况越来越混乱，就连 HP-UX（Hewlett-Packard 的 Unix 系统）也提供了自己的机制来设定一个进程的用户与组 ID。Linux 随后也提供了这些接口：

```
#define _GNU_SOURCE
#include <unistd.h>

int setresuid (uid_t ruid, uid_t euid, uid_t suid);
int setresgid (gid_t rgid, gid_t egid, gid_t sgid);
```

setresuid() 可用于将真实的、有效的以及被保存的用户ID分别设定成 ruid、euid 以及 suid。若将任何参数值设定成 -1，则相应的用户 ID 会维持不变。

root 用户可以将任何的用户 ID 设定成任何值，非 root 用户可以将任何的用户 ID 设定成当前的真实的、有效的或被保存的用户 ID。执行成功时，setuid() 会返回 0；发生错误时，此调用会返回 -1 并且将 errno 设定成下面的其中一个值：

EAGAIN

uid 与真实用户 ID 不符，而且将真实用户 ID 设定成 uid，这会使得用户逾越其 NPROC rlimit（用于指定一个用户所能拥有的进程的数目）。

EPERM

用户不是 root，而且企图将真实的、有效的或被保存的用户 ID 设定成与当前的真实的、有效的或被保存的用户 ID 不符的值。

前面的讨论同样适用于组 —— 仅需将 setresuid() 替换成 setresgid ()、将 ruid 替换成 rgid、将 euid 替换成 egid 以及将 suid 替换成 sgid 。

首选的用户／组 ID 操作

非 root 进程应该使用 seteuid() 来变更它们的有效用户 ID。root 进程如果希望变更这三种用户 ID，则应该使用 setuid()；如果只希望暂时变更有效用户 ID，则应该使用 seteuid()。这些函数不仅简单，而且行为与 POSIX 的规定相符，也适度考虑到了被保存的用户 ID。

尽管提供了额外的功能，BSD 以及 HP-UX 风格的函数并不允许进行 setuid() 与 seteuid() 无法做的任何有用变更。

支持被保存的用户 ID

被保存的用户与组 ID 定义于 IEEE Std 1003.1-2001（POSIX 2001），而且 Linux 自从早期的 1.1.38 版内核便支持这些 ID。专为 Linux 而编写的程序不用担心被保存的用户 ID 是否存在。为较旧的 Unix 系统所编写的程序在对被保存的用户与组 ID 进行任何引用之前，应该先检查宏 `POSIX_SAVED_IDS`。

在被保存的用户与组ID不存在的情况下，前面的讨论依然有效，只要把提及被保存的用户与组 ID 的规则的任何部分忽略掉即可。

取得用户和组 ID

下面这两个系统调用会分别返回真实的用户与组 ID：

```
#include <unistd.h>
#include <sys/types.h>

uid_t getuid (void);
gid_t getgid (void);
```

这两个系统调用不会失败。同样地，下面这两个系统调用会分别返回有效的用户与组 ID：

```
#include <unistd.h>
#include <sys/types.h>

uid_t geteuid (void);
gid_t getegid (void);
```

这两个系统调用也不会失败。

会话与进程组

每个进程都属于一个进程组（*process group*），这个集合里的一个或多进程与其他进程的关系一般是出于作业控制（*job control*）的目的。进程组的主要属性是信号可以被传送给组中所有进程：单一动作可以终止（terminate）、停止（stop）或继续同一个进程组中的所有进程。

每个进程组都会有独一无二的进程组 ID（*process group ID*，常简写为 pgid），而且都会具有一个进程组首领进程（*process group leader*）。进程组 ID 等于进程组首领进程的 pid。只要进程组中还有成员存在着，进程组就会一直存在着。即使进程组首领进程终止，进程组仍会继续存在着。

当一个新用户首次登录一台机器时，登录进程（login process）会创建一个新的会话（*session*），此会话由单一进程组成，也就是用户的 *login shell*。login shell 的作用为会话的首领进程（*session leader*）。会话的首领进程的 pid 可作为会话 ID（*session ID*）用。一个会话由一个或多个进程组组成。会话会替一个已登录的用户安排活动，并且将该用户关联到一个控制终端（一个特定的 tty 设备，用于处理用户的终端 I/O）。因此，会话主要与 shell 的业务有关。事实上，其他的业务与会话并无直接的关联。

进程组提供了一个机制，让信号可以传送给进程组的所有成员，这使得工作控制以及其他 shell 功能变得简单了很多，会话存在的目的在于结合登录进程与控制终端。在一个会话中，进程组会被划分成一个前台进程组（*foreground process group*）以及零个或多个后台进程组（*background process group*）。当一个用户离开一个终端时，会有一个 `SIGQUIT` 信号被传送给前台进程组中的所有进程。当一个终端侦测到网络断线时，会有一个信号 `SIGHUP` 被传送给前台进程组中的所有进程。当用户键入中断键（通常是 Ctrl+C），会有一个 `SIGINT` 信号被传送给前台进程组中的所有进程。因此，会话让终端以及 login shell 的管理变得更简单。

举例来说，假设有一个用户登录系统以及她的 login shell（*bash*），其 pid 为 1700。此用户的 *bash* 实例现在是一个新的进程组（它的进程组 ID 为 1700）的唯一成员和首领进程。此进程组现在被包含在一个新的会话（它的会话 ID 为 1700）之内，而且 *bash* 是此会话的唯一成员以及首领进程。用户在 shell 中所执行的新命令会运行在会话 1700 之内的新进程组中。这些进程组中只会有一个前台进程组（也就是一个直接连接到用户并且掌控终端的进程组），其余皆为后台进程组。

通常一个系统上会存在许多会话：每个用户的登录会话，以及包含与用户登录会话无关的进程的会话（如守护进程）。守护进程（daemon）往往会创建自己的会话，以避免所关联到的其他会话结束所导致的问题。

每一个会话都会包含一个或多个进程组，而且每一个进程组至少会包含一个进程。包含了多个进程的进程组通常用于实现作业控制。

shell 之上若键入了下面这道命令：

```
$ cat ship-inventory.txt | grep booty | sort
```

则会产生一个包含三个进程的进程组。这样，shell 可以同时发信号通知这三个进程。因为用户在控制台（console）上键入这道命令的时候并未加上结尾的“&”符号，所以我们可以说此进程组位于前台。图 5-1 可以看到会话、进程组、进程与控制终端之间的关系。

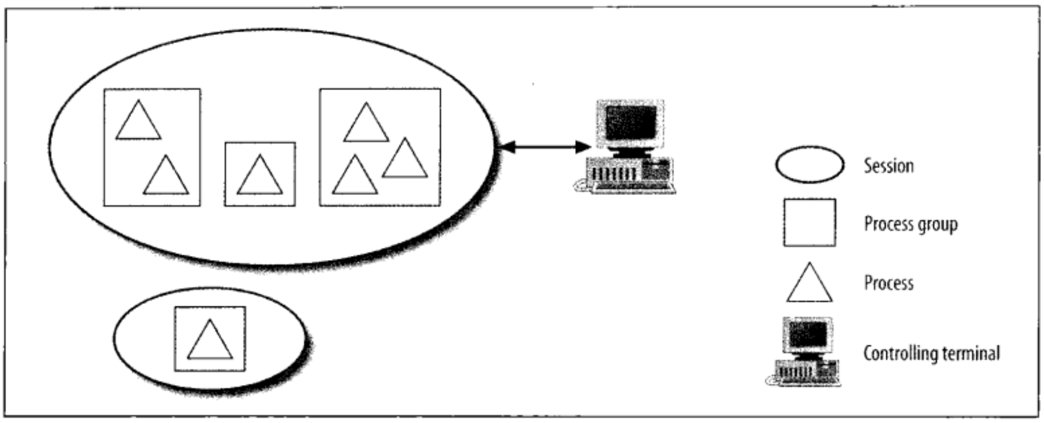


图 5-1:会话、进程组、进程与控制终端之间的关系

Linux 提供了若干接口用于设定与取得跟所指定进程有关的会话以及进程组。这些接口主要是供 shell 所使用，但是也可以用在想要摆脱会话与进程组的业务的进程中，例如守护进程（daemon）。

会话系统调用

shell 会在登录的时候创建新的会话。它们通过一个特殊的系统调用完成此工作：

```
#include <unistd.h>

pid_t setsid (void);
```

在进程尚不是一个进程组首领进程时，调用 setsid() 可创建一个新的会话。calling process（进行调用的进程）将成为新会话的首领进程以及此会话的唯一成员，此会话并没有控制终端。此调用还会在此会话中创建一个新的进程组，并且让 calling process 成为进程组首领进程以及此进程组的唯一成员。新会话与进程组 ID 会被设定成 calling process 的 pid。

换言之，setsid() 会在一个新的会话中创建一个新的进程组，而且让 calling process 成为这两者的首领进程。这对不想成为现有会话成员或者不想控制终端的守护进程而言很有用，而且对想替每个登录系统的用户创建一个新会话的 shell 而言也很有用。

执行成功时，setsid() 会返回所创建会话的会话 ID；发生错误时，此调用会返回 -1。errno 只有一个可能值 EPERM，代表此进程当前已经是一个进程组首领进程。最简单的做法就是确定所要派生的进程不是一个进程组首领进程，让父进程终止以及让子进程执行 setsid()。例如：

```
pid_t pid;

pid = fork ();
if (pid == -1) {
        perror ("fork");
        return -1;
} else if (pid != 0)
        exit (EXIT_SUCCESS);

if (setsid () == -1) {
        perror ("setsid");
        return -1;
}
```

也可以取得当前的会话 ID，尽管用处不大：

```
#define _XOPEN_SOURCE 500
#include <unistd.h>

pid_t getsid (pid_t pid);
```

getsid() 会返回 pid 所指定的进程的会话 ID。如果 pid 参数为 0，则此调用会返回 calling process 的会话 ID。发生错误时，此调用会返回 -1。errno 只有一个可能值 ESRCH，代表 pid 并未对应到一个有效的进程。请注意，其他的 Unix 系统还会将 errno 设定成 EPERM，代表 pid 与 calling process 并不属于同一个会话；Linux 并不会返回此错误代码，而会很愉快地返回任何进程的会话 ID。

此调用很少被用到，主要用于诊断：

```
pid_t sid;

sid = getsid (0);
if (sid == -1)
        perror ("getsid"); /* 应该不可能 */
else
        printf ("My session id=%d\n", sid);
```

进程组系统调用

setpgid() 会将 pid 所指定的进程的进程组 ID 设定成 pgid：

```
#define _XOPEN_SOURCE 500
#include <unistd.h>

int setpgid (pid_t pid, pid_t pgid);
```

如果 pid 的值为 0，则会使用当前的进程；如果 pgid 的值为 0，则 pid 所指定进程的进程 ID 会被当成进程组 ID。

执行成功时，setpgid() 会返回 0。执行成功代表符合以下几个条件：

- pid 所指定的进程必须是 calling process（进行调用的进程），或者是 calling process 的一个未调用 exec 的子进程而且与 calling process 皆隶属同一个会话。
- pid 所指定的进程不可以是一个会话的首领进程。
- 如果 pgid 已经存在，它必须与 calling process 皆隶属同一个会话。
- pgid 不可以是负值。

发生错误时，此调用会返回 -1，而且会将 errno 设定成下面的其中一个错误代码：

EACCESS
: pid 所指定的进程是 calling process 的一个已经调用 exec 的子进程。

EINVAL
: pgid 的值小于 0。

EPERM
: pid 所指定的进程是一个会话的首领进程，或者与 calling process 分属不同会话，亦或者企图将一个进程移往位于不同会话的进程组。

ESRCH
: pid 不是当前进程 0，或者不是当前进程的一个子进程。

如同会话，尽管可以取得一个进程的进程组 ID，但是用处不大：

```
#define _XOPEN_SOURCE 500
#include <unistd.h>

pid_t getpgid (pid_t pid);
```

getpgid() 会返回 pid 所指定进程的进程组 ID。如果 pid 的值为 0，则会使用当前进程的进程组 ID。发生错误时，它会返回 -1 并且将 errno 设定成 ESRCH，这是唯一的可能值，表示 pid 是一个无效的进程 ID。

如同 getsid()，getpgid() 主要用于诊断：

```
pid_t pgid;

pgid = getpgid (0);
if (pgid == -1)
        perror ("getpgid"); /* 应该不可能 */
else
        printf ("My process group id=%d\n", pgid);
```

过时的进程组函数

Linux 支持了两个来自 BSD 的较旧型的接口，用于操纵或取得进程组 ID。因为它们不如先前所探讨的系统调用有用，所以新的程序只应该在迫切需要移植的时候使用它们。setpgrp() 可用于设定进程组 ID：

```
#include <unistd.h>

int setpgrp (void);
```

如下的调用：

```
if (setpgrp () == -1)
        perror ("setpgrp");
```

等效于如下的调用：

```
if (setpgid (0,0) == -1)
        perror ("setpgid");
```

以上两种调用会试图把当前进程指派给数值与当前进程 pid 相同的进程组，调用成功会返回 0，调用失败会返回 -1。setpgid() 的 errno 的所有可能值皆可应用于 setpgrp()(但是 ERSCH 除外)。

同样地，调用 getpgrp() 可用于取得进程组 ID：

```
#include <unistd.h>

pid_t getpgrp (void);
```

如下的调用：

```
pid_t pgid = getpgrp ();
```

等效于：

```
pid_t pgid = getpgid (0);
```

它们都会返回 calling process 的进程组 ID。函数 getpgid() 不会失败。

守护进程

守护进程（*daemon*）是一个在后台运行的进程，它不会连接到任何控制终端。守护进程一般会于引导时启动，并且会以 root 或其他特殊用户（例如 *apache* 或 *postfix*）的身份运行，而且所处理的是系统层的任务。按照惯例，一个守护进程的名称通常会以字母 *d* 结尾（例如 *crond* 以及 *sshd*），但是这并非必要的，甚至不具有普遍性。

此名称源自 Maxwell's demon（麦克斯韦尔的小妖），这是物理学家 James Maxwell 于 1867 年所提出的一个想法。daemon（精灵）还是希腊神话中超自然的存在，它存在于 [illegible]（恶魔），希腊式 daemon（精灵）并不必然是邪恶的。的确，在希腊神话中，daemon 往往会帮奥林匹斯山众神完成它们不愿意做的事——这与 Unix 的 daemon（守护进程）为前台用户完成其想要避免的工作非常像。

一个守护进程通常需要符合两项需求：它必须是 init 的一个子进程，而且它不得连接到一个终端。

总之，一个程序完成以下步骤就可以变成一个 daemon：

1. 调用 fork()。这会创建一个新的进程，这个新进程将变成 daemon。
2. 在父进程中调用 exit()。这样可以确保原始的父进程（daemon 的祖父进程）的需求得到满足：其子进程会被终止、daemon 的父进程不再运行以及 daemon 并非一个进程组的首领进程。最后一点是下个步骤可以成功完成的必要条件。
3. 调用 setsid()，替 daemon 创建一个新的进程组以及会话，这两者均会以 daemon 为首领进程。这样也可以确保该进程不会被关联到控制终端（因为该进程只会创建一个新的会话，而不会指派既有的会话）。
4. 使用 chdir() 将工作目录切换至根目录。这么做是因为所继承的工作目录可以位于文件系统上的任何位置。daemon 通常会在系统正常运行期间运行，而且你不会想保留某个随机打开的目录，因此可避免管理者卸载（unmount）包含该目录的文件系统。
5. 关闭所有文件描述符。你不会想继承已打开的文件描述符并让它们持续打开。
6. 打开文件描述符 0、1 与 2（标准输入、标准输出与标准错误）并将它们重定向至 */dev/null* 。

根据以上规则，我们可以让一个程序本身变成守护进程：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <linux/fs.h>

int main (void)
{
        pid_t pid;
```

```
        int i;

        /* 创建新的进程 */
        pid = fork ();
        if (pid == -1)
                return -1;
        else if (pid != 0)
                exit (EXIT_SUCCESS);

        /* 创建新的会话以及进程组 */
        if (setsid () == -1)
                return -1;

        /* 将工作目录设定成根目录 */
        if (chdir ("/") == -1)
                return -1;

        /* 关闭所有已打开的文件 —— NR_OPEN 有点小题大做，但是可行 */
        for (i = 0; i < NR_OPEN; i++)
                close (i);

        /* 将 fd 的 0、1、2 重定向至 /dev/null */
        open ("/dev/null", O_RDWR);      /* stdin */
        dup (0);                         /* stdout */
        dup (0);                         /* stderror */

        /* 做守护进程该做的事……*/

        return 0;
}
```

多数 Unix 系统会在它们的 C 链接库中提供一个 daemon() 函数，以便自动进行这些步骤，化繁为简：

```
#include <unistd.h>

int daemon (int nochdir, int noclose);
```

如果 nochdir 为非零值，则守护进程不会把它的工作目录切换成根目录。如果 noclose 为非零值，则守护进程不会关闭所有已打开的文件描述符。如果父进程已经设置了这些成为守护进程的过程，这些选项将会很有用。然而，一般会对这两个参数传入 0。

执行成功时，此调用会返回 0；执行失败时，此调用会返回 -1，而且 errno 会被设定成一个来自 fork() 或 setsid() 的有效错误代码。

结束语

我们在本章说明了 Unix 进程管理的基本功能，从进程的创建到进程的终止均涵盖在内。下一章，我们将说明较高级的进程管理接口以及用于变更进程调度行为的接口。

何时运行以及运行多久是进程调度程序的基本责任。

对配备单处理器的机器而言，一个多任务操作系统（multitasking operating system）能够交错执行一个以上的进程，这会给人一个错觉，仿佛有多个进程同时运行。对配备了多处理器的机器而言，一个多任务操作系统允许多个进程实际地在不同的处理器上同时运行。对一个非多任务的操作系统而言，例如 DOS，同一时间内只能运行一个应用程序。

多任务操作系统又分成两类：协同式（*cooperative*）以及抢占式（*preemptive*）。Linux 所实现的是抢占式多任务机制，在此多任务机制中，调度程序会决定何时可以停止一个进程的运行，让不同的进程可以继续运行。我们将暂停一个运行中的进程而让另一个进程继续运行的行为称为抢占（*preemption*）。此外，在调度程序抢占一个进程之前，该进程所能运行的时间长短就称为处理器的时间片（之所以会称为时间片，是因为调度程序会将处理器的每个时间片段分配给每个可运行的进程）。

反过来说，在协同式多任务机制中，一个进程不会停止运行，除非该进程自愿这么做。我们将一个进程自愿暂停自己的行为称为让出（*yielding*）CPU。理想情况下，进程会经常让出 CPU，但是操作系统并不会强迫进程这么做。一个粗略或不完整的程序可能会运行比较久的时间（相较于经过优化的程序），甚至是拖累整个系统。由于此做法有缺点，所以现代的操作系统几乎普遍采用抢占式多任务的做法，Linux 也不例外。

2.5 版内核系列所引进的 *O(1)* 进程调度程序是 Linux 调度的关键（注 1）。Linux 调度算法除了提供抢占式多任务，还支持多处理器、处理器亲和性（processor affinity）、非一致内存访问（nonuniform memory access、常简写成 NUMA）配置、多线程、实时进程以及用户提供优先级（user-provided priority）等功能。

大 O 符号

O(1) —— 念成“big oh of one”（1 的大 O），是大 O 符号（*big-oh notation*）的一个示例，用于表示一个算法的复杂度（complexity）以及可伸缩度（scalability）。它的正式定义为：

$$\text{If } f(x) \text{ is } O(g(x)),$$
$$\text{then}$$
$$\exists c, x' \text{ such that } f(x) \leqslant c.g(x),\ \forall x > x'$$

注 1： 好奇的读者可以在内核源码树中的 *kernel/sched.c* 找到进程调度程序的定义。

它的意思是说，某一算法 f 的值总是小于或等于 g 的值乘以某一任意常量，只要输入的 x 大于某一初始值 x'。也就是说，g 跟 f 一样大，或者大于 f；g 是 f 的上限。

因此，O(1) 意味着算法的复杂度小于某个常量 c。此情况可归纳出一个重要的结论：Linux 进程调度程序的执行时间为常量，与系统上的进程数目无关。这很重要，因为选出可运行的新进程显然涉及了至少一次（如果不是多次）的进程列表迭代。使用较简单的调度程序（包括 Linux 早期版本所使用的那些），当系统上的进程数目增长时，此类迭代（iteration）很快就会增长成一个可能的瓶颈。此类循环（loop）至多会替进程的调度带来不可确定性。

Linux 调度程序的运作时间为常量，不受任何因素的影响，没有这样的瓶颈问题。

时段

Linux 分配给每个进程的时间片（timeslice）在系统的整体行为和性能中是一个重要的变量。如果时间片太大，进程在两次执行之间必须等待一段很长的时间，这会使得并行执行出现的机会降到最小。用户可能会对显著的延迟感到沮丧。反过来说，如果时间片太小，则会有大量的系统时间被花在从一个应用程序切换到另一个应用程序上，而且会丧失 temporal locality（时间局部性）之类的好处。

因此，决定一个理想的时间片值并不容易。有些操作系统会给进程提供较大的时间片，希望能够最大化系统的吞吐量以及整体的性能。有些操作系统会给进程提供非常小的时间片，希望能够为一个系统提供绝佳的交互性能。正如我们将看到的，Linux 的目标是通过动态分配进程的时间片，在这两方面达到最佳的效果。

请注意，一个进程不需要一次用完它的所有时间片。一个进程被指派了 100 ms 的时间片，可能会先运行 20 ms，然后等待某个资源，例如键盘的输入。调度程序会暂时将此进程从可运行进程列表中移除。当所等待的资源变为可用时 —— 此处是指键盘缓冲器变为非空白，调度程序会唤醒此进程。然后此进程可以继续运行，直到它用完时间片的其余 80 ms，或者直到它再次等待某个资源。

I/O 密集型与处理器密集型进程

会继续耗用所有可用时间片的进程称为处理器密集型（*processor-bound*）。此类进程极度渴望 CPU 时间，而且会用完调度程序指派给它的所有时间片。最简单的例子就是一个无穷循环，其他的例子包括科学计算、数学计算以及图像处理。

另一方面，会花比较多的时间等待资源的进程称为 I/O 密集型（*I/O-bound*）。I/O 密集型进程常会发出并等待文件 I/O、等待键盘的输入或者等待用户移动鼠标。I/O 密集型应用程序的例子包括只会送出系统调用请求内核执行 I/O 的实用程序，例如 *cp* 或 *mv*，以及将许多时间花在等待用户输入的 GUI 应用程序。

处理器密集型与 I/O 密集型应用程序的差异在于，调度程序会以对它们最有利的方式来调整调度的行为。处理器密集型应用程序渴望尽可能获得最大的时间片，这让它们可以最大化缓存区的命中率（通过 temporal locality），以及让它们的工作能够尽快完成。相对而言，I/O 密集型进程不一定需要较大的时间片，因为在它们送出 I/O 请求以及等待某个内核资源之前，它们一般会运行一段非常短的时间。然而，I/O 密集型进程将得利于调度程序的持续关注（constant attention）。如果受阻挡以及派送多个 I/O 请求之后此类应用程序可以越快重新开始，则它越能善加利用系统的资源。此外，如果应用程序需要等待用户的输入，则它也是越快被调度，于是用户对执行未间断的感受也就越大。

同时满足处理器密集型以及 I/O 密集型进程的需要并不容易。Linux 调度程序会试图找到并优待 I/O 密集型应用程序：I/O 密集型应用程序的优先级会被提升，而处理器密集型应用程序的优先级则会被降低。

事实上，多数应用程序既是 I/O 密集型也是处理器密集型。音频与视频的编码与译码便是一个好例子。许多游戏也是这种情况。所以不一定总是能够辨识出特定应用程序的倾向，而且任一时间点上，特定进程的行为可能是 I/O 密集型也可能是处理器密集型。

抢占式调度

当一个进程把它的时间片用完时，内核会暂停该进程并着手运行一个新的进程。如果系统上没有可运行的进程，则内核会取得一组已将时间片用完的进程，重新补足它们的时间片并再次着手运行它们。这样，所有进程最终都得以运行，即使系统上有优先级较高的进程 —— 优先级较低的进程只需等待优先级较高的进程用完它们的时间片或者受到阻挡。此行为明确描述了一个重要但心照不宣的 Unix 调度规则：所有的进程都必须前进。

如果系统上并不存在任何可运行的进程，内核会运行空闲进程（*idle process*）。空闲进程事实上根本不是一个进程，它实际上也不会运行（以便节省电池的电力）。事实上，空闲进程是内核所执行的一个特殊例程，用以简化调度程序算法并让进程记账变得更容易。闲时间（idle time）只是用来运行空闲进程的时间。

如果有一个正在运行的进程，而且有一个优先级较高的进程变为可运行的（或许是因为它正在等待键盘的输入，而且用户刚好键入了一个字），于是内核会立即暂停当前正在运行的进程，并且着手运行优先级较高的进程。因此，优先级高于当前正在运行进程的

进程绝不可能进入可运行但未运行的（runnable-but-not-running）状态。正在运行的进程往往是系统中优先级最高的可运行进程。

线程

线程（thread）是单一进程内部的基本执行单位。所有进程至少都会有一个线程。每个线程都会有它自己的虚拟处理器：它自己拥有一套寄存器、指令指针以及处理器状态。尽管多数进程都只具有一个线程，不过进程也可以具有大量的线程，虽然它们各自执行不同的任务，但是共享相同的地址空间（因而共享相同的动态内存、所映射的文件、目标码等）、所打开的文件以及其他内核资源。

Linux 内核对线程具有一种感兴趣且独特的看法。基本上，内核并没有这样的概念。对 Linux 内核而言，所有的线程都是一个独特的进程。广泛来看，两个无关的进程与单一进程中的两个线程之间并无差别。内核只会将线程视为共享资源的进程。也就是说，内核会将由两个线程所组成的进程视为两个共享同一组内核资源（地址空间、所打开的文件等）的进程。

多线程程序设计（*multithreaded programming*）是一门以线程进行程序设计的技术。Linux 上用于以线程进行程序设计的最常见 API 就是经过 IEEE Std 1003.1c-1995（POSIX 1995 或 POSIX.1c）标准化的 API。开发者常会调用实现了此 API 的链接库 *pthreads*。以线程进行程序设计是一个复杂的主题，而且 pthreads API 既大且复杂。因此，pthreads 并不在本书讨论之列。然而，本书会把焦点放在用来建立 pthreads 链接库的接口上。

让出处理器

尽管 Linux 是一个抢占式多任务操作系统，不过它也提供了一个系统调用，允许进程主动让出执行权，以及指示调度程序选出并运行一个新的进程：

```
#include <sched.h>

int sched_yield (void);
```

调用 `sched_yield()` 的结果是暂停当前正在运行的进程，于是进程调度程序会选出并运行一个新的进程，这就好像是内核自己抢占当前正在运行的进程以便执行一个新的进程。请注意，如果不存在任何其他的可运行进程，这是经常发生的情况，则让出执行权的进程会立即继续执行下去。由于有这种不确定性，再加上一般普遍认为尽管普遍存在更好的选择，所以此系统调用的使用并不普遍。

执行成功时，此调用会返回 0；执行失败时，它会返回 -1 并且将 errno 设定成适当的错误代码。在 Linux 上 —— 而且有很大的可能包括多数其他的 Unix 系统 —— sched_yield() 不能执行失败，因此总是会返回 0。然而，一丝不苟的程序设计者或许仍会检查返回值：

```
if (sched_yield ())
        perror ("sched_yield");
```

合法的用途

实际使用中，sched_yield() 在适当的抢占式多任务操作系统（例如 Linux）之上具有若干（如果有的话）合法的用途。内核完全有能力作出理想和最有效的调度决策 —— 当然，相较于个别的应用程序，内核有更好的装备可用以决定如何以及何时进行抢占。这也正是为什么操作系统会放弃协同式多任务而支持抢占式多任务。

那么，为什么我们会有一个“重新调度我”（reschedule me）系统调用可用？答案在于应用程序必须等待外部事件（也许是由用户、一个硬件组件或者另一个进程所造成）。举例来说，如果一个进程需要等待另一个进程，则“让出处理器直到另一个进程完成”是第一个阶段的解决方案。再举一个例子，在 consumer/producer（消费者 / 生产者）机制中，一个简单的 consumer 实现可能会类似下面这个样子：

```
/* the consumer... */
do {
        while (producer_not_ready ())
                sched_yield ();
        process_data ();
} while (!time_to_quit ());
```

值得庆幸的是，Unix 设计者不会动辄写出这样的程序代码。Unix 程序通常是事件驱动的（event-driven）而且倾向于在 consumer 与 producer 之间运用某种可阻挡的机制（例如一个管道）来代替 sched_yield()。在此情况下，consumer 读取管道并在需要时被阻挡，直到数据可供读取。接着，当新数据变为可用时，producer 会将数据写入管道。这会将协调的责任从用户空间进程移往内核，用户空间进程只需要进入忙碌循环（busy-loop），而内核则可以通过让进程进入休眠状态以及仅在有需要时唤醒它们的方式让此情况的管理得以优化。总之，Unix 程序应该把目标放在依赖可阻挡文件描述符（blockable file descriptors）的事件驱动解决方案上。

然而，有一种情况需要用到 sched_yield()：用户空间线程锁定（user-space thread locking）。如果一个线程企图取得另一个线程所持有的锁，则新的线程应该让出处理器直到该锁变为可用。用户空间锁没有内核的支持，这是一个最简单、最有效率的做法。

值得庆幸的是，现代的 Linux 线程实现（New POSIX Threading Library 或简写成 NPTL）引入了一个使用 *futexes*（为用户空间锁提供内核的支持）的优化解决方案。

另一种情况则把 sched_yield() 用得“恰到好处”（playing nicely）：一个处理器密集型程序可以周期性地调用 sched_yield()，试图将它对系统的冲击减到最小。尽管有崇高的理想，但是它的策略有两个缺点。首先，内核能够进行全局的调度决策，这远胜于个别进程所能做的，因此确保系统顺利运作应该是进程调度程序的责任，而不是进程的责任。为此提出了调度程序的交互奖惩值（bonus），目的是为了奖赏 I/O 密集型应用程序以及惩罚处理器密集型应用程序。其次，减轻一个处理器密集型应用程序的开

相对优先值。 （本章稍后会讨论）为应用程序的性能设定

的进程，则内核将切换到该进程，而且会把进行调用的进程（invoking 另一个可运

说：“不！请安排其他进程！”此状况会持续到这两个进程全都用完自己的时间片。如果我们把进程调度程序选取进程的情况绘制成图表，会看到类似“A、B、A、B、A、B”这样的现象 —— 因此会以“乒乓球”为名。

新的行为可避免此现象。进程 A 要求让出处理器，调度程序就会把它从可运行进程列表中移除。同样地，进程 B 作出相同的请求，调度程序就会把它从可运行进程列表中移除。调度程序不会去运行进程 A 或行程 B，直到不存在任何其他的可运行进程，这样就可避免“乒乓球”现象，并让其他进程公平分享处理器时间。

因此，如果一个进程要求让出处理器，该进程就应该真的把处理器让出来！

进程优先级

> 这一节所探讨的内容适合正常、非实时的进程。实时进程需要使用不同的调度规则以及不同的优先级系统。本章稍后会探讨实时系统的相关议题。

Linux 不会随意对进程进行调度。事实上，应用程序会被指派优先级，优先级会对进程何时运行以及运行多久造成影响。以往，Unix 将优先级称为友善值（*nice value*），因为友善值背后的概念是通过调低一个进程的优先级来“善待”系统上的其他进程，这让其他进程可以使用较多的处理器时间。

友善值可控制一个进程何时运行。Linux 会按照优先级（从最高到最低）的顺序来对可运行进程进行调度：优先级较高的进程会比优先级较低的进程先运行。友善值还可以控制一个进程的时间片大小。

友善值的有效范围从 - 20 到 19（包括 - 20 和 19），而且默认值为 0。你可能会有点困惑，一个进程的友善值越低，则它的优先级就越高，而且它的时间片也越大；反过来说，友善值越高，则进程的优先级越低，而且它的时间片也越小。因此调高一个进程的友善值就等于是善待系统上的其他进程。但是以数字来看的确令人混淆。当我们说一个进程具有“高优先级”时，我们的意思是说，相较于优先级较低的进程，调度程序会更快安排它运行，而且会让它运行更长时间。.

nice()

Linux 提供了若干可用于取得以及设定进程友善值的系统调用。`nice()` 是其中最简单的一个：

```
#include <unistd.h>

int nice (int inc);
```

执行成功，nice() 会使用 inc 来调高一个进程的友善值，而且会返回刚才所更新的值。只有具备 CAP_SYS_NICE 能力的进程（实际上就是 root 所拥有的进程）可以对 inc 提供一个负值来调低它的友善值，因而会调高它的优先级。因此，拥有者非 root 的进程只能调低它们的优先级（通过调高它们的友善值）。

发生错误时，nice() 会返回 -1。然而，因为 nice() 会返回新的友善值，所以 -1 也是执行成功时的返回值。为了有所区别，调用 nice() 之前，你可以先将 errno 清为零并在事后检查它的值。例如：

```
int ret;

errno = 0;
ret = nice (10);     /* 将我们的友善值调高 10 */
if (ret == -1 && errno != 0)
        perror ("nice");
else
        printf ("nice value is now %d\n", ret);
```

发生错误时，Linux 只会返回单一错误代码 EPERM，这表示进行调用的进程调高了它的优先级（通过使用一个负的 inc 值），但是它并不具备 CAP_SYS_NICE 能力。当放入 inc 的友善值超越有效值的范围时，有的系统还会返回 EINVAL，但是 Linux 不会。事实上，有需要时，Linux 会将无效的 inc 值自动调整至此限度所允许的范围之内。

只需 186 将 inc 设为 0，就可以轻易取得当前的友善值：

```
printf ("nice value is currently %d\n", nice (0));
```

往往一个进程会想要设定一个绝对的友善值，而不是设定一个相对增加的友善值。如下的程序代码可以完成此事：

```
int ret, val;

/* 取得当前的友善值 */
val = nice (0);

/* 我们想要设定 10 的友善值 */
val = 10 - val;
errno = 0;
ret = nice (val);
if (ret == -1 && errno != 0)
        perror ("nice");
else
        printf ("nice value is now %d\n", ret);
```

getpriority() 与 setpriority()

一个更好的解决方案就是使用 getpriority() 与 setpriority() 系统调用，它们提供了较多的控制能力，但是它们的操作也更复杂：

```
#include <sys/time.h>
#include <sys/resource.h>

int getpriority (int which, int who);
int setpriority (int which, int who, int prio);
```

这两个调用可用于操作进程、进程组或用户，可以通过 which 和 who 来指定。which 的值必须是 PRIO_PROCESS、PRIO_PGRP 或 PRIO_USER 其中之一，在此情况下，who 可分别用于指定进程 ID、进程组 ID 或用户 ID。如果 who 的值为 0，则此类调用可分别用于操作当前的进程 ID、进程组 ID 或用户 ID。

getpriority() 会返回指定的任何进程中优先级最高者（友善值最低者）。而 setpriority() 会将所指定的任何进程的优先级设定为 prio。如同 nice()，只有具备 CAP_SYS_NICE 能力的进程可以调高进程的优先级（调低友善值）。此外，只有具备此能力的进程可以调高或调低非进行调用的用户所拥有的进程的优先级。

如同 nice()，发生错误时，getpriority() 会返回 -1。因为执行成功时也会返回此值，如果程序设计者想要处理错误情况，则进行调用之前，他们应该先清除 errno 的值。setpriority() 并没有这个问题，执行成功时，setpriority() 总是会返回 0；发生错误时，则会返回 -1。

下面的程序代码会返回当前进程的优先级：

```
int ret;

ret = getpriority (PRIO_PROCESS, 0);
printf ("nice value is %d\n", ret);
```

下面的程序代码会将当前进程组中所有进程的优先级设定成 10：

```
int ret;

ret = setpriority (PRIO_PGRP, 0, 10);
if (ret == -1)
        perror ("setpriority");
```

发生错误时，这两个函数会将 errno 设定成下面的其中一个值：

EACCESS

进程试图调高指定进程的优先级，但是并不具备 CAP_SYS_NICE 能力（仅适用

于 setpriority())。

EINVAL

which 所指定的值并非 PRIO_PROCESS、PRIO_PGRP 或 PRIO_USER 其中之一。

EPERM

所指定进程的有效用户ID与正在运行的进程（running process）的有效用户ID并不相符，而且正在运行的进程并不具备 CAP_SYS_NICE 能力（仅适用于 setpriority())。

ESRCH

找不到与 which 和 who 所提供规则相符的进程。

I/O 优先级

除了调度优先级（scheduling priority），Linux 还允许进程指定 I/O 优先级。此值会影响进程 I/O 请求的相对优先级。内核的 I/O 调度程序（参考第四章）在服务来自 I/O 优先级较低进程的请求之前，会先服务来自 I/O 优先级较高进程的请求。

默认情况下，I/O 调度程序会使用进程的友善值来确定 I/O 优先级。因此，设定友善值会自动变更 I/O 优先级。然而，Linux 内核额外提供了两个系统调用，可用于显式设定和取得与友善值无关的 I/O 优先级：

```
int ioprio_get (int which, int who)
int ioprio_set (int which, int who, int ioprio)
```

不幸的是，内核尚未导出这两个系统调用，而且 *glibc* 并未提供任何用户空间访问接口。没有 *glibc* 的支持，使用起来会相当麻烦。此外，等到 *glibc* 开始支持时，其所提供的接口可能会跟这两个系统调用有所不同。开始支持这两个系统调用之前，你有两种具有可移植性的方法可用于操作进程的 I/O 优先级：经友善值或者经实用程序，例如 *ionice*，它是 *util-linux* 包的一部分（注 2）。

并非所有的 I/O 调度程序都支持 I/O 优先级。特别的是，完全公平队列（Complete Fair Queuing，常简写为 CFQ）I/O 调度程序就支持 I/O 优先级；就目前而言，其他的标准调度程序并不支持 I/O 优先级。如果当前I/O 调度程序不支持 I/O 优先级，它们会默默予以忽略。

注 2： *util-linux* 包可以在 *http://www.kernel.org/pub/linux/utils/util-linux* 找到。它被授权在 GNU General Public License v2 之下。

处理器亲和性

Linux 支持具有多个处理器的单一系统。引导过程除外，支持多个处理器的大量工作由进程调度程序负责。在对称式多重处理（symmetric multiprocessing，常简写成 SMP）机器上，进程调度程序必须决定每个 CPU 上要运行哪些进程。有两项挑战源自此责任：调度程序必须想办法充分利用系统上的所有处理器，因为当有一个进程已就绪等待运行，却有一个 CPU 闲置在一旁，这显然没有效率。

然而，一个进程一旦被安排在某个 CPU 上运行，往后进程调度程序也会将它安排在相同的 CPU 上运行。这是有益的，因为将一个进程从一个处理器迁移（*migrating*）到另一个处理器是要付出代价的。

这些代价中的最大者与“迁移的缓存区效应”（*cache effect*）有关。由于现代 SMP 系统的设计，与每个处理器相关的缓存区是不同且独立的。也就是说，如果数据位于一个处理器的缓存区中，就不会位于另一个处理器的缓存区中。因此，如果一个进程被移往一个新的 CPU，而且将新的数据写入内存，那么位于旧 CPU 缓存区中的数据就过时了。现在若使用该缓存区，则会导致数据被破坏。为了避免此现象，当一个缓存区中缓存了一个新的内存块时，这会让其他每个缓存区中的数据失效。因此，任何时刻特定的一段数据只能出现在一个处理器的缓存区中（假设数据全都被缓存起来）。当有一个进程从一个处理器移往另一个处理器时会因而付出两种代价：被移动的进程无法访问被缓存起来的数据，而且位于原处理器缓存区中的数据必须作废。因为必须付出这些代价，所以进程调度程序会尽可能为一个进程安排特定的 CPU 来运行它。

当然，进程调度程序的两项目标可能会无法实现。如果一个处理器的进程负荷比另一个处理器大很多（或者，更糟糕的是，如果一个处理器处于忙碌状态，而另一个处理器却闲置在一旁），明智的做法是对一些进程重新调度，让它们在比较不忙的 CPU 上运行。决定何时移动进程以响应此类不平衡的状态称为负载均衡（*load balancing*），这对 SMP 机器的性能而言非常重要。

处理器亲和性（*processor affinity*）是指一个进程被一致地安排在同一个处理器上运行的可能性。软亲和性（*soft affinity*）是指调度程序继续安排一个进程在同一个处理器上运行的自然倾向。正如我们所作的讨论，这是一个值得利用的特性。Linux 调度程序会尽可能安排相同的进程在相同的处理器上运行，而且只会在负载极不平衡的情况下将一个进程从一个 CPU 移往另一个 CPU。这让处理器可以最小化迁移的缓存区效应，但是仍能确保一个系统中所有处理器的负载是平衡的。

然而，有时用户或应用程序会想把进程与处理器结合在一起。这是因为进程具有强烈的缓存区敏感性，而且想停留在相同的处理器上。结合一个进程与一个特定的处理器，让内核按照此关系行事称为设定一个硬亲和性（*hard affinity*）。

sched_getaffinity() 与 sched_setaffinity()

进程会继承其父进程的 CPU 亲和性，默认情况下，进程可以运行在任何 CPU 之上。Linux 提供了两个系统调用可用于取得和设定一个进程的“硬亲和性”：

```
#define _GNU_SOURCE

#include <sched.h>

typedef struct cpu_set_t;

size_t CPU_SETSIZE;

void CPU_SET (unsigned long cpu, cpu_set_t *set);
void CPU_CLR (unsigned long cpu, cpu_set_t *set);
int CPU_ISSET (unsigned long cpu, cpu_set_t *set);
void CPU_ZERO (cpu_set_t *set);

int sched_setaffinity (pid_t pid, size_t setsize,
                       const cpu_set_t *set);

int sched_getaffinity (pid_t pid, size_t setsize,
                       cpu_set_t *set);
```

sched_getaffinity() 可用于取得进程 pid 的 CPU 亲和性，而且会以特殊的 cpu_set_t 类型来存放它，这可通过特殊的宏来访问。如果 pid 的值为 0，则此调用会取得当前进程的亲和性。setsize 参数是 cpu_set_t 类型的大小，为了配合此类型大小未来的变化，*glibc* 可能会用到此参数。执行成功时，sched_getaffinity() 会返回 0；执行失败时，它会返回 -1 并且设定 errno。请看下面的例子：

```
cpu_set_t set;
int ret, i;

CPU_ZERO (&set);
ret = sched_getaffinity (0, sizeof (cpu_set_t), &set);
if (ret == -1)
        perror ("sched_getaffinity");

for (i = 0; i < CPU_SETSIZE; i++) {
        int cpu;

        cpu = CPU_ISSET (i, &set);
        printf ("cpu=%i is %s\n", i,
                cpu ? "set" : "unset");
}
```

进行调用之前，我们会使用 CPU_ZERO 将 set 中所有位都清零。然后我们会从 0 到 CPU_SETSIZE 迭代处理 set。请注意，CPU_SETSIZE 并不是 set 的大小 —— 你绝对不应该把它传递给 setsize，而是 set 所可能表示的处理器数目。因为当前的实

现是以单一位来表示每个处理器，所以 CPU_SETSIZE 会比 sizeof(cpu_set_t) 大很多。我们还会使用 CPU_ISSET 来检查系统中特定的处理器 i，是否绑定（bound）或未绑定（unbound）此进程。如果返回 0，表示未绑定；如果返回非零值，表示绑定了。

系统中只有实体的处理器会被设定。因此，若在一个具有两个处理器的系统上运行此程序代码，将产生如下的结果：

```
cpu=0 is set
cpu=1 is set
cpu=2 is unset
cpu=3 is unset
...
cpu=1023 is unset
```

如同此输出所显示的，CPU_SETSIZE（从零起算）的值当前是 1,024。

我们所关心的只是 CPU #0 和 CPU #1，因为它们是此系统上仅有的实体处理器。也许我们想确保我们的进程只会运作在 CPU #0 之上，而不会运作在 CPU #1 之上。下面的程序代码可以完成此事：

```
cpu_set_t set;
int ret, i;

CPU_ZERO (&set);          /* 清除所有 CPU */
CPU_SET (0, &set);        /* 允许 CPU #0 */
CPU_CLR (1, &set);        /* 禁止 CPU #1 */
ret = sched_setaffinity (0, sizeof (cpu_set_t), &set);
if (ret == -1)
        perror ("sched_setaffinity");

for (i = 0; i < CPU_SETSIZE; i++) {
        int cpu;

        cpu = CPU_ISSET (i, &set);
        printf ("cpu=%i is %s\n", i,
                cpu ? "set" : "unset");
}
```

一如往常，首先我们会使用 CPU_ZERO 将 set 清零。然后我们会使用 CPU_SET 设定 CPU #0 以及使用 CPU_CLR 清除 CPU #1。CPU_CLR 操作是多余的，因为我们刚刚已经将整个 set 清零，这么做只是为了完整性。

同样在这个具有两个处理器的系统上运行此程序会产生与之前稍微不同的输出：

```
cpu=0 is set
cpu=1 is unset
cpu=2 is unset
...
cpu=1023 is unset
```

现在，CPU #1 会被清除。此进程只会运行在 CPU #0 之上。

errno 有四个可能值：

EFAULT
: 所提供的指针超出了进程地址空间的范围或者无效。

EINVAL
: 在此情况下，set 中并未启用系统上的实体处理器(仅适用于 sched_setaffinity())，或是 setsize 小于内核的内部数据结构（用于表示处理器的设定）的尺寸。

EPERM
: 进行调用的进程当前的有效用户 ID 并未拥有 pid 所关联的进程，而且该进程并不具备 CAP_SYS_NICE 能力。

ESRCH
: 找不到 pid 所关联的进程。

实时系统

在计算机领域，实时（*real-time*）这个术语往往是一些混乱和误解的源头。一个实时系统必须服从操作期限（*operational deadline*）：激励发生之后，必须在预定时间之内作出响应。一个常见的实时系统就是几乎所有现代汽车之上都可以找到的防死锁煞车系统（*anti-lock braking system*，常简写成 ABS）。在此系统中，当煞车被踩下时，计算机会调控煞车压力，通常是每秒施加和释放最大煞车压力许多次。这可以防止因为车轮“被死锁”而降低停车的性能甚至让汽车进入无法控制的滑行状态。在这样的系统中，操作期限就是指系统响应车轮被死锁情况的速度有多快，以及系统施加煞车压力的速度有多快。

多数现代的操作系统，包括 Linux，都会对实时功能提供某种程度的支持。

硬性与软性实时系统

实时系统分成两类：硬性和软性。一个硬性实时系统需要严格遵守操作期限。超过期限便失败，而且是一个重大的错误。另一方面，一个软性实时系统并不认为逾越期限是一个严重的失败。

硬性实时应用程序很容易识别，例如：防死锁煞车系统、医疗装置以及信号处理。软性实时应用程序往往不容易识别，视频处理应用程序是其中一个比较明显的例子：如果错过了期限，用户会意识到质量变差，然而只是漏掉几帧画面是可以忍受的。

许多其他的应用程序具有时序约束（timing constraint），如果没有得到满足，将不利于用户的使用经验，例如：多媒体应用程序、游戏以及网络程序。然而，文本编辑器呢？如果程序无法快速响应按键，就是一个糟糕的使用经验，这会让用户感到不悦或挫败。这是软性实时应用程序吗？没错，开发者在编写应用程序时，已经意识到他们需要及时地响应按键。但是这可以视为操作期限吗？尽管软性实时应用程序的定义不一而足，但是相当明确。

与普遍的概念相反的是，一个实时系统的速度未必快。的确，针对可比较的硬件，一个实时系统的速度或许比一个非实时系统还慢 —— 因为如果什么都没有，支持实时进程需要增加系统的开销。同样地，硬性与软性实时系统的划分与操作期限的长短无关。检测到中子通量（neutron flux）过多的几秒内，若 SCRAM（紧急停炉）系统无法降低控制棒，则核子反应炉（nuclear reactor）将出现过热的现象。这是一个具有“漫长”（就计算机而言）操作期限的硬性实时系统。反过来说，一个视频播放器可能会跳过一帧画面或是让声音结结巴巴，如果它不能在 100 毫秒之内为播放缓冲区重新填满数据的话。这是一个对操作期限有严格要求的软性实时系统。

延迟、抖动以及期限

延迟（*latency*）是指从激励（stimulus）发生到进行响应所经过的时间。如果延迟小于或等于操作期限，则系统可以正确运作。在许多硬性实时系统中，延迟等于操作期限 —— 系统会定时处理激励。在软性实时系统中，所需要的响应时间较不确切，而且延迟呈现出很大程度的变化 —— 目标很简单，就是在期限之内发生响应。

延迟往往很难测量，因为它的估算需要知道激励发生的时间。然而，为激励记录时间的能力往往取决于响应激励的能力。因此，对延迟所进行的许多测量所测量到的往往不是延迟，事实上，所测量到的是响应之间在时序上的差异。连续激励之间在时序上的差异称为抖动（*jitter*）而不是延迟。

举例来说，假设一个激励每 10 毫秒发生一次。为了测量我们系统的性能，我们会记录响应的时间以确定它们每 10 毫秒发生一次。然而，与此目标值的差异并非延迟 —— 是抖动。我们所测量到的是连续响应中的差异。如果不知道激励是何时发生的，我们也无法知道激励与响应在时间上的实际差异。就算知道激励每 10 毫秒发生一次，我们仍旧不知道第一次是何时发生的。或许令人惊讶的是，对延迟所做的许多测量产生了这个错误，并且所汇报的是抖动而不是延迟。可以肯定的是，抖动是一个有用的指标（metric），而且这样的测量工具可能会相当有用。然而，我们必须称鸭子为鸭子！

硬性实时系统往往呈现非常低的抖动，因为它们会在一个确切的时间之后 —— 而不是之内 —— 响应激励。此类系统的目标是零抖动以及延迟等于操作期限。如果延迟逾越期限，则表示系统失败。

软性实时系统容易受抖动影响。在这些系统中，响应时间理想情况下是在操作期限之内 —— 常常会提前很多，但有时不会。因此，抖动往往是延迟的最好替代品，可作为性能指标（performance metric）。

Linux 的实时支持

Linux 经 IEEE Std 1003.1b-1993（常简写成 POSIX 1993 或 POSIX.1b）所定义的一系列系统调用为应用程序提供了软性实时支持。

从技术上来说，POSIX 标准并未规定所提供的实时支持是否为软性或硬性的。事实上，所有 POSIX 标准实际只描述了几个与优先级有关的调度策略。操作系统会对这些方针实施什么样的时间限制则是 OS 设计者的责任。

这些年来，Linux 内核已获得越来越好的实时支持，并且提供越来越低的延迟以及更一致的抖动，而且不会影响系统的性能。这多半是因为改善延迟能够帮助许多类型的应用程序，例如桌面和 I/O 密集型应用程序，而不只是实时应用程序。会有这样的改善也可以归因于 Linux 在嵌入式和实时系统上的成功。

不幸的是，许多对 Linux 内核所进行的嵌入式和实时修改，仅以定制的 Linux 解决方案存在着，并未纳入主流的正式内核。这些修改中有些可以进一步减少延迟，甚至是提供硬性实时的行为。接下来的段落只会探讨正式的内核接口以及主流内核的行为。幸运的是，大部分的实时修改仍继续利用 POSIX 接口。因此，随后的讨论还是跟修改过的系统有关。

Linux 的调度策略与优先级

Linux 调度程序对一个进程所采取的行为取决于该进程的调度策略（*scheduling policy*），也称为调度类（*scheduling class*）。除了一般的默认策略，Linux 还提供了两个实时调度策略。我们可以使用来自头文件 `<sched.h>` 的预处理器宏（preprocessor macro）来表示每个策略，这些宏分别是 `SCHED_FIFO`、`SCHED_RR` 以及 `SCHED_OTHER`。

每个进程都会有一个与友善值无关的静态优先级值（*static priority*）。对一般应用程序而言，此优先级值往往是 0。对实时进程而言，此优先级值的范围可以从 1 到 99（包含 1 和 99）。Linux 调度程序往往会选择让优先级最高的进程（也就是具有最大静态优

先级值的进程）先运行。如果一个正在运行的进程的静态优先级值为 50，而且有一个优先级值为 51 的进程变为可运行的（runnable），则正在运行的进程会立即遭到调度程序的抢占（插队），而且切换到新的可运行进程。反过来说，如果一个正在运行的进程的优先级值为 50，而且有一个优先级值为 49 的进程变为可运行的，则调度程序不会运行该进程，除非优先级值为 50 的进程遭到阻挡而变为不可运行的。因为一般进程的优先级值为 `0`，所以任何可运行的实时进程往往会抢占一般的进程而进入运行状态。

先入先出策略

先入先出（*first in, first out*，常简写为 *FIFO*）类是一个非常简单的实时策略，不需要考虑时间片的问题。只要没有优先级较高的进程变为可运行的，一个 FIFO 类的进程将可继续运行下去。FIFO 类可以使用宏 `SCHED_FIFO` 来表示。

因为此策略不需要考虑时间片的问题，所以它的操作规则相当简单：

- 一个可运行的 FIFO 类的进程总是可以先运行，只要它是系统上优先级最高者。特别是，一旦 FIFO 类的进程变为可运行的，它会立即抢占一般的进程。
- 一个 FIFO 类的进程会继续运行下去，除非它遭到阻挡或是调用了 `sched_yield()`，或者有一个优先级较高的进程变为可运行。
- 当一个 FIFO 类的进程遭到阻挡时，调度程序会把它从可运行进程列表中移除。当该进程再次变为可运行的时，该进程会被插入（与该进程的优先级值相应的）进程列表的尾端。因此，除非有任何其他优先级较高或相等的进程停止运行，否则它将不会运行。
- 当一个 FIFO 类的进程调用 `sched_yield()` 时，调度程序会把该进程移往（与该进程的优先级值相应的）进程列表的尾端。因此，除非有任何其他优先级相等的进程停止运行，否则它将不会运行。如果进行调用的进程是（与它的优先级值相应的）进程列表中唯一的进程，则 `sched_yield()` 将毫无作用。
- 当一个优先级较高的进程抢占一个 FIFO 类的进程时，这个 FIFO 类的进程会继续留在（与它的优先级值相应的）进程列表中的相同位置上。因此，一旦优先级较高的进程停止运行，则被抢占（插队）的 FIFO 类的进程会继续运行下去。
- 当一个进程成为 FIFO 类的进程时，或者当一个进程的静态优先级值有所变动时，它会被摆在（与它的优先级值相应的）进程列表中的首端。因此，一个刚被优先考虑的FIFO 类的进程可以抢占一个具有相同优先级的正在运行的进程。

基本上，我们可以说 FIFO 类的进程可以运行到它们自己满意为止，只要系统上没有优先级更高的进程。以上的规则说明了优先级相同的 FIFO 类的各个进程之间会发生什么事。

轮转策略

轮转策略（*round-robin*，常简写为 *RR*）类如同 FIFO 类，不过 RR 类会在进程具有相同的优先级时采用额外的规则。宏 SCHED_RR 可用于表示此类。

调度程序会替每个 RR 类的进程分配一个时间片（timeslice）。当一个 RR 类的进程用完它的时间片时，调度程序会把该进程移往（与该进程的优先级值相应的）进程列表的尾端。这样，具有相同优先级的 RR 类的各个进程将按照轮转的方式被调度（这样可确保它们都能公平分配到 CPU 时间）。如果特定优先级上只有一个进程，则 RR 类如同 FIFO 类。在此情况下，当该进程用完它的时间片后，该进程仍会继续执行。

我们可以把一个 RR 类的进程视为一个 FIFO 类的进程，除了当 RR 类的进程用完它的时间片后便会停止执行，并被移往（与该进程的优先级值相应的）可运行进程列表的尾端。

至于要使用 SCHED_FIFO 还是 SCHED_RR，则完全是内部优先级行为（intra-priority behavior）的问题。RR 类的时间片只与优先级相同的进程有关。FIFO 类的进程会继续执行到它满意为止；RR 类的进程将与其他优先级相同的 RR 类的进程一起被调度。无论是哪一种情况，只要存在优先级更高的进程，优先级较低的进程就不会运行。

普通策略

宏 SCHED_OTHER 用于表示标准的调度策略，默认的非实时类。每个普通类的进程（normal-classed process）的静态优先级值皆为 0。因此，任何可运行的 FIFO 或 RR 类将抢占一个正在运行的普通类的进程。

调度程序会使用稍早所讨论的友善值，按优先次序排列普通类型的进程。友善值不会受到静态优先级值（值为 0）的影响。

批处理调度策略

宏 SCHED_BATCH 用于表示批处理（*batch*）或空闲（*idle*）调度策略。它的行为有点跟实时策略相反：当系统上没有其他可运行进程时，此类的进程才会运行，即使其他进程已经用完它们的时间片。这不同于具有最大友善值的进程（也就是优先级最低的进程）的行为，因为这样的进程最后还是会由于优先级较高的进程用完它们的时间片而能够运行。

设定 Linux 的调度策略

进程可经 sched_getscheduler() 以及 sched_setscheduler() 来操作 Linux 调度策略：

```
#include <sched.h>

struct sched_param {
        /* ... */
        int sched_priority;
        /* ... */
};

int sched_getscheduler (pid_t pid);

int sched_setscheduler (pid_t pid,
                        int policy,
                        const struct sched_param *sp);
```

一次成功的 sched_getscheduler() 调用会返回 pid 所指定的进程调度策略。如果 pid 的值为 0，则此调用会返回进行调用的进程的调度策略。<sched.h> 中所定义的一个整数可用于表示调度策略：先入先出方针是 SCHED_FIFO；循环方针是 SCHED_RR；一般方针是 SCHED_OTHER。发生错误时，此调用会返回 -1（这绝对不代表一个有效的调度策略）并把 errno 设定为适当的值。

它的用法很简单：

```
int policy;

/* 取得我们的调度策略 */
policy = sched_getscheduler (0);

switch (policy) {
case SCHED_OTHER:
        printf ("Policy is normal\n");
        break;
case SCHED_RR:
        printf ("Policy is round-robin\n");
        break;
case SCHED_FIFO:
        printf ("Policy is first-in, first-out\n");
        break;
case -1:
        perror ("sched_getscheduler");
        break;
default:
        fprintf (stderr, "Unknown policy!\n");
}
```

sched_setscheduler() 可用于将 pid 所指定的进程调度策略设定为 policy。与策略相关的任何参数可经 sp 来设定。如果 pid 的值为 0，则所设定的是进行调用的进程的策略与参数。执行成功时，此调用会返回 0；执行失败时，此调用会返回 -1 并将 errno 设定为适当的值。

sched_param 结构中的有效字段取决于操作系统所支持的调度策略。SCHED_RR 与 SCHED_FIFO 策略需要用到一个字段 sched_priority（用于表示静态优先级值）。SCHED_OTHER 不会用到任何字段，然而未来所支持的调度策略可能会用到新的字段。因此，具有可移植性且合法的程序不应该假设此结构的布局。

一个进程的调度策略和参数的设定很容易：

```
struct sched_param sp = { .sched_priority = 1 };
int ret;

ret = sched_setscheduler (0, SCHED_RR, &sp);
if (ret == -1) {
        perror ("sched_setscheduler");
        return 1;
}
```

上面这段程序代码会将进行调用的进程的调度策略及静态优先级值分别设定为轮转类型及 1。此处，我们假设 1 是一个有效的优先级值 —— 从技术上来说，没有这个必要。稍后我们将讨论如何替特定的策略找到有效的优先级值范围。

设定 SCHED_OTHER 以外的策略时需要具备 CAP_SYS_NICE 能力。因此，一般只有 root 用户能够运行实时进程。从 2.6.12 版内核开始，RLIMIT_RTPRIO 资源限制可让非 root 用户对实时策略的设定达到特定的优先级上限。

发生错误时，errno 会被设定为下面的其中一个值：

EFAULT

指针 sp 指向了一个无效的或无法访问的内存范围。

EINVAL

policy 指定了无效的调度策略，或者 sp 中所指定的值对 policy 所指定的调度策略而言毫无意义（仅适用于 sched_setscheduler()）。

EPERM

进行调用的进程并不具备所需要的能力。

ESRCH

pid 的值并不代表一个正在运行的进程。

设定调度参数

POSIX 所定义的 sched_getparam() 与 sched_setparam() 接口可用于取得并设定一个已经设定好的调度策略：

```
#include <sched.h>

struct sched_param {
        /* …… */
        int sched_priority;
        /* …… */
};

int sched_getparam (pid_t pid, struct sched_param *sp);

int sched_setparam (pid_t pid, const struct sched_param *sp);
```

sched_getscheduler() 接口只会返回调度策略，不会返回任何相关的参数。而 sched_getparam() 则会经 sp 返回与 pid 相关的调度参数：

```
struct sched_param sp;
int ret;

ret = sched_getparam (0, &sp);
if (ret == -1) {
        perror ("sched_getparam");
        return 1;
}

printf ("Our priority is %d\n", sp.sched_priority);
```

如果 pid 的值为 0，则此调用会返回进行调用的进程的参数。执行成功时，此调用会返回 0；执行失败时，它会返回 -1 并将 errno 设定为适当的值。

因为 sched_setscheduler() 也可以设定任何相关的调度参数，所以 sched_setparam() 只在事后用于修改参数值：

```
struct sched_param sp;
int ret;

sp.sched_priority = 1;
ret = sched_setparam (0, &sp);
if (ret == -1) {
        perror ("sched_setparam");
        return 1;
}
```

执行成功时，pid 的调度参数会根据 sp 来设定，而且此调用会返回 0；执行失败时，此调用会返回 -1 并将 errno 设定为适当的值。

如果我们依次执行前面两段程序代码，将看到如下的输出：

```
Our priority is 1
```

此范例中，我们又假设 1 是一个有效的优先级值。但是具有可移植性的应用程序应该

要确定实际的有效优先级值。稍后我们将探讨如何决定有效的优先级值范围。

错误代码

发生错误时，errno 会被设定为下面的其中一个值：

EFAULT

指针 sp 指向了一个无效的或无法访问的内存范围。

EINVAL

sp 中所指定的值对 policy 所指定的调度策略而言毫无意义（仅适用于 ssched_getparam()）。

EPERM

进行调用的进程并不具备所需要的能力。

ESRCH

pid 的值并不代表一个正在运行的进程。

决定有效的优先级值范围

在前面的例子中，我们将固定的优先级值传递给与调度相关的系统调用。POSIX 并未规定一个系统上应该存在哪些调度优先级值，除了最小值与最大值之间必须至少存在 32 个优先级值。正如稍早在“Linux 的调度策略与优先级值”一节所述，Linux 为两种实时调度策略实现了范围从 1 到 99（包含 1 和 99）的优先级值。具可移植性的程序一般会实现它自己的优先级值范围，并且把它们对应至操作系统的范围。举例来说，如果你想以四种实时优先级级别来运行进程，你可以动态决定优先级值的范围并选出四个值。

Linux 提供了两个可用于取得有效优先级值的范围的系统调用。其中一个会返回最小值，另一个会返回最大值：

```
#include <sched.h>

int sched_get_priority_min (int policy);

int sched_get_priority_max (int policy);
```

执行成功时，sched_get_priority_min() 以及 sched_get_priority_max() 会分别返回与policy 所指定的调度算法相应的有效优先级值的最小值以及最大值，而且都会返回 0。执行失败时，这两个调用都会返回 -1。唯一可能的错误是，policy 被指定了无效值，在此情况下 errno 会被设定为 EINVAL。

它们的用法很简单：

```
int min, max;

min = sched_get_priority_min (SCHED_RR);
if (min == -1) {
        perror ("sched_get_priority_min");
        return 1;
}

max = sched_get_priority_max (SCHED_RR);
if (max == -1) {
        perror ("sched_get_priority_max");
        return 1;
}

printf ("SCHED_RR priority range is %d - %d\n", min, max);
```

在一个标准的 Linux 系统上，这段程序代码会产生如下的结果：

```
SCHED_RR priority range is 1 - 99
```

如稍早所述，优先级值越大代表优先级越高。要替一个进程的调度策略设定最高的优先级，你可以这么做：

```
/*
 * set_highest_priority —— 将 pid 的调度策略的优先级值设定为它的当前调度策略
 * 所允许的最高值。如果 pid 的值为零，则设定当前进程的优先级。
 *
 * 执行成功时返回零。
 */
int set_highest_priority (pid_t pid)
{
        struct sched_param sp;
        int policy, max, ret;

        policy = sched_getscheduler (pid);
        if (policy == -1)
                return -1;

        max = sched_get_priority_max (policy);
        if (max == -1)
                return -1;

        memset (&sp, 0, sizeof (struct sched_param));
        sp.sched_priority = max;
        ret = sched_setparam (pid, &sp);

        return ret;
}
```

程序一般会取得系统的最大值或最小值，然后使用 1 的增量（例如 max-1、max-2 等）来指派所需要的优先级值。

sched_rr_get_interval()

如稍早所讨论的，SCHED_RR 进程的行为如同 SCHED_FIFO 进程，除了调度程序会替 SCHED_RR 进程指派时间片。当一个 SCHED_RR 进程用完它的时间片，调度程序会把它移往与它的当前优先级值相应的运行列表的尾端。这样，具有相同优先级的所有 SCHED_RR 进程会以轮转的方式轮流执行。优先级较高的进程（以及优先级相同或较高的 SCHED_FIFO 进程）总是会抢占正在运行的 SCHED_RR 进程，无论它是否还有时间片尚未用完。

POSIX 定义了一个接口可用于取得特定进程的时间片长度：

```
#include <sched.h>

struct timespec {
        time_t  tv_sec;      /* seconds */
        long    tv_nsec;     /* nanoseconds */
};

int sched_rr_get_interval (pid_t pid, struct timespec *tp);
```

执行成功时，sched_rr_get_interval() 会把分配给 pid 的时间片长度存放到 tp 所指向的 timespec 结构，而且会返回 0；执行失败时，此调用会返回 -1 并将 errno 设定为适当的值。

依据 POSIX，此函数只能用于 SCHED_RR 进程。然而，在 Linux 上，它可以取得任何进程的时间片长度。具可移植性的应用程序应该假设此函数只能用于 SCHED_RR 进程；Linux 特有的程序可依需要利用此调用。下面是一个例子：

```
struct timespec tp;
int ret;

/* 取得当前进程的时间片长度 */
ret = sched_rr_get_interval (0, &tp);
if (ret == -1) {
        perror ("sched_rr_get_interval");
        return 1;
}

/* 将秒以及纳秒转换成毫秒 */
printf ("Our time quantum is %.2lf milliseconds\n",
        (tp.tv_sec * 1000.0f) + (tp.tv_nsec / 1000000.0f));
```

如果正在运行的是 FIFO 类型的进程，则 `tv_sec` 与 `tv_nsec` 的值皆为 0，表示无限大。

错误代码

发生错误时，`errno` 会被设定为下面的其中一个值：

`EFAULT`
: 指针 `tp` 指向了无效或无法访问的内存。

`EINVAL`
: `pid` 指定了无效的值（例如负值）。

`ESRCH`
: 尽管 `pid` 指定了有效值，但是却引用了不存在的进程。

实时进程的预防措施

由于实时进程的本质，开发者在开发和调试此类程序时应该谨慎行事。如果一个实时程序突然发脾气（go off the deep end），系统的反应会变得迟钝。任何的 CPU 密集型循环在一个实时程序 —— 也就是任何不会被阻挡的程序代码块 —— 中会继续无止境地运行下去，只要没有优先级更高的实时进程变为可运行的。

因此，设计实时程序的时候需要谨慎为之。这类程序至高无上，可以轻易拖垮整个系统。下面是一些要诀与注意事项：

- 记住，如果系统上没有优先级更高的实时进程，则任何 CPU 密集型循环会运行到完成为止，不会被中断。如果这是个无穷循环，则系统的反应会变得迟钝。
- 因为实时进程运行时会耗用系统上一切资源，所以必须特别注意它们的设计。小心不要让系统其他进程等不到处理器时间。
- 必须非常小心忙碌等待（busy waiting）的问题。如果一个实时进程忙碌等待一个优先级较低的进程所把持的一个资源，则实时进程将永远忙碌等待下去。
- 开发一个实时进程的时候，让一个终端保持开启的状态，以更高的优先级（相较于开发中的进程）来运行该实时进程。发生紧急状况时，终端机依然会有反应，允许你终止（kill）失控的实时进程（因为终端机依然是闲置着，等待键盘的输入，所以不会干扰到其他的实时进程）。
- 使用 *chrt* 实用程序（*util-linux* 工具包的一部分）可轻易取得和设定其他进程的实时属性。使用此工具可轻易执行一个实时调度类中的任何程序（例如前面所提到的终端机）或是变更现有应用程序的实时优先级。

决定论

实时进程的重要性在于决定论（determinism）。在实时运算中，行动已成定论：如果输入一样，则所花的时间一样，所产生的结果也一样。在现代的计算机中则有所不同，因为有些东西的定义已经完全不一样了：缓存区有多个层次（导致命中或失误的不可预测性）、多个处理器、页面调度（paging）、交换（swapping）以及多任务化（multitasking），严重破坏了特定行动需要花多少时间的可估计性。的确，我们已经达到了每个行动（硬盘的访问除外）都"奇快无比"的地步，但是与此同时，现代的系统也让我们难以准确估计特定操作实际要花多少时间。

一般来说，实时应用程序往往会尝试限制不可预测性，特别是情况最糟的延迟。接下来我们会探讨解决此类问题的两种方法。

预防数据出错以及锁进内存

想象以下情况：定制的 incoming ICBM（来袭洲际弹道导弹）监控器送出硬件中断，而且该设备的驱动程序会迅速将数据从硬件复制到内核。驱动程序注意到有个进程处在休眠状态，因为它为了等待数据而被硬件的设备节点阻挡。驱动程序会通知内核去唤醒该进程。内核 —— 注意到此进程的运行会采用一个实时的调度策略以及一个高的优先级值 —— 会立即抢占当前正在运行的进程并且进入高速运转，决定立即调度实时进程。调度程序会切换至该实时进程，并且将运行环境切换至它的地址空间。现在这个进程开始运行了。整个过程需要 0.3 ms，符合最差的情况下 1 ms 的延迟上限。

现在，用户空间中，实时进程注意到来袭的洲际弹道导弹并着手处理它的弹道。根据所计算的弹道，实时进程开始部署一个反弹道导弹系统。只要 0.1 ms 的时间就足以部署 ABM（反弹道导弹系统）以及拯救生命。但是 —— 哦，不！ —— ABM 的程序代码已经被交换到磁盘。发生了页面失误，处理器会切换回内核模式，而且内核会启动硬盘的 I/O 以便取回被交换出的数据。调度程序会让该进程进入休眠状态，直到页面失误获得服务。就这样过去了几秒钟。一切都太迟了。

显然，页面调度以及交换引入了相当不可测定的行为，这严重地破坏了实时进程的可预测性。为了避免此浩劫，实时应用程序往往会把自己的地址空间中所有的页面锁进物理内存，以预防它们进入内存时出错并避免它们被换出（swapped out）。一旦这些页面被锁进内存，内核绝不会将它们换出至磁盘。对这些页面所进行的任何访问将不会导致页面出错。多数实时应用程序会将它们的部分或全部页面锁进物理内存。

Linux 为数据的预防出错和锁定提供了接口。第四章曾讨论过预防数据进入内存时出错的接口，而第八章会讨论将数据锁进物理内存的接口。

CPU 亲和性以及实时进程

实时应用程序第二个要注意的是多任务化（multitasking）问题。尽管 Linux 内核是可抢占的，但是它的调度程序并非总是能够立即重新安排一个进程以便有利于另一个进程。有时候，当前运行的进程正在内核的一个临界区（critical region）中运行，调度程序无法抢占它，除非它离开该关键区。如果正在等待运行的进程是实时进程，这样的延迟或许是不可接受的，因为马上就会越出操作期限。

因此，多任务化所带来的非决定论（indeterminism）本质上类似于页面调度机制的不可预测性。多任务化问题的解决方案也是一样的：消除它。当然，你可能无法消除所有其他进程。如果你的环境可以这么做，或许你一开始就不需要使用 —— 一个简单的自定义操作系统就够了。然而，如果你的系统具有多个处理器，你可以让你的实时进程专门使用其中一个或多个处理器。实际上，你可以让实时进程避开多任务化。

本章稍早我们曾讨论过用于操作一个进程的 CPU 亲和性（affinity）的系统调用。对实时应用程序所可能采用的一个优化手段是替每个实时进程保留一个处理器，并让所有其他进程以分时的方式共享其余的处理器。

达到此目标的最简单做法就是修改 Linux 的 *init* 程序 *SysVinit*（注 3），让它在引导进程之前先进行如下的工作：

```
cpu_set_t set;
int ret;

CPU_ZERO (&set);         /* 清除所有 CPU */
ret = sched_getaffinity (0, sizeof (cpu_set_t), &set);
if (ret == -1) {
        perror ("sched_getaffinity");
        return 1;
}

CPU_CLR (1, &set);       /* 禁用 CPU #1 */
ret = sched_setaffinity (0, sizeof (cpu_set_t), &set);
if (ret == -1) {
        perror ("sched_setaffinity");
        return 1;
}
```

这段程序代码会从 init 取得当前的可用处理器，预期是所有的处理器。然后从中移除一个处理器 CPU #1 并更新这组可用处理器。

注 3：　SysVinit 的源代码可以在 *ftp://ftp.cistron.nl/pub/people/miquels/sysvinit/* 找到。它受到 GNU 通用公共许可证第 2 版的保护。

因为子进程会从父进程处继承这组可用处理器，而且 init 是所有进程的父进程，所以系统的所有进程将运行在这组可用处理器之上。因此，没有进程会运行在 CPU #1 之上。

接着，修改你的实时进程，让它只运行在 CPU #1 之上：

```
cpu_set_t set;
int ret;

CPU_ZERO (&set);          /* 清除所有 CPU  */
CPU_SET (1, &set);        /* 使用 CPU #1 */
ret = sched_setaffinity (0, sizeof (cpu_set_t), &set);
if (ret == -1) {
        perror ("sched_setaffinity");
        return 1;
}
```

结果你的实时进程只会运行在 CPU #1 之上，而且所有其他的进程会运行在其他处理器之上。

资源限制

Linux 内核对进程加上了若干资源限制（*resource limit*）。这些资源限制是一个进程所能耗用的内核资源的硬性上限 —— 也就是所能打开的文件数目、内存页面的数目、未决信号的数目等。这些限制必须严格实施，内核不允许进程的资源耗用逾越硬性限制。举例来说，如果打开一个文件将导致一个进程所打开的文件数目逾越可用的资源限制，则 open() 调用会失败（注 4）。

Linux 提供了两个可用于操纵资源限制的系统调用。尽管 POSIX 将这两个接口纳入了标准，但是除了标准所规定的限制，Linux 还支持一些其他的限制。限制的检查与设定可分别通过 getrlimit() 与 setrlimit() 来进行：

```
#include <sys/time.h>
#include <sys/resource.h>

struct rlimit {
        rlim_t rlim_cur;  /* soft limit */
        rlim_t rlim_max;  /* hard limit */
};

int getrlimit (int resource, struct rlimit *rlim);
int setrlimit (int resource, const struct rlimit *rlim);
```

注 4： 在此情况下，open() 会把 errno 设定为 EMFILE，指出进程已经逾越了可打开文件数目的资源限制。参见第二章中对 open() 系统调用所做的讨论。

整数常量，例如 RLIMIT_CPU，用于表示资源。rlimit 结构用于表示实际的限制。此结构定义了两个上限值：软性限制（*soft limit*）以及硬性限制（*hard limit*）。内核会对进程实施软性资源限制，但是一个进程可以自由将它的软性限制变更成“范围从 0 到硬性限制的”任何值。一个不具 CAP_SYS_RESOURCE 能力的进程（也就是任何的非 root 进程）只能调低它的硬性限制。一个不具特权的进程无法调高它自己的硬性限制，甚至无法调回至之前的较高值；调低硬性限制的行为是不可逆的。一个具特权的进程可以将硬性限制设定成任何有效值。

实际的限制取决于所针对的是哪种资源。举例来说，如果 resource 是 RLIMIT_FSIZE，则实际的限制是一个进程所创建文件的尺寸上限（以字节计）。在此情况下，如果 rlim_cur 是 1,024，则一个进程创建或扩大一个文件时，其尺寸不得大于 1 KB。

所有的资源限制都具有两个特殊值：0 以及无限大（infinity）。前者用于完全禁用相应的资源。举例来说，如果 RLIMIT_CORE 被设为 0，则内核将不会创建 core 文件。反过来说，后者会移除资源上的任何限制。内核是通过 RLIM_INFINITY 这个特殊值（它恰巧就是 -1，这可能会导致一些混乱，因为返回值也是以 -1 来指出错误）来表示无限大。如果 RLIMIT_CORE 是无限大，内核将会创建任何大小的 core 文件。

函数 getrlimit() 会将特定资源（由 resource 指定）上当前的硬性和软性限制放进 rlim 所指向的结构。执行成功时，此调用会返回 0；执行失败时，此调用会返回 -1 并将 errno 设定成适当的值。

相应地，函数 setrlimit() 会将与 resource 相关的硬性和软性限制设定成 rlim 所指向的值。执行成功时，此调用会返回 0；而且内核会更新所要求的资源限制；执行失败时，此调用会返回 -1 并将 errno 设定成适当的值。

提供了哪些限制

Linux 当前提供了 15 种资源限制：

RLIMIT_AS

一个进程的地址空间的大小上限，以字节为单位。试图通过 mmap() 及 brk() 之类的调用增加地址空间的大小，让它逾越此限制，将导致失败以及返回 ENOMEM。如果进程的堆栈（会按需要自动增长）扩大时逾越此限制，内核会传送 SIGSEGV 信号给该进程。此限制通常为 RLIM_INFINITY。

RLIMIT_CORE

core 文件的大小上限，以字节为单位。如果为非零值，则当 core 文件大于此限制时会被截短成上限值；如果值为 0，则不会创建 core 文件。

RLIMIT_CPU

一个进程可耗用的 CPU 时间的上限，以秒为单位。如果一个进程的运行时间逾越此限制，则内核会传送 SIGXCPU 信号给它，让它能够加以捕捉和处理。具可移植性的程序收到此信号时应该终止运行，因为 POSIX 并未规定内核接下来该采取何种行动。有些系统可能会终止该进程，如果它想要继续运行下去的话。然而，Linux 会允许进程继续运行下去，并且以 1 秒的时间间隔持续送出 SIGXCPU 信号。一旦进程的运行时间逾越了硬性限制，它会收到 SIGKILL 信号并且终止运行。

RLIMIT_DATA

一个进程的数据段（data segment）与堆（heap）的上限，以字节为单位。试图通过 brk() 扩大数据段的大小，让它逾越此限制，将导致失败以及返回 ENOMEM。

RLIMIT_FSIZE

指定一个进程所能创建的文件的大小上限，以字节为单位。如果进程扩大一个文件的大小，让它逾越此限制，则内核会传送 SIGXFSZ 信号给该进程。默认情况下，此信号会终止该进程。然而，一个进程可能会选择捕捉并处理此信号，在此情况下，系统调用会失败并返回 EFBIG。

RLIMIT_LOCKS

一个进程所能持有的文件锁的数目上限（参见第七章中对文件锁所做的讨论）。一旦逾越限制，若想再获得其他的文件锁应该会失败并且返回 ENOLCK。然而，Linux 内核 2.4.25 版已删掉了此功能。在当前的内核里，此限制是可设定的，但是毫无效果。

RLIMIT_MEMLOCK

指定一个不具 CAP_SYS_IPC 能力的进程（也就是一个非 root 的进程）最多可经 mlock()、mlockall() 或 shmctl() 将几个字节锁进内存。如果此限制被逾越了，会导致这些调用失败并且返回 EPERM。在实践中，有效的限制将会是页面大小的整数倍。具 CAP_SYS_IPC 能力的进程可以将任何数目的页面锁进内存，于是此限制毫无作用。在内核 2.6.9 版之前，此限制是具 CAP_SYS_IPC 能力的进程可以锁进内存的页面数目的上限，而且不具特权的进程不能锁进任何页面。此限制并非 POSIX 的一部分，BSD 引进了此限制。

RLIMIT_MSGQUEUE

指定一个用户最多可替 POSIX 消息队列分配多少个字节。如果一个新创建的消息队列逾越了此限制，则 mq_open() 会失败并且返回 ENOMEM。此限制并非 POSIX 的一部分，它是在 2.6.8 版加入内核的，而且是 Linux 特有的限制。

RLIMIT_NICE

指定一个进程最多可将它的友善值调到多低（也就是将优先级调到多高）。如同本

章稍早所讨论的，一般进程只能调高其友善值（也就是调低其优先级）。此限制让管理者得以替进程调高其优先级的操作加上上限（也就是友善值的下限）。由于友善值可能是负的，内核会将此值解译成 20 - rlim_cur。因此，如果此限制被设定成 40，则一个进程可以将它的友善值调低至上限值 − 20（最高的优先级）。此限制是在 2.6.12 版加入内核的。

RLIMIT_NOFILE

指定一个进程所能打开的文件描述符数目的上限。试图逾越此限制会导致系统调用失败并且返回 EMFILE。此限制还可被指定成 RLIMIT_OFILE，这是 BSD 所取的名称。

RLIMIT_NPROC

指定任何时刻用户在系统上所能运行的进程数目的上限。试图逾越此限制将导致失败，而且 fork() 会返回 EAGAIN。此限制并非 POSIX 的一部分，这是 BSD 引进的。

RLIMIT_RSS

指定一个进程在内存中所能驻留的页面数目（所谓的 resident set size 或 RSS）的上限。只有早期的 2.4 版内核会实施此限制。虽然当前的内核版本允许设定此限制，但是并未实施。此限制不是 POSIX 的一部分，这是 BSD 引进的。

RLIMIT_RTPRIO

指定一个不具 CAP_SYS_NICE 能力的进程（也就是非 root 进程）所能请求的实时优先等级的上限。一般而言，不具特权的进程无法请求任何的实时调度类。此限制不是 POSIX 的一部分，它是在 2.6.12 版加入内核的，而且是 Linux 特有的限制。

RLIMIT_SIGPENDING

指定替此用户在队列中所暂存的（标准以及实时）信号数目的上限。试图在队列中暂存更多的信号将导致失败，而且 sigqueue() 之类的系统调用会返回 EAGAIN。注意，总是有不顾此限制，在队列中暂存“尚未排入队列的信号”（not-yet-queued signal）的一个实例。因此，总是有可能传送 SIGKILL 或 SIGTERM 信号给进程。此限制不是 POSIX 的一部分，它是 Linux 特有的限制。

RLIMIT_STACK

一个进程的堆栈大小的上限，以字节为单位。逾越此限制会导致传送 SIGSEGV 信号。

内核会替每个进程管理资源限制。一个子进程会在其父进程派生（fork）期间继承其父进程的限制；限制的维护可以跨 exec。

默认限制

你的进程所能使用的默认限制取决于三个变量：初始的软性限制、初始的硬性限制以及你的系统管理者。内核所规定的初始硬性和软性限制如表 6-1 所示。内核会在 init 进程上设定这些限制，而且因为子进程会继承其父进程的限制，所以所有随后的进程都会继承 init 的软性和硬性限制。

表 6-1：默认的软性和硬性资源限制

资源限制	软性限制	硬性限制
RLIMIT_AS	RLIM_INFINITY	RLIM_INFINITY
RLIMIT_CORE	0	RLIM_INFINITY
RLIMIT_CPU	RLIM_INFINITY	RLIM_INFINITY
RLIMIT_DATA	RLIM_INFINITY	RLIM_INFINITY
RLIMIT_FSIZE	RLIM_INFINITY	RLIM_INFINITY
RLIMIT_LOCKS	RLIM_INFINITY	RLIM_INFINITY
RLIMIT_MEMLOCK	8 个页面	8 个页面
RLIMIT_MSGQUEUE	800 KB	800 KB
RLIMIT_NICE	0	0
RLIMIT_NOFILE	1024	1024
RLIMIT_NPROC	0（表示无限制）	0（表示无限制）
RLIMIT_RSS	RLIM_INFINITY	RLIM_INFINITY
RLIMIT_RTPRIO	0	0
RLIMIT_SIGPENDING	0	0
RLIMIT_STACK	8 MB	RLIM_INFINITY

两件事会改变这些默认限制：

- 任何进程可以自由将一个软性限制增加成“范围从 0 到硬性限制的”任何值，或者减少一个硬性限制的值。派生期间，子进程将继承这些更新过的限制。
- 一个具特权的进程可自由将一个硬性限制设定成任何值。派生期间，子进程将继承这些更新过的限制。

在一般进程中不太可能出现一个 root 进程来变更任何硬性限制。因此，相较于第二件事，第一件事比较有可能成为限制被变更的原因。事实上，实际提供给一个进程的限制通常设定自用户的 shell，系统管理者可通过 shell 设置所要提供的各种限制。举例来说，在 Bourne-again shell（*bash*）中，管理者可经 *ulimit* 命令完成此事。请注意，管

理者不需要将值调低，他也可以将软性限制调高至硬性限制，给用户提供较合理的默认值。在许多系统中 RLIMIT_STACK 往往会被设定成 RLIM_INFINITY。

设定与取得限制

解释过各种资源限制之后，让我们来看看如何取得与设定限制。资源限制的取得相当简单：

```
struct rlimit rlim;
int ret;

/* 取得 core 文件大小的限制 */
ret = getrlimit (RLIMIT_CORE, &rlim);
if (ret == -1) {
        perror ("getrlimit");
        return 1;
}

printf ("RLIMIT_CORE limits: soft=%ld hard=%ld\n",
        rlim.rlim_cur, rlim.rlim_max);
```

在一个较大型的程序中编译这段程序代码并且运行它，会产生如下的输出：

```
RLIMIT_CORE limits: soft=0 hard=-1
```

我们所看到的软性限制为 0，而硬件限制为无限大（-1 表示 RLIM_INFINITY）。因此，我们可以将新的软性限制设定为任何值。此例将 core 文件的最大值设定为 32 MB：

```
struct rlimit rlim;
int ret;

rlim.rlim_cur = 32 * 1024 * 1024; /* 32 MB */
rlim.rlim_max = RLIM_INFINITY;     /* 听其自然 */
ret = setrlimit (RLIMIT_CORE, &rlim);
if (ret == -1) {
        perror ("setrlimit");
        return 1;
}
```

错误代码

发生错误时，errno 会被设定成下面的其中一个值：

EFAULT

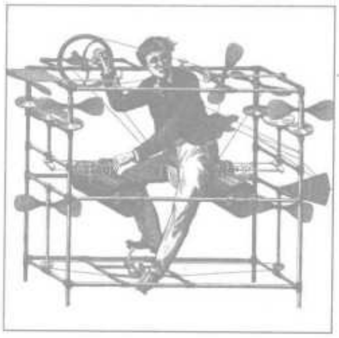
rlim 指向了无效或不可访问的内存。

EINVAL

resource 指定了无效的值,或者 rlim.rlim_cur 大于 rlim.rlim_max(只适用于 setrlimit())。

EPERM

调用者不具 CAP_SYS_RESOURCE 能力，但是试图调高硬性限制。



第七章 文件和目录管理

在第二章、第三章以及第四章中，我们详细探讨了文件 I/O 的接口。本章中，我们将再次回到文件的议题，但是这一次的重点并非文件的读取或写入操作，而是文件与其元数据的操作与管理。

文件与其元数据

如第一章所探讨的，每个文件是通过一个 *inode*（信息节点）被引用的，对文件系统而言这是一个独一无二的数值，也就是所谓的 *inode number*（信息节点编号）。inode 既是一个实际的对象（被放在磁盘上Unix 风格的文件系统中），也是一个概念性实体（由 Linux 内核里的一个数据结构来表示）。inode 可用来存储与某个文件有关的元数据，例如文件的使用权限、上一次的访问时间戳、拥有者、组、长度以及文件内容的摆放位置。

对 *ls* 命令使用 *-i* 标志可以取得文件的 inode number：

```
$ ls -i
1689459 Kconfig     1689461 main.c      1680144 process.c  1689464 swsusp.c
1680137 Makefile    1680141 pm.c        1680145 smp.c      1680149 user.c
1680138 console.c   1689462 power.h     1689463 snapshot.c
1689460 disk.c      1680143 poweroff.c  1680147 swap.c
```

如此输出所示，*disk.c* 的 inode number 为 1689460。在这个特别的文件系统上，不会有其他文件具备此 inode number。然而，在不同的文件系统上，我们就无法保证了。

stat 系列函数

Unix 提供了一系列的函数可用于取得文件的元数据：

```
#include <sys/types.h>
#include <sys/stat.h>
```

```
#include <unistd.h>

int stat (const char *path, struct stat *buf);
int fstat (int fd, struct stat *buf);
int lstat (const char *path, struct stat *buf);
```

这些函数都会返回与文件有关的信息。stat() 会返回 *path* 所代表文件的相关信息，而 fstat() 会返回文件描述符 fd 所代表文件的相关信息。lstat() 如同 stat()，除了关于符号链接的情况：lstat() 所返回的是关于链接本身（而非目标文件）的信息。

这些函数都会将信息存入一个 stat 结构（由用户提供）。stat 结构定义于 <bits/stat.h>，可以从 <sys/stat.h> 来引用它：

```
struct stat {
        dev_t st_dev;         /* ID of device containing file */
        ino_t st_ino;         /* inode number */
        mode_t st_mode;       /* permissions */
        nlink_t st_nlink;     /* number of hard links */
        uid_t st_uid;         /* user ID of owner */
        gid_t st_gid;         /* group ID of owner */
        dev_t st_rdev;        /* device ID (if special file) */
        off_t st_size;        /* total size in bytes */
        blksize_t st_blksize; /* blocksize for filesystem I/O */
        blkcnt_t st_blocks;   /* number of blocks allocated */
        time_t st_atime;      /* last access time */
        time_t st_mtime;      /* last modification time */
        time_t st_ctime;      /* last status change time */
};
```

此结构的字段细节如下所述：

- st_dev 字段描述了包含文件的设备节点（本章稍后会说明设备节点）。如果文件的背后不是一个设备 —— 例如，文件位于经 NFS 挂载的文件系统，则此值为 0。
- st_ino 字段提供了文件的 inode number。
- st_mode 字段提供了文件的模式字节。第一章和第二章说明了模式字节与权限位。
- st_nlink 字段提供了指向文件的硬链接数目。每个文件至少都会有一个硬链接。
- st_uid 字段提供了拥有文件的用户的用户 ID。
- st_gid 字段提供了拥有文件的组的组 ID。
- 如果文件是一个设备节点，则 st_rdev 字段描述了此文件所代表的设备。
- st_size 字段提供了文件的大小，以字节为单位。

- st_blksize 字段描述了有效率的文件 I/O 首选的块大小。此值（或是一个整数倍）是用户缓冲式 I/O 的理想块大小（参见第三章）。
- st_blocks 字段提供了分配给此文件的文件系统块的数目。如果此文件有空洞（也就是说，如果这是一个稀疏文件），则此值会小于 st_size 字段所提供的值。
- st_atime 字段包含了上一次文件被访问的时间 —— 也就是此文件最近一次被访问（例如，通过 read() 或 execle()）的时间。
- st_mtime 字段包含了上一次文件被修改的时间 —— 也就是此文件最近一次被写入的时间。
- st_ctime 字段包含了上一次文件被变更的时间。这通常会被误解成文件被创建的时间，在 Linux 或其他 Unix 风格的系统上并未保存此值。此字段实际上描述最近一次文件的元数据（例如它的拥有者或使用权限）被变更的时间。

执行成功时，这三个调用都会返回 0，而且会将文件的元数据存入用户所提供的 stat 结构；发生错误时，它们都会返回 -1 并且会将 errno 设定成下面的其中一个值：

EACCESS

进行调用的进程对 path 中的一个目录元素不具搜寻权限（仅适用于 stat() 和 lstat()）。

EBADF

fd 是个无效值（仅适用于 fstat()）。

EFAULT

path 或 buf 是个无效指针。

ELOOP

path 中包含了太多的符号链接（仅适用于 stat() 和 lstat()）。

ENAMETOOLONG

path 太长（仅适用于 stat() 和 lstat()）。

ENOENT

path 中有一个元素不存在（仅适用于 stat() 和 lstat()）。

ENOMEM

内存的可用空间不足以完成此项请求。

ENOTDIR

path 中有一个元素不是目录（仅适用于 stat() 和 lstat()）。

下面的范例程序会使用 stat() 来取得命令行所指定的文件的大小信息：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>

int main (int argc, char *argv[])
{
        struct stat sb;
        int ret;

        if (argc < 2) {
                fprintf (stderr,
                         "usage: %s <file>\n", argv[0]);

                return 1;
        }

        ret = stat (argv[1], &sb);
        if (ret) {
                perror ("stat");
                return 1;
        }

        printf ("%s is %ld bytes\n",
                argv[1], sb.st_size);

        return 0;
}
```

下面是对程序本身的源文件运行此程序的结果：

```
$ ./stat stat.c
stat.c is 392 bytes
```

下面这段程序代码会使用 fstat() 来检查一个已经打开的文件是否位于一个物理（相对于网络而言）设备上：

```
/*
 * is_on_physical_device —— 如果 fd 位于一个物理设备上，
 * 则返回一个正整数；如果文件位于一个非物理或虚拟设备（例如
 * 经 NFS 挂载）上，则返回 0；如果发生错误，则返回 -1。
 */
int is_on_physical_device (int fd)
{
        struct stat sb;
        int ret;

        ret = fstat (fd, &sb);
        if (ret) {
                perror ("fstat");
```

```
                return -1;
        }

        return gnu_dev_major (sb.st_dev);
}
```

使用权限

除了可以使用 stat 系列调用来取得特定文件的使用权限，我们还可以使用另外两个系统调用来设定这些值：

```
#include <sys/types.h>
#include <sys/stat.h>

int chmod (const char *path, mode_t mode);
int fchmod (int fd, mode_t mode);
```

chmod() 与 fchmod() 可将一个文件的使用权限设定成 mode。使用 chmod() 时，path 代表所要修改文件的相对路径或绝对路径名称。使用 fchmod() 时，此文件由文件描述符 fd 指定。

mode 的合法值，由 mode_t 整数类型来表示，如同 stat 结构中 st_mode 字段所返回的那些值。尽管这些值都只是简单的整数，但是它们的意义因 Unix 的实现而异。因此，POSIX 定义了一组常量用于表示这些使用权限（相关细节参见“新文件的使用权限”一节）。这些常量可经二元 OR 逻辑运算组合成 mode 的合法值。例如，(S_IRUSR | S_IRGRP) 可将文件的使用权限设定成拥有者与组具有的读取权限。以 chmod() 或 fchmod() 变更一个文件的使用权限时，进行调用的进程的有效 ID 必须与文件的拥有者相符，或者进行调用的进程必须具备 CAP_FOWNER 能力。

执行成功时，这两个调用都会返回 0；执行失败时，这两个调用都会返回 -1 并将 errno 设定成下面的其中一个错误值：

EACCESS
: 进行调用的进程对 path 中的一个元素不具搜索权限（仅适用于 chmod()）。

EBADF
: 文件描述符 fd 是一个无效值（仅适用于 fchmod()）。

EFAULT
: path 是一个无效的指针（仅适用于 chmod()）。

EIO
: 文件系统上发生了一个内部的 I/O 错误。这是一个非常严重的错误，可能代表磁盘或文件系统坏掉了。

ELOOP

解析 path 的时候，内核遇到太多的符号链接（仅适用于 chmod()）。

ENAMETOOLONG

path 太长了（仅适用于 chmod()）。

ENOENT

path 并不存在（仅适用于 chmod()）。

ENOMEM

内存的可用空间不足以完成此项请求。

ENOTDIR

path 中的一个元素并非目录（仅适用于 chmod()）。

EPERM

进行调用的进程的有效ID与文件的拥有者不相符，以及进程不具有 CAP_FOWNER 的能力。

EROFS

文件位于只读文件系统上。

下面这段程序代码会将文件 *map.png* 设定成拥有者具有读取与写入权限：

```
int ret;

/*
 * 将当前目录中的 map.png 设定成拥有者具有
 * 读取与写入权限。此结果等效于
 * chmod 600 ./map.png。
 */
ret = chmod ("./map.png", S_IRUSR | S_IWUSR);
if (ret)
        perror ("chmod");
```

下面的程序代码所完成的是一样的工作，但是会以 fd 来代表已打开的文件 *map.png*：

```
int ret;

/*
 * 将 fd 背后的文件设定成拥有者具有
 * 读取和写入权限。
 */
ret = fchmod (fd, S_IRUSR | S_IWUSR);
if (ret)
        perror ("fchmod");
```

文件的拥有者 *kidd* 不会被变更，因为这段程序代码将 `-1` 传入了 `uid`。

下面这个函数会将 `fd` 所代表的文件的拥有者和组设定为 root：

```
/*
 * make_root_owner —— 将 fd 所指定文件的拥有者和组
 * 设定为 root。执行成功时会返回 0，执行失败时会返回 -1。
 */
int make_root_owner (int fd)
{
        int ret;

        /* root 的 gid 与 uid 皆为 0 */
        ret = fchown (fd, 0, 0);
        if (ret)
                perror ("fchown");

        return ret;
}
```

进行调用的进程必须具备 `CAP_CHOWN` 能力。这通常意味着它必须为 root 所拥有。

扩展属性

扩展属性（*extended attributes*）也称为 *xattrs*，提供了一个机制用来将键／值对（key/value pair）永久地关联到文件。本章中，我们已经探讨过关联到文件的各种键／值元数据：文件的大小、拥有者、上一次的修改时间等。扩展属性让现有的文件系统得以支持原始设计中未提供的功能，例如基于安全而必须进行的访问控制。扩展属性之所以值得注意，是因为用户空间应用程序可以任意地对"键／值对"进行创建、读取和写入操作。

扩展属性是文件系统不可知论者（*filesystem-agnostic*），这意味着应用程序可以使用一个标准的接口来操纵它们，此接口不因文件系统而异。因此，应用程序可以使用扩展属性而不必在意文件被放在文件系统之上，或者文件系统内部如何存储键与值。然而，经扩展的属性的实现与文件系统关系密切。不同的文件系统会以极不同的方式来存储扩展属性，但是内核会隐藏这些差异，将它们抽离并放在扩展属性的接口之后。

举例来说，*ext3* 文件系统会将一个文件的扩展属性存入文件的 inode 的可用空间里（注 1）。此功能让扩展属性的读取变得非常快。因为每当应用程序访问一个文件时，会从磁盘将包含 inode 的文件系统块读进内存，而扩展属性会被"自动"读进内存，所以它们的访问不需要任何额外的开销。

注 1：　当然，直到 inode 的可用空间被用完。接着 *ext3* 会将扩展属性存储到额外的文件系统块。较旧版的 *ext3* 并未提供将扩展属性存入 inode 的功能。

其他的文件系统，例如 *FAT* 以及 *minixfs*，根本就不支持扩展属性。针对这些文件系统上的文件进行扩展属性的操作，这些文件系统将会返回 ENOTSUP。

键与值

每个扩展属性可通过一个独一无二的键（*key*）来区分。键的内容必须是有效的 UTF-8。键的格式为 *namespace.attribute*。每个键必须是使用完全限定（fully qualified）的形式，也就是开头必须是一个有效的命名空间，而且后面紧跟着一个点号。例如，*user.mime_type* 便是一个有效的键，此键位于 *user* 命名空间，而且具有 *mime_type* 属性（attribute）。

一个键（key）可能是已定义（*defined*）或者是未定义（*undefined*）。如果一个键已定义，那么它可能被赋值（非空值）也可能未被赋值（空值）。也就是说，一个已定义的键尽管未被赋值，但并不等于未定义。正如我们将看到的，这意味着一个键的移除需要通过一个特殊的接口（只是将一个空值指派给它是不够的）。

与一个键相关联的值，如果不是空的，它可能是任意字节的数组。因为此值未必是一个字符串，所以它不需要以 null 结尾，尽管以 null 结尾绝对是合理的，如果你打算以一个 C 字符串作为一个键的值的话。既然此值未必是以 null 结尾，那么对扩展属性所进行的任何操作都需要知道值的大小。对一个属性进行读取操作时，内核会提供它的大小；对一个属性进行写入操作时，你必须自己提供它的大小。

Linux 并未对"键的数目、一个键的长度、一个值的大小以及与一个文件相关联的所有键与值所能使用的空间"做任何的限制。然而，文件系统实际上是有限制的。这会导致与特定文件相关联的所有键与值所能使用的空间受到限制。

存储 MIME 类型的更好方法

GUI 的文件管理程序，例如 GNOME 的 Nautilus，其行为因文件类型而异：独一无二的图标（icon）、不同的默认单击（click）行为、所能进行的操作等。要完成此事，文件管理程序必须知道每个文件的格式。要判断文件的格式，Windows 之类的文件系统仅需要查看文件的扩展名即可。然而，基于传统和安全性，Unix 系统往往会查看文件的内容来判断它的类型。此过程称为 MIME 类型嗅探（*MIME type sniffing*）。

有些文件管理程序会直接产生此信息；有些则会将所产生的信息缓存起来以备下次使用。这些会缓存信息的文件管理程序往往会把信息放到特制的数据库中。文件管理程序必须维持此数据库与文件同步，因为文件可能会在文件管理程序不知道的情况下被变更。一个更好的做法就是舍弃特制的数据库，将此类元数据存入扩展属性：这样更容易维护，访问速度更快，而且任何应用程序都很容易访问。

以 *ext3* 为例，特定文件的所有扩展属性可以放进文件的 inode 里的可用空间以及一个额外的文件系统块（较旧版的 *ext3* 只会使用一个文件系统块，而不会用到 inode 里的存储空间）。每个文件的实际限制大约为 1 KB 到 8 KB，这取决于文件系统块的大小。相对而言，*XFS* 并无实际的限制。然而，即使是 *ext3*，这些限制通常也不是问题，因为键与值多半是简单的文本字符串。但是别忘了 —— 当你将一个项目的整个修订版控制历史存入一个文件的扩展属性时，请先三思！

扩展属性的命名空间

与扩展属性相关的命名空间只是一个组织工具。内核可以根据命名空间来实施不同的访问策略。

Linux 当前定义了四种扩展属性命名空间，而且未来可能会定义更多的命名空间。这四种命名空间如下所示：

system

system 命名空间可用于实现利用扩展属性的内核功能，例如访问控制列表（access control list，常简写为 ACL）。举例来说，*system.posix_acl_access* 便是一个位于此命名空间的扩展属性。用户是否可以读取或写入这些属性取决于所使用的安全模块。假设在最坏的情况下，任何用户（包括 root）都无法读取这些属性。

security

security 命名空间可用于实现安全模块，例如 SELinux。用户空间应用程序是否可以访问这些属性同样取决于所使用的安全模块。默认情况下，所有进程可以读取这些属性，但是只有具备 `CAP_SYS_ADMIN` 能力的进程可以对它们进行写入操作。

trusted

trusted 命名空间会把受限制的信息存入用户空间。只有具备 `CAP_SYS_ADMIN` 能力的进程可以对这些属性进行读取或写入操作。

user

user 命名空间是一般进程所使用的标准命名空间。内核会经一般文件权限位来控制此命名空间的访问。要从一个现有的键读取值，一个进程必须对相应的文件具有读取的权限。为了创建一个新键或将值写入一个现有的键，一个进程必须对相应的文件具有写入的权限。你只能对常规文件（而非符号链接或设备文件）进行 user 命名空间中扩展属性的赋值。如果要设计一个使用扩展属性的用户空间应用程序，这可能是你想使用的命名空间。

扩展属性的操作

POSIX 定义了 4 项操作，让应用程序可以用于处理特定文件的扩展属性：

- 指定一个文件以及一个键，返回相应的值。
- 指定一个文件、一个键以及一个值，将值赋给键。
- 指定一个文件，返回文件扩展属性的所有键。
- 指定一个文件以及一个键，从文件移除该扩展属性。

POSIX 又分别替每项操作提供 3 种系统调用（总共有 12 种系统调用）：

- 一个系统调用用于操作所指定的路径名称：如果该路径引用了一个符号链接，则操作该链接的目标文件（标准行为）。
- 一个系统调用用于操作所指定的路径名称：如果该路径引用了一个符号链接，则操作该链接本身（此系统调用的名称前缀字母 l ）。
- 一个系统调用用于操作文件描述符（此系统调用的称前缀字母 f ）。

接下来说明这 12 种系统调用。

取得一个扩展属性

最简单的操作就是从一个文件返回特定扩展属性（所指定的键）的值：

```
#include <sys/types.h>
#include <attr/xattr.h>

ssize_t getxattr (const char *path, const char *key,
                  void *value, size_t size);
ssize_t lgetxattr (const char *path, const char *key,
                   void *value, size_t size);
ssize_t fgetxattr (int fd, const char *key,
                   void *value, size_t size);
```

执行成功时，getxattr() 会从文件 path 将扩展属性 key 的值存入所提供的缓冲区 value 中，缓冲区的长度为 size 个字节，而且会返回此值的实际大小。

如果 size 为 0，则此调用只会返回值的大小，而不会将它存入 value。这让应用程序可以决定缓冲区的大小，以便用来存储键的值。于是应用程序可以按照需要分配缓冲区或是重新调整缓冲区的大小。

lgetxattr() 的行为如同 getxattr()，但是当 path 是一个符号链接时，它会返回链接本身（而非链接的目标文件）的扩展属性。正如前一节所提到的，user 命名空间中的属性无法应用于符号链接 —— 因此，此调用很少被用到。

fgetxattr() 的行为如同 getxattr()，但是 fgetxattr() 所操作的是文件描述符 fd。

发生错误时，这三种调用都会返回 -1 并将 errno 设定为下面的其中一个值：

EACCESS
: 进行调用的进程对 path 中的目录元素不具搜索权限（仅适用于 getxattr() 与 lgetxattr()）。

EBADF
: fd 是无效的（仅适用于 fgetxattr()）。

EFAULT
: path、key 或 value 是一个无效的指针。

ELOOP
: path 中包含了太多的符号链接（仅适用于 getxattr() 与 lgetxattr()）。

ENAMETOOLONG
: path 太长（仅适用于 getxattr() 与 lgetxattr()）。

ENOATTR
: 属性 key 并不存在，或者进程没有办法访问该属性。

ENOENT
: path 中有一个元素不存在（仅适用于 getxattr() 与 lgetxattr()）。

ENOMEM
: 内存的可用空间不足以完成此项请求。

ENOTDIR
: path 中有一个元素不是目录（仅适用于 getxattr() 与 lgetxattr()）。

ENOTSUP
: 包含 path 或 fd 的文件系统并不支持扩展属性。

ERANGE
: size 太小以至于无法保存 key 的值。如先前所做的讨论，应用程序可以再进行一次系统调用，但是这次要把 size 设定成 0。返回值将指出所需的缓冲区大小，于是 value 可以被重新调整成合适的大小。

设定一个扩展属性

下面三种系统调用可用于设定特定的扩展属性：

```
#include <sys/types.h>
#include <attr/xattr.h>

int setxattr (const char *path, const char *key,
              const void *value, size_t size, int flags);
int lsetxattr (const char *path, const char *key,
               const void *value, size_t size, int flags);
int fsetxattr (int fd, const char *key,
               const void *value, size_t size, int flags);
```

执行成功时，setxattr() 会将文件 path 上的扩展属性 key 设定成 value（长度为 size 个字节）。flags 字段用于改变调用的行为。当 flags 的值为 XATTR_CREATE 时，如果扩展属性已经存在，则此调用会失败。当 flags 的值为 XATTR_REPLACE 时，如果扩展属性尚不存在，则此调用会失败。默认行为（当 flags 的值为 0 时）就是允许创建与取代的动作。无论 flags 的值是什么，key 以外的键都不会受到影响。

lsetxattr() 的行为如同 setxattr()，但是当 path 是一个符号链接时，lsetxattr() 所设定的是链接本身（而不是链接的目标文件）的扩展属性。别忘了，user 命名空间中的属性无法应用于符号链接 —— 因此，此调用很少被用到。

fsetxattr() 的行为如同 setxattr()，但是 fsetxattr() 所操作的是文件描述符 fd。

执行成功时，这三种系统调用都会返回 0；执行失败时，它们都会返回 -1 并将 errno 设定为下面的其中一个值：

EACCESS

进行调用的进程对 path 中的一个目录元素不具搜索的权限(仅适用于 setxattr() 与 lsetxattr())。

EBADF

fd 是无效的（仅适用于 fsetxattr()）。

EDQUOT

由于配额的限制导致所请求的操作无法取得所需的空间。

EEXIST

flags 的值被设定为 XATTR_CREATE，但是在指定的文件上 key 已经存在。

EFAULT

path、key 或 value 是一个无效的指针。

EINVAL

flags 是无效的。

ELOOP

path 中包含了太多的符号链接（仅适用于 setxattr() 与 lsetxattr()）。

ENAMETOOLONG

path 太长（仅适用于 setxattr() 与 lsetxattr()）。

ENOATTR

flags 的值被设定为 XATTR_REPLACE，但是指定的文件上 key 尚不存在。

ENOENT

path 中有一个元素并不存在（仅适用于 setxattr() 与 lsetxattr()）。

ENOMEM

内存的可用空间不足以完成此项请求。

ENOSPC

文件系统上的空间不足以存储该扩展属性。

ENOTDIR

path 中有一个元素不是目录（仅适用于 setxattr() 与 lsetxattr()）。

ENOTSUP

包含 path 或 fd 的文件系统并不支持扩展属性。

列出一个文件上的扩展属性

下面三种系统调用可用于列出指派给特定文件的扩展属性列表：

```
#include <sys/types.h>
#include <attr/xattr.h>

ssize_t listxattr (const char *path,
                   char *list, size_t size);
ssize_t llistxattr (const char *path,
                    char *list, size_t size);
ssize_t flistxattr (int fd,
                    char *list, size_t size);
```

执行成功时，listxattr() 会返回文件 path 上的扩展属性列表。这份列表会存入 list 所提供的缓冲区（长度为 size 个字节）。此系统调用返回列表的实际大小，以字节为单位。

每个扩展属性的键会被传入 list 而且会以一个 null 字符结尾，所以这份列表会像这个样子：

```
"user.md5_sum\0user.mime_type\0system.posix_acl_default\0"
```

因此，尽管每个键是一个传统的、以 null 结尾的 C 字符串，但是你仍需要知道整个列表的长度（你可以从此调用的返回值取得此长度），以便处理这份由键所构成的列表。想知道你需要分配多大的缓冲区，进行调用时请把 size 的值设为 0。这会导致所调用的函数返回列表的实际长度。如同 getxattr()，应用程序必须将适当的值传入 value 以便分配缓冲区或调整缓冲区的大小。

llistxattr() 的行为如同 listxattr()，但是当 path 是一个符号链接时，llistxattr() 所返回的是与链接本身（而不是链接的目标文件）有关的扩展属性。别忘了，user 命名空间中的属性无法应用于符号链接 —— 因此，这个调用很少被用到。

flistxattr() 的行为如同 listxattr()，但是 flistxattr() 所操作的是文件描述符 fd。

执行失败时，这三种调用都会返回 -1 并将 errno 设定为下面的其中一个错误代码：

EACCESS
: 进行调用的进程对 path 中的一个目录元素不具搜索权限（仅适用于 listxattr() 与 llistxattr()）。

EBADF
: fd 是无效的（仅适用于 flistxattr()）。

EFAULT
: path 或 list 是一个无效的指针。

ELOOP
: path 中包含了太多的符号链接(仅适用于 listxattr() 与 llistxattr())。

ENAMETOOLONG
: path 太长（仅适用于 listxattr() 与 llistxattr()）。

ENOENT
: path 中的一个元素并不存在（仅适用于 listxattr() 与 llistxattr()）。

ENOMEM
: 内存的可用空间不足以完成此项请求。

ENOTDIR
: path 中的一个元素并非目录（仅适用于 listxattr() 与 llistxattr()）。

ENOTSUPP
: 包含 path 或 fd 的文件系统并不支持扩展属性。

ERANGE

size 为非零值，但是大小不足以保存整份列表。应用程序可以再进行一次系统调用，但是这次要把 size 设成 0，以便取得列表的实际大小。然后程序可以调整 value 的大小，以便再进行一次系统调用。

移除一个扩展属性

最后三种系统调用可用于从特定的文件移除所指定的键：

```
#include <sys/types.h>
#include <attr/xattr.h>

int removexattr (const char *path, const char *key);
int lremovexattr (const char *path, const char *key);
int fremovexattr (int fd, const char *key);
```

执行成功时，removexattr() 会从文件 path 移除扩展属性 key。别忘了，一个未定义的键不同于一个具有空值（长度为零）的已定义的键。

lremovexattr() 的行为如同 removexattr()，但是当 path 是一个符号链接时，lremovexattr() 会移除与链接本身（而不是链接的目标文件）相关的扩展属性。别忘了，user 命名空间中的属性无法应用于符号链接 —— 因此，这个调用也很少被用到。

fremovexattr() 的行为如同 removexattr()，但是 fremovexattr() 所操作的是文件描述符 fd。

执行成功时，这三种调用都会返回 0；执行失败时，它们都会返回 -1 并将 errno 设定为下面的其中一个值：

EACCESS

进行调用的进程对 path 中的一个目录元素不具搜索权限（仅适用于 removexattr() 与 lremovexattr()）。

EBADF

fd 是无效的（仅适用于 fremovexattr()）。

EFAULT

path 或 key 是一个无效的指针。

ELOOP

path 中包含了太多的符号链接（仅适用于 removexattr() 与 lremovexattr()）。

ENAMETOOLONG

path 太长（仅适用于 removexattr() 与 lremovexattr()）。

ENOATTR

`key` 并不存在于所指定的文件中。

ENOENT

`path` 中的一个元素并不存在（仅适用于 `removexattr()` 与 `lremovexattr()`）。

ENOMEM

内存的可用空间不足以完成此项请求。

ENOTDIR

`path` 中的一个元素并非目录(仅适用于 `removexattr()` 与 `lremovexattr()`)。

ENOTSUPP

包含 `path` 或 `fd` 的文件系统并不支持扩展属性。

目录

在 Unix 中，目录（*directory*）是一个简单的概念：它包含了一串文件名称，每个文件名称会被映射到一个 inode number。每个名称又称为一个目录项（*directory entry*），而且每个“名称至 inode”的映射（mapping）称为一个链接（*link*）。一个目录的内容 —— 使用 *ls* 所看到的结果 —— 就是该目录中所有文件名称所构成的列表。当用户在指定的目录中打开一个文件时，内核会在目录列表中寻找该文件的名称，试图找到相应的 inode number。然后内核会把该 inode number 传递给文件系统，文件系统会利用它来找到文件在设备上的物理位置。

目录中还可以包含其他目录。子目录（*subdirectory*）就是一个目录中所包含的另一个目录。基于此定义，所有的目录都是某个父目录（*parent directory*）的子目录，位于文件系统树最根部的目录（*/*）除外。毫不意外，此目录就称为根目录（*root directory*），不要与 root 的 home 目录（*/root*）搞混了。

路径名称是由一个文件名称与它的一个或多个父目录名称所组成。绝对路径名称就是以根目录开头的路径名称，例如 */usr/bin/sextant*。相对路径名称就是并非以根目录开头的路径名称，例如 *bin/sextant*。要使用这样的路径名称，操作系统必须知道该路径相对于哪个目录。当前工作目录（下一节会讨论）可作为起始点。

字符“/”（在路径名称中代表目录）以及 *null*（在路径名称中代表结尾）除外，文件和目录名称可以包含任何字符。也就是说，标准的做法就是将路径名称中的字符限定为当前的 locale 设定下有效的可打印字符，甚至只是 ASCII。然而，因为内核以及 C 链接库都不会进行此工作，所以应用程序只能使用有效的可打印字符。

在较旧版的 Unix 系统中，文件名称限定为 14 个字符。今日，所有现代的 Unix 文件系统允许每个文件名称至少可以有 255 个字节（注 2）。Linux 下的许多文件系统甚至允许更长的文件名称（注 3）。

每个目录包含两个特殊目录："." 以及 ".."（分别称为 *dot* 以及 *dot-dot*）。dot（点）目录引用目录本身，而 dot-dot（点点）目录则引用目录的父目录。例如，目录 */home/kidd/gold/..* 如同目录 */home/kidd* 。根目录的 dot 以及 dot-dot 目录则指向根目录本身 —— 也就是 /、/. 以及 /.. 皆为同一个目录。因此，就技术而言，你可以说根目录也是它自己的子目录。

当前工作目录

每个进程都有一个当前目录，该目录最初继承自父进程。该目录又称为进程的当前工作目录（*current working directory*，常简写为 cwd）。当前工作目录是内核用于解析相对路径名称的起点。举例来说，如果一个进程的当前工作目录是 */home/blackbeard*，而且该进程试图打开 *parrot.jpg*，于是内核将试图打开 */home/blackbeard/parrot.jpg*。相对地，如果进程试图打开 */usr/bin /mast*，则内核会打开 */usr/bin/mast* —— 当前工作目录不会影响绝对路径名称（也就是以一个斜线符号开头的路径名称）。

一个进程可以取得并变更它的当前工作目录。

取得当前工作目录

取得当前工作目录的首选方法就是使用经 POSIX 标准化的 `getcwd()` 系统调用：

```
#include <unistd.h>

char * getcwd (char *buf, size_t size);
```

执行成功时，`getcwd()` 会将当前工作目录（一个绝对路径名称）复制到 `buf` 所指向的缓冲区（长度为 `size` 个字节），而且会返回一个指向 `buf` 的指针。执行失败时，此调用会返回 `NULL`，并且会将 `errno` 设定为下面的其中一个值：

注 2：　请注意，此限制为 255 个字节，而不是 255 个字符。多字节字符在这 255 个字符中显然会用掉 1 个以上的字节。

注 3：　当然，为了向下兼容，Linux 所提供的较旧文件系统，例如 FAT，仍会延续它们自己的限额。以 FAT 的情况来说，此限额为 8 个字符，后面跟着一个点号，点号后面再跟着 3 个字符。是的，以点号作为文件系统的一个特殊字符是个愚蠢的做法。

EFAULT

buf 是一个无效的指针。

EINVAL

size 为 0，但是 buf 不为 NULL。

ENOENT

当前工作目录不再有效。如果当前工作目录被移除就会发生此情况。

ERANGE

size 太小而无法在 buf 中保存当前工作目录。应用程序需要分配一个较大的缓冲区，再试一次。

下面是一个使用 getcwd() 的例子：

```
char cwd[BUF_LEN];

if (!getcwd (cwd, BUF_LEN)) {
        perror ("getcwd");
        exit (EXIT_FAILURE);
}

printf ("cwd = %s\n", cwd);
```

依据 POSIX 的规定，如果 buf 为 NULL，则 getcwd() 的行为未定义。Linux 的 C 链接库在此情况下会分配一个长度为 size 个字节的缓冲区，而且会把当前工作目录存储到该处。如果 size 为 0，C 链接库会分配一个足以存储当前工作目录的缓冲区。然后应用程序的责任是在它完成工作后，经 free() 释放缓冲区。因为这是 Linux 特有的行为，重视可移植性或严格遵守 POSIX 的应用程序不应该使用此功能。然而，此功能让 getcwd() 的使用变得非常简单！下面便是一个例子：

```
char *cwd;

cwd = getcwd (NULL, 0);
if (!cwd) {
        perror ("getcwd");
        exit (EXIT_FAILURE);
}

printf ("cwd = %s\n", cwd);

free (cwd);
```

Linux 的 C 链接库还提供了一个 get_current_dir_name() 函数。当 buf 为 NULL 而且 size 为 0 时，此函数的行为如同 getcwd()：

```
#define _GNU_SOURCE
#include <unistd.h>

char * get_current_dir_name (void);
```

因此，下面的范例程序代码等效于上面的范例程序代码：

```
char *cwd;

cwd = get_current_dir_name ();
if (!cwd) {
        perror ("get_current_dir_name");
        exit (EXIT_FAILURE);
}

printf ("cwd = %s\n", cwd);

free (cwd);
```

较旧版的 BSD 系统使用的是 `getwd()` 调用，为了向下兼容，Linux 也提供此调用：

```
#define _XOPEN_SOURCE_EXTENDED /* or _BSD_SOURCE */
#include <unistd.h>

char * getwd (char *buf);
```

`getwd()` 会将当前工作目录复制到 `buf`（长度必须至少为 `PATH_MAX` 个字节）。执行成功时，此调用会返回 `buf`；执行失败时，它会返回 `NULL`。例如：

```
char cwd[PATH_MAX];

if (!getwd (cwd)) {
        perror ("getwd");
        exit (EXIT_FAILURE);
}

printf ("cwd = %s\n", cwd);
```

基于可移植性与安全性的理由，应用程序不应该使用 `getwd()`，最好使用 `getcwd()`。

变更当前工作目录

当一个用户首次登录她的系统，login 进程会根据 */etc/passwd* 文件中的设定将她的当前工作目录设定成她的 home 目录。然而，有些时候，一个进程会想变更它的当前工作目录。例如，当用户键入 *cd*，shell 可能会想这么做。

Linux 提供了两个系统调用用于变更当前工作目录，一个接受目录的路径名称，而另一个接受代表已打开目录的文件描述符：

```
#include <unistd.h>

int chdir (const char *path);
int fchdir (int fd);
```

chdir() 可用于将当前工作目录变更为 path 所指定的路径名称（可以是绝对的也可以是相对的）。同样地，fchdir() 可用于将当前工作目录变更为 fd（必须对应到已打开的目录）所代表的路径名称。执行成功时，这两个调用都会返回 0；执行失败时，这两个调用都会返回 -1。

执行失败时，chdir() 还会将 errno 设定成下面的其中一个值：

EACCESS
: 进行调用的进程对 path 中的一个目录元素不具搜索权限。

EFAULT
: path 是一个无效的指针。

EIO
: 发生了一个内部的 I/O 错误。

ELOOP
: 解析 path 的时候，内核遇到了太多的符号链接。

ENAMETOOLONG
: path 太长。

ENOENT
: path 指向的目录并不存在。

ENOMEM
: 内存的可用空间不足以完成此项请求。

ENOTDIR
: path 中有一个或多个元素不是目录。

fchdir() 会将 errno 设定为下面的其中一个值：

EACCESS
: 进行调用的进程对 fd 所引用的目录不具搜索权限（也就是 execute 位未设定）。如果顶层目录可供读取但是不可供执行，便会发生此事。在此情况下，open() 会成功，但是 fchdir() 会失败。

EBADF
: fd 并不是一个已打开的文件描述符。

这两个调用的 errno 还有可能被设定为其他的错误值，这因文件系统而异。

这两个系统调用只会影响当前正在运行的进程。Unix 并未提供任何机制用于变更不同进程的当前工作目录。因此，shell 中可以找到的 *cd* 命令（如同多数命令那样）不会成为一个独立的进程，它只会对第一个命令行参数执行 chdir()，然后结束。事实上，*cd* 必须是一个特殊的内置命令，它会导致 shell 自己调用 chdir()，变更自己的当前工作目录。

getcwd() 的最常见用法就是保存当前工作目录，所以进程稍后可以返回该处。例如：

```
char *swd;
int ret;

/* 保存当前工作目录 */
swd = getcwd (NULL, 0);
if (!swd) {
        perror ("getcwd");
        exit (EXIT_FAILURE);
}

/* 变更为不同的目录 */
ret = chdir (some_other_dir);
if (ret) {
        perror ("chdir");
        exit (EXIT_FAILURE);
}

/* 在新目录中进行其他工作 …… */

/* 返回所保存的目录 */
ret = chdir (swd);
if (ret) {
        perror ("chdir");
        exit (EXIT_FAILURE);
}

free (swd);
```

然而，比较好的做法是先以 open() 打开当前目录，然后再对它使用 fchdir()。此做法速度比较快，因为内核不会把当前工作目录的路径名称存入内存，它只会存储 inode。因此，每当有用户调用 getcwd()时，内核就必须游走目录结构来产生路径名称。反过来说，打开当前工作目录的代价比较小，因为内核已经准备好它的 inode，所以不需要使用人类可阅读的路径名称来打开一个文件。下面便是使用此做法的范例程序代码：

```
int swd_fd;

swd_fd = open (".", O_RDONLY);
if (swd_fd == -1) {
        perror ("open");
        exit (EXIT_FAILURE);
}

/* 变更为不同的目录 */
ret = chdir (some_other_dir);
if (ret) {
        perror ("chdir");
        exit (EXIT_FAILURE);
}

/* 在新目录中进行其他工作……*/

/* 返回所保存的目录 */
ret = fchdir (swd_fd);
if (ret) {
        perror ("fchdir");
        exit (EXIT_FAILURE);
}

/* 关闭目录的 fd */
ret = close (swd_fd);
if (ret) {
        perror ("close");
        exit (EXIT_FAILURE);
}
```

这就是 shell 实现暂存先前目录的缓存机制的方法（例如，在 *bash* 中使用 *cd* - 命令）。

一个进程如果不在乎它的当前工作目录是什么 —— 例如一个守护进程（daemon），通常会调用 chdir("/") 把它设定成 “/”。一个连接用户与他的数据的应用程序，例如一个文字处理器，通常会将它的当前工作目录设定为用户的 home 目录或者是特定的文件目录。因为当前工作目录仅与相对路径名称的上下文有关，所以对“用户从 shell 所调用的命令行实用程序”而言，当前工作目录最为有用。

创建目录

Linux 提供了一个经 POSIX 标准化的系统调用，可用于创建新目录：

```
#include <sys/stat.h>
#include <sys/types.h>

int mkdir (const char *path, mode_t mode);
```

执行成功时，mkdir() 会创建权限位为 mode（经过当前 umask 的修改）的目录 path

(可能是相对的，也可能是绝对的)，而且会返回 0。

当前的 umask 会按照通常的方式修改 mode 参数，再加上操作系统特有的任何模式位：在 Linux 中，新建目录的权限位会被设定为 (mode & ~umask & 01777)。换言之，事实上，umask 对进程所强加的限制是 mkdir() 所不能忽视的。如果新目录的父目录的 set group ID (*sgid*) 位被设定过，或者文件系统的挂载使用了 BSD 的组语义，则新目录将从它的父目录继承此组关系。否则，会对新目录使用进程的有效组 ID。

执行失败时，mkdir() 会返回 -1 并且将 errno 设定为下面的其中一个值：

EACCESS
: 当前进程无法对父目录进行写入操作，或者无法对 path 中一个或多个元素进行搜索操作。

EEXIST
: path 已经存在（且不一定是一个目录）。

EFAULT
: path 是一个无效的指针。

ELOOP
: 解析 path 的时候，内核遇到了太多的符号链接。

ENAMETOOLONG
: path 太长。

ENOENT
: path 中有一个元素不存在，或者是一个虚悬（dangling）符号链接。

ENOMEM
: 内存的可用空间不足以完成此项请求。

ENOSPC
: 包含 path 的设备已无空间，或者用户的磁盘配额已逾越限度。

ENOTDIR
: path 中有一个或多个元素不是目录。

EPERM
: 包含 path 的文件系统并不支持创建目录。

EROFS
: 包含 path 的文件系统是以只读模式挂载的。

移除目录

mkdir() 的反向操作就是经过 POSIX 标准化的 rmdir()，可用于从文件系统层次结构移除一个目录：

```
#include <unistd.h>

int rmdir (const char *path);
```

执行成功时，rmdir() 会从文件系统移除 path，而且会返回 0。path 所指定的目录必须是空的，它只能包含“.”（dot）和“..”（dot-dot）目录。系统调用并未实现与 *rm -r* 等效的递归删除功能。此类工具必须以手动方式对文件系统进行深度优先搜寻，从叶节点开始移除所有文件和目录，并在文件系统中往上移动。rmdir() 可用于每个阶段：一旦目录中的文件被移除，就可以移除该目录。

执行失败时，rmdir() 会返回 -1 并将 errno 设定为下面的其中一个值：

EACCESS
: 不允许对 path 的父目录进行写入操作，或者 path 中有一个元素无法被搜索。

EBUSY
: 系统当前正在使用 path，无法移除 path。在 Linux 中，如果 path 是一个挂载点或是一个根目录（由于 chroot()，根目录不一定是挂载点！），才会发生此情况。

EFAULT
: path 是一个无效的指针。

EINVAL
: path 使用“.”（dot）目录作为它的最后一个元素。

ELOOP
: 解析 path 的时候，内核遇到了太多的符号链接。

ENAMETOOLONG
: path 太长。

ENOENT
: path 中有一个元素不存在或者是一个虚悬符号链接。

ENOMEM
: 内存的可用空间不足以完成此项请求。

ENOTDIR
: path 中有一个或多个元素不是目录。

ENOTEMPTY

path 中除了“.”（dot）和“..”（dot-dot）目录还包含其他项。

EPERM

path 的父目录设定了 sticky 位（S_ISVTX），但是进程的有效用户 ID 不是该父进程的用户 ID，也不是 path 本身的用户 ID，而且进程并不具备 CAP_FOWNER 能力。另外，包含 path 的文件系统不允许目录的移除操作。

EROFS

包含 path 的文件系统是以只读模式挂载的。

此调用的用法很简单：

```
int ret;

/* 移除目录 /home/barbary/maps */
ret = rmdir ("/home/barbary/maps");
if (ret)
        perror ("rmdir");
```

读取一个目录的内容

POSIX 定义了一系列的函数，可用于读取目录的内容——也就是特定目录中包含哪些文件。如果你正在实现 *ls* 或是一个图形化的文件保存对话框，如果你需要操作特定目录中的每一个文件，或者如果你想搜寻特定目录中与特定模式相匹配的文件，那么这些函数就会很有用。

着手读取一个目录的内容之前，你需要创建一个目录流（*directory stream*），此流会被表示成 DIR 对象：

```
#include <sys/types.h>
#include <dirent.h>

DIR * opendir (const char *name);
```

执行成功时，opendir() 会创建一个目录流来代表 name 所指定的目录。

目录流只是一个信息比较多的文件描述符，它代表已打开的目录、某些元数据以及保存目录内容的缓冲区。因此，你可以取得在特定目录流背后的文件描述符：

```
#define _BSD_SOURCE /* or _SVID_SOURCE */
#include <sys/types.h>
#include <dirent.h>

int dirfd (DIR *dir);
```

执行成功时，dirfd() 会返回目录流 dir 的文件描述符；发生错误时，此调用会返

回 -1。因为目录流函数会在内部使用这个文件描述符，所以程序应该只调用此类函数而不操纵文件位置。dirfd() 是 BSD 自行扩展的函数，并未经过 POSIX 的标准化，希望声明自己的程序符合 POSIX 标准的程序设计者应该避免使用此函数。

从一个目录流中读取数据

使用 opendir() 创建一个目录流后，你的程序可以开始从该目录读取数据项。要进行此项操作，你可以使用 readdir() 从特定的 DIR 对象一个接一个地返回其数据项。

```
#include <sys/types.h>
#include <dirent.h>

struct dirent * readdir (DIR *dir);
```

执行成功时，readdir() 会返回 dir 所代表的目录里下一个数据项。dirent 结构代表一个目录项。在 Linux 上，此结构定义于 <dirent.h>，下面是它的定义：

```
struct dirent {
        ino_t d_ino; /* inode number */
        off_t d_off; /* offset to the next dirent */
        unsigned short d_reclen; /* length of this record */
        unsigned char d_type; /* type of file */
        char d_name[256]; /* filename */
};
```

POSIX 只需要 d_name 字段，这是目录中单一文件的名称。其他字段则是可选项，或者是 Linux 特有的字段。想要移植到其他系统或符合 POSIX 标准的应用程序只应该访问 d_name。

应用程序可以调用 readdir()，取得目录中每个文件，直到它们找到所要搜索的文件或是直到读取整个目录，此时 readdir() 会返回 NULL。

执行失败时 readdir() 也会返回 NULL。为了区分错误以及读完所有文件，每次调用 readdir() 之前，应用程序必须将 errno 设定为 0，并于事后检查返回值和 errno。只有在 errno 的值被 readdir() 设定为 EBADF 时才代表 dir 是无效的。因此，许多应用程序并不会去检查错误，而且会假设 NULL 代表已经读完所有文件。

关闭目录流

要关闭 opendir() 所打开的一个目录流，可以使用 closedir()：

```
#include <sys/types.h>
#include <dirent.h>

int closedir (DIR *dir);
```

执行成功时，closedir() 会关闭 dir 所表示的目录流，包括背后的文件描述符，而且会返回 0。执行失败时，此函数会返回 -1 并将 errno 设定为 EBADF，这是唯一的错误代码，表示 dir 不是一个已打开的目录流。

下面的范例程序代码实现了一个函数 find_file_in_dir()，它会使用 readdir() 在特定目录中搜索特定文件名称。如果文件存在于目录中，则函数会返回 0；否则，会返回非零值：

```
/*
 * find_file_in_dir —— 搜索目录 path 以便
 * 找到名为 file 的文件。
 *
 * 如果 file 存在于 path 中，则返回 0；
 * 否则，返回非零值。
 */
int find_file_in_dir (const char *path, const char *file)
{
        struct dirent *entry;
        int ret = 1;
        DIR *dir;

        dir = opendir (path);

        errno = 0;
        while ((entry = readdir (dir)) != NULL) {
                if (!strcmp(entry->d_name, file)) {
                        ret = 0;
                        break;
                }
        }

        if (errno && !entry)
                perror ("readdir");

        closedir (dir);
        return ret;
}
```

用于读取目录内容的系统调用

前面所探讨的读取目录内容的函数都是经过 POSIX 标准化而且是 C 链接库都提供的。就内部而言，这些函数会使用 readdir() 与 getdents() 其中的一个系统调用，在此处列出它们只是为了完整性：

```
#include <unistd.h>
#include <linux/types.h>
#include <linux/dirent.h>
#include <linux/unistd.h>
```

```
#include <errno.h>

/*
 * Not defined for user space: need to
 * use the _syscall3() macro to access.
 */

int readdir (unsigned int fd,
             struct dirent *dirp,
             unsigned int count);

int getdents (unsigned int fd,
              struct dirent *dirp,
              unsigned int count);
```

你不会想使用这两个系统调用！因为不好使用，而且不具可移植性。事实上。用户空间应用程序应该使用 C 链接库的 `opendir()`、`readdir()` 以及 `closedir()` 系统调用。

链接

前面在讨论目录的时候提到过，每个名称至 inode 的映射（mapping）称为一个链接（*link*）。另一个简单的定义 —— 链接基本上就是一个列表（目录）中指向特定 inode 的一个名称 —— 说明了为什么会有多个链接指向相同的 inode。也就是说，单一 inode（因此是单一文件）可以被如 */etc/customs* 及 */var/run/ledger* 者所引用。

事实上，在此情况下会有一个问题：因为链接会映射至 inode，而 inode number 专属于特定的文件系统，所以 */etc/customs* 与 */var/run/ledger* 必须位于相同的文件系统。在单一文件系统中，可能会有大量的链接指向任何给定的文件。唯一的限制是用于保存链接数量的整数数据类型的大小。这些链接中，没有一个链接是“最初的”或“主要的”链接。所有这些链接均享有同样的状态：指向同一个文件。

我们称这种类型的链接为硬链接（*hard link*）。文件可能具有零个、一个或多个链接。多数文件都会具有值为 1 的链接计数—— 也就是会有一个目录项指向它们，但是有些文件具有两个甚至多个链接。文件若具有值为 0 的链接计数，表示它在文件系统上并没有相应的目录项。当一个文件的链接计数变为 0 时，该文件就会被标记为“释放”(free)，所以它的磁盘块可供再利用（注 4）。然而，此类文件仍可能存在于文件系统上，如果有任何进程已经打开该文件的话。一旦无任何进程打开该文件，该文件就会被移除。

注 4： 找出链接计数为 0 而且相应块被标记为已分配的文件是文件系统检查程序 *fsck* 的主要工作。如果一个文件被删除，但是仍维持打开的状态，而且在该文件被关闭之前系统崩溃了，便会发生此情况。内核未能将相应的文件系统块标记成释放（free），因此会出现不一致的地方。日志型文件系统可以消除此类型的错误。

Linux 内核是通过一个链接计数以及一个使用计数（*usage count*）来实现此行为的。使用计数就是文件已被打开的副本的总数。如果一个文件的链接及使用计数皆为 0，它就会从文件系统中被移除。

另一种链接类型称为符号链接（*symbolic link*）。尽管此类链接不具文件系统的映射，但是它具有一个在运行时解释的较高级指针。此类链接可以跨文件系统 —— 稍后就会看到。

硬链接

link() 系统调用原本是 Unix 的一个系统调用，现在是经过 POSIX 标准化的系统调用，可替现有的文件创建一个新链接：

```
#include <unistd.h>

int link (const char *oldpath, const char *newpath);
```

执行成功时，link() 会替现有文件 oldpath 以路径 newpath 创建一个新链接，然后返回 0。完成调用之后，oldpath 与 newpath 会指向同一个文件 —— 事实上，你无法区分哪一个是最初的链接。

执行失败时，此调用会返回 -1 并将 errno 设定为下面的其中一个值：

EACCESS
: 进行调用的进程对 oldpath 中的一个元素不具搜索权限，或者进行调用的进程对包含 newpath 的目录不具有写入权限。

EEXIST
: newpath 已经存在 —— link() 无法改写现有的目录项。

EFAULT
: oldpath 或 newpath 是一个无效的指针。

EIO
: 发生了一个内部的 I/O 错误（这是很严重的情况！）。

ELOOP
: 解析 oldpath 或 newpath 的时候遇到了太多的符号链接。

EMLINK
: oldpath 所指向的 inode 已经达到了可以指向它的链接数目上限。

ENAMETOOLONG
: oldpath 或 newpath 太长。

ENOENT

oldpath 或 newpath 中有一个元素不存在。

ENOMEM

内存的可用空间不足以完成此项请求。

ENOSPC

包含 newpath 的设备已无空间用于存放新的目录项。

ENOTDIR

oldpath 或 newpath 中有一个元素不是目录。

EPERM

包含 newpath 的文件系统不允许创建新的硬链接，或者 oldpath 是一个目录。

EROFS

newpath 位于只读文件系统上。

EXDEV

newpath 与 oldpath 位于不同的经挂载的文件系统上（Linux 允许单一文件系统于多个地点被挂载，不过硬链接是无法跨越挂载点创建的）。

下面的例子会创建一个新的目录项 *pirate*，它会指向现有文件 *privateer* 所指向的 inode（因此是同一个文件），它们都会被放在 */home/kidd* 之下：

```
int ret;

/*
 * 创建新的目录项
 * /home/kidd/pirate，它会指向
 * /home/kidd/privateer 所指向的 inode
 */
ret = link ("/home/kidd/privateer", /home/kidd/pirate");
if (ret)
        perror ("link");
```

符号链接

符号链接又称为软链接（*soft link*），同硬链接一样，它也会指向文件系统中的文件。它与硬链接的差别在于，符号链接不仅是一个额外的目录项，也是一个特殊类型的文件。这个特殊的文件中包含了另一个文件的路径名称，这称为符号链接的目标（target）。运行时，内核会以目标的路径名称来取代符号链接的路径名称（除非你使用的是名称前缀字母“l”的系统调用，例如 lstat()，在此情况下才会使用链接本身的路径名称）。因此，虽然你很难区分一个硬链接与指向相同文件的另一个硬链接，但是你却很容易区分一个符号链接与它的目标文件。

一个符号链接的路径可以是相对的或是绝对的，它也可以包含稍早所提到的“.”目录（代表它所处的目录）或是“..”目录（代表此目录的父目录）。

与硬链接不同的是，软链接可以跨越文件系统。事实上，它们可以指向任何地方！符号链接可以指向现有的文件（普遍的做法）或指向不存在的文件。后者的链接类型称为虚悬符号链接（*dangling symlink*）。有时，虚悬符号链接是不受欢迎的 —— 例如链接的目标被删除，但是符号链接并未被删除 —— 不过有时这是故意的。

用于创建符号链接的系统调用与它的硬链接表亲非常相似：

```
#include <unistd.h>

int symlink (const char *oldpath, const char *newpath);
```

执行成功时，symlink() 会创建一个指向目标 oldpath 的符号链接 newpath，然后会返回 0。

发生错误时，symlink() 会返回 -1 并将 errno 设定为下面的其中一个值：

EACCESS
: 进行调用的进程对 oldpath 中的一个元素不具搜索权限，或者进行调用的进程对包含 newpath 的目录不具有写入权限。

EEXIST
: newpath 已经存在 —— symink() 无法改写现有的目录项。

EFAULT
: oldpath 或 newpath 是一个无效的指针。

EIO
: 发生了一个内部的 I/O 错误（这是很严重的情况！）。

ELOOP
: 解析 oldpath 或 newpath 的时候遇到了太多的符号链接。

EMLINK
: oldpath 所指向的 inode 已经达到了可以指向它的链接数目上限。

ENAMETOOLONG
: oldpath 或 newpath 太长。

ENOENT
: oldpath 或 newpath 中有一个元素不存在。

ENOMEM

内存的可用空间不足以完成此项请求。

ENOSPC

包含 newpath 的设备已无空间可用于存放新的目录项。

ENOTDIR

oldpath 或 newpath 中有一个元素不是目录。

EPERM

包含 newpath 的文件系统不允许创建新的符号链接。

EROFS

newpath 位于只读文件系统上。

下面这段范例程序代码如同前面的例子，但是它会将 */home/kidd/pirate* 创建成指向 */home/kidd/privateer* 的符号链接（相对于硬链接而言）：

```
int ret;

/*
 * 创建一个符号链接
 * /home/kidd/pirate,
 * 它会指向 /home/kidd/privateer
 */
ret = symlink ("/home/kidd/privateer", "/home/kidd/pirate");
if (ret)
        perror ("symlink");
```

解除链接

创建链接（linking）的反向操作就是解除链接（unlinking），也就是从文件系统中移除其路径名称。unlink() 系统调用可以完成此工作：

```
#include <unistd.h>

int unlink (const char *pathname);
```

执行成功时，unlink() 会从文件系统中删除 pathname 并且返回 0。如果此名称是对文件的最后一个引用，则该文件会从文件系统中被删除。然而，如果有一个进程已经打开该文件，则内核不会从文件系统中删除该文件，除非进程关闭该文件。一旦没有任何进程打开该文件，该文件就会被删除。

如果 pathname 引用了一个符号链接，则链接本身（而不是目标）被破坏。

如果 pathname 引用了另一个类型的特殊文件，例如一个设备、FIFO 或 socket，则

特殊文件会从文件系统中被移除，但是已经打开该文件的进程可以继续使用它。

发生错误时，unlink() 会返回 -1 并将 errno 设定为下面的其中一个错误代码：

EACCESS
: 进行调用的进程对 pathname 的父进程不具有写入权限，或者进行调用的进程对 pathname 中的一个元素不具有搜索权限。

EFAULT
: pathname 是一个无效的指针。

EIO
: 发生了一个 I/O 错误（这是一个严重的情况！）。

EISDIR
: pathname 引用了一个目录。

ELOOP
: 解析 pathname 的时候遇到了太多的符号链接。

ENAMETOOLONG
: pathname 太长。

ENOENT
: pathname 中有一个元素不存在。

ENOMEM
: 内存的可用空间不足以完成此项请求。

ENOTDIR
: pathname 中有一个元素不是目录。

EPERM
: 系统不允许解除文件的链接。

EROFS
: pathname 位于只读文件系统上。

unlink() 无法移除目录。要完成此事，应用程序应该使用 rmdir()，参见稍早在“移除目录”一节的讨论。

为了避免肆意破坏任何类型的文件，C 语言提供了 remove() 函数：

```
#include <stdio.h>

int remove (const char *path);
```

执行成功时，remove() 会从文件系统中删除 path 并且会返回 0。如果 path 是一个文件，则 remove() 会调用 unlink()；如果 path 是一个目录，则 remove() 会调用 rmdir()。

发生错误时，remove() 会返回 -1 并将 errno 设定为适当的值，remove() 的有效错误代码如同 unlink() 以及 rmdir()的。

文件的复制以及移动

文件的复制（copying）以及移动（moving）是两个最基本的文件操作，一般是由 *cp* 以及 *mv* 命令来完成。就文件系统层而言，复制就是将特定文件的内容拷贝成一个新路径名称的动作。这不同于对创建文件的新硬链接，改变一个文件不会影响另一个文件，也就是说，现在该文件存在两个不同的副本，而且（至少）具有两个不同的目录项。反过来说，移动就是替一个文件的目录项进行更名的动作。此动作的结果不是创建第二个副本。

复制

有些人可能会感到惊讶，Unix 并未对文件和目录的复制提供任何的系统或链接库调用。事实上，可以手动使用实用程序，例如 *cp* 或 GNOME 的 Nautilus 文件管理程序，来完成这些工作。

将一个文件 src 复制成一个名为 dst 的文件，步骤如下：

1. 打开 src 。
2. 打开 dst，如果它尚不存在，则创建它；如果它已经存在，则将它截短成零长度。
3. 将 src 的一个块读进内存。
4. 将这个块写入 dst。
5. 继续下去，直到 src 的所有内容已经被读取以及写入 dst。
6. 关闭 dst。
7. 关闭 src。

如果复制一个目录，则会由 mkdir() 创建该目录以及任何子目录，然后单独复制其中的每个文件。

移动

与复制不同，Unix 为文件的移动提供了一个系统调用。ANSI C 标准替文件的移动引进了这个调用，而 POSIX 则针对文件与目录的移动将这个调用标准化：

```
#include <stdio.h>

int rename (const char *oldpath, const char *newpath);
```

执行成功时，rename() 会将路径名称 oldpath 更名为 newpath。文件的内容与 inode 则维持不变。oldpath 与 newpath 必须位于相同的文件系统（注 5）；否则，此调用将执行失败。*mv* 之类的实用程序必须通过复制以及解除链接来处理此情况。

执行成功时，rename() 会返回 0，文件从前被 oldpath 引用而现在被 newpath 引用。执行失败时，此调用会返回 -1，不会动到 oldpath 或 newpath，而且会把 errno 设定为下面的其中一个值：

EACCESS

进行调用的进程对 oldpath 或 newpath 的父目录不具有写入权限，对 oldpath 或 newpath 的一个元素不具有搜索权限，或者对 oldpath 具有写入权限但是 oldpath 是一个目录。最后一种状况是因为，如果 oldpath 是一个目录，则 rename() 必须更新 oldpath 中的“..”。

EBUSY

oldpath 或 newpath 是一个挂载点。

EFAULT

oldpath 或 newpath 是一个无效的指针。

EINVAL

newpath 被包含在 oldpath 中，因此把 oldpath 更名为 newpath 会使得 oldpath 变成自己的子目录。

EISDIR

newpath 存在而且是一个目录，但是 oldpath 不是一个目录。

ELOOP

解析 oldpath 或 newpath 的时候遇到了太多的符号链接。

注 5：尽管 Linux 允许你在目录结构的多个挂载点上挂载设备，但你还是无法将这些挂载点的其中一个更名为另一个，即使它们的背后是相同的设备。

EMLINK

oldpath 已经到达了本身的链接数目上限，或者 oldpath 是一个目录而且 newpath 已经到达了本身的链接数目上限。

ENAMETOOLONG

oldpath 或 newpath 太长。

ENOENT

oldpath 或 newpath 中有一个元素不存在，或者是一个虚悬符号链接。

ENOMEM

内核内存的可用空间不足以完成此项请求。

ENOSPC

设备上的可用空间不足以完成此项请求。

ENOTDIR

oldpath 或 newpath 中的一个元素（可能的最终元素除外）不是目录，或者 oldpath 是一个目录，而且 newpath 存在但不是一个目录。

ENOTEMPTY

newpath 是一个目录，但不是空的。

EPERM

参数中所指定的路径中至少有一个存在，父目录设定了 sticky 位，进行调用的进程的有效用户ID既不是文件的用户ID，也不是父进程的用户ID，以及进程不具有特权。

EROFS

文件系统被标记为只读。

EXDEV

oldpath 与 newpath 位于不同的文件系统上。

表 7-1 整理出了移动对各种类型的文件所造成的影响。

表 7-1：移动对各种类型的文件所造成的影响

	目的地是一个文件	目的地是一个目录	目的地是一个路径	目的地不存在
来源是一个文件	目的地会被来源改写	执行失败并返回 EISDIR	文件被更名而且目的地被改写	文件被更名
来源是一个目录	执行失败并返回 ENOTDIR	如果目的地是空的，则来源会被更名成目的地；否则，会执行失败并返回 ENOTEMPTY	目录会被更名，而且目的地会被改写	目录会被更名

表 7-1：移动对各种类型的文件所造成的影响（续表）

	目的地是一个文件	目的地是一个目录	目的地是一个路径	目的地不存在
来源是一个链接	链接会被更名而且目的地会被改写	执行失败并返回 EISDIR	链接会被更名而且目的地被改写	链接会被更名
来源不存在	执行失败并返回 ENOENT	执行失败并返回 ENOENT	执行失败并返回 ENOENT	执行失败并返回 ENOENT

如果来源与目的地位于不同的文件系统，则以上所有情况均与它们的类型无关，执行会失败并返回 EXDEV。

设备节点

设备节点（*device node*）是一种特殊文件，应用程序可以通过它连接设备驱动程序（device driver）。当一个应用程序在一个设备节点上执行平常的 Unix I/O（打开、关闭、读取、写入等）时，内核并不会像一般文件 I/O 那样来处理那些请求。事实上，内核会将此类请求传递给一个设备驱动程序。设备驱动程序会处理该 I/O 操作并将结果返回给用户。设备节点所提供的设备抽象层让应用程序不需要熟悉设备的具体情况，甚至不需要掌握特殊的接口。的确，在 Unix 系统上，设备节点是访问硬件的标准机制。网络设备是罕见的例外，而且在 Unix 的历史中，对此例外是否为错误有些争论。使用 read()、write() 以及 mmap() 等调用来操纵一个机器的所有硬件，的确具有一种优雅的美感。

内核如何辨别应该由哪个设备驱动程序来处理请求？每个设备节点会被赋予两个数值，称为主要编号（*major number*）以及次要编号（*minor number*）。这些主要和次要编号会映射至被加载进内核的特定设备驱动程序。如果一个设备节点所具有的主要和次要编号并未映射至内核内部的设备驱动程序 —— 会因为各种原因突然发生此情况，则设备节点上的一个 open() 请求会返回 -1 并且将 errno 设定为 ENODEV。我们说，此设备节点面对了不存在的设备。

特殊设备节点

特殊设备节点存在于所有 Linux 系统上。这些设备节点是 Linux 开发环境的一部分，而且它们的存在被视为 Linux ABI 的一部分。

null 设备，主要编号为 1，次要编号为 3，位于 */dev/null*。此设备文件的拥有者应该是 root 而且可供所有用户读取和写入。内核会默默丢弃针对该设备文件的所有写入请求。针对该设备文件的所有读取请求，内核会返回 end-of-file（EOF）。

zero 设备位于 */dev/zero*，它的主要编号为 1，次要编号为 5。如同 null 设备，内核会默默丢弃针对 zero 设备的所有写入请求。针对该设备的读取请求，内核会返回 null 字节的无穷流。

full 设备，主要编号为 1，次要编号为 7，位于 */dev/full*。如同 zero 设备，内核会针对该设备的读取请求返回 null 字符（`\0`）。然而，写入请求总是会触发 `ENOSPC` 错误，以便指出底层的设备已满。

这些设备的用途各有不同。它们可用于测试一个应用程序如何处理一些极限的问题 —— 例如一个被填满的文件系统。因为 null 和 zero 设备会忽略写入操作，所以它们也是一种不具有任何开销的丢弃多余 I/O 的方法。

随机数产生器

内核的随机数产生器设备位于 */dev/random* 以及 */dev/urandom*。它们的主要编号为 1，次要编号分别为 8 和 9。

内核的随机数产生器会从设备驱动程序以及其他源头收集 noise（杂乱的数据），而且内核会把收集到的 noise 连接在一起并对其进行单向散列运算。所得到的结果会被存入一个熵池（*entropy pool*）。内核会不断估算池中熵的位数目。

读取 */dev/random* 会从此池返回熵。这些结果可作为随机数产生器的种子，产生密钥，并且进行加密以强化熵的其他必要工作。

理论上，敌人可以从熵池获得足够的数据，而且成功破解单向散列就可以知道熵池其余的状态。然而此类攻击目前只是理论上可能 —— 并未听说发生过此类攻击，内核要降低这个可能性，可以对每个读取请求递减池中熵的估算数量。当估算数量为 `0` 时，读取请求将受到阻挡，除非系统产生更多的熵而且熵的估算数量足以完成读取请求。

/dev/urandom 不具此特性，即使内核的熵估算量不足以完成读取请求，读取此设备仍会成功。因为只有最讲究安全的应用程序 —— 例如在 GNU Privacy Guard 中为了数据交换的安全而产生密钥，才应该关心以加密强化熵的事情，多数应用程序应该使用 */dev/urandom* 而不使用 */dev/random*。如果没有 I/O 活动可以提供 noise 给内核的熵池，则读取请求可能会被阻挡非常久的时间。这通常会发生在 diskless（无磁盘驱动器）、headless（译注 1）服务器上。

译注 1：headless 一般是指没有屏幕、键盘或鼠标。

带外通信

Unix 的文件模型令人印象深刻。只需要进行简单的读取和写入操作，理论上 Unix 几乎就可以对一个对象进行任何可以想象得到的行为。然而，有些时候，程序设计者需要与主要数据流以外的文件通信。举例来说，考虑一个串行端口设备。读取该设备就等于读取位于串行端口的远端的硬件；写入数据到该设备就等于传送数据到该硬件。一个进程如何读取串行端口的状态脚位，例如数据终端就绪（data terminal ready，常简写为 DTR）信号？另外，一个进程如何完成串行端口的奇偶校验位（parity）设定？

答案就是使用 `ioctl()` 系统调用。`ioctl()`，代表 *I/O control*（输入输出控制），可进行带外通信（*out-of-band communication*）：

```
#include <sys/ioctl.h>

int ioctl (int fd, int request, ...);
```

此系统调用需要两个参数：

`fd`
: 一个文件的文件描述符。

`request`
: 一个特殊的请求代码值，经过内核与进程的预定义与同意，指出可以对 `fd` 所引用的文件采取何种行动。

它还可以接受一个或更多个可有可无的参数（通常是无符号的整数或指针），然后传递给内核。

下面的程序会使用 `CDROMEJECT` 请求弹出（用户在程序命令行的第一个参数所提供的）CD-ROM 设备的媒体托盘。因此，这个程序的功能如同标准的 *eject* 命令：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <linux/cdrom.h>
#include <stdio.h>

int main (int argc, char *argv[])
{
        int fd, ret;

        if (argc < 2) {
                fprintf (stderr,
                        "usage: %s <device to eject>\n",
```

```
                        argv[0]);
                return 1;
        }

        /*
         * 以只读模式打开 CD-ROM 设备。O_NONBLOCK 用于通知内核，
         * 即使设备中没有媒体，我们也要打开该设备。
         */
        fd = open (argv[1], O_RDONLY | O_NONBLOCK);
        if (fd < 0) {
                perror ("open");
                return 1;
        }

        /*给 CD-ROM 设备送出弹出命令。*/
        ret = ioctl (fd, CDROMEJECT, 0);
        if (ret) {
                perror ("ioctl");
                return 1;
        }

        ret = close (fd);
        if (ret) {
                perror ("close");
                return 1;
        }

        return 0;
}
```

CDROMEJECT 请求是 Linux 的 CD-ROM 设备驱动程序所提供的一个功能。当内核收到一个 ioctl() 请求时，它会找出负责处理所提供文件描述符的文件系统（对常规文件而言）或设备驱动程序（对设备节点而言），然后将请求传递给它们处理。就此例而言，CD-ROM 设备驱动程序会收到请求并且实际弹出光驱。

本章稍后我们将看到一个 ioctl() 的例子，这个例子会使用一个可选参数返回信息给发出请求的进程。

监视文件事件

Linux 提供了一个接口 *inotify*，用于监视文件——例如，监视文件何时被移动、读取、写入或删除。假设你正在编写一个图形化文件管理程序，例如 GNOME 的 Nautilus。当 Nautilus 正在显示一个目录的内容时，如果有一个文件被复制到该目录，则文件管理程序所看到的目录与实际的目录会出现不一致的现象。

一个解决方案就是不断重读目录的内容，检测变动以及更新所显示的内容。这会带来一个周期性的开销，所以是一个有缺陷的解决方案。更糟糕的是，一个文件从目录被移除

或被加入目录与文件管理程序重新读取目录的内容之间往往是一种竞争关系。

使用 inotify，内核可以在事件发生的瞬间通知应用程序。文件一被删除，内核就会通知 Nautilus。而 Nautilus 的响应就是立即从目录的图形化显示中移除被删除的文件。

多数其他的应用程序也会对文件事件有兴趣。现在考虑一个备份实用程序或者一个数据索引工具。inotify 让这两个程序可以进行实时的操作：在一个文件被创建、删除或写入的瞬间，这两个工具分别可以更新备份归档文件（backup archive）或数据索引（data index）。

inotify 用于取代 *dnotify* 这个采用笨重信号接口的较早期文件监视机制。相较于 dnotify，应用程序应该优先采用 inotify。inotify，引进自内核 2.6.13 版，不仅灵活而且容易使用，因为程序对常规文件所进行的操作 —— 尤其是 `select()` 与 `poll()` —— 可以跟 inotify 一起运作。本书只会提到 inotify。

初始化 inotify

在一个进程可以使用 inotify 之前，该进程必须对它进行初始化。`inotify_init()` 系统调用可用于初始化 inotify 并返回一个代表已初始化实例的文件描述符：

```
#include <inotify.h>

int inotify_init (void);
```

发生错误时，`inotify_init()` 会返回 -1，而且会将 `errno` 设定为下面的其中一个值：

`EMFILE`
: 已经到达了 inotify 实例数目上限的每用户限制（per-user limit）。

`ENFILE`
: 已经到达了文件描述符数目上限的全系统限制（system-wide limit）。

`ENOMEM`
: 内存的可用空间不足以完成此项请求。

让我们初始化 inotify，这样我们才可以在后续的步骤使用它：

```
int fd;

fd = inotify_init ();
if (fd == -1) {
        perror ("inotify_init");
        exit (EXIT_FAILURE);
}
```

监视项目

一个进程初始化 inotify 之后，接着会设置监视项目（*watches*）。一个监视项目会被表示成一个监视描述符（*watch descriptor*），它是一个标准的 Unix 路径以及一个相关联的监视掩码（*watch mask*），用于告知内核进程感兴趣的是哪些事件——例如，读取、写入或者这两者。

inotify 可以监视文件与目录。如果要监视一个目录，inotify 会报告该目录（以及该目录所包含的任何文件）上所发生的事件（但是不包含被监视目录的子目录里的文件——监视操作不具递归性）。

加入一个新的监视项目

系统调用 inotify_add_watch() 可以为 path（文件或目录的）上的 mask 所描述的事件加入一个监视项目至 fd 所表示的 inotify 实例：

```
#include <inotify.h>

int inotify_add_watch (int fd,
                       const char *path,
                       uint32_t mask);
```

执行成功时，inotify_add_watch() 会返回一个新的监视描述符；执行失败时，此调用会返回 -1，并且会将 errno 设定为下面的其中一个值：

EACCESS
: 不允许读取 path 所指定的文件。进行调用的进程必须能够读取该文件，这样才能对它加入一个监视项目。

EBADF
: 文件描述符 fd 不是一个有效的 inotify 实例。

EFAULT
: path 不是一个有效的指针。

EINVAL
: 监视掩码 mask 包含无效的事件。

ENOMEM
: 内存的可用空间不足以完成此项请求。

ENOSPC
: 已经到达了 inotify 实例数目上限的每用户限制。

监视掩码

监视掩码就是一个或多个 inotify 事件的二元 OR 运算，这些 inotify 事件定义于 `<inotify.h>`：

IN_ACCESS
: 读取自文件。

IN_MODIFY
: 写入至文件。

IN_ATTRIB
: 文件的元数据（例如，拥有者、使用权限或扩展属性）被改变。

IN_CLOSE_WRITE
: 文件被关闭，而且已经打开以备写入。

IN_CLOSE_NOWRITE
: 文件被关闭，而且并未打开以备写入。

IN_OPEN
: 文件被打开。

IN_MOVED_FROM
: 一个文件从所监视的目录中被移开。

IN_MOVED_TO
: 一个文件被移往所监视的目录。

IN_CREATE
: 一个文件被创建在所监视的目录中。

IN_DELETE
: 一个文件从所监视的目录中被删除。

IN_DELETE_SELF
: 所监视的对象本身被删除。

IN_MOVE_SELF
: 所监视的对象本身被移动。

还会定义下列事件，可将两个或多个事件聚集成单一值：

IN_ALL_EVENTS
: 所有合法的事件。

IN_CLOSE

与关闭相关（当前为 IN_CLOSE_WRITE 以及 IN_CLOSE_NOWRITE）的所有事件。

IN_MOVE

与移动相关（当前为 IN_MOVED_FROM 以及 IN_MOVED_TO）的所有事件。

现在，让我们来看如何对一个现有的 inotify 实例加入一个新的监视项目：

```
int wd;

wd = inotify_add_watch (fd, "/etc", IN_ACCESS | IN_MODIFY);
if (wd == -1) {
        perror ("inotify_add_watch");
        exit (EXIT_FAILURE);
}
```

此范例会为目录 */etc* 上的所有读取或写入事件加入一个监视项目。如果 */etc* 中有任何文件被写入或读取，inotify 会传送一个事件给 inotify 文件描述符 fd，提供给监视描述符 wd。让我们来看看 inotify 如何表示这些事件。

inotify 事件

inotify_event 结构定义于 <inotify.h>，可用于表示 inotify 事件：

```
#include <inotify.h>

struct inotify_event {
        int wd;           /* watch descriptor */
        uint32_t mask;    /* mask of events */
        uint32_t cookie;  /* unique cookie */
        uint32_t len;     /* size of 'name' field */
        char name[];      /* null-terminated name */
};
```

wd 用于指定监视描述符，取自 inotify_add_watch()，而 mask 代表事件。如果 wd 指定了一个目录，而且要监视该目录中一个文件上所发生的一个事件，则 name 所提供的文件名称会相对于该目录。在此情况下，len 是一个非零值。请注意，len 不同于 name 的字符串长度，name 的结尾可能不只一个 null 字符，其作用有如填充，确保后续的 inotify_event 能够被正确对齐。因此，你必须使用 len（而不要使用 strlen()）来计算一个数组里下一个 inotify_event 结构的偏移值。

举例来说，如果 wd 代表 */home/rlove* 并具有值为 IN_ACCESS 的 mask，而且读取自

文件 */home/rlove/canon*，于是 name 将等于 canon，而且 len 将至少为 6。因此，如果我们要以相同的掩码值直接监视 */home/rlove/canon*，则 len 的值将是 0，而且 name 将是零长度 —— 你不可以动它。

cookie 用于将两个相关但分开的事件连接在一起。我们稍后会加以讨论。

读取 inotify 事件

取得 inotify 事件很容易：你只需要从与 inotify 实例相关联的文件描述符读取。inotify 提供了一个功能，称为 *slurping*，允许你以单一读取请求读进多个事件 —— 直到提供给 read() 的缓冲区被填满。因为 name 字段的长度不定，所以这是读取 inotify 事件最常见的方式。

我们之前的例子创建了一个 inotify 实例，而且对该实例加入了一个监视项目。现在，让我们读取悬而未决的事件：

```
char buf[BUF_LEN]__attribute__((aligned(4)));
ssize_t len, i = 0;

/* 读取 BUF_LEN 个字节的事件 */
len = read (fd, buf, BUF_LEN);

/* 读取每个事件直到读完为止 */
while (i < len) {
        struct inotify_event *event =
                (struct inotify_event *) &buf[i];

        printf ("wd=%d mask=%d cookie=%d len=%d dir=%s\n",
                event->wd, event->mask,
                event->cookie, event->len,
                (event->mask & IN_ISDIR) ? "yes" : "no");

        /* 如果有一个名称，则输出它 */
        if (event->len)
                printf ("name=%s\n", event->name);

        /* 将索引更新为下一个事件的开头 */
        i += sizeof (struct inotify_event) + event->len;
}
```

因为 inotify 文件描述符的行为如同一个常规文件，程序可经 select()、poll() 以及 epoll() 来监视。这让进程可以从单一线程使用其他文件 I/O 对 inotify 事件进行多任务处理。

高级 inotify 事件

除了标准的事件，inotify 还可以产生其他事件：

IN_IGNORED
: wd 所表示的监视项目已经被移除。发生此事件可能是因为用户手动移除了监视项目，或是因为所监视的对象已经不存在。稍后我们将讨论此事件。

IN_ISDIR
: 受影响的对象是目录（如果未设定，受影响的对象是文件）。

IN_Q_OVERFLOW
: inotify 队列溢出。内核会限制事件队列的大小以避免内核内存被无止境地浪费。一旦未决事件的数目等于最大值减 1，内核就会产生此事件并把它添加到队列尾端。除非队列中的事件被读走，使得它的大小低于限制，否则不会再产生事件。

IN_UNMOUNT
: 支持所监视对象的设备被卸载。因此，该对象不再可用，内核将移除监视项目并产生 IN_IGNORED 事件。

任何监视项目都可以产生这些事件，用户并不需要显式设定它们。

程序设计者必须将 mask 视为未决事件的一个位掩码。因此，不要使用等效的直接测试来检查事件：

```
/* 不要这么做！ */

if (event->mask == IN_MODIFY)
        printf ("File was written to!\n");
else if (event->mask == IN_Q_OVERFLOW)
        printf ("Oops, queue overflowed!\n);
```

而要进行逐位测试：

```
if (event->mask & IN_ACCESS)
        printf ("The file was read from!\n");
if (event->mask & IN_UNMOUNTED)
        printf ("The file's backing device was unmounted!\n);
if (event->mask & IN_ISDIR)
        printf ("The file is a directory!\n");
```

链接移动事件

IN_MOVED_FROM 以及 IN_MOVED_TO 事件只代表整个移动操作的一半：前者代表从特定的位置移出，而后者代表来到一个新位置。因此，对于一个企图智能追踪文件的程序，当文件在文件系统上移动时（考虑一个不会对移动过的文件进行重新索引的索引程

序)，真正有用的做法是将这两个移动事件链接在一起。加入 inotify_event 结构中的 cookie 字段。

cookie 字段，如果是非零值，则包含了一个链接起两个事件的唯一值。考虑一个正在监视 */bin* 以及 */sbin* 的进程。假设 */bin* 有一个值为 7 的监视描述符，而 */sbin* 有一个值为 8 的监视描述符。如果文件 */bin/compass* 被移往 */sbin/compass*，则内核会产生两个 inotify 事件。

第一个事件的 wd 等于 7，mask 等于 IN_MOVED_FROM，而 name 等于 compass。第二个事件的 wd 等于 8，mask 等于 IN_MOVED_TO，而 name 等于 compass。在这两个事件中，cookie 将是相同的 —— 例如 12。

如果一个文件被更名，则内核仍会产生两个事件。这两个事件的 wd 是相同的。

请注意，如果一个文件从一个未被监视的目录移进或移出，则进程将不会收到相应事件的其中一个。由程序负责注意与 cookie 相匹配的第二个事件并未到达。

高级监视选项

创建一个新的监视项目时，你可以把下面的一个或多个值加入 mask 以便控制该监视项目的行为：

IN_DONT_FOLLOW

　　如果设定了此值，而且如果 path 的目标或者 path 中有任何元素是一个符号链接，则链接不会被跟随而且 inotify_add_watch() 会失败。

IN_MASK_ADD

　　通常，如果你对一个文件调用 inotify_add_watch()，则你在该文件上就会存在一个监视项目，监视掩码会被更新以反映刚才所提供的 mask。如果在 mask 中设定了此标志，则所提供的事件会被加入现有的掩码中。

IN_ONESHOT

　　如果设定了此值，则当特定的对象产生第一个事件之后，内核会自动移除此监视项目。此监视项目实际上“只做一次”(one shot)。

IN_ONLYDIR

　　如果设定了此值，则只有在提供的对象是一个目录时，才会加入监视项目。如果 path 代表一个文件而不是一个目录，则 inotify_add_watch() 会执行失败。

举例来说，下面这段范例程序代码，只有在 *init.d* 是一个目录而且 */etc* 和 */etc/init.d* 都不是符号链接时才会在 */etc/init.d* 上加入监视项目：

```
int wd;

/*
 * 只有在 /etc/init.d 是一个目录，而且它的路径中没有一个部分是符号链接时
 * 才会监视 /etc/init.d 是否移动过。
 */
wd = inotify_add_watch (fd,
                        "/etc/init.d",
                        IN_MOVE_SELF |
                        IN_ONLYDIR |
                        IN_DONT_FOLLOW);
if (wd == -1)
        perror ("inotify_add_watch");
```

移除一个 inotify 监视项目

如此处的例子所示，你可以使用系统调用 `inotify_rm_watch()` 从一个 inotify 实例中移除一个监视项目：

```
#include <inotify.h>

int inotify_rm_watch (int fd, uint32_t wd);
```

执行成功时，`inotify_rm_watch()` 会从 inotify 实例（以文件描述符表示）`fd` 移除以监视描述符 `wd` 表示的监视项目并返回 `0`。

例如：

```
int ret;

ret = inotify_rm_watch (fd, wd);
if (ret)
        perror ("inotify_rm_watch");
```

执行失败时，此系统调用会返回 `-1` 并且会将 `errno` 设定为下面的其中一个值：

`EBADF`
: `fd` 不是一个有效的 inotify 实例。

`EINVAL`
: 在所指定的 inotify 实例上 `wd` 不是一个有效的监视描述符。

移除一个监视项目时，内核会产生 `IN_IGNORED` 事件。内核不仅会在手动移除期间送出此事件，而且会因为另一种操作的副作用而销毁监视项目。举例来说，当所监视的一个文件被删除时，此文件上的任何监视项目均会被移除。以上两种情况下，内核都会送出 `IN_IGNORED`。此行为让应用程序可以固定在一处进行移除监视项目的处理：`IN_IGNORED` 的事件处理程序。这对 inotify 的高端消费者（需要管理每个 inotify 监

视项目背后复杂的数据结构，例如 GNOME 的 Beagle 搜索基础架构）而言非常有用。

取得事件队列的大小

未决事件队列（pending event queue）的大小可由 inotify 实例的文件描述符上的 FIONREAD ioctl 请求来取得。此请求的第一个参数用于接收队列的大小（以字节为单位），这是一个无符号整数。

```
integer:
unsigned int queue_len;
int ret;

ret = ioctl (fd, FIONREAD, &queue_len);
if (ret < 0)
        perror ("ioctl");
else
        printf ("%u bytes pending in queue\n", queue_len);
```

请注意，此请求所返回的是队列的大小（以字节为单位），而不是队列中事件的数目。一个程序可以根据 inotify_event 结构已知的大小（由 sizeof() 取得），从字节数目估算出事件数目，并且猜测出 name 字段的平均大小。然而，比较有用的是字节数目，因为这可让进程进行大小适当的读取操作。

FIONREAD 常量定义于头文件 <sys/ioctl.h> 中。

销毁一个 inotify 实例

销毁一个 inotify 实例以及任何相关联的监视项目就如关闭该实例的文件描述符一样简单：

```
int ret;

/* fd 经 inotify_init() 取得 */
ret = close (fd);
if (fd == -1)
        perror ("close");
```

当然，如同任何的文件描述符一样，内核会自动关闭文件描述符，并且在进程结束时清理资源。

第八章

内存管理

对一个进程而言内存是最基本而且是最重要的资源。本章将说明此资源的管理：内存的分配、操作以及最后的释放。

动词“*allocate*”（分配）——常用于指称取得内存——有些误导，因为它会让人有“需求超过供应而对不足的资源进行限额配给”的印象。用户当然都会喜欢内存越多越好。然而，在现代的系统上，真正的问题不是需要的太多但分配到的太少，而是如何适当地使用与记录所分配到内存。

本章中，你将学到在程序的各个区域分配内存的所有方法，包括每种方法的优缺点。我们还会看到如何设定和操作任何内存区域，以及如何锁定内存以让数据逗留在 RAM 中，使得程序的运行不必等待内核将数据所属的页面从交换空间（swap space）换入 RAM。

进程地址空间

Linux，如同任何现代的操作系统，会虚拟化它的物理内存。进程不会直接寻址物理内存。事实上，内核会将每个进程关联到唯一的虚拟地址空间。此地址空间是线性的，所使用的地址从零起算，而且递增至某个上限值。

页面与页面调度

虚拟地址空间由页面（*page*）组成。一个页面的大小是固定的，实际的大小取决于系统架构以及机器类型，典型的大小包括 4 KB（32-bit 系统）以及 8 KB（64-bit 系统）

（注 1）。页面不是有效的，就是无效的。一个有效页面（*valid page*）会被关联到物理内存（*physical memory*）中的一个页面或某个辅助存储器（*secondary storage*），例如一个交换分区（swap partition）或是磁盘上的一个文件。一个无效页面（*invalid page*）不会被关联到任何地方，而是会被表示成一段未使用、未分配的地址空间。访问这样的页面会导致分段违例（segmentation violation）。地址空间未必是连续的。尽管是线性寻址，然而其中可能包含许多不可寻址的空隙。

一个程序无法使用存在于辅助存储器（而非物理内存）中的一个页面，直到它被关联到物理内存中的一个页面。当一个进程试图访问此类页面上的一个地址时，内存管理单元（memory management unit，常简写为 MMU）会产生一个页面失误（page fault）异常事件。然后内核会介入，透明地（自动地）从辅助存储器将所需要的页面换入物理内存。因为虚拟内存可以比物理内存大很多（许多系统上甚至只有虚拟地址器！），内核也会经常从物理内存“换出页面”（*paging out*）至辅助储存器，以便让出空间给换入的页面。内核会试图换出不久的将来最不可能被用到页面，因此可以优化性能。

共享以及写入时才复制

虚拟内存中的许多页面即使位于不同进程所拥有的不同虚拟地址空间，可能也会映射至单一的物理页面。这让不同的虚拟地址空间得以共享物理内存中的数据。这个共享的数据可能仅供读取，或者可供读取和写入。

当一个进程写入一个被共享的可供写入的页面时，可能会发生两种情况。其中最简单的一种情况：内核允许进行写入操作，在此情况下，共享该页面的所有进程都可以看到写入操作的结果。允许多个进程读取或写入一个被共享的页面，通常需要经过一定程度的协调以及同步化。

然而，MMU 可能会拦截此写入操作，并且产生一个异常事件；内核的响应是自动替进行写入操作的进程创建一个新的副本，这让写入操作得以对新的页面副本继续进行。我们称此做法为写入时才复制（*copy-on-write*，常简写为 COW）（注 2）。事实上，允许多个进程对被共享的数据进行读取操作可节省内存空间。当一个进程想写入一个被共享的页面时，它马上会收到该页面独一无二的副本，这让内核的行动好像是进程具有它自己私有的副本。因为 COW 是逐页进行的，所以采用此做法，一个大型文件将可以有效地为许多进程所共享，而且个别的进程只会在它们写入那些页面时取得独一无二的物理页面。

注 1： 有些系统会支持一段区域的页面大小。因此，页面大小不是 ABI 的一部分。应用程序必须以设计程序的方式在运行时取得页面的大小。此议题已在第四章说明过了，本章将回顾此议题。

注 2： 第五章所提到的 `fork()` 就是采用 COW 的做法让子进程得以复制与共享父进程的地址空间。

内存区域

内核会把页面安排进块以分享某些特性，例如访问权限。这些块称为内存区域（*region*）、段（*segment*）或映射（*mapping*）。你可以在每个进程中找到特定类型的内存区域：

- 文本段（*text segment*）包含了一个进程的程序代码、字符直接量（string literals）、常变量（constant variable）以及其他仅供读取的数据。Linux 中，此段会被标记为仅供读取（read-only）而且直接映射自目标码文件（此程序可以是一个可执行文件，或是一个链接库）。
- 堆栈（*stack*）包含了进程的执行堆栈（execution stack），此堆栈会因为堆栈深度的增加和减少而动态变大和变小。执行堆栈中包含局部变量以及函数所返回的数据。
- 数据段（*data segment*）或堆（*heap*），包含了一个进程的动态内存。这是一个可供写入的段，所以此段的尺寸会变大或变小。这也是 `malloc()`（下一节会讨论）所返回的内存。
- *bss* 段（注 3）中包含了未初始化的全局变量。根据 C 标准的规定，这些变量中包含了特殊值（基本上全为零）。
- Linux 会通过两种方法来优化这些变量。首先，因为 bss 段只用于未初始化的数据，所以链接程序（*ld*）实际上不会在目标文件（object file）中存储这些特殊值。这样可以缩减二进制文件的大小。其次，当此段被加载至内存时，内核会以写入时才复制的做法将此段映射至一个填满零的页面，有效地将这些变量设定为它们的默认值。
- 多数地址空间包含了一些映射文件（*mapped file*），例如程序可以执行它自己、C 与其他经链接的链接库以及数据文件。文件 */proc/self/maps* 或程序 *pmap* 的输出可作为进程中有哪些映射文件的最佳范例。

本章将说明 Linux 所提供的用于取得与返回内存、创建与销毁新的映射以及其间一切的接口。

分配动态内存

内存还会以自动和静态变量的形式出现，但是任何内存管理系统的基础是动态内存的分配、使用以及最后的返回。动态内存分配于运行时（而不是在编译时），分配之前它的大小可能是未知的。需要用到动态内存的时机如下：你需要数量相当大的内存，或者这

注 3： 这个名字源自 *block started by symbol*（以符号开头的区块）。

些内存需要使用多久并不一定，在程序运行之前无法知道。例如，你可能会想将一个文件的内容或读取自键盘的输入数据存放在内存中。因为并不知道文件的大小，而且用户可能会键入任意多的数据，因此无法确定缓冲区的大小，所以你可能会因为所读取到的数据越来越多而动态调大缓冲区的大小。

C 变量本身并不会用到动态内存。例如，C 语言并未提供任何机制可以取得一个存在于动态内存的 `struct pirate_ship`。事实上，C 语言提供了一个机制可以分配足以保存一个 `pirate_ship` 结构的动态内存。然后，程序设计者可由一个指针（此例中就是 `struct pirate_ship *`）与它交互。

`malloc()` 就是一个用于取得动态内存的典型 C 接口：

```
#include <stdlib.h>

void * malloc (size_t size);
```

执行成功时，`malloc()` 会分配 `size` 个字节的内存，而且会返回一个指针指向刚才所分配内存区域的开头位置。此区域区域的内容并未定义，不要预期它会被填满零值。执行失败时，`malloc()` 会返回 `NULL` 并将 `errno` 设定为 `ENOMEM`。

`malloc()` 的使用相当简单，下面的范例程序可用于分配固定数量的内存：

```
char *p;

/* 给我 2 KB! */
p = malloc (2048);
if (!p)
        perror ("malloc");
```

下面的范例程序可用于分配一个结构：

```
struct treasure_map *map;

/*
 * 分配足够的内存来保存一个 treasure_map 结构
 * 并且以 map 指向它
 */
map = malloc (sizeof (struct treasure_map));
if (!map)
        perror ("malloc");
```

赋值时，C 会自动将指向 `void` 的指针提升为指向任何类型。因此，以上范例程序在赋值时并不需要将 `malloc()` 的返回值转型成左值（lvalue）的类型。然而，C++ 程序语言并不会对 `void` 进行自动提升的操作。因此，C++ 的用户需要以如下的方式对 `malloc()` 的返回值进行转型：

```
char *name;

/* 分配 512 个字节 */
name = (char *) malloc (512);
if (!name)
        perror ("malloc");
```

有些 C 程序设计者喜欢对会返回 void 指针的任何函数（包括 malloc()）的结果进行转型。我反对这种做法，如果函数的返回值曾变更为 void 指针以外的东西，这么做就会有一个错误隐藏在其中。此外，如果一个函数未做适当的声明（注 4），此类转型还会有一个缺陷隐藏在其中。前者不会危及 malloc() 的使用，不过后者会。

因为 malloc() 可能会返回 NULL，所以开发者务必检查并处理错误的情况。许多程序会定义并使用一个 malloc() 包裹函数（wrapper），以便在 malloc() 返回 NULL 时输出错误信息并终止程序。按惯例，开发者会调用 xmalloc() 这个共同的包裹函数：

```
/* 如同 malloc()，但是在失败时会终止程序 */
void * xmalloc (size_t size)
{
        void *p;

        p = malloc (size);
        if (!p) {
                perror ("xmalloc");
                exit (EXIT_FAILURE);
        }

        return p;
}
```

分配数组

当指定的 size 本身是动态的时，动态内存分配会变得相当复杂。数组的动态分配就是这种情况，数组中每个元素的大小是固定的，但是元素的数目则是动态分配的。为了简化此情况，C 提供了 calloc() 函数：

```
#include <stdlib.h>

void * calloc (size_t nr, size_t size);
```

执行成功时，calloc() 会返回一个指针，指向适合保存数组（包含 nr 个元素，每个元素都有 size 个字节）的内存块。因此，如下前后两次调用所请求的内存数量是

注 4：　未声明的函数默认会返回一个 int 值。整数至指针的转换（integer-to-pointer cast）并不会自动进行，而且会产生一个警告信息。手动进行类型转换可抑制警告信息的产生。

一样的（这两次调用都不会返回比所请求的数量还多的内存，但是也不会返回更少的内存）：

```
int *x, *y;

x = malloc (50 * sizeof (int));
if (!x) {
        perror ("malloc");
        return -1;
}

y = calloc (50, sizeof (int));
if (!y) {
        perror ("calloc");
        return -1;
}
```

然而，它们的行为却是不一样的。calloc() 会将返回的内存块中的所有字节全部填入零值，而 malloc() 并未对所分配内存的内容做如此的保证。因此，整数数组 y（共有 50 个元素）中每个元素的内容均为 0，而 x 中每个元素的内容均未定义。除非程序准备立即设定这 50 个值，否则程序设计者应该使用 calloc() 来确保数组元素不会被填入任意值。请注意，二进制的零不同于浮点数的零！

用户通常会想将动态内存填满零值，即使在不处理数组的时候。本章稍后会探讨 memset()，此接口可用于将一个内存块中的每个字节设定成一个特定值。然而，让 calloc() 填满零值会比较快，因为内核可以提供已经填满零值的内存。

执行失败时，如同 malloc()，calloc() 会返回 NULL 并将 errno 设定为 ENOMEM。

为什么标准组织不把 malloc() 定义成“分配并填满零值”的函数是一个谜。然而，开发者轻易就能定义自己的接口：

```
/* 如同 malloc()，但是会将内存区域填满零值 */
void * malloc0 (size_t size)
{
        return calloc (1, size);
}
```

因此，我们可以将这个 malloc0() 与前面的 xmalloc() 合并：

```
/* 如同 malloc()，但是会将内存区域填满零值，而且会在失败时终止程序 */
void * xmalloc0 (size_t size)
{
        void *p;

        p = calloc (1, size);
        if (!p) {
```

```
                perror ("xmalloc0");
                exit (EXIT_FAILURE);
        }

        return p;
}
```

调整内存分配的大小

C 语言提供了一个接口，用于调整现有内存分配的大小（变大或变小）：

```
#include <stdlib.h>

void * realloc (void *ptr, size_t size);
```

执行成功时，realloc() 会将 ptr 所指向的内存区域的大小调整为 size 个字节的新大小。它会返回一个指针，指向刚调整过大小的内存区域，这个指针可能跟 ptr 一样，也可能不一样 —— 扩大一个内存区域时，如果 realloc() 无法在原处通过扩张块的方式扩大现有的内存块，则此函数会分配一个新的内存区域（长度为 size 个字节），然后将旧区域的内容复制到新区域并释放旧的区域。不管是哪种情况，内存区域的内容被保存下来的部分取决于旧大小与新大小中最小者。因为可能会进行复制操作，所以使用 realloc() 扩大一个内存区域可能是一个代价相当高的行动。

如果 size 的值为 0，则结果如同针对 ptr 调用 free()。

如果 ptr 的值为 NULL，则此操作的结果如同 malloc()；如果 ptr 的值非 NULL，则它必须是之前所进行的 malloc()、calloc() 或 realloc() 调用的返回值。

执行失败时，realloc() 会返回 NULL 并将 errno 设定为 ENOMEM，而且 ptr 所指向的内存的状态不会改变。

让我们来看一个缩小内存区域的例子。首先，我们会使用 calloc() 分配足够的内存区域来存放具有两个元素的 map 结构数组：

```
struct map *p;

/* 替两个 map 结构分配内存  */
p = calloc (2, sizeof (struct map));
if (!p) {
        perror ("calloc");
        return -1;
}

/* 使用 p[0] 与 p[1]…… */
```

现在，假设我们已经找到其中一份宝藏，不再需要第二张地图，所以我们决定调整内存的大小，以便将区域的一半还回系统（一般而言这么做并不值得，但是如果 map 结构非常大，而且我们打算长时间保存地图，就值得这么做）：

```
struct map *r;

/* 现在我们只需要足以保存一张地图的内存区域 */
r = realloc (p, sizeof (struct map));
if (!r) {
        /* 注意，p 仍然有效！ */
        perror ("realloc");
        return -1;
}

/* 使用 r …… */

free (r);
```

此例中，realloc() 完成调用之后p[0] 会被保留下来。之前的数据仍然存在于该处。如果调用失败，则 p 维持不变，因此仍然有效。我们可以继续使用它，而且最后需要释放它。反过来说，如果调用成功，则我们可以忽略 p，并转而使用 r（可能跟 p 一样，因为新的内存区域几乎是就地调整大小）。现在，当我们完成工作之后，我们有责任释放 r。

释放动态内存

不同于自动分配（stack unwind（译注 1）时会自动解除分配），动态分配的内存是进程地址空间中永久保存的部分，除非它们被手动释放。因此，程序设计者必须承担将动态分配的内存释放回系统的责任（当然，如果整个进程结束运行，则静态和动态分配的内存区域都会不见）。

使用 malloc()、calloc() 或 realloc() 所分配的内存在不再使用时必须由 free() 释放回系统：

```
#include <stdlib.h>

void free (void *ptr);
```

free() 可用于释放 ptr 所指向的内存。参数 ptr 的值必须是先前所调用的 malloc()、calloc() 或 realloc() 的返回值。也就是说，你无法通过将指针指向所分配的块

译注 1：stack unwind 通常是指从被调用函数返回时，由堆栈顶端弹出相应堆栈帧（内含局部变量）。此外，自动内存分配通常与自动变量有关。自动变量就是程序代码块里的局部变量。执行流程进入程序代码块时就会在堆栈上自动分配这些局部变量，离开程序代码块就会自动解除这些局部变量的分配，所以自动变量通常称为局部变量。

的某个部分的位置，让 free() 释放部分内存块 —— 例如，某个内存块的一半。

ptr 的值可以是 NULL，在此情况下 free() 会悄悄返回。因此，调用 free() 之前检查 ptr 是否为 NULL 值的常见做法其实是多余的。

让我们来看下面这个例子：

```
void print_chars (int n, char c)
{
        int i;

        for (i = 0; i < n; i++) {
                char *s;
                int j;

                /*
                 * 分配并且将一个有 i+2 个元素的字符数组填满零值。
                 * 注意，sizeof (char) 的结果总是为 1。
                 */
                s = calloc (i + 2, 1);
                if (!s) {
                        perror ("calloc");
                        break;
                }

                for (j = 0; j < i + 1; j++)
                        s[j] = c;

                printf ("%s\n", s);

                /* 做好了。交回给内存。 */
                free (s);
        }
}
```

此范例所分配的 n 个字符（chars）数组中包含了递增的元素数目，区域从 2 个元素（2 个字节）到 n + 1 个元素（n + 1 个字节）。对每个数组而言，循环会把字符 c 写入它的每个元素，最后一个元素除外（让最后一个字节保持零值），接着将数组当成一个字符串输出，然后释放动态分配的内存。

调用 print_chars() 时，若 n 等于 5 且 c 设定为 X，则会获得如下的结果：

```
X
XX
XXX
XXXX
XXXXX
```

当然，实现此函数还有更有效率的做法。然而，关键是我们可以动态分配以及释放内存，即使在运行时才获知分配的大小和数目。

> Unix 系统，例如 SunOS 与 SCO，为 free() 提供了一个变体，名为 cfree()，依系统而定，其行为可能跟 free() 一样，或者接受三个参数，仿照 calloc()。Linux 中，free() 可以处理取自（迄今我们所提过的）任何分配机制的内存。除非顾及向下兼容，否则不应该使用。Linux 版本的 cfree() 跟 free() 一样。

如果此范例并未调用 free()，注意会有什么结果。程序不会把内存释放回系统，而且更糟的是，程序可能会失去它对内存的唯一引用 —— 指针 s，这让它再也无法访问内存。我们将此类程序设计错误称为内存泄漏（*memory leak*）。内存泄漏类似于常见的动态内存错误，而且不幸地，这在 C 程序设计中是最严重的错误。因为 C 语言将管理内存的所有责任全都加诸在程序设计者身上，所以 C 程序设计者必须严谨看待所有内存分配。

另一个 C 程序设计常犯的错误就是“*use-after-free*”。一个内存块被释放后，随后又被访问便会发生此错误。对一个内存块调用 free() 之后，程序不得再访问它的内容。程序设计者必须特别留意虚悬指针（*dangling pointer*），或是指向无效内存块的非 NULL 指针。有两个常用的工具可协助你进行此工作：*Electric Fence* 与 *valgrind*（注 5）。

对齐

数据的对齐（*alignment*）是指从硬件来看数据的地址与内存块之间的关系。一个变量所位于的内存地址若是其大小的倍数，则称为自然对齐（*naturally aligned*）。举例来说，如果一个大小为 32 位（4 个字节）的变量在内存中的地址是 4 的倍数—— 也就是说，如果地址中最低的 2 位的值为 0，则称为自然对齐 。因此，一个数据类型的大小若为 2^n 个字节，则它的地址中最低的 n 位必须设定为 0。

与对齐有关的规则源自硬件。有些机器架构在数据的对齐上有非常严格的要求。在有些系统上，若加载未对齐的数据会触发陷阱（trap）异常事件。而在其他系统上，则可以安全地访问未对齐的数据，但是会造成性能降低的后果。要编写具可移植性的程序代码，必须避免对齐问题，而且所有数据类型应该被自然对齐。

注 5： 分别参见 *http://perens.com/FreeSoftware/ElectricFence/* 以及 *http://valgrind.org*。

分配已对齐的内存

多数情况下，编译器和 C 链接库会透明地（自动地）处理对齐的问题。POSIX 规定由 malloc()、calloc() 以及 realloc() 所返回的内存必须适当地对齐所使用的任何标准 C 类型。就 Linux 而言，这些函数所返回的内存在 32 位系统上，总是会对齐 8 字节边界；在 64 位系统上，则会对齐 16 字节边界。

有时，程序设计者需要动态内存对齐较大的边界，例如一个页面。尽管动机不同，不过最常见的就是需要适当对齐直接块 I/O 或其他的软件／硬件通信中所使用的缓冲区。因此，POSIX 1003.1d 提供了一个名为 posix_memalign() 的函数：

```
/* 只有一个定义会符合需要 */
#define _XOPEN_SOURCE 600
#define _GNU_SOURCE

#include <stdlib.h>

int posix_memalign (void **memptr,
                    size_t alignment,
                    size_t size);
```

执行成功时，posix_memalign() 会分配 size 个字节的动态内存，并确保它的地址对齐 alignment 的倍数。参数 alignment 必须是 2 的幂次方，而且是一个 void 指针大小的倍数。所分配内存的地址会被放入 memptr，而且此调用会返回 0。

执行失败时不会分配内存，也不会定义 memptr，而且此调用会返回下面的其中一个错误代码：

EINVAL

参数 alignment 并非 2 的幂次方或者并非是一个 void 指针大小的倍数。

ENOMEM

内存的可用空间不足以完成所请求的分配动作。

注意，此处并不会设定 errno —— 函数会直接返回这些错误代码。

经 posix_memalign() 取得的内存可通过 free() 释放。posix_memalign() 的用法很简单：

```
char *buf;
int ret;

/* 分配 1 KB 对齐 256 byte 边界 */
ret = posix_memalign (&buf, 256, 1024);
if (ret) {
        fprintf (stderr, "posix_memalign: %s\n",
                 strerror (ret));
```

```
        return -1;
}

/* 使用 buf…… */

free (buf);
```

较旧的接口

POSIX 定义 posix_memalign() 之前，BSD 与 SunOS 分别提供了如下的接口：

```
#include <malloc.h>

void * valloc (size_t size);
void * memalign (size_t boundary, size_t size);
```

除了所分配的内存会对齐一个页面的边界，函数 valloc() 的其他操作如同 malloc()。第四章所提到的系统的页面大小可由 getpagesize() 轻易取得。

函数 memalign() 也一样，不过它会将所分配的内存对齐 boundary 个字节（必须是 2 的幂次方）的边界。下面的例子中，这两种分配方式都会返回足以存放一个 ship 结构的内存块（对齐一个页面的边界）：

```
struct ship *pirate, *hms;

pirate = valloc (sizeof (struct ship));
if (!pirate) {
        perror ("valloc");
        return -1;
}

hms = memalign (getpagesize (), sizeof (struct ship));
if (!hms) {
        perror ("memalign");
        free (pirate);
        return -1;
}

/* 使用 pirate 与 hms ……*/

free (hms);
free (pirate);
```

在 Linux 上，经这两个函数所取得的内存可通过 free() 释放。在其他的 Unix 系统上情况可能不同，有些系统并未提供任何机制用于安全地释放这两个函数所分配的内存。考虑可移植性的程序没有别的选择，只能不释放由这两个接口所分配的内存！

Linux 程序设计者只应该在有移植到较旧系统的需要时才使用这两个函数，应该优先使用 posix_memalign()。除非所要求的对齐边界大于 malloc() 所能提供的，否则并不需要使用这三种接口。

其他的对齐议题

此处所要探讨的对齐议题已经超出了标准数据类型以及动态内存分配的自然对齐。例如，相较于标准的数据类型，非标准以及复杂的数据类型具有更复杂的需求。再者，当要对不同类型的指针赋值以及进行类型转换时，此类对齐议题变得加倍重要。

非标准的数据类型

非标准与复杂的数据类型的对齐需求超出了自然对齐的简单要求。下面是四个有用的规则：

- 一个结构的对齐需求就是该结构的组成类型（constituent type）中最大者。举例来说，如果一个结构成员中最大的数据类型是一个 32 位的整数，因此需要对齐 4 字节的边界，则此结构必须至少对齐 4 字节的边界。
- 结构还会有进行填充的需要，用于确保每个组成类型被适当对齐该类型自己的需求。因此，如果发现一个 `char`（需要对齐 1 个字节）后面跟着一个 `int`（需要对齐 4 个字节），编译器将在这两个类型之间插入 3 个填充字节，确保 `int` 位于 4 字节的边界上。程序设计者有时会调整结构中成员的次序 —— 例如按照尺寸由大至小 —— 以降低填充所浪费的空间。GCC 选项 `-Wpadded` 会努力协助此事，每当编译器插入隐式的填充时，它会产生一个警告信息。
- 一个联合类型（union）的对齐需求就是成员类型中最大者。
- 一个数组的对齐需求就是基本类型（base type）。因此，数组的对齐需求不会越出其类型的单一实例。此行为是数组中所有成员自然对齐的结果。

操弄指针

因为编译器会透明地处理多数的对齐需求，它会花一点工夫揭露可能的问题。然而，它并未考虑到处理指针与转型时所遇到的对齐议题。

通过指针转型，将对齐较小块的数据转换成对齐较大块的数据来访问数据时，会导致处理器加载的数据并未适当对齐较大类型的结果。举例来说，下面的程序代码片段会将 `c` 赋值给 `badnews`，试图将 `c` 读取成 `unsigned long`：

```
char greeting[] = "Ahoy Matey";
char *c = greeting[1];
unsigned long badnews = *(unsigned long *) c;
```

一个 `unsigned long` 可能会对齐 4 或 8 字节的边界，而在相同的边界上 `c` 几乎只会用到 1 个字节。因此，类型转换时，C 的加载会导致对齐违例（alignment violation）。根据架构，这样所导致的结果可能小到性能下降，也可能大到程序崩溃。某些机器架构可以检测出对齐违例，但无法适当地予以处理，内核会送出 `SIGBUS` 信号给违例的进

程、以便终止该进程。我们将在第九章探讨信号的议题。

这类例子可能比你所想象的还常见。真实世界中将不会出现如此胡涂的情况，但也可能只是不那么明显罢了。

管理数据段

Unix 系统以往就提供了可用于直接管理数据段的接口。然而，多数程序并未直接使用这些接口，因为 malloc() 以及其他分配机制更容易使用而且功能强大。我将在此处说明这些接口，以满足好奇的读者及想自己实现基于堆的分配机制的少数读者：

```
#include <unistd.h>

int brk (void *end);
void * sbrk (intptr_t increment);
```

这些函数的名称源自老旧的 Unix 系统，其中堆（heap）与堆栈（stack）位于相同的段。堆中的动态内存分配从段的底部向上增长；而堆栈则会从段顶部向下朝堆方向增长。堆与堆栈之间的分界线称为打断（*break*）或断点（*break point*）。现代计算机上，数据段位于它自己的内存映射中，我们会继续将映射的结尾地址标示为断点。

brk() 可用于将断点（数据段的结尾）设定为 end 所指定的地址。执行成功时，它会返回 0；执行失败时，它会返回 -1 并将 errno 设定为 ENOMEM。

sbrk() 会根据 increment 的值（可能是正或负的增量）来增加数据段的结尾地址。sbrk() 会返回修改过的断点。因此，若 increment 的值为 0，则会输出当前的断点：

```
printf ("The current break point is %p\n", sbrk (0));
```

事实上，POSIX 与 C 标准都未定义这两种函数。然而，几乎所有 Unix 系统都支持这两种函数。具可移植性的程序应该使用基于标准的接口。

匿名内存映射

glibc 中的内存分配使用的是数据段（data segment）以及内存映射（memory mapping）。实现 malloc() 的典型做法就是将数据段分割成一系列“2 的幂”的分区，而且可以通过返回最接近所要求大小的分区来满足内存分配需求。要释放内存，只需要将相应的分区标记为“可用”（free）就行了。如果比邻的分区也被标记为可用，则它们会被合并成一个较大的分区。如果堆的顶部是整体可用的，则系统会使用 brk() 调低断点，缩

小堆并将内存返回内核。

此算法称为伙伴内存分配方案（*buddy memory allocation scheme*）。此机制的优点是速度快并且容易实现，但缺点是会导致两种碎片（*fragmentation*）。如果使用比所请求大小还多的内存来满足一个分配需求，就会发生内部碎片（*internal fragmentation*）的情况。这会导致可用的内存变成无法使用的。如果可用的内存可以满足所请求的大小，但是却被分隔成两个或多个非比邻的块，就会发生外部碎片（*external fragmentation*）的情况。这会导致内存无法使用（因为也许使用了一个较大、较不适当的块）或内存分配失败（如果不存在可供选择的块）。

再者，此机制会让一个内存分配“钉住”（pin）另一个，以避免 *glibc* 将内存释放回内核。假设有两个内存块——块 *A* 和块 *B*—— 被分配。块 *A* 就在断点上，而块 *B* 位于块 *A* 之下。即使程序释放块 *B*，*glibc* 仍旧无法调整断点，除非块 *A* 也被释放。这样，一个长期运行的应用程序就会“钉住”内存中所有其他的分配。

这往往不是重点所在，因为 *glibc* 并不会按惯例将内存释放回系统（注 6）。通常，堆并不会在每次内存被释放后就被缩减。事实上，*glibc* 会保留被释放的内存以供后续的分配需求使用。只有在堆的尺寸显著大于内存分配所需的数量时，*glibc* 才会缩减数据段。然而，大型的内存分配不会出现缩减的现象。

因此，对于大型的内存分配，*glibc* 并不会使用堆。事实上，*glibc* 会创建一个匿名内存映射（*anonymous memory mapping*）以满足内存的分配请求。匿名内存分配类似于第四章所探讨的基于文件的映射，不过其背后并未有任何文件的支持 —— 因而有“匿名”之称。事实上，匿名内存映射只是一个可供你使用的大型、填满零值的内存块。你可以把它想成一个全新的堆，不过仅用于单一内存分配。因为此类映射位于堆之外，所以它们并不会造成数据段的碎片问题。

通过匿名映射分配内存有以下好处：

- 与碎片无关。当程序不再需要一个匿名内存映射时，映射会被解除，而且内存会被立即释放回系统。
- 匿名内存映射的大小是可调整的，具有可调整的访问权限，而且可以像一般映射（参见第四章）那样收到警告信息。
- 每个内存分配存在于独立的内存映射中。不需要管理全局性的堆。

注 6：*glibc* 会使用比这个简单的伙伴分配方案还要高级的内存分配算法，称为 *arena algorithm*。

与堆相比，匿名内存映射有两项缺点：

- 每个内存映射的大小是系统页面尺寸的整数倍。因此，如果一个内存分配的大小不是页面尺寸的整数倍，则会造成 slack space（译注 2）的浪费。slack space 与小型的内存分配更有关，相较于内存分配的大小，这会浪费比较多的空间。
- 相较于从堆将内存释放回系统（并未涉及任何内核的交互），创建一个新的内存映射会导致更多的开销。内存分配的规模越小，此现象越显著。

为了让优点有发挥的空间，同时避开缺点，*glibc* 的 `malloc()` 会使用数据段来满足小型的分配需求，以及使用匿名内存映射来满足大型的分配需求。分配需求是大型或小型取决于一个可设定的阈值（参见“高级内存分配”一节），而且不同的 *glibc* 版本可能会有所差异。当前的阈值为 128 KB：内存分配小于或等于 128 KB 者来自堆，而大于 128 KB 的内存分配则来自匿名内存映射。

创建匿名内存映射

或许是因为你想为特定的分配需求使用内存映射（而非堆），或许是因为你要编写自己的内存分配系统，你可能会想手动创建你自己的匿名内存映射 —— 无论是哪种情况，对 Linux 而言都很容易。第四章所提到的系统调用 `mmap()` 与 `munmap()` 可分别用于创建与销毁一个内存映射：

```
#include <sys/mman.h>

void * mmap (void *start,
             size_t length,
             int prot,
             int flags,
             int fd,
             off_t offset);

int munmap (void *start, size_t length);
```

事实上，创建一个匿名内存映射比创建一个基于文件的映射还要容易，因为没有文件需要开启以及管理。这两种映射的主要差别在于，会有一个特别的标志用于表示映射为匿名的。

让我们来看一个例子：

译注 2： slack space 是指内部碎片所浪费的存储空间。

```
void *p;

p = mmap (NULL,                        /* 此处无关紧要 */
          512 * 1024,                  /* 512 KB */
          PROT_READ | PROT_WRITE,      /* 可供读取与写入 */
          MAP_ANONYMOUS | MAP_PRIVATE, /* 匿名与私有 */
          -1,                          /* fd(忽略) */
          0);                          /* offset(忽略) */

if (p == MAP_FAILED)
        perror ("mmap");
else
        /* p 指向 512 KB 的匿名内存……*/
```

一般而言，使用 mmap() 创建匿名映射时，所传递的参数值会像这个例子所呈现的那样。当然，length 的参数值（以字节为单位）取决于程序设计者的需要。其他的参数值一般会做如下的设定：

- 第一个参数 start 会被设定成 NULL，这意味着匿名映射的起始地址让内核自行安排。此处也可以指定非 NULL 值，只要它能够对齐页面，但是这会对可移植性造成限制。鲜有程序会在乎映射存在于内存中的何处！
- 为了让映射可供读取和写入，prot 参数通常会设定 PROT_READ 与 PROT_WRITE 这两个位。一个空的映射是没有任何用处的（如果无法对它进行读取和写入的操作）。另一方面，很少会有从一个匿名映射执行程序代码的需要，而且允许这么做会导致一个潜在的安全漏洞。
- flags 参数会设定 MAP_ANONYMOUS 位(此映射为匿名的)以及 MAP_PRIVATE 位（此映射为私有的）。
- 当 MAP_ANONYMOUS 被设定时，参数 fd 与 offset 会被忽略。然而，有些较旧的系统会希望 fd 的值为 -1，如果考虑到可移植性，应该将 -1 传入 fd 参数。

通过匿名映射所取得的内存看起来会像是经堆所取得的内存。从匿名映射分配内存的好处是页面已经填满零值。完成此事不需要任何代价，因为内核会通过写入时复制（copy-on-write）将应用程序的匿名页面映射至填满零值的页面。因此，不需要对所返回的内存使用 memset()。的确，相较于使用 malloc() 之后再使用 memset()，直接使用 calloc() 会更好：*glibc* 知道匿名映射已经被填满零值，而且 calloc() 会对一个映射不需要显式填入零值感到满意。

系统调用 munmap() 会释放一个匿名映射，并且将已分配的内存释放回内核：

```
int ret;

/* 释放 p 之后，会给回 512 KB 的映射 */
ret = munmap (p, 512 * 1024);
if (ret)
        perror ("munmap");
```

mmap()、munmap() 以及映射的相关细节可参考第四章。

映射 /dev/zero

其他的 Unix 系统，例如 BSD，并不具有 MAP_ANONYMOUS 标志。事实上，它们是通过映射一个特殊设备文件*/dev/zero*来实现类似的解决方案。此设备文件为匿名内存提供了一致的语义。一个映射包含了填满零值的写入时才复制（copy-on-write）页面，因此映射 */dev/zero* 的行为如同使用匿名内存。

Linux 上总是可以找到 */dev/zero* 设备，而且提供了映射此文件与取得填满零值的内存的能力。的确，引进 MAP_ANONYMOUS 之前，Linux 程序设计者就是采用此做法。为了向下兼容于较旧版的 Linux，或是移植到其他 Unix 系统，开发者仍然可以映射 */dev/zero* 以便取代创建一个匿名映射的做法。这与映射任何其他文件并无不同。

```
void *p;
int fd;

/* 打开 /dev/zero 以备读取与写入 */
fd = open ("/dev/zero", O_RDWR);
if (fd < 0) {
        perror ("open");
        return -1;
}

/* /dev/zero 的 [0,page size) 映射  */
p = mmap (NULL,                   /* 此处无关紧要 */
          getpagesize (),         /* 映射一个页面 */
          PROT_READ | PROT_WRITE, /* 可供读取与写入 */
          MAP_PRIVATE,            /* 私有映射 */
          fd,                     /* 映射 /dev/zero */
          0);                     /* 无偏移量 */

if (p == MAP_FAILED) {
        perror ("mmap");
        if (close (fd))
                perror ("close");
        return -1;
```

```
}

/* 关闭 /dev/zero，不再需要 */
if (close (fd))
        perror ("close");

/* p 指向内存的一个页面，予以使用 ……*/
```

以此方式映射内存时，当然要以 munmap() 解除映射。

此做法涉及打开与关闭设备文件的额外系统调用开销。因此，匿名内存是一个速度比较快的解决方案。

高级内存分配

在本章所探讨的分配操作中，有许多受到了内核参数的制约，程序设计者可加以变更。要完成此事，可以使用 mallopt()：

```
#include <malloc.h>

int mallopt (int param, int value);
```

mallopt() 可用于将 param 所指定的与内存管理有关的参数设定为 value 所指定的值。执行成功时，此调用会返回一个非零值；执行失败时，它会返回 0 。注意，mallopt() 不会设定 errno。它总是会成功返回，所以应该避免任何过度乐观的想法：可以从返回值取得有用的信息。

Linux 当前为 param 提供了六种值（定义于 <malloc.h>）：

M_CHECK_ACTION

MALLOC_CHECK_ 环境变量的值（下一节会讨论到）。

M_MMAP_MAX

系统可用于满足动态内存的请求的映射的数目上限。当达到此限度时，数据段将被用于所有分配，直到这些映射中有一个被释放。值为 0 表示禁止将匿名映射用于动态内存分配。

M_MMAP_THRESHOLD

大小阈值（以字节为单位），分配请求超过此值时，将由一个匿名映射（而不是数据段）获得满足。注意，分配请求小于此阈值时，在系统的判断之下，也可能由匿名映射获得满足。值为 0 代表将匿名映射用于所有分配，实际上就是禁止将数据段用于动态内存分配。

M_MXFAST

fast bin（快速二进制文件）的大小上限。*fast bin* 是堆中的特殊块，它们绝对不会与比邻的块结合，也绝对不会被返回给系统，可用于进行非常快速的分配，代价是碎片会增加。此值为 0 时代表禁用所有的 fast bin。

M_TOP_PAD

填充的数目（以字节为单位），用于调整数据段的大小。每当 *glibc* 使用 brk() 增加数据段的大小时，它可以请求比需要的更多的内存，以避免没多久又需要进行另外一次 brk() 调用。同样地，每当 *glibc* 缩减数据段的大小时，它会保留额外的内存，因此实际给回的内存比它应该给回的少。这些额外的字节就是填充（*padding*）。值为 0 时代表禁止使用所有填充。

M_TRIM_THRESHOLD

数据段顶部所允许的可用内存(以字节为单位)的最小数量。如果超出此阈值，*glibc* 会调用 brk() 以便将内存给回内核。

XPG 标准所定义的 mallopt() 多了三个参数：M_GRAIN、M_KEEP 以及 M_NLBLKS。尽管 Linux 也定义了这些参数，但是设定它们的值毫无作用。表 8-1 完整地列出了每个有效的参数、它们的默认值以及它们的有效值。

表 8-1：mallopt() 参数

参数	起源	默认值	有效值	特殊值
M_CHECK_ACTION	Linux 特有	0	0-2	
M_GRAIN	XPG 标准	Linux 不支持	>=0	
M_KEEP	XPG 标准	Linux 不支持	>=0	
M_MMAP_MAX	Linux 特有	64 * 1024	>=0	0 禁用 mmap()
M_MMAP_THRESHOLD	Linux 特有	128 * 1024	>=0	0 禁用堆
M_MXFAST	XPG 标准	64	0-80	0 禁用 fast bin
M_NLBLKS	XPG 标准	Linux 不支持	>=0	
M_TOP_PAD	Linux 特有	0	>=0	0 禁用填充

程序在首次调用 malloc() 或任何其他内存分配接口之前，必须先对 mallopt() 进行任何的调用。它的用法很简单：

```
int ret;

/* 将 mmap() 用在超过 64 KB 的所有分配需求 */
ret = mallopt (M_MMAP_THRESHOLD, 64 * 1024);
if (!ret)
        fprintf (stderr, "mallopt failed!\n");
```

使用 malloc_usable_size() 与 malloc_trim() 进行微调

Linux 提供了一对函数，可以对 *glibc* 的内存分配系统进行低端的控制。第一个函数让一个程序可以询问特定内存分配中包含多少可用字节：

```
#include <malloc.h>

size_t malloc_usable_size (void *ptr);
```

执行成功时，malloc_usable_size() 会返回 ptr 所指向的内存块的实际分配大小。因为 *glibc* 会将分配填补成适合放进一个现有的块或匿名映射，所以一个分配中的可用空间可能会大于所请求的空间。当然，一个分配中的可用空间绝不会小于所请求的空间。下面是此函数的使用范例：

```
size_t len = 21;
size_t size;
char *buf;

buf = malloc (len);
if (!buf) {
        perror ("malloc");
        return -1;
}

size = malloc_usable_size (buf);

/* 我们可以实际使用 size 个字节的 buf ……*/
```

第二个函数让一个程序得以要求 *glibc* 将所有立即可用的内存释放回内核：

```
#include <malloc.h>

int malloc_trim (size_t padding);
```

执行成功时，malloc_trim() 会尽可能缩减数据段（扣除所保留的填充字节）。然后，它会返回 1 。执行失败时，此调用会返回 0 。通常，每当可用内存到达 M_TRIM_THRESHOLD 个字节，*glibc* 就会自动进行这样的缩减动作。它会使用 M_TOP_PAD 的填充数目。

除了调试或教学的目的，你几乎不会将这两个函数用在其他地方。它们不具可移植性，而且可让你的程序取得 *glibc* 内存分配系统的低端细节。

调试内存分配

通过设定环境变量 MALLOC_CHECK_，一个程序可以在内存子系统中启用调试功能。额外的调试检查会降低内存分配的效率，但是在应用程序开发期间的调试阶段这些开销是值得的。

因为可以通过一个环境变量来控制调试，所以不需要重新编译你的程序。例如，你只需要执行如下的命令：

```
$ MALLOC_CHECK_=1 ./rudder
```

如果 MALLOC_CHECK_ 被设定为 0，则内存子系统会默默地忽略任何错误；如果它被设定为 1，则会将提供信息的消息输出至 stderr；如果它被设定为 2，则程序会由 abort() 立即终止。因为 MALLOC_CHECK_ 会变更运行中程序的行为，故 setuid 程序会忽略此变量。

取得统计数据

Linux 所提供的 mallinfo() 函数可用于取得关于内存分配系统的统计数据：

```
#include <malloc.h>

struct mallinfo mallinfo (void);
```

mallinfo() 会返回 mallinfo 结构中的统计数据。此结构的返回是通过值，而不是通过一个指针。此结构的内容也定义于 <malloc.h>：

```
/* all sizes in bytes */
struct mallinfo {
        int arena;    /* size of data segment used by malloc */
        int ordblks;  /* number of free chunks */
        int smblks;   /* number of fast bins */
        int hblks;    /* number of anonymous mappings */
        int hblkhd;   /* size of anonymous mappings */
        int usmblks;  /* maximum total allocated size */
        int fsmblks;  /* size of available fast bins */
        int uordblks; /* size of total allocated space */
        int fordblks; /* size of available chunks */
        int keepcost; /* size of trimmable space */
};
```

它的用法很简单：

```
struct mallinfo m;

m = mallinfo ();

printf ("free chunks: %d\n", m.ordblks);
```

Linux 还提供了 malloc_stats() 函数，此函数会将关于内存的统计数据输出至 stderr：

```
#include <malloc.h>

void malloc_stats (void);
```

在一个内存密集型（memory-intensive）程序中调用 malloc_stats() 会产生一些大数字：

```
Arena 0:
system bytes     =  865939456
in use bytes     =  851988200
Total (incl. mmap):
system bytes     = 3216519168
in use bytes     = 3202567912
max mmap regions =      65536
max mmap bytes   = 2350579712
```

基于堆栈的分配

目前我们已探讨过的动态内存分配的所有机制都是使用堆或者内存映射来取得动态内存的。我们预期也应该是这样，因为堆与内存映射本质上就是动态决定的。堆栈是程序的地址空间中另一个常见的数据结构，用于存放程序的自动变量。

然而，没有理由说一个程序设计者无法为动态内存分配使用堆栈。只要分配不会造成堆栈溢出，此类做法应该很容易，而且应该可以执行得相当好。要从堆栈进行动态内存分配，可以使用 alloca() 系统调用：

```
#include <alloca.h>

void * alloca (size_t size);
```

执行成功时，alloca() 会返回一个指针，指向 size 个字节的内存（空间）。此内存（空间）位于堆栈上，而且进行调用的函数返回时会自动被释放。有些实现会在执行失败时返回 NULL，但是多数 alloca() 实现不会失败或者不能汇报失败的情况。失败的原因只可能是堆栈溢出。

用法如同 malloc()，然而你不需要（的确不必）释放所分配的内存。下面的例子是一个函数，此函数会在系统的分配目录（可能是 */etc*，不过如果要移植到其他系统的话最好在编译时决定）中打开所指定的文件。此函数会分配一个新的缓冲区，将系统分配目录复制到缓冲区，然后将缓冲区与所提供的文件名衔接在一起：

```
int open_sysconf (const char *file, int flags, int mode)
{
        const char *etc = SYSCONF_DIR; /* "/etc/" */
        char *name;

        name = alloca (strlen (etc) + strlen (file) + 1);
        strcpy (name, etc);
        strcat (name, file);

        return open (name, flags, mode);
}
```

返回时，因为会进行 stack unwind 操作，所以 `alloca()` 所分配的内存会被自动释放。这意味着，一旦从调用 `alloca()` 的函数返回，你就无法使用此内存！然而，因为你不必调用 `free()` 来进行任何清理的动作，所以程序代码的编写较为简洁。下面使用 `malloc()` 实现相同的函数：

```
int open_sysconf (const char *file, int flags, int mode)
{
        const char *etc = SYSCONF_DIR; /* "/etc/" */
        char *name;
        int fd;

        name = malloc (strlen (etc) + strlen (file) + 1);
        if (!name) {
                perror ("malloc");
                return -1;
        }

        strcpy (name, etc);
        strcat (name, file);
        fd = open (name, flags, mode);
        free (name);

        return fd;
}
```

注意，你不应该在一个函数调用的参数中直接使用 `alloca()` 分配内存，因为这样所分配的内存会存在于保留给函数参数使用的堆栈空间中。例如，你不应该这么做：

```
/* 不要这么做！ */
ret = foo (x, alloca (10));
```

`alloca()` 接口有一段坎坷的历史。在许多系统上，它的表现不佳，甚至会出现未定义的行为。如果系统的堆栈尺寸不够大而且大小固定，使用 `alloca()` 很容易造成堆栈溢出，而且会导致程序终止运行。有些系统上甚至不提供 `alloca()`。渐渐地，这些有缺陷以及行为不一致的实现坏了 `alloca()` 的名声。

所以，如果你的程序必须考虑可移植性，你应该避免使用 alloca()。然而，Linux 上的 alloca() 是一个非常有用却未被充分利用的工具。它的表现非常好 —— 在许多架构上，经 alloca() 分配内存的话只需要增加堆栈指针的值即可 —— 而且方便性优于 malloc()。在 Linux 专用的程序代码中分配少量的内存时，alloca() 可以有绝佳的性能表现。

将字符串复制到堆栈

alloca() 常用于临时复制一个字符串。例如：

```
/* 我们想要复制 "song" */
char *dup;

dup = alloca (strlen (song) + 1);
strcpy (dup, song);

/* 操作 dup ……*/

return; /* dup 会被自动释放 */
```

因为时常会有这种需要，而且 alloca() 有速度快的优点，所以 Linux 系统为 strdup() 提供了若干变体，用于将所指定的字符串复制到堆栈：

```
#define _GNU_SOURCE
#include <string.h>

char * strdupa (const char *s);
char * strndupa (const char *s, size_t n);
```

调用 strdupa() 会返回 s 的一个副本。调用 strndupa() 会复制 s 中的 n 个字符。如果 s 的字符数目大于 n，则会在第 n 个字符停止复制，而且函数会在最后添加一个 null 字节。这两个函数的优点如同 alloca()。当进行调用的函数返回时，所复制的字符串就会自动被释放。

POSIX 并未定义 alloca()、strdupa() 和 strndupa() 等函数，而且它们在其他操作系统上有不良的记录。如果考虑到可移植性，不建议使用这些函数。然而，在 Linux 上，alloca() 等函数的表现不俗，而且可以提供绝佳的性能，因为仅需调整堆栈帧指针，而不必进行复杂的动态内存分配。

变长数组

C99 引进了变长数组（variable-length array，常简写为 VLA），此数组的长度在运行时（而非编译时）设定。GNU C 对变长数组的支持已经有一段时间了，但后来 C99 才

将它们标准化，这为它们的使用提供了更大的诱因。如同 alloca()，VLA 免除了动态内存分配的开销。

它们的用法正如你所预期的那样：

```
for (i = 0; i < n; ++i) {
        char foo[i + 1];

        /* 使用 foo ……*/
}
```

在这个程序代码片段中，foo 是一个长度为 i+1 的 char 数组（也就是说，这个数组是由 i+1 个类型为 char 的变量所构成）。在每次的循环中，foo 会被动态创建，并且在离开作用域时被自动清除。如果我们以 alloca() 来取代一个 VLA，则内存不会被释放，除非函数返回。使用 VLA 可确保内存在每次的循环中被释放。因此，使用一个 VLA 至多会耗用 n 个字节，然而使用 alloca() 则会耗用 n*(n+1)/2 个字节。

我们可以使用一个变长数组将 open_sysconf() 改写成下面这样：

```
int open_sysconf (const char *file, int flags, int mode)
{
        const char *etc = SYSCONF_DIR; /* "/etc/" */
        char name[strlen (etc) + strlen (file) + 1];

        strcpy (name, etc);
        strcat (name, file);

        return open (name, flags, mode);
}
```

alloca() 与变长数组的主要差别在于，由前者所取得的内存会一直保留到函数返回，而经后者所取得的内存会一直保留到所保存的变量离开作用域（这可以发生在当前函数返回之前）为止。在我们刚才所看到的 for 循环中，于每次循环中回收内存可以降低内存的耗用量，而不会有任何副作用（我们不需要让额外的内存悬置在一旁）。然而，如果基于某个原因，我们想保留内存的时间超出一次循环迭代，那么此时使用 alloca() 会更有意义。

在单一函数中混用 alloca() 和变长数组会招来奇怪的行为。所以最安全的做法就是在单一函数里只使用其中一种机制。

选择内存分配机制

本章提到了各式各样的内存分配选择，可能会使得程序设计者无法确定哪种是适合自己的解决方案。多数情况下，malloc() 是你的最佳选择。然而，有时另一种做法才是更好的选择。表 8-2 整理出了选择分配机制的指导方针。

表 8-2：Linux 中内存的分配机制

分配机制	优点	缺点
malloc()	容易、简单、通用	所返回的内存未必填满零值
calloc()	简化数组的分配，所返回的内存填满零值	如果不是分配数组，接口过于复杂
realloc()	调整现有分配的大小	只能用于调整现有分配的大小
brk()与sbrk()	可对堆（heap）进行内部的控制	对多数用户而言太过低水平
匿名内存映射	容易使用、可共享、允许开发者调整保护等级以及提供警告信息；是大型映射的优选项	是小型分配的次优选项；为优选项时，malloc() 会自动使用匿名内存映射
posix_memalign()	分配内存并对齐任何适当的边界	相对较新，因此可移植性令人质疑；必须密切关注对齐问题，否则会有很大的杀伤力
memalign() 与 valloc()	相较于 posix_memalign()，较常用在其他的 Unix 系统上	并非 POSIX 标准，所提供的对齐控制不如 posix_memalign()
alloca()	分配速度非常快，不需要显式释放内存；非常适合小型的分配	不会返回错误，不适合大型的分配，在某些 Unix 系统上会出问题
变长数组	如同 alloca()，但是会在数组离开作用域（而不是在函数返回）时释放内存	仅用于数组；在某些情况下，alloca()释放内存的行为可能更胜一筹；在其他的 Unix 系统上不如 alloca() 常用

最后，我们不要忘了以上这些选项的替代方案：自动与静态内存分配。在堆栈上分配自动变量，或者在堆上分配全局变量通常更容易，而且不需要程序设计者管理指针或者担心内存释放的问题。

操作内存

C 语言提供了一系列函数用于操作内存的原始字节。这些函数的运作在许多方面均类似于字符串操作接口，例如 strcmp() 与 strcpy()，但是它们所依靠的是用户提供的缓冲区大小而不是以 null 结尾的字符串。注意，这些函数都不会返回错误信息。避免错误是程序设计者的责任 —— 传入错误的内存区域，没有别的选择，只会导致分段违例的结果！

设定字节

这些内存操作函数中最常用的当然是 memset()：

```
#include <string.h>

void * memset (void *s, int c, size_t n);
```

调用 memset()，可从 s 开始将 n 个字节设定成字节 c 并返回 s 。此函数常用于将一个内存块填满零值：

```
/* 将 [s,s+256] 填满零值 */
memset (s, '\0' , 256);
```

bzero() 是一个较旧的即将废弃的接口，由 BSD 所引进以用于执行相同的工作。新的程序代码应该使用 memset()，但是考虑到向下兼容以及对其他系统的可移植性，Linux 仍旧提供 bzero()：

```
#include <strings.h>

void bzero (void *s, size_t n);
```

下面的例子如同前面的 memset() 范例：

```
bzero (s, 256);
```

注意，bzero() —— 以及其他名称以 b 开头的接口 —— 需要用到的头文件是 <strings.h> 而不是 <string.h>。

> 如果你可以使用 calloc()，你就不应该使用 memset()！避免先使用 malloc() 分配内存，接着使用 memset() 将它填满零值。尽管结果可能一样，但将两个函数调用(malloc() 与 memset())换成一个函数调用(calloc() 会返回填满零值的内存)会有更好的性能。这样你不但可以少进行一次函数调用，而且 calloc() 还可以从内核那里取得填满零值的内存。在此情况下，你不必自己动手将每个字节设定为 0，而且可以提升性能。

比较字节

类似于 strcmp()，memcmp() 可用于比较两个内存块是否相等：

```
#include <string.h>

int memcmp (const void *s1, const void *s2, size_t n);
```

此函数会比较 s1 与 s2 的前 n 个字节，如果这两个内存块相等则返回 0；如果 s1 小于 s2 则返回小于零的值；如果 s1 大于 s2 则返回大于零的值。

同样，BSD 也提供了一个现在即将废弃的接口，用来执行大致相同的工作：

```
#include <strings.h>

int bcmp (const void *s1, const void *s2, size_t n);
```

bcmp() 会比较 s1 与 s2 的前 n 个字节，如果这两个内存块相等则返回 0；如果它们有所不同则返回非零值。

因为结构会被填充（参见“其他的对齐议题”一节），所以经 memcmp() 或 bcmp() 比较两个结构是否相等并不可靠。一个结构的两个实例会因为填充中包含未初始化的垃圾而导致不相等。因此，如下的程序代码并不安全：

```
/* 两艘船是否相同? (不安全) */
int compare_dinghies (struct dinghy *a, struct dinghy *b)
{
        return memcmp (a, b, sizeof (struct dinghy));
}
```

事实上，想比较两个结构的程序设计者应该比较两个结构中的每个成员。此类做法可以经过优化，但是所花的工夫绝对会比不安全的 memcmp() 做法还多。下面是等效的程序代码：

```
/* 两艘船是否相同? */
int compare_dinghies (struct dinghy *a, struct dinghy *b)
{
        int ret;

        if (a->nr_oars < b->nr_oars)
                return -1;
        if (a->nr_oars > b->nr_oars)
                return 1;

        ret = strcmp (a->boat_name, b->boat_name);
        if (ret)
                return ret;

        /* 等等，比较每个成员……*/
}
```

移动字节

memmove() 会将 src 的前 n 个字节复制到 dst 并返回 dst：

```
#include <string.h>

void * memmove (void *dst, const void *src, size_t n);
```

此外，BSD 也提供了一个即将废弃的接口，用以执行相同的工作：

```
#include <strings.h>

void bcopy (const void *src, void *dst, size_t n);
```

注意，尽管这两个函数所需的是相同的参数，但是 bcopy() 中前两个参数的次序是相反的。

bcopy() 与 memmove() 可以安全地处理重叠的内存区域（例如，如果 dst 的一部分位于 src 的区域内）。例如，这让内存中的字节得以在指定的区域内上移或下移。因为此情况并不常见，所以遇到这种情况时程序设计者将会知道，C 标准所定义的 memmove() 变体并不支持重叠的内存区域。这个变体的执行速度可能更快：

```
#include <string.h>

void * memcpy (void *dst, const void *src, size_t n);
```

此函数的行为如同 memmove()，但是 dst 和 src 不可以重叠。如果重叠，结果会变成未定义的。

memccpy() 是另一个安全的复制函数：

```
#include <string.h>

void * memccpy (void *dst, const void *src, int c, size_t n);
```

memccpy() 函数的行为如同 memcpy()，但是如果此函数在 src 的前 n 个字节中发现字节 c 则会停止复制。此函数会返回一个指针，指向 dst 中 c（如果找不到 c 则是 NULL）之后的下一个字节。

最后，你可以使用 mempcpy()，将数据复制到连续的内存位置：

```
#define _GNU_SOURCE
#include <string.h>

void * mempcpy (void *dst, const void *src, size_t n);
```

mempcpy() 函数的执行如同 memcpy()，但是它会返回一个指针，指向所复制的最后一个字节之后的下一个字节。如果你有一组数据要复制到连续的内存位置，这会很有

用——但是并没多大的不同，因为只不过是返回值成为 dst+n。这是 GNU 特有的函数。

搜索字节

函数 memchr() 与 memrchr() 可用于找出特定字节在一个内存块中的位置：

```
#include <string.h>

void * memchr (const void *s, int c, size_t n);
```

memchr() 函数会在 s 所指向的内存的 n 个字节中搜索字符 c（会被解释成一个 unsigned char）：

```
#define _GNU_SOURCE
#include <string.h>

void * memrchr (const void *s, int c, size_t n);
```

此调用会返回一个指针，指向第一个与 c 相匹配的字节，如果找不到 c 则会指向 NULL。

memrchr() 函数如同 memchr() 函数，但是它会从 s 所指向的 n 个字节的尾端往回搜索（而不是从首端往前搜索）。与 memchr() 不同的是，memrchr() 是 GNU 的一个扩充，而且不是 C 语言的一部分。

至于更复杂的搜索工作，可以使用 memmem() 函数在一个内存块中搜索任意的字节数组：

```
#define _GNU_SOURCE
#include <string.h>

void * memmem (const void *haystack,
               size_t haystacklen,
               const void *needle,
               size_t needlelen);
```

memmem() 函数会返回一个指针，指向长度为 haystacklen 个字节的内存块 haystack 中第一次出现长度为 needlelen 个字节的子块 needle。如果此函数没有在 haystack 中找到 needle，则会返回 NULL。此函数也是 GNU 的一个扩充。

旋转字节

Linux C 链接库提供了一个接口，可用于对数据的字节进行简单的旋转（convolute）：

```
#define _GNU_SOURCE
#include <string.h>

void * memfrob (void *s, size_t n);
```

memfrob() 会隐藏 s 开始的内存的前 n 个字节，方法是使用数值 42 对每个字节进行 exclusive-OR（异或，常简写为 XOR）逻辑运算。此调用会返回 s。

调用 memfrob() 的结果可以被反转回来，方法是对相同的内存区域再调用一次 memfrob()。因此，如下的程序代码片段等于没有对 secret 进行任何操作 。

```
memfrob (memfrob (secret, len), len);
```

此函数无法正确替代加密（或者仅是一个差劲的替代品），它的用途仅限于普通的字符串隐藏，是 GNU 特有的函数。

锁定内存

Linux 实现了请求页面调度（*demand paging*），这意味着在有需要时，页面会从磁盘换入（swap in）内存；不再需要时，页面会被换出（swap out）至磁盘。这让系统上进程的虚拟地址空间与物理内存的总数之间没有直接的关系，因为磁盘上的交换空间可以呈现物理内存几乎无限供应的假象。

页面的交换会自动发生，而且应用程序一般不需要关心（甚至是知道）Linux 内核的页面调度行为。然而，在两种情况下，应用程序会想影响系统的页面调度行为：

确定性

有时间限制的应用程序需要有确定性的行为。如果某些内存访问导致页面失误——这会发生代价昂贵的磁盘 I/O 操作，应用程序会因此而用完它的时间配额。若能够确保应用程序所需要的页面总是位于物理内存中而不会被交换到磁盘上，则应用程序可以保证内存访问不会产生页面失误的结果，提供一致性、确定性并改善性能。

安全性

如果有私密的数据被保存在内存中，这些私密的数据可能会被交换出去并以未加密的形式存放到磁盘上。举例来说，如果一个用户的密钥在正常的情况下会以加密过的形式存放在磁盘上，则内存中密钥的一个未加密副本可能会出现在交换文件中。在高度重视安全性的环境中，这种行为也许是不能被接受的。有此类顾虑的应用程序可以要求包含密钥的内存总是保留在物理内存中。

当然，改变内核的行为可能会对系统整体的性能造成负面的影响。一个应用程序的确定性或安全性可以获得改善，但是当它的页面被锁定在内存中时，另一个应用程序的页面将会被交换出去。内核（如果我们信赖它的设计）总是会将最优选的页面换出至磁盘——也就是未来最不可能被用到的页面，所以当你变更内核的行为时，它必须将次优选的页面换出至磁盘。

锁定部分地址空间

POSIX 1003.1b-1993 定义了两个接口，可以把一个或多个页面锁定在物理内存中，以确保它们不会被换出至磁盘。第一个接口可以锁定指定地址区间：

```
#include <sys/mman.h>

int mlock (const void *addr, size_t len);
```

调用 `mlock()` 可以从 `addr` 开始把虚拟内存中 `len` 个字节锁定在物理内存里。执行成功时，此调用会返回 `0`；执行失败时，此调用会返回 `-1`，而且会将 `errno` 设定为适当的值。

一次成功的调用可以把包含 `[addr,addr+len]` 的所有物理页面锁定在内存里。举例来说，如果一次调用仅指定了单一字节，则包含该字节的页面会被整体锁定在内存中。POSIX 标准规定 `addr` 应该对齐一个页面的边界。Linux 不会强制执行此要求，只会在有需要时默默地将 `addr` 调整至最接近的页面。然而，需要移植到其他系统的程序应该确保 `addr` 位于一个页面的边界上。

有效的 `errno` 代码包括：

`EINVAL`

参数 `len` 为负值。

`ENOMEM`

调用者试图锁定比 `RLIMIT_MEMLOCK` 资源限制所允许的更多的页面（参见“锁定限制”一节）。

`EPERM`

`RLIMIT_MEMLOCK` 资源限制的值为 `0`，但是进程并不具有 `CAP_IPC_LOCK` 能力（同样参见“锁定限制”一节）。

> 一个子进程无法跨越 `fork()` 继承被锁定的内存。然而，由于 Linux 中地址空间的写入时才复制（copy-on-write）行为，一个子进程的页面会被有效地锁定在内存中，直到子进程写入它们。

举一个例子，假设程序会在内存中保存一个已解密的字符串，于是它的进程可以使用如下的程序代码锁定包含该字符串的页面：

```
int ret;

/* 将 "secret" 锁定在内存中 */
ret = mlock (secret, strlen (secret));
if (ret)
        perror ("mlock");
```

锁定整个地址空间

如果一个进程想将它的整个地址空间锁定在物理内存中，mlock() 并不适用。对于此类用途 —— 常见于实时应用程序，POSIX 定义了一个系统调用，可用于锁定整个地址空间：

```
#include <sys/mman.h>

int mlockall (int flags);
```

调用 mlockall() 可以把当前进程地址空间中的所有页面锁定在物理内存中。flags 参数（此参数可以被设定成下列两值的按位 OR 逻辑运算）可用于控制此行为：

MCL_CURRENT

如果设定了，此值会指示 mlockall() 把当前被映射的所有页面 —— 堆栈、数据段、映射文件等 —— 锁定在进程的地址空间。

MCL_FUTURE

如果设定了，此值会指示 mlockall() 把将来会被映射至地址空间的所有页面也锁定在内存。

多数应用程序会指定这两值的按位 OR 逻辑运算。

执行成功时，此调用会返回 0；执行失败时，它会返回 -1 并且将 errno 设定为下列其中一个错误代码：

EINVAL

参数 flags 是一个负值。

ENOMEM

调用者试图锁定比 RLIMIT_MEMLOCK 资源限制所允许的更多的页面（参见“锁定限制”一节）。

EPERM

RLIMIT_MEMLOCK 资源限制的值为 0，但是进程并不具有 CAP_IPC_LOCK 能力（同样参见 “锁定限制” 一节）。

替内存解锁

为了替物理内存中的页面解锁，再次允许内核于有需要时将相应页面换出至磁盘，POSIX 标准化了另外两个接口：

```
#include <sys/mman.h>

int munlock (const void *addr, size_t len);
int munlockall (void);
```

系统调用 munlock() 可以替从 addr 开始的页面（扩展至 len 个字节）解锁，它可以让 mlock() 的作用失效。系统调用 munlockall() 可以让 mlockall() 的作用失效。这两个调用会在执行成功时返回 0，以及在错误发生时返回 -1 并且将 errno 设定成下列的其中一个值：

EINVAL
: 参数 len 是无效的（仅适用于 munlock()）。

ENOMEM
: 所指定的页面有些是无效的。

EPERM
: RLIMIT_MEMLOCK 资源限制的值为 0，但是进程并不具有 CAP_IPC_LOCK 能力（同样参见“锁定限制”一节）。

内存的锁定不可嵌套。因此单一 munlock() 或 munlockall() 替一个被锁定的页面解锁时，不会理会该页面被 mlock() 或 mlockall() 锁定了多少次。

锁定限制

因为锁定内存会影响系统的整体性能 —— 的确，如果有太多页面被锁定，会导致内存分配失败，所以 Linux 会对一个进程可以锁定多少页面加上限制。

具备 CAP_IPC_LOCK 能力的进程可以将任何数目的页面锁定至内存。不具此能力的进程只能锁定 RLIMIT_MEMLOCK 个字节。默认情况下，此资源限制为 32 KB —— 足以将一笔或两笔具私密性的数据锁定在内存中，却又不至于影响系统的性能（第六章探讨过资源限制以及如何取得与变更此值）。

页面位于物理内存内吗？

为了调试与诊断，Linux 提供了 mincore() 函数，此函数可判断指定内存区域是否位于物理内存中或是已被换出至磁盘：

```
#include <unistd.h>
#include <sys/mman.h>

int mincore (void *start,
             size_t length,
             unsigned char *vec);
```

调用 mincore()会提供一个向量，描述了进行系统调用时一个映射的哪些页面位于物理内存内。mincore() 会经vec 返回此向量，用于描述从 start（必须与页面对齐）开始的页面并扩展至 length 个字节（不必与页面对齐）。vec 中每个字节对应于所指定区域中的一个页面，第一个字节用于描述第一个页面，依此类推。因此，vec 必须至少足以包含 (length-1+ *page size*) / *page size* 个字节。如果页面存在于物理内存中，则相应字节中的最低位会被设为 1，否则会被设为 0。其他位目前未定义，保留给未来使用。

执行成功时，此调用会返回 0；执行失败时，它会返回 -1 并且将 errno 设定为下列其中一个值：

EAGAIN
: 内核的可用资源不足以完成此请求。

EFAULT
: 参数 vec 指向了一个无效的地址。

EINVAL
: 参数 start 并未对齐一个页面的边界。

ENOMEM
: [address,address+1] 所包含的内存并非基于文件映射的一部分。

目前，此系统调用仅适用于以 MAP_SHARED 所创建的基于文件的映射。这是使用此调用时最大的限制。

投机取巧的分配策略

Linux 采用了投机取巧的分配（*opportunistic allocation*）策略。当一个进程向内核请求额外的内存时 —— 例如，通过扩大它的数据段，或者通过创建一个新的内存映射 —— 内核会对额外的内存做出承诺，但是不会实际提供任何物理存储空间。只有在进程写入刚才所分配的内存时，内核才会通过把所承诺的内存转换成物理的内存分配来兑现它的承诺。内核会按照需要以逐页的方式进行请求页面调度以及写入时才复制的操作。

此行为具有若干优点。首先，延迟分配内存的做法让内核得以延后大部分的工作，直到最后一刻——必须满足所承诺的内存时。其次，因为内核会以逐页的方式进行请求页面调度来满足此类请求，所以只有实际在使用的物理内存才需要耗用物理的存储空间。最后，所承诺的内存数量可远超过物理内存的数量以及可利用的交换空间。此特性称为过度承诺（*overcommitment*）。

过度承诺与内存不足

如果每个所请求的内存页面在分配的时候（而非使用的时候）必须存在于物理存储空间内，则过度承诺让系统得以运行比原本更多且更大的应用程序。若无过度承诺，一个 2 GB 文件的写入时才复制映射将需要内核保留 2 GB 的存储空间；若有过度承诺，一个 2 GB 文件的映射只需要替进程实际写入数据的每个页面保存存储空间。同样，若无过度承诺，每个 `fork()` 将需要足够的可用空间以便复制地址空间，即使大多数的页面从未经历写入时才复制。

然而，如果进程试图满足多个未决的承诺，而且超出了系统所具有的物理内存以及交换空间，结果会如何？在此情况下，必定会有一个或多个承诺无法被满足。因为内核已对内存做出了承诺——请求内存的系统调用已成功返回，而且进程会试图使用所承诺的内存，内核只能通过杀掉一个进程来释放可用的内存。

当过度承诺导致内存无法满足所承诺的请求时，我们说发生了内存不足（*out of memory*，常简写为 OOM）的情况。内核对 OOM 情况的响应是使用 *OOM killer*（内存不足杀手程序）选出适合终止的进程。因此，内核会试图找出耗用了最多内存而且最不重要的进程。

OOM 情况并不常见——因此允许过度承诺仍然可以获得很大的好处。然而，可以肯定的是，此情况是不受欢迎的，而且通过 OOM killer 以不可预测的方式终止一个进程往往是不可接受的。

对于会发生此情况的系统而言，内核允许我们由文件 */proc/sys/vm/overcommit_memory* 以及等效的 *sysctl* 参数 `vm.overcommit_memory` 来停止过度承诺。

此参数的默认值为 `0`，用于指示内核执行探试性过度承诺策略（heuristic overcommitment strategy），允许合理的过度承诺，但是不允许异乎寻常的过度承诺。值为 `1` 时，允许所有的过度承诺，不管后果为何。这个选项适用于某些内存密集型应用程序，例如科学领域所用到的那些程序，它们会试图请求比它们所需要的还要多很多的内存。

值为 2 时会完全禁止过度承诺，并且会启用 *strict accounting*（严格记账）模式。在此模式中，内存的过度承诺会被限制成交换区的大小加上一个可设定物理内存的百分比。这个可设定的百分比可由文件 */proc/sys/vm/overcommit_ratio* 或等效的 *sysctl* 参数 `vm.overcommit_ratio` 来设定。默认值为 `50`，用于将内存的承诺限制成交换区的大小加上物理内存的一半。因为物理内存中还包含了内核、页表、系统所保留的页面、被锁定的页面等，所以实际上只有部分的物理内存是可交换的以及保证能够满足承诺。

strict accounting 的使用须谨慎为之！许多系统设计者拒绝接受 OOM killer 的做法，认为 strict accounting 是一个万灵丹。然而，应用程序所进行的许多非必要的分配往往会达到过度承诺的地步，让此行为得以成为虚拟内存背后的主要动机之一。

第九章

信号

信号（*signal*）就是软件中断所提供的用于处理异步事件的一个机制。事件可以来自系统外部，例如当用户产生中断字符（通常是由 Ctrl+C）时；或者来自程序或内核内部的活动，例如当进程执行到“除以零”（divides by zero）的程序代码。因为信号是进程间通信的主要形式，一个进程还可以传送信号给另一个进程。

关键在于不只是事件发生的方式是异步的 —— 例如，程序执行时用户可以在任何时间点上按下 Ctrl+C 键—— 而且程序处理信号的方式也是异步的。当信号被递送时，程序端会以异步的方式调用向内核注册的信号处理函数。

早期的 Unix 就已经包含信号。然而随着时间的推移，它们已经有所演进 —— 最值得注意的是可靠性（因为有可能会失去信号）以及功能性（因为信号现在可能会承载用户定义的有效载荷）。起初，不同的 Unix 系统对信号做了不兼容的更动。所幸，POSIX 将信号的处理标准化，解决了此类问题。这个标准也就是 Linux 所提供的以及此处所要探讨的信号处理机制。

本章的一开始我们会概述信号并探讨信号的使用与误用。接着会探讨用于管理和操作信号的各种 Linux 接口。

具有实用价值的应用程序多半必须跟信号交互。就算你刻意把应用程序设计成进行通信时不需要用到信号 —— 这通常是一个好主意，某些情况下你仍然不得不与信号为伍，例如当程序收到终止信号时。

信号的概念

信号具有一个非常精确的生命周期。首先要引发（raise,有时又称为送出或产生）一个信号。然后内核会存储该信号，直到它可以被传递出去。最后，一旦有时间，内核会适当地处理该信号。内核可以执行下列三种动作中的一种，这取决于进程要求它做什么：

忽略该信号

不会采取任何动作。有两种信号不能忽略：SIGKILL 与 SIGSTOP。这是因为系统管理者需要能够杀掉或停止进程，如果一个进程可以选择忽略 SIGKILL（使得它无法被杀掉）或 SIGSTOP（使得它无法被中止），则它将有权利予以规避。

捕获以及处理该信号

内核可暂停进程的当前程序代码执行路径，跳跃至先前所注册的一个函数。然后进程将会执行此函数。一旦进程从此函数返回，它将会跳回它捕获到信号时暂停之处。

SIGINT 与 SIGTERM 是两个常被捕获的信号。进程捕获 SIGINT 可以处理用户所产生的中断字符，例如一个终端可以捕获此信号并且返回主要的提示符。进程捕获 SIGTERM 可以在终止之前进行必要的清理工作，例如关闭网络连接或移除临时文件。至于 SIGKILL 与 SIGSTOP 则无法被捕获。

执行默认的动作

此动作取决于被送出的信号。默认的动作通常是终止进程。例如，使用 SIGKILL 的情况。然而，许多信号是提供给在特殊情况下与特定目的有关的程序设计者，而且这些信号在默认情况下会被忽略，因为许多程序对它们并不感兴趣。稍后我们将查看各种信号以及它们的默认动作。

照惯例，当一个信号被递送时，处理信号的函数无法知道发生了什么事，除非所发生的是特定的信号。如今，内核会替想接收信号的程序设计者提供许多操作环境，而且信号甚至可以传递用户定义的数据，如同较新且更高级的 IPC 机制。

信号标识符

每个信号都会有一个以 SIG 开头的标识符。例如。SIGINT 代表当用户按下 Ctrl+C 键时所送出的信号，SIGABRT 代表当进程调用 abort() 函数时所送出的信号，而 SIGKILL 则代表当一个进程被强行终止时所送出的信号。

这些信号标识符全部定义在一个头文件中，可从 <signal.h> 引入。这些信号标识符只是用于代表正整数的预处理器定义 —— 也就是每个信号的标识符（名称）会被关联到

一个整数值。信号的名称与整数值的映射取决于实现，而且不同的 Unix 系统上会有所差异，虽然最前面的 12 个信号通常采用相同的映射方式（例如，SIGKILL 就是声名狼藉的编号为 9 的信号）。一个好的程序设计者总是会使用信号的可供人类阅读的名称，而绝对不会使用它的整数值。

信号的编号始于 1（通常就是 SIGHUP），而且会继续线性地向上增长。总共大约有 31 个信号，但是多数程序只会固定使用其中的几个信号。没有任何信号会使用值为 0 的编号，这个特殊值就是所谓的空信号（*null signal*）。事实上，空信号并不是十分重要——它并不会具有一个特殊的名称，但是有些系统调用（例如 kill()）会使用值为 0 的编号来代表一个特殊情况。

你可以使用 *kill -l* 这道命令列出你的系统所支持的信号。

Linux 所支持的信号

表 9-1 列出了 Linux 所支持的信号。

表 9-1：信号列表

信号	说明	默认行为
SIGABRT	由 abort() 送出	终止（terminate）并转储核心（core dump）
SIGALRM	由 alarm() 送出	终止
SIGBUS	硬件或对齐错误	终止并转储核心
SIGCHLD	子进程已被终止	忽略
SIGCONT	进程停止后继续执行	忽略
SIGFPE	算术异常事件	终止并转储核心
SIGHUP	进程的控制终端被关闭（最常见的情况是用户注销系统）	终止
SIGILL	进程试图执行一个非法指令	终止并转储核心
SIGINT	用户产生了中断字符（Ctrl+C）	终止
SIGIO	异步 I/O 事件	终止（注 a）
SIGKILL	无法捕获的进程终止信号	终止
SIGPIPE	进程将数据写入一个管道，但是读取者并不存在	终止
SIGPROF	概要分析定时器到期（profiling timer expired）	终止
SIGPWR	电力供应失败	终止
SIGQUIT	用户产生了结束字符（Ctrl+\）	终止并转储核心
SIGSEGV	无效的内存访问	终止并转储核心

表 9-1：信号列表（续表）

信号	说明	默认行为
SIGSTKFLT	协处理器（coprocessor）堆栈错误	终止（注 b）
SIGSTOP	暂时停止进程的执行	停止（stop）
SIGSYS	进程试图执行一个无效的系统调用	终止并转储核心
SIGTERM	可以捕获的进程终止信号	终止
SIGTRAP	遇到断点（break point）	终止并转储核心
SIGTSTP	用户产生了暂停字符（Ctrl+Z）	停止
SIGTTIN	后台进程需要读取控制终端	停止
SIGTTOU	后台进程需要写入控制终端	停止
SIGURG	紧急 I/O 挂起	忽略
SIGUSR1	进程所定义的信号	终止
SIGUSR2	进程所定义的信号	终止
SIGVTALRM	以 ITIMER_VIRTUAL 标志调用 setitimer() 时会产生此信号	终止
SIGWINCH	控制终端窗口的大小被变更	忽略
SIGXCPU	超出了处理器资源限制	终止并转储核心
SIGXFSZ	超出了文件资源限制	终止并转储核心

注 a： 其他的 Unix 系统，例如 BSD，会忽略此信号。

注 b： Linux 内核不再产生此信号，保留它只是为了向下兼容。

还存在一些其他的信号值，但是它们被定义成与其他值等效：SIGINFO 被定义成 SIGPWR（注 1），SIGIOT 被定义成 SIGABRT，而 SIGPOLL 与 SIGLOST 被定义成 SIGIO。

有了这个速查表之后，让我们深入探讨表中的每个信号：

SIGABRT

abort() 函数会将此信号送至调用它的进程。此进程然后会终止并产生一个 core 文件。在 Linux 中，当条件失败时，assert() 之类的断言（assertion）会调用 abort()。

注 1： 只有 Alpha 架构会定义此信号，它并不会存在于任何其他的机器架构上。

SIGALRM

当定时器到期时，alarm() 以及(使用了 ITIMER_REAL 标志的)setitimer() 函数会将此信号送至调用它们的进程。第十章会讨论这两个函数以及相关的函数。

SIGBUS

当进程遇到内存保护（这会产生 SIGSEGV）以外的硬件失误时，内核便会引发此信号。在传统的 Unix 系统上，此信号用于表示各种不能恢复的错误，例如未对齐的内存访问。然而，Linux 内核会自动修正此类错误，而不会产生此信号。当一个进程以不适当的方式访问由 mmap()（参见第八章对内存映射所做的讨论）所创建的一个内存区域时，内核并不会引发此信号。除非此信号被捕获，否则内核将终止进程以及产生核心转储。

SIGCHLD

每当一个进程终止（terminate）或停止（stop）时，内核会将此信号送往该进程的父进程。因为 SIGCHLD 默认会被忽略，所以如果父进程对其子进程的存活感兴趣，必须显式予以捕获与处理。此信号的处理程序通常会调用 wait()（参见第五章所做的讨论）以便取得子进程的 pid 以及结束代码。

SIGCONT

当进程在停止之后继续执行时，内核便会送出此信号。默认情况下，此信号会被忽略，但是如果进程想在继续执行时进行一个动作，则可以予以捕获。此信号常用于希望刷新屏幕的终端或编辑器。

SIGFPE

尽管它如此命名，但此信号可用于表示任何的算术异常事件，而不仅是只与浮点运算（floating-point operation）有关。此类异常事件包括上溢出（overflow）、下溢出（underflow）以及除以零（division by zero）。默认的动作是终止进程以及产生一个 core 文件，但是进程也可以依需要捕获及处理此信号。注意，如果进程选择继续运行下去，进程的行为以及出问题的运算结果并未定义。

SIGHUP

每当会话的终端断线时，内核便会将此信号送往会话的首领进程(session leader)。当会话的首领进程终止时，内核还会将此信号送往前台进程组中的每个进程。默认动作是终止进程，这是有道理的 —— 此信号间接表示用户已经注销。daemon 进程使用了一个机制“过载”(overload)此信号，以指示它们重新加载它们的配置文件。例如，传送 SIGHUP 给 Apache，可指示它重新加载 *httpd.conf*。将 SIGHUP 用于此目的是一个共同的惯例，但是并非强制性的。此做法之所以安全是因为 daemon 并不具有控制终端，因此一般情况下应该不会收到此信号。

SIGILL

当一个进程试图执行一个非法的机器指令时，内核便会送出此信号。默认的动作是终止此进程并产生核心转储。进程可以选择捕获及处理 SIGILL，但是这么做之后它们的行为是未定义的。

SIGINT

当用户键入中断字符（通常是 Ctrl+C）时，此信号会被送给前台进程组中所有进程。默认行为就是终止，然而进程可以选择捕获并处理此信号，而且通常这么做是为了在终止之前进行清理的工作。

SIGIO

当一个 BSD 形式的异步 I/O 事件产生时，便会送出此信号。此形式的 I/O 很少被使用在 Linux 上（参见第四章对 Linux 上常见高级 I/O 技术所做的讨论）。

SIGKILL

此信号由 kill() 系统调用发送，它的存在为系统管理者提供了一个必定会成功的方法，可用于无条件杀掉一个进程。此信号无法被捕获或忽略，而且它的结果总是终止进程。

SIGPIPE

如果一个进程将数据写入一个管道，但是读取者已经被终止，则内核便会引发此信号。默认动作就是终止进程，不过此信号可以被捕获及处理。

SIGPROF

概要分析定时器到期（profiling timer expire）时，如果 setitimer() 函数使用了 ITIMER_PROF 标志，便会产生此信号。默认动作是终止进程。

SIGPWR

此信号因系统而异。在Linux 系统上，它代表电池电力不足的情况（例如，在一个不断电系统（UPS）中）。于是监视 UPS 的 daemon 会将此信号传送给 *init*，接着 *init* 的响应就是清理以及关闭系统 —— 希望在电源耗尽之前完成！

SIGQUIT

当用户将结束字符（通常是 Ctrl+\）提供给终端时，内核便会将此信号送往前台进程组中所有进程。默认动作就是终止进程以及产生核心转储。

SIGSEGV

当进程试图进行一个无效的内存访问时，此信号（它的名称源自 *segmentation violation*）就会被送给该进程。这包括访问未被映射的内存、读取禁止读取的内存、执行禁止执行内存中的程序代码或者写入禁止写入的内存。进程可以捕获及处理此信号，但是默认的动作是终止进程以及产生核心转储。

SIGSTOP

此信号仅由 kill() 函数发送。它会无条件停止一个进程，而且无法被捕获或忽略。

SIGSYS

当进程试图调用一个无效的系统调用时，内核便会将此信号传送给该进程。如果二进制文件被构建在较新版的操作系统上（具有较新版的系统调用），但是却在较旧版的操作系统上运行便会发生此情况。适当地构建二进制文件，让它们通过 *glibc* 进行系统调用应该就不会收到此信号了。事实上，无效的系统调用应该返回 -1 并将 errno 设定成 ENOSYS。

SIGTERM

此信号仅由 kill() 发送，它让用户得以优雅地终止一个进程（默认动作）。进程可以选择捕获此信号，以及在终止之前进行清理的工作，但是捕获此信号被认为是粗鲁的行为而且无法及时终止。

SIGTRAP

当进程遇到一个断点（break point）时，内核便会将此信号传送给该进程。通常，调试程序会捕获此信号，而其他进程会忽略它。

SIGTSTP

当用户提供暂停字符（通常是 Ctrl+Z）时，内核便会将此信号送往前台进程组中所有进程。

SIGTTIN

当进程试图读取它的控制终端时，此信号会被传送给位于后台的该进程。默认动作是停止该进程。

SIGTTOU

当进程试图写入它的控制终端时，此信号会被传送给位于后台的该进程。默认动作是停止该进程。

SIGURG

当带外（out-of-band，常简写为 OOB）数据已经抵达一个 socket 时，内核便会将此信号送往进程。带外数据已经超出了本书的范围。

SIGUSR1 与 SIGUSR2

这两个信号可用于用户所定义的用途，内核绝不会引发这两个信号。进程可将 SIGUSR1 与 SIGUSR2 用于任何用途。一个常见的用途是指示 daemon 进程表现不同的行为。默认动作是终止进程。

SIGVTALRM

当使用 ITIMER_VIRTUAL 标志所创建的定时器到期时，setitimer() 函数便会送出此信号。定时器的相关细节可参考第十章。

SIGWINCH

当终端窗口的大小有所变动时，内核便会将此信号送往前台进程组中所有进程。默认情况下，进程会忽略此信号，但是如果它们知道自己的终端的窗口大小，它们可以选择捕获及处理此信号。*top* 程序便是一个捕获此信号的好例子，它会在运行时调整窗口的大小并且监视它的响应。

SIGXCPU

当一个进程超出它的软性处理器限制（soft processor limit）时，内核便会引发此信号。内核将继续引发此信号，每秒一次，直到进程结束或超出它的硬性处理器限制（hard processor limit）。一旦超出硬性限制，内核便会给进程传送一个 SIGKILL。

SIGXFSZ

当一个进程超出它的文件大小限制（file size limit）时，内核便会引发此信号。默认的动作是终止进程，但如果此信号被捕获或忽略，系统调用将会因为超出了文件大小的限制而返回 -1，并且将 errno 设定为 EFBIG。

基本的信号管理

signal() 是最简单且最旧的信号管理接口。ISO C89 标准（仅标准化了所支持信号的最小公分母）所定义的 signal() 系统调用非常基本。Linux 经其他接口（本章稍后有说明）对信号提供了更充分的控制。因为 signal() 最为基本，而且由于它存在于 ISO C 中，相当常见，所以我们首先会说明它：

```
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal (int signo, sighandler_t handler);
```

signal() 用于移除收到 signo 信号时所采取的当前动作，改用 handler 所指定的信号处理程序来处理该信号。signo 可设定成前一节所探讨的信号名称，例如 SIGINT 或 SIGUSR1。别忘了，进程无法捕获 SIGKILL 以及 SIGSTOP，所以替这两个信号设置处理程序毫无意义。

handler 函数必须返回 void，这是有道理的，因为（不同于一般函数）程序中没有一个标准的地方可供此函数返回。此函数具有一个参数（一个整数），也就是所要处理

信号的信号标识符（例如SIGUSR2）。这样一个函数就可以处理多个信号。此函数的原型具有如下的形式：

```
void my_handler (int signo);
```

Linux 是以一个类型定义（typedef）sighandler_t 来定义此原型的。其他 Unix 系统则会直接使用函数指针，有些系统会使用自己所定义的类型，但可能不是叫做 sighandler_t。寻求可移植性的程序不应该直接引用此类型。

当内核送出信号给一个注册过信号处理程序的进程时，内核会暂停程序的正常指令流的执行，并且调用信号处理程序。信号的值（也就是最初提供给 signal() 的 signo）会被传递给处理程序。

你还可以使用 signal() 指示内核为当前进程忽略指定的信号，或者将信号重置为默认行为。这可以通过为 handler 参数使用下面的值来完成：

SIG_DFL
: 将 signo 所指定的信号的行为设定为默认值。以 SIGPIPE 为例，进程将会终止。

SIG_IGN
: 忽略 signo 所指定的信号。

执行成功时，signal() 函数会返回信号先前的行为，这将会是一个指针，指向一个信号处理程序（SIG_DFL 或 SIG_IGN）。发生错误时，此函数会返回 SIG_ERR，但是不会设定 errno。

等待一个信号，任何信号

POSIX 所定义的 pause() 可用于调试以及编写示范性程序代码片段，它会让一个进程进入休眠状态，直到它收到一个信号，而若不处理此信号，则会终止进程：

```
#include <unistd.h>

int pause (void);
```

pause() 只会在接收到被捕获的信号时才会返回，在此情况下，信号会被处理，而且 pause() 会返回 -1 并且将 errno 设定为 EINTR。如果内核引发了一个被忽略的信号，则进程不会被唤醒。

在 Linux 内核中，pause() 是最简单的系统调用之一。它只会完成两项工作。首先，它会让进程进入可中断的休眠状态。其次，它会调用 schedule()，以便调用 Linux

的进程调度程序，找出另一个进程来运行。因为此进程实际上并未在等待什么，所以内核将不会唤醒它，除非它收到一个信号。这整个过程只需要用到两行 C 程序代码（注2）。

范例

让我们来看两个简单的例子。第一个例子替 SIGINT 注册了一个信号处理程序（只会输出一段信息），然后终止程序（因为 SIGINT 总是会这么做）：

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

/* SIGINT 的处理程序 */
static void SIGINT_handler (int signo)
{
        /*
         * 在技术上，你不应该在单一处理程序中使用 printf()，
         * 但它不是世界末日。
         * 我将在“可重入性”那一节说明为什么。
         */
        printf ("Caught SIGINT!\n");
        exit (EXIT_SUCCESS);
}

int main (void)
{
        /*
         * 将 SIGINT_handler 注册为我们用于处理 SIGINT 的
         * 信号处理程序。
         */
        if (signal (SIGINT, SIGINT_handler) == SIG_ERR) {
                fprintf (stderr, "Cannot handle SIGINT!\n");
                exit (EXIT_FAILURE);
        }

        for (;;)
                pause ();

        return 0;
}
```

在接下来的例子中，我们替 SIGTERM 与 SIGINT 注册了相同的处理程序。我们还将 SIGPROF 的行为重置为默认值（也就是终止进程）并且忽略 SIGHUP（否则将会终止进程）：

注 2： 事实上，pause() 还不算最简单的系统调用，getpid() 与 gettid() 才是最后的优胜者，因为它们只需要用到一行 C 程序代码。

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

/* SIGINT 的处理程序*/
static void signal_handler (int signo)
{
        if (signo == SIGINT)
                printf ("Caught SIGINT!\n");
        else if (signo == SIGTERM)
                printf ("Caught SIGTERM!\n");
        else {
                /* 这应该不可能发生 */
                fprintf (stderr, "Unexpected signal!\n");
                exit (EXIT_FAILURE);
        }
        exit (EXIT_SUCCESS);
}

int main (void)
{
        /*
         * 将 signal_handler 注册为我们用于处理 SIGINT 的
         * 单一处理程序。
         */
        if (signal (SIGINT, signal_handler) == SIG_ERR) {
                fprintf (stderr, "Cannot handle SIGINT!\n");
                exit (EXIT_FAILURE);
        }

        /*
         * 将 signal_handler 注册为我们用于处理 SIGTERM 的
         * 单一处理程序。
         */
        if (signal (SIGTERM, signal_handler) == SIG_ERR) {
                fprintf (stderr, "Cannot handle SIGTERM!\n");
                exit (EXIT_FAILURE);
        }

        /* 将 SIGPROF 的行为重置为默认值。*/
        if (signal (SIGPROF, SIG_DFL) == SIG_ERR) {
                fprintf (stderr, "Cannot reset SIGPROF!\n");
                exit (EXIT_FAILURE);
        }

        /* 忽略 SIGHUP。*/
        if (signal (SIGHUP, SIG_IGN) == SIG_ERR) {
                fprintf (stderr, "Cannot ignore SIGHUP!\n");
                exit (EXIT_FAILURE);
        }

        for (;;)
```

```
                pause ();

        return 0;
}
```

执行与继承

当一个进程首次被执行时，所有的信号会被设定为它们的默认行为，除非父进程（也就是执行新进程的进程）忽略它们，在此情况下，新创建的进程也会忽略那些信号。换句话说，父进程所捕获的任何信号在新的进程中都会被重置为默认行为，然而其他信号则保留原样。这是有道理的，因为一个刚被执行的进程不会共享其父进程的地址空间，因此不会存在任何已注册的信号处理程序。

此行为在进程的执行上具有一个重要的用途：当 shell 在后台执行一个进程时（或者当另一个后台进程在执行另一个进程时），刚被执行的进程应该会忽略中断与结束字符。因此，在 shell 执行一个后台进程之前，它应该将 SIGINT 与 SIGQUIT 设定为 SIG_IGN。因此，处理这些信号的程序通常会先检查以确定它们没有被忽略。例如：

```
/* 处理 SIGINT，但只发生在它未被忽略的情况下 */
if (signal (SIGINT, SIG_IGN) != SIG_IGN) {
        if (signal (SIGINT, SIGINT_handler) == SIG_ERR)
                fprintf (stderr, "Failed to handle SIGINT!\n");
}

/* 处理 SIGQUIT，但只发生在它未被忽略的情况下 */
if (signal (SIGQUIT, SIG_IGN) != SIG_IGN) {
        if (signal (SIGQUIT, SIGQUIT_handler) == SIG_ERR)
                fprintf (stderr, "Failed to handle SIGQUIT!\n");
}
```

这里需要设定一个信号行为来检查信号的行为，这显然是 signal() 接口中的一个缺陷。稍后，我们将研究一个没有此缺陷的函数。

使用 fork() 时的行为表现如你所预期的会有所不同。当一个进程调用 fork() 时，子进程会继承跟父进程完全相同的信号语义。这也是有道理的，因为子进程与父进程共享同一个地址空间，所以父进程的信号处理程序存在于子进程中。

将信号的编号映射至字符串

到目前为止所举的例子中，我们都是直接使用信号的名称。但是有些时候若能够将信号的编号转换成代表其名称的字符串会更方便（甚至是有此需要）。完成此事的方法不止一种。其中一种方法就是从一个静态定义的列表中取得字符串：

```
extern const char * const sys_siglist[];
```

sys_siglist 是一个字符串数组，用于保存系统所支持的信号名称，而且以信号的编号作为索引。

另一种方法就是 BSD 所定义的 psignal() 接口，Linux 也支持此接口：

```
#include <signal.h>

void psignal (int signo, const char *msg);
```

psignal() 会将你提供给 msg 参数的字符串输出至 stderr，后面跟着一个冒号、一个空格以及 signo 所指定信号的名称。如果 signo 是无效的，则所输出的信息也是如此。

strsignal() 是一个比较好的接口。尽管它未被标准化，但是 Linux 以及许多非 Linux 的系统也都支持它：

```
#define _GNU_SOURCE
#include <string.h>

char *strsignal (int signo);
```

strsignal() 会返回一个指针，指向 signo 所指定信号的一段描述。如果 signo 是无效的，则所返回的描述通常也是如此（事实上，有些支持此函数的 Unix 系统会返回 NULL）。所返回的字符串只有在 strsignal() 下次被调用之前是有效的，所以此函数并不具有线程安全性。

sys_siglist 通常是你的最佳选择。采用此做法，我们可以把稍早的信号处理程序改写成：

```
static void signal_handler (int signo)
{
        printf ("Caught %s\n", sys_siglist[signo]);
}
```

发送一个信号

kill() 系统调用是常见的 *kill* 实用程序的基础，可用于从一个进程将一个信号发送给另一个进程：

```
#include <sys/types.h>
#include <signal.h>

int kill (pid_t pid, int signo);
```

一般情况下（也就是 pid 的值大于 0），kill() 会将信号 signo 发送给 pid 所指定的进程。

如果 pid 的值为 0，则 signo 会被发送给进行调用的进程所属进程组中的每个进程。

如果 pid 的值为 -1，则 signo 会被发送给进行调用的进程有权对其发送信号的每个进程，不含其本身与 *init*。我们将在下一小节探讨控制信号发送的权限。

如果 pid 的值小于 -1，则 signo 会被发送给进程组 -pid。

执行成功时，kill() 会返回 0。只要有信号被送出，此调用就会被视为成功。执行失败时（没有信号被送出），此调用会返回 -1 并且将 errno 设定为下面的其中一个值：

EINVAL

　　signo 指定了无效的信号。

EPERM

　　进行调用的进程的权限不足以将一个信号发送给任何所请求的进程。

ESRCH

　　pid 所指定的进程或进程组不存在，或者是一个僵尸进程。

权限

为了发送一个信号给另一个进程，发送端进程需要具备适当的权限。一个具有 CAP_KILL 能力的进程（通常为 root 所拥有）可以发送一个信号给任何进程。若不具备此能力，发送端进程的有效或真实的用户ID必须等于接收端进程的真实或被保存的用户 ID。简而言之，一个用户只能传送一个信号给他自己的一个进程。

> Unix 系统替 SIGCONT 定义了一个异常：一个进程可以发送此信号给位于相同会话的任何其他进程。用户 ID 不需要相符。

如果 signo 的值为 0 —— 之前所提到的空信号，则此调用不会送出信号，但是仍旧会进行错误检查。这可用于测试一个进程是否具有适当的权限，能够将信号发送给所提供的进程或进程组。

范例

下面的例子会将 SIGHUP 发送给 pid 为 1722 的进程：

```
int ret;

ret = kill (1722, SIGHUP);
if (ret)
        perror ("kill");
```

上面这段范例程序代码与下面的 *kill* 实用程序的调用等效：

```
$ kill -HUP 1722
```

若想检查我们是否有权限将一个信号发送给 1722，但是不要实际送出任何信号，我们可以这么做：

```
int ret;

ret = kill (1722, 0);
if (ret)
        ; /* 我们的权限不足 */
else
        ; /* 我们具有权限 */
```

给自己发送一个信号

如果一个进程要发送一个信号给它自己，raise() 函数是一个简单的方法：

```
#include <signal.h>

int raise (int signo);
```

此调用：

```
raise (signo);
```

等效于下面的调用：

```
kill (getpid (), signo);
```

执行成功时，此调用会返回 0；执行失败时，则返回一个非零值，但是不会设定 errno。

给整个进程组传送一个信号

另一个方便的函数可让你轻易地将一个信号传送给指定进程组中的所有进程，因此对进程组 ID 取负值以及使用 kill() 也被视为太复杂：

```
#include <signal.h>

int killpg (int pgrp, int signo);
```

此调用：

```
killpg (pgrp, signo);
```

等效于下面的调用：

```
kill (-pgrp, signo);
```

即使 pgrp 的值为 0，以上的说明仍为真，在此情况下，killpg() 会将信号 signo 传送给进行调用的进程所属组中的每个进程。

执行成功时，killpg() 会返回 0；执行失败时，它会返回 -1 并且将 errno 设定为下面的其中一个值：

EINVAL
: signo 指定了无效的信号。

EPERM
: 进行调用的进程的权限不足以将一个信号发送给任何所请求的进程。

ESRCH
: pgrp 所指定的进程组并不存在。

可重入性

当内核引发了一个信号时，进程可能正在任意地方执行程序代码。举例来说，它可能正在进行一个重要的运算，如果被中断，将会在进程中留下不一致的状态 —— 例如，一个数据结构只被更新了一半，或者一项计算仅完成了一部分。进程甚至可能正在处理另一个信号。

当一个信号被引发时，信号处理程序无法知道进程正在执行什么程序代码，处理程序可能运行在任何程序代码中间。因此，重点在于，你的进程所安装的任何信号处理程序必须十分小心它所执行的动作以及它所触及的数据。信号处理程序不应该假设当进程遭中断时正在做什么。尤其是当它们在修改全局（也就是共享的）数据时，必须特别谨慎小心。一般而言，不让信号处理程序触及全局数据是一个好主意，然而下一节我们将探讨一个暂时阻挡信号递送的方法，因为此方法让信号处理程序与进程的其余部分所共享的数据可以被安全地操作。

那么系统调用以及其他链接库函数会怎么样呢？如果你的进程正在写入一个文件或正在配置内存，但是信号处理程序所写入的是相同的文件或也在调用 malloc()，会怎么样呢？或者，当信号被递送时，如果你的进程正在调用一个使用静态缓冲区的函数，例如 strsignal()，又会怎么样呢？

有些函数显然不可重入。如果一个程序正在执行一个不可重入的函数，此时内核引发了一个信号，接着信号处理程序也在调用相同的不可重入函数，于是混乱随之而来。一个可重入函数（*reentrant function*）就是一个可以在自己内部调用自己（或者在相同的进程中并行地从另一个线程来调用自己）的函数。为了可重入，一个函数不得操作静态数据，只能操作分配在堆栈上的数据或调用者所提供的数据，而且不得调用任何不可重入函数。

保证可重入的函数

编写一个信号处理程序时，你必须考虑到被中断的进程可能位于不可重入函数中（或者任何其他地方）。因此，信号处理程序只能使用可重入的函数。

各种标准颁布了具有信号安全性的函数 —— 也就是可重入的，因此可以安全地在信号处理程序中使用。其中最值得注意的是 POSIX.1-2003 与 Single UNIX Specification 所颁布的函数，在所有兼容的平台上，它们保证具有信号安全性。表 9-2 列出了这些函数。

表 9-2：保证具有信号安全性的可重入的函数

abort()	accept()	access()
aio_error()	aio_return()	aio_suspend()
alarm()	bind()	cfgetispeed()
cfgetospeed()	cfsetispeed()	cfsetospeed()
chdir()	chmod()	chown()
clock_gettime()	close()	connect()
creat()	dup()	dup2()
execle()	execve()	Exit()
_exit()	fchmod()	fchown()
fcntl()	fdatasync()	fork()
fpathconf()	fstat()	fsync()
ftruncate()	getegid()	geteuid()
getgid()	getgroups()	getpeername()

表 9-2：保证具有信号安全性的可重入的函数（续表）

getpgrp()	getpid()	getppid()
getsockname()	getsockopt()	getuid()
kill()	link()	listen()
lseek()	lstat()	mkdir()
mkfifo()	open()	pathconf()
pause()	pipe()	poll()
posix_trace_event()	pselect()	raise()
read()	readlink()	recv()
recvfrom()	recvmsg()	rename()
rmdir()	select()	sem_post()
send()	sendmsg()	sendto()
setgid()	setpgid()	setsid()
setsockopt()	setuid()	shutdown()
sigaction()	sigaddset()	sigdelset()
sigemptyset()	sigfillset()	sigismember()
signal()	sigpause()	sigpending()
sigprocmask()	sigqueue()	sigset()
sigsuspend()	sleep()	socket()
socketpair()	stat()	symlink()
sysconf()	tcdrain()	tcflow()
tcflush()	tcgetattr()	tcgetpgrp()
tcsendbreak()	tcsetattr()	tcsetpgrp()
time()	timer_getoverrun()	timer_gettime()
timer_settime()	times()	umask()
uname()	unlink()	utime()
wait()	waitpid()	write()

尽管还有更多具有安全性的函数，但是 Linux 与其他兼容于 POSIX 的系统仅保证这些函数是可重入的。

信号集

本章稍后我们将看到几个需要操作一组信号（信号集）的函数，例如被进程阻挡的一组信号或是一组等待进程处理的信号。信号集（*signal set*）函数可用于管理这些信号集：

```
#include <signal.h>

int sigemptyset (sigset_t *set);

int sigfillset (sigset_t *set);

int sigaddset (sigset_t *set, int signo);

int sigdelset (sigset_t *set, int signo);

int sigismember (const sigset_t *set, int signo);
```

sigemptyset() 用于初始化 set 所指定的信号集，以便清空它（将所有信号排除在信号集之外）。sigfillset() 用于初始化 set 所指定的信号集，以便填满它（将所有信号纳入信号集）。这两个函数都会返回 0。进一步使用信号集之前，你应该对信号集调用其中一个函数。

sigaddset() 用于将 signo 加入 set 所指定的信号集，而 sigdelset() 用于从 set 所指定的信号集中移除 signo。执行成功时，这两个函数都会返回 0；发生错误时，则会返回 -1，在此情况下，errno 会被设定为错误代码 EINVAL，表示 signo 是一个无效的信号标识符。

如果 signo 位于 set 所指定的信号集中，sigismember() 会返回 1；如果不是，会返回 0；如果发生错误，则返回 -1，在此情况下，errno 同样会被设定为 EINVAL，表示 signo 是无效的。

其他信号集函数

前面所提到的都是经过 POSIX 标准化的函数，你可以在任何现代的 Unix 系统上找到它们。此外，Linux 还提供以下非标准的函数：

```
#define _GNU_SOURCE
#define <signal.h>

int sigisemptyset (sigset_t *set);

int sigorset (sigset_t *dest, sigset_t *left, sigset_t *right);

int sigandset (sigset_t *dest, sigset_t *left, sigset_t *right);
```

如果 set 所指定的信号集是空的，则 sigisemptyset() 会返回 1，否则会返回 0。

sigorset() 会将信号集 left 与 right 的并集（二元 OR 运算）摆入 dest。sigandset() 会将信号集 left 与 right 的交集（二元 AND 运算）摆入 dest。执行成功时，这两个函数都会返回 0；发生错误时，则会返回 -1 并且将 errno 设定为 EINVAL。

尽管这些函数都很有用，但是想要与 POSIX 完全兼容的程序应该避免使用这些函数。

阻挡信号

稍早，我们曾讨论过可重入性以及在任何时候异步执行信号处理程序所引发的问题。我们已探讨了不可以在信号处理程序中调用的函数，因为它们本身就不是可重入的。

但如果你的程序需要让一个信号处理程序与程序中其他部分共享数据，该怎么做？如果你的程序中正在执行的部分不想要遇到任何的中断，包括来自信号处理程序的，该怎么做？我们将以上这些部分称为程序的临界区（*critical region*），而且我们会通过临时中止信号的递送来保护它们 —— 我们会说信号遭到了阻挡（*blocked*）。任何信号只要遭到阻挡就不会被处理，除非它们已不再遭到阻挡。一个进程可以阻挡任何数量的信号，遭到进程阻挡的一组信号称为信号掩码（*signal mask*）。

POSIX 定义了 —— 而且 Linux 实现了 —— 一个函数，可用于管理进程的信号掩码：

```
#include <signal.h>

int sigprocmask (int how,
                 const sigset_t *set,
                 sigset_t *oldset);
```

sigprocmask() 的行为取决于 how 的值，此值可以是下面其中一个标志：

SIG_SETMASK

将进行调用的进程的信号掩码变更为 set。

SIG_BLOCK

将 set 中的信号加入进行调用的进程的信号掩码。换言之，信号掩码会变更为当前掩码与 set 的并集（二元 OR 运算）。

SIG_UNBLOCK

将 set 中的信号从进行调用的进程的信号掩码中移除。换言之，信号掩码会变更为当前掩码与 set 的取反（二元 NOT 运算）的交集（二元 AND 运算）。对一个未受阻挡的信号进行 unblock（疏通）操作是不符合规定的。

如果 oldset 的值不是 NULL，则此函数会将先前的信号集放入 oldset。

如果 set 的值为 NULL，则此函数会忽略 how，而且不会变更信号掩码，但是会将信号掩码放入 oldset。换言之，将 null 值传进 set 可作为取得当前信号掩码的方法。

执行成功时，此调用会返回 0；执行失败时，它会返回 -1 并且会将 errno 设定为 EINVAL（表示 how 是无效的）或 EFAULT（表示 set 或 oldset 是一个无效的指针）。

阻挡 SIGKILL 或 SIGSTOP 是不被允许的。若试图将这两个信号其中之一加入信号掩码，则 sigprocmask() 会默默予以忽略。

取得未决信号

当内核引发了一个遭阻挡的信号时，它不会被递送。我们称它为未决（*pending*）信号。当一个未决信号不再被阻挡时，内核可以传递它，让进程能够加以处理。

POSIX 定义了一个函数，用于取得一组未决信号：

```
#include <signal.h>

int sigpending (sigset_t *set);
```

执行成功时，sigpending() 会把一组未决信号放入 set 并且返回 0；执行失败时，此调用会返回 -1 并且将 errno 设定为 EFAULT（表示 set 是一个无效的指针）。

等待一组信号

第三个 POSIX 所定义的函数让一个进程得以临时变更它的信号掩码，然后等待一个信号被引发，直到进程终止或是该信号被进程所处理：

```
#include <signal.h>

int sigsuspend (const sigset_t *set);
```

如果一个信号会终止进程，则 sigsuspend() 不会返回。如果一个信号被引发与处理，则 sigsuspend() 会在从信号处理程序返回之后返回 -1，并且将 errno 设定为 EINTR。如果 set 是一个无效的指针，则 errno 会被设定为 EFAULT。

sigsuspend() 常用于取得程序执行临界区期间已经到达而且被阻挡的信号。进程首先会使用 sigprocmask() 来阻挡一组信号并将旧的掩码存入 oldset。离开临界区之后，进程接着会调用 sigsuspend()，将 oldset 提供给 set。

高级信号管理

我们在本章一开始探讨了非常基本的 signal() 函数。因为它是标准 C 链接库的一部分，因此它能够反映它所运行的操作系统上关于操作系统能力的最小假设（minimal assumption），它对信号管理所提供的能力只不过是最小公分母（lowest common denominator）。另一种选择是经 POSIX 标准化的 sigaction() 系统调用，它所提供的信号管理能力就大得多了。尤其是当你的处理程序运行的时候，你可以使用它来阻挡指定信号的接收，以及当一个信号被引发的时候，你可以使用它来取得关于系统与进程状态的广泛数据：

```
#include <signal.h>

int sigaction (int signo,
               const struct sigaction *act,
               struct sigaction *oldact);
```

sigaction() 可用于变更 signo 所指定的信号行为，它可以是任何值（SIGKILL 与 SIGSTOP 除外）。如果 act 的值不是 NULL，则系统调用会变更 act 所指定信号的当前行为。如果 oldact 的值不是 NULL，则此调用会存储指定信号先前的行为（或是当前的行为，如果 act 的值是 NULL 的话）。

sigaction 结构让你可以对信号进行细粒度的控制（fine-grained control）。头文件 <sys/signal.h>（从 <signal.h> 引用）定义了下面的结构：

```
struct sigaction {
        void (*sa_handler)(int);   /* signal handler or action */
        void (*sa_sigaction)(int, siginfo_t *, void *);
        sigset_t sa_mask;          /* signals to block */
        int sa_flags;               /* flags */
        void (*sa_restorer)(void); /* obsolete and non-POSIX */
}
```

sa_handler 字段用于指示收到信号时所要采取的行动。如同 signal()，此字段的值可以是 SIG_DFL（代表默认动作）、SIG_IGN（用于指示内核忽略进程的信号）或是一个指向一个信号处理函数的指针。此函数的原型如同 signal() 所安装的一个信号处理程序：

```
void my_handler (int signo);
```

如果 SA_SIGINFO 被设定在 sa_flags 中，则会使用 sa_sigaction（而非 sa_handler）来指示信号处理函数。此函数的原型有些微不同：

```
void my_handler (int signo, siginfo_t *si, void *ucontext);
```

此函数的第一个参数是信号编号，第二个参数是一个 siginfo_t 结构，第三个参数是一个 ucontext_t 结构(转型成一个 void 指针)。此函数没有返回值。siginfo_t 替信号处理程序提供了大量的信息，我们马上会看到它。

注意，在某些机器架构（以及可能的其他 Unix 系统）上，sa_handler 与 sa_sigaction 位于同一个 union（联合数据结构）中，你不应该同时对这两个字段赋值。

sa_mask 字段提供了一组信号，系统应该在信号处理程序执行期间阻挡这组信号。这让程序设计者得以适当地保护多个信号处理程序的可重入性。当前正在被处理的信号也会被阻挡，除非 SA_NODEFER 标志也设定在 sa_flags 字段中。你无法阻挡 SIGKILL 或 SIGSTOP，这两个信号若出现在 sa_mask 中，此调用将默默地予以忽略。

sa_flags 字段是零个、一个或多个标志的位掩码，可用于变更 signo 所指定信号的处理方式。我们已经看过了 SA_SIGINFO 与 SA_NODEFER 等标志。sa_flags 字段的其他值还包括：

SA_NOCLDSTOP

如果 signo 的值为 SIGCHLD，则此标志用于指示系统不要在一个子进程停止或继续时送出通知。

SA_NOCLDWAIT

如果 signo 的值为 SIGCHLD,则此标志用于启用自动子进程回收功能(*automatic child reaping*)：子进程不会在终止时转变为僵尸进程，而且父进程不需要（也不会）对它们调用 wait()。参见第五章对子进程、僵尸进程以及 wait() 的详细说明。

SA_NOMASK

这是一个已废弃的非 POSIX 标志，等效于 SA_NODEFER（本节稍早讨论过）。你可以使用 SA_NODEFER 来取代此标志，但是你会在较旧的程序代码中看到此标志。

SA_ONESHOT

这是一个已废弃的非 POSIX 标志，等效于 SA_RESETHAND（稍后会讨论）。你可以使用 SA_RESETHAND 来取代此标志，但是你会在较旧的程序代码中看到此标志。

SA_ONSTACK

此标志用于指示系统在一个替代的信号堆栈（*alternative signal stack*）上—— 提供自 sigaltstack() ——调用指定的信号处理程序。如果你并未提供一个替代的堆栈，则会使用默认的堆栈 —— 也就是系统的行为如同你并未提供此标志时一

样。替代的信号堆栈很少被用到，尽管它们可以被用在某些线程堆栈较小的 pthread 应用程序上，以避免某些信号处理程序把堆栈用光。本书并不会进一步探讨 sigaltstack()。

SA_RESTART

此标志会使得遭信号中断的系统调用以 BSD 的形式重新启动。

SA_RESETHAND

此标志会启用“one-shot”（只做一次）模式。一旦从信号处理程序返回，该信号的行为会被重置为默认值。

sa_restorer 字段已废弃，而且不再用于 Linux。反正它不是 POSIX 的一部分，就假装它不存在，不要跟它打交道。

执行成功时，sigaction() 会返回 0；执行失败时，此调用会返回 -1 并且将 errno 设定为下面的其中一个错误代码：

EFAULT

act 或 oldact 是一个无效的指针 。

EINVAL

signo 是一个无效的信号，SIGKILL 或 SIGSTOP。

siginfo_t 结构

siginfo_t 结构也定义于 <sys/signal.h>，如下所示 ：

```
typedef struct siginfo_t {
        int si_signo;       /* signal number */
        int si_errno;       /* errno value */
        int si_code;        /* signal code */
        pid_t si_pid;       /* sending process's PID */
        uid_t si_uid;       /* sending process's real UID */
        int si_status;      /* exit value or signal */
        clock_t si_utime;   /* user time consumed */
        clock_t si_stime;   /* system time consumed */
        sigval_t si_value;  /* signal payload value */
        int si_int;         /* POSIX.1b signal */
        void *si_ptr;       /* POSIX.1b signal */
        void *si_addr;      /* memory location that caused fault */
        int si_band;        /* band event */
        int si_fd;          /* file descriptor */
};
```

此结构包含了许多要传递给信号处理程序的信息(如果你使用的是 sa_sigaction 而不是 sa_sighandler)。以现代计算机技术来看，许多人认为使用 Unix 信号模型来进行 IPC 是一个相当可怕的方法。或许问题在于，当这些人应该以 SA_SIGINFO 标志来使用 sigaction() 的时候，他们还是坚持使用 signal()。siginfo_t 结构为我们开启了一扇门，让我们得以在信号之外编写出更多的功能。

此结构中还有许多值得注意的数据，包括送出信号的进程的相关信息、信号起因的相关信息。下面是这些字段的详细描述：

si_signo
: 信号的编号。在你的信号处理程序中，第一个参数所提供的也是这个信息（因此可避免指针解除引用)。

si_errno
: 如果为非零值，则错误代码与此信号相关联。此字段适用于所有信号。

si_code
: 说明进程为何收到信号以及从何处收到（例如从 kill())。下一节我们将探讨它的可能值。此字段适用于所有信号。

si_pid
: 适用于 SIGCHLD，代表被终止进程的 PID。

si_uid
: 适用于 SIGCHLD，代表被终止进程拥有者的 UID。

si_status
: 适用于 SIGCHLD，代表被终止进程的结束状态。

si_utime
: 适用于 SIGCHLD，代表被终止进程所耗用的用户时间。

si_stime
: 适用于 SIGCHLD，代表被终止进程所耗用的系统时间。

si_value
: si_int 与 si_ptr 的 union（联合数据结构)。

si_int
: 适用于由 sigqueue()(参见“以 payload 送出信号”一节）所传送的信号，所提供的 payload 是一个整数。

si_ptr

适用于由 sigqueue()(参见“以 payload 送出信号”一节）所传送的信号，所提供的 payload 是一个 void 指针。

si_addr

适用于 SIGBUS、SIGFPE、SIGILL、SIGSEGV 以及 SIGTRAP，这个 void 指针包含了导致错误的地址。例如，就 SIGSEGV 的情况而言，此字段包含了发生内存访问错误的地址（因此通常为 NULL！）。

si_band

适用于 SIGPOLL，被列在 si_fd 中的文件描述符的 out-of-band（带外或紧急）与 priority（优先级）信息。

si_fd

适用于 SIGPOLL，已完成操作的文件的文件描述符。

si_value、si_int 以及 si_ptr 是相当复杂的议题，因为一个进程可以通过它们给另一个进程传递任意数据。因此，你可以通过它们来传送简单的整数或是传送指向一个数据结构的指针（注意，如果进程之间并未共享同一个地址空间，则指针的帮助也不大）。“以 payload 送出信号”一节将会探讨这些字段。

POSIX 只保证最前面三个字段适用于所有信号，所以你只应该在处理适用信号时才去访问其他字段。例如，只有在信号是 SIGPOLL 时，你才应该访问 si_fd 字段。

si_code 的奇妙世界

si_code 字段用于指示信号的起因。就用户所传送的信号而言，此字段用于指示如何传送信号。就内核所传送的信号而言，此字段用于指示为何传送信号。

下面的 si_code 字段值适用于任何信号，用于指示如何／为何（how/why）传送信号：

SI_ASYNCIO

之所以传送信号是因为已完成异步 I/O（参见第五章）。

SI_KERNEL

信号由内核引发。

SI_MESGQ

之所以传送信号是因为有一个 POSIX 信息队列（本书并未说明）的状态发生了变化。

SI_QUEUE

信号由 sigqueue() 传送（参见下一节）。

SI_TIMER

　　之所以传送信号是因为有一个 POSIX 定时器到期（参见第十章）。

SI_TKILL

　　此信号由 tkill() 或 tgkill() 传送。这些系统调用被线程所执行的链接库使用，但是本书并未说明。

SI_SIGIO

　　之所以传送信号是因为 SIGIO 被排入队列。

SI_USER

　　此信号由 kill() 或 raise() 传送。

下面的 si_code 字段值仅适用于 SIGBUS，用于指示所发生的是何种硬件错误：

US_ADRALNB

　　进程遇到了一个对齐错误（对齐的相关细节可参考第八章）。

BUS_ADRERR

　　进程访问了一个无效的物理地址。

BUS_OBJERR

　　进程引起了其他形式的硬件错误。

就 SIGCHLD 而言，下面各值用于指出是哪个子进程传送信号给它的父进程：

CLD_CONTINUED

　　子进程被停止但是已经恢复。

CLD_DUMPED

　　子进程被不正常终止。

CLD_EXITED

　　子进程由 exit() 正常终止。

CLD_KILLED

　　子进程被杀掉了。

CLD_STOPPED

　　子进程被停止了。

CLD_TRAPPED

　　子进程遇到了一个陷阱。

下面的值仅适用于 SIGFPE，用于说明所产生的是何种算术错误：

FPE_FLTDIV

进程所进行的浮点运算产生了除以零的结果。

FPE_FLTOVF

进程所进行的浮点运算产生了上溢出（overflow）的结果。

FPE_FLTINV

进程进行了一项无效的浮点运算。

FPE_FLTRES

进程所进行的浮点运算产生了不精确或无效的结果。

FPE_FLTSUB

进程所进行的浮点运算产生了一个超出范围的下标（out-of-range subscript）。

FPE_FLTUND

进程所进行的浮点运算产生了下溢出（underflow）的结果。

FPE_INTDIV

进程所进行的整数运算产生了除以零的结果。

FPE_INTOVF

进程所进行的整数运算产生了上溢出的结果。

下面的 si_code 字段值仅适用于 SIGILL，用于说明非法指令执行的性质：

ILL_ILLADR

进程试图进入一个非法的寻址模式。

ILL_ILLOPC

进程试图执行一个非法的操作码（opcode）。

ILL_ILLOPN

进程试图执行一个非法的操作数（operand）。

ILL_PRVOPC

进程试图执行一个需要特权的操作码。

ILL_PRVREG

进程试图执行一个需要特权的寄存器。

ILL_ILLTRP

进程试图进入一个非法的陷阱。

就以上这些值而言，si_addr 会指向发生错误的地址。

就 SIGPOLL 而言，下面的值用于指出信号是何种 I/O 事件产生的：

POLL_ERR

　　发生了一个 I/O 错误。

POLL_HUP

　　设备被挂断或者 socket 断线。

POLL_IN

　　文件有数据可供读取。

POLL_MSG

　　有信息可用。

POLL_OUT

　　文件可供写入。

POLL_PRI

　　文件有高优先级的数据可供读取。

下面的值适用于 SIGSEGV，用于描述两种无效的内存访问：

SEGV_ACCERR

　　进程以无效的方式访问一个有效的内存区域——也就是说，进程违反了内存访问权限。

SEGV_MAPERR

　　进程访问了一个无效的内存区域。

无论是以上何种情况，si_addr 都会包含发生错误的地址。

就 SIGTRAP 而言，下面的 si_code 字段值可用于指出所遇到的是何种陷阱：

TRAP_BRKPT

　　进程遇到了一个断点。

TRAP_TRACE

　　进程遇到了一个追踪陷阱。

注意，si_code 是一个值字段（value field）而不是一个位字段（bit field）。

以 payload 送出信号

正如我们在前一节所看到的，信号处理程序的注册需要用到 SA_SIGINFO 标志以及传递一个 siginfo_t 参数。siginfo_t 结构包含了一个名为 si_value 的字段，这是一个可选的 payload（有效载荷）（译注 1），会从信号产生者递送至信号接收者。

sigqueue() 函数（定义自 POSIX）让一个进程得以使用此 payload 来传送信号：

```
#include <signal.h>

int sigqueue (pid_t pid,
              int signo,
              const union sigval value);
```

sigqueue() 的行为类似于 kill()。执行成功时，signo 所指定的信号会被排入由 pid 所指定的进程或进程组的队列，而且此函数会返回 0。此信号的 payload 由 value 来指定，value 是一个整数与一个 void 指针所构成的 union：

```
union sigval {
        int sival_int;
        void *sival_ptr;
};
```

执行失败时，此调用会返回 -1 并且将 errno 设定为下面的其中一个值：

EINVAL

signo 指定了无效的信号。

EPERM

进行调用的进程的权限不足以将信号送往任何所请求的进程。传送一个信号所请求的权限如同 kill()的（参见“传送一个信号”一节）。

ESRCH

pid 所指定的进程或进程组不存在，或者是一个僵尸进程。

如同 kill()，测试权限的方法就是将空字符串（0）传入 signo。

范例

下面所举的例子会传送 SIGUSR2 信号给 pid 为 1722 的进程，此信号有一个整数的 payload，所要传送的是 404 这个值：

译注 1：payload（有效载荷）原指火箭所载送的弹头（或人造卫星），后来在数据通信中被引申为封包中不含标头，只包含实际要传送给接收者的数据。

```
sigval value;
int ret;

value.sival_int = 404;

ret = sigqueue (1722, SIGUSR2, value);
if (ret)
        perror ("sigqueue");
```

如果进程 1722 对 SIGUSR2 的处理使用了 SA_SIGINFO 处理程序,则它将发现 signo 被设定为 SIGUSR2，si->si_int 被设定为 404，以及 si->si_code 被设定为 SI_QUEUE。

结束语

许多 Unix 程序设计者并不看好信号。就内核与用户之间的通信而言，它是一个老旧过时的机制，最多只不过是 IPC 的原始形式。在多线程程序与事件循环的世界中，通常不适合使用信号。

然而，无论好坏，我们仍需要它们。信号是从内核接收许多通知（例如非法操作码执行的通知）的唯一方法。此外，信号是 Unix（因而也是 Linux）终止进程以及管理父／子进程关系的方法。因此，我们不得不使用信号。

信号不被看好的主要原因在于,很难让所编写出的信号处理程序能够适当地避开可重入(reentrancy）的问题。然而，如果你能够让处理程序保持简单，而且只使用表 9-2 所列的函数（如果你需要用到任何函数的话），则它们应该是安全的。

另一个问题，就信号的管理而言，许多程序设计者仍旧使用 signal() 与 kill() 而不使用 sigaction() 与 sigqueue()。如最后两节所示，使用 SA_SIGINFO 形式的信号处理程序时会大幅改善信号的能力和表现。虽然我自己不是信号的爱用者 —— 我希望看到信号机制被基于文件描述符的可轮询机制所取代，事实上 Linux 内核未来的版本将此事纳入了考虑 —— 但是为了让日子好过一点，我会避开它们的缺点并使用 Linux 的高级接口。

第十章
时间

在现代操作系统中，时间（time）可用于各种目的，许多程序都需要记录时间。内核会以三种方式测量流逝的时间：

壁钟时间（或真实时间）

这是真实世界中的实际时间和日期——也就是人们可以在壁钟上看到的时间。进程会在与用户交互或替事件记录时间戳时使用壁钟时间（*wall time*）。

进程时间

这就是一个进程在用户空间程序代码中所花的时间或是内核替进程工作所花的时间。进程之所以要在乎这种形式的时间，主要是为了概要分析（profiling）（译注 1）与统计（statistic）的目的——例如，测量特定运算要花多少时间完成。壁钟时间会令人误解测量进程的行为，因为由于 Linux 多任务的本质，一个运算的进程时间会远低于壁钟时间。一个进程还会将大量的时间花在等待 I/O（特别是键盘的输入）上。

单调时间

此时间来源会严格线性递增（strictly linearly increasing）。多数操作系统，包括 Linux，会使用系统正常运行时间（system uptime，从开机起算系统已经运作了多长时间）。壁钟时间有可能会变动——例如因为用户会设定它，而且系统会因时间偏移而持续调整它；此外，例如闰秒（leap second）也会让壁钟时间变得不精确。另一方面，system uptime 是时间的一个具确定性与不变性的表示法。一个单调时间来源（monotonic time source）的重要方面不在于当前值，而在于保证时间来源是严格线性递增的，因而可用于计算两个样本的时间差。

因此，单调时间适合用于计算相对时间，而壁钟时间适合用来测量绝对时间。

译注 1：所谓的“概要分析”，是指根据程序的行为表现对程序作出分析。

这三种测量时间的方法可以被表示成如下两种形式之一：

相对时间

这是一个相对于某个基准点（例如当前瞬间）的值：如 *5 seconds from now*（从现在起 5 秒后）或是 *10 minutes ago*（10 分钟前）。

绝对时间

不使用任何基准点来表示时间：如 *noon on 25 March 1968*（在 1968 年 3 月 25 日的中午）。

时间的相对及绝对形式都会被用到。一个进程可能需要在 500 毫秒取消一个请求、每秒刷新屏幕 60 次或者注意到自从一个运算开始已经过去7 秒了。以上这些都需要进行相对时间的计算。相反，一个日历应用程序会将用户的 Toga Party（译注 2）日期记录为 8 February（2 月 8 日），当一个文件被创建时，文件系统将会写出完整的日期与时间（而非 “5 秒之前”），并且用户的时钟会以格里高里历法（Gregorian date）显示时间，而不会以自系统引导后所经过的秒数。

Unix 系统将绝对时间表示成自纪元（被定义为 1970 年 1 月 1 日 00:00:00 UTC）后所经过的秒数。UTC（Universal Time Coordinated，协调世界时）大致为 GMT（Greenwich Mean Time，格林威治标准时间）或 Zulu（祖鲁）时间。说来奇怪，这意味着在 Unix 中，即使是绝对时间，本质上还是相对时间。为了存储“自纪元后的秒数”，Unix 引进了一个特殊的数据类型，下一节我们将看到。

操作系统会由软件时钟（*software clock*）—— 内核以软件所维护的时钟 —— 来追踪时间的进展。内核会初始化一个周期性定时器，就是所谓的系统定时器（*system timer*），它会以特定的频率发出中断事件。当一个定时器的时间间隔结束，内核就会替已经过去的时间递增一个单元，就是所谓的一个时钟周期（*tick* 或 *jiffy*）。用于计算已经过去了几个时钟周期的计数器称为 *jiffies counter*（时钟周期计数器）。jiffies counter 原先是一个 32 位的计数器，不过到了 2.6 版的 Linux 内核则是一个 64 位的计数器（注 1）。

在 Linux 上，系统定时器的频率称为 HZ，因为有一个代表它的预处理器定义了相同的名称。HZ 的值与架构有关，而不是 Linux ABI 的一部分 —— 也就是说，程序无法依赖或要求任何特定的值。以往，x86 架构会使用“100” 这个值，这意味系统定时器每秒运转 100 次（也就是系统定时器具有 100 hertz 的频率）。这让每个 jiffy 具有 0.01

译注 2：Toga Party 是指参见派对的人必须围上床单当衣服穿，有如古罗马人所穿的宽外袍。

注 1：Linux 内核的未来版本可能会走 tickless（无时钟周期）的路线或实现 dynamic tick（动态时钟周期），在此情况下，内核不会记录明确的 jiffy 值。取而代之的是，所有以时间为基础的运算将执行自动态初始化的定时器而非系统定时器。

(1/HZ) 秒的值。到了 2.6 版的 Linux 内核发布时，内核开发者将 HZ 的值提高为 1000，这让每个 jiffy 具有 0.001 秒的值。然而，在 2.6.13 与之后的版本中，HZ 的值变为 250，这让每个 jiffy 具有 0.004 秒的值（注 2）。这是 HZ 的值无法避免的权衡：较高的值可以提供较高的分辨率，但是定时器的开销也较大。

虽然进程不应该依赖任何固定的 HZ 值，不过 POSIX 定义了一个机制可用于在运行时确定系统定时器的频率：

```
long hz;

hz = sysconf (_SC_CLK_TCK);
if (hz == -1)
        perror ("sysconf"); /* 应该不会发生 */
```

当一个程序想确定系统定时器的分辨率时便可以使用此接口，但是它不需要将系统时间值转换成秒数，因为多数的 POSIX 接口所导出的时间测量值都已做好了转换或者已被转换成一个固定的频率，与 HZ 无关。不同于 HZ，这个固定的频率是系统 ABI 的一部分，在 x86 上值为 100。根据时钟周期返回时间值的 POSIX 函数会使用 CLOCKS_PER_SEC 来表示固定的频率。

偶尔，多个事件凑在一起会导致计算机关机；有时，计算机的电源插头甚至会被拔掉。然而，计算机开机后却能够拥有正确的时间。这是因为多数计算机有一个以电池供电的硬件时钟（*hardware clock*）用来存储计算机关机时的时间与日期。当内核启动时，它会从硬件时钟来初始化它的当前时间。同样，当用户关闭系统时，内核会将当前时间写回硬件时钟。系统的管理者可通过 *hwclock* 命令在其他的时间点上进行时间的同步。

Unix 系统上的时间管理涉及了几项任务，任何指定的进程只与其中一些任务有关：这包括了设定与取得当前的壁钟时间、计算流逝的时间、修眠指定的时间量、进行高精确度的时间测量以及控制定时器。本章将全方位说明与时间有关的任务。首先，让我们来看 Linux 会使用哪些数据结构来表示时间。

时间的数据结构

在各种 Unix 系统发展的同时，也连带为时间的管理实现了它们自己的接口，有多种数据结构用于表示看似简单的时间概念。这些数据结构的范围从简单的整数到各种多字段的结构。探讨实际的接口之前，让我们先来了解这些数据结构。

注 2： HZ 现在也是编译时内核的选项，在 x86 架构上支持 100、250 及 1000 等值。无论如何，用户空间可以不依赖于任何特定的 HZ 值。

原始的表示法

time_t 是其中最简单的数据结构，定义于头文件 <time.h> 中。time_t 是一个不透明的类型（opaque type）。然而，在多数 Unix 系统上 —— 包括 Linux —— 此类型只是 C 语言的 long 类型的 typedef：

```
typedef long time_t;
```

time_t 代表自纪元后已过去的秒数。典型的反应是“发生溢出之前它就会被废弃不用！”事实上，它的寿命可能会比你所预期的还久，尽管有大量的 Unix 系统仍在使用它，但是它的确即将溢出。由于是一个 32 位 的 long 类型，time_t 至多可以表示自纪元后过去的 2,147,483,647 秒。这意味我们将于 *2038* 年再度遭遇 Y2K 的窘境！然而，幸运的是，等到 2038-01-18 22:14:07 来临时，多数的系统与软件将会使用 64 位的时间值。

然而现在是微秒级精确度

与 time_t 有关的另一个议题是一秒之内会发生许多次。timeval 结构用于扩充 time_t，加入了微秒级精确度。此结构定义于头文件 <sys/time.h> 中，如下所示：

```
#include <sys/time.h>

struct timeval {
        time_t       tv_sec;     /* seconds */
        suseconds_t  tv_usec;    /* microseconds */
};
```

tv_sec 用于度量秒数，而 tv_usec 用于度量微秒数。令人困惑的 suseconds_t 通常是一个整数类型的 typedef。

甚至更好：纳秒级精确度

timespec 结构则提升至奈秒级分辨率。此结构定义于头文件 <time.h> 中，如下所示：

```
#include <time.h>

struct timespec {
        time_t  tv_sec;      /* seconds */
        long    tv_nsec;     /* nanoseconds */
};
```

假设可以选择，接口会更喜欢使用纳秒级（而非微秒级）分辨率（注 3）。所以，自从 timespec 结构被引进之后，多数与时间相关的接口均转而使用它，因此可以获得较高的精确度。然而，如我们将看到的，有一个重要的函数仍在使用 timeval。

实际上，这两个结构都没办法提供它们所陈述的精确度，因为系统定时器并无法提供纳秒甚至是微秒级分辨率。然而，在接口中能够支持这样的分辨率会比较好，这样不管系统提供哪种分辨率它都可以容纳。

分解时间

我们将介绍的函数中有些会在 Unix 的时间与字符串之间进行转换，或是以程序设计的方式建立字符串来表示特定的日期。为了协助此过程，C 标准提供了 tm 结构，以便使用人类可阅读的格式来表示“经分解”（broken-down）的时间。此结构也定义于 <time.h>：

```
#include <time.h>

struct tm {
        int tm_sec;           /* seconds */
        int tm_min;           /* minutes */
        int tm_hour;          /* hours */
        int tm_mday;          /* the day of the month */
        int tm_mon;           /* the month */
        int tm_year;          /* the year */
        int tm_wday;          /* the day of the week */
        int tm_yday;          /* the day in the year */
        int tm_isdst;         /* daylight savings time? */
#ifdef _BSD_SOURCE
        long tm_gmtoff;       /* time zone's offset from GMT */
        const char *tm_zone;  /* time zone abbreviation */
#endif /* _BSD_SOURCE */
};
```

tm 结构让我们更容易判断 time_t 的值，例如 314159 是星期日还是星期六（答案是前者）。以所占用的空间来看，用它来表示日期与时间显然是一个差劲的选择，但是用它来回转换面向用户的值却相当方便。

tm 结构中包含以下字段：

tm_sec

指定分钟数后已过去的秒数。此值的正常范围是 0 到 59，但是也可能出现 61 这样的值，代表多了 2 闰秒。

注 3：　此外，timespec 结构拿掉了令人困惑的 suseconds_t，转而采用了简单易懂的 long。

tm_min

指定小时数后已过去的分钟数。此值的范围是 0 到 59。

tm_hour

午夜之后已过去的小时数。此值的范围是 0 到 23。

tm_mday

一个月份中的哪一天。此值的范围是 0 到 31。POSIX 并未指定 0 这个值，然而 Linux 会使用它表示前一个月份的最后一天。

tm_mon

自 1 月起算的月份数目。此值的范围是 0 到 11。

tm_year

自 1900 年起算的年份数目。

tm_wday

自星期日起算的一周中的天数。此值的范围是 0 到 6。

tm_yday

自 1 月 1 日起算的一年中的天数。此值的范围是 0 到 365。

tm_isdst

用于指示其他字段所描述的时间是否在使用日光节约时间(daylight savings time，常简写为 DST)。如果此字段为正值，代表使用了 DST；如果此字段的值为 0，代表没有使用 DST；如果此字段为负值，代表 DST 处在未知状态。

tm_gmtoff

当前时区与格林威治标准时间的偏移量（以秒计）。引用 <time.h> 之前必须先定义 _BSD_SOURCE 才会出现此字段。

tm_zone

当前时区的缩写 —— 例如EST。引用 <time.h> 之前必须先定义 _BSD_SOURCE 才会出现此字段。

进程时间的一种类型

类型 clock_t 代表时钟周期（clock tick）。它是一个整数类型，通常是 long。根据接口的不同，clock_t 可能代表系统的实际定时器频率（HZ）或 CLOCKS_PER_SEC。

POSIX 时钟

本章所讨论的系统调用中有一些会使用 POSIX 时钟（这是一个用于实现和表示时间来源的标准）。类型 clockid_t 用于表示一种特定的 POSIX 时钟，Linux 支持其中四种：

CLOCK_MONOTONIC

一种单调递增时钟，任何进程均无法设定它。用于表示自某个未特别指定的起点（例如系统引导）之后已过去的时间。

CLOCK_PROCESS_CPUTIME_ID

一种可从处理器取得高分辨率的每个进程（per-process）的时钟。例如，在 i386 架构上，此时钟使用的是时间戳计数器（timestamp counter，常简写为 TSC）寄存器。

CLOCK_REALTIME

全系统的真实时间（壁钟时间）。设定此时钟需要特权。

CLOCK_THREAD_CPUTIME_ID

类似于每个进程的时钟，但是对一个进程中的每个线程而言是唯一的。

虽然 POSIX 定义了四种时间来源，但是它只需要 CLOCK_REALTIME。因此，尽管 Linux 能够可靠地提供四种时钟，但具可移植性的程序代码应该只使用 CLOCK_REALTIME。

时间来源的分辨率

POSIX 所定义的 clock_getres() 函数可用于取得给定时间来源的分辨率：

```
#include <time.h>

int clock_getres (clockid_t clock_id,
                  struct timespec *res);
```

执行成功时，clock_getres() 会将 clock_id 所指定时钟的分辨率存入 res，如果它不是 NULL，则会返回 0。执行失败时，此函数会返回 -1 并将 errno 设定为下面其中一个错误代码：

EFAULT

res 是一个无效的指针。

EINVAL

在此系统上，clock_id 不是一个有效的时间来源。

下面的范例程序代码会输出前一节所讨论到的四种时间来源：

```
clockid_t clocks[] = {
        CLOCK_REALTIME,
        CLOCK_MONOTONIC,
        CLOCK_PROCESS_CPUTIME_ID,
        CLOCK_THREAD_CPUTIME_ID,
        (clockid_t) -1 };
int i;

for (i = 0; clocks[i] != (clockid_t) -1; i++) {
        struct timespec res;
        int ret;

        ret = clock_getres (clocks[i], &res);
        if (ret)
                perror ("clock_getres");
        else
                printf ("clock=%d sec=%ld nsec=%ld\n",
                        clocks[i], res.tv_sec, res.tv_nsec);
}
```

在现代的 x86 系统上，输出会像下面这样：

```
clock=0 sec=0 nsec=4000250
clock=1 sec=0 nsec=4000250
clock=2 sec=0 nsec=1
clock=3 sec=0 nsec=1
```

注意，4,000,250 纳秒就是 4 毫秒，也就是 0.004 秒。反过来说，0.004 秒就是 HZ 值为 250 的 x86 系统时钟的分辨率，如同我们在本章第一节所做的讨论。因此，我们会看到 CLOCK_REALTIME 与 CLOCK_MONOTONIC 被绑定到 jiffy 以及系统定时器所提供的分辨率。相反，CLOCK_PROCESS_CPUTIME_ID 与 CLOCK_THREAD_CPUTIME_ID 所运用的是分辨率较高的时间来源 —— 在此 x86 机器上，我们看到 TSC 提供了纳秒级分辨率。

在 Linux（以及多数其他 Unix 系统）上，使用 POSIX 时钟的所有函数需要链接 *librt* 与最后的目标文件。举例来说，如果要将前面的程序代码片段编译成一个可执行的程序，你可以使用如下这道命令：

```
$ gcc -Wall -W -O2 -lrt -g -o snippet snippet.c
```

取得当前时间

应用程序会因为一些理由而想取得当前的时间与日期：显示给用户看、计算相对的或已过去的时间、一个事件的时间戳等。最简单及以往最常用来取得当前时间的方法是

time() 函数：

```
#include <time.h>

time_t time (time_t *t);
```

time() 会返回被表示成自纪元后已过去的秒数的当前时间。如果参数 t 不是 NULL，此函数还会将当前时间写入所提供的指针。

发生错误时，此函数会返回 -1（转型成 time_t），而且会适当地设定 errno。唯一可能的错误是 EFAULT，代表 t 是一个无效的指针。

例如：

```
time_t t;

printf ("current time: %ld\n", (long) time (&t));
printf ("the same value: %ld\n", (long) t);
```

一个简单的时间处理方法

time_t 的“自纪元后已过去的秒数”表示法，并不是自从重大事件发生之后实际过去的秒数。Unix 进行计算时会假设所有可以被 4 整除的年份都是闰年，但是会完全忽略闰秒。time_t 表示法的重点并不在于它是准确的，而是在于它是一致的 —— 这就是关键所在。

一个更好的接口

gettimeofday() 用于扩充 time()，它加入了微秒级分辨率：

```
#include <sys/time.h>

int gettimeofday (struct timeval *tv,
                  struct timezone *tz);
```

执行成功时，gettimeofday() 会将当前时间放入 tv 所指向的 timeval 结构并且返回 0。timezone 结构与 tz 参数已被废弃，不应该使用在 Linux 上。NULL 总是被传入 tz。

执行失败时，此调用会返回 -1 并将 errno 设定为 EFAULT，这是唯一可能发生的错误，代表 tv 或 tz 是一个无效的指针。

例如：

```
struct timeval tv;
int ret;

ret = gettimeofday (&tv, NULL);
if (ret)
        perror ("gettimeofday");
else
        printf ("seconds=%ld useconds=%ld\n",
                (long) tv.tv_sec, (long) tv.tv_usec);
```

timezone 结构之所以会被废弃是因为内核并未管理时区，*glibc* 拒绝使用 timezone 结构的 tz_dsttime 字段。下一节我们将看到时区的操作方式。

一个高级接口

POSIX 所提供的 clock_gettime() 接口可用于取得给定时间来源的时间。然而，更有用的是此函数提供了纳秒级精确度：

```
#include <time.h>

int clock_gettime (clockid_t clock_id,
                   struct timespec *ts);
```

执行成功时，此调用会返回 0，而且会将 clock_id 所指定的时间来源的当前时间存入 ts。执行失败时，此调用会返回 -1，而且会将 errno 设定为下面其中一个值：

EFAULT
: ts 是一个无效的指针。

EINVAL
: clock_id 在此系统上是一个无效的时间来源。

下面的范例程序代码可以从四种标准的时间来源取得当前时间：

```
clockid_t clocks[] = {
        CLOCK_REALTIME,
        CLOCK_MONOTONIC,
        CLOCK_PROCESS_CPUTIME_ID,
        CLOCK_THREAD_CPUTIME_ID,
        (clockid_t) -1 };
int i;

for (i = 0; clocks[i] != (clockid_t) -1; i++) {
        struct timespec ts;
        int ret;
```

```
        ret = clock_gettime (clocks[i], &ts);
        if (ret)
                perror ("clock_gettime");
        else
                printf ("clock=%d sec=%ld nsec=%ld\n",
                        clocks[i], ts.tv_sec, ts.tv_nsec);
}
```

取得进程时间

times() 系统调用可用于取得正在运行的进程与它的子进程的进程时间，以 clock tick（时钟周期）计：

```
#include <sys/times.h>

struct tms {
        clock_t tms_utime;   /* user time consumed */
        clock_t tms_stime;   /* system time consumed */
        clock_t tms_cutime;  /* user time consumed by children */
        clock_t tms_cstime;  /* system time consumed by children */
};

clock_t times (struct tms *buf);
```

执行成功时，此调用会将进行调用的进程与它的子进程所耗用的进程时间填入 buf 所指向的 tms 结构。所汇报的时间会被分解成用户时间和系统时间。用户时间（*user time*）就是在用户空间中执行程序代码所花的时间。系统时间（*system time*）就是在内核空间中执行程序代码所花的时间 —— 例如，系统调用或页面失误期间。只有在子进程终止以及父进程对进程调用 waitpid()（或一个相关函数）之后才会汇报子进程的时间。此调用会返回自从过去某一参考点之后单调递增的时钟周期数目。这个参考点曾经是系统引导时间 —— 因此times() 函数会返回 *system uptime*（以时钟周期计），但是现在这个参考点大约是系统引导之前4.29亿秒。内核开发者会实现此变更来避免内核程序代码无法处理 system uptime 绕回而变为零的情况。此函数所返回的是绝对值（而非相对值），因而毫无用处，然而两次调用之间的相对变化仍是有价值的。

执行失败时，此调用会返回 -1 并且将 errno 设定为适当的值。在 Linux 上，只有一个可能的错误代码 EFAULT，代表 buf 是一个无效的指针。

设定当前时间

前面所描述的是如何取得时间，然而应用程序偶尔也需要将当前的时间与日期设定为一个指定的值。这几乎总是可以使用专为此用途而设计的实用程序来处理，例如 *date*。

time() 的时间设定版本为 stime()：

```
#define _SVID_SOURCE
#include <time.h>

int stime (time_t *t);
```

执行成功时，stime() 会将系统时间设定为 t 所指的值并返回 0。调用此调用的用户必须具备 CAP_SYS_TIME 能力。通常只有 root 用户具有此能力。

执行失败时，此调用会返回 -1 并将 errno 设定为 EFAULT（代表 t 是一个无效的指针）或 EPERM（代表进行调用的用户并不具有 CAP_SYS_TIME 能力）。

它的用法相当简单：

```
time_t t = 1;
int ret;

/* 将时间设定为自纪元之后 1 秒 */
ret = stime (&t);
if (ret)
        perror ("stime");
```

下一节我们将看到能够把时间的人类可阅读形式转换成一个 time_t 的工作简化的函数。

以微秒级精确度设定时间

gettimeofday() 的时间设定版本为 settimeofday()：

```
#include <sys/time.h>

int settimeofday (const struct timeval *tv ,
                  const struct timezone *tz);
```

执行成功时，settimeofday() 会将系统时间设定为 tv 所提供的值并且返回 0。如同 gettimeofday()，将 NULL 传入 tz 是最好的做法。执行失败时，此调用会返回 -1 并且将 errno 设定为下面其中一个值：

EFAULT

tv 或 tz 指向了一段无效的内存区域。

EINVAL

所提供的结构之一中的一个字段是无效的。

EPERM

进行调用的进程没有 CAP_SYS_TIME 能力。

下面的范例程序代码会将当前时间设定为 1979 年 12 月中的一个星期六：

```
struct timeval tv = { .tv_sec  = 31415926,
                      .tv_usec = 27182818 };
int ret;

ret = settimeofday (&tv, NULL);
if (ret)
        perror ("settimeofday");
```

一个用于设定时间的高级接口

正如 clock_gettime() 改善了 gettimeofday()，clock_settime() 淘汰了 settimeofday()：

```
#include <time.h>

int clock_settime (clockid_t clock_id,
                   const struct timespec *ts);
```

执行成功时，此调用会返回 0 并将 clock_id 所指定的时间来源设定为 ts 所指定的时间。执行失败时，此调用会返回 -1 并将 errno 设定为下面其中一个值：

EFAULT

ts 是一个无效的指针。

EINVAL

clock_id 在此系统上是一个无效的时间来源。

EPERM

进程缺乏设定指定时间来源所需的权限，或所指定的时间来源不可以被设定。

在多数系统上，CLOCK_REALTIME 是唯一可设定的时间来源。所以此函数唯一胜过 settimeofday() 的地方就是它提供了纳秒级分辨率(而且不必处理无用的 timezone 结构)。

操作时间

Unix 系统以及 C 语言提供了一系列的函数可用于转换经分解的时间（用于表示时间的 ASCII 字符串）与 time_t。asctime() 会将一个 tm 结构 —— 经分解的时间 —— 转换为一个 ASCII 字符串：

```
#include <time.h>

char * asctime (const struct tm *tm);
char * asctime_r (const struct tm *tm, char *buf);
```

asctime() 会返回一个指向静态分配的字符串的指针。随后对任何时间函数所进行的调用可能会改写此字符串，asctime() 并不具有线程安全性。

因此，多线程程序（以及厌恶接口设计不当的开发者）应该使用 asctime_r()。此函数并不会返回一个指向静态分配的字符串的指针，它会通过 buf 提供字符串（长度至少为 26 个字符）。

发生错误时，这两个函数都会返回 NULL。

mktime() 也可以用于转换一个 tm 结构，但是转换的结果为 time_t：

```
#include <time.h>

time_t mktime (struct tm *tm);
```

mktime() 还会在指定过 tm 时经 tzset() 设定时区（time zone）。发生错误时，它会返回 -1（转型成一个 time_t）。

ctime() 可用于将一个 time_t 转换成它的 ASCII 表示法：

```
#include <time.h>

char * ctime (const time_t *timep);
char * ctime_r (const time_t *timep, char *buf);
```

执行失败时，它会返回 NULL。例如：

```
time_t t = time (NULL);

printf ("the time a mere line ago: %s", ctime (&t));
```

注意，少了换行符。或许你会觉得不方便，ctime() 会在它所返回的字符串中添加换行符。

如同 asctime()，ctime() 会返回一个指向静态字符串的指针。因为这么做并不具

有线程安全性，所以多线程程序应该改用 ctime_r()，它所操作的是 buf 提供的缓冲区。此缓冲区的长度至少为 26 个字符。

gmtime() 可用于将指定的 time_t 转换成一个 tm 结构,结果会根据UTC 时区来表示：

```
#include <time.h>

struct tm * gmtime (const time_t *timep);
struct tm * gmtime_r (const time_t *timep, struct tm *result);
```

执行失败时，它会返回 NULL。

此函数会静态分配返回的结构，因而同样不具有线程安全性。多线程程序应该使用 gmtime_r()，它操作的是 result 所指向的结构。

localtime() 与 localtime_r() 的功能分别如同 gmtime() 与 gmtime_r()，但是它们会以用户的时区来表示特定的 time_t：

```
#include <time.h>

struct tm * localtime (const time_t *timep);
struct tm * localtime_r (const time_t *timep, struct tm *result);
```

如同 mktime()，调用 localtime() 也会调用 tzset() 并初始化时区。localtime_r() 是否会进行此步骤并未特别指出。

difftime() 会返回两个 time_t 值（会被转型成 double）之间已过去的秒数：

```
#include <time.h>

double difftime (time_t time1, time_t time0);
```

在所有的 POSIX 系统上，time_t 是一个算术类型，而且 difftime() 等效于下面这道忽略溢出检测的减法运算：

```
(double) (time1 - time0)
```

在 Linux 上，因为 time_t 是一个整数类型，所以不需要转型成 double。然而，如果考虑到可移植性，则应该使用 difftime()。

调整系统时钟

壁钟时间中巨大而突然的跳动(jump)会对那些运算依赖绝对时间的应用程序造成大混乱。考虑一个 *make* 的例子，它会根据 *Makefile* 的描述构建软件项目。每次调用 *make* 并不会重新构建整个源码树；否则，在大型的软件项目中，即使只变更了一个文件，结

果也可能要进行好几个小时的重新构建。事实上，*make* 会查看源码文件（例如 *wolf.c*）与目标文件最近被写入的时间（文件的时间戳）。如果源码文件 —— 或者它的任何必要条件（prerequisite），例如 *wolf.h* —— 比目标文件还新，*make* 便会重新构建该源码文件，以更新目标文件。然而，如果源码文件没有比目标文件新，则不会采取任何行动。

因此，如果用户发现他的时钟偏移了两三个小时，于是执行 *date* 来更新系统时钟，想想看会发生什么事。如果用户接着更新与重新保存 *wolf.c*，则可能会有问题。如果用户将当前时间往回调，*wolf.c* 将比 *wolf.o* 还旧 —— 即使事实并非如此，因而不会有重新构建的事情发生。

为了避免此类问题，Unix 提供了 adjtime() 函数，可用于朝着所指定的增量（delta）逐步调整当前时间。这主要是由 Network Time Protcol（NTP）之类的监控程序所进行的后台活动，以便不断地以正确的时钟偏移来调整时间，使用 adjtime() 将可最小化它们对系统所造成的影响。

```
#define _BSD_SOURCE
#include <sys/time.h>

int adjtime (const struct timeval *delta,
             struct timeval *olddelta);
```

一个成功的 adjtime() 调用会指示内核根据 delta 慢慢开始调整时间并且返回 0。如果 delta 所指定的时间是正值，则内核会以 delta 加快系统时钟直到完成修正；如果 delta 所指定的时间是负值，则内核会让系统时钟减慢直到完成修正。内核会应用所有的调整，让时钟总是呈现单调递增，而且绝对不会造成时间的突然改变。即使是负值的 delta，所进行的调整也不会将时钟的时间往回调；事实上，时钟会变慢，直到系统时间与所要修正的时间趋于一致。

如果 delta 的值不是 NULL，则内核会停止处理任何先前已注册的修正。然而，已进行的修正（如果有）只有部分会被维护。如果 olddelta 的值不是 NULL，则任何先前已注册但是尚未应用的修正会被写入所提供的 timeval 结构。将 NULL 传入 delta 而且 olddelta 是有效的，可以取得任何正在进行的修正。

adjtime() 所进行的应该是小量的修正 —— 理想的情况是使用 NTP，如稍早所述，其所进行的就是小量的修正（几秒钟）。Linux 维护了几千秒的最低与最高的修正阈值（threshold）。

发生错误时，adjtime() 会返回 -1 并将 errno 设定为下面其中一个值：

EFAULT

delta 或 olddelta 是一个无效的指针。

EINVAL

delta 所描述的调整太大或太小。

EPERM

进行调用的用户并不具有 CAP_SYS_TIME 的能力。

相较于 adjtime() 所采用的逐步修正做法，RFC 1305 所定义的是一个更强而有力，也相对更复杂的时钟调整算法。Linux 将这个算法实现成 adjtimex() 系统调用：

```
#include <sys/timex.h>

int adjtimex (struct timex *adj);
```

adjtimex() 可将与时间有关的内核参数读进 adj 所指向的 timex 结构。取决于此结构的 modes 字段，此系统调用可能会额外设定一些参数。

timex 结构定义于头文件 <sys/timex.h> 中，如下所示：

```
struct timex {
        int modes;              /* mode selector */
        long offset;            /* time offset (usec) */
        long freq;              /* frequency offset (scaled ppm) */
        long maxerror;          /* maximum error (usec) */
        long esterror;          /* estimated error (usec) */
        int status;             /* clock status */
        long constant;          /* PLL time constant */
        long precision;         /* clock precision (usec) */
        long tolerance;         /* clock frequency tolerance (ppm) */
        struct timeval time;    /* current time */
        long tick;              /* usecs between clock ticks */
};
```

modes 字段的值是下面一个或多个标志的按位 OR 运算：

ADJ_OFFSET

由 offset 设定时间偏移值。

ADJ_FREQUENCY

由 freq 设定频率偏移值。

ADJ_MAXERROR

由 maxerror 设定最大误差。

ADJ_ESTERROR

由 esterror 设定所估计的误差。

ADJ_STATUS

由 status 设定时钟状态。

ADJ_TIMECONST

由 constant 设定锁相环（phase-locked loop，常简写为 PLL）的时间常量。

ADJ_TICK

由 tick 设定时钟周期。

ADJ_OFFSET_SINGLESHOT

只由 offset 设定一次时间偏移值，使用简单的算法，如同 adjtime()。

如果 modes 的值为 0，不会设定任何值。只有具备 CAP_SYS_TIME 能力的用户可以提供一个非零的 modes 值；任何用户均可以将 0 提供给 modes，以便取得所有参数，但是不会对它们进行设定。

执行成功时，adjtimex() 会返回当前时钟的状态，可以是下面其中一个值：

TIME_OK

时钟已被同步。

TIME_INS

闰秒将被插入。

TIME_DEL

闰秒将被删除。

TIME_OOP

闰秒正在进行中。

TIME_WAIT

闰秒刚发生。

TIME_BAD

时钟未被同步。

执行失败时，adjtimex() 会返回 -1 并将 errno 设定为下面其中一个错误码：

EFAULT

adj 是一个无效的指针。

EINVAL

modes、offset 或 tick 中有一个或多个是无效的。

EPERM

modes 是非零值，但是进行调用的用户并不具有 CAP_SYS_TIME 能力。

adjtimex() 是 Linux 特有的系统调用。考虑到可移植性的应用程序应该使用 adjtime()。

RFC 1305定义的是一个复杂的算法，所以 adjtimex() 的完整说明已超过了本书的范畴。进一步的信息请参见 RFC。

休眠与等待

有些函数可让一个进程休眠（暂停执行）指定的时间量。此类函数中第一个要介绍的是 sleep()，它会让进行调用的进程休眠 seconds 所指定的秒数：

```
#include <unistd.h>

unsigned int sleep (unsigned int seconds);
```

此调用会返回尚未睡完的秒数。因此，一次成功的调用会返回 0，但是函数可能会返回介于 0 与 seconds（含）的其他值（如果休眠的时候遇到一个信号中断）。此函数不会设定 errno。sleep() 的多数用户并不在乎进程实际睡了多长的时间，因而不会检查返回值：

```
sleep (7);          /* 休眠7秒的时间 */
```

如果睡完指定的时间真的很重要，你可以使用它的返回值持续调用 sleep()，直到它返回 0：

```
unsigned int s = 5;

/* 休眠5秒的时间：没有任何借口 */
while ((s = sleep (s)))
```

以微秒级精确度进行休眠

仅以秒的粒度进行休眠毫无用处。1秒在现代的系统上是一个漫长的时间，所以程序通常会想以亚秒级分辨率进行休眠。本节要介绍的是 usleep()：

```
/* BSD version */
#include <unistd.h>

void usleep (unsigned long usec);

/* SUSv2 version */
#define _XOPEN_SOURCE 500
#include <unistd.h>

int usleep (useconds_t usec);
```

一次成功的 usleep() 调用可让进行调用的进程休眠 usec 微秒。不幸的是，BSD 与

Single UNIX Specification（SUS）对此函数的原型有歧义。此函数的 BSD 版本具有一个类型为 unsigned long 的参数，而且没有返回值。然而，此函数的 SUS 版本却具有一个类型为 useconds_t 的参数，而且会返回一个 int 值。如果 _XOPEN_SOURCE 被定义成 500 或更高的值，Linux 采用 SUS 的做法；如果 _XOPEN_SOURCE 未定义或者被设定成小于 500 的值，则 Linux 会采用 BSD 的做法。

执行成功时，SUS 版本会返回 0；发生错误时，则会返回 -1。有效的 errno 值是 EINTR（如果休眠的时候遇到一个信号中断）或是 EINVAL（如果 usecs 太大。在 Linux 上，此类型的整个范围都是有效的，因此绝不会发生此错误）。

按规定，useconds_t 类型是一个无符号整数，可以保存高达 1,000,000 的值。

由于这两种不兼容原型之间的差异以及有些 Unix 系统仅支持其中一种原型的事实，所以聪明的做法就是不要显式地在你的程序代码中引用 useconds_t 类型。要达到最大的可移植性，应该假设参数是一个 unsigned int，而不要依赖 usleep() 的返回值：

```
void usleep (unsigned int usec);
```

它的用法如下所示：

```
unsigned int usecs = 200;

usleep (usecs);
```

这适用于此函数的任何一种变体，而且仍然可以进行错误检查：

```
errno = 0;
usleep (1000);
if (errno)
        perror ("usleep");
```

然而，多数程序并不检查或关心 usleep() 的错误。

以纳秒级精确度进行休眠

Linux 不赞成使用 usleep() 函数，所以使用 nanosleep()（提供纳秒级分辨率以及功能更多的接口）来取代它：

```
#define _POSIX_C_SOURCE 199309
#include <time.h>

int nanosleep (const struct timespec *req,
               struct timespec *rem);
```

一次成功的 nanosleep() 调用可让进行调用的进程修眠 req 所指定的时间，然后返回 0。发生错误时，此调用会返回 -1 并且对 errno 进行适当的设定。如果有一个信号让休眠被中断，此调用会在指定的时间过去之前先行返回。在此情况下，nanosleep() 会返回 -1 并且将 errno 设定为 EINTR。如果 rem 的值不是 NULL，则此函数会将剩下的休眠时间（req 所指定的时间尚未睡完的部分）放入 rem。然后程序可以重新发出此调用，将 rem 传入 req（如稍后所示）。

下面是其他可能的 errno 值：

EFAULT

　　req 或 rem 是一个无效的指针。

EINVAL

　　req 的字段中有一个是无效的。

基本情况下，它的用法很简单：

```
struct timespec req = { .tv_sec = 0,
                        .tv_nsec = 200 };

/* 休眠 200 ns */
ret = nanosleep (&req, NULL);
if (ret)
        perror ("nanosleep");
```

下面的范例用到了第二个参数，如果被中断，可以继续休眠：

```
struct timespec req = { .tv_sec = 0,
                        .tv_nsec = 1369 };
struct timespec rem;
int ret;

/* 休眠 1369 ns */
retry:
ret = nanosleep (&req, &rem);
if (ret) {
        if (errno == EINTR) {
                /* 以所提供的剩余时间再试一次 */
                req.tv_sec = rem.tv_sec;
                req.tv_nsec = rem.tv_nsec;
                goto retry;
        }
        perror ("nanosleep");
}
```

下面是完成相同目标的另一种做法（或许更有效率，但是较不具可读性）：

```
struct timespec req = { .tv_sec = 1,
```

```
                    .tv_nsec = 0 };
struct timespec rem, *a = &req, *b = &rem;

/* 休眠 1秒 */
while (nanosleep (a, b) && errno == EINTR) {
        struct timespec *tmp = a;
        a = b;
        b = tmp;
}
```

相较于 sleep() 与 usleep()，nanosleep() 具有以下优点：

- 具有纳秒（相对于秒或微秒）级分辨率。
- 经 POSIX.1b 标准化。
- 并未通过信号实现（稍后会讨论此陷阱）。

尽管不赞成使用，许多程序仍偏好使用 usleep() 而非 nanosleep() —— 值得庆幸的是，至少现在使用 sleep() 的应用程序越来越少了。因为 nanosleep() 是POSIX 标准，而且不会使用信号，所以相较于 sleep() 与 usleep()，新的程序应该偏好使用它（或者下一节所要探讨的接口）。

以一个高级的方式进行休眠

如同我们至今探讨过的所有时间函数，POSIX 的 clock 系列函数提供了最高级的休眠接口：

```
#include <time.h>

int clock_nanosleep (clockid_t clock_id,
                     int flags,
                     const struct timespec *req,
                     struct timespec *rem);
```

clock_nanosleep() 的行为类似于 nanosleep()。事实上，此调用：

```
ret = nanosleep (&req, &rem);
```

如同此调用：

```
ret = clock_nanosleep (CLOCK_REALTIME, 0, &req, &rem);
```

差别在于 clock_id 与 flags 参数。前者用于指定要测量的时间来源。多数的时间来源都是有效的，但是你无法指定进行调用的进程的CPU 时钟（例如 CLOCK_PROCESS_CPUTIME_ID）。这么做是没有意义的，因为此调用会暂停进程的执行，因此进程时间停止增加。

时间来源的指定取决于你的程序休眠的目的。如果你要休眠某个绝对时间值，则 CLOCK_REALTIME 最有意义；如果你要休眠相对的时间量，则 CLOCK_MONOTONIC 绝对是理想的时间来源。

flags 参数可以是 TIMER_ABSTIME 或 0。如果它是 TIMER_ABSTIME，则 req 所指定的值会被视为绝对值而非相对值。这解决了一个潜在的竞争条件。为了说明此参数，让我们假设有一个进程位于时间 T_0，想休眠到时间 T_1。进程在 T_0 调用 clock_gettime()，取得了当前时间（T_0）。接着从 T_1 减去 T_0，取得了 Y（会被传入 clock_nanosleep()）。然而，取得时间的时刻与进程去休眠的时刻之间已经过去了某个时间量。更糟的是，如果进程被安排离开处理器、遇到了页面失误或者其他类似的事情呢？在取得当前时间、计算时间差以及实际进入休眠期间往往会存在着一个潜在的竞争条件。

TIMER_ABSTIME 标志可以通过让进程直接指定 T_1 来消除竞争条件。内核会暂停该进程，直到指定的时间来源到达 T_1。如果指定的时间来源的当前时间已经超过 T_1，则此调用会立即返回。

让我们来看看相对与绝对休眠的差异。下面的例子会休眠 1.5 秒：

```
struct timespec ts = { .tv_sec = 1, .tv_nsec = 500000000 };
int ret;

ret = clock_nanosleep (CLOCK_MONOTONIC, 0, &ts, NULL);
if (ret)
        perror ("clock_nanosleep");
```

相对而言，下面的例子会休眠绝对的时间值——确切的1秒，来自 clock_gettime() 调用针对 CLOCK_MONOTONIC 时间来源的返回值：

```
struct timespec ts;
int ret;

/* 从现在开始我们想要休眠 1 秒的时间 */
ret = clock_gettime (CLOCK_MONOTONIC, &ts);
if (ret) {
        perror ("clock_gettime");
        return;
}

ts.tv_sec += 1;
printf ("We want to sleep until sec=%ld nsec=%ld\n",
        ts.tv_sec, ts.tv_nsec);
ret = clock_nanosleep (CLOCK_MONOTONIC, TIMER_ABSTIME,
                       &ts, NULL);
if (ret)
        perror ("clock_nanosleep");
```

多数程序只需要进行相对的休眠，因为它们的休眠时间并不需要非常精确。然而，有些实时进程有非常确切的时序要求，而且需要进行绝对的休眠来避免可能的竞争条件。

以可移植的方式进行休眠

现在召回我们在第二章的好朋友 select()：

```
#include <sys/select.h>

int select (int n,
            fd_set *readfds,
            fd_set *writefds,
            fd_set *exceptfds,
            struct timeval *timeout);
```

如第二章所述，以亚秒级分辨率进行休眠时，若考虑到可移植性，可以使用 select()。长久以来，可移植的 Unix 程序都一直在使用 sleep() 来提出它们需要的休眠时间：usleep() 并非随处可用，而且 nanosleep() 尚未被写出来。开发者发现，让进程进入休眠状态时，若将 0 传入 select() 的 n，将 NULL 传入三个 fd_set 指针，以及将所需要的休眠时间传入 timeout，就是一个可移植而且有效的做法。

```
struct timeval tv = { .tv_sec = 0,
                      .tv_usec = 757 };

/* sleep for 757 us */
select (0, NULL, NULL, NULL, &tv);
```

如果考虑要移植到较旧的 Unix 系统，则 select() 可能不是你的最佳选择。

超限运行

本节所探讨的所有接口保证至少可以休眠所请求的时间(否则会返回错误代码以指出此实事)。所请求的延迟时间没有过去，这些接口绝不会成功返回。然而，间隔时间比所请求的延迟时间还长也是有可能的。

此现象可能是因为简单的调度行为 —— 所请求的时间可能已经过去，而且内核可能已经准时唤醒进程，但是调度程序可能已经选择运行不同的任务。

然而，还存在一个更诡异的起因：定时器超限运行（*timer overrun*）。当定时器的粒度比所请求的时间间隔还大时便会发生此现象。举例来说，假设系统定时器的时钟周期具有 10 ms 的时间间隔，而且有一个进程请求了 1 ms 的休眠时间。系统只能够在 10 ms 的时间间隔（interval）测量时间并响应时间相关事件（例如唤醒一个休眠中的进程）。如果进程发出休眠请求时，定时器在距离一个时钟周期届满前 1 ms 的地方，则一切相

安无事 —— 在 1 ms 中，所请求的时间（1 ms）将过去，而且内核将唤醒进程。然而，如果进程送出休眠请求的同时，定时器刚好届满一个周期，进程将休眠额外的 9 ms！也就是将有 9 次的 1 ms 超限运行。平均而言，一个周期为 *X* 的定时器将具有 *X/2* 的超限运行率。

使用高精确度的时间来源，例如 POSIX 时钟所提供的那些，以及较高值的 `HZ`，可最小化定时器的超限运转。

休眠的替代方案

如果可能，你应该避免休眠。通常你无法避免，那也没关系 —— 特别是，如果你的程序代码的修眠时间小于 1 秒。然而，若程序代码以休眠来忙碌等待（busy-wait）事件，通常是差劲的设计。在一个文件描述符上阻挡程序代码，让内核处理休眠与唤醒进程的事宜是比较好的做法。进程在一个循环中自旋（spinning）直到事件抵达的做法可以替换成内核阻挡进程的执行，只在有需要时才唤醒它。

定时器

当指定的时间量过去后，定时器提供了一个机制用于通知进程。一个定时器到期之前的时间量称为延迟时间（*delay*）或者到期时间（*expiration*）。内核如何通知进程定时器已到期取决于定时器的类型。Linux 内核提供了若干种类型的定时器，稍后我们将一一探究。

定时器之所以有用，基于若干理由。例子包括每秒刷新屏幕 60 次，以及如果 500 毫秒之后仍然持续下去就取消未决的事务。

简单的警报

`alarm()` 是最简单的定时器接口：

```
#include <unistd.h>

unsigned int alarm (unsigned int seconds);
```

调用此函数，我们可以安排在真实时间已过去 `seconds` 秒后，将一个 `SIGALRM` 信号递送给进行调用的进程。如果有一个之前所安排的信号悬而未决，则此调用会取消该警报，将它替换成刚才所请求的警报并返回之前的警报所剩下的秒数。如果 `seconds` 的值为 `0`，则之前的警报（如果有）会被取消，但是不会安排新的警报。

要成功使用此函数还需要替 `SIGALRM` 信号注册一个信号处理程序（信号与信号处理程

序可参考前一章所做的说明）。下面的程序代码片段会注册一个 SIGALRM 处理程序 alarm_handler() 以及替 5 秒设定一个警报：

```
void alarm_handler (int signum)
{
        printf ("Five seconds passed!\n");
}

void func (void)
{
        signal (SIGALRM, alarm_handler);
        alarm (5);

        pause ();
}
```

时间间隔定时器

interval timer（时间间隔定时器）系统调用自从被 POSIX 标准化后，首次出现于 4.2BSD，能够提供比 alarm() 还多的控制：

```
#include <sys/time.h>

int getitimer (int which,
               struct itimerval *value);

int setitimer (int which,
               const struct itimerval *value,
               struct itimerval *ovalue);
```

interval timer 的运作如同 alarm()，但是可以选择自动重新启动它们，而且可运作在下面其中一种模式之中：

ITIMER_REAL
: 测量真实时间。当所指定的真实时间值已过去，内核会送出一个 SIGALRM 信号给进程。

ITIMER_VIRTUAL
: 只会在进程的用户空间程序代码执行时递减其值。当所指定的进程时间过去后，内核会送出一个 SIGVTALRM 信号给进程。

ITIMER_PROF
: 会在进程执行时以及内核替进程服务时（例如进行系统调用）递减其值。当所指定的时间量过去后，内核会送出一个 SIGPROF 信号给进程。此模式常会与 ITIMER_VIRTUAL 结合，让程序可以测量进程所耗用的用户时间与内核时间。

ITIMER_REAL 所测量的时间如同 alarm()，另两种模式则可用于概要分析(profiling)。

itimerval 结构允许用户指定时间量直到定时器到期为止以及指定到期时间，这让你能够以指定的到期时间重新启动定时器：

```
struct itimerval {
        struct timeval it_interval;  /* next value */
        struct timeval it_value;     /* current value */
};
```

如稍早所述，timeval 结构可以提供微秒级分辨率：

```
struct timeval {
        long tv_sec;   /* seconds */
        long tv_usec;  /* microseconds */
};
```

setitimer() 会使用 it_value 所指定的到期时间来启动一个 which 类型的定时器。一旦 it_value 所指定的时间过去后，内核会使用 it_interval 所提供的时间重新启动该定时器。因此，it_value 是当前定时器上剩下的时间。一旦 it_value 的值为零时，它会被设定为 it_interval。如果定时器到期，而且 it_interval 的值为 0，则不会重新启动该定时器。同样地，如果一个活动中的定时器的 it_value 被设为 0，则定时器会停止运行，而且不会被重新启动。

如果 ovalue 的值不是 NULL，则 which 类型的时间间隔定时器先前的值会被返回。

getitimer() 会返回 which 类型的时间间隔定时器当前的值。

执行成功时，这两个调用都会返回 0；发生错误时，则会返回 -1，在此情况下，errno 会被设定为下面其中一个值：

EFAULT

value 或 ovalue 是一个无效的指针。

EINVAL

which 不是一个有效的时间间隔定时器类型。

下面的程序代码片段会创建一个 SIGALRM 信号处理程序（同样参见第九章），接着会以 5 秒的初始到期时间来启动一个时间间隔定时器，随之而来的是 1 秒的时间间隔：

```
void alarm_handler (int signo)
{
        printf ("Timer hit!\n");
}

void foo (void) {
```

```
        struct itimerval delay;
        int ret;

        signal (SIGALRM, alarm_handler);

        delay.it_value.tv_sec = 5;
        delay.it_value.tv_usec = 0;
        delay.it_interval.tv_sec = 1;
        delay.it_interval.tv_usec = 0;
        ret = setitimer (ITIMER_REAL, &delay, NULL);
        if (ret) {
                perror ("setitimer");
                return;
        }

        pause ();
}
```

有些 Unix 系统会通过 `SIGALRM` 来实现 `sleep()` 与 `usleep()` —— 显然，`alarm()` 与 `setitimer()` 会使用 `SIGALRM`。因此，程序设计者必须小心不要重复调用这些函数，否则结果是未定义的。如果只是短暂等待，程序设计者应该使用 `nanosleep()`，按照 POSIX 的规定将不会使用信号。如果使用定时器，程序设计者应该使用 `setitimer()` 或 `alarm()`。

高级定时器

最强大的定时器接口来自 POSIX 时钟系列。

如果使用的是 POSIX 基于时钟的定时器，则创建、初始化以及删除一个定时器的行动被分为三个不同的函数：`timer_create()`（创建定时器）、`timer_settime()`（初始化定时器）以及 `timer_delete()`（销毁它）。

> POSIX 时钟系列的定时器接口无疑是最高级的接口，但也是最新（因此最不具可移植性）而且用法最复杂的接口。如果简单性或可移植性是主要目标，则 `setitimer()` 很有可能是更好的选择。

创建一个定时器

`timer_create()` 可用于创建一个定时器：

```
#include <signal.h>
#include <time.h>

int timer_create (clockid_t clockid,
                  struct sigevent *evp,
                  timer_t *timerid);
```

一次成功的 timer_create() 调用会创建一个与POSIX 时钟 clockid 相关联的新定时器，将独一无二的定时器标识符存入 timerid 并且返回 0。此调用只会替定时器的运行设置环境，实际上任何事都不会发生，直到定时器被启动，如下一节所示。

下面的范例会创建一个新的定时器 CLOCK_PROCESS_CPUTIME_ID 的POSIX 时钟，以将定时器的标识符存入 timer：

```
timer_t timer;
int ret;

ret = timer_create (CLOCK_PROCESS_CPUTIME_ID,
                    NULL,
                    &timer);
if (ret)
        perror ("timer_create");
```

执行失败时，此调用会返回 -1，不定义 timerid，而且此调用会将 errno 设定为下面其中一个值：

EAGAIN

系统的资源不足以完成此请求。

EINVAL

clockid 指定了无效的POSIX 时钟。

ENOTSUP

虽然 clockid 指定了有效的POSIX 时钟，但是系统并不支持使用该时钟的定时器。POSIX 保证所有实现均支持使用 CLOCK_REALTIME 时钟的定时器。至于是否支持其他时钟，则由实现自行决定。

evp 参数，如果非 NULL 值，用于定义当定时器到期时所发生的异步通知。此结构定义于头文件 <signal.h> 中。它的内容对程序设计者而言应该是不透明的，但是它至少具有以下字段：

```
#include <signal.h>

struct sigevent {
        union sigval sigev_value;
        int sigev_signo;
        int sigev_notify;
        void (*sigev_notify_function)(union sigval);
        pthread_attr_t *sigev_notify_attributes;
};

union sigval {
        int sival_int;
        void *sival_ptr;
};
```

当一个定时器到期时，相较于内核通知进程的方式，POSIX 基于时钟的定时器可以提供更多的控制权，它允许进程指定内核将送出何种信号，甚至允许内核派生一个线程，并且执行一个函数以响应定时器到期的事实。一个进程可通过 sigev_notify 指定当定时器到期的行为，它必须是下面其中一个值：

SIGEV_NONE

一个空的通知。当定时器到期，不会有任何事发生。

SIGEV_SIGNAL

当定时器到期，内核会将 sigev_signo 所指定的信号传送给进程。在信号处理程序中，si_value 会被设定为 sigev_value。

SIGEV_THREAD

当定时器到期，内核会（在此进程内）派生一个新的线程，并且让它执行 sigev_notify_function，传入 sigev_value 作为唯的一参数。从此函数返回后，线程会终止。如果 sigev_notify_attributes 的值不是 NULL，则所提供的 pthread_attr_t 结构用于定义新线程的行为。

如果 evp 的值是 NULL（就像我们前面的例子），则所设置的定时器到期通知宛如 sigev_notify 的值是 SIGEV_SIGNAL、sigev_signo 的值是 SIGALRM 以及 sigev_value 的值是定时器的标识符。因此，在默认情况下，这些定时器的通知方式类似于 POSIX 的时间间隔定时器。然而，通过自定义，它们可以做的事还有很多、很多！

下面的例子会创建一个基于 CLOCK_REALTIME 的定时器。当定时器到期时，内核将送出 SIGUSR1 信号并把 si_value 设定成存储定时器标识符的地址：

```
struct sigevent evp;
timer_t timer;
int ret;

evp.sigev_value.sival_ptr = &timer;
evp.sigev_notify = SIGEV_SIGNAL;
evp.sigev_signo = SIGUSR1;
ret = timer_create (CLOCK_REALTIME,
                    &evp,
                    &timer);
if (ret)
        perror ("timer_create");
```

启动一个定时器

timer_create() 所创建的定时器并未启动。要将它关联到一个到期时间以及启动时钟周期，可以使用 timer_settime()：

```
#include <time.h>

int timer_settime (timer_t timerid,
                   int flags,
                   const struct itimerspec *value,
                   struct itimerspec *ovalue);
```

一次成功的 `timer_settime()` 调用会使用到期时间 `value`(这是一个 `itimerspec` 结构）启动 `timerid` 所指定的定时器：

```
struct itimerspec {
        struct timespec it_interval;  /* next value */
        struct timespec it_value;     /* current value */
};
```

如同 `setitimer()`，`it_value` 用于指定当前的定时器到期时间。当定时器到期，`it_value` 的值会被更新成 `it_interval` 的值。如果 `it_interval` 的值为 0，则定时器不是一个时间间隔定时器，一旦 `it_value` 到期就会回到未启动状态。

如稍早所示，`timespec` 结构提供了纳秒级分辨率：

```
struct timespec {
        time_t  tv_sec;       /* seconds */
        long    tv_nsec;      /* nanoseconds */
};
```

如果 `flags` 的值为 `TIMER_ABSTIME`，则 `value` 所指定的时间值会被解读成绝对值（此值默认的解读方式为相对于当前的时间）。这个经修改的行为可避免取得当前时间、计算“该时间”与“所期望的未来时间”的相对差额以及启动定时器期间造成竞争条件。细节可参考 “以一个高级的方式进行休眠”一节。

如果 `ovalue` 的值不是 `NULL`，则之前的定时器到期时间会被存入其所提供的 `itimerspec`。如果定时器之前处在未启动状态，则此结构的成员全都会被设定为 0。

使用稍早由 `timer_create()` 初始化的 `timer` 值，下面的例子可以创建一个每秒到期一次的周期性（periodic）定时器：

```
struct itimerspec ts;
int ret;

ts.it_interval.tv_sec = 1;
ts.it_interval.tv_nsec = 0;
ts.it_value.tv_sec = 1;
ts.it_value.tv_nsec = 0;

ret = timer_settime (timer, 0, &ts, NULL);
if (ret)
        perror ("timer_settime");
```

取得一个定时器的到期时间

你可以在任何时刻由 timer_gettime() 取得一个定时器的到期时间而不会重置它：

```
#include <time.h>

int timer_gettime (timer_t timerid,
                   struct itimerspec *value);
```

一次成功的 timer_gettime() 调用会将 timerid 所指定的定时器的到期时间存入 value 所指定的结构并且返回 0。执行失败时，此调用会返回 -1 并且将 errno 设定为下面其中一个值：

EFAULT

value 是一个无效的指针。

EINVAL

timerid 是一个无效的定时器。

例如：

```
struct itimerspec ts;
int ret;

ret = timer_gettime (timer, &ts);
if (ret)
        perror ("timer_gettime");
else {
        printf ("current sec=%ld nsec=%ld\n",
                ts.it_value.tv_sec, ts.it_value.tv_nsec);
        printf ("next sec=%ld nsec=%ld\n",
                ts.it_interval.tv_sec, ts.it_interval.tv_nsec);
}
```

取得一个定时器的超限运行次数

POSIX 定义了一个接口用于确定指定定时器上发生超限运行的次数（如果有）：

```
#include <time.h>

int timer_getoverrun (timer_t timerid);
```

执行成功时，timer_getoverrun() 会返回定时器初次到期与通知进程（例如通过信号）定时器已到期之间额外发生的定时器到期次数。举例来说，在我们之前的例子中，一个 1 ms 的定时器运行了 10 ms，则此调用会返回 9。

如果超限运行的次数等于或大于 DELAYTIMER_MAX，则此调用会返回 DELAYTIMER_MAX。

执行失败时，此函数会返回 -1 并将 errno 设定为 EINVAL，这个唯一的错误情况代表 timerid 指定了无效的定时器。

例如：

```
int ret;

ret = timer_getoverrun (timer);
if (ret == -1)
        perror ("timer_getoverrun");
else if (ret == 0)
        printf ("no overrun\n");
else
        printf ("%d overrun(s)\n", ret);
```

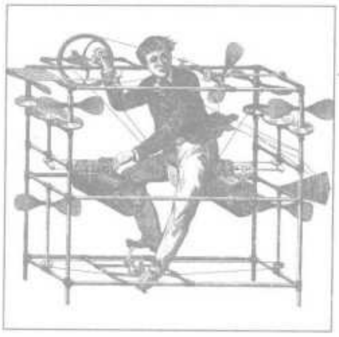
删除一个定时器

要删除一个定时器很容易：

```
#include <time.h>

int timer_delete (timer_t timerid);
```

一次成功的 timer_delete() 调用会销毁关联到 timerid 的定时器并且返回 0。执行失败时，此调用会返回 -1 并将 errno 设定为 EINVAL，这个唯一的错误情况代表 timerid 不是一个有效的定时器。



附录

GCC对C语言的扩展

GNU Compiler Collection（常简写为GCC）对C语言做了许多扩展，其中有一些已被证实对系统程序设计者具有特殊的价值。本附录将介绍的C语言扩展主要是让程序设计者可以把关于程序代码的行为与用法的额外信息提供给编译器。编译器会利用此信息产生更有效的机器码。其他的扩展用于填补C程序语言的缺口，特别是较低级的部分。

GCC所提供的若干扩展现在已经存在于最新的C标准ISO C99中。有些扩展功能与它们的C99表亲相似，但是ISO C99所实现的其他扩展则颇为不同。新的程序代码应该使用这些功能的ISO C99变体版本。本附录不会说明此类扩展，我们只会探讨GCC特有的扩展。

GNU C

GCC所支持的C语言称为GNU C。20世纪90年代，GNU C填补了C语言若干的缺口，提供了复杂变量（complex variable）、零长度数组（zero-length array）、内嵌（inline）函数以及命名初始器（named initializer）。但经过了将近十年，C语言终于升级，并进行ISO C99的标准化，GNU C的扩展变得无关紧要。尽管如此，GNU C仍继续提供有用的功能，而且许多Linux程序设计者仍在他们与C90或C99兼容的程序代码中使用GNU C的扩展。

在完全以GNU C写成的程序中，Linux内核是广为人知的一个例子。然而，最近Intel已投入工程方面的努力，让Intel C Compiler（ICC）能够了解内核所用到GNU C扩展。因此，现在越来越少采用GCC特有的扩展了。

内嵌函数

编译器会把一个“内嵌”(inline) 函数的所有程序代码复制到该函数被调用的位置上。所以不是将函数存储在外部并在它每次被调用时跳到它那里运行,而是直接运行函数的内容。这样的行为可以省去函数调用的开销,而且可以在调用的位置上进行可能的优化,因为编译器可以同时优化调用者 (caller) 与被调用者 (callee)。如果在调用的位置上传递给函数的参数为常量,则后者特别有效。当然,将一个函数复制到每个调用它的位置上会对程序代码的大小造成不利的影响。因此,如果函数既小又简单或者并未在许多不同的位置上被调用,则可以采取内嵌的方式。

GCC 已经支持 `inline` 关键字许多年,此关键字用于指示编译器嵌入指定的函数。后来 C99 才将此关键字标准化:

```
static inline int foo (void) { /* ... */ }
```

然而,就技术而言,此关键字是一个提示 —— 建议编译器嵌入指定的函数。GCC 还提供了一个扩展,用于指示编译器务必嵌入指定的函数:

```
static inline __attribute__ ((always_inline)) int foo (void) { /* ... */ }
```

程序中最可能使用内嵌函数的地方就是预处理器宏 (preprocessor macro)。在 GCC 中,内嵌函数的执行和宏一样,不过还会进行数据类型的检查。举例来说,如下的宏:

```
#define max(a,b) ({ a > b ? a : b; })
```

可以改用相应的内嵌函数:

```
static inline max (int a, int b)
{
        if (a > b)
                return a;
        return b;
}
```

程序设计者往往会过度使用内嵌函数。函数调用的开销在最现代的架构上 —— 尤其是 x86 —— 非常非常的低。只有最关键的函数才应该考虑此问题!

禁止内嵌

在最积极的优化模式中,GCC 会自动选择看来适合嵌入的函数并且嵌入它们。这通常是一个好主意,但有时程序设计者知道,某个函数如果进行内嵌会有执行错误的情况。一个可能的例子就是使用 `__builtin_return_address` (本附录稍后会说明) 的时候。要禁止内嵌,可以使用 `noinline` 关键字:

```
__attribute__ ((noinline)) int foo (void) { /* ... */ }
```

纯函数

“纯”(pure) 函数就是一个无任何影响的函数，它的返回值仅反映了函数的参数或非易失性的 (nonvolatile) 全局变量。任何参数或全局变量的访问权限必须是仅供读取。循环的优化以及子表达式的消除则可以应用至此类函数。要标示纯函数可使用 pure 关键字：

```
__attribute__ ((pure)) int foo (int val) { /* ... */ }
```

strlen() 是一个常见的例子。输入相同时，即使多次调用此函数，其返回值不变，因此可以放在循环外，而且调用一次就行了。例如，考虑下面的程序代码：

```
/* 逐字符地以大写形式输出“p” 中每个字母 */
for (i = 0; i < strlen (p); i++)
        printf ("%c", toupper (p[i]));
```

如果编译器不知道 strlen() 是一个纯函数，则它会在每次循环中调用该函数！

如果 strlen() 被标记成纯函数，聪明的程序设计者 —— 以及编译器 —— 将会写出或产生出如下的程序代码：

```
size_t len;

len = strlen (p);
for (i = 0; i < len; i++)
        printf ("%c", toupper (p[i]));
```

顺便一提，更聪明的程序设计者（例如本书的读者）会写出如下的程序代码：

```
while (*p)
        printf ("%c", toupper (*p++));
```

纯函数返回 void 是不合法的，而且的确毫无意义，因为返回值是此类函数的唯一目的。

常量函数

“常量”(constant) 函数是纯函数更严格的一种变体。此类函数无法访问全局变量，而且无法以指针作为参数。因此，常量函数的返回值只反映了以值传递的参数。纯函数上可能进行的额外优化动作也可能应用于此类函数上。数学 (math) 函数，例如 abs()，便是常量函数的一个例子(假设它们不会保存状态或是以其他方式在名称的优化上动手

脚)。要标记常数函数可以使用 const 关键字：

```
__attribute__ ((const)) int foo (int val) { /* ... */ }
```

如同纯函数，让一个常量函数返回 void 毫无意义。

无返回值的函数

如果一个函数没有返回值 —— 或许是因为它会固定调用 exit()，程序设计者可以使用 noreturn 关键字标记该函数，让编译器知道此事实：

```
__attribute__ ((noreturn)) void foo (int val) { /* ... */ }
```

于是，编译器可以进行额外的优化动作，因为它知道在任何情况下调用该函数都不会有返回值。此类函数返回 void 以外的任何值皆毫无意义。

分配内存的函数

如果一个函数所返回的指针绝对不可能是现有内存分配的别名（注 1）—— 这几乎可确定是因为该函数刚完成新的内存分配并返回指向它的指针，程序设计者可以使用 malloc 关键字来表示该函数，于是编译器可以对该函数进行适当的优化动作：

```
__attribute__ ((malloc)) void * get_page (void)
{
        int page_size;

        page_size = getpagesize ();
        if (page_size <= 0)
                return NULL;

        return malloc (page_size);
}
```

注 1： 现有内存分配的别名意味着有两个或多个指针变量指向同一个内存地址。这可能肇因于一些小问题（例如将一个指针赋值给另一个指针），也可能肇因于一些较复杂、较不明显的问题。如果一个函数所返回的是刚分配的内存空间的地址，就不应该存在指向相同地址的其他指针。

强迫调用者检查返回值

warn_unused_result 属性不是优化动作，而是一个程序设计辅助工具，用于指示编译器，当函数的返回值未被存储或者未被用在条件语句中时产生警告信息：

```
__attribute__ ((warn_unused_result)) int foo (void) { /* ... */ }
```

当一个函数的返回值特别重要时，这让程序设计者得以确保所有调用者（caller）都会检查与处理来自此函数的返回值。当函数具有重要但是常被忽略的返回值时，例如 read()，便可以使用此属性。此类函数不能返回 void。

将函数标记为已废弃

deprecated 属性用于指示编译器，在指定函数被调用时在调用该函数的位置上产生一个警告信息：

```
__attribute__ ((deprecated)) void foo (void) { /* ... */ }
```

这可协助程序设计者放弃已被废弃和过时的接口。

将函数标记为已使用

有时，对于特殊函数的调用编译器是看不到程序代码的。以 used 属性标记一个函数，可提示编译器：程序已使用该函数，尽管看起来似乎从未引用该函数：

```
static __attribute__ ((used)) void foo (void) { /* ... */ }
```

编译器会因此而输出作为结果的汇编语言，而且不会显示关于函数未使用的警告信息。如果一个静态函数仅被调用自手写的汇编程序代码，便可以使用此属性。一般情况下，若编译器不知道有任何调用，则会产生警告信息并对该函数进行可能的优化。

将函数或参数标记为未使用

unused 属性用于提示编译器指定的函数或函数参数未使用并提示它不要发出任何相应的警告信息：

```
int foo (long __attribute__ ((unused)) value) { /* ... */ }
```

如果你以 -W 或 -Wunused 进行编译，而且你想要捕获未使用的函数参数，但是你偶尔会使用必须符合预定特征（predetermined signature）的函数（常见于事件驱动 GUI 程序设计或信号处理程序），便可以使用此属性。

打包结构

packed 属性用于提示编译器，应该尽可能以最少量的空间将一个数据类型或变量打包（pack）进内存，这可能会导致无法符合对齐的需求。如果此属性在 struct 或 union 上被指定，则其所有的变量皆会被打包进内存。如果只对一个变量指定此属性，则只有特定的对象会被打包进内存。

如下的定义会将结构中的所有变量打包进最少量的内存空间：

```
struct __attribute__ ((packed)) foo { ... };
```

举例来说，一个结构中包含了一个 char 且后面跟着一个 int，但是你很有可能会发现，对齐内存地址的整数并未紧跟在 char 后面，例如在3个字节之后。编译器为了对齐变量，在 char 与 int 之间插入了未使用的填充字节。一个被打包进内存的结构缺少填充字节，因此所耗用的内存比较少，但是无法符合架构的对齐需求。

扩大变量的对齐边界

除了允许变量的打包，GCC 还允许程序设计者为给定的变量指定一个最小的对齐边界。于是 GCC 至少会将指定的变量对齐此值，对照于架构和 ABI 所规定的最小对齐需求。例如，下面这条语句以 32 B 的最小对齐边界（对照于具有 32位整数的机器上 4 B 的典型对齐边界）来声明一个名为 beard_length 的整数。

```
int beard_length __attribute__ ((aligned (32))) = 0;
```

强行设定数据类型的对齐边界通常仅用于处理对齐需求大于架构本身的硬件，或是当你手动混合 C 和汇编代码，而你想使用需要特别对齐边界的指令时。一个例子是为存储在处理器缓存线上的常用变量使用对齐功能以便优化缓存行为。Linux 内核便会利用此技术。

指定最小对齐边界的另一种做法是，你可以要求 GCC 将指定数据类型对齐任何数据类型曾用过的最大的最小对齐边界。例如，下面这道指令会提示 GCC 将 parrot_height 对齐它曾用过的最大对齐边界，而这可能是 double 的对齐边界：

```
short parrot_height __attribute__ ((aligned)) = 5;
```

这一决定通常涉及空间／时间的权衡：以此方式对齐变量会耗用较多的空间，但是对它们进行复制（以及其他复杂的操作）会比较快，因为编译器可以发出处理最大内存数量的机器指令。

各种架构或系统的工具链可能会替变量的对齐边界加上最大限额。例如，在某些 Linux 架构上，链接器（linker）无法认出超出默认值的对齐边界。在此情况下，使用此关键字所提供的对齐边界会被转换成所允许的最小对齐边界。举例来说，如果你所要求的是 32 字节的对齐边界，但是系统链接器的对齐边界不可以超过 8 个字节，于是变量将对齐 8 个字节的边界。

将全局变量放入寄存器

GCC 允许程序设计者将全局变量放入特定的机器寄存器，于是变量在程序执行期间会驻留于该处。GCC 将此类变量称为全局寄存器变量（*global register variables*）。

设定的语法需要程序设计者指定机器寄存器。下面的例子使用的是 ebx：

```
register int *foo asm ("ebx");
```

程序设计者必须选择不会对函数造成影响的变量：也就是所选用的变量必须可供局部函数使用，可存储和恢复函数调用的调用，以及不得指定架构或操作系统 ABI 所使用的特殊用途寄存器。如果寄存器选用不当，编译器将产生警告信息；如果寄存器选用适当——此例所使用的 ebx 适用于 x86 架构，编译器将会停止使用寄存器本身。

如果变量常被使用，此类优化动作可提供巨大的性能提升。虚拟机便是一个好例子。例如，将存储虚拟堆栈帧指针的变量放入寄存器会获得相当大的增益。另一方面，如果架构本来就极需要使用寄存器（如 x86 架构），则此优化动作毫无意义。

全局寄存器变量无法用于信号处理程序或供一个以上的线程使用。它们也无法有初始值，因为不存在任何机制让可执行文件为寄存器提供默认的内容。全局寄存器变量应该在任何函数定义之前声明。

分支注释

GCC 允许程序设计者注释一个表达式的预期值——例如，提示编译器某个条件语句是否有可能为真或假。于是 GCC 可以重新安排执行块，以及进行其他的优化动作来改善条件分支的性能。

用于分支注释的 GCC 语法相当难看。为了让分支注释能够顺眼一点，我们会使用如下的预处理器宏：

```
#define likely(x)     __builtin_expect (!!(x), 1)
#define unlikely(x)   __builtin_expect (!!(x), 0)
```

下面的例子将一个分支标记成不可能为真，也就是有可能为假：

```
int ret;

ret = close (fd);
if (unlikely (ret))
        perror ("close");
```

反过来说，下面的例子将一个分支标记成有可能为真：

```
const char *home;

home = getenv ("HOME");
if (likely (home))
        printf ("Your home directory is %s\n", home);
else
        fprintf (stderr, "Environment variable HOME not set!\n");
```

如同内嵌函数，程序设计者往往会过度使用分支注释。一旦开始标记表达式，你就可能会有想标记所有表达式的冲动。但是请注意，除非你可以当场毫无疑问地看出表达式几乎在任何时刻（例如有 99 %的确定性）将为真或假，否则你不应该将表达式标记成 likely 或 unlikely。很少发生错误者适合使用 unlikely()。然而，请记住，预测错误比不预测还糟糕。

取得一个表达式的数据类型

GCC 所提供的 typeof() 关键字可用于取得指定表达式的数据类型。就语义而言，此关键字的运作如同 sizeof()。例如，下面的表达式会返回 x 所指向的对象的数据类型：

```
typeof (*x)
```

我们可以使用此数据类型来声明一个数组 y：

```
typeof (*x) y[42];
```

typeof() 常用于编写“安全的”宏，用来求得宏参数的类型：

```
#define max(a,b) ({            \
        typeof (a) _a = (a); \
        typeof (b) _b = (b); \
       _a > _b ? _a : _b; \
})
```

取得一种数据类型的对齐边界

GCC 所提供的 `__alignof__` 关键字可用于取得指定对象的对齐边界。此值因架构和 ABI 而异。如果当前的架构没有要求对齐边界，则此关键字会返回 ABI 所建议的对齐边界；否则，此关键字会返回所要求的最小对齐边界。

语法如同 `sizeof()`：

```
__alignof__(int)
```

依据架构的不同，这可能会返回 4，因为 32 位的整数一般会对齐 4 个字节的边界。

此关键字也可以用于左值。在此情况下，所返回的是支持的数据类型的最小对齐边界，而不是指定左值的实际对齐边界。如果最小对齐边界已被 `aligned` 属性改变（参见稍早在“扩大变量的对齐边界”一节所做的说明），则 `__alignof__` 会反映改变的结果。

例如，下面的结构：

```
struct ship {
        int year_built;
```

数目（由零起算）。例如，下面的结构：

```
struct rowboat {
        char *boat_name;
        unsigned int nr_oars;
        short length;
};
```

实际的偏移量取决于变量的大小以及架构的对齐需求与填充的行为，但是在 32 位机器上，我们可以预期，对 struct rowboat 以及 boat_name、nr_oars 和 length 调用 offsetof() 会分别返回 0、4 和 8。

在 Linux 系统上，应该使用 GCC 的关键字来定义 offsetof() 宏，而不需要重新定义。

取得函数的返回地址

GCC 提供了一个关键字用于取得当前函数的返回地址或当前函数的其中一个调用者：

```
void * __builtin_return_address (unsigned int level)
```

参数 level 用于指出应该返回调用链（call chain）中哪个函数的地址。值为 0 用于要求当前函数的返回地址，值为 1 用于要求当前函数调用者的返回地址，值为 2 用于要求“该”函数调用者的返回地址，依此类推。

如果当前函数是一个内嵌函数，则所返回的就是进行调用的函数的地址。如果这是不可接受的，可使用 noinline 关键字（参见稍早在“禁止内嵌”一节所做的说明）迫使编译器不要嵌入函数。

__builtin_return_address 关键字有若干用法。一种是作为调试或提供信息用，另一种是用于解开调用链（call chain），以便实现“自省”（introspection）的能力、crash dump 工具、调试程序等。

注意，有些架构只会返回进行调用的函数的地址。在此类架构上，非零的参数值会产生随机的返回值。因此，任何非零的参数都是不具可移植性的，而且应该只用于调试。

案例范围

GCC 允许 case 语句标签为单一块指定一个范围值。语法如下：

```
case low ... high:
```

例如：

```
switch (val) {
case 1 ... 10:
        /* ... */
        break;
case 11 ... 20:
        /* ... */
        break;
default:
        /* ... */
}
```

此功能对于 ASCII 的案例范围也相当有用：

```
case 'A' ... 'Z' :
```

注意，三点号（...）的前后应该有一个空格，否则编译器可能会被搞混，尤其是在使用整数范围的时候。所以你一定要这么做：

```
case 4 ... 8:
```

不要这么做：

```
case 4...8:
```

void 以及函数指针计算

GCC 允许对指向 void 类型的指针以及指向函数的指针进行加法以及减法的运算。一般而言，ISO C 并不允许对此类指针进行运算，因为“void 的大小”是一个愚蠢的概念，而且取决于指针实际所指向的对象。为了让此类运算容易进行，GCC 将所指向对象的大小视为一个字节。因此，如下的程序代码片段便可以让 a 前进一步：

```
a++;          /* a 是一个 void 指针 */
```

选项 -Wpointer-arith 用于提示 GCC 在使用此扩展功能时产生一个警告信息。

更具可移植性并且更美观

老实说，__attribute__ 的语法并不美观。本章所提到的扩展功能中有一些基本上需要使用预处理器宏以便粉饰其外观，好让它们的使用赏心悦目。

这并不难，只需要用到些许的预处理器魔法。此外，相同的动作在非 GCC 编译器（无论它是哪种编译器）的情况下，通过取消 GCC 扩展的定义，我们可以让 GCC 的扩展具有可移植性。

要完成此事，请将如下的程序代码放入头文件中，并且在你的源码文件中引用该头文件：

```
#if __GNUC__ >= 3
# undef  inline
# define inline         inline __attribute__ ((always_inline))
# define __noinline     __attribute__ ((noinline))
# define __pure         __attribute__ ((pure))
# define __const        __attribute__ ((const))
# define __noreturn     __attribute__ ((noreturn))
# define __malloc       __attribute__ ((malloc))
# define __must_check   __attribute__ ((warn_unused_result))
# define __deprecated   __attribute__ ((deprecated))
# define __used         __attribute__ ((used))
# define __unused       __attribute__ ((unused))
# define __packed       __attribute__ ((packed))
# define __align(x)     __attribute__ ((aligned (x)))
# define __align_max    __attribute__ ((aligned))
# define likely(x)      __builtin_expect (!!(x), 1)
# define unlikely(x)    __builtin_expect (!!(x), 0)
#else
# define __noinline     /* no noinline */
# define __pure         /* no pure */
# define __const        /* no const */
# define __noreturn     /* no noreturn */
# define __malloc       /* no malloc */
# define __must_check   /* no warn_unused_result */
# define __deprecated   /* no deprecated */
# define __used         /* no used */
# define __unused       /* no unused */
# define __packed       /* no packed */
# define __align(x)     /* no aligned */
# define __align_max    /* no align_max */
# define likely(x)      (x)
# define unlikely(x)    (x)
#endif
```

例如，下面的程序代码使用我们的简写将一个函数标注为纯函数：

```
__pure int foo (void) { /* ... */
```

如果编译器是 GCC，则该函数会被标上 `pure` 属性；如果编译器不是 GCC，则预处理器会将 `__pure` 记号（token）替换成 no-op（无操作）。注意，你可以把多个属性放在单一的定义项中，因此你可以在单一定义项中使用多个此类定义而毫无问题。

更方便、更美观并且可移植！

参考书目

此处将所推荐的系统编程相关书籍分成四类。读者并不需要阅读这些著作，它们只是关于特定议题我所阅读的书籍。你是否发现自己渴望了解此处所推荐的书籍，它们都是我的最爱。

这些书籍中，有些可让读者获得阅读本书所需具备的基本知识，例如 C 程序语言；有些可作为本书的绝佳补充资料，例如探讨 *gdb*、Subversion（*svn*）或操作系统设计的书籍；有些所探讨的是本书范围以外的课题，例如 multithreading of socket。无论如何，你都可以参考它们。当然，这份书目绝不可能列得完整 —— 你可以随心所欲地探索其他资源。

C 程序语言相关书籍

下面所列的是探讨 C 程序语言的书籍。C 是系统编程的共同语言。如果你并不编写 C 程序，而是使用过其他的程序语言，其中应该也可以找到对你有所帮助的书籍。如果真的找不到，不要错过第一本书 —— 著名的 *K&R*。它简明扼要地呈现了 C 语言的简单性。

《*The C Programming Language*, 2nd ed.》（Prentice Hall 1988 年出版）， Brian W. Kernighan 和 Dennis M. Ritchie 编写。

本书是 C 程序设计的圣经。它由 C 程序语言的作者以及他当时的合作者共同编写。

《*C in a Nutshell*》(O'Reilly Media 2005 年出版)，Peter Prinz 和 Tony Crawford编写。

这是一本好书，内容涵盖 C 语言以及标准 C 链接库。

《*C Pocket Reference*》（O'Reilly Media 2002 年出版），Peter Prinz 和 Ulla Kirch-Prinz 编写。

这是一本简明的 C 语言参考手册，内容涵盖 ANSI C99。

《*Expert C Programming*》（Prentice Hall 1994 年出版），Peter van der Linden 编写。

对 C 程序语言较不为人知的方面进行了精彩的讨论，内容深富智慧及幽默感。这本书充满了风马牛不相及的笑话，但是我非常喜欢。

《*C Programming FAQs: Frequently Asked Questions*, 2nd ed》（Addison-Wesley 1995 年出版），Steve Summit 编写。

本书包含了 400 个以上关于 C 程序语言的常见问题与解答（FAQ）。FAQ 中有许多问题与解答在 C 专家眼中视为理所当然，但是其中一些更重要的问题与解答甚至会令最博学的 C 程序设计者印象深刻。如果你可以回答所有的问题，你才是一个真正的 C 语言高手！本书的唯一缺点是尚未更新成符合 ANSI C99，因为已经发生了一些变化（我在自己的书上直接做了修正）。注意本书的在线版本，有可能最近已经做了更新。

Linux 编程相关书籍

下面是有关 Linux 编程的书籍，包括本书并未探讨的主题（socket、IPC 以及 pthread）以及 Linux 编程工具（CVS、GNU Make 以及 Subversion）。

《*Unix Network Programming, Volume 1: The Sockets Networking API*, 3rd ed.》（Addison-Wesley 2003 年出版），W. Richard Stevens 等人编写。

这是一本关于 socket API 的权威巨著，可惜不是专为 Linux 而写，不过最近加入了 IPv6 的内容。

《*UNIX Network Programming, Volume 2: Interprocess Communications*, 2nd ed.》（Prentice Hall 1998 年出版），W. Richard Stevens 编写。

本书对进程间通信（IPC）做了绝佳的探讨。

《*PThreads Programming: A POSIX Standard for Better Multiprocessing*》（O'Reilly Media 1996 年出版），Bradford Nichols 等人编写。

本书介绍了 POSIX 的 threading API（pthread）。

《*Managing Projects with GNU Make*, 3rd ed.》(O'Reilly Media 2004 年出版), Robert Mecklenburg 编写。

这是一本绝佳的 GNU Make 指南，在 Linux 中，GNU Make 是建立软件项目的典型工具。

《*Essential CVS*, 2nd ed.》(O'Reilly Media 2006 年出版), Jennifer Versperman 编写。

这是一本绝佳的 CVS 指南，在 Unix 系统中，CVS 是进行版本控制以及源码管理的典型工具。

《*Version Control with Subversion*》(O'Reilly Media 2004 年出版), Ben Collins-Sussman 等人编写。

Subversion 是 CVS 的替代品，在 Unix 系统中，Subversion 可用于进行版本控制以及源码管理。本书由 Subversion 的三位作者共同编写。

《*GDB Pocket Reference*》(O'Reilly Media 2005 年出版), Arnold Robbins 编写。

这是一本关于 gdb（Linux 的调试程序）的参考手册。

《*Linux in a Nutshell*, 5th ed.》(O'Reilly Media 2005 年出版), Ellen Siever 等人编写。

这是一本无所不包的 Linux 参考书籍，书中包括构成 Linux 开发环境的许多工具。

Linux 内核相关书籍

此处列出了两本探讨 Linux 内核的著作。探究这个主题的理由有三个。首先，内核为用户空间提供了系统调用接口，因此它是系统编程的关键所在。其次，内核的行为与特色揭示了它与所运行的应用程序的交互方式。最后，Linux 内核是一个令人赞赏的程序实现，而且这两本书都很有趣。

《*Linux Kernel Development*, 2nd ed.》(Novell Press 2005 年出版), Robert Love 编写。

此书非常适合想了解 Linux 内核设计和实现的系统编程者（当然，我还是必须提到我自己在这个主题的著作！）。本书不是一本 API 参考手册，书中对 Linux 内核所使用的算法以及所作的决策着墨甚多。

《*Linux Device Drivers*, 3rd ed.》(O'Reilly Media 2005 年出版), Jonathan Corbet 等人编写。

这是一本非常重要的指南，它会告诉你如何替 Linux 内核编写驱动程序，随附绝佳的 API 参考资料。尽管本书的重点放在设备驱动程序上，但是书中所作的讨论

将有益于任何类型的编程人员，包括仅想深入了解 Linux 内核内幕的系统编程人员。此书可以弥补我这本书的不足。

操作系统设计相关书籍

下面两本著作并未具体论述 Linux，而是以抽象的概念来探讨操作系统设计。正如我在本书所强调的，充分了解你所设计的系统的唯一目的就是改善你的输出。

《*Operating Systems*, 3rd ed.》（Prentice Hall 2003 年出版），Harvey Deitel 等人编写。

本书是一本精心的杰作，它除了探讨操作系统设计的理论，还提供了一流的案例研究，并将理论应用到实践中。环顾所有的操作系统设计相关书籍，我最喜欢这本书：因为它的现代性、可读性和完整性。

《*UNIX Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programming*》（Addison-Wesley 1994 年出版），Curt Schimmel 编写。

虽然本书主要探讨系统编程，但是本书替并行性与现代缓存机制的问题提供了绝佳的解决方案，我甚至向我的牙医推荐这本书。