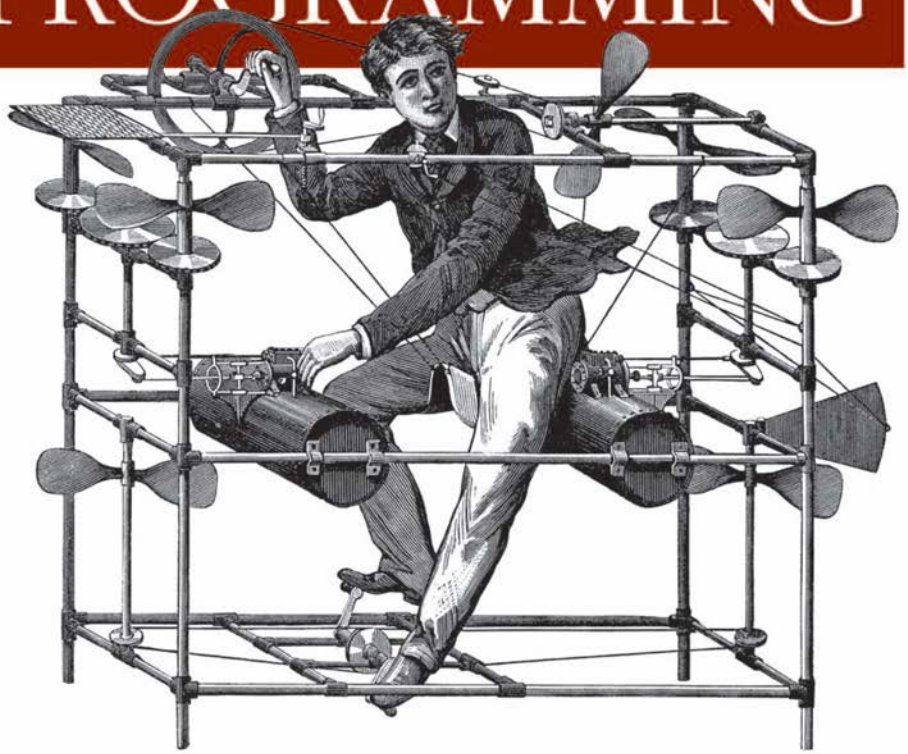
SYSTEM AND LIBRARY CALLS EVERY PROGRAMMER NEEDS TO KNOW

LINUX SYSTEM PROGRAMMING



O'REILLY®

ROBERT LOVE

LINUX

System Programming

Robert Love

Translated by: Kevin Zhang

目录

序

Linux 内核开发者在脾气暴躁的时候，总是喜欢甩出这么一句：用户空间只是内核的测试负载而已。

内核开发者希望通过这句话，来尽可能地推脱任何用户空间代码运行失败的责任。在他们的观点里，用户空间程序的开发者应该立刻去改正他们自己的代码，因为任何问题都绝对不会是内核引起的。

为了证明错误通常都不是内核导致的，一名 Linux 内核开发者的领军人物早在三年前，就在一次会议中发表了一篇名为“Why User Space Sucks”的演讲，指出了很多可怕的用户空间代码的实际例子，而几乎所有人每天都要依赖于这些代码。其它一些内核开发者则创建了各种工具，来展示用户空间程序是如何的恶劣，滥用硬件以及放光笔记本电池所有的电。

尽管内核开发者嘲笑用户空间代码仅仅只是“测试负载”，实际上所有这些内核开发者同样也每天依赖于用户空间代码。如果没有用户空间代码，内核所能做的就只是以交互的方式将 ABABAB 打印到屏幕上。

现在，Linux 已经成为有史以来最灵活和强大的操作系统之一，从最小的蜂窝电话和各种嵌入式设备，到超过 70%的世界前 500 台超级计算机，都运行着 Linux 操作系统。没有任何的操作系统具有如此大的伸缩性，能够迎接各种不同类型的硬件和环境的挑战。

除了内核，Linux 用户空间的代码也能够在所有这些平台中运行，并且给人们提供了大量的应用程序和实用工具。

在本书中，作者 Robert Love 几乎给大家讲解了 Linux 的每一个系统调用。他的这本书，能够让你从用户空间的角度来完全理解 Linux 内核是如何工作的，同时也向你展示了如何利用和增强 Linux 系统的功能。

本书的内容将向你展示如何编写代码，使它们能够运行于所有不同 Linux 发行版和各种类型的硬件。它还能帮你理解 Linux 的工作机制，以及如何利用 Linux 的灵活性。

最后，这本书教你怎样避免编写愚蠢的代码，这是一件好事情，不是吗？

—Greg Kroah-Hartman

前言

本书是关于系统编程的——更明确的说，是关于 Linux 系统编程。系统编程就是编写系统软件，这是一些比较底层的代码，直接与内核和核心系统库打交道。换句话说，本书的主题就是 Linux 系统调用，以及其它一些底层的函数，例如 C 库。

虽然已经有了很多关于 Unix 系统编程的书，但却很少有完全专注于 Linux 系统的，能够跟上 Linux 最新发布版和 Linux 专有高级接口的书就更少了（如果有的话）。而且，本书还有另一层好处：我已经为 Linux 编写了大量的代码，内核和系统软件都有。实际上，本书讲解的一些系统调用和某些特性，就是我本人实现的。因此，本书传达了很多核心内部知识，不仅仅是系统接口应该怎样工作，还包括系统接口实际上是怎样工作的，以及你（作为一名程序员）怎样才能最有效地使用这些系统调用。所以，本书实际上将 Linux 系统编程指南、Linux 系统调用参考手册、以及如何编写更巧妙更快速的代码组合在一起。本书的内容是有趣而容易理解的，无论你是不是在系统层面上编程，本书都能教你各种技巧，让你编写更好的代码。

对读者的假设

本书假设读者已经熟悉 C 语言编程和 Linux 编程环境——不需要精通，但至少得知道它们。如果你还没有阅读过任何 C 语言编程的书，例如 Brian W. Kernighan 和 Dennis M. Ritchie 的经典书籍《The C Programming Language》（Prentice Hall；本书也被称为 K&R），那我强烈推荐你去买一本。如果你用不惯 Unix 的文本编辑器——Emacs 和 Vim 是最常用的，也是最值得推荐的——那你最好现在开始就去熟悉一个。同时你还需要熟悉 gcc，gdb，make 等工具的基本使用。有很多的书讲解 Linux 编程工具和实践，本书最后的参考书目列出了一些比较好的参考资料。

我对读者的 Unix 或 Linux 系统编程知识没有做太多的假设。本书将从基础知识开始，然后慢慢到最高级的接口以及优化技巧。任何水平的读者，都会觉得本书有用而且能够学到一些新知识（至少我希望如此）。我在写作本书的过程中，确实就学到不少。

同样我也不对读者阅读本书的动机作任何假设。工程师希望能够在底层更好地编程，这当然是可以的；高层的开发者希望能够更加深入地理解系统基础，也会对本书感兴趣；纯粹的黑客也是欢迎的，本书同样也能满足他们的欲望。无论读者想要和需要什么，本书都能使他们满意，只要是关于 Linux 系统编程。

无论你的动机是什么，have fun！

本书内容

本书一共分为 10 个章节，一个附录，一个参考书目。

第一章，　简介与基本概念

本章作为入门介绍，提供了 Linux、系统编程、内核、C 库、C 编译器的简介。即使是高级读者也应该浏览一下本章——相信我。

第二章，　文件 I/O

本章介绍了文件，Unix 环境里最重要的抽象；文件 I/O，Linux 编程模式的基础。包含读取和写入文件，以及其它一些基本的文件 I/O 操作。本章的末尾讨论了 Linux 内核对文件的实现和管理机制。

第三章， 缓冲 I/O

本章讨论基本文件 I/O 接口的一个特性——缓冲区管理——简单介绍了缓冲 I/O，然后重点讲解标准 I/O。

第四章， 高级文件 I/O

本章完成对 I/O 的讲解，包括高级 I/O 接口、内存映射、和优化技术。本章结尾讨论了如何避免 seeks，以及 Linux 内核在 I/O 调度中的角色。

第五章， 进程管理

本章介绍了 Unix 第二重要的抽象，进程，以及基本的进程管理系统调用家族，包括古老的 fork 调用。

第六章， 高级进程管理

本章继续讨论进程的高级管理，包括实时进程等。

第七章， 文件和目录管理

本章讨论创建、移动、复制、删除以及其它文件和目录管理方式。

第八章， 内存管理

本章讲解内存管理。首先介绍 Unix 中内存的概念，例如进程地址空间和页等等；然后讨论从内核获得内存和归还内存的接口。本章结尾讨论内存的高级接口。

第九章， 信号

本章讲解信号机制。首先是对信号的基本讨论，以及信号在 Unix 系统中的角色。然后是信号的接口，从基本接口开始，最终到信号的高级接口。

第十章， 时间

本章讨论时间、睡眠、以及时钟管理。从最基本的接口到 POSIX 时钟，以及高精度定时器，都有详细的讲解。

附录，C 语言的 GCC 扩展特性

这个附录回顾了 gcc 和 GNU C 提供的许多优化机制，例如标记函数常量，纯函数（pure），以及内联函数（inline）等等。

本书最后包含一个推荐阅读的参考书目，列出了一些对本书非常有用的补充，以及阅读本书之前的必备先决条件。

本书使用的版本

Linux 系统接口可以定义为 ABI（Application Binary Interface）和 API（Application Programming Interface），由 Linux 内核、GNU C 库（glibc）和 GNU C 编译器提供。本书所讲解的系统接口版本是：Linux 内核 2.6.22、glibc2.5、gcc4.2。书中的接口应该能够向后兼容以前的老版本（新接口除外），同时向前兼容新版本。

如果把所有正在发展的操作系统看作一个移动的对象，Linux 就是一头狂暴的印度豹。它的发展以天来计算，而不是以年，内核与其它组件频繁地发布新版本。没有任何一本书能够跟得上这样的变化。

无论如何，系统编程的编程环境是基本不变的。内核开发者花了很大力气来避免破坏系统调用，glibc 的开发者也同样高度重视向前和身后兼容性，并且 Linux 工具链也能够产生跨版本的兼容代码（特别是 C 语言）。因此，在 Linux 不断快速发展的同时，Linux 系统编程却能够保持稳定。基于系统某个版本的书籍，特别是从 Linux 开发的角度来讲，可以非常好的保持它的有效性。我想说的其实很简单：不要担心系统接口会变化，*赶快购买本书吧！*

本书使用的惯例

（印刷字体使用约定省略）

提供简练但却可用的示例代码片断是件很痛苦的事情。本书不会创建庞大的程序，取而代之的是许多简单的示例程序。这些例子都具有描述性而且是完全可用的，同时还短小而清晰，我希望在第一遍阅读时它们能提供有用的指导，然后在后续的阅读过程中仍然能够保持很好的参考作用。

本书的所有例子程序几乎都是自包含的，这意味着你可以简单地拷贝它们到你的编辑器，将它们使用到实际情况中。除非特别提到，所有的代码片断都应该可以不使用任何标志进行编译（某些情况下，可能需要链接指定的库）。我推荐使用下面命令来编译源文件：

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

这样就把源文件 snippet.c 编译成了可执行的二进制 snippet，允许所有的警告检查、重要而健全的优化、也允许调度。本书的所有代码使用这个命令进行编译应该不会有错误和警告——当然，你可能需要给代码片断加上一个基本的骨架。

使用示例代码

本书的目的是帮助你更好地完成工作。一般来说，你可以在你的程序和文档中使用本书的代码。你不需要联系我们获得许可，除非你要复制和发布代码的重要部分。例如，编写一个程序，其中几块使用到本书的代码并不需要许可。销售或者发布 O'Reilly 书籍的示例代码的 CD-ROM 则需要获得许可。引用本书正文和代码来回答别人的问题不需要许可，合并大量本书的示例代码到你的项目中则需要许可。

我们感激对本书添加引用。通常一条引用包括标题、作者、出版社和 ISBN。例如："*Linux System Programming* by Robert Love. Copyright 2007 O'Reilly Media, Inc., 978-0-596-00958-8."

如果你确认你对本书代码的使用不在许可规定范围内，请通过 *permissions@oreilly.com* 联系我们。

感谢

（这部分内容省略，估计大家也不会看。）

第一章 简介与基本概念

本书是关于系统编程的，系统编程也就是编写系统软件的技术。系统软件一般都处于底层，直接与内核和核心系统库相连。系统软件包括你的 Shell 和文本编辑器、编译器、调试器、核心实用工具、以及系统后台 daemon 程序。这些都完全是系统软件，基于内核与 C 库。其它一些软件（例如高层的 GUI 应用程序）大多数时候都处于更高的层次，只是偶尔才会深入到底层。某些程序员每天的每个时候都在编写系统软件；其它程序员则在某些时候才会做这项工作。但是，理解系统编程对任何程序员都是有好处的。无论如何，系统编程都是所有软件开发的核心。

特别地，本书关注 Linux 系统编程。Linux 是一个类 Unix 的现代操作系统，最早由 Linus Torvalds 编写，然后是一个遍布全球的黑客社区。尽管 Linux 继承了 Unix 的目标和哲学，Linux 却并不是 Unix。相反，Linux 遵循着自己的路线，在出现分歧的时候，总是聚焦于实际。通常情况下，Linux 系统编程的核心与其它 Unix 系统是一样的。除了基本的核心部分，Linux 和 Unix 还是存在一定的区别——和传统的 Unix 系统比较起来，Linux 普遍拥有更多的系统调用、不同的行为和一些新特性。

系统编程

传统地讲，所有的 Unix 编程都是系统级的编程。由于历史的原因，Unix 系统并没有包含太多的高级抽象。即使是在 X Window 下的开发环境里，也完全暴露了 Unix 系统的核心 API。因此，可以说这是一本关于 Linux 通用系统编程的书。但请注意本书并没有涉及到 Linux 的编程环境——例如本书并没有 make 的指南。本书讲解的是现代 Linux 的系统编程 API。

与系统编程形成鲜明对比的是应用编程。系统级和应用级编程既存在很大区别，又有相似之处。系统编程与应用编程的区别主要在于系统程序员必须对他们工作的硬件环境和操作系统非常熟悉。当然，库的使用和函数调用也存在区别。根据某个程序所处的层次，有时候很区分到底是系统编程还是应用编程。但是，通常来说，从应用编程切换到系统编程（或者反过来）都不会很难。即使应用非常高级，而且远离系统底层，了解系统编程也是很重要的。任何形式的编程都能够使用系统编程的最佳实践。

过去的几年里，应用编程有越来越远离系统编程的趋势，应用开发已经达到很高的层次，例如 Web 应用（JavaScript 或 PHP），或者托管代码（C#或 Java）。但是这些开发并没有预示系统编程的死亡。实际上，总得有人去实现 JavaScript 解释器和 C#运行时环境吧，这本身就是系统编程。此外，PHP 或 Java 的开发者也能从系统编程中获得一些好处，理解了系统的核心，则无论做哪个层次的开发，都能够编写更好的代码。

撇开应用开发的这个发展趋势不谈，大部分的 Unix 和 Linux 代码却依然在系统级编写。这些代码大部分都是 C 语言编写，主要依赖于 C 库和内核所提供的接口。下面这些都是传统的系统编程——Apache，bash，cp，Emacs，init，gcc，gdb，glibc，ls，mv，vim 以及 X，这些应用都不会在短期内消失。

系统编程的大伞常常都包括内核开发，或者至少是设备驱动开发。但本书和其它系统编程书籍一样，并不关注内核开发。实际上，我们关注的是用户空间态的系统级编程，也就是内核之上的开发（当然理解内核对学习本书是很有帮助的）。同样，网络编程——Sockets 等——也没有涉及。设备驱动开发和网络编程都是非常庞大的主题，需要专门的书来讲解。

什么是系统级接口，我们怎样在 Linux 中编写系统级应用呢？内核和 C 库到底给我们提

供了什么？怎样编写优化的代码，Linux 提供了什么技巧？Linux 提供的系统调用对比其它类 Unix 系统有什么更加优雅的地方？它们又是怎样工作的？本书将为你解答这些问题。

Linux 系统编程有三个基础组成部分：系统调用、C 库、C 编译器，每个都应该单独解释一下。

系统调用

系统编程的起点是系统调用。系统调用（通常简短地称为 syscalls）是从用户空间态（你的文本编辑器、最喜欢的游戏等）向内核（系统的核心部分）发起的函数调用，这些调用向操作系统请求某些服务或资源。从最熟悉的 read()和 write()，到奇怪的 get_thread_area()和 set_tid_address()，都是系统调用。

Linux 内核实现的系统调用比大部分其它操作系统都要少得多。例如，i386 体系架构下的系统调用大概只有 300 个，而 Windows 操作系统则拥有上千个系统调用。在 Linux 内核里，各种机器体系架构（如 Alpha，i386，PowerPC）都实现了它们自己的系统调用。因此，某一种体系架构下可用的一个系统调用，在另一种架构下可能会有差别。无论如何，大部分系统调用——超过 90%——在所有体系架构下都有实现。本书讲解的就是这部分公共接口。

调用系统调用

用户空间态的应用是不可能直接链接到内核空间的。为了安全和可靠性，用户空间态应用不允许直接执行内核代码或者操作内核数据。相反，内核必须提供一种机制，这样用户空间态应用才能通知内核说它希望调用某个系统调用。然后应用就可以通过这个定义良好的机制，受限制地进入到内核，执行内核允许的那段代码。具体的机制和各种机器的体系架构有关，例如在 i386 下，用户空间应用执行一个软件中断指令，int，使用 0x80。这个指令促使切换到内核空间，也就是内核保护领域，在这里内核将执行软件中断的处理器。那么中断 0x80 的处理器是什么呢？就是系统调用的处理器。

应用告诉内核需要执行哪个系统调用，同时通过机器寄存器传递相关的参数。系统调用使用数字来表示，从 0 开始。在 i386 体系架构下，用户空间应用要请求系统调用 5（就是 open()），只需将 5 装入 eax 寄存器，然后发起 int 中断即可。

参数的传递通过类似的方法处理，例如在 i386 架构下，每个可能的参数使用一个对应的寄存器——ebx，ecx，edx，esi 和 edi 分别保存前五个参数。极少数情况下，系统调用超过五个参数，一个单独的寄存器将来用指向用户空间的缓冲区，在那里保存着所有的参数。当然，大部分系统调用都只有一两个参数。

其它体系架构处理系统调用的方式并不一样，但思想是一样的。作为一名系统程序员，你通常不需要了解内核是怎样处理系统调用的。在各个体系架构下都有相应的调用规范，编译器和 C 库能够自动地处理它们。

C 库

C 库（libc）是 Unix 应用的核心。即使你使用其它语言开发，也很可能需要使用 C 库，它被高层的库封装起来，提供核心服务，实现系统调用。在现代 Linux 系统里，C 库由 GNU libc 提供，简称 glibc，发音为 gee-lib-see，有时候也叫 glib-see。

GNU C 库比它的名字提供了更多的东西，除实现了 C 语言标准库，glibc 还封装了系统

调用，提供线程支持，以及基本的应用开发基础设施。

C 编译器

在 Linux 中，标准 C 编译器由 GNU Compiler Collection(gcc)提供。最初，gcc 只是 GNU 版本的 cc，也就是 C 编译器。那时 gcc 代表的是 GNU C 编译器。后来，GNU 添加了越来越多的语言支持。因此，现在的 GCC 是 GNU 编译器家族的通用名称。而 gcc 仍然被用来调用 C 语言编译器。在本书中，当我说到 gcc，一般都是指 gcc 这个程序，除非上下文是其它意思。

类 Unix 系统——包括 Linux——使用的编译器与系统编程关系非常紧密，因为编译器实现了 C 语言标准和系统 ABI，本章后面会介绍这两个概念。

API 和 ABI

程序员很自然地对程序的可移植性非常感兴趣，希望确保程序能够运行于任何需要支持的系统之上，不管是现在还是将来。他们希望在自己的 Linux 发行版中开发的程序，能够很好的运行在其它的 Linux 发行版中，同时也能运行于其它体系架构下的 Linux 或者新（老）版本的 Linux 系统中。

在系统层次，影响可移植性的因素，有两个单独的定义和描述。一个是 Application Programming Interface(API)，另一个是 Application Binary Interface(ABI)。这两个一起定义和描述了计算机软件不同模块之间的接口。

API

API 定义了软件各个模块之间在源码级的接口。它提供一组标准接口来实现抽象——通常是函数，这样软件的一个模块（通常是高层模块）就可以调用另一个模块（通常是低层模块）。例如，可能有一个 API 抽象了屏幕绘制文本，通过一组函数来提供文本显示的所有功能。API 仅仅定义了接口，实际提供 API 实现的那个模块被称为 API 的实现。

通常把 API 称为“契约”，这并不正确，至少从“契约”的字面意思来理解，API 并不是一个双向的协议。API 的使用者对 API 和实现没有任何输入，要么按照约定使用 API，要么就完全不用。API 仅仅保证如果双方都遵循 API，那它就是源代码兼容的。也就是说，API 的使用者可以在不同的 API 实现之间成功编译。

一个实际的例子就是 C 标准定义的 API，由 C 标准库实现。这个 API 定义了一组核心和基本的函数，像字符串操作等。

贯穿整本书，我们都依赖于现有的 API，例如第三章讨论的标准 I/O 库。Linux 系统编程中最重要的 API 将在本章后面的“标准”一节中讨论。

ABI

API 定义了源代码的接口，ABI 则定义了某个特定体系架构下，不同软件模块之间的底层二进制接口。它定义了一个应用如何与自身相结合，如何与内核结合，如何与库相结合。ABI 确保二进制的兼容性，保证了目标代码能够运行于拥有相同 ABI 的任何系统中，而不需

要重新编译。

与 ABI 有关的是调用规范、字节顺序、寄存器使用、系统调用、链接、库的行为、二进制目标代码格式等等。例如调用规范，定义了函数怎样被调用，参数怎样传递给函数，哪个寄存器被保留或抛弃，以及调用方怎样获得返回值。

尽管做了很多努力，希望能够为某个特定的体系架构下的多个操作系统（特别是 Unix 系统上的 i386 架构）定义同一个 ABI，但实际上并没有取得太大的成功，包括 Linux 在内的操作系统往往都定义了自己的 ABI。ABI 与体系架构紧密地绑定在一起，大多数的 ABI 都与机器的特殊架构相关，例如特殊的寄存器或者汇编指令。因此，每种机器体系架构在 Linux 下都有它自己的 ABI。实际上，我们往往以机器名来称呼特定的 ABI，例如 alpha 或者 x86-64。

系统程序员应当了解 ABI，不过通常并不需要记住它。工具链（编译器、链接器等）能够执行 ABI，一般不需要其它的支持。当然，理解 ABI 能够引导更加优化的编程，编写汇编代码或者定制工具链（这也是系统编程）时则可能需要 ABI。

Linux 下特定体系架构的 ABI，可以在因特网上找到，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，这么多年来 Unix 系统编程的基础都没有发生什么变化。当然，Unix 系统本身发展是相当快的，添加了很多新特性，某些行为也有一定的改变。为了给混乱的局面添加秩序，标准化组织就把系统接口编成正式的标准。这样的标准已经有无数个，但从技术上来说，Linux 并没有正式的遵守任何一个标准。实际上，Linux 的目标是尽量遵守最重要和最流行的两个标准：POSIX 和 Single Unix Specification(SUS)。

POSIX 和 SUS 为类 Unix 操作系统接口文档化 C 语言 API。POSIX 和 SUS 为遵守标准的 Unix 系统很好的定义了系统编程。

POSIX 和 SUS 的历史

在 1980 年代中期，美国电气和电子工程师协会（IEEE）最早开始 Unix 系统接口的标准化工作。Richard Stallman（自由软件运动的发起者）建议把标准命名为 POSIX，代表 Portable Operation System Interface（可移植操作系统接口）。

这项努力的第一个成果产生在 1988 年，就是 IEEE Std 1003.1-1988(POSIX 1988)。在 1990 年，IEEE 修订了 POSIX 标准，命名为 IEEE Std 1003.1-1990(POSIX 1990)。IEEE Std 1003.1b-1993(POSIX 1993 或者 POSIX.1b)添加了可选的实时特性，IEEE std 1003.1c-1995(POSIX 1995 或者 POSIX.1c)添加线程支持。到了 2001 年，所有可选的标准都被加进 POSIX1990 中，统一为一个标准：IEEE Std 1003.1-2001(POSIX 2001)。POSIX 标准的最后修订发布于 2004 年四月，也就是 IEEE Std 1003.1-2004。所有的核心 POSIX 标准都简称为 POSIX.1。

在整个 1980 年代和 1990 年代早期，所有 Unix 系统厂商都陷入了一场“Unix 大战”，每个厂商都在努力定义自己的 Unix 为标准的 Unix 操作系统。于是几个主要的 Unix 厂商组成了“开放组织”，一个合并自开放软件基金会（OSF）和 X/Open 的工业联盟。这个“开放组织”提供证明、白皮书和遵循标准测试。到了 1990 年代早期，Unix 大战到达顶峰，开放组织发布了单一 UNIX 规范。SUS 很快就流行起来，这很大程度上是因为它免费，而 POSIX 标准的费用却很高。今天，SUS 已经合并了最新的 POSIX 标准。

第一个 SUS 标准公布于 1994 年，遵循 SUSv1 标准的系统被标识为 UNIX95。SUS 的第二版公布于 1997 年，遵循该标准的系统被称为 UNIX98。第三个也是最新的 SUS 版本，SUSv3，

公布于 2002 年，遵循标准的系统则被标识为 UNIX03。SUSv3 结合并修订了 IEEE Std 1003.1-2001 和其它几个标准。贯穿本书，我都会明确提示被 POSIX 标准化的系统调用和其它接口，使用 POSIX 而不是 SUS 的原因是后者实际上包含了前者。

C 语言标准

Dennis Ritchie 和 Brian Kernighan 编写了著名的《C 程序设计语言》，这本书从 1978 年出版以来，很多年一直扮演着非正式 C 语言规范的角色，这个版本的 C 语言也被称为 K&R C。C 语言快速地取代了 BASIC 和其它一些语言，成为了计算机编程的“国际语言”。为了标准化这个非常流行的新语言，美国国家标准学会（ANSI）在 1983 年组织了 C 语言标准委员会，开始合并不同厂商的特性和对 C 的改进，同时也包括合并新的 C++语言。这是一个漫长和艰苦和过程，最终在 1989 年完成 ANSI C 标准。1990 年，国际标准化组织（ISO）批准了 ISO C90，标准基于 ANSI C，并做了一些很小的修改。

1995 年，ISO 发布了更新版本的 C 语言，ISO C95（虽然很少有实现）。然后到了 1999 年，C 语言有了一次巨大的更新，这就是 ISO C99，添加了许多新的特性，包括内联函数、新的数据类型、可变长度数组、C++风格的注释和一些新的库函数。

Linux 和标准

前面已经说过，Linux 的目标是尽可能地遵守 POSIX 和 SUS 标准。Linux 提供了 SUSv3 和 POSIX.1 的接口实现，包括可选的实时（POSIX.1b）和线程支持（POSIX.1c）。更重要的是，Linux 努力提供与 POSIX 和 SUS 标准需求一致的行为。通常，违反了标准就会被认为是一个 bug。Linux 希望能够遵循 POSIX.1 和 SUSv3 标准，但由于并没有执行正式的 POSIX 和 SUS 检验和证明（特别是当 Linux 有新版本或修订版本时），我也不能保证 Linux 是正式遵循 POSIX 和 SUS 标准的。

由于 Linux 非常尊重语言标准，Linux 发展得很好。GCC 的 C 编译器支持 ISO C99 标准，此外，gcc 还提供许多自己的 C 语言扩展。这些扩展统称为 GNU C，附录列出了它们。

Linux 早期在向前兼容上做得并不太好，不过现在这种情况已经不复存在。标准定义的所有接口，例如标准 C 库，很明显总是可以保证源码级兼容的。二进制兼容性则由 glic 来维持，至少每个 glibc 主要发布版本都是兼容的。由于 C 语言已经标准化，gcc 总是可以正确地编译合法的 C 代码，尽管 gcc 特定的扩展可能会不推荐使用，甚至最终在新版本中被移除。最重要的是，Linux 内核可以保证系统调用的稳定性，一旦 Linux 内核的稳定版本实现了某个系统调用，这个系统调用就会被一直保持下来。

LSB（Linux Standard Base）为大部分的 Linux 发行版进行了标准化。LSB 是由几个 Linux 厂商联合组成的项目，由 Linux 基金会（以前叫自由标准组织）赞助。LSB 扩展了 POSIX 和 SUS 标准，并且添加了几个自己的标准进去。LSB 试图提供一个二进制的标准，使目标代码能够运行于所有遵循标准的系统。大部分 Linux 厂商都在一定程度上遵循 LSB。

本书和标准

本书有意地避免倾向于任何一个标准，很多时候，Unix 系统编程书籍不应该阐述不同标准下相同接口的不同行为，或者某个系统调用是否被不同的系统实现。本书明确地关注于

现代 Linux 操作系统的系统编程，使用最新的 Linux 内核（2.6），gcc C 编译器（4.2）和 C 库（2.5）。

系统接口通常都保持不变——Linux 内核开发者花了相当大的力气不破坏系统调用接口，提供一定程度上的源码和二进制兼容性，这允许我们深入到 Linux 系统接口的细节，而不用关心它与其它类 Unix 系统和标准的兼容性问题。本书也将深入地讲解某些最新的 Linux 特定接口，这些接口在将来肯定也是有效的。基于我对 Linux 内部的深入理解，特别是 gcc 和内核的实现和行为，本书提供了一个内部的视角，讲解了 Linux 老手的最佳实践和优化技巧。

Linux 编程概念

本节简单的概述 Linux 系统所提供的服务。所有的 Unix 系统，包括 Linux，都提供了一组公共的接口和抽象。实际上，正是这些公共特性定义了一个完整的 Unix。文件和进程的抽象概念、管道和套接字的管理接口等等，构成了 Unix 系统的核心。

下面的概述假定你已经熟悉了 Linux 环境：我假定你能够正常使用 Shell、会使用基本的命令、知道如何编译 C 程序。本节并不是对 Linux 本身的概述，也不是对 Linux 编程环境的介绍，仅仅是对 Linux 系统编程基本组成的概述。

文件和文件系统

文件是 Linux 最最基础的抽象，Linux 遵循一种“任何事物都是文件”的哲学（尽管不像某些其它系统那样严格，例如 Plan9）。因此，很多交互作用都通过读取和写入文件来实现，即使某些时候它看上去并不像是你日常使用的那种文件。

要访问文件，首先必须打开它。文件的打开模式可以是读取、写入、或者两个一起。系统使用一个唯一的描述符来引用已经打开的文件，这个描述符是从所打开文件的元数据到这个文件本身的一种映射。在 Linux 内核的实现中，描述符实际上是一个整数（C 语言的 int 类型），被称为文件描述符（File Descriptor），简称 fd。文件描述符在整个用户空间中共享，并且直接被用户程序使用来访问文件。Linux 系统编程的很大一部分都是由打开、读写、关闭和其它一些对文件描述符的操作组成。

普通文件

我们平常所说的“文件”一般指的是 Linux 的普通文件。普通文件存放着字节数据，被组织到一个线性数组中，这个线性数组被称为字节流。在 Linux 中，对文件没有太多其它的组织或者格式。字节数据可以是任何值，并且它们可能以任何一种形式组织在文件中。在系统级，Linux 也没有强制文件的结构。有一些操作系统，例如 VMS，提供高度结构化的文件，支持记录（records）等概念，Linux 并没有这样做。

文件中的任何字节数据都可以被读取或者写入。这些操作从一个指定的字节开始，也就是文件概念上的“位置”。这个位置叫做 file position 或者文件偏移量。文件位置是内核中文件元数据的核心部分。文件第一次被打开时，这个位置是 0。通常随着字节数据的读取或写入，文件位置也同样在增大。文件位置也可以手动设置为一个给定的值，甚至是超过文件末尾的值。往文件结尾之外的位置写入数据，将导致中间连续的那段字节被填充为 0。通过这种方式在文件结尾之外的位置写入数据是可以的，但在文件开头之前的位置写入数据却是不允许的。这本身就没什么意义，实际上也没有太大的用处。文件位置从 0 开始，它不能为负

数。往文件的中间部分写入数据，将覆盖掉从文件位置开始的那些数据。一般大部分的文件写入操作都在文件的末尾进行。文件位置的最大值只由定义它的 C 语言类型决定，在现代的 Linux 系统中一般是 64 位。

文件的大小通过字节数来计算，被称为文件的长度。换句话说，组成文件的线性数组的字节数就是文件的长度。文件长度可以通过截断操作来改变，文件可以被截断为一个更小的长度，新文件末尾之后的字节数据都将被移除。让人疑惑的是，截断操作也可以使文件的长度变得比原来更大，这种情况下，文件末尾到新长度之间的字节数据将用 0 填充。文件可以是空的，这时候的文件长度为 0，也不包含任何有效的字节。文件的最大长度，和文件位置一样，只由内核中管理文件使用的 C 语言类型大小决定。但特定的文件系统，可能会对文件长度加上限制，文件允许的最大长度会更小。

不同的进程甚至同一个进程，可以多次打开同一个文件。每次打开文件都会得到一个唯一的文件描述符；多个进程之间可以共享文件描述符，这样一个文件描述符就可以被多个进程使用。内核没有对文件的并发访问强加任何限制。多个进程可以同时自由的向同一个文件中读取和写入数据。这种并发访问的结果依赖于每个单独操作的顺序，通常是不可预料的。用户空间程序必须妥善地安排操作的顺序，保证并发访问是同步的。

尽管文件通常是通过文件名来访问，实际上文件并没有直接与文件名关联在一起。文件其实是通过 inode 来引用的，这是一个唯一的数字值，被称为 inode 数值，通常简称为 i-number 或者 ino。inode 结构体保存了文件对应的元数据，例如修改时间戳、拥有者、类型、长度和文件存储的位置（不是文件名）。inode 既是一个物理的对象，存储在 Unix 文件系统中；又是一个概念实体，表示 Linux 内核中的一个数据结构。

目录和链接

通过 inode 数值来访问文件是相当麻烦的（也是一个潜在的安全漏洞），所以用户空间总是通过文件名来打开文件。目录就是用来提供访问文件的名称，实际上就是将可读的名称映射为相应的 inode 数值。名字和 inode 组合起来，就是一个链接。这种映射在磁盘上的物理组成可能是一个简单的表、哈希或者其它形式，内核对每种支持的文件系统，都有相应的代码来实现和管理。概念上来讲，目录和正常的文件看起来是一样的，区别是目录只包含名字到 inode 的映射。内核直接使用这个映射来执行名字到 inode 的转换。

当用户空间应用请求打开一个指定的文件时，内核打开包含文件名的那个目录，并查找给定的名称。通过文件名，内核得到 inode 数值，通过 inode 数值，找到 inode 结构体。inode 包含了文件的元数据，其中就包括文件数据的物理存储位置。

磁盘一开始只有一个根目录，根目录通常表示为路径/。我们知道，一个系统中一般都会有许多目录。那么给定一个名称，内核是怎样查找到对应的目录呢？

前面我们提到过，目录和普通文件是很相似的。实际上，它们的 inode 是相关联的。一个目录下的链接可以指向另一个目录的 inode。这意味着目录可以嵌套在另一个目录下，组成一个目录层次。这样我们就可以使用 Unix 用户熟悉的路径，例如 /home/blackbeard/landscaping.txt。

当我们让内核打开上面这个路径的文件时，内核会依次经过路径中的每一个目录实体，找到下一个入口的 inode。在上面的例子中，内核从/开始，获得 home 的 inode 并进入到那里，然后再获得 blackbeard 的 inode，最终获得 landscaping.txt 的 inode。这个操作被称为目录或路径查找。Linux 内核还使用了缓存机制，叫做 dentry 缓存，用来保存目录查找的结果，提供后续的快速查找。

以根目录开始的路径被认为是完全限定的，称为绝对路径。另外一些路径没有完全限定，

只提供相对于其它目录的路径，这就是相对路径。当指定一个相对路径时，内核会从当前工作目录开始查找路径。例如 todo/plunder，内核会从当前工作目录开始，查找目录 todo，从那里找到 plunder 文件的 inode。

尽管内核对目录的处理和普通文件一样，但内核并不允许像普通文件那样打开和操作目录。实际上，目录必须由一组特殊的系统调用来操作。这些系统调用允许添加和删除链接，目录也只有这两个操作是有意义的。如果用户空间可以不经内核仲裁直接操作目录，那一个简单的错误，就能轻易的破坏整个文件系统。

硬链接

到目前为止，我们所讲的内容都没有提到如何阻止多个名称指向同一个 inode，这是因为允许这样做。当多个链接把不同的名字映射到同一个 inode 时，我们就把它们称为硬链接。

复杂的文件系统结构，允许使用硬链接来将多个路径指向相同的文件数据。硬链接可以在同一个目录下，也可以在不同的目录中。不管哪种情况，内核都只是简单地通过路径查找到正确的 inode。例如，可以从/home/bluebeard/map.txt 和/home/blackbeard/treasure.txt 这两个硬链接查找到同一个 inode，它指向同一块数据。

删除文件其实就是从目录结构中去除链接，简单地从目录中删除名字和 inode 组合即可完成这一操作。由于 Linux 支持硬链接，文件系统并不能在每一次去除链接时销毁相应的 inode 和它所关联的数据。不然如果文件系统其它地方还存在另一个硬链接怎么办？为了确保文件在所有链接都被去除之前不被破坏，每一个 inode 结构体都包含一个链接计数，用来跟踪文件系统中指向它的链接数目。当一个链接被去除时，链接计数就减 1；只有当它到达 0 时，inode 和相关联的数据才会从文件系统中实际地删除。

符号链接

硬链接不能跨越文件系统，因为 inode 数值在自己的文件系统之外是毫无意义的。为了允许跨文件系统的链接，Unix 系统实现了符号链接（简称 symlinks），符号链接比硬链接更简单一点，但也更加地不透明。

符号链接和普通文件差不多，一个符号链接有它自己的 inode 和数据块，在数据块中包含了它所链接的文件的完整路径。这意味着符号链接可以指向任何地方，包括不同文件系统的文件和目录，甚至可以指向并不存在的文件和目录。指向不存在文件的符号链接被称为已破坏的链接（broken link）。

符号链接比硬链接的开销更大，因为有效地查找一个符号链接实际上需要查找两个文件：符号链接本身，然后是链接指向的文件。硬链接并不需要这个额外的开销——访问一个被多次链接到文件系统的文件，和访问只被链接一次的文件没有任何区别。尽管符号链接的额外开销非常小，这仍然是一个负面影响。

符号链接同时也不如硬链接透明。使用硬链接是完全透明的，实际上你很难确定一个文件是否被链接多次，因为链接一次和多次并没有区别。另一方面，操作符号链接却需要特殊的系统调用。这种透明度的缺乏通常被认为是积极的，因此符号链接更多的是用做快捷方式而不是文件系统内部的链接。

特殊文件

特殊文件指的是被表示为文件的内核对象。这么多年以来，Unix 系统已经能够支持几种不同类型的特殊文件。Linux 支持四种：块设备文件、字符设备文件、命名管道和 Unix 域套接字。把某一种抽象对应为文件系统中的特殊文件，这就是“任何事物都是文件”的哲学范式。Linux 提供了一个系统调用来创建特殊文件。

Unix 系统的设备访问由设备文件来完成，它看起来和实际操作起来都和文件系统中的普通文件一样。设备文件可以被打开、读取和写入，允许用户空间访问和控制系统中的设备（不管是物理的还是虚拟的）。Unix 设备一般可以划分为两种类型：字符设备和块设备。每种类型的设备都有自己特殊的设备文件。

字符设备通过线性字节队列来访问，设备驱动把字节一个一个地放置到队列中，然后用户空间按照它们放入队列的顺序一个一个读出来。键盘就是字符设备的典型例子，如果用户输入“peg”，应用程序就会从键盘设备读取到 p，e，最后是 g。当没有更多的字符能够读取时，设备返回文件结束标志（EOF）。丢失某个字符，或者以其它顺序来读取，都没有什么意义。字符设备通过字符设备文件来访问。

块设备则是通过字节数组来访问的，设备驱动把可 seek 的设备映射成字节数组，这样用户空间就可以自由的访问数组中任何有效字节，可以任意顺序地访问数据，例如读取第 12 字节，然后读取第 7 字节，然后再次读取第 12 字节都是允许的。块设备通常都是存储设备，例如硬盘、软盘、CD-ROM 驱动器、U 盘存储器等等。块设备通过块设备文件来访问。

命名管道（通常叫做 FIFO，也就是“first in, first out”）是进程间通信（IPC）的一种机制，通过文件描述符来提供通信通道，命名管道也是通过特殊文件来访问的。普通的管道用来把一个程序的输出输送到另一个程序的输入；它们通过一个系统调用在内存中创建，不会存在于文件系统中。命名管道的行为和普通管道差不多，但它是通过一个叫做 FIFO 的特殊文件来访问的。两个不相关的进程可以通过访问这个特殊文件进行通信。

套接字是最后一种特殊文件，套接字是 IPC 的高级形式，它不仅允许同一台机器上的两个进程之间通信，还允许不同机器上的两个进程之间进行通信。实际上，套接字是网络和因特网编程的核心组成部分。套接字有多个流行的种类，包括用于本地机器通信的 Unix 域套接字。因特网上的套接字通信需要一个主机名和端口号来标识通信的目标，Unix 域套接字使用文件系统中的一个特殊文件来通信，一般这个文件就称为套接字文件。

文件系统和名字空间

Linux 和所有 Unix 系统一样，提供了一个全局和统一的文件和目录名字空间。有一些操作系统则把不同的磁盘划分为不同的名字空间，例如软盘中的某个文件可能通过路径 A:\plank.jpg 来访问，而硬盘驱动器则位于 C:\下。在 Unix 下，软盘中的同一个文件可能可以通过路径/media/floppy/plank.jpg 来访问，也可以通过/home/captain/stuff/plank.jpg 来访问，就好像这个文件和其它存储介质的文件在一起。因此，Unix 的名字空间是统一的。

文件系统就是文件和目录组成的整齐有效的层次结构，文件系统可以单独地从全局名字空间里添加或者删除。这样的操作被称为挂载（mounting）和卸载（unmounting）。每个文件系统都会被挂载到名字空间的一个特定位置，也就是所谓的挂载点，于是这个文件系统的根目录就可以从这个挂载点访问。例如 CD 被挂载到/media/cdrom 下，那 CD 文件系统的根目录就可以从/media/cdrom 访问到。第一个被挂载的文件系统位于名字空间的根位置（/），被称为根文件系统。Linux 系统必须有一个根文件系统，挂载其它的文件系统则是可选的。

文件系统通常是物理存在的（例如存储在磁盘上），但 Linux 同样也支持内存中的虚拟文件系统，还有存在于网络上其它机器的网络文件系统。物理文件系统存在于块存储设备中，例如 CD、软盘、小型闪存卡、或者硬盘。有一些设备是可以分区的，也就是说可以被划分为多个文件系统，每个都可以单独操作。Linux 支持相当多的文件系统类型，用户可能遇到的所有文件系统几乎都能被支持，包括介质相关的文件系统（如 ISO9660）、网络文件系统（NFS）、本地文件系统（ext3）、其它 Unix 系统的文件系统（XFS），和某些非 Unix 系统的文件系统（FAT）等等。

块设备上的最小可寻址单元是扇区，扇区是块设备的一个物理特性。扇区的大小一般都是 2 的指数，512 字节是最常见的。块设备不能移动或者访问小于一个扇区的数据单元，所有的 I/O 操作都是一个或者多个扇区的。

同样，文件系统的最小逻辑可寻址单元是块。块是文件系统的一个抽象概念，而不是文件系统所在的物理介质。通常块的大小都是扇区的某个倍数，这个倍数一般也是 2 的指数。块的大小一般都比扇区要大，但是块的大小绝对不能超过页的大小，页是内存管理中的最小可寻址单元。比较常见的块大小有 512 字节、1K 和 4K。

由于历史的原因，Unix 系统只有一个共享的名字空间，对系统中的所有用户和进程都是可见的。Linux 采用了一个创新的方法，支持单个进程的名字空间，允许每个进程对系统的文件和目录结构有一个唯一的视角。子进程默认继承了父进程的名字空间，进程也可以创建自己的名字空间，使用自己的挂载点和单独的根目录。

进程

如果文件是 Unix 系统最基本的抽象，进程就是第二基本的抽象。进程是执行中的目标代码：动态运行着的程序。但进程不仅仅只是目标代码——它包括数据、资源、状态和虚拟的计算机。

目标代码是一种内核能够理解、机器可以运行的代码格式（Linux 最常见的格式是 ELF），进程则是从目标代码的执行开始拥有生命。可执行的代码包含元数据、多个代码段和数据段。段是可装载到内存中的线性目标代码，段中的所有字节都同样处理，有相同的权限，一般也是实现相同的目的。

最重要和常见的段是 text 段、数据段和 bss 段。text 段包含可执行代码和一些只读的数据（如常量），text 段一般都被标识为只读和可执行的。数据段包含已初始化的数据（如 C 语言的变量），而且一般被标识为可读写的。bss 段包含未初始化的全局数据，由于 C 标准规定全局变量的默认值必须全为 0，目标代码没有必要在磁盘中存储这些 0。实际上，目标代码可以简单地列出 bss 段中所有未初始化的变量，当 bss 段被装载到内存中时，由内核把它映射到一个空页中（全为 0 的页）。bss 段这样设计完全是为了最优化。bss 段的命名是一个历史遗留问题，它代表的意思是 block started by symbol，或者 block storage segment。ELF 可执行代码中其它常见的段有绝对段（包含不可定位符号）和未定义段。

进程同样也拥有许多的系统资源，由内核分配和管理。进程一般都是通过系统调用来请求和操作资源。资源包括定时器、未决信号、打开的文件、网络连接、硬件设备和 IPC 机制。进程的资源、数据、和进程相关的统计信息，都存储在内核中的进程描述符中。

进程是一个虚拟的抽象，Linux 内核支持抢先式多任务和虚拟内存，进程拥有一个虚拟的处理器和一个虚拟的内存镜像。从进程的角度来看，就好像整个系统中只有它自己一个进程。虽然系统可能同时调度多个进程，但每个进程在运行的时候都好像独自控制着整个系统。内核透明无缝的抢占和重新调度进程，使所有运行中的进程共享处理器，进程本身并不知道这一过程。类似地，每个进程都拥有一个单独的线性地址空间，就好像它控制着系统的所有

内存。通过虚拟内存和分页，内核允许系统中同时存在多个进程，每个操作不同的地址空间。内核通过现代处理器提供的硬件支持来管理这种虚拟机制，使得操作系统能够同时管理多个无关进程的状态。

线程

每个进程包含一个或者多个执行线程，线程是处理器的执行单元，是代码执行的抽象，保持着进程的运行状态。

大多数进程只包含一个线程，也就是单线程；而进程中包含多个线程就被称为多线程。由于 Unix 历史悠久的简单性、快速的进程创建、和健壮的 IPC 机制等因素，传统的 Unix 程序都是单线程的，这些因素降低了线程的需要。

每个线程的组成包括一个栈（存储着它自己的本地变量，就像非线程系统中的进程栈一样）、处理器状态、和目标代码的当前执行位置（通常存储在处理器的指令指针中）。进程的其它部分对所有线程都是共享的。

在内部，Linux 内核对线程的实现很独特：它们就是普通的进程，只不过这些进程恰好共享某些资源（特别显著的就是地址空间）。在用户空间，Linux 的线程实现遵循 POSIX 1003.1c 标准（pthreads）。当前 Linux 线程的实现是 Native POSIX Threading Library(NPTL)，是 glibc 的一部分。

进程层次

每个进程都由一个唯一的正整数标识，也就是进程 ID（pid）。第一个进程的 pid 是 1，随后的每个进程都将接收到一个新的唯一的 pid。在 Linux 中，进程之间组成一个严格的层次结构，称为进程树。进程树的根是系统的第一个进程，init 进程，通常也就是 init 程序。通过 fork()系统调用来创建新进程，fork 系统调用复制当前调用进程。原来的进程被称为父进程，新创建的进程被称为子进程。除了系统的第一个进程，每个进程都有一个父进程。如果父进程在子进程之前终止，内核将把子进程提升为父进程。

当一个进程终止时，它并不会立即从系统中移除。相反，Linux 内核在内存中保留了进程的部分，以便父进程能够查询被移除进程的终止状态。这个过程被称为等待已终止进程。一旦父进程完成等待已终止的子进程，子进程就会被完全销毁。一个进程已经终止，但还没有被父进程完成等待，就被称为 zombie。init 进程例行地等待所有子进程，确保任何父进程都没有遗留 zombie 子进程。

用户和组

Linux 通过用户和组来实现授权。每个用户都和一个被称为用户 ID（uid）的唯一正整数相关联。每个进程又严格地与一个 uid 相关联，用来标识运行当前进程的用户，这就是进程的实际 uid。在 Linux 内核内部，一个用户只有 uid 这一个概念。操作系统用户却是通过用户名来引用自己和其它用户的，而不使用数值。用户名和相应的 uid 存储在/etc/passwd 文件中，库程序负责把用户名映射到相应的 uid。

在用户登录的过程中，用户提供用户名和密码给 login 程序。如果给出的是一个合法的用户并且密码正确，login 程序就启动用户的 login shell，然后使 shell 的 uid 等于用户的 uid，login shell 也是在/etc/passwd 中指定。子进程继承了父进程的 uid。

uid 0 与特殊用户 root 相关联，root 用户拥有超级特权，在系统中几乎可以做任何事情。例如，只有 root 用户才能改变进程的 uid。因此，login 程序以 root 运行。

除了实际 uid，每个进程同时还有一个有效 uid、一个已保存 uid、和一个文件系统 uid。实际 uid 总是启动进程的那个用户；有效 uid 在不同规则下有可能改变，允许一个进程以另一个用户的权限运行；已保存 uid 存储原始的有效 uid，它用来确定用户将切换回去哪一个有效 uid；文件系统 uid 通常都和有效 uid 相同，用来验证文件系统访问。

每个用户都可能属于一个或多个组，包括一个主要组或登录组，在/etc/passwd 文件中列出；用户可能还属于多个附加组，在/etc/group 中列出。因此每个进程也与一个相应的组 ID（gid）相关联，同样拥有一个实际 gid、一个有效 gid、一个已保存 gid 和一个文件系统 gid。进程通常与用户的登录组相关联，而不是任何附加组。

可靠的安全检查，允许进程只有在符合某些条件时，才能执行某些特定的操作。历史性的，Unix 对这个决定非常黑白分明：只有 uid 为 0 的进程能够访问，其它进程都不行。目前 Linux 已经用一个更通用的安全系统替换掉它，不再使用简单的二进制检查。现在内核能够基于更加细粒度的设置来决定基本的访问。

权限

Linux 的标准文件权限与安全机制和 Unix 一样。

每个文件都与一个拥有用户、拥有组、一组权限设置位相关联。这些二进制位描述了文件拥有者、拥有组、其它人能否读取、写入或执行文件；这三种访问权限都使用三位来表示，总共 9 个二进制位。拥有者和文件权限都存储在文件的 inode 中。

对普通文件来说，权限是非常明显的：它们指定了文件能否读取、写入和执行。特殊文件的读写权限和普通文件一样，只是读取或者写入的数据取决于不同的特殊文件，特殊文件的执行权限被忽略。对于目录来说，读取权限允许目录的内容被列表出来；写权限允许往目录中添加新的链接；执行权限则允许使用路径名进入到目录中。表 1-1 列出所有 9 个权限位、它们的八进制值（一种流行的表示方式）、它们的文本值（ls 命令的结果）、和各自的含义。

表 1-1. 权限位和相应的值

二进制位	八进制值	文本值	相应的权限
8	400	r--------	拥有者可读
7	200	-w-------	拥有者可写
6	100	--x------	拥有者可执行
5	040	---r-----	组可读
4	020	----w----	组可写
3	010	-----x---	组可执行
2	004	------r--	其它可读
1	002	-------w-	其它可写
0	001	--------x	其它可执行

除了传统的 Unix 权限，Linux 还支持访问控制列表（ACL）。ACL 允许更详细和更精确的权限与安全控制，代价则是增大了复杂度和磁盘存储空间。

信号

信号是一种单向异步通知机制，信号可能从内核发送到进程，从一个进程发送到另一个进程，甚至从一个进程发送给自身。信号通常向进程警报某些事件，例如段错误、或者用户键入 Ctrl-C。

Linux 内核实现了大约 30 种信号（准确的数字与处理器架构相关）。每种信号都由一个数值常量和字符名称来描述。例如在终端被挂起时将发出 SIGHUP 信号，在 i386 架构下它的值是 1。

除了 SIGKILL 信号（它总是终止进程），SIGSTOP 信号（它总是停止进程），进程可以控制接收到信号后的动作。进程可以接受默认动作，根据不同的信号，可能是终止进程、终止并 coredump 进程、停止进程、或者不做任何事情。同样，进程也可以选择明确地忽略或者处理信号。被忽略信号将被默默地抛弃，处理信号将会执行一个用户提供的信号处理函数。程序一接收到信号，就马上跳转到信号处理函数，信号处理函数执行完毕并返回，程序的控制重新回到之前被中断的指令。

进程间通信

允许进程间交换信息、相互进行事件通知是操作系统最重要的工作之一。Linux 内核实现了大部分的传统 Unix IPC 机制——包括那些被 System V 和 POSIX 定义和标准化——也实现了一两个 Linux 自己的 IPC 机制。

Linux 支持的 IPC 机制包括：管道(pipes)、命名管道(named pipes)、信号量(semaphores)、消息队列(message queues)、共享内存(shared memory)、以及快速用户空间互斥(futexes)。

头文件

Linux 系统编程涉及到一些头文件，内核本身和 glibc 提供了系统级编程所需的头文件。这些头文件包括标准 C 头文件（例如<string.h>），以及常见的 Unix 头文件（如<unistd.h>）。

错误处理

毫无疑问，检查和处理错误是极为重要的。在系统编程中，通过函数的返回值来表示错误，并且通过一个特别的变量 errno 来描述。glibc 为库和系统调用透明地提供了 errno 支持。本书讲解的大多数接口都是使用这一机制来传达错误。

函数通过一个特殊的返回值来向调用者通报错误，这个特殊值通常是-1（具体依赖于特定的函数）。错误值警告调用者发生了某个错误，但并没有提供为什么错误会发生的具体原因。errno 变量被用来发现错误的具体原因。

这个变量定义在<errno.h>中，如下：

```
extern int errno;
```

只有紧接着函数指示发生错误，设置 errno 之后，errno 变量才是有效的，因为函数在成功执行的过程中修改了 errno 变量也是合法的。

errno 变量可以直接读取和写入，它是一个可修改的值。errno 的值映射到一串文本，描

述了特定的错误信息。预处理器#define 同样把宏映射到 errno 的数值上。例如，预处理器定义 EACCESS 等于 1，代表“permission denied”，表 1-2 列出了标准定义的所有错误和相应的错误描述。

表 1-2. 错误以及描述

预处理定义	描述	描述
E2BIG	Argument list too long	参数列表太长
EACCESS	Permission denied	权限拒绝
EAGAIN	Try again	重新尝试
EBADF	Bad file number	错误的文件数值
EBUSY	Device or resource busy	设备或资源忙
ECHILD	No child processes	没有子进程
EDOM	Math argument outside of domain of function	数学参数超出函数范围
EEXIT	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	系统调用被中断
EINVAL	Invalid argument	非法参数
EIO	I/O error	I/O 错误
EISDIR	Is a directory	目标是一个目录
EMFILE	Too many open files	打开太多的文件
EMLINK	Too many links	链接太多
ENFILE	File table overflow	文件表溢出
ENODEV	No such device	设备不存在
ENOENT	No such file or directory	文件或目录不存在
ENOEXEC	Exec format error	执行格式错误
ENOMEM	Out of memory	内存不够
ENOSPC	No space left on device	设备无空间
ENOTDIR	Not a directory	不是一个目录
ENOTTY	Inappropriate I/O control operation	不适当的 I/O 控制操作
ENXIO	No such device or address	设备或地址不存在
EPERM	Operation not permitted	操作不允许
EPIPE	Broken pipe	管道已破坏
ERANGE	Result too large	结果太大
EROFS	Read-only file system	只读文件系统
ESPIPE	Invalid seek	非法 Seek 操作
ESRCH	No such process	进程不存在
ETXTBSY	Text file busy	文本文件忙
EXDEV	Improper link	错误的链接

C 库提供了几个函数来把 errno 值转化为相应的文本描述，这些函数在错误报告时才需要使用。检查和处理错误可以直接使用预处理器的定义和 errno 变量。

关于错误处理的第一个函数是 perror()：

```
#include <stdio.h>
void perror (const char *str);
```

这个函数向 stderr（标准错误输出）打印出字符串 str，紧跟着一个冒号，然后打印出 errno 表示的当前错误的字符串描述信息。为了更加有用，执行失败的函数名称应该包含在字符串中，例如：

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

C 库同时还提供了 strerror()和 strerror_r()函数，原型如下：

```
#include <string.h>
char * strerror (int errnum);
```

和

```
#include <string.h>
int strerror_r (int errnum, char *buf, size_t len);
```

前一个函数返回一个字符串指针，描述 errnum 指定的错误，应用程序不能修改这个字符串，但后续的 perror()和 strerror()调用可能会修改它。因此，strerror 函数不是线程安全的。

strerror_r 函数是线程安全的，它填充缓冲区 buf 的 len 长度。strerror_r()调用成功时返回 0；失败返回-1。有趣的是，在错误发生时它也会设置 errno。

对于某些函数来说，返回类型的所有范围都是合法的返回值。在这种情况下，调用函数之前必须首先把 errno 设置为 0，调用函数之后再检查 errno（这类函数在发生错误时只会返回一个非 0 的 errno）。例如：

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

检查 errno 的一个常见错误是：没有意识到每个库函数或系统调用都可以修改 errno。例如，下面的代码就有 bug：

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

如果你需要在多个函数调用后保持 errno 的值，请使用变量保存 errno：

```
if (fsync (fd) == -1) {
    int err = errno;
    fprintf (stderr, "fsync failed: %s\n", strerror (errno));
    if (err == EIO) {
        /* if the error is I/O-related, jump ship */
        fprintf (stderr, "I/O error on %d!\n", fd);
        exit (EXIT_FAILURE);
    }
```

```
}
```

在单线程程序中，errno 是一个全局变量，就像前面所显示的。然而在多线程程序中，errno 被存储在每个线程中，因此也是线程安全的。

开始系统编程

本章着眼于 Linux 系统编程的基础，提供了一个针对程序员的 Linux 系统概述。下一章讨论基本的文件 I/O，包括读取和写入文件。但是由于 Linux 实现了文件的许多接口，文件 I/O 也不仅仅只是关于文件。

我们现在已经拥有足够的初步知识，是开始深入到实际的系统编程的时候了，Let's go！

第二章 文件 I/O

本章讲解读取和写入文件的基本知识，这类操作组成了 Unix 系统的核心。下一章讲解标准 C 库中的标准 I/O，第四章继续讲解更高级和特殊的文件 I/O 接口，第七章讲解文件和目录操作，结束我们对文件 I/O 的讨论。

在文件被读取和写入之前，它必须先被打开。内核为每个进程保存了一个打开文件的列表，被称为文件表（file table）。文件表通过一个非负整数来索引，也就是文件描述符（通常简称为 fd）。列表中的每一个条目都包含一个打开文件的信息，包括一个指针，指向文件 inode 在内存中的一份拷贝，和相应的元数据，例如文件位置和访问模式。用户空间和内核空间的每个进程使用的文件描述符都是独立的。打开文件返回一个文件描述符，随后的操作（读取、写入等等）都使用文件描述符作为主要的参数。

默认情况下，子进程将复制父进程的文件表。打开的文件列表、访问模式、当前文件位置等等都是一样的，但子进程对文件表的更改（例如子进程关闭文件）并不会影响到其它进程。但是，正如你在第五章中将看到的那样，子进程和父进程共享父进程的文件表也是可能的（线程就是这样的）。

文件描述符使用 C 语言 int 类型描述，而没有使用类似于 fd_t 这样的特殊类型，通常都被认为很奇怪，但有其历史原因，这是 Unix 使用的方式。每个 Linux 进程都有一个允许打开文件的最大数量值，文件描述符从 0 开始，增长到比最大数量值小 1。默认情况下，这个最大值是 1024，但最高可以配置到 1,048,576。因为负数值不是合法的文件描述符，返回值为合法文件描述符的函数，通常都使用-1 来指示发生了错误。

每个进程通常都至少有三个已打开的文件描述符：0，1 和 2，除非进程显式地关闭它们。文件描述符 0 是标准输入（stdin），文件描述符 1 是标准输出（stdout），文件描述符 2 是标准错误（stderr）。除了直接使用整数来引用这三个文件描述符，C 库也提供了预处理定义 STDIN_FILENO，STDOUT_FILENO，和 STDERR_FILENO。

注意文件描述符不仅仅只是引用普通文件，根据“任何事物都是文件”的哲学，它们也被用来访问设备文件和管道、目录和 futexes、FIFO、套接字等，任何你能够读取或者写入的东西都是通过文件描述符来访问的。

打开文件

read()和 write()系统调用是访问文件最基本的方法，但在一个文件能够被访问之前，它必须先使用 open()或者 creat()系统调用来打开。一旦文件使用结束，应该使用 close()系统调用来关闭文件。

open()系统调用

使用 open()系统调用来打开文件，获得文件描述符：

```
#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()系统调用把指定路径名的文件映射到一个文件描述符上，成功时将返回该文件描述符。文件位置被设置为 0，打开文件时通过 flags 来指定文件的访问模式。

open()的标志

flags 参数必须是 O_RDONLY，O_WRONLY，或者 O_RDWR 中的一个，这三个参数分别代表只读方式打开文件、只写方式打开文件、和读写方式打开文件。

例如，下面代码以只读方式打开/home/kidd/madagascar：

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

以只写方式打开的文件不能够被读取，反之亦然。发起 open()系统调用的进程必须有足够的权限，才能访问文件。

flags 参数可以和下面的一个或者多个值进行按位或，修改打开文件的行为：

O_APPEND

文件将以附加模式打开，在每一次写入文件之前，文件位置都会被更新到末尾处。即使另一个进程已经向文件写入数据，当前进程发起的每次写入操作也会更新文件位置。(请参考本章后面“附加模式”一节)。

O_ASYNC

当指定的文件可以被读取或者写入时，会产生一个信号（默认是 SIGIO）。O_ASYNC 标记只能用于终端和套接字，普通文件无法使用。

O_CREAT

如果 name 指示的文件不存在，内核将会创建它。如果文件已经存在，O_CREAT 标记没有任何影响，除非同时指定了 O_EXCL 标记。

O_DIRECT

文件将以直接 I/O 方式打开（请参考本章后面“直接 I/O”一节）。

O_DIRECTORY

如果 name 不是一个目录，open()调用将失败。opendir()函数内部就是使用 O_DIRECTORY 标记。

O_EXCL

O_EXCL 与 O_CREAT 标记一起使用时，如果 name 指定的文件已经存在，open()调用将失败。这样使用是为了防止创建文件时的竞争条件。

O_LARGEFILE

这个标记表示将使用 64 位偏移量，允许打开的文件大于 2GB。在 64 位体系架构下，这个标记默认是隐含的。

O_NOCTTY

如果给定的 name 指向终端设备（如/dev/tty），它将不会成为进程的控制终端，即使进程当前并没有一个控制终端。这个标记不经常使用。

O_NOFOLLOW

如果 name 是一个符号链接，调用 open()将失败。一般情况下，链接会被跟随，最终打

开目标文件。如果给定路径的其它部分也包含链接，open()调用仍然会成功。例如，假设 name 为/etc/ship/plank.txt，如果 plank.txt 是符号链接则 open()调用将失败；但即使 etc 或者 ship 都是符号链接，只要 plank.txt 不是，调用仍然会成功。

O_NONBLOCK

如果可能，文件将会以非阻塞模式打开。无论是 open()调用，还是任何其它操作，都不会引起进程的 I/O 阻塞。这个特性可能只有 FIFO 才有定义。

O_SYNC

文件将会以同步 I/O 方式打开，写操作只有在数据被完全写入到物理磁盘之后才会完成。正常的读取操作本身就是同步的，因此这个标记对于读取没有影响。POSIX 为 Linux 定义了额外的 O_DSYNC 和 O_RSYNC，这两个标记和 O_SYNC 是等同的（参考本章后面 O_SYNC 标记一节）。

O_TRUNC

如果文件已存在，而且是一个普通文件，同时也允许写入，O_TRUNC 标记将导致文件被截断为 0 长度。对 FIFO 或者终端设备使用 O_TRUNC 标记会被忽略，使用在其它文件类型上的结果则是未定义的。同时指定 O_TRUNC 和 O_RDONLY 也是未定义的，因为你需要写入权限才能够将文件截断。

例如，下面代码以写入模式打开文件/home/teach/pearl，如果文件已经存在，它将会被截断为 0 长度。因为没有指定 O_CREAT 标记，如果文件不存在，open()调用将失败：

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

新文件的拥有者

决定哪个用户拥有新创建的文件是非常简单的：文件的拥有者 uid 就是创建文件的那个进程的有效 uid。

决定文件的组则更加复杂一些，默认情况下，文件的组会被设置为创建文件的那个进程的有效 gid，这是 System V 定义的行为（大部分 Linux），也是标准 Linux 采用的方式。

然而 BSD 却定义了它自己的方式：文件的组将被设置为父目录的 gid。Linux 可以通过挂载时的选项来获得这个行为——而且如果文件的父目录已经设置了组 ID(setgid)位，Linux 默认就采用这种方式。尽管大多数 Linux 系统都采用 System V 方式（新文件接收创建进程的 gid），BSD 行为（新文件接收父目录的 gid）的可能性也意味着有些时候我们需要通过 chown()系统调用来手动设置组（第七章）。

值得庆幸的是，需要关心文件组的情况并不常见。

新文件的权限

前面给出的两种 open()系统调用的形式都是合法的。除非创建了新文件，mode 参数都会被直接忽略；当指定了 O_CREAT 标记时，mode 参数就是必需的。如果你使用 O_CREAT 时忘记了提供 mode 参数，结果将是未定义的，而且通常都很糟糕——所以千万别忘记了。

在创建文件时，mode 参数提供新创建文件的权限。但 mode 参数并不会限制当前创建

文件的这个 open 调用，因此你可以执行一些带有矛盾性质的操作，例如以写入模式打开（创建）一个文件，但同时又给文件指定只读权限。

mode 参数和 Unix 的权限位类似，例如八进制的 0644（拥有者可以读取和写入、其它人只读）。从技术上来讲，POSIX 允许具体的权限值有特定的实现，允许不同的 Unix 系统以自己的意愿自由地组合权限位。为了弥补这个不可移植性，POSIX 引入了下列常量，通过二进制的“或”操作，组合在一起来指定 mode 参数：

S_IRWXU
　　拥有者有读取、写入和执行权限
S_IRUSR
　　拥有者有读取权限
S_IWUSR
　　拥有者有写入权限
S_IXUSR
　　拥有者有执行权限
S_IRWXG
　　组有读取、写入、执行权限
S_IRGRP
　　组有读取权限
S_IWGRP
　　组有写入权限
S_IXGRP
　　组有执行权限
S_IRWXO
　　其它人有读取、写入、执行权限
S_IROTH
　　其它人有读取权限
S_IWOTH
　　其它人有写入权限
S_IXOTH
　　其它人有执行权限

实际存储在磁盘中的权限位由 mode 参数和用户的文件创建掩码（umask）来确定，计算方法是将 mode 参数和 umask 的补码进行“与”操作。非正式地说，open()调用的 mode 参数将会关闭 umask 中相应的权限位。因此，实际的掩码 022 会导致 mode 参数 0666 最终变成 0644(0666 & ~022)。作为一名系统程序员，你通常在设置权限时不需要考虑 umask——umask 的存在是为了允许用户限制应用程序设置新文件的权限。

例如，下面代码以写入模式打开 file 指定的文件。如果文件不存在，它将被创建，假设 umask 是 022，则文件的权限将会是 0644。如果文件已存在，它将被截断为 0 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)
    /* error */
```

creat()函数

O_WRONLY | O_CREAT | O_TRUNC 的组合是如此常用，有一个专门的系统调用来提供这个功能：

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

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

没错，这个函数名缺少了一个“e”。Unix 的创建者 Ken Thompson 曾经开玩笑说：这个缺失的字母是他设计 Unix 最大的遗憾。

下面是 creat()的典型用法：

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

它等同于下面代码：

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

在大部分 Linux 体系架构下，creat()都是一个系统调用，尽管它也可以简单地在用户空间实现如下：

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

creat()的存在是一个历史遗留问题，那时候 open()函数只有两个参数。今天，为了保持兼容性，creat()系统调用仍然被保留。新的体系架构可以在 glibc 中实现 creat()。

返回值和错误代码

open()和 creat()在成功时都返回一个文件描述符；出错时，两个函数都返回-1，而且会设置 errno 为相应的错误值（第一章讨论了 errno 并且列出了所有的错误值）。处理打开文件的错误并不复杂，因为通常在打开文件之前并没有做太多的工作。典型的处理方法是让用户选择另一个文件，或者简单地终止程序。

使用 read()读取

现在你知道了怎样打开文件，让我们来看看怎样读取它，接下来的一节则是写入。

最基本也是最常用的读取机制是 read()系统调用，定义在 POSIX.1 中：

```
#include <unistd.h>
ssize_t read (int fd, void *buf, size_t len);
```

每次调用 read()将从 fd 指向文件的当前位置开始，读取最多 len 个字节到缓冲区 buf 中。读取成功时，函数返回写入到 buf 中的字节数。出现错误时，调用将返回-1，同时设置 errno。文件位置根据已读取的字节数更新。如果 fd 代表的对象不支持 seek（例如字符设备文件），读取操作总是从当前文件位置开始。

read()的使用很简单，下面例子从文件描述符中读取数据到 word 中，读取的字节数等于 unsigned long 类型的大小，在 32 位 Linux 系统中它是 4 个字节，而在 64 位 Linux 系统中它是 8 个字节。读取完成以后，nr 包含已读取的字节数，出错时它是-1：

```
unsigned long word;
ssize_t nr;

/* read a couple bytes into 'word' from 'fd' */
nr = read (fd, &word, sizeof (unsigned long));
if (nr == -1)
    /* error */
```

这个简单的程序有两个问题：调用可能会在没有读取到 len 个字节就返回；它也可能引起一些错误，代码里并没有检查和处理这些错误。不幸的是，像这样的代码非常普遍，让我们看看如何改进它。

返回值

read()返回一个小于 len 的非负值是合法的，这可能在多种情况下发生：少于 len 字节可读取数据；系统调用可能被信号中断；管道被破坏（如果 fd 是管道）等等。

使用 read()时另一个需要考虑的是函数可能返回 0，read()系统调用返回 0 表示已到文件尾 end-of-file(EOF)；在这种情况下当然不会读取到字节数据。EOF 并不被认为是一个错误（因此函数也就不会返回-1）；它只是表明文件位置已经超过了文件的最后一个合法偏移量，因此文件已经没有任何东西可以读取。如果一个 read()调用读取 len 字节，但文件中并没有足够的字节可供读取，则这个调用将会阻塞（睡眠）直到有足够的字节（假设文件描述符不是以非阻塞模式打开，请参考“非阻塞读取”一节）。注意这种情况和返回 EOF 不同，也就是说“没有数据可以读取”和“到达数据末尾”是不同的。在返回 EOF 的情况下，读取已经到达文件末尾。而在阻塞情况下，读取操作在等待更多的数据——例如读取套接字或者设备文件。

有一些错误是可以恢复的，例如，如果 read()调用还没有读取到任何字节，就被某个信号中断，它将返回-1（如果返回 0 会引起混淆，让人误以为是 EOF），errno 被设置为 EINTR。遇到这种情况，你只需重新发起读取操作即可。

事实上，read()调用可能导致以下结果：

- read()返回 len，所有 len 个读取的字节都存储在 buf 中，这是期望的结果。
- read()返回一个小于 len，但大于 0 的值。已读取的字节存储在 buf 中。导致这种结果的情况有：读取操作中途被信号中断、读取操作中途出现错误、大于 0 但小于 len 个可读字节、读取到 len 个字节之前遇到 EOF。解决的办法是重新发起读取（使用更新后的 buf 和 len）将剩余的字节继续读取到 buf 中，或者指出问题产生的原因。
- read()返回 0。这表示到达 EOF，没有东西可以读取。
- read()阻塞，因为当前没有足够的数据以供读取，在非阻塞模式中不会发生。
- read()返回-1，errno 被设置为 EINTR。这表示在读取到任何字节之前接收到中断信号。这种情况下应该重新发起读取操作。
- read()返回-1，errno 被设置为 EAGAIN。这表示当前没有数据可以读取，read()操作将要被阻塞，应该稍后再发起读取操作。这种情况只发生在非阻塞模式中。
- read()返回-1，errno 被设置为非 EINTR 和 EAGAIN 的值。这表示发生了一个更严重的错误。

读取所有字节

read()函数的这些可能性，说明我们之前写的代码都过于简单，不能够处理所有的错误情况。我们希望能够完整的读取到 len 个字节（至少在到达 EOF 之前），要做到这个，需要一个循环，再加几个条件语句：

```
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;
}
```

这段代码处理了所有五种情况，整个循环语句从 fd 的当前文件位置读取 len 个字节到 buf 中。除非它读取到了所有的 len 个字节，或者遇到 EOF，读取操作会一直进行。如果读取到大于 0 但小于 len 个字节，len 将被减去已读取的数量，buf 则增加已读取的数量，然后重新发起读取操作。如果 read()返回-1，而且 errno 等于 EINTR，则马上重新发起读取操作而不更新参数。如果 read()返回-1，而且 errno 被设置为其它值，则调用 perror()向标准错误输出打印一条错误消息，并且终止循环。

部分读取不仅仅是合法的，而且还很常见。无数的 bug 都源于程序员没有正确地检查和处理读取请求。你最好别加入这个行列！

非阻塞读取

有时候程序员并不希望 read()调用在没有数据可读时阻塞，他们宁愿 read()调用立即返

回，指示出目前没有数据可读。这就是非阻塞 I/O，它允许应用对多个文件执行 I/O 操作，避免 I/O 阻塞，和错过其它文件的可读数据。

因此，需要检查一个额外的 errno 值：EAGAIN。前面已经讨论过，如果给定的文件描述符以非阻塞模式打开（open()调用指定 O_NONBLOCK 参数），当没有数据可以读取时，read()调用会立即返回-1，同时设置 errno 为 EAGAIN，而不会阻塞。当执行非阻塞读取时，你必须检查 EAGAIN 错误，否则仅仅是缺少可读数据就可能会产生严重的错误。例如，你可能使用如下代码：

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

start:
nr = read (fd, buf, BUFSIZ);
if (nr == -1) {
    if (errno == EINTR)
        goto start; /* oh shush */

    if (errno == EAGAIN)
        /* resubmit later */
    else
        /* error */
}
```

本例中使用一条 goto start 处理 EAGAIN 错误实际上没有太大意义——你可以直接不使用非阻塞 I/O。这样使用并没有节省时间，反而引入了更多的循环跳转开销。

其它错误值

其它的错误代码都和编程错误有关、或者是一些底层问题。read()调用失败之后可能的 errno 值包括：

EBADF

给定的文件描述符非法，或者打开模式不是读取。

EFAULT

buf 指向的地址不在调用进程的地址空间内。

EINVAL

文件描述符映射到一个不允许读取的对象上。

EIO

发生了一个底层的 I/O 错误。

read()的大小限制

size_t 和 ssize_t 类型是 POSIX 标准定义的，size_t 类型用来存储字节数大小，ssize_t 是 size_t 的带符号版本（负数值用来报告错误）。在 32 位系统中，它们背后的 C 类型通常分别是 unsigned int 和 int。因为这两个类型经常一起使用，ssize_t 类型潜在地更小范围对 size_t 类型的范围大小也造成了一定的限制。

size_t 类型的最大值是 SIZE_MAX，ssize_t 类型的最大值是 SSIZE_MAX。如果 len 比 SSIZE_MAX 还要大，调用 open()的结果是未定义的。在大多数 Linux 系统中，SSIZE_MAX 就是 LONG_MAX，32 位机器上也就是 0x7FFFFFFF。这对一个单独的读取操作来说通常是足够大的，但无论如何请留意这个情况。如果你使用前面的循环来作为通用读取方法，你可能会像下面这样做：

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

如果 len 为 0，read()调用会立即返回，返回值为 0。

使用 write()写入

写入数据最基本也是最常用的系统调用是 write()，write()和 read()是相对的，二者均在 POSIX.1 标准中定义：

```
#include <unistd.h>
ssize_t write (int fd, const void *buf, size_t count);
```

调用 write()把 buf 指向的 count 个字节数据写入到文件描述符 fd 指向文件的当前文件位置。不支持 seek 操作的文件（例如字符设备）总是从“文件开头”处开始写数据。

write()成功时返回写入的字节数，同时更新文件位置；出错时返回-1，并设置 errno 为相应的错误代码。write()调用可以返回 0，但这个返回值没有任何特殊的意义，它只是简单地表示 0 字节数据被写入。

和 read()类似，write()调用最基本的使用是很简单的：

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

/* write the string in 'buf' to 'fd' */
nr = write (fd, buf, strlen (buf));
if (nr == -1)
    /* error */
```

但是就像 read()一样，这种用法并不非常正确。函数调用方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)
    /* error, check errno */
else if (nr != count)
    /* possible error, but 'errno' not set */
```

部分写入

write()系统调用部分写入的可能性比 read()部分读取的可能性要小，而且 write()调用也不存在 EOF 的条件判断。对于普通文件，write()确保执行完整的写入请求，除非发生了错误。

因此对于普通文件，你不需要使用循环来执行 write。但是其它文件类型（例如套接字）则可能需要使用循环来确保写入了所有的字节数据。使用循环的另一个好处是第二次调用 write()可能会返回错误，这可以显示出第一次 write 调用为什么只有部分写入的原因（虽然这种用法并不常见），下面是例子：

```
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;
}
```

Append 模式

当 fd 以附加模式（O_APPEND）打开时，write 操作不会在文件描述符的当前文件位置进行，而总是在文件的当前结尾位置进行写入。

例如，假设两个进程向同一个文件写入数据。如果不使用附加模式，当第一个进程在文件末尾写入数据，然后第二个进程执行同样的操作。这时候第一个进程的文件位置就不再指向文件的末尾，它指向的是文件的实际末尾减去第二个进程写入的数据数。这就意味着多个进程不允许在没有显式的同步操作下，向同一个文件添加数据，这里存在竞争条件。

附加模式避免了这个问题，它确保文件位置总是设置为文件的末尾，这样所有的写入操作都是附加，即使是多个写入操作。你可以把它想象成每次写入操作之前，都有一个原子的文件位置更新操作。文件位置自动被设置为新写入数据的末尾处，这不会影响到下一次 write()调用，但可能会影响你下一次的读取操作。

附加模式对某些特定的任务有很大的意义，例如更新日志文件；但对于很多其它任务来说，又是不合适的。

非阻塞写入

当 fd 以非阻塞模式（O_NONBLOCK）打开时，原本会阻塞的 write()调用就会立即返回-1，同时设置 errno 为 EAGAIN，写操作应该稍后再次发起。一般来说，普通文件不会发生这种问题。

其它错误代码

其它值得注意的 errno 值包括：

EBADF

给定的文件描述符无效，或者不是以写入模式打开。

EFAULT

buf 指向的地址超出进程的地址空间。

EFBIG

写操作将使文件大于单个进程允许的最大文件限制，或者大于内部实现的大小限制。

EINVAL

给定的文件描述符映射到一个无法写的对象。

EIO

发生低层 I/O 错误。

ENOSPC

给定的文件描述符在文件系统中没有足够的空间。

EPIPE

给定的文件描述符关联的管道或者套接字已经关闭。进程将接收到 SIGPIPE 信号，SIGPIPE 信号的默认动作是终止接收进程。因此，只有进程显式地忽略、阻塞或者处理这个信号，否则进程不会接收到这个 errno 值。

write()的大小限制

如果 count 大于 SSIZE_MAX，调用 write()的结果是未定义的。

如果 count 为 0，调用 write()会立即返回 0。

write()的行为

当调用 write()返回时，内核已经把数据从提供的缓冲区复制到了内核缓冲区中，但并不保证数据已经被写入到了指定的目标。实际上，write 调用一般都很快返回，数据却来不及完全写入到目标。处理器与硬盘之间性能的巨大差别，造成了这种令人痛苦的行为。

实际上，当用户空间程序发起 write()系统调用时，Linux 内核首先会执行一些检查，然后就简单地把数据复制到一个缓冲区。稍后，内核会在后台集合所有这些缓冲区，把它们最佳排序，最终写入到磁盘中（writeback 进程）。这样就使得 write 调用非常快速，几乎是立即返回。内核可以把写入操作推迟更多的空闲周期，把许多写入操作组合起来批处理。

延迟写并没有改变 POSIX 的语义。例如：如果一个读取操作请求刚刚写入的数据，而这部分数据存在于缓冲区中，尚未写入到磁盘，则读取操作将直接从缓冲区中读取，这样就不会导致读取到磁盘中的“过期”数据。这个行为实际上提高了性能，因为读取操作直接从内存缓存而不是磁盘读取数据。读取和写入请求交织在一起时，我们总是能够得到期望的结果，只要系统在数据写到磁盘前没有崩溃！尽管应用程序可能认为写入操作已经成功，但如果发生系统崩溃，数据将无法写入到磁盘。

延迟写的另一个问题是无法确定写入操作的顺序，尽管应用程序可能关心写入请求的顺序，这样才能以指定的顺序写入到磁盘。但内核会以自己的方式来重新排序写入操作，主要

是考虑性能。一般来说所有的缓冲区最终都能够写入到磁盘，只有在系统崩溃时才会导致问题。实际上大多数应用并不关心写入操作的顺序。

延迟写的最后一个问题与 I/O 错误的报告有关。写入磁盘时的任何一个 I/O 错误（例如物理驱动器失败）都无法报告给发起写入请求的进程。实际上，缓冲区和进程没有任何关联。多个进程的数据可能存放在一个单独的缓冲区中，而且进程还可能在写入操作完成但数据尚未写入磁盘时退出。因此无法与写入操作失败的进程进行通信。

内核试图最小化延迟写所带来的风险。为了确保数据能够及时地写入到磁盘，内核制定了一个最大缓冲区时限，所有缓冲区的数据都会在到达这个时间之前被写出去。用户可以通过/proc/sys/vm/dirty_expire_centiseconds 来配置这个值，单位是百分之一秒。

强制文件缓冲区数据写回，甚至使所有写入操作同步都是可能的。这个主题在下一节中讨论：同步 I/O。

本章后面的“深入内核内部”部分将详细讲解 Linux 内核的缓冲区写回子系统。

同步 I/O

尽管同步 I/O 是一个很重要的主题，但我们也不应该过分担心延迟写的相关问题，缓冲写入能够提供巨大的性能提升。因此，任何一个现代的操作系统都通过缓冲实现了延迟写。无论如何，应用程序有时候可能希望控制数据何时写入到磁盘中。Linux 内核提供许多选项，允许牺牲一些性能，来执行同步 I/O 操作。

fsync()和 fdatasync()

确保数据已经写入到磁盘中最简单的方法是使用 fsync()系统调用，它由 POSIX.1b 定义：

```
#include <unistd.h>
int fsync (int fd);
```

fsync()系统调用确保文件描述符 fd 指向的文件相关联的所有数据都被写入磁盘。文件描述符 fd 必须以写入模式打开。这个调用同时写入数据和元数据，例如创建时间戳、以及其它包含在 inode 中的属性。只有当磁盘驱动器显示数据和元数据全部写到磁盘后，fsync()调用才会返回。

在数据被写入在硬盘缓存(cache)的情况下，fsync()也不知道数据是否已经写入到物理磁盘中。磁盘驱动器报告数据已经写入，但实际上数据可能存在于驱动器的写缓存中。幸运的是，磁盘驱动器缓存中的数据会很快提交到磁盘中。

Linux 还提供了 fdatasync()系统调用：

```
#include <unistd.h>
int fdatasync (int fd);
```

这个调用和 fsync()做相同的事情，不过它只刷新数据。fdatasync()不保证元数据也被同步到磁盘中，因此可能会比 fsync 更快。通常调用 fdatasync 就足够了。

两个函数使用方式一样，非常地简单：

```
int ret;
ret = fsync (fd);
if (ret == -1)
```

```
    /* error */
```

两个函数都不保证包含文件的那个目录的任何更新被同步到磁盘中。这意味着如果一个文件链接最近被更新，文件的数据成功写入到磁盘中，但相关联的那个目录可能并没有马上更新，就会表现为文件无法到达。为了确保目录的任何更新同样也被提交到磁盘中，必须为目录打开一个文件描述符，并使用它调用 fsync()。

返回值和错误代码

成功时这两个调用都返回 0；失败时都返回-1，并设置 errno 为下面三个值之一：

EBADF

给定的文件描述符无效，或者不是以写入模式打开。

EINVAL

给定的文件描述符映射到一个不支持同步的对象。

EIO

同步的过程中发生了低层 I/O 错误。这表示一个实际的 I/O 错误，通常类似的错误都是在这里引起。

目前，调用 fsync()可能会失败，因为 fsync()可能没有被文件系统实现，即使 fdatasync 被实现。妄想主义的程序可能会在 fsync()返回 EINVAL 时再尝试 fdatasync()，例如：

```
if (fsync (fd) == -1) {
    /*
    * We prefer fsync(), but let's try fdatasync( )
    * if fsync( ) fails, just in case.
    */
    if (errno == EINVAL) {
        if (fdatasync (fd) == -1)
            perror ("fdatasync");
    } else {
        perror ("fsync");
    }
}
```

由于 POSIX 强制要求 fsync()，而把 fdatasync()标记为可选，任何一个常见的 Linux 文件系统都应该对普通文件实现了 fsync()系统调用。然而特殊的文件类型（多半是那种没有元数据可以同步）和奇怪的文件系统可能仅仅实现了 fdatasync()。

sync()

sync()系统调用的性能稍低一点，但范围更广，用来同步*所有*缓冲区的数据到磁盘中：

```
#include <unistd.h>
void sync (void);
```

sync 函数没有参数，也没有返回值。它总是成功返回，所有缓冲区（包括数据和元数据）

都确保被写入到磁盘中。

标准并没有强制 sync()一定要等待所有缓冲区都刷新到磁盘后才能返回；标准只是要求 sync 调用使进程开始向磁盘提交所有缓冲区。由于这个原因，通常推荐同步多次来确保所有数据都安全到达磁盘。然而 Linux 却总是等到所有缓冲区都提交之后才返回，因此一个 sync()调用就足够了。

sync()唯一实际的用处就是实现 sync 实用工具。应用程序应该使用 fsync()和 fdatasync()来向磁盘提交仅仅必需的文件描述符的数据。注意在繁忙的系统中，sync()可能需要几分钟来完成操作。

O_SYNC 标记

O_SYNC 标记可以传递给 open()调用，表示当前文件上的所有 I/O 操作都应该被同步：

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

read()调用本身总是同步的，如果不同步，那读取到缓冲区中的数据的合法性就是未知的。然而正如前面讨论的，write()调用正常情况下是不同步的。write()调用返回和数据提交给磁盘之间没有直接联系。O_SYNC 标志强制实施这种关联，确保 write()调用执行同步 I/O。

我们可以把 O_SYNC 看成这样一种执行方式：在每一个 write()操作之后，都立即强制执行一个隐式的 fsync()调用，然后再返回结果。实际上这只是语义上的方式，Linux 内核对 O_SYNC 的实现更加高效。

O_SYNC 导致写操作的用户时间和内核时间稍微差一些。此外，根据写入文件的不同大小，O_SYNC 可能导致操作总时间增加一到两个数量级，因为进程会带来大量的 I/O 等待时间（等待 I/O 操作完成的时间）。同步的代价增加非常巨大，因此同步 I/O 只应该在所有其它可能的选择都用完时才使用。

正常情况下，如果应用程序需要确保写入操作的数据到达磁盘，应该使用 fsync()和 fdatasync()。它们产生的代价都比 O_SYNC 小得多，因为它们不需要频繁地调用（只在关键的操作完成之后才调用）。

O_DSYNC 和 O_RSYNC

POSIX 定义了另外两个同步 I/O 相关的 open()标志：O_DSYNC 和 O_RSYNC。在 Linux 系统中，这两个标志定义为 O_SYNC 同义，它们也提供相同的行为。

O_DSYNC 标志指定每个写操作之后只有普通数据被同步，不包括元数据。可以认为就是在每个写请求之后隐式地调用 fdatasync()。由于 O_SYNC 提供更强的保证，即使不显式地支持 O_DSYNC 也不会有功能上的损失；O_SYNC 提供的更强保证仅仅有潜在的性能损失。

O_RSYNC 标志同时指定读和写请求的同步性。它必须与 O_SYNC 或 O_DSYNC 之一同时使用。先前提到过，读取已经是同步的——它们在读取到数据给用户之前是不会返回的。O_RSYNC 标志保证读取操作的所有影响都被同步。这意味着读取产生的元数据更新结果也必须在返回前被写入到磁盘。从实践的角度来讲，read()调用返回之前，在文件访问时间中，

必须更新磁盘中的 inode，获得一份新的 inode 拷贝。Linux 定义 O_RSYNC 和 O_SYNC 一样，尽管这看上去没有什么实际意义（不像 O_SYNC 和 O_DSYNC 那样相关联）。Linux 目前没有办法获得 O_RSYNC 定义的行为，开发者能做的最接近的就是在每个 read()之后调用 fdatasync()。不过实践中确实极少需要使用这种行为。

直接 I/O

Linux 内核和其它现代操作系统内核一样，在设备和应用程序之间实现了一个复杂的缓存、缓冲和 I/O 管理层（请参考本章末尾“深入内核内部”一节）。高性能要求的应用程序可能希望绕过这个层次的复杂性，并且执行自己的 I/O 管理。

创建自己的 I/O 系统通常都是不值得的，但实际上操作系统级的工具可能比应用级程序能获得更好的 I/O 性能。另外，数据库系统常常更喜欢执行自己的缓存，并且尽可能切实可行地最小化操作系统的执行。

open()时提供 O_DIRECT 标志，指示内核最小化 I/O 管理的存在。当使用这个标志时，I/O 将直接从用户空间缓冲区向设备发起，绕过页面缓存。所有 I/O 都是同步的，操作没有完成前不会返回。

当执行直接 I/O 时，操作的请求长度、缓冲区、文件偏移量都必须是底层设备扇区大小的整数倍——通常是 512 字节。在 Linux 内核 2.6 之前，这个限制更加严格：内核 2.4 中，所有这些都必须以文件系统的逻辑块大小（通常 4KB）对齐。为了保持兼容性，应用程序应该与更大的逻辑块大小对齐（可能更不方便）。

关闭文件

程序完成对文件描述符的操作之后，可以解除文件描述符与相应文件的映射，使用 close()系统调用：

```
#include <unistd.h>
int close (int fd);
```

调用 close()解除打开的文件描述符 fd 的映射，并且把进程与文件分离。给定的文件描述符随后不再合法，在接下来的 open()或 creat()调用中，内核可以自由地重用并返回它。close()调用成功时返回 0；错误时返回-1，并适当地设置 errno。使用是简单的：

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

注意关闭文件并不会使文件数据写入到磁盘。如果应用程序希望确保文件被提交到磁盘之后再关闭，则需要使用前面讨论过的“同步 I/O”选项。

关闭文件确实有几个副作用。当与文件相关联的最后一个已打开文件描述符被关闭时，内核中描述该文件的数据结构将被释放。当这个数据结构被释放时，它将拔去文件相关联的 inode 在内存中的拷贝。如果 inode 没有与任何其它相连接，则它可能从内存中释放（它可能逗留的原因是内核为了提高性能缓存 inode）。如果文件从磁盘中被删除链接，但在删除链接前保持打开，则它不会被物理移除，直到它被关闭并且 inode 从内存中移除。因此，调用 close()同样可能导致一个无链接文件最终从磁盘中物理删除。

错误代码

一个常见的错误就是不检查 close()的返回值。这可能导致错过一个至关紧要的错误条件，因为一个延迟操作可能要稍后才能表现出错误，而 close()可以报告这种错误。

close()失败时有许多可能的 errno 值。除了 EBADF（给定的文件描述符无效），最重要的错误值就是 EIO，表示一个低层的 I/O 错误，这个错误很可能与实际的关闭无关。无论报告哪种错误，只要文件描述符是合法的，它就一定会被关闭，并且相关的数据结构会被释放。

尽管 POSIX 标准允许，close()永远不会返回 EINTR。Linux 内核开发人员对此理解得更好——这样实现确实并不聪明。

使用 lseek()

正常情况下，I/O 从头到尾线性地穿过文件，读取和写入需要使用隐式的 seek 来更新文件位置。然而有一些应用，需要在文件中到处自由跳转。lseek()系统调用设置指定的文件描述符的文件位置为一个给定的值。除了更新文件位置，lseek()不执行任何其它动作，也不会发送任何 I/O 请求。

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

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

lseek()的行为依赖于 origin 参数，它可以是下列值之一：

SEEK_CUR

fd 的文件位置将被设置为当前文件位置加上 pos，pos 可以是负数、0 或者正数。pos 为 0 时返回当前文件位置。

SEEK_END

fd 的文件位置将被设置为文件的长度加上 pos，pos 可以是负数、0 或者正数。pos 为 0 时设置偏移量为文件的末尾。

SEEK_SET

fd 的文件位置将被设置为 pos。pos 为 0 时设置偏移量指向文件的开始位置。

lseek()调用成功时返回新的文件位置；出错则返回-1，并且设置 errno 为相应的错误码。例如，下面代码设置文件位置为 1825：

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

下面则是设置 fd 的文件位置为文件末尾：

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

由于 lseek 返回更新后的文件位置，通过 SEEK_CUR 加上 0 就可以得到当前文件位置：

```
int pos;
pos = lseek (fd, 0, SEEK_CUR);
if (pos == (off_t) -1)
    /* error */
else
    /* 'pos' is the current position of fd */
```

到目前为止，lseek 最常见的用途是寻找文件的开头、结尾位置，或者确定指定文件描述符的当前文件位置。

Seek 越过文件末尾

使用 lseek()更新文件指针，使其超过文件末尾是可以的。例如，下面代码对 fd 进行 seek 到文件末尾之后的 1688 字节处：

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

seek 超过文件末尾时，lseek 函数本身并不做什么事情——读取新文件位置将会返回 EOF。但是如果随后对该位置进行写入，则原始文件长度和新文件长度之间的那段空间会被创建，并填充为 0。

用 0 填充被称为“洞”，在 Unix 风格的文件系统中，“洞”并不占用任何物理磁盘空间。这意味着某个文件系统的所有文件的总大小，加起来可能会比物理磁盘的大小更大。有“洞”的文件则被称为“稀疏文件”，“稀疏文件”可以节省相当可观的空间，而且能增强性能，因为操作“洞”不会发起任何物理 I/O 操作。

读取文件“洞”中的部分数据，将会返回正确数量的二进制 0 字节数组。

错误代码

出错时 lseek()返回-1，并设置 errno 为下面四个错误值之一：

EBADF

给定的文件描述符无效，没有指向一个打开的文件。

EINVAL

origin 参数的值不是 SEEK_SET、SEEK_CUR 和 SEEK_END，或者执行 lseek()之后文件位置为负数。EINVAL 表示的这两种错误都很不幸，前者几乎肯定是编译时的程序错误，而后者则代表更严重的运行时逻辑错误。

EOVERFLOW

seek 之后的文件偏移量不能用 off_t 来表示。这个错误只会在 32 位架构下产生。通常文件位置已经更新，这个错误只是表示函数无法返回这个更新后的值。

ESPIPE

给定的文件描述符关联的对象不可 seek，例如管道、FIFO 或者套接字。

限制

最大的文件位置受到 off_t 类型大小的限制。大多数机器架构定义 off_t 为 C 语言 long 类型，long 在 Linux 里总是字大小（通常也就是机器通用寄存器的大小）。但是在内核的内部实现中，偏移量却是以 C 语言的 long long 类型存储。这在 64 位机器上没有问题，但却意味着 32 位机器在执行相对 Seek 时可能产生 EOVERFLOW 错误。

定位读取和写入

在需要使用 lseek()的场合，Linux 提供 read()和 write()系统调用的两个变体，它们分别接受一个文件位置参数并从那里读取或者写入。操作完成后，它们并不更新文件位置。

定位读取是 pread()：

```
#define _XOPEN_SOURCE 500
#include <unistd.h>

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

这个调用从文件描述符 fd 指向文件的文件位置 pos 处开始，读取 count 字节到 buf 中。

定位写入是 pwrite()：

```
#define _XOPEN_SOURCE 500
#include <unistd.h>

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

这个调用从文件描述符 fd 指向文件的文件位置 pos 处开始，把 buf 中的 count 字节写入到文件中。

这两个调用的行为和它们不带“p”的兄弟几乎一样，除了它们完全忽略文件的当前文件位置，它们使用 pos 提供的值而不是当前文件位置。同样，当操作完成后，它们不会更新文件位置。换句话说，任何混杂的 read()和 write()调用都可能潜在地破坏定位读写的结果。

两个定位调用都只能用于可 Seek 的文件描述符。它们提供的语义类似于在 read()或者 write()之前调用 lseek()，但有三个不同点：首先，这两个调用更加容易使用，特别是当你做类似于在文件中向后或者随机移动这种操作；其次，它们在操作完成后并不更新文件位置；最后也是最重要的，它们避免了使用 lseek()时可能存在的任何潜在竞争条件。由于线程共享文件描述符，同一个程序的另一个线程可能在第一个线程调用 lseek()之后、执行读取和写入操作之前，再次更新文件位置。这一类的竞争条件可以通过使用 pread()和 pwrite()来避免。

错误代码

成功时，两个调用都返回读取或者写入的字节数，pread()返回 0 表示到达 EOF；pwrite()返回 0 则表示调用并没有写入任何东西。错误时，两个调用均返回-1，并设置 errno。对于

pread()，任何 read()或者 lseek()的错误值都有可能产生；对于 pwrite()，任何 write()或者 lseek()的错误值都有可能产生。

截短文件

Linux 提供两个系统调用来截短文件的长度，这两个调用都由各种 POSIX 标准定义和强制要求（一定程度上）。它们是：

```
#include <unistd.h>
#include <sys/types.h>
int ftruncate (int fd, off_t len);
```

和：

```
#include <unistd.h>
#include <sys/types.h>
int truncate (const char *path, off_t len);
```

这两个系统调用将指定的文件的长度截短为 len。ftruncate()系统调用对 fd 指定的文件进行操作，必须以写入方式打开。truncate()系统调用则对路径名 path 指定的文件进行操作，文件必须具有写入权限。二者成功时均返回 0；出错时返回-1，并设置 errno 值。

这两个系统调用最常见的用途是把一个文件截短为比当前长度更小。当调用成功返回时，文件的新大小将是 len。原先那些在 len 和文件末尾之间的数据被丢弃，读取请求不能再访问它们。

这两个函数还能用来“截短”文件为更大的长度，类似于前面“Seek 越过文件末尾”中讨论过的 seek 和 write 结合使用。文件新增的字节将被填充为 0。

这两个调用都不更新当前文件位置。

例如，假设文件 pirate.txt 长度为 74 字节，文件内容如下：

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

在相同目录下，运行下面程序：

```
#include <unistd.h>
#include <stdio.h>
int main( )
{
        int ret;
        ret = truncate ("./pirate.txt", 45);
        if (ret == -1) {
                perror ("truncate");
                return -1;
        }
        return 0;
}
```

将导致文件长度变为 45 字节，内容截短为：

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

Multiplexed(多路) I/O

应用程序常常需要对多个文件描述符进行阻塞，在键盘输入(stdin)、进程间通信、以及其它一些文件之间杂耍。现代的事件驱动图形用户界面(GUI)应用则可能需要通过它们的主循环，来处理几百个未决的事件请求。

在没有线程帮助的情况下（本质上是单独地维护每一个文件描述符），一个单独的进程不能够同时对多个文件描述符阻塞。使用多个文件描述符是可以的，只要它们总是准备好被读取或者写入。但是一旦遇到某个文件描述符尚未准备好（例如，发起一个 read()系统调用，但却无任何数据可读），进程就会阻塞，从而不能够再为其它文件描述符服务。进程可能只会阻塞几秒钟，使得应用程序低效并且惹恼用户；然而如果文件描述符一直没有数据可用，进程将永远阻塞。由于文件描述符的 I/O 通常都是相关的（考虑管道），很有可能一个文件描述符在完成服务之前，另一个描述符将一直无法准备好。特别是网络应用程序，可能会同时打开许多套接字连接，这就可能是个大问题。

假设进程间通信时进程阻塞在某个文件描述符，而这时候 stdin 进来了未决的数据。应用程序将察觉不到键盘输入未决，直到已阻塞的 IPC 文件描述符最终返回数据——但如果已阻塞的操作永远不返回呢？

本章前面，我们考虑使用非阻塞 I/O 作为这个问题的解决办法。使用非阻塞 I/O，应用程序可以发起只会返回特殊的错误代码而永远不会阻塞的 I/O 请求。但是这种解决办法很低效，有两个原因：首先，进程需要以某种顺序不断地重新发起 I/O 操作，等待某个打开的文件描述符准备好 I/O，这是一种很糟糕的程序设计；其次，如果程序能够适当地睡眠，则系统会更加高效，释放处理器以处理其它任务，只有当进程的一个或多个文件描述符准备好执行 I/O 时才唤醒进程。

下面开始多路 I/O。

多路 I/O 允许应用程序并发地阻塞在多个文件描述符上，当其中任何一个描述符准备好读取或者写入时，进程都会接收到通知并停止阻塞。多路 I/O 因此成为应用程序的轴心部分，类似于下面这样设计：

1. 多路 I/O：当任何一个文件描述符准备好 I/O 时告诉我。
2. 睡眠直到一个或者多个文件描述符准备好。
3. 唤醒：哪个准备好了？
4. 处理所有准备好 I/O 的文件描述符，不阻塞。
5. 回到步骤 1，重新再来

Linux 提供三个多路 I/O 解决办法：select、poll、和 epoll 接口。我们先在这里讲解前面两个，最后一个是 Linux 特定的高级解决方案，留到第四章。

select()

select()系统调用提供一个机制来实现同步多路 I/O：

```
#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()将阻塞，直到指定的文件描述符准备好执行 I/O，或者可选参数 timeout 指定的时间已经过去。

监视的文件描述符分为三类 set，每一种对应等待不同的事件。readfds 中列出的文件描述符被监视是否有数据可供读取（如果读取操作完成则不会阻塞）。writefds 中列出的文件描述符则被监视是否写入操作完成而不阻塞。最后，exceptfds 中列出的文件描述符则被监视是否发生异常，或者无法控制的数据是否可用（这些状态仅仅应用于套接字）。这三类 set 可以是 NULL，这种情况下 select()不监视这一类事件。

select()成功返回时，每组 set 都被修改以使它只包含准备好 I/O 的文件描述符。例如，假设有两个文件描述符，值分别是 7 和 9，被放在 readfds 中。当 select()返回时，如果 7 仍然在 set 中，则这个文件描述符已经准备好被读取而不会阻塞。如果 9 已经不在 set 中，则读取它将可能会阻塞（我说可能是因为数据可能正好在 select 返回后就可用，这种情况下，下一次调用 select()将返回文件描述符准备好读取）。

第一个参数 n，等于所有 set 中最大的那个文件描述符的值加 1。因此，select()的调用者负责检查哪个文件描述符拥有最大值，并且把这个值加 1 再传递给第一个参数。

timeout 参数是一个指向 timeval 结构体的指针，timeval 定义如下：

```
#include <sys/time.h>
struct timeval {
    long tv_sec;        /* seconds */
    long tv_usec;       /* microseconds */
};
```

如果这个参数不是 NULL，则即使没有文件描述符准备好 I/O，select()也会在经过 tv_sec 秒和 tv_usec 微秒后返回。当 select()返回时，timeout 参数的状态在不同的系统中是未定义的，因此每次调用 select()之前必须重新初始化 timeout 和文件描述符 set。实际上，当前版本的 Linux 会自动修改 timeout 参数，设置它的值为剩余时间。因此，如果 timeout 被设置为 5 秒，然后在文件描述符准备好之前经过了 3 秒，则这一次调用 select()返回时 tv_sec 将变为 2。

如果 timeout 中的两个值都设置为 0，则调用 select()将立即返回，报告调用时所有未决的事件，但不等待任何随后的事件。

文件描述符 set 不会直接操作，一般使用几个助手宏来管理。这允许 Unix 系统以自己喜欢的方式来实现文件描述符 set。但大多数系统都简单地实现 set 为位数组。FD_ZERO 移除指定 set 中的所有文件描述符。每一次调用 select()之前都应该先调用它。

```
    fd_set writefds;
    FD_ZERO(&writefds);
```

FD_SET 添加一个文件描述符到指定的 set 中，FD_CLR 则从指定的 set 中移除一个文件描

述符：

```
FD_SET(fd, &writefds); /* add 'fd' to the set */
FD_CLR(fd, &writefds); /* oops, remove 'fd' from the set */
```

设计良好的代码应该永远不使用 FD_CLR，而且实际情况中它也确实很少被使用。

FD_ISSET 测试一个文件描述符是否指定 set 的一部分。如果文件描述符在 set 中则返回一个非 0 整数，不在则返回 0。FD_ISSET 在调用 select()返回之后使用，测试指定的文件描述符是否准备好相关动作：

```
if (FD_ISSET(fd, &readfds))
    /* 'fd' is readable without blocking! */
```

因为文件描述符 set 是静态创建的，它们对文件描述符的最大数目强加了一个限制，能够放进 set 中的最大文件描述符的值由 FD_SETSIZE 指定。在 Linux 中，这个值是 1024。本章后面我们还将看到这个限制的衍生物。

返回值和错误代码

select()成功时返回准备好 I/O 的文件描述符数目，包括所有三个 set。如果提供了 timeout，返回值可能是 0；错误时返回-1，并且设置 errno 为下面几个值之一：

EBADF

给某个 set 提供了无效文件描述符。

EINTR

等待时捕获到信号，可以重新发起调用。

EINVAL

参数 n 为负数，或者指定的 timeout 非法。

ENOMEM

不够可用内存来完成请求。

select()例子

让我们考虑一个琐细却有完全功能的示例程序，来说明 select()的用法。这个例子阻塞等待 stdin 标准输入 5 秒。由于只监视一个单独的文件描述符，这个例子实际上不是多路 I/O，但是它非常清晰地演示了系统调用的使用方法：

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

#define TIMEOUT 5 /* select timeout in seconds */
#define BUF_LEN 1024 /* read buffer in bytes */

int main (void)
{
    struct timeval tv;
```

```
fd_set readfds;
int ret;

/* Wait on stdin for input. */
FD_ZERO(&readfds);
FD_SET(STDIN_FILENO, &readfds);

/* Wait up to five seconds. */
tv.tv_sec = TIMEOUT;
tv.tv_usec = 0;

/* All right, now block! */
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;
}

/*
* Is our file descriptor ready to read?
* (It must be, as it was the only fd that
* we provided and the call returned
* nonzero, but we will humor ourselves.)
*/
if (FD_ISSET(STDIN_FILENO, &readfds)) {
    char buf[BUF_LEN+1];
    int len;

    /* guaranteed to not block */
    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()实现可移植睡眠

由于 select()在不同的 Unix 系统都有实现，而各个 Unix 系统对秒级睡眠的机制不如 select()统一，所以 select()常常被采用来实现可移植的睡眠。给 select()调用提供一个非 NULL 的 timeout 参数，但其它三个 set 都使用 NULL 即可：

```
struct timeval tv;
tv.tv_sec = 0;
tv.tv_usec = 500;

/* sleep for 500 microseconds */
select (0, NULL, NULL, NULL, &tv);
```

当然 Linux 提供了更高精度的睡眠接口，我们将在第十章讲解。

pselect( )

select()系统调用最早在 4.2BSD 中引入，已经非常流行。但 POSIX 还是定义了自己的方案 pselect()，在 POSIX 1003.1g-2000 和后来的 POSIX 1003.1-2001 中定义：

```
#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()和 select()之间有三个区别：

1. pselect()的 timeout 参数使用 timespec，而不是 timeval 结构体。timespec 结构体使用秒和纳秒，而不是秒和微秒，理论上能够提供更高精度的时间控制。但实际上，二者甚至都不能可靠地提供微秒级精度。

2. 调用 pselect()并不会修改 timeout 参数。因此，timeout 参数在随后的 pselect()调用时不需要重新初始化。
3. select()系统调用没有 sigmask 参数，这个参数和信号有关。当 sigmask 参数为 NULL 时，pselect()和 select()的行为是一致的。

timespec 结构体定义如下：

```
#include <sys/time.h>
struct timespec {
    long tv_sec;        /* seconds */
    long tv_nsec;       /* nanoseconds */
};
```

Unix 工具箱增加 pselect()的主要动机是增加 sigmask 参数，试图解决等待文件描述符和信号之间的竞争条件（信号将在第九章深入讲解）。假设一个信号处理器设置了一个全局标志（大多数情况下都需要这样做），然后进程在调用 select()之前检查这个标志。现在假设信号在进程检查之后，但调用 select()之前到达，则应用可能阻塞不确定，而且永远不响应所设置的标志。pselect()调用解决了这个问题，允许应用程序提供一组信号来阻塞。已阻塞的信号将不会被处理，直到它们被唤醒。一旦 pselect()返回，内核就恢复原有的信号掩码。请参考第九章。

直到内核 2.6.16，Linux 对 pselect()的实现都不是系统调用，而仅仅是对 select()的一个简单封装，由 glibc 提供。这个封装最小化（但并没有完全消除）竞争条件产生的风险。引入一个真实的系统调用之后，竞争条件最终消除了。

虽然 pselect()对 select()有一定的改进（相对很小），但大多数应用程序仍然继续使用 select()，要么出于习惯，要么就是打着可移植性更好的名义。

poll()

poll()系统调用是 System V 的多路 I/O 解决方案。它解决了 select()的几个不足，尽管 select()仍然经常使用（多数还是出于习惯，或者打着可移植的名义）：

```
#include <sys/poll.h>
int poll (struct pollfd *fds, unsigned int nfds, int timeout);
```

和 select()不一样，poll()没有使用低效的三个基于位的文件描述符 set，而是采用了一个单独的结构体 pollfd 数组，由 fds 指针指向这个组。pollfd 结构体定义如下：

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

struct pollfd {
    int fd; /* file descriptor */
    short events; /* requested events to watch */
    short revents; /* returned events witnessed */
};
```

每一个 pollfd 结构体指定了一个被监视的文件描述符，可以传递多个结构体，指示 poll()监视多个文件描述符。每个结构体的 events 域是监视该文件描述符的事件掩码，由用户来设置这个域。revents 域是文件描述符的操作结果事件掩码。内核在调用返回时设置这个域。

events 域中请求的任何事件都可能在 revents 域中返回。合法的事件如下：

POLLIN

有数据可读。

POLLRDNORM

有普通数据可读。

POLLRDBAND

有优先数据可读。

POLLPRI

有紧迫数据可读。

POLLOUT

写数据不会导致阻塞。

POLLWRNORM

写普通数据不会导致阻塞。

POLLWRBAND

写优先数据不会导致阻塞。

POLLMSG

SIGPOLL 消息可用。

此外，revents 域中还可能返回下列事件：

POLLER

指定的文件描述符发生错误。

POLLHUP

指定的文件描述符挂起事件。

POLLNVAL

指定的文件描述符非法。

这些事件在 events 域中无意义，因为它们在合适的时候总是会从 revents 中返回。使用 poll()和 select()不一样，你不需要显式地请求异常情况报告。

POLLIN | POLLPRI 等价于 select()的读事件，POLLOUT | POLLWRBAND 等价于 select()的写事件。POLLIN 等价于 POLLRDNORM | POLLRDBAND，而 POLLOUT 则等价于 POLLWRNORM。

例如，要同时监视一个文件描述符是否可读和可写，我们可以设置 events 为 POLLIN | POLLOUT。在 poll 返回时，我们可以检查 revents 中的标志，对应于文件描述符请求的 events 结构体。如果 POLLIN 事件被设置，则文件描述符可以被读取而不阻塞。如果 POLLOUT 被设置，则文件描述符可以写入而不导致阻塞。这些标志并不是互斥的：它们可能被同时设置，表示这个文件描述符的读取和写入操作都会正常返回而不阻塞。

timeout 参数指定等待的毫秒数，无论 I/O 是否准备好，poll 都会返回。timeout 指定为负数值表示无限超时；timeout 为 0 指示 poll 调用立即返回并列出准备好 I/O 的文件描述符，但并不等待其它的事件。这种情况下，poll()就像它的名字那样，一旦选举出来，立即返回。

返回值和错误代码

成功时，poll()返回结构体中 revents 域不为 0 的文件描述符个数；如果在超时前没有任何事件发生，poll()返回 0；失败时，poll()返回-1，并设置 errno 为下列值之一：

EBADF

一个或多个结构体中指定的文件描述符无效。

EFAULT

fds 指针指向的地址超出进程的地址空间。

EINTR

请求的事件之前产生一个信号，调用可以重新发起。

EINVAL

nfds 参数超出 PLIMIT_NOFILE 值。

ENOMEM

可用内存不足，无法完成请求。

poll()示例

让我们来看一个例子程序，使用 poll()同时检测 stdin 是否可读、以及 stdout 是否可写：

```
#include <stdio.h>
#include <unistd.h>
#include <sys/poll.h>
#define TIMEOUT 5 /* poll timeout, in seconds */

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

        /* watch stdin for input */
        fds[0].fd = STDIN_FILENO;
        fds[0].events = POLLIN;

        /* watch stdout for ability to write (almost always true) */
        fds[1].fd = STDOUT_FILENO;
        fds[1].events = POLLOUT;

        /* All set, block! */
        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;
    }
```

运行它，我们就得到期望的结果如下：

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

再次运行它，这次重定向一个文件到标准输入，我们就能看到两个事件：

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

如果我们在真实的应用中使用 poll()，则不需要每次调用时都重新构造 pollfd 结构体。同一个结构体可以重复传递给多次 poll()调用，内核会根据需要对 revents 域进行清零。

ppoll()

Linux 提供了 poll()的一个堂兄弟调用：ppoll()，就像 pselect()和 select()那样。然而和 pselect()不同的是：ppoll()是 Linux 特定的接口：

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

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

和 pselect()一样，timeout 参数以秒、纳秒指定超时，sigmask 参数则提供一组需要等待的事件信号。

poll()对比 select()

尽管它们执行的工作基本相同，基于下面这些理由，poll()系统调用比 select()更高级：

- poll()不需要用户计算并传递最大文件描述符的值加 1 的那个参数。
- poll()对于大的文件描述符更加高效。设想通过 select()来监视单独的一个值为 900 的文件描述符，内核不得不检查传递进来的 set 的每一位，总共 900 位。
- select()的文件描述符 set 的大小是静态的，这就引入了一个折衷：要么它比较小，限制了 select()可以监视的最大文件描述符；要么它就不够高效。对于巨大的掩码进行操作是低效的，特别是当不确定它们是否被稀疏地组装时。使用 poll()，可以准确地创建适当大小的数组。如果只监视一项？那就只传递一个结构体就可以。
- 使用 select()时，每次返回都会重新构造文件描述符 set，所以下一次调用必须首先重新初始化。poll()系统调用可以分离输入（events 域）和输出（revents 域），允许

数组不经修改而再次使用。

- select()的 timeout 参数在返回后未定义。可移植的代码需要重新初始化 timeout，但是使用 pselect()不存在这个问题。

不过 select()系统调用也确实有一些自己的优点：

- select()可移植性更好，因为某些 Unix 系统并不支持 poll()。
- select()提供了更好的超时 timeout 精度：微秒级。ppoll()和 pselect()理论上提供纳秒级精度，但实际上它们两个甚至都无法提供可靠的微秒精度。

比 poll()和 select()更高级的是 epoll 接口，这是一个 Linux 特定的多路 I/O 解决方案，我们将在第四章讨论。

深入内核内部

这一节考虑 Linux 内核如何实现 I/O，集中于内核的三个主要子系统：虚拟文件系统(VFS)、页面缓存(page cache)、和页面写回(page writeback)。这三个子系统共同帮助 I/O 实现无缝、高效和最优化。

我们将在第四章讨论第四个子系统：I/O 调度。

虚拟文件系统

虚拟文件系统有时候也叫虚拟文件转换，它是一个抽象机制，允许 Linux 内核调用文件系统函数和操作文件系统数据，而不需要知道（甚至完全不关心）所使用的文件系统的特定类型。

VFS 通过提供一个公共文件模型来实现这个抽象，公共文件模型是 Linux 中所有文件系统的基础。通过函数指针和各种面向对象技术的实践（没错，是 C 语言），公共文件模型提供了一个框架，Linux 内核中的文件系统必须遵循这个框架。这就允许 VFS 向文件系统提出通用的请求。框架提供钩子支持读取、创建链接、同步等等操作。然后每个文件系统对自己能够处理的操作注册相应的函数。

这种方式强制各个文件系统之间必须有相当数量的公共性。例如，VFS 一般与 inodes、superblocks 和目录项进行对话。对于那些非 Unix 起源的文件系统，可能完全没有类似 Unix 中 inode 这样的概念，但又不得不处理这些文件系统。实际上，Linux 已经支持像 FAT 和 NTFS 这一类的文件系统。

VFS 带来的好处是巨大的：一个单独的系统调用就可以读取任何媒介上的任何文件系统；一个单独的工具就可以从任何一个文件系统向另一个文件系统复制文件。所有文件系统都支持相同的概念、相同的接口、和相同的调用。一切都能工作，而且工作得很好。

当一个应用发起 read()系统调用时，它会经历一个有趣的旅程。C 库提供系统调用的定义，并在编译时期把系统调用转换为适当的 trap（陷阱）语句。一旦用户空间进程被内核捕获，就会传递系统调用处理器，对 read()系统调用进行处理，内核会判断出给定文件描述符所代表的对象。然后内核再调用那个对象的 read 函数。对于文件系统，这个 read 函数是文件系统代码的一部分。接下来不同的 read 函数做自己的事情（例如，从文件系统物理读取数据）并返回数据给用户空间 read()调用，read()调用再返回给系统调用处理器，处理器又把数据还给用户空间，最后 read()系统调用在这里返回，进程继续往下执行。

对于系统程序员来说，VFS 的分支是很重要的。程序员不需要担心文件系统的类型，或者文件的存储介质。通用的系统调用——read()、write()等等——可以操作任何媒介上的任何文件系统中的文件，只要这个文件系统被支持。

页面缓存

页面缓存是对最近访问过的磁盘文件系统数据在内存中的一份缓存。访问磁盘是相当慢的，特别是相对于今天的处理器速度来说。把请求过的数据存储在内存中，当随后再次请求相同的数据时，内核就可以直接使用内存中的数据，从而避免重复访问磁盘。

页面缓存使用了时间局部性原理，也就是说在某个时间被访问过的资源，短时间内被再次访问的可能性非常高。缓存消耗的内存在它第一次被访问时就已经还清，因为它防止了昂贵的磁盘访问。

页面缓存是内核查找文件系统数据的第一个地方，只有当缓存中找不到需要的数据时，内核才会调用存储子系统来从磁盘中读取数据。任何数据第一次被读取时，都会从磁盘转移到页面缓存中，然后再从缓存返回给应用。如果随后再次读取这部分数据，就直接从缓存中返回。页面缓存的所有操作都是透明的，确保它的数据总是正确的。

Linux 的页面缓存在大小上是动态的，随着 I/O 操作将越来越多的数据读取到内存中，页面缓存也增长得越来越大，消耗任何可用的内存。如果页面缓存最终消耗了所有可用内存，然后又有内存分配请求额外的内存，则页面缓存会被剪除（pruned），并释放最少使用的那些页，为“真正”的内存使用腾出空间。剪除操作无缝而且自动地发生。动态大小页面缓存允许 Linux 使用系统中的所有内存，并缓存尽可能多的数据。

但是通常来说，如果把很少使用的数据块交换到磁盘中，会比剪除一个经常使用的页面缓存更有意义，因为下一个读取请求很可能就会把这个页重新读取到内存中（交换允许内核在磁盘中存储数据，这样能够使用比机器 RAM 更大的内存空间）。Linux 内核实现了 heuristic 来平衡交换数据和剪除页面缓存（以及内存保留容量）。在需要剪除页面缓存时由 heuristic 决定是否把数据交换到磁盘中，特别是当数据没有被使用时。

交换和缓存的权衡通过/proc/sys/vm/swappiness 来调节。这个虚拟文件有一个 0 到 100 的值，默认值为 60。越大的值意味着更加倾向于在内存中保留尽可能多的页面缓存，因此交换也更加频繁。更小的值则意味着倾向于剪除页面缓存而不交换数据。

局部性原理的另一种形式是顺序局部性，也就是说数据通常会被顺序地访问。为了利用这个原理，内核同样实现了预读页面缓存。预读就是在每次执行读取操作之后，继续从磁盘中读取一定的额外数据到页面缓存中。当内核从磁盘中读取一块数据时，它同时读取接下来的一个或者两个块。一次读取大量的顺序块数据是非常高效的，因为这时候磁盘通常不需要寻址。此外，内核可以在进程正在操作第一块数据时完成预读请求。如果进程继续发起新的读取操作，请求随后的数据块（这经常发生），内核就可以把最初预读的数据移交过来，而不用发起实际的磁盘 I/O 操作。

和页面缓存一样，内核动态地管理预读。如果内核注意到某个进程持续地使用预读的数据，内核会自动增大预读窗口，从而预读越来越多的数据。预读窗口可能小到 16KB，也可以大到 128KB。反过来，如果内核注意到预读并没有产生任何的有效命中（也就是说，应用程序在文件中四处跳转，不是顺序地读取），它也可能完全禁止预读功能。

页面缓存的存在是透明的。系统程序员通常并不能利用页面缓冲来更加优化他们的代码，other than, perhaps, not implementing such a cache in user space themselves。一般来说，高效的代码都需要最佳地利用页面缓存。另一方面，有效利用预读则是可能的。顺序文件 I/O 总是应该优先于随机访问，尽管这并不总是切实可行。

页面写回

正如前面“write()的行为”一节讨论的，内核通过缓冲区来延迟写入。当一个进程发起写入请求时，数据会被拷贝到一个缓冲区中，然后这个缓冲区被标记为 dirty，表示内存中的数据拷贝比磁盘拷贝更新，然后写入请求就简单地返回。如果文件的同一块数据发生另一个写入请求，则直接使用新的数据来更新缓冲区。同一个文件其它位置的写入请求将产生新的缓冲区。

最后 dirty 缓冲区需要提交到磁盘，使用内存中的数据来同步磁盘文件。这就是写回，它发生在以下两种情况下：

- 当可用内存低于系统配置的极限值时，dirty 缓冲区将被写回到磁盘中，这些清空的缓冲区可以被移除，释放内存空间。
- 当一个 dirty 缓冲区的时间大于系统中配置的极限值时，缓冲区会被写回到磁盘。这防止了数据保持 dirty 的不确定性。

写回操作由内核的一组被称为 pdflush 的线程来执行（名字推测起来可能是 page dirty flush，可谁知道呢）。当前面两种条件之一吻合时，pdflush 线程被唤醒，并开始提交 dirty 缓冲区到磁盘中，直到两个条件都不满足为止。

同一时间可能会有多个 pdflush 线程实例同时执行写回。这样做是为了利用并行的好处和实现拥塞避免。*拥塞避免*试图在等待向任何块设备写入时保持其它设备的写入。如果存在多个块设备的 dirty 缓冲区，这些 pdflush 线程就可以完全利用每个块设备。这解决了早期内核的一个缺陷：pdflush 线程的前任（bdflush，一个单独的线程）会花费所有的时间来等待一个单独的块设备，而使其它设备空闲。在现代机器上，Linux 内核现在可以同时保持大量的磁盘工作饱和。

内核中使用数据结构 buffer_head 来描述缓冲区。buffer_head 保持了缓冲区相关的各种元数据，例如缓冲区是 clean 还是 dirty。它同时还包含一个指向实际数据的指针。实际的数据保存在页面缓存中。按这种方式，缓冲子系统和页面缓存是统一的。

在早期的 Linux 内核版本中（2.4 以前），缓冲子系统和页面缓存是分离的，因此同时有页面缓存和缓冲区缓存。这意味着数据可能同时存在于缓冲区缓存（dirty 缓冲区）和页面缓存中（缓存的数据）。很自然地，同步这两个独立的缓存需要费一些力气。Linux 内核 2.4 引入的统一页面缓存是一个很受欢迎的改进。

Linux 的延迟写和缓冲子系统使得写操作非常快速，代价则是在电源故障时可能会有数据丢失的风险。要避免这个风险，应用程序可以使用同步 I/O（本章前面讨论过）。

总结

本章讨论了 Linux 系统编程的基础：文件 I/O。在类似 Linux 这样的系统中，一般都努力把尽可能多的东西描述为文件，于是知道如何打开、读取、写入和关闭文件是非常重要的。所有这些操作都是经典的 Unix 方式，已经被许多标准描述。

下一章讲解缓冲 I/O，和标准 C 库的标准 I/O 接口。标准 C 库不仅仅更加方便，用户空间的缓冲 I/O 还提供至关紧要的性能改进。

第三章 缓冲 I/O

回想一下第一章讲的“块”，这是文件系统的一个抽象概念，是 I/O 的 *lingua franca*——所有磁盘操作都以块的形式发生。因此，当 I/O 请求按照块对齐，也就是块大小的整数倍发起时，I/O 性能是最佳的。

随着读取操作所需要的系统调用数目的增加，I/O 性能退化将会加剧。比如，1024 次单字节的读取和一次读取 1024 字节。如果读取的大小不是块大小的整数倍，即使以大于块的大小来执行一系列读取操作，性能也可能达不到理想状态。例如，如果块大小是 1K，则读取 1130 字节的操作会比读取 1024 字节更慢。

用户缓冲 I/O

经常需要向普通文件发起许多小 I/O 请求的程序，一般都会执行用户缓冲 I/O。用户缓冲 I/O 指的是在用户空间完成缓冲而不是由内核来实现，通常要么由应用程序手动执行，要么由库透明地实现。正如第二章讨论过的，由于性能的原因，内核内部也有数据缓冲，通过延迟写、接合邻近的 I/O 请求、和预读来实现。尽管方式不一样，用户缓冲 I/O 的目标同样也是为了提高性能。

考虑下面用户空间程序 dd 的例子：

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

因为参数 bs=1，这个命令将会从设备/dev/zero（一个虚拟设备，提供无穷的 0 流）复制 2 兆字节到文件 pirate，每次一个字节，总共 2,097,152。换句话说，它将会通过大约两百万次读取和写入操作来复制数据（每次一个字节）。

现在考虑同样的 2 兆字节，但使用 1024 字节块：

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

这将复制相同的两兆字节到相同的文件中，但只发起 1024 次读取和写入操作。性能的提升非常巨大，你可以从表 3-1 看到这一点。我记录了四个 dd 命令所花费的时间（使用三种不同的度量），每个命令只有块大小不一样。实际时间是总共花费的实际时钟；用户时间是在用户空间执行程序代码花费的时间；系统时间则是进程在内核空间中执行系统调用花费的时间。

表 3-1. 块大小对性能的影响

块大小(byte)	实际时间(s)	用户时间(s)	系统时间(s)
1	18.707	1.118	17.549
1024	0.025	0.002	0.023
1130	0.035	0.002	0.027

相比单字节块，使用 1024 字节块导致巨大的性能提升。然而，上表同时也证明了使用一个大于块大小的值（意味着更少的系统调用），如果操作不是以磁盘块大小的整数倍进行，也会导致一定的性能损失。尽管需要更少的系统调用，1130 字节块会产生非对齐的 I/O 请求，因此效率上比 1024 字节的请求稍微差一些。

要利用这个性能提升，需要提前知道物理块大小。表 3-1 的结果显示块大小最有可能是

1,024、1,024 的整数倍、或者 1,024 的除数。对于/dev/zero，块大小实际上是 4,096 字节。

块大小

在实践中，块大小通常是 512、1024、2048 或者 4096 字节。

正如表 3-1 证明的，仅仅按块大小的整数倍或者除数来执行 I/O 操作，就能够获得很大的性能提升。因为内核和硬件是通过块来进行对话的。所以，使用块大小或者某个整数倍的值，可以保证块对齐的 I/O 请求，防止内核执行无关的操作。

获得某个设备的块大小很容易，可以使用 stat()系统调用（第七章讲解）或者 stat 命令。但实际情况下，你通常并不需要知道块的确切大小。

对 I/O 操作选择一个合适的块大小，首先是不能选择像 1130 这样奇怪的值。Unix 历史上从来没有使用过 1130 字节作为块的大小，选择这样的值会导致第一个 I/O 请求之后将产生未对齐的 I/O 请求。使用块大小的倍数或者约数，能够阻止未对齐请求。所以只要你选择的值能够使所有 I/O 操作都块对齐，性能就是好的。越大的倍数则需要越少的系统调用。

因此，执行 I/O 操作最简单的选择是使用一个大的缓冲区，它的大小是典型块大小的整数倍。4096 和 8192 字节都工作得很好。

问题是我们的程序很少直接处理块数据。程序一般以域、行、或单字符工作，而不是块这样的抽象。正如前面说明过的，为了弥补这种情况，程序通常采用用户缓冲 I/O。数据被写入时，它首先被存储在程序地址空间的一个缓冲区中。当缓冲区达到特定的大小时，整个缓冲区将被一次单独的写操作写入；同样，读取操作会一次读取缓冲区大小、块对齐的数据块。应用程序发起零散大小的读取请求时，将从缓冲区中一小块一小块数据取出来。最终当缓冲区为空时，另一个块对齐的大数据块将被读进缓冲区。如果缓冲区大小合适，性能会有很大提升。

在自己的程序中手动实现用户缓冲是可能的，而且实际上，许多要求严格的应用程序就是这样做。但是大多数程序都直接使用流行的标准 I/O 库（标准 C 库的一部分），它提供了一个健壮而强大的用户缓冲解决方案。

标准 I/O

标准 C 库提供了标准 I/O 库（通常简称 stdio），标准 I/O 库提供平台无关的用户缓冲解决方案。标准 I/O 库使用简单，但功能强大。

和 FORTRAN 这样的编程语言不一样，除了流程控制、算术运算等基本功能外，C 语言并没有包含其它内建的支持或者关键字，来提供更高级的功能，因此 C 语言没有内建的 I/O 支持。随着 C 编程语言的发展，用户开发了一些标准函数来提供核心功能，如字符串操作、数学函数、时间和日期、和 I/O 等。这些函数随着时间越来越成熟，最终在 1989 年由 ANSI C 标准化(C89)，这些函数也正式成为标准 C 库。尽管 C95 和 C99 增加了一些新接口，标准 I/O 库自从 1989 年创建以来，却基本没有什么变化。

本章讨论用户缓冲 I/O，它属于文件 I/O 范畴，并且由标准 C 库实现。主要讲解如何通过标准 C 库来打开、关闭、读取和写入文件。不管应用程序是否使用标准 I/O，采用用户缓冲还是直接使用系统调用，都是开发人员在权衡应用的需求和行为之后，必须仔细做的一个决定。

标准 C 总是把一些细节留给具体的实现，而且通常的实现都会添加一些额外的特性。本章和本书其它部分一样，所有列出的接口和行为，都以现代 Linux 系统的 glibc 实现为基础。

当 Linux 与标准背离时，会特别地注明。

文件指针

标准 I/O 函数并不直接操作文件描述符，而是使用自己定义的唯一标识，就是文件指针。在 C 库里，文件指针映射至一个文件描述符。文件指针是指向 FILE 类型的指针，在<stdio.h>中定义。

按照标准 I/O 的说法，一个打开的文件被称为流。流可以打开用来读取(输入流)、写入(输出流)、或者二者一起(输入/输出流)。

打开文件

使用 fopen()打开文件来读取或者写入：

```
#include <stdio.h>
FILE * fopen (const char *path, const char *mode);
```

这个函数根据指定的模式打开路径 path 指向的文件，并把它关联到一个新的流上。

模式

mode 参数描述怎样打开给定的文件。它可以是下面值之一：

r

打开文件来读取。流被定位到文件的开始位置。

r+

打开文件来读取和写入。流被定位到文件的开始位置。

w

打开文件来写入。如果文件已存在，将被截短为 0 长度。如果文件不存在，将被创建。流被定位到文件的开始位置。

w+

打开文件来读取和写入。如果文件已存在，将被截短为 0 长度。如果文件不存在，将被创建。流被定位到文件的开始位置。

a

以附加模式打开文件来写入。如果文件不存在，则将被创建。流被定位到文件的末尾，所有写入的数据都会附加到文件的末尾。

a+

以附加模式打开文件来读取和写入。如果文件不存在，则将被创建。流被定位到文件的末尾，所有写入的数据都会附加到文件的末尾。

模式也可以包含字符”b”，尽管 Linux 总是忽略这个值。有一些操作系统区别对待文本和二进制文件，使用”b”来指示以二进制模式打开文件。Linux 和所有其它遵循 POSIX 的系统一样，不区分文本文件和二进制文件。

成功时 fopen()返回一个合法的 FILE 指针；失败时返回 NULL，并设置 errno 为相应的错误代码。

例如，下列代码打开/etc/manifest 文件来读取，并把它关联到 stream 上：

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

通过文件描述符打开流

函数 fdopen()把一个已经打开的文件描述符(fd)转换为一个流：

```
#include <stdio.h>
FILE * fdopen (int fd, const char *mode);
```

可以使用的 mode 和 fopen()函数一样，但是必须与最初打开文件描述符时使用的模式兼容。fdopen 可以指定 w 和 w+模式，但文件不会被截短。流总是被定位到文件描述符关联的文件末尾。

一旦文件描述符被转换为流，就不应该再对文件描述符直接执行 I/O 操作。不过这样操作确实是合法的。注意文件描述符没有被复制，仅仅只是关联到一个新的流而已。关闭流同时也会关闭文件描述符。

成功时，fdopen()返回一个合法的文件指针；失败时返回 NULL。

例如，下列代码通过 open()系统调用打开/home/kidd/map.txt，然后使用文件描述符来创建一个关联流：

```
FILE *stream;
int fd;
fd = open ("/home/kidd/map.txt", O_RDONLY);
if (fd == &#8722;1)
    /* error */
stream = fdopen (fd, "r");
if (!stream)
    /* error */
```

关闭流

使用 fclose()函数关闭指定的流：

```
#include <stdio.h>
int fclose (FILE *stream);
```

所有已缓冲但尚未写入的数据会首先被 flush。成功时 fclose()返回 0；失败时返回 EOF 并设置 errno 为相应的错误代码。

关闭所有流

fcloseall()函数关闭当前进程关联的所有流，包括标准输入、标准输出和标准错误流：

```
#define _GNU_SOURCE
#include <stdio.h>
int fcloseall (void);
```

在关闭之前，所有流都将被 flush。这个函数总是返回 0；它是 Linux 特有的。

从流中读取

标准 C 库实现了多个函数从打开的流中读取数据，从最普通到最复杂的函数都有。本节会讲解其中最流行的三个读取方法：一次读取一个字符、一次读取一整行、和读取二进制数据。要从流中读取数据，必须指定合适的 mode 以输入流的方式打开。也就是除 w 或者 a 以外的其它模式。

一次读取一个字符

通常理想的 I/O 方式就是简单地一次读取一个字符。fgetc()函数用来从流中读取一个单独的字符：

```
#include <stdio.h>
int fgetc (FILE *stream);
```

这个函数从 stream 中读取下一个字符，然后把它从 unsigned char 转换为 int 并返回。执行转换是为了拥有足够的范围来通知到达文件末尾或者出现错误：这时候返回 EOF。fgetc()的返回值必须储存在 int 中，把它存储为 char 是一个普遍但却危险的错误。

下面的例子从 stream 中读取一个字符，检查是否出现错误，然后把结果打印出来：

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

stream 指向的流必须以读取方式打开。

回退字符

标准 I/O 提供一个函数来把一个字符回退到流中，允许你“偷看”一下流，然后如果发现并不是你需要的字符，就可以回退字符：

```
#include <stdio.h>
int ungetc (int c, FILE *stream);
```

每次调用把 c 转换为 unsigned char，然后回退到流中。成功时函数返回 c；失败时返回 EOF。接下来对流的读取操作将返回字符 c。如果多个字符被回退，之后的读取操作将以相反的顺序返回这些字符。也就是，最后一个被回退的字符将首先返回。POSIX 规定在不穿插读取请求时，只保证一个回退操作成功。因此某些实现只允许一次回退操作，Linux 允许无限数量的回退，只要可用内存足够。一次回退当然总是会成功的。

如果你在调用 ungetc()之后，但是发起读取请求之前，穿插调用了 seek 函数，将导致所有回退的字符被丢弃。对于一个进程的多个线程来说，这也是正确的，因为线程共享缓冲区。

一次读取一整行

fgets()函数从指定的流中读取一个字符串：

```
#include <stdio.h>
char * fgets (char *str, int size, FILE *stream);
```

这个函数最多从 stream 中读取 n 减 1 个字节，并把结果存储在 str 中。空字符(\0)会存储在读取的字节后面。遇到 EOF 或者换行字符时，读取操作停止。如果读取到了换行字符，则\n 也会存储在 str 中。

成功时返回 str；失败时返回 NULL。

例如：

```
char buf[LINE_MAX];
if (!fgets (buf, LINE_MAX, stream))
    /* error */
```

POSIX 在<limits.h>中定义了 LINE_MAX：它是 POSIX 行操作接口能处理的最大输入行大小。Linux 的 C 库没有这个限制——行可以是任意大小——但有了 LINE_MAX 定义，就没有办法与超出的部分交互。可移植的程序可以使用 LINE_MAX 来保证安全；在 Linux 中 LINE_MAX 被设置得相当大。Linux 特定的程序不需要担心行大小的限制。

读取任意字符串

通常，基于行的 fgets()读取很有用，不过它也很烦人。有时候，开发者希望使用一个分隔符而不是换行符。其它时候，开发者可能完全不需要分隔符——开发者很少希望分隔符也被存储在缓冲区中。现在回顾起来，把换行符存储在返回缓冲区的决定很少看上去是正确的。

使用 fgetc()编写一个 fgets()的替代品并不困难。例如，下面代码片断从 stream 读取 n-1 字节到 str 中，然后添加一个\0 字符：

```
char *s;
int c;
s = str;
while (--n > 0 && (c = fgetc (stream)) != EOF)
    *s++ = c;
*s = '\0';
```

这段代码也可以扩展为读取到分隔符时停止，指定分隔符 d（在这个例子中不允许为空

字符）：

```
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';
```

设置 d 为\n 可以提供类似于 fgets()的行为，除了后者把\n 存储在缓冲区中以外。

依赖于 fgets()的具体实现，我们的代码可能会慢一些，因为它发起重复的 fgetc()函数调用。不过这与前面列举的 dd 例子并不一样！尽管这段代码会导致额外的函数调用开销，它却并不会像 dd 使用 bs=1 时那样导致系统调用和未对齐 I/O 的开销。后者是更大的问题。

读取二进制数据

对于某些应用来说，读取单个字符或者整行是不够的。有时候，开发者希望读取或者写入复杂的二进制数据，例如 C 结构体。标准 I/O 库提供 fread()实现这个功能：

```
#include <stdio.h>
size_t fread (void *buf, size_t size, size_t nr, FILE *stream);
```

调用 fread()最多将从 stream 读取 nr 个元素，每个 size 字节，到 buf 指向的缓冲区中。文件指针将更新所读取的字节数。

函数返回读取的元素个数（而不是读取的字节数）。fread()返回小于 nr 的值时表示发生错误或者到达 EOF。不幸的是，除非调用 ferror()和 feof()，我们无法判断到底是哪一种情况发生（请参看后面“错误和 EOF”一节）。

由于变量大小、对齐方式、填充方式、字节顺序等方面的区别，一个应用写入的二进制数据，有可能不能被另一个应用读取，甚至是不同机器上的同一个应用程序。

fread()最简单的例子是从指定的流中读取一个线性字节的单个元素：

```
char buf[64];
size_t nr;
nr = fread (buf, sizeof(buf), 1, stream);
if (nr == 0)
    /* error */
```

当我们学习 fread()的配对函数 fwrite()时，会举更复杂的例子。

对齐相关的问题

所有机器架构都有数据对齐的需求。程序员通常简单地把内存看成是一个大的字节数组。但处理器并不是按字节大小来从内存中读取和写入数据。相反，处理器通过一个特定的粒度来访问内存，例如 2、4、8、或者 16 字节。因为每个进程的地址空间都从地址 0 开始，处理器访问的内存地址必须是粒度值的整数倍。

因此，C 变量必须存储和访问对齐的地址。通常，变量是自然对齐的，也就是说对齐是相应于 C 数据类型的大小的。例如，一个 32 位整数会对齐在 4 字节的边界上。换句话说，int 类型会被存储在一个 4 的倍数的内存地址上。

访问未对齐数据有各种处罚，这和具体的机器架构有关。有一些处理器可以访问未对齐数据，但会有巨大的性能降低。其它一些处理器则完全不能访问未对齐数据，试图访问这样的数据将引起一个硬件异常。更严重的是，有一些处理器为了强制地址对齐，会悄悄地丢弃低位数据，这总是会导致未意料的行为。

通常，编译器自然地对齐所有数据，因此对于程序员来说对齐是不可见的。处理结构体，手工执行内存管理、保存二进制数据到磁盘中、网络通讯都可能带来对齐的问题。因此系统程序员应该精通这些。

第八章将更深入地讲解对齐。

向流中写入

和读取一样，标准 C 库定义了许多函数来向打开的流中写入数据。本节将查看最流行的三种写入方法：写入一个单独的字节、写入一个字符串、和写入二进制数据。这种写入方式的多样性，对于缓冲 I/O 来说是非常合适的。要写入到一个流中，必须以指定的模式打开输出流。也就是说，除了 r 以外的任何合法模式。

写一个单独的字符

Fgetc()的配对函数是 fputc():

```
#include <stdio.h>
int fputc (int c, FILE *stream);
```

fputc()函数写入 c 指定的字节（转换为 unsigned char）到 stream 流。成功结束时，函数返回 c；否则返回 EOF，并且设置 errno 为适当的值。

使用非常简单：

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

这个例子写入字符 p 到 stream，它必须以写入模式打开。

写入一个字符串

函数 fputs()用来写入一个完整字符串到指定的流：

```
#include <stdio.h>
int fputs (const char *str, FILE *stream);
```

调用 fputs()写入 str 指向的 null 结尾的字符串到 stream 流中。成功时，fputs()返回一个非负整数；失败时返回 EOF。

下面例子以附加写入模式打开文件，然后写入给定的字符串到相关联的流，最后关闭流：

```
FILE *stream;
stream = fopen ("journal.txt", "a");

if (!stream)
        /* error */

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

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

写入二进制数据

当程序需要写入复杂数据时，写单个字符和行字符串就不够了。为了直接存储二进制数据（如 C 变量），标准 I/O 提供了 fwrite()：

```
#include <stdio.h>
size_t fwrite (void *buf,
               size_t size,
               size_t nr,
               FILE *stream);
```

调用 fwrite()将从 buf 指向的数据中，最多写入 nr 个元素到 stream，每个大小 size 字节。文件指针会增加写入的总字节数。

函数成功时返回成功写入的元素个数（而不是字节数），小于 nr 的返回值表示发生错误。

使用缓冲 I/O 的示例程序

现在让我们来看一个例子（实际上是一个完整的程序），整合了本章我们已经介绍过的许多接口。这个程序首先定义了结构体 pirate，然后声明了两个 pirate 类型的变量。程序初始化其中一个变量，然后通过输出流把它写入到磁盘文件 data。通过另一个输入流，程序把这块数据重新从文件 data 中直接读取到另一个 struct pirate 变量中。最后，程序把结构体的内容写入到标准输出流：

```
#include <stdio.h>
int main (void)
{
        FILE *in, *out;
        struct pirate {
                char name[100]; /* real name */
                unsigned long booty; /* in pounds sterling */
                unsigned int beard_len; /* in inches */
        } 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;
}
```

输出当然就是原来的值：

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

再一次说明，由于变量大小、对齐等许多原因，一个程序写入的二进制数据，可能无法被另一个程序读取，记住这一点非常重要。也就是说，另一个不同的程序（甚至是运行于不同机器上的同一个程序）可能无法正确地读取 fwrite()写入的数据。在我们的例子中，需要考虑 unsigned long 类型的大小改变，或者填充的数量改变。这些东西只有在特定的机器类型和特定的 ABI 下才能确保不变。

Seeking 流

通常操作流的当前位置是很有用的。不过应用读取复杂的基于记录的文件时，可能需要特定的跳跃。另外，有时候也需要把流重置到文件位置 0。无论哪种情况，标准 I/O 提供了一组接口，在功能上基本等同于系统调用 lseek()。fseek()函数是标准 I/O 最常用的 seek 接口，根据 offset 和 whence 来操作 stream 流的文件位置：

```
#include <stdio.h>
int fseek (FILE *stream, long offset, int whence);
```

如果 whence 设置为 SEEK_SET，文件位置将被设置为 offset；如果 whence 被设置为 SEEK_CUR，文件位置将被设置为当前位置加上 offset；如果 whence 设置为 SEEK_END，文件位置将被设置为文件末尾加上 offset。

成功结束时，fseek()返回 0，清除 EOF 指示器，并且撤消 ungetc()的效果（如果有）；错误时返回-1，并设置 errno 为适当的值。最常见的错误是无效流(EBADF)和无效 whence 参数(EINVAL)。

作为一种选择，标准 I/O 还提供了 fsetpos()：

```
#include <stdio.h>
int fsetpos (FILE *stream, fpos_t *pos);
```

这个函数设置流 stream 的位置为 pos。它的工作与 fseek()参数 whence 为 SEEK_SET 时一样。成功时返回 0；否则返回-1，并设置 errno。fsetpos 函数（以及它的配对函数 fgetpos）完全是给其它非 Unix 平台提供的，它们使用复杂的类型来表示流位置。在这些平台上，fsetpos 函数是设置流位置到任意位置的唯一方法，因为 C 语言 long 类型可能并不足够。Linux 特定的应用不需要使用这个接口，除非要对所有可能的平台可移植。

标准 I/O 同样也提供了 rewind()函数，作为一个捷径：

```
#include <stdio.h>
void rewind (FILE *stream);
```

调用 rewind(stream)重置位置到流的开始位置。它等同于：

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

除了后者同时会清除错误指示器之外。

注意 rewind()没有返回值，因此不能直接与错误情况交互。调用者如果希望确认错误是否存在，应该在调用函数之前先清除 errno，然后在函数结束之后检查 errno 变量的值是否非 0。例如：

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

获得流的当前位置

和 lseek()不一样，fseek()并不返回更新后的位置，标准 I/O 提供一个单独的接口来完成

这个任务。ftell()函数返回流 stream 的当前位置：

```
#include <stdio.h>
long ftell (FILE *stream);
```

出错时返回-1，并且设置 errno。

作为一个选择，标准 I/O 还提供了 fgetpos()：

```
#include <stdioh.h>
int fgetpos (FILE *stream, fpos_t *pos);
```

成功时 fgetpos()返回 0，并且把流 stream 的当前位置值存入 pos。失败时返回-1，并设置 errno。和 fsetpos()一样，fgetpos()也完全是提供给文件位置类型很复杂的非 Linux 平台。

Flushing 流

标准 I/O 库提供一个接口来把用户缓冲写出到内核，确保通过 write()写出的所有数据被刷新到流中。fflush()函数提供这个功能：

```
#include <stdio.h>
int fflush (FILE *stream);
```

调用 fflush()后，流 stream 中的任何未写入数据都被 flushed 到内核。如果 stream 为 NULL，则当前进程所有已打开的输入流都将被 flushed。成功时 fflush()返回 0；失败时返回 EOF，并设置 errno 为适当的值。

要理解 fflush()的作用，你必须理解 C 库管理的缓冲和内核自己的缓冲之间的区别。本章描述的所有调用都工作在 C 库管理的缓冲中，是处于用户空间而不是内核空间的。这也是性能获得提升的地方，你一直在用户空间执行代码，而不发起系统调用。只有需要访问磁盘或者其它媒介时，才会发起系统调用。

fflush()仅仅把用户缓冲数据写到内核缓冲中，效果和不采用用户缓冲而直接使用 write()一样。它不能确保数据被物理提交到任何媒介（如果需要这样，使用 fsync）。你很可能会先调用 fflush()，然后马上跟着调用 fsync()：也就是，首先确保用户缓冲被写到内核，然后确保内核缓冲被写出到磁盘中。

错误和 End-of-File

标准 I/O 的一些接口——例如 fread()，出现错误时与调用方的交互非常差，因为它们没有提供区分错误和 EOF 的机制。使用这些接口时，以及另外一些场合下，检查某个给定流的状态并确定流是否遇到错误、或者到达文件末尾，这非常有用。标准 I/O 提供了两个接口。

函数 ferror()检测 stream 的错误指示器是否被设置：

```
include <stdio.h>
int ferror (FILE *stream);
```

错误指示器被其它标准 I/O 接口在发生错误时设置。如果错误指示器被设置，函数返回一个非 0 值，否则返回 0。

函数 feof()检测 stream 的 EOF 指示器是否被设置：

```
include <stdio.h>
int feof (FILE *stream);
```

EOF 指示器被其它标准 I/O 接口在到达文件末尾时设置。如果 EOF 指示器被设置，函数返回一个非 0 值，否则返回 0。

clearerr()函数清除 stream 流的错误和 EOF 指示器：

```
#include <stdio.h>
void clearerr (FILE *stream);
```

它没有返回值，并且不会失败（没有办法知道是否提供无效的流）。你只应该在检测错误和 EOF 指示器之后调用 clearerr()，因为调用 clearerr()之后指示器将不可恢复。例如：

```
/* 'f' is a valid stream */
if (ferror (f))
    printf ("Error on f!\n");
if (feof (f))
    printf ("EOF on f!\n");
clearerr (f);
```

获取关联的文件描述符

有时候获取给定流背后的文件描述符是很有用的。例如，当相应的标准 I/O 函数不存在时，需要通过文件描述符对流执行系统调用。要获取流背后的文件描述符，使用 fileno()：

```
#include <stdio.h>
int fileno (FILE *stream);
```

成功时 fileno()返回 stream 相关联的文件描述符；失败时返回-1。只有在给定的流无效时才可能发生失败，这时候函数将设置 errno 为 EBADF。

混合标准 I/O 和系统调用通常并不被推荐。使用 fileno()时程序员必须小心确保正确的行为。特别地，在操作文件描述符之前先 flush 流可能是明智的。你永远不应该混合执行实际的 I/O 操作。

控制缓冲

标准 I/O 实现了三种类型的用户缓冲，并且提供开发者控制缓冲类型和大小的接口。不同类型的用户缓冲实现不同的目的，并且适用于不同的情形。下面是几个选项：

无缓冲

不执行用户缓冲，数据直接提交给内核。由于这是用户缓冲的对立，这个选项并不常用。标准错误默认是无缓冲的。

行缓冲

对每一行数据执行缓冲。遇到每个换行字符时，提交缓冲到内核中。行缓冲对于输出到

屏幕的流来说非常有意义。因此，它是终端默认使用的缓冲（标准输出默认也是行缓冲）。

块缓冲

对每一个块数据执行缓冲。这就是本章开头讨论过的缓冲类型，对于文件来说是最理想的。所有与文件相关的流默认都是块缓冲的。标准 I/O 使用术语完全缓冲表示块缓冲。

大部分时候，默认缓冲类型是正确而且最佳的。标准 I/O 仍然提供了一个接口来控制所采用的缓冲类型：

```
#include <stdio.h>
int setvbuf (FILE *stream, char *buf, int mode, size_t size);
```

setvbuf()函数设置 stream 的缓冲类型为 mode，它必须是以下之一：

_IONBF　无缓冲

_IOLBF　行缓冲

_IOFBF　块缓冲

当使用_IONBF 时，buf 和 size 参数将被忽略。buf 指向一个 size 字节的缓冲区，标准 I/O 将使用它作为流的缓冲区。如果 buf 是 NULL，glibc 将自动分配一个缓冲区。

setvbuf()函数必须在打开流之后，但是在任何其它操作被执行之前调用。成功时它返回 0；否则返回一个非 0 值。

如果提供了缓冲区，则在流关闭时它必须存在。一个常见的错误是在流被关闭之前，在局部范围中声明缓冲区为一个自动变量。特别需要小心的是，不要在 main()中提供一个局部的缓冲区，然后又忘记显式地关闭流。例如，下面代码就存在 Bug：

```
#include <stdio.h>
int main (void)
{
    char buf[BUFSIZ];
    /* set stdin to block-buffered with a BUFSIZ buffer */
    setvbuf (stdout, buf, _IOFBF, BUFSIZ);
    printf ("Arrr!\n");
    return 0;
}
```

这个 bug 可以通过显式关闭流来修正，或者使用全局的 buf 变量。

通常，开发者不需要关心流缓冲。标准错误、终端是行缓冲；文件是块缓冲。块缓冲的默认缓冲大小是 BUFSIZ，在<stdio.h>中定义，通常它都是最佳选择（典型块大小的大整数倍）。

线程安全

线程是单个进程内的多个执行单元，概念化线程的一种方法是把它看成是共享地址空间的多个进程。除非同步线程对数据的访问或者使用 thread-local 数据，线程可以在任何时候执行，也可能覆盖共享数据。支持线程的操作系统提供了锁机制（确保互斥的编程构造）来确保线程不会互相干扰。标准 I/O 也使用这些机制，不过这还不够。例如，有时候你可能需要锁定一组调用，从一个 I/O 操作向多个 I/O 操作扩大临界区（不被其它线程影响的代码片断）。另外一些情况下，你可能又需要完全消除锁来提高性能。在这一节，我们讨论如何实现这两个目的。

标准 I/O 函数天生就是线程安全的。在内部，它们关联了一个锁、锁计数、和每个已打开流的拥有线程。任何线程都必须获得锁才能成为流的拥有线程，然后才能发起 I/O 请求。两个或者更多线程操作同一个流时不能穿插标准 I/O 操作。因此，在单个函数调用的情况下，标准 I/O 操作是原子的。

当然在实践中，许多应用需要比单个函数调用级别更高的原子性。例如，如果多个线程发起写请求，尽管单个的写入不可能被穿插而导致混淆输出。应用程序可能希望所有写入请求全部完成而不被中断。为了实现这些，标准 I/O 提供了一组函数来单独操作流关联的锁。

手动文件锁

flockfile()函数等待流 stream 直到它不再被锁定，然后请求锁、改变锁计数，把当前线程变成流的拥有线程，最后返回：

```
#include <stdio.h>
void flockfile (FILE *stream);
```

funlockfile()函数则减少流 stream 关联的锁计数：

```
#include <stdio.h>
void funlockfile (FILE *stream);
```

如果锁计数变为 0，当前线程将放弃流的拥有权。这时候另一个线程才可以请求锁。

这些调用可以嵌套。也就是说，一个线程可以发起多个 flockfile()调用，这样除非进程发起相应数量的 funlockfile()调用，流就一直被锁定而不释放。

ftrylockfile()函数是 flockfile()的非阻塞版本：

```
#include <stdio.h>
int ftrylockfile (FILE *stream);
```

如果流 stream 当前已被锁定，则 ftrylockfile()不做任何事情，立即返回一个非 0 值。如果流 stream 当前没有被锁定，它就请求锁、增加锁计数，然后成为 stream 的拥有线程，并返回 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);
```

尽管单个 fputs()操作永远不会引起竞争条件——例如，我们在写入”List of treasure”的时候不可能会穿插其它任何东西——但另一个线程对同一个流的其它标准 I/O 操作，则可能会穿插在当前线程的两个 fputs()调用之间。理想情况下，应用程序应该设计成多个线程不向同一个流提交 I/O 操作。如果你的应用确实需要这样做，则你需要比单个函数更高的原子区域，flockfile()以及其它几个函数可以帮助你实现。

无锁流操作

这里有第二个理由执行手动流锁定。由应用程序员提供锁，可以获得更加细粒度和更精确的锁控制，它也可以最小化锁的开销并提高性能。Linux 提供了一组与通用标准 I/O 函数相近的堂兄弟，它们完全不执行任何锁控制。从效果上看，就是标准 I/O 的无锁版本：

```
#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);
```

这些函数的行为和他们带锁的版本完全一样，除了它们不检查或者请求给定流 stream 关联的锁之外。如果确实需要锁，由程序员负责确保手动请求并释放锁。

尽管 POSIX 确实定义了标准 I/O 函数的一些无锁定变体，上面列出的函数却都没有被 POSIX 定义。它们是 Linux 特定的函数，许多其它 Unix 系统也支持部分子集。

标准 I/O 批评

标准 I/O 使用非常广泛，有一些专家已经指出它的一些缺点。某些函数——例如 fgets()——有时候可能并不适用。其它一些函数——例如 gets()——是如此可怕以致已经完全从标准中去除。

对标准 I/O 最大的抱怨是**双重复制**的性能影响。当读取数据时，标准 I/O 向内核发起 read()系统调用，把内核中的数据复制到标准 I/O 缓冲区。当应用随后通过标准 I/O 发起读取请求时——例如使用 fgetc()——数据将再次被复制，这一次则是从标准 I/O 缓冲区复制到应用提供的缓冲区。写入请求则以相反的方式进行：数据先从应用提供的缓冲区中复制到标准 I/O 缓冲区，然后再通过 write()从标准 I/O 缓冲区复制到内核中。

避免双重复制的一种选择是在每次读取请求时返回一个指向标准 I/O 缓冲区的指针，然后数据就可以直接从标准 I/O 缓冲区中读取，而不需要额外的复制。在应用程序确实需要数据存储在本地缓冲区的情况下（可能需要写入），它也总是可以执行手动复制。这种实现能够提供一个“自由”的接口，允许应用使用完读取缓冲后发出信号。

写入可能会更加复杂些，但双重复制仍然可以避免。当发起一个写入请求时，I/O 实现可以记录下指针。最终当准备好向内核 flush 数据时，I/O 实现可以遍历已存储的指针链表，

并写出数据。这可以使用分散－集中(scatter-gather)I/O 来完成，只需要一个系统调用：writev()（我们在下一章讨论 scatter-gather I/O）。

已经有高度优化的用户缓冲库，采用类似于我们讨论过的实现方式，解决双重复制问题。有一些开发者则选择实现自己的用户缓冲解决方案。不过不管怎样，标准 I/O 仍然非常流行。

总结

标准 I/O 是一个用户缓冲库，作为标准 C 库的一部分提供。虽然有一些缺陷，标准 I/O 库是一个强大而且非常流行的解决方案。实际上许多 C 程序员只知道标准 I/O。当然对于终端 I/O，基于行缓冲的 I/O 非常理想，标准 I/O 是唯一的选择。有谁直接使用过 write()向标准输出打印呢？

标准 I/O 以及通常意义上的用户缓冲，在以下任何条件满足时都是有意义的：

- 你需要发起许多系统调用，并且你希望通过合并许多调用来最小化开销。
- 性能至关重要，并且你希望确保所有 I/O 都以块边界对齐块大小进行。
- 你的访问方式基于字符、或者基于行，并且你希望调用接口尽量简单，而不发起无关额外的系统调用。
- 你更喜欢高级的接口而不是低级的 Linux 系统调用。

无论如何，直接使用 Linux 系统调用是最灵活的。下一章，我们将讨论 I/O 的高级组成以及相关的系统调用。

第四章 高级文件 I/O

在第二章，我们讨论了 Linux 的基本 I/O 系统调用。这些调用不仅仅构成文件 I/O 的基础，而且事实上也是 Linux 上所有通信的基础。在第三章，我们讨论了为什么在基本 I/O 系统调用之上，还需要用户空间缓冲。我们学习了一个特定的用户空间缓冲解决方案——C 标准 I/O 库。在本章，我们将讨论 Linux 提供的更加高级的 I/O 系统调用：

分散/集中(Scatter/gather) I/O

允许单个调用一次读取数据到多个缓冲区中、或者从多个缓冲区一次写入数据；这对于捆绑不同数据结构的域到一起组成一个 I/O 事务非常有用。

Epoll

第二章讨论的 poll()和 select()系统调用的改进；当一个程序需要 poll 成百上千个文件描述符时非常有用。

内存映射 I/O

把文件映射到内存中，允许通过简单的内存操作进行文件 I/O 的操作；对于某些 I/O 模式非常有用。

文件建议(File advice)

允许进程向内核提供自己对 I/O 使用场景的暗示；可以提升 I/O 性能。

异步 I/O

允许进程发起 I/O 请求而不需等待 I/O 操作结束；对于不使用线程而处理繁重 I/O 工作非常有用。

本章结尾讨论性能相关事项以及内核的 I/O 子系统。

分散/集中（Scatter/Gather）I/O

Scatter/Gather I/O 是一种输入输出方法：单个系统调用从单个数据流向一组缓冲区中写入数据；或者从单个数据流中读取数据到一组缓冲区中。这种类型的 I/O 如此命名，是因为数据被分散到缓冲区组或者从缓冲区组集中到一起，它的另一个名字是*向量 I/O*。作为对比，我们在第二章中讨论的标准读写系统调用则是*线性 I/O*。

Scatter/Gather I/O 相比线性 I/O 有以下几个优点：

更加自然的处理

如果你的数据是自然分段的——例如预定义头文件的域——向量 I/O 操作更加直观。

高效

一个向量 I/O 操作可以代替多个线性 I/O 操作。

性能

除了减少发起系统调用的数量，向量 I/O 实现可以通过内部优化提供比线性 I/O 更加改良的性能。

原子性

和多个线性 I/O 操作不同，进程可以自由执行单个向量 I/O 操作，而不会与其它进程的 I/O 操作交叉。

不使用 Scatter/gather I/O，更加自然的 I/O 方法和原子性也是可以达到的。进程可以在写入之前，把分散的向量连接到一个缓冲区中；以及在读取操作之后把返回缓冲区分解到多个向量缓冲区中。也就是说，用户空间应用可以手动执行分散和集中 I/O 操作。不过这种解决方案既不高效，实现起来也不有趣。

readv()和 writev()

POSIX 1003.1-2001 定义了一组系统调用来实现 Scatter/gather I/O，Linux 提供的实现满足前一节列出的所有目标。

readv()函数从文件描述符 fd 中读取 count 段数据到 iov 表示的缓冲区中：

```
#include <sys/uio.h>
ssize_t readv (int fd,
               const struct iovec *iov,
               int count);
```

writev()函数从 iov 表示的缓冲区中，向文件描述符 fd 中写入至多 count 段数据：

```
#include <sys/uio.h>
ssize_t writev (int fd,
               const struct iovec *iov,
               int count);
```

readv()和 writev()函数的行为分别与 read()和 write()一样，除了读取或者写入多个缓冲区。每个 iovec 结构体描述了一个独立的缓冲区，被称为段：

```
#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 iovec 段就是一个向量 ，向量中的每个段都描述了数据应该写入或者读取的缓冲区在内存中的地址和大小。readv()函数依次填充每个缓冲区 iov_len 字节。writev()函数总是写出所有 iov_len 字节之后，才继续下一个缓冲区。这两个函数都是按段顺序进行操作的，从 iov[0]开始，然后是 iov[1]，直到 iov[count - 1]。

返回值

成功时 readv()和 writev()分别返回读取或者写入的字节数，这个数值应该是所有 iov_len 的总和；错误时返回-1，并设置 errno 为合适的值。这两个系统调用可能产生 read()和 write()调用的所有错误，当发生错误时，会设置相同的 errno 代码。此外，标准还定义了另外两个错误情况。

首先，由于返回类型是 ssize_t，如果所有 count 个 iov_len 的总和大于 SSIZE_MAX，则不会发生数据转移，函数返回-1，并将 errno 设置为 EINVAL。

其次，POSIX 规定 count 必须大于 0，并且小于或等于 IOV_MAX(<limits.h>中定义)，如果 count 大于 IOV_MAX，则不会发生数据转移，函数返回-1，并将 errno 设置为 EINVAL。

优化 Count
在操作向量 I/O 时，Linux 内核必须分配内部数据结构来表示每个段。通常分配是根据 count 的大小动态进行的。然而作为一种优化手段，如果 count 足够小，Linux 内核将在堆栈中创建一个小的段数组，从而消除动态分配段的需求，因此也提供了性能上的一点小提升。当前这个极限值是 8，因此如果 count 小于或者等于 8，向量 I/O 操作将会在进程的内核堆栈中以非常高效的内存方式进行。 大部分时候，在一个特定的向量 I/O 操作中，你无法确定需要一次转移多少个段。然而如果你足够灵活，可以选择 8 或者更小的值，这也能改进性能。

writev()示例

让我们考虑一个简单的例子，写出到三个段的向量中，每个段包括一个不同大小的字符串。这个自说明的程序完全足够演示 writev()函数，而且作为一段有用的代码片断足够简单：

```
#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;
        }

        /* fill out three iovec structures */
        for (i = 0; i < 3; i++) {
                iov[i].iov_base = buf[i];
                iov[i].iov_len = strlen (buf[i]);
        }

        /* with a single call, write them all out */
        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;
}
```

运行程序将如期望产生以下结果：

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

读取文件的内容如下：

```
$ 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()示例：

现在我们再来看一个使用 readv()系统调用的例子程序：使用向量 I/O 从前面产生的文本文件中读取数据。这个自说明的示例同样简单而完整：

```
#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;
        }

        /* set up our iovec structures */
        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);

        /* read into the structures with a single call */
        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;
}
```

在运行前面一个程序之后，再运行本程序，将产生以下结果：

```
$ ./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()的一种比较天真的实现，可以在用户空间使用一个简单的循环来完成，类似以下代码：

```
#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;
}
```

幸亏这不是 Linux 的实现：Linux 实现 readv()和 writev()为系统调用，并且内部执行 scatter/gather I/O。实际上，Linux 内核中的所有 I/O 都是向量化的；read()和 write()以只有一个段的向量 I/O 实现。

事件 poll 接口

认识到 poll()和 select()的局限性之后，Linux 内核 2.6 引入了事件 poll（epoll）。虽然 epoll 比之前的两个接口更加复杂，但 epoll 解决了前面两个接口共有的根本性能问题，并且增加了几个新的特性。

poll()和 select()都需要在每次调用时查看文件描述符的完整列表。内核必须遍历被监控的文件描述符列表。当列表增长到非常大时——它可能包含几百甚至几千个文件描述符——每次调用时遍历列表成为明显的瓶颈。

epoll 通过解耦注册和实际的监控来解决这个问题。一个系统调用用来初始化 epoll 实例、另一个向 epoll 实例添加被监控文件描述符或者移除文件描述符、然后第三个调用执行实际

的事件等待。

创建新的 epoll 实例

通过 epoll_create()创建 epoll 实例：

```
#include <sys/epoll.h>
int epoll_create (int size)
```

成功调用 epoll_create()将创建一个新的 epoll 实例，并返回一个与之相关的文件描述符。这个文件描述符与实际的文件没有任何关系；它仅仅是随后的 epoll 接口使用的一个句柄。size 参数是给内核提供的需要监控的文件描述符数量的提示；它并不是文件描述符数量的最大值。传递一个良好的近似值能够获得更好的性能，但并不需要提供实际的准确数值；错误时调用返回-1，并设置 errno 为以下值之一：

EINVAL

size 参数不是正数

ENFILE

系统已经到达打开文件最大数量的限制

ENOMEM

内存不足，无法完成操作

一种典型的使用如下：

```
int epfd;
epfd = epoll_create (100); /* plan to watch ~100 fds */
if (epfd < 0)
    perror ("epoll_create");
```

epoll_create()返回的文件描述符在使用结束之后，应该调用 close()来销毁。

控制 epoll

epoll_ctl()系统调用可以用来向指定 epoll 实例中添加文件描述符、以及从 epoll 实例中移除文件描述符：

```
#include <sys/epoll.h>
int epoll_ctl (int epfd,
               int op,
               int fd,
               struct epoll_event *event);
```

头文件<sys/epoll.h>定义 epoll_event 如下：

```
struct epoll_event {
    __u32 events; /* events */
    union {
        void *ptr;
```

```
        int fd;
        __u32 u32;
        __u64 u64;
    } data;
};
```

成功调用 epoll_ctl()控制 epoll 实例与文件描述符 epfd 相关联。参数 op 指定对文件描述符 fd 进行的操作。event 参数进一步描述了所要进行操作的行为。

下面是 op 参数合法的取值：

EPOLL_CTL_ADD

对文件描述符 fd 添加一个监控到 epoll 实例 epfd 中，由 event 定义具体的事件。

EPOLL_CTL_DEL

从 epoll 实例 epfd 中移除对文件描述符 fd 相关的文件的监控。

EPOLL_CTL_MOD

使用更新后的 event 参数修改一个已存在 fd 的监控。

epoll_event 结构体中的 events 域列出了对指定文件描述符进行的所有监控事件。多个事件可以使用“按位或”操作组合到一起，下面是所有合法的取值：

EPOLLERR

文件发生错误，这个事件总是被监控，即使没有指定。

EPOLLET

允许监控文件的 Edge 触发行为（参考接下来的“Edge 对比 Level 触发事件一节”）。默认的行为是 Level 触发。

EPOLLHUP

文件发生挂起，这个事件总是被监控，即使没有指定。

EPOLLIN

文件可以读取而不会发生阻塞。

EPOLLONESHOT

在事件产生并且被读取之后，文件自动不再被监控。一个新的事件必须使用 EPOLL_CTL_MOD 来重新注册。

EPOLLOUT

文件可以被写入而不会阻塞。

EPOLLPRI

有紧迫的 out-of-band 数据能够被读取。

event_poll 结构体中的 data 域提供给用户私人使用，其中的内容在用户收到所请求的事件时返回。通常的实践是设置 event.data.fd 为 fd，这样当事件发生时查找哪个文件描述符触发事件就非常容易。

成功时，epoll_ctl()返回 0；失败时返回-1，并设置 errno 为下列值之一：

EBADF

epfd 不是一个合法的 epoll 实例，或者 fd 不是一个合法的文件描述符。

EEXIST

op 为 EPOLL_CTL_ADD，但 fd 已经存在并与 epfd 相关联。

EINVAL

epfd 不是 epoll 实例，epfd 和 fd 相同，或者 op 非法。

ENOENT

op 为 EPOLL_CTL_MOD 或者 EPOLL_CTL_DEL，但 fd 并没有与 epfd 相关联。

ENOMEM

内存不足，无法处理请求。

EPERM

fd 不支持 epoll。

作为一个例子，要给文件描述符 fd 向 epoll 实例 epfd 中添加一个新的监控，你可以如下编写代码：

```
struct epoll_event event;
int ret;

event.data.fd = fd; /* return the fd to us later */
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 关联的文件描述符 fd 已经存在的事件，可以如下编写代码：

```
struct epoll_event event;
int ret;
event.data.fd = fd; /* return the fd to us later */
event.events = EPOLLIN;

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

反过来，要从 epoll 实例 epfd 中移除文件描述符 fd 的事件，可以如下编写代码：

```
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 参数可以为 NULL，因为这时候不需要提供事件。但是 2.6.9 之前的内核，错误地检查 op 参数不可为 NULL。为了对这些老版本内核可移植，你应该传递一个合法非 NULL 的指针，它不会被函数修改。内核 2.6.9 修正了这个 bug。

等待 epoll 事件

系统调用 epoll_wait()等待指定 epoll 实例相关联的文件描述符的事件：

```
#include <sys/epoll.h>
int epoll_wait (int epfd,
                struct epoll_event *events,
                int maxevents,
                int timeout);
```

调用 epoll_wait()最多等待 epoll 实例 epfd 相关联的文件描述符的事件 timeout 毫秒。成功时 events 指向一个描述每个事件的 epoll_event 结构体，最多可以有 maxevents 个事件。返回值是事件个数，或者错误时返回-1，在发生错误时函数设置 errno 为以下值之一：

EBADF

epfd 不是合法的文件描述符。

EFAULT

进程没有对 events 指向内存的写权限。

EINTR

系统调用完成之前被信号中断。

EINVAL

epfd 不是合法的 epoll 实例，或者 maxevents 等于小于 0。

如果 timeout 为 0，调用将立即返回，即使没有事件产生，这时候函数也将返回 0。如果 timeout 为-1，调用将一直阻塞不返回，除非有事件发生。

当调用返回时，epoll_event 结构体的 events 域描述了发生的事件。data 域包含用户在调用 epoll_ctl()之前设置的值。

完整的 epoll_wait()示例如下：

```
#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);

        /*
        * We now can, per events[i].events, operate on
        * events[i].data.fd without blocking.
        */
    }
    free (events);
```

我们将在第八章讲解 malloc()和 free()函数。

Edge-触发对比 Level-触发事件

如果传递给 epoll_ctl()函数的参数 events 的域有设置 EPOLLET 值，则对 fd 的监控就是 Edge-触发的，与默认的 Level-触发相反。

考虑以下通过 Unix 管道通信的生产者－消费者之间的事件：

1. 生产者写入 1KB 数据到管道中。
2. 消费者对管道执行 epoll_wait()，等待管道包含数据变为可读。

使用 Level-触发监控时，步骤 2 的 epoll_wait()调用会立即返回，表示管道已经准备好被读取。当使用 Edge-触发监控时，这个调用则不会返回，直到步骤 1 发生。也就是说，即使管道在调用 epoll_wait()时有数据可读取，调用也不会返回，直到有数据被再次写入到管道。

Level-触发是默认的行为，也是 poll()和 select()的行为，是大部分开发者期望执行的操作。Edge-触发的行为需要不同的编程方法，通常在非阻塞 I/O 中使用，并且小心的检查 EAGAIN。

映射文件到内存

作为标准文件 I/O 的另一种选择，内核提供了一个接口允许应用映射一个文件到内存中，意味着在内存地址与文件内容之间有一对一的对应关系。然后程序员可以直接通过内存来访问文件，和访问其它内存数据块完全一样——甚至可以直接写入到内存区域，同时透明地映射到磁盘文件中。

POSIX.1 标准化了（Linux 实现了）mmap()系统调用来映射对象到内存中。本节将讨论使用 mmap()映射文件到内存中来执行 I/O；在第八章，我们将讨论 mmap()的其它应用。

mmap()

调用 mmap()请求内核映射文件描述符 fd 表示对象的 len 字节到内存中，从 offset 字节处开始。如果提供了 addr 参数，表示优先使用它作为内存中的开始地址。访问权限由 prot 指定，额外的行为则由 flags 指定：

```
#include <sys/mman.h>
void * mmap (void *addr,
             size_t len,
             int prot,
```

int flags,
int fd,
off_t offset);

addr 参数只是向内核建议映射文件到内存的地址，仅仅是一个提示而已，大多数用户都直接传递 0。mmap()调用返回内存中实际映射的开始地址。

prot 参数指定映射期望的内存保护权限。它可以是 PROT_NONE，表示本次映射内存的页都不能访问（基本没有意义）；或者是以下一个或多个标志的按位或：

PROT_READ

页可以被读取

PROT_WRITE

页可以被写入

PROT_EXEC

页可以被执行

内存的保护权限不能与文件的打开模式相冲突。例如，如果程序以只读模式打开文件，prot 就不能够指定 PROT_WRITE。

保护标志、体系架构、与安全

POSIX 定义了四个保护标志位（读、写、执行、和无法访问），有些体系架构只支持部分子集。例如，处理器不区分读取和执行动作是很常见的。这种情况下，处理器可能只有一个“读”标志。在这样的系统中，PROT_READ 隐含 PROT_EXEC。直到目前为止，x86 体系架构就是这样一种系统。

当然，依赖于这样一种行为是不可移植的。如果想要执行映射后的代码，可移植的程序总是应该明确地设置 PROT_EXEC。

这种情况是缓冲区溢出攻击流行的一个原因：即使某个给定的映射并没有指定执行权限，处理器依然可能允许执行。

近来的 x86 处理器引入了 NX（不可执行）位，允许映射内存可以被读取但不可被执行。在这些新的系统中，PROT_READ 不再隐含 PROT_EXEC。

flags 参数指定映射的类型，以及相关的一些行为。它是以下值的按位或：

MAP_FIXED

指示 mmap()对待 addr 为必要条件而不仅仅是建议。如果内核无法把映射放到指定的地址，调用将失败。如果地址和长度参数与某个已存在的映射重叠，则重叠的页将被丢弃并被新的映射覆盖。由于这个选项需要知道进程地址空间的内部信息，它不是可移植的，不推荐使用。

MAP_PRIVATE

规定映射不共享。文件以“写入时复制”（copy-on-write）方式进行映射，进程对内存进行的任何修改都不会反应到实际的文件中，或者其它进程对同一文件的映射。

MAP_SHARED

与其它进程一起共享相同文件的映射。向映射内存写入数据等同于向文件中写入数据。从映射内存中读取数据也会反映出其它进程的写入。

MAP_SHARED 或者 MAP_PRIVATE 必须被指定，但不能同时指定。更多高级标志将在第

八章中讨论。

当你映射一个文件描述符时，文件的引用计数会增加 1。因此，你可以在映射文件完成后关闭文件描述符，你的进程仍然可以继续访问文件。相应的文件引用计数减 1 会在你解除文件映射时进行，或者当进程结束时进行。

作为一个例子，下面代码片断映射文件描述符 fd 到一个只读的映射中，从文件首字节开始延伸到 len 个字节：

```
void *p;
p = mmap (0, len, PROT_READ, MAP_SHARED, fd, 0);
if (p == MAP_FAILED)
    perror ("mmap");
```

图 4-1 显示了 mmap()的参数对映射的文件和进程地址空间的作用

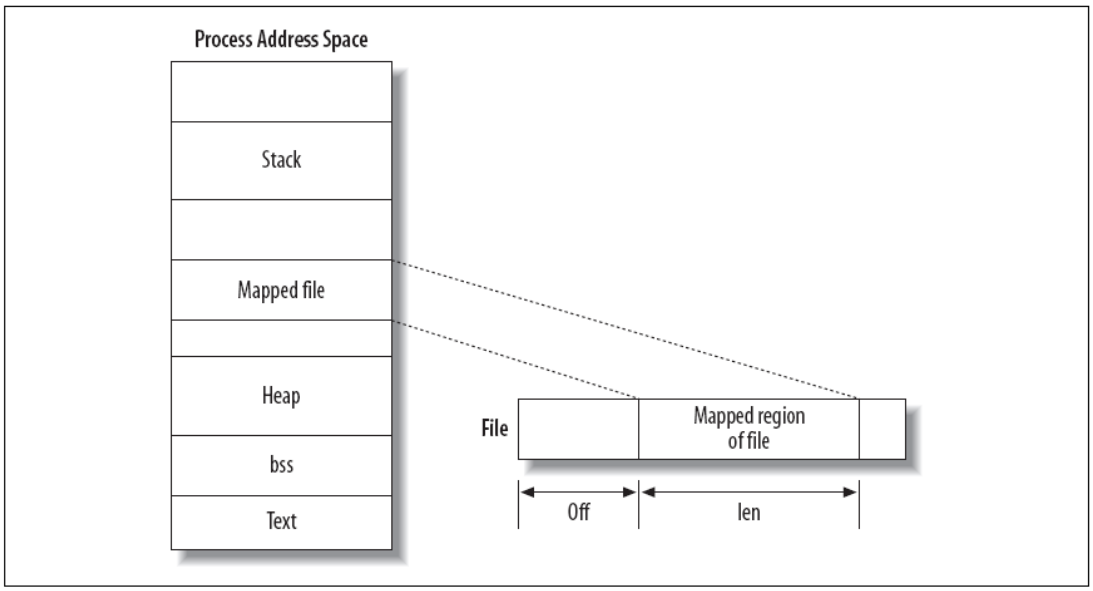


图 4-1. 映射文件到进程地址空间

页大小

页是内存中拥有特定权限和行为的最小单元。因此，页是内存映射的构造块，同时也是构成进程地址空间的块。

mmap()系统调用对页进行操作。addr 和 offset 参数都必须按页大小边界对齐。也就是说，它们的值必须是页大小的整数倍。

因此映射也是页的整数倍。如果调用者提供的 len 参数不是按页边界对齐的——可能因为文件大小本身就不是页大小的整数倍——则映射将被扩展到下一个完整的页。这部分增加的内存（最后有效字节到映射内存末尾）将被填充为 0。对这个区域的任何读取操作都返回 0；任何写入操作都不会影响到背后的文件，即使映射指定为 MAP_SHARED。只有原来的 len 字节会被写回到文件中。

sysconf( )：获得页大小的标准 POSIX 方法是使用 sysconf()，该函数可以获得各种系统特定的信息：

```
#include <unistd.h>
long sysconf (int name);
```

调用 sysconf()返回配置项 name 的值，或者当 name 非法时返回-1。错误时调用设置 errno 为 EINVAL。因为-1 可能是某些配置项的合法值（例如对于 limits，-1 表示无限制），在调用 sysconf()之前清除 errno，然后在调用之后再检查是明智的做法。

POSIX 定义_SC_PAGESIZE（以及一个同义_SC_PAGE_SIZE）为页大小（按字节）。因此，获取页大小是非常简单的：

```
long page_size = sysconf (_SC_PAGESIZE);
```

getpagesize()：Linux 也提供了 getpagesize()函数：

```
#include <unistd.h>
int getpagesize (void);
```

调用 getpagesize()同样也返回页的大小（按字节），使用比 sysconf()又更简单：

```
int page_size = getpagesize ( );
```

并不是所有的 Unix 系统都支持这个函数；在 POSIX 标准 1003.1-2001 修订版本中它被去除。这里列出它仅仅是为了讨论的完整性。

PAGE_SIZE：页大小也被静态存储在宏 PAGE_SIZE 中，在<asm/page.h>中定义。因此，获得页大小第三种可能的方法是：

```
int page_size = PAGE_SIZE;
```

但是和前两种方法不同，这种方法获得系统页大小是在编译时，而不是运行时。有一些体系架构支持不同页大小的多种机器类型，还有一些机器类型甚至本身就支持多种页大小！一个二进制文件应该能够运行在给定体系架构下的所有机器类型上——也就是，你应该可以只构建一次，然后在所有地方运行它。硬编码的页大小会使这种可能性失去。因此，你应该选择在运行时获得页大小。由于 addr 和 offset 通常使用 0，实现这个需求并不会太难。

此外，内核的未来版本在用户空间很有可能不再导出这个宏。我们在这里讲到它主要是因为它在当前 Unix 代码中的频繁出现。不管怎么样，你不应该在自己的代码中使用它，sysconf()是你的最佳选择。

返回值和错误代码

成功时 mmap()返回映射的地址；失败时 mmap()返回 MAP_FAILED，并设置 errno 为合适的值。调用 mmap()永远不会返回 0。

可能的 errno 值包括：

EACCESS

给定的文件描述符不是正常文件，或者文件的打开模式与 prot 或者 flags 冲突。

EAGAIN

文件被“文件锁”锁定。

EBADF

指定的文件描述符无效。

EINVAL

addr、len、或者 off 参数中的一个或多个非法。

ENFILE

到达打开文件的系统范围限制。

ENODEV

映射文件所在的文件系统不支持内存映射。

ENOMEM

进程没有足够的可用内存。

EOVERFLOW

addr + len 的结果超过了进程的地址空间大小。

EPERM

指定了 PROT_EXEC，但文件系统却以 noexec 挂载。

相关信号

有两个信号与映射区域相关：

SIGBUS

当进程试图访问一段不再合法的映射区域时产生——例如，文件在映射之后又被截断。

SIGSEGV

当进程试图向只读映射区域写入数据时产生。

munmap()

Linux 提供了 munmap()系统调用来移除通过 mmap()创建的映射：

```
#include <sys/mman.h>
int munmap (void *addr, size_t len);
```

调用 munmap()移除进程地址空间中地址 addr 开始（必须为页对齐），一直到 len 字节之间的所有映射页。一旦映射被移除，之前关联的内存区域将不再合法，任何试图访问的操作都将导致 SIGSEGV 信号。

通常，传递给 munmap()的参数就是之前调用 mmap()使用的 len 以及 mmap()的返回值。

成功时 munmap()返回 0；失败时返回-1，并设置 errno 为合适的值。唯一的标准 errno 值是 EINVAL，表示一个或多个参数非法。

作为一个例子，下面代码片断解除[addr, addr + len]之间的所有内存映射区域：

```
if (munmap (addr, len) == -1)
    perror ("munmap");
```

映射示例

让我们考虑一个使用 mmap()的简单示例程序，打印用户选择的文件到标准输出：

```
#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;
    }
```

这个程序中唯一不熟悉的系统调用应该只有 fstat()，我们会在第七章讲解它。现在你需要知道的只是 fstat()返回指定文件的信息。S_ISREG()宏则可以检查其中的某些信息，这样我们就能在映射文件之前确定指定的文件是正常文件（不是设备文件或者目录）。映射非正常文件的行为依赖于相关的设备。某些设备文件可以被映射；其它一些则不能够被映射，会设置 errno 为 EACCESS。

例子程序的其它部分都非常简单。程序的参数被传递进来一个文件，程序打开文件，确保它是一个正常文件，映射文件，关闭文件，一个字节一个字节地打印文件到标准输出，然后从内存中解除文件的映射。

mmap()的优点

通过 mmap()操作文件比标准 read()和 write()系统调用有许多优点，其中包括：

- 从内存映射文件中读取和写入数据避免了使用 read()和 write()系统调用时存在的额外复制，后者的数据必须复制到或者从用户空间缓冲区中复制。
- 撇开可能的页错误，从内存映射文件中读取或者写入数据不会发生任何系统调用或者上下文切换开销。它就是访问内存一样简单。
- 当多个进程映射相同的对象到内存时，数据在所有进程间共享。只读和共享可写映射都完全共享；私有可写的映射则共享所有尚未 COW（copy-on-write 写入时复制）的页。
- 在映射内存中寻址只需要简单的指针操作。完全不需要使用 lseek()系统调用。

基于以上理由，mmap()是许多应用的明智选择。

mmap()的缺点

当使用 mmap()时需要记住几点：

- 内存映射总是页大小的整数倍。因此，文件大小与页大小整数倍之间的差总是被浪费掉。对于小文件，映射所浪费的空间百分比可能非常大。例如，当页大小为 4KB 时，映射一个 7 字节的文件将浪费 4089 字节空间。
- 内存映射必须适合进程的地址空间。对于 32 位地址空间来说，大量的可变大小映射可能导致地址空间的碎片，使得寻找大的可用连续内存区域非常困难。这个问题在 64 位地址空间中则基本不存在。
- 在内核中创建和维护内存映射以及相关的数据结构有一定的开销。这点开销通常都被我们之前讨论过的双重复制引起的开销所抵销，特别是对于频繁访问的大文件。

由于以上原因，当映射文件很大时 mmap()能够获得最大的好处（这时候浪费的空间对于总空间来说基本可以忽略）；或者当映射文件的总大小正好是页大小的除数（这时候没有任何空间浪费）。

调整映射大小

Linux 提供 mremap()系统调用来扩大或者缩小指定映射的大小，这个函数是 Linux 系统特有的：

```
#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);
```

调用 mremap()扩大或者缩小区域[addr, addr + old_size)之间的映射到新的大小 new_size。根据进程地址空间可用内存空间和 flags 的值，内核在调整映射大小的同时可以选择移动映射。

flags 参数可以是 0 或者 MREMAP_MAYMOVE，指定内核在执行映射大小调整时，是否可以自由移动映射。如果内核可以移动映射，一个大的调整会更有可能成功。

返回值和错误代码

成功时 mremap()返回调整后新大小映射的内存地址；失败时它返回 MAP_FAILED，并设置 errno 为以下值之一：

EAGAIN

内存区域被锁定，不能调整大小。

EFAULT

指定范围内的某些页不是进程地址空间中的合法页，或者在重新映射指定页时出现问题。

EINVAL

参数非法

ENOMEM

指定范围的映射在不移动的情况下无法调整大小（没有设置 MREMAP_MAYMOVE），或者进程地址空间没有足够的可用内存。

glibc 库使用 mremap()来实现高效的 realloc()，这是一个重新调整 malloc()函数分配的内存块大小的接口。例如：

```
void * realloc (void *addr, size_t len)
{
    size_t old_size = look_up_mapping_size (addr);
    void *p;

    p = mremap (addr, old_size, len, MREMAP_MAYMOVE);
    if (p == MAP_FAILED)
        return NULL;
    return p;
}
```

只有在 malloc()分配是唯一匿名映射时，上面代码才能工作。无论如何，它是一个有用的例子。上面例子假设程序员已经实现了 look_up_mapping_size()函数。

GNU C 库确实使用 mmap()以及它的家族函数执行一些内存分配工作。我们会在第八章深入探讨这个主题。

改变映射的保护权限

POSIX 定义了 mprotect()接口，允许程序改变已存在内存映射区域的保护权限：

```
#include <sys/mman.h>
int mprotect (const void *addr,
              size_t len,
              int prot);
```

调用 mprotect()会改变[addr, addr + len)之间内存页的保护权限，addr 必须是页对齐的。prot 参数和 mmap()的 prot 参数一样，可以是：PROT_NONE、PROT_READ、PROT_WRITE 和 PROT_EXEC。这些参数不是附加的，如果一个内存区域是可读的，然后 prot 仅仅设置了 PROT_WRITE，则调用 mprotect()会使这段区域只能写。

在某些系统中，mprotect()只能操作之前通过 mmap()创建的映射内存区域。而在 Linux 中，mprotect()可以操作内存的任何区域。

返回值和错误代码

成功时 mprotect()返回 0；失败时返回-1，并设置 errno 为以下值之一：

EACCESS

内存不能够被设置为 prot 指定的保护权限。例如，如果你试图设置只读文件映射为可写时，会发生这个错误。

EINVAL

参数 addr 非法或者不是页对齐的。

ENOMEM

内核没有足够的可用内存来执行请求，或者指定内存区域的一个或多个页不是进程地址空间中合法的页。

使用映射同步文件

POSIX 提供了一个内存映射同步系统调用，它和第二章讨论的 fsync()系统调用具有相同的意义：

```
#include <sys/mman.h>
int msync (void *addr, size_t len, int flags);
```

调用 msync()把 mmap()映射内存的任何更改 flush 到磁盘中，同步映射文件与映射内存。确切地说，是把文件或者文件映射到内存地址 addr 到 len 字节处的那部分内容同步到磁盘。addr 参数必须是页对齐的，它通常是之前 mmap()调用的返回值。

没有调用 msync()，则不能保证脏(dirty)映射被写回到磁盘中，除非文件被解除映射。这

和 write()的行为不一样，write()的脏缓冲区属于写入进程的一部分，它排队等候写回至磁盘。而写入内存映射时，进程直接修改内核中页面缓存中的文件页，而不需要内核调度。内核可能不会在短时间内同步页面缓存和磁盘。

flags 参数控制同步操作的行为。它是以下值的按位或：

MS_ASYNC

指定同步操作异步进行。更新是经过调度的，但 msync()调用会立即返回，而不等待数据写回至磁盘。

MS_INVALIDATE

指定所有其它已缓存的映射拷贝无效。任何接下来对映射文件的访问，都能反映出对磁盘内容进行的最新同步。

MS_SYNC

指定同步操作同步进行。msync()调用会一直等待直到所有页面被写回至磁盘才返回。

必须指定 MS_ASYNC 或者 MS_SYNC，但不能同时指定二者。

使用非常简单：

```
if (msync (addr, len, MS_ASYNC) == -1)
    perror ("msync");
```

上面例子按异步方式同步文件映射区域[addr, addr + len)到磁盘中。

返回值和错误代码

成功时 msync()返回 0；失败时返回-1，并设置 errno 为以下值之一：

EINVAL

flags 同时设置了 MS_SYNC 和 MS_ASYNC、设置了三个合法标志之外的值、或者 addr 不是页对齐的。

ENOMEM

指定的内存区域（或部分）没有被映射。注意 Linux 在同步部分区域没有被映射时，会按 POSIX 的要求返回 ENOMEM，但 Linux 仍然会同步区域中其它合法的映射。

在 Linux 内核版本 2.4.19 之前，msync()返回 EFAULT 而不是 ENOMEM。

向映射提出建议

Linux 提供了 madvise()系统调用，让进程向内核给出自己想要怎样使用映射的提示。然后内核可以根据映射期望的使用方式进行优化。虽然 Linux 内核能够动态调节映射的行为，通常不显式的提供建议也能够进行优化。但显式地提供类似建议可以确保缓存和预读行为是最优的。

调用 madvise()建议内核内存映射 addr 开始、直到 len 字节之间的页的行为：

```
#include <sys/mman.h>
int madvise (void *addr,
             size_t len,
             int advice);
```

如果 len 为 0，内核将对 addr 开始的整个映射应用建议。advice 参数描述给内核的建议，可以是以下值之一：

MADV_NORMAL

应用对指定范围的内存没有特别的建议，按正常情况对待。

MADV_RANDOM

应用希望随机访问指定范围内的页。

MADV_SEQUENTIAL

应用希望顺序访问指定范围内的页，从低地址到高地址。

MADV_WILLNEED

应用在不久将访问指定范围内的页。

MADV_DONTNEED

应用在短时间内不会访问指定范围内的页。

内核对建议的响应以及实际行为的改变与具体的实现相关：POSIX 仅仅指示了建议的含义，没有规定任何必须的结果。当前的 2.6 版本内核对建议的响应行为如下：

MADV_NORMAL

内核的行为和通常一样，执行适当数量的预读。

MADV_RANDOM

内核禁用预读、每次物理读取操作时只读取最小数量的数据。

MADV_SEQUENTIAL

内核执行激进的预读。

MADV_WILLNEED

内核启动预读，开始读取指定的页到内存中。

MADV_DONTNEED

内核释放指定页相关的所有资源，丢弃任何脏页和尚未同步的页。随后对映射数据的访问将导致数据从文件中读取到页。

典型的使用如下：

```
int ret;
ret = madvise (addr, len, MADV_SEQUENTIAL);
if (ret < 0)
    perror ("madvise");
```

上面调用指示内核进程希望顺序访问内存范围[addr, addr + len)。

预读

当 Linux 内核从磁盘中读取文件时，它会执行被称为预读的优化操作。预读就是当请求读取文件的指定数据块时，内核同时读取接下来的文件数据块。如果随后有对那块数据的读取请求（顺序读取文件时），内核可以立即返回请求的数据。由于磁盘有磁道缓冲区（基本上硬盘会执行自己内部的预读），并且文件通常在磁盘中也是顺序存储，预读操作的优化开销相当小。

执行一定程度的预读通常都是有利的，但优化的程度依赖于到底执行多少预读。频繁访问的文件可以从更大的预读窗口中获得更多的好处，而一个随机访问的文件则会发现预读只

是毫无用处的开销。

正如第二章“深入内核内部”所讨论的，内核根据预读窗口的命中率，动态调整预读窗口的大小。高命中率意味着更大的预读窗口可能是有利的；低命中率则表明更小的窗口更加合适。madvise()系统调用允许应用程序来改变预读窗口的大小。

返回值和错误代码

成功时 madvise()返回 0；失败时返回-1，并设置 errno 为以下值之一：

EAGAIN

内核内部资源不可用（或许是内存），进程可以重试。

EBADF

区域已存在，但并没有映射文件。

EINVAL

参数 len 为负数、addr 不是页对齐的、advice 参数非法、页被锁定、或者页同时指定 MADV_DONTNEED 和共享。

EIO

使用 MADV_WILLNEED 时发生内部 I/O 错误。

ENOMEM

指定区域不是进程地址空间中合法的映射、或者指定了 MADV_WILLNEED，但指定区域没有足够的内存进行页操作。

向普通文件 I/O 提出建议

在前一小节，我们讨论了如何对内存映射提出建议。在这一节，我们将讨论如何向内核对普通文件 I/O 提供建议。Linux 提供了两个接口来提供此类建议：posix_fadvise()和 readahead()。

posix_fadvise()系统调用

第一个提供建议的接口是 posix_fadvise()，正如名字所指示的，它由 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()向内核提供文件描述符 fd 的[offset, offset + len)范围的使用建议。如果 len 为 0，则建议应用到范围[offset, 文件的长度]。通常的用法是指定 len 和 offset 为 0，这样建议会应用到整个文件范围。

可用的 advice 和 madvise()类似，advice 必须是以下值之一：

POSIX_FADV_NORMAL

应用对指定范围的文件没有特别的建议，按正常情况处理。

POSIX_FADV_RANDOM

应用希望以随机方式访问指定范围的文件。

POSIX_FADV_SEQUENTIAL

应用希望以顺序方式访问指定范围的文件，从低地址到高地址。

POSIX_FADV_WILLNEED

应用在不久将访问指定范围的文件。

POSIX_FADV_NOREUSE

应用在不久将访问指定范围的文件，但只访问一次。

POSIX_FADV_DONTNEED

应用在短时期内不会访问指定范围的文件。

和 madvise()一样，对给定建议的实际反应依赖于具体的实现——甚至不同版本的 Linux 内核也可能有不一样的反应。下面是当前内核版本的响应：

POSIX_FADV_NORMAL

内核按正常情况处理，执行适当数量的预读。

POSIX_FADV_RANDOM

内核禁止预读，每次物理读取操作只读取最少数量的数据。

POSIX_FADV_SEQUENTIAL

内核执行激进的预读，加倍预读窗口的大小。

POSIX_FADV_WILLNEED

内核发起预读，开始读取指定的页到内存中。

POSIX_FADV_NOREUSE

当前，内核的行为和 POSIX_FADV_WILLNEED 一样；将来的内核可能会执行额外的优化来达到“使用一次”的行为。这个建议在 madvise()中没有对应物。

POSIX_FADV_DONTNEED

内核驱逐指定范围内的页缓存的所有数据。注意这个建议不像上面几个，它的行为和 madvise()对应建议的行为并不一样。

作为一个例子，下面代码片断向内核建议，文件描述符 fd 代表的整个文件都将以随机非顺序方式访问：

```
int ret;
ret = posix_fadvise (fd, 0, 0, POSIX_FADV_RANDOM);
if (ret == -1)
    perror ("posix_fadvise");
```

返回值和错误代码

成功时 posix_fadvise()返回 0；失败时返回-1，并设置 errno 为以下值之一：

EBADF

指定的文件描述符非法。

EINVAL

指定的 advice 非法、指定的文件描述符指向管道、或者指定的 advice 不能应用到指定的文件。

readahead()系统调用

posix_fadvise()是 Linux 内核 2.6 新增的系统调用，在这之前，readahead()系统调用用来提供等同于 POSIX_FADV_WILLNEED 建议的功能。和 posix_fadvise()不一样，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()把文件描述符 fd 范围[offset, offset + count)的数据装载到页缓存中。

返回值和错误代码

成功时 readahead()返回 0；失败时返回-1，并设置 errno 为以下值之一：

EBADF

指定的文件描述符非法。

EINVAL

指定的文件描述符没有映射到支持预读的文件。

建议是廉价而有用的

许多通用的应用程序只需要向内核提供一点点好的建议，就能够轻易地获益。这样的建议对于减轻 I/O 负担有非常大的作用。在当今磁盘如此慢，而现代处理器又如此快的情况下，任何小的帮助以及好的建议，都大有帮助。

在读取文件的大块数据之前，进程可以先向内核提供 POSIX_FADV_WILLNEED 建议，指示内核读取文件到页缓存。I/O 操作会在后台异步进行。当应用最后需要访问文件时，I/O 操作就可以不产生阻塞 I/O 而直接完成了。

反过来，在读取或者写入许多数据以后——例如，持续流媒体到磁盘——进程可以提供 POSIX_FADV_DONTNEED 建议，指示内核从页缓存中逐出指定的文件数据块。大的流操作可能持续填充页缓存。如果应用永远不再需要访问这些数据，则意味着页缓存填充着多余的数据，潜在地损失了更多有用的数据。因此，对于流媒体应用来说，定期地请求内核清理页缓存的数据是有意义的。

进程希望读取整个文件时，可以提供 POSIX_FADV_SEQUENTIAL 建议，指示内核执行激进的预读。反过来，如果进程需要随机访问文件，来回地 Seek，则可以提供 POSIX_FADV_RANDOM 建议，指示内核预读没有任何作用，只是无谓的开销。

Synchronized、Synchronous、和 Asynchronous 操作

Unix 系统自由地使用术语 synchronized(同步)、nonsynchronized(非同步)、synchronous(同步)、和 asynchronous(异步)，完全不管它们会引起混淆——在英语中，“synchronous”和“synchronized”基本没有区别。

*synchronous(同步)*写操作在数据被存储到内核缓冲区之前不会返回。synchronous(同步)读操作则在读取数据被存储到用户空间应用程序提供的缓冲区之前不会返回。另一方面，*asynchronous(异步)*写操作则可能在数据离开用户空间之前就返回；asynchronous(异步)读操作则可能会在数据可用之前返回。也就是说，操作仅仅排队等待以后执行。当然在这种情况下，必须有某种机制来确定操作已经实际完成，以及操作完成的程度。

*synchronized(同步的)*操作比仅仅 synchronous(同步)操作更加受限也更加安全。同步的写操作 flush 数据到磁盘，确保磁盘上的数据总是与内核对应的缓冲区同步。而同步的读操作总是返回最新的数据，可能从磁盘而不是内核缓冲区中返回。

总而言之，术语 synchronous(同步)和 asynchronous(异步)指的是 I/O 操作在返回前是否等待某些事件（例如数据的存储）；而术语 synchronized(同步的)和 nonsynchronized(非同步的)则是明确的指定某个事件必须发生（例如写入数据到磁盘）。

通常 Unix 写操作是 synchronous(同步)和 nonsynchronized(非同步的)；读操作既是 synchronous 又是 synchronized 的。对于写操作来说，所有这些特性的组合都是可能的，如表 4-1 说明的：

	Synchronized	Nonsynchronized
Synchronous	写操作在数据 flush 到磁盘之前不会返回。这是打开文件时指定 O_SYNC 标志的行为。	写操作在数据存储到内核缓冲区之前不会返回。这是正常情况下的行为。
Asynchronous	写操作在请求排队后立即返回。一旦写操作最终得到执行，数据将确保写入到磁盘。	写操作在请求排队后立即返回。一旦写操作最终得到执行，数据将至少确保存储到内核缓冲区。

表 4-1. 写操作的同步性

读操作总是 synchronized(同步的)，因为读取过期的数据没有任何意义。不过读操作也可以是 synchronous 或者 asynchronous 的，如表 4-2 所示：

	Synchronized
Synchronous	读操作在磁盘中最新的数据存储到指定的缓冲区之前不会返回（这是通常的行为）。
Asynchronous	读操作在请求排队之后立即返回，但当读操作最终得到执行时，返回的数据确保是最新的。

表 4-2. 读操作的同步性

在第二章，我们讨论了怎样使写操作成为 synchronized(同步的)——通过 O_SYNC 标志，以及如何确保所有 I/O 在指定的时间都同步到磁盘——通过 fsync()以及相关调用。现在，让我们来看看怎样使读和写操作异步进行。

Asynchronous I/O

执行异步 I/O 需要内核在非常低的层次提供支持。POSIX 1003.1-2003 定义了 aio 接口，Linux 实现了该接口。aio 库提供了一组函数来提交异步 I/O 以及接收 I/O 完成通知。

```
#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 仅仅对那些以 O_DIRECT 标志打开的文件支持 aio。要对不以 O_DIRECT 标志打开的普通文件执行异步 I/O，我们需要从内部入手，实现我们自己的解决方案。没有内核的支持，我们只能希望近似地实现异步 I/O，得到与实际类似的结果。

首先，让我们看一看为什么应用开发者会需要异步 I/O：

- 执行 I/O 而不阻塞
- 分离 I/O 排队、提交 I/O 到内核、以及接收操作完成通知。

第一点和性能有关，如果 I/O 操作从不阻塞，则 I/O 的等待开销将接近 0，进程就不会被 I/O 束缚；第二点则和程序编写相关，它仅仅是另一种处理 I/O 的方式而已。

达到这些目标最常用的方法是使用线程（调度相关的问题将在第五章和第六章彻底讨论）。这个方法涉及到以下的编程任务：

1. 创建一个线程池，其中的“工作线程”处理所有 I/O。
2. 实现一组接口，用于放置 I/O 操作到工作队列中。
3. 让上面的每个接口都返回一个 I/O 描述符，唯一标识相关联的 I/O 操作。每个工作线程从队列开头抓取 I/O 请求并提交它们，然后等待 I/O 操作的结束。
4. I/O 操作结束后，把操作结果（返回值、错误代码、读取数据）放置到结果队列。
5. 实现一组接口，用于从结果队列接收状态信息，使用最初返回的 I/O 描述符来标识每个 I/O 操作。

这样就提供了类似 POSIX aio 接口的行为，虽然带来一定的线程管理开销。

I/O 调度器与 I/O 性能

在现代系统中，磁盘和系统其它部分之间的相对性能差距非常大——而且越来越大。磁盘性能最坏的部分是把读/写头从磁盘的某个位置移动到另一个位置，这个操作被称为 seek（寻址）。在多个操作以少数几个处理器周期来度量的情况下（每个可能只需要 1/3 纳秒），一个单独的磁盘寻址平均花费就要超过 8 毫秒——这还是个小数目，但它还是比一个单独的处理器周期慢 2 千 5 百万倍。

正因为磁盘驱动器和系统其它部分之间这种性能上的巨大差距，按照 I/O 请求发起的顺序发送 I/O 操作到磁盘是非常拙劣而低效的。因此，现代操作系统内核实现了 I/O 调度器，通过调整 I/O 请求处理的顺序、以及 I/O 处理的时间，来最小化磁盘寻址的数量和寻址的大小。I/O 调度器非常努力地减少磁盘访问相关的性能损失。

磁盘寻址

要理解 I/O 调度器的角色与任务，我们必须首先掌握一些背景知识。硬盘数据的寻址使用熟悉的基于几何的寻址方法：柱面值、读/写头值、和扇区值，这也被称为 CHS 寻址。硬盘由多个盘片组成，每个盘片又由一个磁盘、转动轴、和读/写头组成。你可以把每个盘片想像成一部 CD 机，磁盘中的盘片组合就是一叠 CD 机。每个盘片都被划分成环状的圆形磁道，和 CD 唱片一样。然后每个磁道又被划分为许多扇区。

要定位到磁盘指定单元的数据，驱动器需要三个信息：柱面值、读/写头值、和扇区值。柱面值指定数据所在的磁道。如果把多个盘片组合在一起，则指定的磁道就通过每个盘片组成了一个柱面。换句话说，柱面是由与每个盘片中心相同距离的一组磁道组成的。读/写头值表示要使用的那个读/写头（因此也就确定了哪个盘片）。现在寻址就已经缩小到了一个盘片的一个磁道上。磁盘然后使用扇区值来标识磁道上的准确扇区。这时候寻址结束：磁盘知道去哪个盘片、哪个磁道、哪个扇区寻找数据。于是它就可以把读/写头定位到准确盘片的准确磁道上，然后从需要的扇区读取或者写入数据。

幸运的是，现代硬盘驱动器并不强制计算机按照柱面、读/写头、扇区来和磁盘通信。相反，现代的硬盘驱动器把每个[柱面/读写头/扇区]的组合映射到一个唯一的块编号（也称为物理块或者设备块）——每个块都映射到一个特定的扇区。然后现代操作系统就可以通过这些块编号来寻址硬盘驱动器——这个过程被称为逻辑块寻址(LBA)——硬盘驱动器内部自动把块编号转换为正确的 CHS 地址。尽管没有保证，块－CHS 的映射趋向于顺序的：物理块 n 与逻辑块 n+1 在磁盘上也趋向于物理相邻。这种顺序映射是非常重要的，我们在后面会看到这一点。

而文件系统则仅仅由软件组成。它们操作自己的单元：逻辑块（有时候也叫文件系统块、或者就叫块）。逻辑块大小必须是物理块大小的整数倍。换句话说，文件系统的逻辑块映射到一个或多个磁盘物理块上。

I/O 调度器的工作

I/O 调度器执行两个基本的操作：合并和排序。合并是组合两个或多个相邻的 I/O 请求到一个单独的 I/O 请求的过程。考虑两个请求的情况，其中一个从磁盘块 5 读取、另一个从磁盘块 6 到 7 读取。这些请求可以被组合到一个单独的 I/O 请求：读取磁盘块 5 到 7。I/O

读取的总数量是一样的，但 I/O 操作的数目则已经减半。

排序是两个操作中更为重要的一个，它按块顺序的升序重新排列所有的未决 I/O 请求。例如，假定几个 I/O 请求分别操作块 52、109、和 7，I/O 调度器会排列这些请求为新的顺序：7、52、和 109。如果接着又有一个块 81 的操作请求，它会被插入到块 52、109 的请求之间。I/O 调度器然后就可以按照块 7、52、81、最后 109 的顺序分派 I/O 请求。

按这种方式，磁盘头的移动最小。磁盘头以平坦、线性的方式移动，而不是随机偶然性的移动（从这里移到那里又返回，在整个磁盘到处寻址）。由于寻址是磁盘 I/O 最昂贵的部分，I/O 性能因此得到了提升。

帮助读取操作

每个读取请求必须返回最新的数据。因此，如果请求数据不在页缓存中，读取过程就必须阻塞直到数据能够从磁盘中读取——这是一个可能非常耗时的操作。我们把这个性能影响称为 *read latency*。

一个典型的应用可能会在短时期内发起多个 I/O 读取请求。由于每个请求都是各自同步的，后面的请求依赖于之前请求的完成。考虑读取某个目录下的所有文件。应用打开第一个文件、从中读取一块数据、等待数据、读取另一块数据，一直进行直到整个文件读取完成。然后应用重新开始读取下一个文件。这样请求就变成连续的：一个后续的请求不能在前一个请求完成之前发起。

相比较写请求则没有这么刻板，写请求（默认非同步的）不需要先发起任何磁盘 I/O 就能进行。因此从用户空间应用的角度来看，写操作需要的是流，不会受到磁盘性能的影响。这种流的行为仅仅与读操作的问题有关：写入流时，它们可能占据内核和磁盘。这个现象被称为“写饿死读”(writes-starving-reads)问题。

如果 I/O 调度器总是按插入顺序排列新的请求，则有可能会导致某些请求永远阻塞。考虑我们之前的例子，如果新的请求持续地发起到块 50，那对块 109 的请求就永远得不到处理。因为 read latency 是至关重要的，这种行为会严重影响系统性能。因此，I/O 调度器必须采用某种机制来避免饥饿。

一个简单的方法——这也是 Linux 内核 2.4 版本 I/O 调度器所采用的：Linus Elevator——是当队列中存在许多老的请求时，停止插入排序。这种方法权衡了总性能与单个请求的公平性，对于读取操作来说，改善了 latency。问题在于这种启发式的方法过于单纯。Linux 内核 2.6 承认了这一点，并把 Linus Elevator 替换成了几个新的 I/O 调度器。

最终期限(Deadline)I/O 调度器

Deadline I/O 调度器被引入来解决内核 2.4 的 I/O 调度器、以及传统的 elevator 算法的问题。Linus Elevator 维护着一个未决 I/O 请求的排序列表。队列开头的 I/O 请求将是下一个处理的对象。Deadline I/O 调度器保留了这个队列，但同时引入了两个额外的新队列：读 FIFO 队列、和写 FIFO 队列。两个队列中的元素按提交时间排序（第一个将首先出列）。读 FIFO 队列包含读请求；而写 FIFO 队列则包含写请求。FIFO 队列中的每个请求都被分配一个过期时间值。读 FIFO 队列的到期时间是 500 毫秒；写 FIFO 队列的到期时间则是 5 秒。

当一个新的 I/O 请求提交时，它按顺序插入到标准的队列中，同时被放置到它对应（读或写）的 FIFO 队列尾。正常情况下，从标准的已排序队列头发送 I/O 请求给硬盘驱动器。这样通过最小化寻址操作提高了总的吞吐量，因为队列按块编号排序。

但是当某个 FIFO 队列头的请求超过它关联的过期时间时，I/O 调度器将停止从标准队列分派 I/O 请求，并开始处理有超期请求的那个队列——此 FIFO 队列头那个过期的 I/O 请求将得到处理，并继续处理接下来的几个过期 I/O 请求。I/O 调度器只需要检查并处理队列头已过期的那些请求，因为那些都是最老的 I/O 请求。

按这种方式，Deadline I/O 调度器可以对 I/O 请求强制执行软最终期限。尽管它不保证一个 I/O 请求会在过期之前得到处理，但 I/O 调度器通常都能在过期时间附近处理这些 I/O 请求。因此 Deadline I/O 调度器继续提供了一个好的全局吞吐量，而又不会长时间饥饿任何一个 I/O 请求；同时由于读请求指定了更短的过期时间，写饥饿读的问题也最小化了。

预测(Anticipatory)I/O 调度器

Deadline I/O 调度器的行为很不错，但并不完美。回想我们讨论过的读取依赖问题。使用 Deadline I/O 调度器时，一系列读取请求中的第一个短期内就会被处理，在它的过期时间到达或者到达之前，然后 I/O 调度器继续处理已排列队列中接下来的请求——到目前为止工作得很好。但是假设应用接下来忽然发起另一个读取请求呢？最后到达它的过期时间，I/O 调度器会把它提交到磁盘，这就需要 seek 然后处理该请求，然后 seek 回到原来的地方继续处理已排序队列中的请求。这种 seek 操作有时候会继续，因为许多应用都有这个行为。Read latency 仍然最小化，但总吞吐量却并不怎么好，因为读取请求持续进来，磁盘就需要不停地来回 Seek 以处理这些请求。如果磁盘能够继续等待另一个请求，而不是马上回去处理已排序队列中的请求，性能就能够得到提升。但不幸的是，在应用被调度并提交下一个相关的读取请求时，I/O 调度器已经改变处理了。

当发起依赖读取请求（每个新的读取请求只有在前一个返回之后才会发起）时问题同样也会产生。当应用接收读取返回数据时，会发起下一个读取请求，I/O 调度器却已经在处理其它请求了。这样每个读取操作都需要两个浪费的 Seek：磁盘 seek 到读取位置、处理请求、然后 seek 回去。如果能有其它办法让 I/O 调度器知道（也就是预测）另一个相同磁盘部分的读取请求马上就会被提交，而不需要来回 seek，调度器就可以等待预测到的下一个读取请求。从而省下这些可怕的每个都需要许多毫秒的 seek 操作。

这正是预测 I/O 调度器执行的操作。它开始启动时和 Deadline I/O 调度器一样，但增加了预测机制。当一个读取请求提交时，预测 I/O 调度器和通常一样按最后期限处理读取请求。但是和 Deadline I/O 调度器不同的是，预测 I/O 调度器接下来会等待六毫秒，而不做任何事情。应用程序在这六毫秒中，很有可能会发起另一个相同文件系统位置的读取请求。如果确实是这样，请求会立即被处理，然后预测 I/O 调度器又继续等待六毫秒。如果六毫秒没有读取请求，预测 I/O 调度器就确定自己猜测错误，并返回到它上一步处理请求的地方（例如处理标准排序队列）。如果能够预测到许多请求，就能省下许多时间——每个请求两个昂贵的 seek 操作。由于大多数读取操作都是依赖的，预测赢得了许多时间。

CFQ I/O 调度器

完全公平排队(CFQ：Complete Fair Queuing) I/O 调度器同样为了达到类似的目标，但却采用了不同的方法。使用 CFQ 时，每个进程都分配一个自己的队列，并且每个队列都分配一个时间片。I/O 调度器按循环方式依次访问每个队列，从队列中处理请求直接时间片耗尽，或者直到没有剩余的请求。在后一种情况中，CFQ I/O 调度器会继续空闲一段时间——默认 10ms——等待队列中到来新的读取请求。如果预测成功，I/O 调度器就避免了 seek。如果没有测试成功，则等待是无益的，然后调度器继续下一个进程的队列。

在每个进程的队列中，同步请求（例如读）比非同步请求更优先。按这种方式，CFQ 优先读操作，防止了写饥饿读问题。因为每个进程都设置了队列，CFQ I/O 调度器对所有进程是公平的，同时仍然提供了很好的总吞吐量。

CFQ I/O 调度器非常适合大工作量的 I/O，这是非常好的选择。

Noop I/O 调度器

Noop I/O 调度器是最基本的调度器。它不执行任何排序，只简单的合并请求。它只用在特定的设备上，它们不需要自己的请求排序。

选择并配置自己的 I/O 调度器

在启动时可以通过内核命令行参数 iosched 来选择默认的 I/O 调度器。合法的选项有：as、cfq、deadline、和 noop。I/O 调度器同样可以在运行时，对每个设备进行选择，通过/sys/block/device/queue/scheduler，其中的 device 是需要设置的块设备。读取这个文件返回当前 I/O 调度器；写入上述合法选项到文件中则设置 I/O 调度器。例如，要设置设备 hda 为 CFQ I/O 调度器，可以这样做：

```
# echo cfq > /sys/block/hda/queue/scheduler
```

目录/sys/block/device/queue/iosched 包含的文件，允许系统管理员获得和设置 I/O 调度器相关的参数。确切的选项依赖于当前 I/O 调度器，改变任何一个设置都需要 root 权限。

好的程序员编写的程序并不会知道底层的 I/O 子系统。但无论如何，这个子系统的知识可以帮助你编写更优化的代码。

优化 I/O 性能

由于磁盘 I/O 相对于系统其它部分的性能来说如此慢，I/O 是现代计算的重要部分，最大化 I/O 性能至关重要。

最小化 I/O 操作数量（组合许多小的操作到更少的大操作）、执行块大小对齐 I/O、使用用户缓冲、利用高级 I/O 技术，例如向量 I/O、定位 I/O、异步 I/O，都是系统编程时需要考虑的重要步骤。

但是任务紧急和 I/O 至关重要的应用，仍然可以采用额外的技巧来最大化性能。如前面所说，尽管 Linux 内核利用高级 I/O 调度器最小化可怕的磁盘 seek，用户空间应用同样可以采用类似的方式，进一步提高性能。

用户空间 I/O 调度

I/O 强烈的应用发起大量的 I/O 请求，如果希望提高性能，可以排序与合并自己未决的 I/O 请求，执行 Linux I/O 调度器相同的工作。

I/O 调度器会按块序号排序请求、最小化 Seek、并允许磁盘头按照平滑、线性方式移动，但为什么要执行相同的工作两次？考虑一个应用发起大量未排序 I/O 请求，这些请求按一般的随机顺序到达 I/O 调度器的队列。I/O 调度器在把它们提交到磁盘之前，排序并合并请求

——但是在请求提交到磁盘的同时应用仍然产生 I/O 并提交请求。I/O 调度器只能排序一小部分的请求——例如，这个应用的几个请求，然后其它未决的请求。应用的每一小批请求都能被整齐的排序，但整个队列以及任何未来的请求都不是其中的一部分。

因此，如果应用产生许多请求——特别是如果请求磁盘全部的数据——在提交请求前先排序是有益的，能够确保它们按期望的顺序到达 I/O 调度器。

但是用户空间应用不像内核，有些数据并不能访问。在 I/O 调度器内部的最底层，请求已经用物理磁盘块指定。对它们排序非常简单。但是在用户空间，请求是通过文件和偏移量来指定的。用户空间应用必须检测某些信息，并对文件系统的布局作出一些高级的推测。

给定一组 I/O 请求的列表，要确定对 Seek 操作最友好的顺序，用户空间应用有几个选择，可以基于以下选项排序：

- 完全路径
- inode 值
- 文件的物理磁盘块

每个选项都有一定的折衷，我们每个都简短地看一下：

按路径排序。按路径排序是最简单的，但也是最无效的近似于块排序的方法。由于大多数文件系统使用的布局算法，每个目录的文件——以及同一父目录下的所有子目录——倾向于在磁盘中相邻。相同目录下在相近时间内创建的文件则更增加了这个特性的可能。

因此按路径排序，粗略地近似于文件在磁盘中的物理位置。相同目录下的两个文件位于相邻位置的机会比文件系统不同部分的两个文件高很多。按路径排序这种方法的缺点是它无法考虑碎片：文件系统碎片越多，按路径排序的作用就越弱。即使忽略碎片，按路径排序也仅仅是近似于实际的块排序。按路径排序的好处则是它至少对所有文件系统都适用。无论文件如何布局，采用按路径排序都至少有一定的精确度，而且它也是一种非常容易执行的排序。

按 inode 排序。inode 是 Unix 结构，包含单个文件相关的元数据：如文件大小、权限、拥有者等等。一个文件的数据可能存在多个物理磁盘块，但每个文件却只有一个 inode。我们在第七章会深入讨论 inode。现在你只需要知道两个事情：每个文件都有一个相关联的 inode；并且 inode 被赋予一个唯一的数值。

按 inode 排序比路径排序要更好，假设两个文件存在下面关系：

文件 i 的 inode 值 < 文件 j 的 inode 值

则通常意味着：

文件 i 的物理块 < 文件 j 的物理块

这对于 ext2 和 ext3 这样的 Unix 风格的文件系统来说是肯定正确的。文件系统不采用实际 inode 也是可能的，但 inode 值仍然是首选的近似值。

通过 stat()系统调用可以获取 inode 值，同样在第七章会有讨论。有了每个 I/O 请求的文件相关联的 inode 值，就可以按 inode 值对 I/O 请求进行升序排序。

下面是一个简单的程序，打印指定文件的 inode 值：

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

/*
* get_inode - returns the inode of the file associated
```

```
 * with the given file descriptor, or -1 on failure
 */
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;
}
```

get_inode()函数很容易就能适用于你自己的程序。

按 inode 值排序有几个优点：inode 值很容易获取、很容易排序、同时也非常接近于物理文件布局。主要的缺点是碎片降低了近似程度，而且这个接近只是猜测，对于非 Unix 文件系统则更加不准确。无论如何，按 inode 值排序是用户空间最常用的 I/O 请求调度方法。

按物理块排序。设计你自己的 elevator 算法时，最佳的方法当然是按物理磁盘块排序。如前面所讨论的，每个文件被分割为逻辑块，逻辑块是文件系统最小的分配单元。逻辑块的大小与文件系统相关；每个逻辑块映射到一个单独的物理块。我们因此可以首先获得文件的逻辑块，然后找出它映射的物理块，再按物理块进行排序。

内核提供了从文件逻辑块获取物理磁盘块的方法。通过 ioctl()系统调用和 FIBMAP 命令来完成，第七章有深入讨论。

```
        ret = ioctl (fd, FIBMAP, &block);
        if (ret < 0)
```

```
        perror ("ioctl");
```

fd 是请求文件的文件描述符，block 是逻辑块。成功返回时，block 的值将替换成物理块号。传入的逻辑块号以 0 开始索引并与文件相关。也就是说，如果一个文件由八个逻辑块组成，则合法的逻辑块号是 0 ~ 7。

因此逻辑块到物理块的映射需要两个步骤。首先，我们必须确定指定文件的块数量，这可以通过 stat()系统调用来完成；然后对于每个逻辑块，我们必须发起一个 ioctl()调用来找到相应的物理块。

下面是一个例子程序，找到命令行参数指定文件的物理块：

```
#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 - for the file associated with the given fd, returns
* the physical block mapping to logical_block
*/
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 - returns the number of logical blocks
* consumed by the file associated with fd
*/
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 - for each logical block consumed by the file
* associated with fd, prints to standard out the tuple
* "(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;
}
```

由于文件往往是相邻的，并且对 I/O 请求进行每个逻辑块排序也更加困难，对指定文件第一个逻辑块进行排序会更加有意义。因此，get_nr_blocks()并不一定需要，我们的应用可以基于 get_block(fd, 0)的返回值进行排序。

FIBMAP 的缺点是它需要 CAP_SYS_RAWIO——也就是 root 权限。因此，非 root 权限的应用不能使用这个方法。此外，虽然 FIBMAP 命令是标准的，它的实际实现却依赖于具体的文件系统。常见的 ext2 和 ext3 文件系统支持 FIBMAP，但也有许多文件系统并不支持它。ioctl()在 FIBMAP 不被支持时会返回 EINVAL。

在这个方法的所有优点中，按文件的实际物理磁盘块来排序就是你所希望的排序方式。即使你仅仅基于一个块来排序文件的所有 I/O 请求，这个方法也非常接近于最优的排序。然而需要 root 权限，确实对许多应用不适用。

总结

通过这三章的课程，我们已经接触到了 Linux 文件 I/O 的所有方面。在第二章，我们查看了 Linux 文件 I/O 的基本部分——也是 Unix 编程的根本——系统调用 read()、write()、open()、和 close()等。在第三章，我们讨论了用户空间缓冲以及标准 C 库的缓冲实现。在本章，我们讨论了许多高级 I/O，从最强大最复杂的 I/O 系统调用，到优化技术，以及避免磁盘 seek。

在接下来的两章中，我们将查看进程管理：创建、销毁、以及管理进程。前进！

第五章 进程管理

正如第一章提到的，进程是 Unix 系统中继文件之后最基本的抽象。进程是执行中的目标代码——动态、活泼、正在运行的程序——进程不仅仅是汇编语言；它还包含数据、资源、状态、和一个虚拟的计算机。

在本章，我们将查看进程的基础，从进程创建到终止。从 Unix 最早时候起，进程基本就相对保持不变。进程管理的长寿和前瞻性的思想，体现了 Unix 最初设计的优秀和光芒。Unix 采取了一个有趣而少见的方式，分离了创建一个新进程与装载一个新的二进制镜像。尽管这两个任务多数时候都前后顺序执行，对他们进行分离允许对每个任务更加自由地处理和演变。这种方式在今天依然保持不变，多数操作系统提供一个单独的系统调用来启动一个新程序，而 Unix 则需要两个：fork 和 exec。不过在我们讲解这两个系统调用之前，先进一步考察一下进程本身。

进程 ID

每一个进程都有一个唯一标识：进程 ID（经常简称 pid）。pid 在任何时间点都保证是唯一的。也就是说，在时间 t0 上只能有一个进程的 pid 是 770（如果存在 pid 为 770 的进程的话），但不能保证在时间 t1 一定不存在另一个 pid 为 770 的不同进程。不过本质上来讲，大部分代码都假设内核不会重用进程 ID——马上我们就会看到，这是一个相当安全的假设。

idle（空闲）进程的 pid 为 0，它是在没有其它进程可运行时内核“运行”的一个进程。内核在系统启动之后执行的第一个进程被称为 init 进程，拥有 pid1。通常 Linux 中的 init 进程就是 init 程序。我们使用术语 init 来同时表示内核运行的初始化进程，以及实现这个目标的特定的 init 程序。

除非用户明确地告诉内核运行哪个进程（通过 init 内核命令行参数），内核必须自己确定一个合适的 init 进程——这是个罕见的内核规定。Linux 内核尝试四个可执行文件，按以下顺序依次进行：

1. /sbin/init：init 进程的首选也是最可能的位置。
2. /etc/init：init 进程的另一个很可能的位置。
3. /bin/init：init 进程可能存在的位置。
4. /bin/sh：Bourne Shell 的位置，当内核查找 init 进程失败时会运行它。

上面第一个存在的程序将被作为 init 进程运行。如果所有四个进程都执行失败，Linux 内核会停止系统。

内核将控制交给 init 进程，然后 init 进程处理启动进程的剩余部分。典型地，包括初始化系统、启动不同的服务、以及启动登录程序。

进程 ID 分配

默认情况下，内核强制最大进程 ID 数值为 32768。这是为了与老的 Unix 系统保持兼容性，后者使用更小的 16 位类型表示进程 ID。系统管理员可以通过/proc/sys/kernel/pid_max 设置最大进程值，通过减少兼容性来获得更大的 pid 空间。

内核按照严格的线性方式给进程分配 ID。如果 pid 17 是当前已分配的最大数值，则下次将分配 pid 18，即使当创建新进程时最后那个分配 pid 为 17 的进程已经不再运行。内核

不会重新使用进程 ID，除非它最终到达了最大值——也就是，先前的值不会被重用，除非/proc/sys/kernel/pid_max 设置的值已经被分配。因此，虽然 Linux 不保证进程 ID 在长时期的唯一性，它的分配行为确实提供了 pid 值短期的稳定与唯一性。

进程层次

产生新进程的那个进程称为父进程；新创建的进程则被称为子进程。每个进程都由另一个进程创建（当然除了 init 进程）。因此，每个子进程都有一个父进程。这个关系在每个进程的父进程 ID(ppid)中记录，它是父进程的 pid。

每个进程都属于一个用户和组。这个所有权用来控制资源的访问权限。对于内核来说，用户和组就是整数值而已。通过文件/etc/passwd 和/etc/group，这些整数值被映射到易读的 Unix 用户名，例如用户 root 或者组 wheel（通常来说，Linux 内核对易读的字符串没有兴趣，而更喜欢使用整数来标识）。每个子进程都继承父进程的用户和组属性。

每个进程同时还属于一个进程组，进程组只是简单地表示进程之间的关系，千万不要和前面说的用户/组概念混淆。子进程通常和父进程属于同一个进程组。此外，当 shell 启动一个管道（例如用户输入 ls | less 时），管道中的所有命令都会属于同一个进程组。进程组的概念使得发送信号或者接收整个管道的信息非常容易，进程的所有子进程也属于同一个管道。从用户的角度来看，进程组和工作(job)比较接近。

pid_t

进程 ID 在程序中由 pid_t 类型表示，在头文件<sys/types.h>中定义。pid_t 后面的 C 类型和具体的体系架构有关，没有任何 C 标准强制定义。不过在 Linux 中，pid_t 通常是 C 语言 int 类型的 typedef 定义。

获得进程 ID 和父进程 ID

getpid()系统调用返回调用进程的 pid：

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid (void);
```

getppid()系统调用返回调用进程的父进程 pid：

```
#include <sys/types.h>
#include <unistd.h>
pid_t getppid (void);
```

这两个调用都不会返回错误，因此使用非常简单：

```
printf ("My pid=%d\n", getpid ( ));
printf ("Parent's pid=%d\n", getppid ( ));
```

我们怎么知道 pid_t 是有符号整数呢？好问题！答案很简单，我们也不知道。即使我们

可以安全地假设 pid_t 在 Linux 中是 int，这样一个假设也破坏了抽象类型的意图，进而损害了可移植性。不幸地是，没有简单的方法打印 pid_t 值，C 语言中的所有 typedef 类型都是这样——这是抽象的一部分，从技术上讲我们需要一个 pid_to_int()函数，而这正是我们缺乏的。然而把 pid_t 值当成整数，至少在 printf()中是很常见的。

运行新进程

在 Unix 中，创建一个新进程的动作被分离成：装载程序到内存、和执行程序镜像两个动作。一个系统调用（实际上，一个家族调用中的一个）装载二进制程序到内存，替换地址空间中原有的内容，然后开始执行新程序。这就是执行新程序的过程，这个功能由 exec 家族调用提供。

另一个系统调用用来创建新进程，它最初只是复制父进程。通常，新创建的进程会立即执行一个新程序。创建新进程的动作被称为 fork，这个功能由 fork()系统调用提供。首先是 fork 创建一个新进程，然后 exec 装载新镜像到进程，在一个新进程中执行一个新程序时需要这两个动作一起。我们先讨论 exec 系列调用，然后是 fork()调用。

exec 家族调用

Unix 中没有单个的 exec 函数，而是有一组 exec 函数基于同一个系统调用。我们首先来看最简单的 execl()调用：

```
#include <unistd.h>
int execl (const char *path,
        const char *arg,
        ...);
```

调用 execl()将装载 path 指定的程序到内存中，并替换当前进程镜像。arg 是程序运行的第一个参数。省略号表示参数个数可变——execl()函数是可变参数的，也就是说更多的参数是可选的，紧跟着 arg 参数之后。参数列表必须以 NULL 结束。

例如，下面代码使用程序/bin/vi 替换当前执行进程：

```
int ret;
ret = execl ("/bin/vi", "vi", NULL);
if (ret == -1)
    perror ("execl");
```

注意我们遵循了 Unix 的传统，传递“vi”作为程序的第一个参数。shell 把最后一部分添加到路径上，当 forks/execs 进程时，程序就可以获得它的第一个参数：argv[0]，也就是二进制镜像的名字。许多情况下，有一些系统工具在用户看来具有不同的名字，实际上只是同一个程序的不同名字的硬链接。程序使用第一个参数来决定自己的行为。

如果你想要编辑文件/home/kidd/hooks.txt，你可以执行下面代码：

```
int ret;
ret = execl ("/bin/vi", "vi", "/home/kidd/hooks.txt", NULL);
if (ret == -1)
    perror ("execl");
```

通常情况下，execl()不会返回。成功调用 execl()将跳到新程序的入口，原先执行的代码在进程地址空间中已不复存在；错误时 execl()返回-1，并设置 errno 指示错误原因。我们在本节后面会讨论可能的 errno 值。

成功调用 execl()不仅仅改变地址空间和进程镜像，还有其它一些进程属性：

- 任何未决的信号都将丢失。
- 任何进程捕获的信号都恢复到默认行为，因为信号处理器在进程地址空间中已经不存在了。
- 任何内存锁都将被丢弃。
- 大部分线程属性恢复为默认值。
- 大部分进程统计数据重置。
- 与进程内存相关的任何东西，包括任何映射文件，都将被丢弃。
- 用户空间存在的任何东西，包括 C 库（如 atexit()行为），都将被丢弃。

但是进程也有许多属性不会改变。例如 pid、父进程 pid、优先级、以及拥有者用户和组，都保持不变。

通常执行 exec 时打开文件被继承，这意味着新执行的程序能够完全访问原始进程打开的所有文件，只要它知道确切的文件描述符。但是，这通常不是期望的行为。一般我们会在调用 exec 之前关闭文件，虽然也可以通过 fcntl()指示内核自动关闭文件。

exec 家族其它成员

除了 execl()，exec 家族还有五个其它成员：

```
#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[]);
```

要记住这些调用非常简单，l 和 v 描述调用的参数以列表(list)还是数组(vector)提供。p 表示搜索指定文件时使用用户的完整 path 路径。使用带 p 的调用可以仅仅指定文件名，只

要文件存在于用户 path 路径中。最后，e 表示给新进程提供一个新的环境(environment)。奇怪的是，exec 家族并没有同时搜索 path 并创建新环境的成员，虽然从技术上来说并没有什么理由漏掉这样一个调用。可能由于带 p 的调用本身是为 Shell 而实现，而 Shell 执行的进程通常从 Shell 继承了它们的环境。

exec 家族中参数是数组的那几个调用，和其它几个调用的工作完全一样，除了传递的参数是数组，而不是可变参数之外。使用数组允许参数在运行时确定。和可变参数一样，数组也必须以 NULL 结束。

下面代码片断使用 execvp()执行 vi，和前面代码做的一样：

```
const char *args[] = { "vi", "/home/kidd/hooks.txt", NULL };
int ret;
ret = execvp ("vi", args);
if (ret == -1)
        perror ("execvp");
```

这里我们假设/bin 在用户的 path 路径中，这个例子和前面例子类似。

在 Linux 中，只有一个 exec 家族成员是系统调用。其余都只是 C 库对系统调用的封装。因为可变参数系统调用比较难实现，同时用户路径的概念只存在于用户空间中，内核实现系统调用的唯一选择是 execve()。这个系统调用的原型和用户调用相同。

错误代码

成功时 exec 系统调用没有返回；失败时调用返回-1，并设置 errno 为以下值之一：

E2BIG

提供的参数列表(arg)或者环境(envp)的总字节太大。

EACCESS

进程没有搜索 path 的权限；path 不是普通文件；目标文件不是可执行的；或者 path 和 file 所在的文件系统挂载为不可执行(noexec)。

EFAULT

给定的指针无效。

EIO

发生底层 I/O 错误(这非常糟糕)。

EISDIR

path 中指定的文件是目录。

ELOOP

系统解析 path 时遇到过多的符号链接。

EMFILE

调用进程达到打开文件的最大限制。

ENFILE

达到系统最多可打开文件数目限制。

ENOENT

path 或者 file 目标不存在，或者需要的共享库不存在。

ENOEXEC

path 或者 file 目标是无效的二进制文件，或者是另一种机器体系架构的文件。

ENOMEM

内核没有足够的可用内存来执行新程序。

ENOTDIR

path 的中间某个部分不是目录。

EPERM

path 或者 file 所在的文件系统以 nosuid 方式挂载，用户不是 root，并且 path 或者 file 已设置 suid 和 sgid。

ETXTBSY

path 或者 file 的目标已经被另一个进程以写入方式打开。

fork()系统调用

fork()系统调用使用当前运行进程的镜像来创建一个新进程：

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork (void);
```

成功调用 fork()将创建一个新进程，新进程几乎在所有方面都和调用进程相同。两个进程都继续运行，从调用 fork()的地方返回，就好像什么事都没有发生一样。

新进程被称为原始进程的子进程，后者就是父进程。在子进程中，成功调用 fork()返回 0。而在父进程中，fork()返回子进程的 pid。子进程和父进程的处理方式几乎相同，除了以下必要的区别：

- 子进程的 pid 是新分配的，因此肯定与父进程 pid 不相同。
- 子进程的 ppid 被设置为父进程的 pid。
- 子进程的资源统计信息重置为 0。
- 清除子进程的所有未决信号，子进程不继承父进程的信号。
- 子进程不继承父进程已获得的任何文件锁。

错误时子进程将不会被创建，fork()返回-1，而且 errno 被设置为合适的值。可能的 errno 代码只有两个，有三种不同的意思：

EAGAIN

内核分配某些资源失败，例如新的 pid；或者 RLIMIT_NPROC 资源限制已经达到。

ENOMEM

内核没有足够的可用内存来完成请求。

使用非常简单：

```
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()最常见的用途是创建一个新进程，然后装载一个新的二进制镜像——想像 Shell 为用户运行一个新程序、或者进程创建一个帮助程序。首先进程 fork 一个新进程，然后子进程执行一个新的二进制镜像。这种“fork 加 exec”的组合非常频繁也很简单。下面例子创建一个运行二进制文件/bin/windlass 的新进程：

```
pid_t pid;
pid = fork ( );
if (pid == -1)
    perror ("fork");

/* the child ... */
if (!pid) {
    const char *args[] = { "windlass", NULL };
    int ret;
    ret = execv ("/bin/windlass", args);
    if (ret == -1) {
        perror ("execv");
        exit (EXIT_FAILURE);
    }
}
```

父进程没什么变化，在创建子进程后继续运行。子进程则调用 execv()来运行/bin/windlass 程序。

Copy-on-Write

在早期的 Unix 系统中，fork 操作很简单，甚至可以说很幼稚。调用 fork()时，内核复制所有内部数据结构，复制进程的页表项、然后对父进程的地址空间执行复制，一页一页的复制到新的子进程地址空间中。但这种一页一页的复制相当耗时，至少从内核的角度来说。

现代的 Unix 系统则更加优化。像 Linux 这样的现代 Unix 系统采用 Copy-on-Write(COW)页，而不是复制父进程的整个地址空间。

Copy-on-Write 是一种懒惰优化策略，设计用来减轻复制资源时的开销。设想很简单：如果多个消费者请求读取他们自己的资源拷贝，则不需要对资源进行重复拷贝。实际上，每个消费者可以拥有一个指向相同资源的指针。只要没有消费者修改自己那份资源“拷贝”，则等同于对资源进行互斥访问，同时避免了复制资源的开销。如果某个消费者确实要修改自己的资源拷贝，在那个时候，将会透明地复制一份资源拷贝给需要修改的消费者。然后这个消费者就可以修改自己的资源拷贝，而其它消费者继续共享原始未变化的资源。Copy-on-Write 因此得名：复制只发生在写入时。

最主要的优点是如果一个消费者从不修改自己的资源拷贝，那就不会进行实际的拷贝操作。因此，只要进程不修改自己的地址空间，就不需要拷贝整个地址空间。在 fork 结束后，父进程与子进程看上去拥有自己单独的地址空间，而实际上它们共享父进程的原始页——而这又可能与它的父进程或其它子进程共享。

内核的实现很简单。页面在内核的页面相关数据结构中被标识为只读和写入时复制。如果某个进程尝试修改某个页，将产生一个页面错误。然后内核处理页面错误，透明地复制这个页。在这个时候，当前进程的页面的写入时复制属性被清除，从此不再共享。

由于现代机器架构在内存管理单元(MMU)对写入时复制提供了硬件层的支持，写入时复制实现起来非常简单。

写入时复制在 fork 操作时有一个更大的优点。因为很大一部分 fork 都紧跟着一个 exec，复制父进程的地址空间到子进程地址空间经常是完全浪费时间：如果子进程立即执行一个新的二进制镜像，它原来的地址空间将全部擦去。写入时复制优化了这种情况。

vfork()

在写入时复制页面出现之前，Unix 设计者已经关注 fork 紧跟着 exec 时的地址空间复制浪费问题。因此 BSD 开发者在 3.0BSD 公开了 vfork()系统调用：

```
#include <sys/types.h>
#include <unistd.h>
pid_t vfork (void);
```

成功调用 vfork()和 fork()的行为一样，除了子进程必须立即发起一个 exec 函数，或者通过调用_exit()退出（下一节讨论）。vfork()系统调用避免了地址空间和页表复制，通过挂起父进程直到子进程终止或者执行新的二进制镜像。在此之间，父进程和子进程共享地址空间和页表项（虽然没有写入时复制的语义）。实际上，在调用 vfork()期间唯一做的事情是复制内核内部数据结构。因此，子进程不能修改地址空间内的任何内存。

vfork()系统调用是一个历史遗物，永远不应该被 Linux 实现。应该指出的是，尽管 fork()使用了写入时复制，vfork()也比 fork()更快，因为 vfork()不需要复制页表项。无论如何，写入时复制的采用，削弱了 fork()其它变体的需要。实际上直到 Linux 内核 2.2.0 版本，vfork()只是 fork()的简单封装。由于对 vfork()的需求远远小于对 fork()的需求，这样一种 vfork()实现是切实可行的。

严格地说，没有哪种 vfork()实现是无 bug 的：考虑 exec 调用失败的情况！父进程将被挂起不确定时间，直到子进程决定自己该怎么做或者直到子进程退出。

终止进程

POSIX 和 C89 都定义了标准函数来终止当前进程：

```
#include <stdlib.h>
void exit (int status);
```

调用 exit()首先执行一些基本的停止步骤，然后指示内核终止进程。这个函数没有办法返回错误——实际上它永远不返回。因此，exit()调用之后的指令是没有任何意义的。

status 参数用来指示进程的退出状态。其它程序——例如用户的 Shell——可以检查这个值。特别地，状态 & 0377 将返回给父进程。本章后面我们会讨论如何获得这个返回值。EXIT_SUCCESS 和 EXIT_FAILURE 以可移植的方式定义了成功和失败。在 Linux 中，0 一般代表成功；非 0 值，例如 1 或-1 对应于失败。

因此，成功退出可以使用下面这行代码：

```
exit (EXIT_SUCCESS);
```

在终止进程前，C 库执行下列停止步骤，按顺序进行：

1. 调用所有通过 atexit()或者 on_exit()注册的函数，按注册相反的顺序依次调用（我们

马上会在本章后面讨论这些函数）。

2. flush 所有打开的标准 I/O 流。
3. 移除所有通过 tmpfile()函数创建的临时文件。

这些步骤完成了进程退出时在用户空间需要做的所有事情，于是 exit()调用系统调用_exit()来指示内核处理进程终止剩下的工作：

```
#include <unistd.h>
void _exit (int status);
```

当进程退出时，内核清理进程不再使用的那些资源。包括但不限于：已分配内存、打开文件、系统 V 信号量。清理之后，内核销毁进程并通知父进程子进程已死。

应用可以直接调用_exit()，但是这样做很少有意义：大部分应用需要完全 exit 提供的某些清理工作，例如 flush 到 stdout 流。不过注意，使用 vfork()时应该调用_exit()而不是 exit()。

其它终止方式

结束程序的典型方式不是通过显式的系统调用，而是简单的执行程序到结束。在 C 语言中，当 main()函数返回时程序结束。但是这种执行到结束的方式，仍然调用了一个系统调用：编译器简单地在停止代码之后插入一个隐式的_exit()调用。好的编码实践总是显式地返回一个退出状态，要么通过 exit()，要么从 main()返回一个值。Shell 使用退出状态码确定指令执行成功还是失败。注意成功返回是 exit(0)，或者从 main()函数返回 0。

进程接收到信号，并且信号的默认动作是终止时，进程也会终止运行。类似的信号包括 SIGTERM 和 SIGKILL（第九章）。

结束程序执行的最后一种方式是引发内核的愤怒。进程执行非法指令、导致段越界、用完内存等等情况下，内核都可以杀死进程。

atexit()

POSIX 1003.1-2001 定义并且 Linux 实现了 atexit()库调用，用来注册函数，当进程终止时自动调用：

```
#include <stdlib.h>
int atexit (void (*function)(void));
```

成功调用 atexit()注册指定的函数，在进程正常终止时调用：例如进程通过 exit()或者从 main()函数返回。如果进程调用了 exec 函数，则已注册的函数列表将被清空（因为这些函数在新的进程地址空间中不再存在）。如果进程通过信号终止，则已注册的函数不会被调用。

指定的函数没有参数，也不返回值。函数原型如下：

```
void my_function (void);
```

函数按注册相反的顺序调用。也就是说，函数存放在堆栈中，最后一个进来的将最先出去(LIFO)。注册的函数不允许调用 exit()，以免引起无穷递归。如果函数确实需要提前终止进程，它应该调用_exit()。但是类似的行为不被推荐，因为这时候有可能某些重要的函数将得不到运行。

POSIX 标准要求 atexit()支持至少 ATEXIT_MAX 个注册函数，并且这个值最少要达到 32。

实际的 ATEXIT_MAX 值可以通过 sysconf()和_SC_ATEXIT_MAX 获得：

```
long atexit_max;
atexit_max = sysconf (_SC_ATEXIT_MAX);
printf ("atexit_max=%ld\n", atexit_max);
```

成功时 atexit()返回 0；失败时返回-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 定义了自己的函数 on_exit()，等同于 atexit()，Linux glibc 也支持它：

```
#include <stdlib.h>
int on_exit (void (*function)(int , void *), void *arg);
```

这个函数和 atexit()一样工作，但是注册函数的原型不同：

```
void my_function (int status, void *arg);
```

status 参数是传递给 exit()或者 main()返回的值。arg 参数是传递给 on_exit()的第二个参数。必须非常小心地确保在函数最终被调用时，arg 指向的内存是合法的。

最新版本的 Solaris 不再支持这个函数。你应该使用标准的 atexit()函数代替它。

SIGCHLD

当一个进程终止时，内核会发送信号 SIGCHLD 给父进程。默认时该信号被忽略，父进程不做任何动作。但是进程可以通过 signal()或者 sigaction()系统调用选择处理这个信号。关于信号的精彩世界，我们将在第九章讨论。

SIGCHLD 信号可能在任何时间产生并分派出去，因为子进程的终止对于父进程来说是异步的。但是通常父进程希望知道更多关于子进程终止的情况，或者显式地等待子进程终止事件的发生。这可以通过接下来要讨论的系统调用来完成。

等待子进程终止

通过信号接收通知是一种很好的方式，但是许多父进程希望当子进程终止时，能够获得更多的信息——例如，子进程的返回值。

如果子进程在终止时完全消失，可以想像，这时就没有剩下任何东西让父进程查询子进程的信息。因此，早期的 Unix 设计者决定当子进程在父进程之前死亡时，内核应该把子进程置为一个特殊的进程状态。在这种状态下的进程被称为僵尸进程(zombie)。只有极少量的进程信息被保留——内核基本数据结构中可能有用的那部分数据。在这种状态下的进程等待父进程询问自己的状态（这个过程被称为等待僵尸进程）。只有父进程获取了已终止子进程保留的信息之后，僵尸进程才会最终消失。

Linux 内核提供几个接口获取已终止子进程的信息。其中最简单的一个接口由 POSIX 定义，就是 wait()：

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait (int *status);
```

调用 wait()返回一个已终止的子进程 pid，或者错误时返回-1。如果没有子进程已终止，则调用阻塞直到有子进程终止。如果子进程已经终止，则调用将立即返回。因此，调用 wait()来响应子进程的死亡——例如接收到 SIGCHLD 信号时——总是会立即返回而不阻塞。

错误时，有两个可能的 errno 错误代码：

ECHILD

调用进程没有任何子进程。

EINTR

等待期间接收到信号，调用提前返回。

当 status 不为 NULL 时，status 指针包含子进程的额外信息。因为 POSIX 允许实现定义 status 各个位代表的含义，标准提供了一组宏定义来解释 status 参数：

```
#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);
```

前两个宏定义根据进程终止的方式，可能返回 true(非 0 值)。第一个 WIFEXITED，如果进程正常终止则返回 true——也就是，如果进程调用了_exit()。在这种情况下，宏 WEXITSTATUS 的低八位就是传递给_exit()的值。

如果信号导致进程终止，WIFSIGNALED 返回 true。在这种情况下，WTERMSIG 返回引起

终止的信号值，并且如果进程响应信号进行了 dump core，则 WCOREDUMP 返回 true。WCOREDUMP 不是 POSIX 标准定义，尽管许多 Unix 系统（包括 Linux）支持它。

如果进程停止或者继续，并且当前正通过 ptrace()系统调用跟踪，WIFSTOPPED 和 WIFCONTINUED 分别返回 true。这些条件通常只适用于实现调试器，尽管当使用 waitpid()时，它们也被用来实现工作控制。正常情况下，wait()只用在进程终止信息的通知上。如果 WIFSTOPPED 为 true，WSTOPSIG 提供停止进程的信号值。WIFCONTINUED 不是 POSIX 标准定义，不过将来的标准将为 waitpid()定义它。在 Linux 2.6.10 内核中，Linux 也为 wait()提供了这个宏定义。

让我们来看一个使用 wait()的例子，看看子进程究竟会发生什么：

```
#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;
}
```

这个程序首先 fork 一个子进程，子进程立即退出。父进程然后执行 wait()系统调用来确定子进程的状态。进程打印子进程 pid、以及它如何死亡。因为这里子进程通过从 main()返回而终止，我们可以明确知道程序会有类似下面输出：

```
$ ./wait
pid=8529
Normal termination with exit status=1
```

如果，子进程并不是通过返回而终止，我们让子进程调用 abort()，自己给自己发送 SIGABRT 信号，我们就会得到类似下面这样的输出：

```
$ ./wait
pid=8678
Killed by signal=6
```

等待指定的进程

观察子进程的行为非常重要。但是通常进程有多个子进程，而且并不希望等待所有，而是等待某个特定的子进程。一个解决方案是发起多次 wait()调用，每次都通过返回值判断子进程。这样非常麻烦，如果稍后你又想检查另一个终止进程的状态时怎么办？父进程需要保存所有 wait()的输出，稍后才能再次使用。

如果你知道你想等待的进程 pid，就可以调用 waitpid()系统调用：

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid (pid_t pid, int *status, int options);
```

waitpid()系统调用是 wait()的加强版。增加的参数用于更好地控制调用。

pid 参数指定等待的进程（或者进程组），它的值有以下几种情况：

< -1

等待所有进程组 ID 等于这个值的绝对值的进程。例如，传递-500 将等待进程组 500 中的所有进程。

-1

等待所有子进程，这时和 wait()的行为一样。

0

等待所有与调用进程属于同一个组的子进程。

> 0

等待指定 pid 的子进程。例如传递 500 将等待子进程 500。

status 参数与 wait()完全一样，可以被之前讨论过的宏定义操作。

options 参数是以下选项的“或”：

WNOHANG

如果没有匹配的子进程已终止（或者停止、继续），调用不阻塞而立即返回。

WUNTRACED

如果设置，则即使调用进程没有跟踪子进程，WIFSTOPPED 也被设置。这个标志允许实现更通用的工作控制，例如 Shell。

WCONTINUED

如果设置，则即使调用进程没有跟踪子进程，WIFCONTINUED 也被设置。和 WUNTRACED 一样，这个标志也对实现 Shell 有用。

成功时 waitpid()返回子进程的 pid。如果指定了 WNOHANG，并且指定的子进程并没有状态改变，waitpid()返回 0；错误时调用返回-1，并且设置 errno 为以下三个值之一：

ECHILD

通过 pid 指定的进程不存在，或者不是调用进程的子进程。

EINTR

WNOHANG 选项没有被指定，并且等待过程中接收到信号。

EINVAL

options 参数非法。

举个例子，假设你的程序希望获得子进程 pid 1742 的返回值，并且如果子进程没有终止，则立即返回。你可能会像下面这样编写代码：

```
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)" : "");
}
```

作为最后一个例子，注意 wait()的下面用法：

```
wait (&status);
```

等同于 waitpid()的下面用法：

```
waitpid (-1, &status, 0);
```

更多等待功能

对于需要更多等待子进程功能的应用，POSIX 标准的扩展 XSI 定义(Linux 实现)了 waitid()：

```
#include <sys/wait.h>
int waitid (idtype_t idtype,
        id_t id,
```

```
siginfo_t *infop,
int options);
```

和 wait()、waitpid()一样，waitid()用来等待和获取子进程的状态改变信息（终止、停止、继续）。它提供了更多选项和功能，但也带来了更大的复杂性。

和 waitpid()一样，waitid()允许开发者指定等待的子进程。但是 waitid()通过两个参数来实现这个工作。idtype 和 id 参数指定具体等待的子进程，完成 waitpid()的 pid 参数相同的目标。idtype 可以是以下值：

P_PID

等待 pid 匹配 id 的子进程。

P_GID

等待组 ID 匹配 id 的进程。

P_ALL

等待所有子进程，忽略 id 参数。

id 参数是罕见的 id_t 类型，代表通用的标识数值。采用它是考虑将来实现增加新的 idtype 值，提供更保险的预定义类型，可以保存新创建的标识。id_t 确保足够大能够存放任何 pid_t。在 Linux 中，开发者可以像使用 pid_t 那样使用它——例如，直接传递 pid_t 或者数值常量。书生气的程序员，当然也可以自由使用类型转换。

options 参数是以下值的二进制“或”：

WEXITED

调用等待指定的子进程终止。

WSTOPPED

调用等待指定的子进程响应信号并停止。

WCONTINUED

调用等待指定的子进程响应信号并继续运行。

WNOHANG

调用不会阻塞而立即返回，即使没有匹配的子进程已终止（停止、或继续）

WNOWAIT

调用不会移除匹配进程的僵尸状态。进程可以在未来再次等待。

成功等待子进程时 waitid()填充 infop 参数，它必须指向一个合法的 siginfo_t 类型对象。确切的 siginfo_t 结构体与实现相关，但是调用 waitid()之后某些域通常都是合法的。也就是，成功调用 waitid()可以保证下面域被填充：

si_pid

子进程 pid

si_uid

子进程 uid

si_code

根据子进程终止、通过信号死亡、通过信号停止、通过信号继续，分别被设置为 CLD_EXITED、CLD_KILLED、CLD_STOPPED、CLD_CONTINUED。

si_signo

设置为 SIGCHLD

si_status

如果 si_code 是 CLD_EXITED，这个域是子进程的退出代码；否则这个域是引起子进程状态改变的信号值。

成功时 waitid()返回 0；失败时返回-1，并设置 errno 为以下值之一：

ECHLD

id 和 idtype 表示的进程不存在。

EINTR

WNOHANG 没有被指定，信号中断执行。

EINVAL

options 参数或者 id 和 idtype 的组合非法。

waitid()函数提供了额外的有用的语义，这些在 wait()和 waitpid()中没有。特别地，从 siginfo_t 结构体中可获得的信息非常有用。如果不需要这些信息，则可以考虑前面两个更简单的函数，它们的系统支持范围更广，因此在非 Linux 系统也更加可移植。

BSD 等待函数：wait3()和 wait4()

waitpid()来源于 AT&T System V Release 4，而 BSD 则选择了自己的路线，提供另外两个函数用来等待子进程状态改变：

```
#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);
```

3 和 4 分别表示函数有三个和四个参数。

wait3()和 wait4()函数和 waitpid()工作类似，除了 rusage 参数以外。下面 wait3()调用：

```
pid = wait3 (status, options, NULL);
```

等同于下面 waitpid()调用：

```
pid = waitpid (-1, status, options);
```

而下面 wait4()调用：

```
pid = wait4 (pid, status, options, NULL);
```

等同于下面这个 waitpid()调用：

```
pid = waitpid (pid, status, options);
```

也就是：wait3()等待所有子进程改变状态；而 wait4()等待 pid 参数标识的特定子进程改变状态。options 参数与 waitpid()的行为一样。

前面说过，这些调用与 waitpid()的最大区别在于 rusage 参数。如果它非 NULL，函数将把子进程的信息填充指针指向的 rusage 结构体。这个结构体提供了子进程资源使用的信息：

```
#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 */
};
```

下一章我们会进一步讨论资源的使用。

成功时这两个函数返回改变状态的子进程 pid；失败时返回-1，并和 waitpid()一样设置 errno 值。

因为 wait3()和 wait4()并不是 POSIX 定义，只有在特别需要资源使用信息时，才应该使用它们。不过虽然缺乏 POSIX 标准定义，几乎每个 Unix 系统都支持这两个调用。

运行并等待新进程

ANSI C 和 POSIX 都定义了一个接口，结合创建新进程和等待进程终止——把它想像成同步创建进程。如果一个进程创建子进程之后，唯一需要做的就是等待它的终止，则可以调用这个接口：

```
#define _XOPEN_SOURCE /* if we want WEXITSTATUS, etc. */
#include <stdlib.h>
int system (const char *command);
```

system()函数这样命名是因为同步进程调用被称为 shelling out to the system。使用 system()来运行一个简单的程序或者 shell 脚本，然后获取它的返回值，这种用法非常普遍。

调用 system()将执行 command 参数指定的命令，其中包含所有额外的命令参数。command 参数被添加到/bin/sh -c 的后面。按这种方式，参数就被完全传递给 shell。

成功时函数的返回值是命令的返回状态，和 wait()的返回一样。因此，执行命令的返回代码通过 WEXITSTATUS 获得。如果调用/bin/sh 本身失败，WEXITSTATUS 的返回值和通过调用 exit(127)返回的一样。由于执行命令本身也可能返回 127，没有办法准确地检查到底是 shell

还是命令返回错误；错误时 system()返回-1。

如果 command 为 NULL，而/bin/sh 可用，system()将返回非 0 值；否则返回 0。

在命令执行期间，SIGCHLD 信号将被阻塞，SIGINT 和 SIGQUIT 信号被忽略。忽略 SIGINT 和 SIGQUIT 有几层含义，特别是当 system()在循环体中被调用时。如果在循环中调用 system()，你应该确保程序正确地检查了子进程的退出代码。例如：

```
do {
    int ret;
    ret = system ("pidof rudderd");
    if (WIFSIGNALED (ret) &&
            (WTERMSIG (ret) == SIGINT ||
            WTERMSIG (ret) == SIGQUIT))
        break; /* or otherwise handle */
} while (1);
```

使用 fork()、exec 家族中的某个函数、以及 waitpid()实现 system()是一个很有用的练习。你应该自己尝试实现一个，因为它把本章的许多概念结合在一起。不过为了本书的完整性，下面是一个简单的示例实现：

```
/*
* my_system - synchronously spawns and waits for the command
* "/bin/sh -c <cmd>".
*
* Returns -1 on error of any sort, or the exit code from the
* launched process. Does not block or ignore any signals.
*/
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;
    }
```

注意上面例子没有阻塞或者禁止任何信号，这和标准的 system()实现不一样。根据你程序的情况，这样实现有好有坏。但至少保持 SIGINT 为非阻塞通常是聪明的，因为它允许你调用命令时按用户通常希望的方式来中断它。一个更好的实现可以添加额外的指针参数，当非 NULL 时，表示不同的错误。例如，可以添加 fork_failed 和 shell_failed。

僵尸进程(Zombie)

前面讨论过，一个进程已经终止，但尚未被父进程等待就称为“僵尸进程”。僵尸进程继续消耗系统资源，尽管只是很小的比例——维护子进程基本骨架就可以。保留这些资源以便父进程可以获得子进程终止等相关的信息。一旦父进程这样做了，内核就清除进程，僵尸进程也就不再存在了。

但是，任何使用 Unix 一段时间的人都看到过僵尸进程。这些进程通常叫做幽灵进程，有不负责任的父进程。如果你的应用 fork 一个子进程，那就是你应用的责任去等待子进程，即使你并不需要子进程的信息。否则进程的所有子进程都成为幽灵进程而一直存在，拥挤系统的进程列表，并对应用的执行造成影响。

当父进程在子进程之前死亡，或者在它等待僵尸子进程之前死亡，会发生什么呢？一旦某个进程终止，Linux 内核遍历它的子进程列表，并且把 init 进程设置为它们的父进程。这确保了不会存在进程没有直接父进程的现象。init 进程周期性地等待它的所有子进程，确保不会有进程长期地保持僵尸状态——消除幽灵进程！因此，如果父进程在子进程之前死亡，或者没有在退出之前等待子进程，init 将成为它们的父进程并执行等待，允许它们最终完全退出。这样做到目前为止仍然被认为是好的实践，这个保护措施意味着短生命周期的进程不需要过分地担心等待子进程的事情。

用户和组

正如本章前面提到的，第一章也讨论过，进程关联到用户和组。用户和组标识都是数值，分别使用 C 类型 uid_t 和 gid_t 描述。数值和可读名字之间的映射——例如 root 用户有 uid 0——在用户空间文件/etc/passwd 和/etc/group 中执行。内核只处理数值。

在 Linux 系统中，进程的用户和组 ID 决定进程可以执行的操作。因此进程必须以适当的用户和组运行。许多进程以 root 用户运行，但是软件开发的最佳实践鼓励“最小权限”教条，意味着进程应该以可能的最小权限级别运行。这个需求是动态的：如果进程开始需要 root 权限执行操作，但随后并不需要这些权限，则它应该尽快地丢弃 root 权限。因此许多进程——特别是那些需要 root 权限执行某些操作的进程——经常操作自己的用户和组 ID。

在我们查看如何操作用户和组 ID 之前，首先需要讨论用户和组 ID 的复杂性。

实际、有效、已保存用户和组 ID

以下讨论集中于用户 ID，但对于组 ID 来说情况是一样的。

实际上每个进程都关联到四个而不是一个用户 ID：实际、有效、已保存、和文件系统用户 ID。实际用户 ID 是最初运行进程的用户 ID。它被设置为父进程的实际用户 ID，并且在执行 exec 调用时不会改变。通常登录进程设置用户登录 shell 的实际用户 ID，然后用户的所有进程继续使用这个用户 ID。超级用户(root)可以改变实际用户 ID 为任何值，但其它用户均无法改变它。

有效用户 ID 是进程当前使用的用户 ID。权限检查通常就是检查这个值。最初有效用户 ID 与实际用户 ID 相同，因为当进程 fork 时，父进程的有效用户 ID 被子进程继承下来。此外，当进程发起 exec 调用时，有效用户 ID 一般也不改变。但是正是在 exec 过程中体现出实际和有效 ID 的关键区别：通过执行 setuid(suid)二进制镜像，进程可以改变有效用户 ID。更准确地说，有效用户 ID 被设置为程序文件拥有者的用户 ID。例如，因为/usr/bin/passwd 文件是 setuid 文件，而且 root 是它的拥有者，当一个普通用户的 shell 创建一个进程并 exec 这个文件时，不管执行命令的用户是谁，进程都将把 root 作为有效用户 ID。

非特权用户可以把有效用户 ID 设置为实际或者已保存用户 ID，你马上就会看到如何实现。超级用户则可以设置有效用户 ID 为任何值。

已保存用户 ID 是进程最初的有效用户 ID。当进程调用 fork 时，子进程继承父进程的已保存用户 ID。但是在 exec 调用时，内核设置已保存用户 ID 为有效用户 ID，从而在 exec 时记录下有效用户 ID。非特权用户不可以改变已保存用户 ID；超级用户则可以把已保存用户 ID 改变为实际用户 ID。

所有这些 ID 到底有什么意义呢？有效用户 ID 是最要紧的一个：它是检查进程权限使用的用户 ID。实际用户 ID 和已保存用户 ID 作为它的代理，或者非特权用户可以切换到的潜在用户 ID。实际用户 ID 是运行程序的实际用户，已保存用户 ID 是 exec 期间 suid 二进制改变之前的有效用户 ID。

改变实际或者已保存用户和组 ID

用户和组 ID 可以通过两个系统调用设置：

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

int setuid (uid_t uid);
int setgid (gid_t gid);
```

调用 setuid()设置当前进程的有效用户 ID。如果进程当前的有效用户 ID 是 0(root)，则实际和已保存用户 ID 也同时设置。root 用户可以给 uid 提供任何值，从而设置所有三个用户 ID 的值为 uid。非 root 用户只允许提供实际或者已保存用户 ID 给 uid。换句话说，非 root 用户只能设置有效用户 ID 为实际或者已保存用户 ID。

成功时 setuid()返回 0；失败时返回-1，并设置 errno 为以下值之一：

EAGAIN

uid 与实际用户 ID 不相同，设置实际用户 ID 为 uid 会导致用户超过 NPROC 限制（指定用户可以拥有的进程最大数量）。

EPERM

用户非 root，并且 uid 不是有效和已保存用户 ID。

以上的讨论也适用于组——简单地替换 setuid()为 setgid()，以及 uid 换成 gid 即可。

改变有效用户和组 ID

Linux 提供两个 POSIX 要求的函数，用来设置当前执行进程的有效用户和组 ID：

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

int seteuid (uid_t euid);
int setegid (gid_t egid);
```

调用 seteuid()调用有效用户 ID 为 euid。root 可以提供任何值给 euid。非 root 用户只能设置有效用户 ID 为实际或者已保存用户 ID。成功时 seteuid()返回 0；失败时返回-1，并设置 errno 为 EPERM，表示当前进程的拥有者不是 root，而且 euid 不等于实际和有效用户 ID。

注意在非 root 的情况下，seteuid()和 setuid()的行为一样。因此标准实践也是一个好主意是总是使用 seteuid()，除非你的进程以 root 运行，这时候 setuid()更有意义。

前面的讨论同样适用于组——简单地替换 seteuid()为 setegid()，euid 换成 egid 即可。

改变用户和组 ID：BSD 风格

BSD 坚持使用自己的接口来设置用户和组 ID。Linux 为了兼容性也提供了这些接口：

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

int setreuid (uid_t ruid, uid_t euid);
int setregid (gid_t rgid, gid_t egid);
```

调用 setreuid()分别设置进程的实际和有效用户 ID 为 ruid 和 euid。参数的值指定为-1 则保持相应的用户 ID 不变。非 root 进程只允许设置有效用户 ID 为实际或者已保存用户 ID，以及设置实际用户 ID 为有效用户 ID。如果实际用户 ID 改变了，或者有效用户 ID 变为与之前的实际用户 ID 不同的值，则已保存用户 ID 将变成新的有效用户 ID。至少这是 Linux 和大多数 Unix 系统对这种改变的反应。这些行为 POSIX 没有定义。

成功时 setreuid()返回 0；失败时返回-1，并设置 errno 为 EPERM，表示当前进程拥有者不是 root，而且 euid 不等于实际或者已保存用户 ID，或者 ruid 不等于有效用户 ID。

上面的讨论同样适用于组——简单地替换 setreuid()为 setregid()，ruid 换成 rgid，euid 换成 egid 即可。

改变用户和组 ID：HP-UX 风格

你可能觉得情况已经很混乱了，但 HP-UX（Hewlett-Packard Unix）系统还是引入了自己的机制来设置进程的用户和组 ID。Linux 同样也提供了这些接口：

```
#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);
```

调用 setresuid()分别设置实际、有效、已保存用户 ID 为 ruid、euid、suid。某个参数指定-1 则保持相应的用户 ID 不变。

root 用户可以设置所有用户 ID 为任何值。非 root 用户可能设置用户 ID 为当前的实际、有效、或者已保存用户 ID。成功时 setresuid()返回 0；失败时返回-1，并设置 errno 为以下值之一：

EAGAIN

uid 与实际用户 ID 不匹配，并且设置实际用户 ID 会导致用户超过 NPROC 限制（指定用户可以拥有的最大进程数目）。

EPERM

用户非 root，却尝试设置实际、有效、或者已保存用户 ID 为不匹配当前进程的实际、有效、或者已保存用户 ID。

上面的讨论同样也适用于组——简单地替换 setresuid()为 setresgid()，ruid 换成 rgid，euid 换成 egid，suid 换成 sgid 即可。

首选的用户/组操作

非 root 进程应该使用 seteuid()来改变有效用户 ID。root 进程如果想要改变所有三个用户 ID，可以使用 setuid()；如果只想暂时改变有效用户 ID，则可以使用 seteuid()。这些函数都很简单，行为也与 POSIX 一致，只需要适当地考虑已保存用户 ID。

无论是否提供额外的功能，BSD 和 HP-UX 风格的函数都不允许执行 setuid()和 seteuid()不允许的功能。

对已保存用户 ID 的支持

IEEE Std 1003.1-2001(POSIX 2001)对已保存用户和组 ID 提出了要求，而 Linux 早在内核 1.1.38 就已经支持了这些 ID。Linux 特定的程序可以重置已保存用户 ID。老 Unix 系统的程序应该在引用已保存用户和组 ID 之前，先检查_POSIX_SAVED_IDS 宏定义。

即使不存在已保存用户和组 ID，上面的讨论仍然是有效的；只需要忽略关于已保存用户和组 ID 的那部分内容。

获取用户和组 ID

下面两个系统调用分别返回实际用户和组 ID：

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

uid_t getuid (void);
gid_t getgid (void);
```

它们都永远不会失败。同样下面两个系统调用分别返回有效用户和组 ID：

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

uid_t geteuid (void);
gid_t getegid (void);
```

这两个系统调用同样也永远不会失败。

会话和进程组

每个进程都是某个进程组的成员，进程组通常是一个工作控制中相关联的一些进程。进程组的主要属性是可以把信号发送给组中的所有进程：一个单独的动作就可以终止、停止、或者继续相同进程组中的所有进程。

每个进程组都由一个进程组 ID 标识(pgid)，并且有一个进程组领导者(leader)。进程组 ID 就等于进程组领导者的 pid。只要组中还有进程存在，进程组就会一直存在。即使进程组领导者终止，进程组仍然继续存在。

当一个新用户登录到某台机器时，登录进程会创建一个新的会话，会话包含一个单独的进程：用户的登录 shell。登录 shell 的功能是作为会话的领导者(leader)。会话领导者的 pid 被用作会话 ID。会话通常是一个或者多个进程组的集合。会话安排已登录用户的动作，并关联用户到一个控制终端上，控制终端是一个特定的 tty 设备，处理用户的终端 I/O。因此，会话主要是 shell 相关的。实际上没有其它东西关心会话。

进程组提供发送信号到所有成员的机制，使得工作控制以及其它 shell 功能更加简单；而会话进一步加强了终端控制。会话中的进程组被分成一个单独的前台进程组，0 个或者多个后台进程组。当用户退出终端时，一个 SIGQUIT 信号会发送到前台进程组的所有进程。当终端检测到网络连接时，一个 SIGHUP 信号会被发送到前台进程组的所有进程。当用户输入中断键(通常 Ctrl-C)时，一个 SIGINT 信号会被发送到前台进程组的所有进程。因此，会话使得管理终端和登录更加简单。

作为一个回顾，假设一个用户登录到系统，他的登录 bash shell 的 pid 为 1700。用户的 bash 现在是一个新进程组的唯一成员，也是该进程组的领导者，进程组 ID 为 1700。这个进程组在一个会话 ID 为 1700 的会话中，并且 bash 是这个会话的唯一成员，也是会话的领导者。用户在 shell 中运行的新命令将运行于同一个会话的新进程组中。其中的一个进程组（直接连接到用户并控制终端的那个）就是前台进程组。所有其它的进程组都是后台进程组。

在一个特定的系统中，存在许多会话：每个用户的登录会话、以及其它不捆绑于用户登录的会话，例如 daemon。Daemon 创建自己的会话，避免与其它可能退出的会话相关联。

每个会话包含一个或者多个进程组，每个进程组包含至少一个进程。进程组包含多个进程通常用来实现工作控制。

shell 中执行如下命令：

```
$ cat ship-inventory.txt | grep booty | sort
```

会产生一个进程组包含三个进程。通过这种方式，shell 可以一次信号所有三个进程。因为用户在控制台输入命令没有结尾的&，我们知道这个进程组会在前台运行。图 5-1 说明了会话、进程组、进程以及控制终端之间的关系。

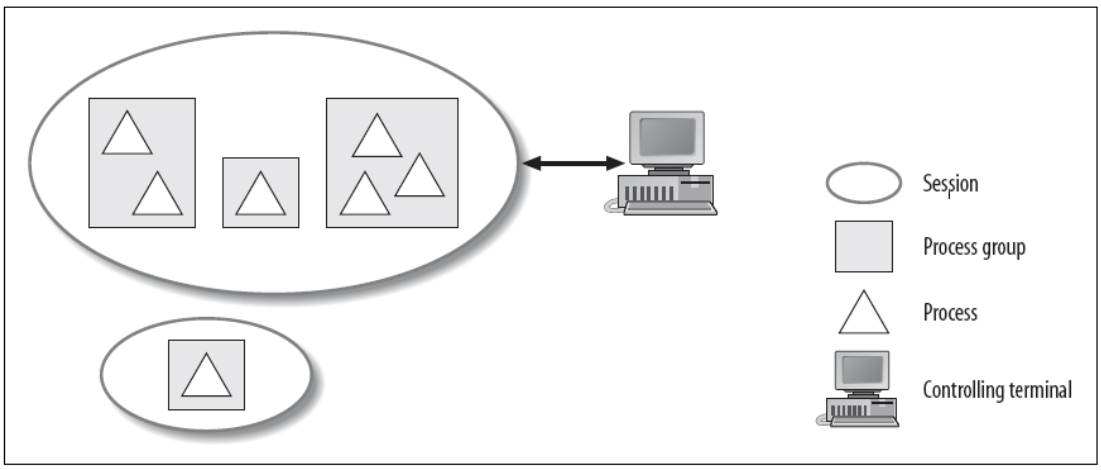


图 5-1. 会话、进程组、进程、和控制终端之间的关系

Linux 提供了几个接口设置和获取某个进程相关联的会话和进程组。这些接口主要用在 shell 上，但是对 daemon 进程也有用。

会话系统调用

shell 在登录时创建新会话，通过一个特殊的系统调用完成这个操作，这个系统调用使得创建新会话非常简单：

```
#include <unistd.h>
pid_t setsid (void);
```

调用 setsid()创建一个新会话，假设当前进程还不是某个进程组领导者。调用进程成为新创建会话的领导者和唯一的成员，会话没有控制 tty。调用同时在会话中创建一个新的进程组，并使调用进程成为进程组领导者和唯一成员。新会话和进程组的 ID 被设置为调用进程的 pid。

换句话说，setsid()在一个新会话中创建一个新的进程组，并使调用进程成为二者的领导者。这对 daemon 非常有用，daemon 通常不希望成为已存在会话的成员，也没有控制终端；对于 shell 也同样有用，对每个登录用户都要创建一个新的会话。

成功时 setsid()返回新创建会话的 ID；错误时返回-1，唯一可能的 errno 值是 EPERM，表示进程当前已经是进程组领导者。确保指定进程不是进程组领导者最简单的办法是 fork，父进程终止，然后让子进程执行 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;
    }
```

获取当前会话 ID 不那么有用，但也是可能的：

```
#define _XOPEN_SOURCE 500
#include <unistd.h>
pid_t getsid (pid_t pid);
```

调用 getsid()返回进程 pid 的会话 ID。如果 pid 参数为 0，getsid()返回调用进程的会话 ID。错误时调用返回-1，唯一可能的 errno 值是 ESRCH，表示 pid 没有对应到合法进程。注意老的 Unix 系统也可能设置 errno 为 EPERM，表示 pid 和调用进程不属于同一个会话；Linux 不会返回这个错误，可以返回任何进程的会话 ID。

使用比较少，主要是诊断目的：

```
pid_t sid;
sid = getsid (0);
if (sid == -1)
    perror ("getsid"); /* should not be possible */
else
    printf ("My session id=%d\n", sid);
```

进程组系统调用

调用 setpgid()设置 pid 进程的进程组 ID 为 pgid：

```
#define _XOPEN_SOURCE 500
#include <unistd.h>

int setpgid (pid_t pid, pid_t pgid);
```

如果 pid 参数为 0，则设置当前进程。如果 pgid 为 0，则 pid 进程的 ID 将作为进程的组 ID。

成功时 setpgid()返回 0。成功需要几个条件：

- pid 进程必须是调用进程、或者是调用进程的子进程，而且必须没有发起过 exec 调用，还要与调用进程在同一个会话中。
- pid 进程必须不是会话领导者。
- 如果 pgid 已经存在，它必须与调用进程在同一个会话中。
- pgid 必须非负。

错误时 setpgid()返回-1，并设置 errno 为下列错误代码之一：

EACCESS

pid 进程是调用进程的子进程，但是已经调用过 exec。

EINVAL

pgid 小于 0。

EPERM

pid 进程是会话领导者，或者与调用进程在不同的会话中。或者尝试移动进程到另一个

会话的进程组中。

ESRCH

pid 不是当前进程，0，或者当前进程的子进程。

和会话一样，获取进程的进程组 ID 是可能的，但不是特别有用：

```
#define _XOPEN_SOURCE 500
#include <unistd.h>
pid_t getpgid (pid_t pid);
```

调用 getpgid()返回 pid 进程的进程组 ID。如果 pid 为 0，返回当前进程的进程组 ID。错误时返回-1，并设置 errno 为 ESRCH，这是唯一可能的值，表示 pid 是无效的进程标识。

使用同样主要是出于诊断目的：

```
pid_t pgid;
pgid = getpgid (0);
if (pgid == -1)
    perror ("getpgid"); /* should not be possible */
else
    printf ("My process group id=%d\n", pgid);
```

废弃的进程组函数

Linux 支持两个更老的 BSD 接口，用来操作和获取进程组 ID。由于它们不如前面讨论过的系统调用有用，新程序应该只有在需要兼容性时才使用它们。setpgrp()可以设置进程组 ID：

```
#include <unistd.h>
int setpgrp (void);
```

下面调用：

```
if (setpgrp ( ) == -1)
    perror ("setpgrp");
```

等同于下面调用：

```
if (setpgid (0,0) == -1)
    perror ("setpgid");
```

二者都尝试分配当前进程到进程组 ID 为当前进程 pid 的进程组中。setpgrp()成功时返回 0；失败时返回-1。setpgid()的所有 errno 代码都适用于 setpgrp()，除了 ERSCH。

类似地，调用 getpgrp()可以获取进程组 ID：

```
#include <unistd.h>
pid_t getpgrp (void);
```

下面调用：

```
pid_t pgid = getpgrp ( );
```

等同于：

```
pid_t pgid = getpgid (0);
```

二者都返回调用进程的进程组 ID。函数 getpgid()不会失败。

Daemon

daemon 是在后台运行的进程，不与任何控制终端相关联。daemon 通常在启动时间开始运行，以 root 用户或者其它特殊用户（例如 apache 或者 postfix）运行，处理系统级任务。作为一个惯例，daemon 的名字通常以 d 结束（例如 crond 和 sshd），但这不是必需的。

daemon 这个名字来源于麦克斯韦 daemon，1867 年物理学家詹姆斯·麦克斯韦的思想实验。daemon 同时也是希腊神话中超自然的意思，存在于人和上帝之间的某个地方，有着天才和神奇的能力。不像犹太基督学里的 daemon，希腊的 daemon 不是代表邪恶。实际上，神话中的 daemon 往往是上帝的助手，执行奥林匹斯山的居民不愿意做的事情——和 Unix daemon 执行前台用户避免做的事情非常类似。

daemon 有两个通用的需求：它必须以 init 的子进程运行；并且它不能连接到终端。

一般来说，一个程序执行以下步骤成为一个 daemon 程序：

1. 调用 fork()。这将创建一个新进程，这个新进程将成为 daemon。
2. 在父进程中调用 exit()。确保原始父进程（daemon 的祖父）控制它的子进程完全终止，daemon 的父进程不再运行，并且 daemon 不是进程组领导者。最后一点是成功完成下一步骤的必需条件。
3. 调用 setsid()，给 daemon 创建一个新的进程组和会话，它也成为二者的领导者。这同时也保证进程不与控制终端相关联（因为进程刚刚创建了一个新会话）。
4. 通过 chdir()改变工作目录为 root 目录。这样做是因为继承到的工作目录可能在文件系统的任何地方。daemon 一般在系统运行期间一直运行，你不会想保持某个随机目录打开，从而阻止管理员卸载包含该目录的文件系统。
5. 关闭所有文件描述符。你并不想继承已打开的文件描述符，或者保持它们打开。
6. 打开文件描述符 0、1、2（标准输入、标准输出、标准错误），并重定向到/dev/null

按照这些规则，下面是一个程序把自己变成 daemon：

```
#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;
        /* create new process */
        pid = fork ( );
        if (pid == -1)
                return -1;
        else if (pid != 0)
                exit (EXIT_SUCCESS);
```

```
        /* create new session and process group */
        if (setsid ( ) == -1)
                return -1;

        /* set the working directory to the root directory */
        if (chdir ("/") == -1)
                return -1;

        /* close all open files--NR_OPEN is overkill, but works */
        for (i = 0; i < NR_OPEN; i++)
                close (i);

        /* redirect fd's 0,1,2 to /dev/null */
        open ("/dev/null", O_RDWR); /* stdin */
        dup (0); /* stdout */
        dup (0); /* stderror */

        /* do its daemon thing... */

        return 0;
}
```

大部分 Unix 系统在 C 库中提供 daemon()函数，自动化这些步骤，把烦琐变为简单：

```
#include <unistd.h>
int daemon (int nochdir, int noclose);
```

如果 nochdir 非 0，daemon 将不改变工作目录为 root 目录；如果 noclose 非 0，daemon 将不关闭所有已打开文件描述符。如果父进程已经设置 daemon 过程，这些选项就非常有用。不过通常都给这两个参数传递 0。

成功时调用返回 0；失败时返回-1，并设置 errno 为 fork()或者 setsid()的错误代码。

总结

本章我们讲解了 Unix 进程管理的基础，从进程创建到进程终止。在下一章，我们讨论更加高级的进程管理接口，例如改变进程调度行为的接口等。

第六章 高级进程管理

第五章介绍了进程的抽象，并且讨论了创建、控制、和销毁进程的内核接口。本章基于这些基本知识，开始讨论内核进程调度器和它的调度算法，然后介绍高级进程管理接口。这些系统调用操作进程的调度行为和语义，以应用和用户指示的目标影响调度器的行为。

进程调度

进程调度器是内核的一个组成，由它选择下一个运行的进程。换句话说，进程调度器（或者调度器）是内核的子系统，划分有限的处理器资源给系统的所有进程。在决定哪个进程可以运行时，调度器负责最大化处理器的使用率，同时提供多个进程同步无缝执行的印象。

在本章，我们会多次提到“可运行”的进程。可运行进程首先是没有被阻塞。与用户交互、大量读取和写入文件、或者响应 I/O 或者网络事件的进程，在等待资源可用时往往花费大量时间阻塞，它们在这个长时期内阻塞不能被运行（长时期，是相对于执行机器指令的时间来说的）。可运行进程还必须至少拥有剩余的时间片——调度器允许进程运行的时间数量。内核把所有可运行进程放置到一个运行列表。一旦一个进程耗尽了自己的时间片，就把它从列表中移除，直到所有其它可运行进程耗尽时间片之前，这个进程都不再是可进行的。

假设只有一个可运行进程（或者一个都没有），进程调度器的工作将非常简单。然而当有超过处理器数量的可运行进程时，调度器就体现出它的价值。在这种情况下，某些进程在运行，而另一些则必须等待。决定哪个进程运行，什么时候运行，运行多长时间就是进程调度器最基本的职责。

单处理器机器上的操作系统如果可以穿插多个进程的执行，则称为多任务操作系统。在多处理器机器上，多任务操作系统允许进程并行运行在不同的处理器上。非多任务操作系统，例如 DOS，则同时只能运行一个应用。

多任务操作系统有两个种类：协作式和抢先式。Linux 实现了多任务的后一种形式，调度器决定什么时候一个进程停止运行，而让另一个不同的进程继续运行。我们把挂起一个运行中的进程的动作称为抢占。再次重复，进程在被调度器抢占之前能够运行的时间称为进程时间片（这样称呼是因为调度器为每个可运行进程分配了一小片处理器时间）。

反过来在协作式多任务操作系统中，进程不会被停止运行，除非进程自己决定停止运行。我们把进程自愿挂起的动作称为让步。理想情况下，进程应该经常地让步，但操作系统没有办法强制这个行为。一个粗鲁或者破坏的程序可以长时间运行，甚至破坏整个系统。由于这种方式的这个缺点，现代操作系统几乎都是抢先式多任务；Linux 也不例外。

O(1)进程调度器，在 2.5 系列内核被引入，是 Linux 调度的核心。Linux 调度算法提供抢先多任务实现，支持多处理器、处理器 affinity、非统一内存访问(NUMA)配置、多线程、实时进程、和用户指定优先权。

Big-Oh 符号

O(1)是 big-oh 符号的一个例子，用来表示一个算法的复杂度和伸缩度。形式上：

If $f(x)$ is $O(g(x))$,

then

$\exists c, x'$ such that $f(x) \leq c \cdot g(x), \forall x > x'$

O(1)表示算法是常量级的。这一点提供了一个非常重要的承诺：Linux 进程调度器总是同样地完成任务，无论系统的进程数目。这一点很重要，因为挑选一个新进程运行可能需要多次遍历进程列表。糟糕的调度器（包括 Linux 早期版本使用的那些调度器）在系统进程数目增长时，这样的遍历会迅速增大并成为瓶颈。即使是最好的情况下，这种循环也为进程调度引入了不确定性。

Linux 调度器在任何情况下都会在常量时间内完成操作，没有类似的瓶颈。

时间片

Linux 为每个进程分配的时间片，是系统行为和性能非常重要的变量。如果时间片过长，进程必须在两次执行之间长时间等待，降低了系统的同步执行表现。用户可能因为延迟而感到沮丧。反过来，如果时间片过短，则系统时间的很大一部分被花费在切换不同的应用，同时失去了时间局部性的好处。

因此确定一个理想的时间片并不容易。有一些操作系统给进程分配大的时间片，希望最大化系统的吞吐量和整体性能；另一些操作系统则为进程分配非常小的时间片，希望为系统提供卓越的交互性能。后面我们将会看到，Linux 的目标是通过动态分配进程时间片，来同时达到这两个方面的最佳状态。

注意进程并不需要一次消耗完自己的所有时间片。分配 100ms 时间片的进程可以运行 20ms，然后对某些资源阻塞（例如键盘输入）。调度器会从可运行进程列表中临时地移除这个进程。当阻塞的资源可用时（在这种情况下，就是当键盘缓冲区不再为空时），调度器会唤醒这个进程。进程然后可以继续运行，直到耗尽剩余的 80ms 时间片，或者直到再次对某个资源阻塞。

I/O 限制进程 VS 处理器限制进程

持续消耗自己所有可用时间片的进程被称为处理器限制。这种进程对 CPU 时间饥饿，会消耗掉调度器给它的所有 CPU 时间。最简单的例子是无限循环。其它的例子包括科学计算、数学计算、和图像处理等。

另一方面，花费比执行更多时间在阻塞等待资源上的进程被称为 I/O 限制。I/O 限制进程通常发起并等待文件 I/O、阻塞键盘输入、或者等待用户移动鼠标。I/O 限制应用的例子包括那些只发起系统调用请求内核执行 I/O 操作的文件工具，例如 cp 或者 mv；以及许多 GUI 应用，它们花费更多时间在等待用户输入上。

处理器和I/O限制应用的区别在于为了获得最大的好处，调度器对他们采取的不同行为。处理器限制应用希望获得最大可能的时间片，允许他们最大化缓存命中率（通过时间局部性），并且尽可能快地完成工作。反过来，I/O 限制进程不需要大的时间片，因为它们在发起 I/O 请求和阻塞在内核资源之前，一般只运行非常短的时间。但是调度器定期的关注，确实能为 I/O 限制进程带来好处。这种应用在阻塞和分派 I/O 请求之后越快重新运行，就能越好地利用系统硬件。此外，如果应用等待用户输入，它越快被调度，用户就越能够感觉到无缝运行。

满足处理器限制和 I/O 限制进程的要求并不简单。Linux 调度器试图识别并提供 I/O 限制应用优先待遇：沉重的 I/O 限制应用被给予优先启动权，而沉重处理器限制应用则将受到优先权惩罚。

实际情况中，大部分应用都混合了 I/O 和处理器限制。音频/视频编码/解码是这种应用的一个好例子。许多游戏同样也混合这两种限制。并不总是能够识别某个应用的类型，并且

在任何时候，给定进程都可能是任何一种类型。

抢先式调度

当进程耗尽时间片时，内核挂起它并开始运行另一个进程。如果系统中没有可运行进程，内核对所有已耗尽时间片的进程重新分配时间片，然后再次运行进程。按这种方式，所有进程最终都能得到运行，即使系统中有更高优先级的进程存在——低优先级进程只需要等待高优先级进程耗尽它们的时间片或者阻塞。这种行为阐明了 Unix 调度的一个重要而默认的规则：所有进程都必须进行下去。

如果系统没有剩下可运行进程，内核将“运行”idle 进程。idle 进程实际上根本就不是一个进程；它也并没有实际地运行。相反 idle 进程是一个特殊的例程，内核执行它来简化调度算法，并且使得统计更加简单。空闲时间就是花费在运行 idle 进程的时间总和。

如果一个进程正在运行，而这时候一个更高优先级的进程变为可运行状态（可能它阻塞在等待键盘输入，而用户刚刚输入一个词），当前运行进程会立即被挂起，内核切换到更高优先级进程。因此，系统中不可能存在可运行却又没有运行的比当前运行进程更高优先级的进程。运行进程总是系统所有可运行进程优先级最高的那个。

线程

线程是一个单独进程内的执行单元，所有进程至少有一个线程。每个线程都有它自己的虚拟处理器：它自己的寄存器、指令指针、和处理器状态。虽然多数进程只有一个线程，进程也可以拥有大量的线程，所有都执行不同的任务，但共享相同的地址空间（因此也共享相同的动态内存、映射文件、目标代码等等）、打开文件列表、和其它内核资源。

Linux 内核对线程有一个有趣而独特的视角。本质上内核并没有线程这个概念。对 Linux 内核来说，所有线程都是单独的进程。从更宽泛的层次上来说，两个无关联进程和同一个进程内的两个线程没有区别。内核简单地把线程看成是共享资源的进程。也就是，内核认为包含两个线程的进程，就是两个共享一组内核资源（地址空间、打开文件列表等）的独立进程。

Multithreaded programming 就是多线程编程。Linux 线程编程最常用的 API 是由 IEEE Std 1003.1c-1995(POSIX 1995 或者 POSIX.1c)标准化的。开发者通常把实现这个 API 的库称为 pthreads。线程编程是一个复杂的话题，并且 pthreads API 也非常庞大和复杂。因此，pthreads 超出了本书的范围。本书关注于 pthreads 实现所基于的系统接口。

让出处理器

尽管 Linux 是抢先式多任务操作系统，它同时也提供了一个系统调用允许进程显式地让出执行，并指示调度器选择一个新进程执行：

```
#include <sched.h>
int sched_yield (void);
```

调用 sched_yield()将引起当前运行进程挂起，然后进程调度器选择另一个进程运行，运行新进程采用的方式与内核调度该进程抢占当前运行进程一样。注意如果没有其它可运行进程存在（这经常发生），yield 进程会立即继续执行。由于这个不确定性，通常都有比使用这

个系统调用更好的方式，sched_yield()并不常用。

成功时调用返回 0；失败时返回-1，并设置 errno 为适合的错误代码。在 Linux（以及多数 Unix 系统）中，sched_yield()永远不会失败，因此总是返回 0。然而彻底的程序员也仍然可以检查它的返回值：

```
if (sched_yield ( ))
    perror ("sched_yield");
```

合理的使用

在实践中，Linux 这样的抢先式多任务系统也有少数情况是 sched_yield()的合理使用方式。内核完全有能力做出最佳和最有效的调度决定——当然，内核能比单个应用程序更好地决定什么时候抢占哪个进程。这也正是为什么强调多任务协作的操作系统更喜欢抢先式多任务。

那么为什么我们需要一个请求“重新调度我”的系统调用呢？答案就在那些需要等待外部事件的应用程序上，事件可能由用户、硬件部分、或者其它进程引起。例如，如果一个进程需要等待另一个进程，“让出处理器直到另一个进程完成”就是第一个解决办法。如下面消费者/生产者的例子，一个幼稚的消费者实现类似如下：

```
/* the consumer... */
do {
    while (producer_not_ready ( ))
        sched_yield ( );
    process_data ( );
} while (!time_to_quit ( ));
```

值得感谢地是 Unix 程序员一般不会编写这样的代码。Unix 程序通常是事件驱动的，倾向于在消费者/生产者问题中利用一些可阻塞机制（例如管道），而不是使用 sched_yield()。在这种情况下，消费者从管道中读取，一直阻塞直到管道数据可用。而生产者则当有新数据时写入到管道中。这就去除了用户空间程序对同步的职责，也就是那个循环。对于内核来说，它可以优化地管理这种情况，通过睡眠进程，然后在需要时唤醒它。通常 Unix 程序应该使用基于事件驱动的解决方案，依赖于可阻塞的文件描述符。

到目前为止，有一种情况需要使用 sched_yield()：用户空间线程锁。当一个线程试图获得另一个线程已经拥有的锁时，新线程应该让出处理器，直到锁可用为止。在没有用户空间锁的内核支持时，这个方法是最简单的，也最高效。感谢的是，现代 Linux 线程实现（新的 POSIX 线程库，或者 NPTL）引入了一个使用 futexes 的优化解决方案，它提供了用户空间锁的内核支持。

sched_yield()的另一个使用是实现“精确控制”：一个处理器强烈的程序可以定期地调用 sched_yield()，以最小化它对系统的影响。不过仔细追究起来，这个策略有两个缺点：首先内核有能力做出比单个进程更好的全局调度决定，因此确保操作系统平滑运行的职责应该依赖于进程调度器而不是进程自己。更进一步说，调度器倾向于 I/O 强烈的应用，而对处理器强烈的应用作出一定惩罚。第二，照顾其它进程而减低一个处理器强烈的应用的开销，是用户的职责，而不是单个应用。用户可以传达他对应用性能的意愿，通过 nice 命令，本章后面我们会讨论它。

让出处理器：过去和现在

在 Linux 内核 2.6 之前，调用 sched_yield()只有很小的效果。如果另一个可运行的进程可用，内核就会切换到那个进程，并把当前调用进程放到可运行进程列表的末尾。很快内核就又可以重新调度调用进程。在没有其它可运行进程可用的情况下，调用进程继续执行。

内核 2.6 改变了这个行为，当前的算法如下：

1. 进程是不是实时进程？如果是则把它放到可运行进程列表末尾，然后返回（这是原来老的行为）；如果不是，则继续下一步骤。（关于更多实时进程，请参考本章后面“实时系统”一节）
2. 把进程从可运行进程列表中移除，并把它放到过期进程列表。这意味着在调用进程和其它过期进程能够继续执行之前，必须先等所有可运行进程执行完毕并耗尽它们的时间片。
3. 调度列表中的下一个可运行进程执行。

因此如果进程已经耗尽了自己的时间片，则调用 sched_yield()的作用是一样的。sched_yield()的行为与之前内核的区别在于现在它更加温和（等价于“如果另一个进程准备好并在等待，那就运行它，然后返回继续运行我”）。

这种改变的一个原因是防止被称为“乒乓”的病态行为。想像两个进程 A 和 B，两个都调用 sched_yield()。假设只有这两个进程是可运行进程（可能存在其它进程可运行，但都耗尽了时间片）。如果是老的 sched_yield()行为，这种情况的结果是内核会反复调度这两个进程，两个进程都向内核说“不，调度另一个进程吧”！这会持续到两个进程最终耗尽它们的时间片。如果我们把进程调度器选择进行运行画成图，就类似于“A, B, A, B, A, B…”。

sched_yield()的新行为防止了这种情况的发生。一旦进程 A 请求让出处理器，调度器就把它从可运行进程列表中移除。同样，一旦进程 B 做出相同请求，调度器也把它从可运行进程列表中移除。调度器不会再考虑运行进程 A 或者 B，除非没有其它可运行进程，防止了“乒乓”效果，并允许其它进程拥有公平的处理器时间。

因此，当进程请求让出处理器时，它最好是真正希望让出！

进程优先级

本节的讨论适合于普通非实时进程。实时进程需要不同的调度标准，和一个单独的优先级系统。我们会在本章后面讨论实时计算。

Linux 并不是随意地调度进程。相反，应用程序在运行时被指派一个优先级，并指定多长时间。Unix 历史性地把优先级称为 nice 值，因为它背后的意思是通过降低进程的优先级，对其它进程更加友好，允许其它进程消耗更多的系统处理器时间。

nice 值在进程运行时指定。Linux 按优先级由高到低调度可运行进程：拥有更高优先级的进程在优先级更低的进程之前运行。nice 值同时还指定进程的时间片。

合法的 nice 值范围是[-20, 19]，默认值为 0。有点混乱的是，进程的 nice 值越低，它的优先级越高，时间片也越大；反过来，nice 值越大，则进程的优先级越低，时间片也越小。因此增大进程的 nice 值对于系统其它进程来说是“更友好的”。数值倒置非常容易混淆。当我们说一个进程拥有“高优先级”时，意味着它将更快地被选择运行，并且可以比低优先级进程运行更长的时间，但这样一个进程的 nice 值会更小。

nice()

Linux 提供几个系统调用来获得和设置进程的 nice 值，最简单的一个是 nice()：

```
#include <unistd.h>
int nice (int inc);
```

成功调用 nice()对进程的 nice 值增加 inc，并返回更新后的 nice 值。只有拥有 CAP_SYS_NICE 能力的进程（也就是 root 用户的进程）才能提供非负的 inc 值，来减小 nice 值而提高进程的优先级。因此，非 root 进程只能降低自己的优先级（通过增大 nice 值）。

错误时 nice()返回-1。不过由于 nice()返回新的 nice 值，-1 同时也是合法的返回值。为了区分 nice()调用成功还是失败，你可以在调用之前把 errno 置为 0，调用后再检查 errno 值，例如：

```
int ret;
errno = 0;
ret = nice (10); /* increase our nice by 10 */
if (ret == -1 && errno != 0)
    perror ("nice");
else
    printf ("nice value is now %d\n", ret);
```

Linux 只会返回一个错误：EPERM。表示调用进程尝试增大优先级（通过负的 inc 值），但是它并没有 CAP_SYS_NICE 能力。其它 Unix 系统在 inc 值导致 nice 值超出合法范围时，也可能返回 EINVAL。但 Linux 不这样，相反，Linux 在需要时默默地舍入非法的 inc 值到 nice 值的合法范围。

传递 0 给 inc 是得到当前 nice 值的一个简单方法：

```
printf ("nice value is currently %d\n", nice (0));
```

通常一个进程希望设置 nice 值的绝对值，而不是相对的增长。这可以通过类似下面的代码来实现：

```
int ret, val;

/* get current nice value */
val = nice (0);

/* we want a nice value of 10 */
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()和 setpriority()

更好的解决方案是使用 getpriority()和 setpriority()系统调用，它们允许更多地控制，但操作起来也更加复杂：

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

int getpriority (int which, int who);
int setpriority (int which, int who, int prio);
```

这些调用对进程、进程组、或者用户进行操作，由 which 和 who 指定。which 的值必须是 PRIO_PROCESS、PRIO_PGRP、或者 PRIO_USER，在这三种情况下 who 分别指定进程 ID、进程组 ID、或用户 ID。如果 who 为 0，则调用分别对当前进程 ID、进程组 ID、或者用户 ID 进行操作。

调用 getpriority()返回所有指定进程中最大的优先级（也就是最小的 nice 值）。调用 setpriority()设置所有指定进程的优先级为 prio。和 nice()一样，只有拥有 CAP_SYS_NICE 能力的进程才能提升进程的优先级（通过降低 nice 值）。此外，只有拥有这个能力的进程才能提升或者降低其它用户的进程。

和 nice()一样，getpriority()在错误时返回-1。由于这也可能是成功调用时的返回值，如果程序员想处理错误条件，应该在调用前清除 errno 值。调用 setpriority()没有这个问题，setpriority()成功时总是返回 0；错误时返回-1。

下面代码返回当前进程的优先级：

```
int ret;
ret = getpriority (PRIO_PROCESS, 0);
printf ("nice value is %d\n", ret);
```

下面代码设置当前进程所在组的所有进程的优先级为 10：

```
int ret;
ret = setpriority (PRIO_PGRP, 0, 10);
if (ret == -1)
    perror ("setpriority");
```

错误时，两个函数都设置 errno 为以下值之一：

EACCESS

进程试图提升指定进程的优先级，但不拥有 CAP_SYS_NICE 权限。

EINVAL

which 指定的值不是 PRIO_PROCESS、PRIO_PGRP、或者 PRIO_USER。

EPERM

相应进程的有效用户 ID 与运行进程的有效用户 ID 不匹配，并且运行进程不拥有 CAP_SYS_NICE 权限。

ESRCH

未找到与 which 和 who 指定相匹配的进程。

I/O 优先级

除了调度优先级，Linux 也允许进程指定一个 I/O 优先级。这个值影响进程 I/O 请求的相关优先级。内核的 I/O 调度器按进程 I/O 优先级从高到低的顺序提供服务。

默认情况下 I/O 调度器使用进程的 nice 值来确定 I/O 优先级。因此设置 nice 值自动改变 I/O 优先级。然而 Linux 内核额外提供了两个系统调用来显式地设置和获得进程的 I/O 优先级，它们和 nice 值无关：

```
int ioprio_get (int which, int who)
int ioprio_set (int which, int who, int ioprio)
```

不幸地是，内核至今没有导出这两个系统调用，glibc 也没有提供任何用户空间访问。没有 glibc 的支持，使用非常麻烦。此外，当 glibc 支持到来时，接口可能已经和系统调用不同了。在相关的支持到来之前，有两个可移植的方法来操作进程的 I/O 优先级：通过 nice 值、或者类似 ionice 的工具，它是 linux 工具包的一个。

不是所有 I/O 调度器都支持 I/O 优先级。特别地，完全公平队列（CFQ）I/O 调度器支持，而目前其它标准调度器均不支持。如果当前 I/O 调度器确实不支持 I/O 优先级，它将被默默地忽略。

处理器亲和力

Linux 在单个系统中可以支持多处理器。除了启动进程，多处理器支持的主要工作都由进程调度器完成。在一个对称多处理器(SMP)机器中，进程调度器必须决定哪个进程运行在哪个 CPU 上。完成这项职责有两个挑战：调度器必须最大可能地利用系统的所有处理器，因为有进程在等待运行时让某个 CPU 空闲是非常低效的。

但是，一旦进程被调度到某个 CPU，进程调度器在将来也应该把该进程调度到同一个 CPU 上。这样做是因为把进程从一个 CPU 移到另一个 CPU 有一定的开销。

这其中最大的开销与移动进程的缓存作用有关。由于现代 SMP 系统的优化设计，每个处理器相关联的缓存是独立而不同的。也就是说，某个处理器的缓存数据在另一个处理器的缓存中并不存在。因此，如果一个进程被移动到另一个 CPU，并且写新数据到内存中，则原来那个 CPU 的缓存数据会变得陈旧。依赖于那段缓存就会引起问题。要防止这种情况，当处理器缓存新的内存块时要清除其它的数据。因此，某一段数据在任何给定时间内严格地只存在于一个处理器的缓存中。当进程从一个处理器移动到另一个处理器时，就会有两个方面的花费：被移动进程的缓存数据不再可访问；并且必须使进程原有的缓存数据无效。因为这些开销，进程调度器会试图尽可能久地保持进程运行在同一个处理器上。

进程调度器的这两个目标可能是相互冲突的。如果一个处理器比另一个处理器的进程负载大得多——或者更严重的，如果一个处理器繁忙而另一个空闲——这时候把某些进程调度到空闲 CPU 是有意义的。在这种不平衡的情况下决定何时移动进程，被称为负载均衡，是 SMP 系统性能的重要方面。

处理器亲和力指的是进程持续被调度在相同处理器上的可能性。术语“软亲和”指的是调度器持续调度进程到相同处理器的自然倾向。正如我们讨论过的，这是个值得做的特性。Linux 调度器试图尽可能久地调度同一进程到相同的处理器上，只有在负载极端不平衡时才会把进程从一个处理器调度到另一个处理器。这样就允许调度器最小化移动时的缓存影响，但仍然保证了系统中所有处理器的负载。

不过有时候用户或者应用希望强制“进程—处理器”之间的绑定。这通常是因为进程对缓存要求非常高，或者希望保持在同一个处理器上。绑定进程与特定的处理器，并且让内核强制这种关联就被称为“硬亲和”。

sched_getaffinity()和 sched_setaffinity()

进程继承父进程的 CPU 亲和力，并且默认可以运行在任何一个 CPU 上。Linux 提供两个系统调用来获得和设置进程的硬亲和：

```
#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);
```

调用 sched_getaffinity()获得 pid 进程的 CPU 亲和力，并且把它保存在特殊的 cpu_set_t 类型中，后者通过特殊的宏来访问。如果 pid 为 0，则调用返回当前进程的亲和力。setsize 参数是 cpu_set_t 类型的大小，glibc 使用它来保持兼容性以便将来修改这个类型的大小。成功时 sched_getaffinity()返回 0；失败时返回-1，并设置 errno 值。下面是一个例子：

```
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");
}
```

在调用前，我们使用 CPU_ZERO 来“清零”set 中的所有位。然后对 set 进行从 0 到 CPU_SETSIZE 的遍历。注意 CPU_SETSIZE 并不是 set 的实际大小——千万不要把它传递给 setsize

——它仅仅是 set 可能表示的处理器数目。由于目前的实现用一个位表示每个处理器，CPU_SETSIZE 比 sizeof(cpu_set_t)要大得多。我们使用 CPU_ISSET 来检查当前进程有没有与某个给定的处理器相绑定。它返回 0 表示未绑定，非 0 值表示绑定。

只有系统中物理存在的处理器才被设置。因此，在有两个处理器的系统中运行上面代码将得到如下输出：

```
cpu=0 is set
cpu=1 is set
cpu=2 is unset
cpu=3 is unset
...
cpu=1023 is unset
```

上面输出显示，CPU_SIZE（从 0 开始）当前的值是 1024。

我们只关心 CPU#0 和#1，因为系统中只有这两个物理处理器。如果我们希望确保进程只运行在 CPU#0，而永远不会运行在 CPU#1，下面代码可以实现：

```
cpu_set_t set;
int ret, i;

CPU_ZERO (&set); /* clear all CPUs */
CPU_SET (0, &set); /* allow CPU #0 */
CPU_CLR (1, &set); /* forbid 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");
}
```

我们同样先使用 CPU_ZERO 清零 set，然后通过 CPU_SET 设置 CPU#0，通过 CPU_CLR 清除 CPU#1。CPU_CLR 操作是多余的，因为之前已经清空过整个 set 了，不过为了讨论的完整性，我们还是使用了它。

在相同的两个处理器的系统中运行这个程序，会得到与前面稍微不同的输出：

```
cpu=0 is set
cpu=1 is unset
cpu=2 is unset
...
cpu=1023 is unset
```

现在，CPU#1 是未设置的。无论发生什么，这个进程都只会运行在 CPU#0 上。

有四个可能的 errno 错误代码：

EFAULT

提供的指针超出进程地址空间或者非法。

EINVAL

在这种情况下，系统中没有物理处理器在 set 中激活(仅对 sched_setaffinity())，或者 setsize 比内核内部描述处理器 set 的数据结构大小更小。

EPERM

pid 关联的进程与调用进程的当前有效用户 ID 不同，而且进程没有 CAP_SYS_NICE 权限。

ESRCH

pid 没有关联到进程(非法 pid)。

实时系统

在计算领域，术语“实时”通常是某些混淆和误解的来源。如果一个系统对操作有最终期限限制：开始操作和响应之间有最小和强制的时间，则这个系统就是实时的。一个大家都很熟悉的实时系统就是几乎所有现代汽车都拥有的“防锁刹车系统(ABS)”。在这个系统中，当踩下刹车时，有一个计算机控制刹车的压力，每秒钟内应用和释放最大刹车压力几次。这防止了车轮被锁住，否则将降低制动性能，甚至使汽车无控制刹车。在这样的系统中，操作最终期限就是系统对车轮锁住的响应时间，和系统应用刹车压力的时间。

硬 VS 软实时系统

实时系统有两种：硬实时和软实时。硬实时系统对操作最终期限有绝对要求。超过最终期限就是失败，而且是重大的 bug。而另一方面，软实时系统并不认为超过最终期限是严重的失败。

硬实时应用很容易识别：如防锁刹车系统、军用武器系统、医疗设备、和信号处理。软实时应用却并不总是这样容易识别。一个明显的例子是视频处理应用：如果最终期限到达，用户会注意到视频质量的下降，但丢失几帧却是可以接受的。

许多其它应用也有时间限制，如果达不到要求，将损害用户体验。多媒体应用、游戏、和网络程序都属于这种类型。但是文本编辑器呢？如果程序不能快速地响应按键，用户体验将非常糟糕，用户会因此而愤怒或沮丧。这是一个软实时应用吗？当然，当开发者编写这个应用时，他们意识到必须及时地响应按键输入。但是这是不是操作最终期限呢？定义软实时应用的界限是不清晰的。

与通常的想法相反，实时系统实际上并一定是快速的。实际上，给定相当的硬件，一个实时系统可能比非实时系统更慢——因为增加了支持实时处理的开销。同样，区分硬实时和软实时系统与操作最终期限的时间长短无关。如果 SCRAM 系统比控制棒更快，核反应堆将在几秒钟内过热。这就是一个拥有长时间最终期限的硬实时系统。反过来，如果应用不能在 100ms 内重新填充回放缓冲区，视频播放器可能跳过几帧或者影响声音。这就是一个对操作最终期限有严格要求的软实时系统。

Latency, Jitter 和 Deadlines

Latency 指的是从开始到执行返回之间的时间。如果 latency 少于或者等于操作最终期限，则系统是正确运行的。在许多硬实时系统中，操作最终期限和 latency 是相等的——系统按固定间隔以精确的时间处理 stimuli。在软实时系统中，响应则没有这么精确，latency 也有一定的变化——目标只是简单地使响应在最终期限内产生。

通常测量 latency 都比较困难，因为 latency 的计算需要准确地知道何时刺激事件发生。记录刺激事件通常需要响应它的能力。因此许多测量 latency 的工具实际上测量的并不是 latency，相反它们测量的响应之间的时间。相临事件之间的时间是 jitter，而不是 latency。

例如，考虑刺激事件每 10 毫秒发生一次。要测量系统的性能，我们可能记录下响应时间以确保它们每 10 毫秒发生一次。这里的时间并不是 latency 而是 jitter。我们测量的是相临两个响应之间的时间变化。在不知道刺激事件的确切发生时间时，我们不能够得到刺激事件与响应之间的时间差。即使知道刺激事件每 10 毫秒产生一次，我们也仍然不知道什么时候发生了第一次刺激事件。令人惊讶的是许多试图测量 latency 的行为都犯了这个错误，最终报告了 jitter 而不是 latency。当然 jitter 也是一个非常有用的值，类似的测量工具也是有用的。但无论如何，我们必须把鸭子称为鸭子。

硬实时系统通常有非常低的 jitter，因为他们在一个精确的时间量上响应刺激事件。类似系统的目标是使 jitter 值为 0，而 latency 等于操作延迟时间。如果 latency 超过了延迟时间，系统就失败了。

软实时系统对 jitter 的要求更加宽松。在这些系统中，响应时间只要在操作延迟内都是理想的——通常都更快一点。因此 jitter 通常是性能测量时 latency 的一个绝佳替代品。

Linux 的实时支持

Linux 通过一组由 IEEE Std 1003.1b-1993(通常简称为 POSIX 1993 或 POSIX.1b)定义的系统调用来提供软实时支持。

从技术上讲，POSIX 标准并没有规定系统所提供的实时支持是软还是硬。实际上 POSIX 标准所做的只是描述了几个支持优先权的调度策略。操作系统对这些策略强制的时间限制则完全留给 OS 设计者决定。

多年以来，Linux 内核已经获得了越来越好的实时支持，提供越来越低的 latency，和更一致的 jitter，而不影响系统的性能。这其中很大一部分原因是改进 latency 不仅仅只对实时应用有好处，也有助于许多种类的应用程序，例如桌面应用与 I/O 限制进程。这些改进同时也归因于 Linux 在嵌入式和实时系统领域的成功。

不幸的是，Linux 内核中的许多嵌入式和实时系统的修正只存在于特定的 Linux 内核解决方案中，而官方主流内核并不包含它们。其中一些修正提供进一步精确的 latency，甚至是硬实时行为。接下来的几节仅仅讨论官方内核接口和主流内核的行为。庆幸的是，多数实时修正也使用了 POSIX 接口。因此，后面的讨论也同样适用于特定的修改后系统。

Linux 调度策略与优先权

Linux 调度器对进程的行为依赖于进程的调度策略，也被称为调度类型。除了正常的默认策略，Linux 还提供了两个实时调度策略。头文件<sched.h>中的预处理宏定义描述了每一

种策略：SCHED_FIFO、SCHED_RR、和 SCHED_OTHER。

每个进程都拥有一个静态优先权，这个值与 nice 值无关。对于普通的应用，静态优先权总是 0。对于实时进程，它的范围是[1 ~ 99]。Linux 调度器总是选择最高优先权的进程运行。如果某个正在运行的进程有静态优先权 50，而这时另一个静态优先权为 51 的进程变成可运行状态，调度器会立即抢占正在运行的进程，并切换到那个新的可运行进程。反过来，如果正在运行的进程静态优先权为 50，而另一个优先权 49 的进程可运行，调度器则不会运行它除非优先权 50 的进程阻塞变为不可运行。由于普通进程的静态优先权为 0，任何可运行的实时进程都会抢占普通进程。

先进先出策略

先进先出(FIFO)是一种没有时间片的非常简单的实时策略。只要没有更高优先权的进程变为可运行，FIFO 类型的进程就会一直运行。FIFO 类型由宏 SCHED_FIFO 描述。

由于这个调度策略没有时间片，它运转的规则非常简单：

- 一个可运行 FIFO 类型的进程，如果它的优先权在系统中最高，则它会一直运行。特别地，一旦 FIFO 类型的进程成为可运行，它就会立即抢占普通进程。
- FIFO 类型的进程会持续运行，直到阻塞或者调用 sched_yield()，或者直到另一个更高优先权进程变为可运行。
- 当一个 FIFO 类型的进程阻塞时，调度器把它从可运行进程列表中移除。当它再次成为可运行时，它将会按优先权被插入到列表的末尾。因此，在其它相同或者更高优先权进程结束执行之前，这个进程将不会得到运行。
- 当一个 FIFO 类型的进程调用 sched_yield()时，调度器根据它的优先权把它移动到列表的末尾。因此，在其它相同优先权进程结束执行之前它将不会再次运行。如果调用进程优先权最高且唯一，sched_yield()将没有任何效果。
- 当一个更高优先权的进程抢占一个 FIFO 类型的进程时，FIFO 进程保留在进程列表的相同位置上。因此，一旦更高优先权进程结束执行，被抢占的 FIFO 类型进程会马上继续执行。
- 当一个进程加入 FIFO 类型，或者当一个进程的静态优先权变化时，它会被放置在相同优先权进程列表的开头。因此，一个新的 FIFO 类型而且有优先权的进程，可以抢占当前正在运行的具有相同优先权的进程。

本质上，我们可以说 FIFO 类型进程总是按自己想要的运行尽可能长的时间，只要它是系统中最高优先权的进程。上面的规则主要针对具有相同优先权的 FIFO 类型进程。

循环(round-robin)策略

round-robin(RR)类型和 FIFO 类型相同，除了它对相同优先权进程强加额外的规则。宏定义 SCHED_RR 描述了 round-robin 类型。

调度器为每个 RR 类型进程分配一个时间片。当一个 RR 类型进程耗尽它的时间片时，调度器把它移到当前优先权进程列表的末尾。按这种方式，相同优先权的所有 RR 类型进程将以循环的方式依次被调度。如果给定优先权只有一个进程，RR 类型就等同于 FIFO 类型。在这种情况下，当它的时间片耗尽时，进程会继续执行。

我们可以认为 RR 类型进程等同于 FIFO 类型的进程，除了当时间片耗尽时进程会停止运行，在这个时候它会被移到当前优先权可运行进程列表的末尾。

决定使用 SCHED_FIFO 还是 SCHED_RR 完全取决于相同优先权进程的行为。RR 类型的时间片仅仅对相同优先权的进程有效。FIFO 类型进程仍然会继续运行；RR 类型进程会在指定的优先权内进行调度。两种情况下低优先权的进程都不可能抢占高优先权进程。

标准策略

SCHED_OTHER 描述标准调度策略，它是默认的非实时调度类型。所有普通类型进程的静态优先权都为 0。因此，任何可运行的 FIFO 或者 RR 类型进程都将抢占正在运行的普通类型进程。

调度器使用前面讨论过的 nice 值，对普通类型进程进行优先权区分。nice 值和静态优先权没有任何关系，后者总是保持为 0。

批调度策略

SCHED_BATCH 是批或者空闲调度策略。它的行为和实时策略有一点对立：这个类型的进程只有在系统中没有其它可运行进程时才能运行，即使其它进程已经耗尽了时间片。这和拥有最大 nice 值（最低优先权）的进程也不一样，因为这样的进程最终仍然会运行，只要更高优先权的进程都耗尽了时间片。

设置 Linux 调度策略

进程可以通过 sched_getscheduler()和 sched_setscheduler()操作 Linux 的调度策略：

```
#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);
```

成功调用 sched_getscheduler()返回 pid 进程的调度策略。如果 pid 为 0，函数返回调用进程的调度策略。<sched.h>中定义的整数值描述了调度策略：先进先出策略是 SCHED_FIFO；循环策略是 SCHED_RR；普通策略是 SCHED_OTHER。错误时调用返回-1（它不是任何合法的调度策略），并且适当的设置 errno 值。

使用非常简单：

```
int policy;
```

```
/* get our scheduling policy */
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");
}
```

调用 sched_setscheduler()设置 pid 进程的调度策略为 policy。与具体策略相关的所有参数都通过 sp 设置。如果 pid 为 0，则设置调用进程的策略和参数。成功时调用返回 0；失败时返回-1，并设置 errno 为适当的值。

sched_param 结构体的合法域取决于操作系统支持的调度策略。SCHED_RR 和 SCHED_FIFO 策略需要一个域：sched_priority，表示静态优先权。SCHED_OTHER 策略不使用任何域。由于将来支持的调度策略可能使用新的域，可移植和合法的的程序不应该对结构体的结构作出假设。

设置进程的调度策略和参数也非常简单：

```
struct sched_param sp = { .sched_priority = 1 };

int ret;
ret = sched_setscheduler (0, SCHED_RR, &sp);

if (ret == -1) {
        perror ("sched_setscheduler");
        return 1;
}
```

这个代码片断设置调用进程的调度策略为循环策略，静态优先权为 1。我们假定 1 是合法的优先权——技术上讲它并不一定是合法的。我们会在下面一节讨论如何查找给定策略的合法优先权范围。

设置非 SCHED_OTHER 调度策略时，需要 CAP_SYS_NICE 权限。因此，一般由 root 用户运行实时进程。自从 2.6.12 内核开始，RLIMIT_RTPRIO 资源限制允许非 root 用户设置一定优先权最高限制的实时策略。

错误代码。出错时，有四个可能的 errno 值：

EFAULT

sp 指针指向非法或者不可访问的内存区域。

EINVAL

policy 表示的调度策略非法，或者 sp 中设置的某个值对指定策略没有意义(只对 sched_setscheduler())。

EPERM

调用进程没有必须的权限。

ESRCH

pid 值表示的不是正在运行的进程。

设置调度参数

POSIX 定义的 sched_getparam()和 sched_setparam()接口获得和设置调度策略相关的参数：

```
#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()接口只返回调度策略，而不返回任何相关的参数。调用 sched_getparam()通过 sp 返回 pid 相关联的调度参数：

```
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 为 0，函数返回调用进程的调度参数。成功时，函数返回 0；失败时返回-1，并设置 errno 为适当的值。

由于 sched_setscheduler()也设置相关的调度参数，sched_setparam()通常在稍后修改调度参数时很有用。

```
struct sched_param sp;
int ret;

sp.sched_priority = 1;
```

```
ret = sched_setparam (0, &sp);
if (ret == -1) {
    perror ("sched_setparam");
    return 1;
}
```

成功时，pid 的调度参数根据 sp 进行设置，并且调用返回 0；失败时调用返回-1，并设置 errno 为适当的值。

如果我们按顺序运行前面两个代码片断，可以看到如下输出：

```
Our priority is 1
```

这个例子再次假设 1 是合法的优先权。它确实是，不过可移植应用应该确保使用合法的优先权。我们马上就会看到如何检查合法优先权的范围。

错误代码

出错时，有四个可能的 errno 值：

EFAULT

指针 sp 指向非法或者不可访问的内存区域。

EINVAL

sp 中设置的某个值对指定的调度策略没有意义（只对 sched_setparam）

EPERM

调用进程没有必需的权限。

ESRCH

pid 值表示的不是正在运行的进程。

确定有效优先权的范围

我们前面例子传递的是硬编码优先权给调度系统调用。POSIX 并不保证哪些调度优先权在系统中一定存在，它只要求最小和最大值之间至少有 32 个优先权。在前面“Linux 调度策略和优先权”我们提到，Linux 对两个实时调度策略实现[1~99]的范围。可移植的程序通常都会实现自己的优先权范围，然后映射到操作系统的范围。例如，如果你想让所有进程运行在四个不同的实时优先级别上，你可以动态地确定优先权范围然后选择四个值。

Linux 提供两个系统调用来获取有效优先权的范围。一个返回最小优先权值，另一个返回最大优先权值：

```
#include <sched.h>

int sched_get_priority_min (int policy);
int sched_get_priority_max (int policy);
```

调用 sched_get_priority_min()成功时获得 policy 表示的调度策略的最小优先权值，调用 sched_get_priority_max()获得最大的有效优先权值，两个函数都返回 0；失败时，调用返回-1。唯一的错误是 policy 非法，这时候 errno 被设置为 EINVAL。

使用很简单：

```
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);
```

在标准 Linux 系统中，上面代码得到如下输出：

```
SCHED_RR priority range is 1 - 99
```

如前面讨论的，越大的优先权值表示越高的优先权。要设置进程为当前调度策略的最大优先权，你可以这样做：

```
/*
* set_highest_priority – set the associated pid's scheduling
* priority to the highest value allowed by its current
* scheduling policy. If pid is zero, sets the current
* process's priority.
*
* Returns zero on success.
*/
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;
}
```

程序一般先获得系统的最小和最大优先权值，然后使用增 1 或者减 1（例如 max-1，max-2）来给进程赋予优先权。

sched_rr_get_interval()

前面已经讨论过，SCHED_RR 进程的行为和 SCHED_FIFO 进程相同，除了调度器为 SCHED_RR 进程分配时间片以外。当 SCHED_RR 进程耗尽时间片时，调度器把进程移到当前优先权进程列表的末尾。按这种方式，所有具有相同优先权的 SCHED_RR 进程以循环的方式执行。更高优先权的进程（以及相同或更高优先权的 SCHED_FIFO 进程）总是会抢占一个正在运行的 SCHED_RR 进程，无论它的时间片是否耗尽。

POSIX 定义了一个接口获得指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);
```

成功调用 sched_rr_get_interval()把 pid 进程分配的时间片保存在 timespec 结构体中，然后返回 0；失败时调用返回-1，并适当地设置 errno 值。

根据 POSIX 的定义，这个函数只需要对 SCHED_RR 进程进行操作。然而在 Linux 中，它可以获得任何进程的时间片长度。可移植的应用应该假设这个函数只对循环调度进程工作正常；而 Linux 特定的程序则可以根据需要随意调用这个函数。下面是一个例子：

```
struct timespec tp;
int ret;

/* get the current task's timeslice length */
ret = sched_rr_get_interval (0, &tp);
if (ret == -1) {
        perror ("sched_rr_get_interval");
        return 1;
}

/* convert the seconds and nanoseconds to milliseconds */
printf ("Our time quantum is %.2lf milliseconds\n",
                (tp.tv_sec * 1000.0f) + (tp.tv_nsec / 1000000.0f));
```

如果进程是 FIFO 类型，tv_sec 和 tv_nsec 都将为 0，表示无穷。

错误代码

出错时，有三个可能的 errno 值：

EFAULT

指针 tp 指向的内存非法或者不可访问。

EINVAL

pid 数值非法（例如负数）。

ESRCH

pid 数值合法，但指向一个不存在的进程。

警惕实时进程

由于实时进程的这些特性，开发者在开发和调试实时程序时需要格外小心。如果草率地实现一个实时程序，系统可能变得无响应。实时程序中的任何 CPU 限制的循环——也就是不会阻塞的代码块——都会持续运行直到结束，只要没有更高优先权的实时进程成为可运行。

因此，设计实时程序需要特别的小心和注意。这种类型的程序有极大的优势，可以轻易地使整个系统崩溃。下面是一些技巧和警惕：

- 时刻记住如果系统中没有更高优先权的实时进程存在，任何 CPU 限制的循环都会一直运行直到结束，而不会被中断。如果是无限循环，系统将变得无响应。
- 因为实时进程以牺牲系统其它部分的代价来运行，必须对实时进程的设计给予特殊的关注。小心实现而不饥饿其它进程的系统处理器时间。
- 对繁忙等待需要非常小心。如果一个实时进程等待一个更低优先权进程拥有的资源，则这个实时进程将会永远等待。
- 当开发一个实时进程时，保持一个终端打开，运行为一个比开发中的进程更高优先权的实时进程。在紧急情况下，终端可以保持响应，并允许你杀死失控的实时进程（因为终端保持空闲，等待键盘输入，它不会受到其它实时进程的干扰）。
- chrt 实用工具是 util-linux 包工具中的一个，使获得和设置其它实时进程的属性非常简单。使用这个工具可以很容易地按实时调度类型启动任意程序（例如前面讲的终端），或者改变已存在应用的实时优先权。

确定性

实时进程的确定性很强。在实时计算中，如果给定相同的输入，实时计算的动作就是确定的，它总是在相同的时间里产生相同的结果。现代计算机因为许多因素却是非常不确定的：多级缓存（不可预测的命中率）、多处理器、页面、交换、和多任务，给估算一个动作需要花费的时间带来巨大的破坏。当然，我们已经使得每个操作都相当的快（特别是访问硬件驱动器），但同时现代系统也因此很难精确地计算给定操作将要花费的时间。

实时应用通常试图限制不可预料性，在最差的情况下会采取延迟特定的时间来实现。下面几节讨论两个实现这种功能的方法。

Prefaulting 数据和锁定内存

想像以下场景：硬件因自定义引入的 ICBM 监视器命中而中断，设备驱动快速地从硬件复制数据到内核中。驱动注意到某个进程处于睡眠中，阻塞在当前硬件设备节点，等待数据。驱动会告诉内核唤醒这个进程。内核注意到这个进程正在以实时调度策略以及一个高优先权运行，就会马上抢占当前正在运行的进程，并立即调度那个实时进程。调度器切换到实时进程，并切换到它的地址空间，这时候实时进程开始运行。整个过程需要花费 0.3ms，良好地在最差可以接受的 1ms 延迟内。

现在，在用户空间中，实时进程注意到进来的 ICBM，开始处理它的轨道。计算出轨道

之后，实时进程初始化部署反弹道导弹系统。这时候又过去 0.1ms——足够快部署 ABM 响应并拯救生命。但是——哦不！——ABM 代码刚刚竟然被交换到磁盘中去，引发一个页面错误，处理器切换回内核模式，然后内核发起硬盘 I/O 来获得已交换出去的数据。调度器把进程睡眠直到页面错误被解决。已经过去很多秒了，一切都晚了！

很明显，页面和交换引入了非常多可以破坏实时进程的不确定性行为。为了防止这种灾难，实时应用需要经常地“锁定”或者“hardwire”地址空间中的所有页面到物理内存中，prefaulting 它们到内存中，并防止它们被交换出去。一旦页面被锁定到内存中，内核永远不会把它们交换到磁盘中。对这些页面的任何访问都不会导致页面错误。大多数实时应用都会锁定部分或者所有页面到物理内存中。

Linux 提供 prefaulting 和锁定数据的接口。第四章讨论过把数据 prefaulting 到物理内存中的接口。第 8 章将讨论锁定数据到物理内存中的接口。

CPU 亲和力和实时进程

实时应用需要关心的第二点是多任务。尽管 Linux 内核是抢占式的，它的调度器却并不总是能够立即调度一个进程。有时候，当前运行的进程正在内核关键区域中运行，这时候调度器不能抢占它，直到它退出这个区域。如果正在等待运行的是实时进程，这个延迟可能是不可接受的，这样就会很快超出操作最后期限。

因此，多任务引入了类似于页面引起的不确定性。对多任务的解决方案也一样：消除它。当然，你不能简单地废除所有其它进程。如果在你的环境中可以这样做，你根本就不需要 Linux 了——简单的自定义操作系统就够了。如果你的系统有多个处理器，你可以为你的实时进程指定一个或者多个处理器。这样你就可以从多任务中保护你的实时进程。

本章前面我们已经讨论过操作进程的 CPU 亲和的系统调用。对于实时应用，一个可能的优化是为每个实时进程保留一个处理器，并让其它所有进程共享剩下的处理器。

实现这个目标最简单的方式是修改 Linux 的 init 程序，SysVinit，在它开始启动过程之前类似下面这样做：

```
cpu_set_t set;
int ret;

CPU_ZERO (&set); /* clear all CPUs */
ret = sched_getaffinity (0, sizeof (cpu_set_t), &set);
if (ret == -1) {
      perror ("sched_getaffinity");
      return 1;
}

CPU_CLR (1, &set); /* forbid CPU #1 */
ret = sched_setaffinity (0, sizeof (cpu_set_t), &set);
if (ret == -1) {
      perror ("sched_setaffinity");
      return 1;
}
```

这个代码片断夺取 init 允许使用的处理器集，我们想要的就是这个。然后从 set 中移除一个处理器 CPU#1，并更新允许使用处理器的列表。

由于子进程能够使用的处理器继承自父进程，而且 init 是所有进程的父进程，这样系统中所有进程都只能使用这个 set 中的处理器。因此，没有其它进程能够运行在 CPU#1 上。

接下来，修改你的实时进程，让它只运行在 CPU#1 上：

```
cpu_set_t set;
int ret;

CPU_ZERO (&set); /* clear all CPUs */
CPU_CLR (1, &set); /* forbid CPU #1 */
ret = sched_setaffinity (0, sizeof (cpu_set_t), &set);
if (ret == -1) {
    perror ("sched_setaffinity");
    return 1;
}
```

结果是你的实时进程只运行在 CPU#1，而且所有其它进程都运行在其它处理器上。

资源限制

Linux 内核对进程强加了几个资源限制。这些资源限制对进程能够消耗的内核资源做了最高限度的限制——也就是打开文件数量、内存页数、未决信号等等。限制是严格强制的；内核不允许进程消耗资源超出一个限制。例如，如果打开一个文件会导致进程拥有超过资源限制，open()调用将失败。

Linux 提供两个系统调用来操作资源限制。POSIX 标准化了这两个接口，但 Linux 比标准定义提供了几个额外资源限制的支持。资源限制使用 getrlimit()来检查，使用 setrlimit()来进行设置：

```
#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);
```

整数常量描述了这些资源，例如 RLIMIT_CPU。rlimit 结构体则描述了实际的限制。结构体定义了两个最高限度：软限制和硬限制。内核对进程执行软资源限制，但进程可以自由地改变自己的软限制，范围从 0 到硬限制。没有 CAP_SYS_RESOURCE 权限的进程（所有非 root 进程）只能降低自己的硬限制。无权限的进程永远不能提高自己的硬限制，即使是之前使用过的硬限制值也不行。降低硬限制是不能取消的。拥有权限的进程则可以设置硬限制为任意合法的值。

限制实际代表的含义依赖于具体的资源。例如，如果 resource 是 RLIMIT_FSIZE，限制表示进程可以创建的文件的最大大小（字节）。如果 rlim_cur 是 1024，进程就不能创建或者扩展文件大小超出 1K。

所有资源限制都有两个特殊的值：0 和无限。前者完全禁止进程使用该资源。例如，如果 RLIMIT_CORE 是 0，内核就永远不会创建 core 文件；反过来，后者对资源移除任何限制。内核通过特殊的 RLIM_INFINITY 来表示无限，它实际上恰好是-1（这可能会引起一定的混淆，因为-1 是出错时的返回值）。如果 RLIMIT_CORE 设置为无限，内核就可以创建任意大小的 core 文件。

getrlimit()函数把当前进程资源 resource 的硬限制和软限制存放到 rlim 指向的结构体中。成功时调用返回 0；失败时返回-1，并设置 errno 为合适的值。

对应的，函数 setrlimit()设置资源 resource 的硬限制和软限制为 rlim 指向的值。成功时调用返回 0，内核根据请求更新资源限制；失败时返回-1，并设置 errno 为适当的值。

限制

Linux 当前提供 15 个资源限制：

RLIMIT_AS

限制进程地址空间的最大大小（字节）。试图增大地址空间的大小而超过这个限制——通过类似于 mmap()和 brk()调用——将会失败，并返回 ENOMEM。如果进程堆栈不够，它将会按需要自动增长，并超出这个限制，内核也会给进程发送 SIGSEGV 信号。这个限制通常都是 RLIM_INFINITY。

RLIMIT_CORE

规定 core 文件的最大大小（字节）。如果值非 0，超过这个限制的 core 文件将被截断为 RLIMIT_CORE 值。如果 RLIMIT_CORE 为 0，将不会创建 core 文件。

RLIMIT_CPU

规定进程能够消耗的最大 CPU 时间（秒）。如果一个进程运行时间超过了这个限制，内核会给它发送 SIGXCPU 信号，进程可以捕获并处理这个信号。可移植的程序应该在收到信号后终止运行，因为 POSIX 对内核对此信号的处理动作未做定义。有一些系统可能会终止进程。但 Linux 允许进程继续执行，并按一秒的间隔持续发送 SIGXCPU 信号。一旦进程到达硬限制，内核将发送 SIGKILL 信号并终止进程。

RLIMIT_DATA

控制进程数据段和堆的最大大小（字节）。试图增大数据段超出这个限制(通过 brk())将会失败并返回 ENOMEM。

RLIMIT_FSIZE

指定进程可以创建的最大文件大小（字节）。如果进程增大一个文件超出这个大小，内核会给进程发送 SIGXFSZ 信号。默认情况下该信号会终止进程。但进程也可以选择捕获和处理这个信号，在这种情况下，I/O 系统调用失败，并返回 EFBIG。

RLIMIT_LOCKS

控制进程可以占有的文件锁最大数量（请参考第七章对文件锁的讨论）。一旦到达这个限制，再次尝试获得额外的文件锁将失败，并返回 ENOLCK。但是 Linux 内核 2.4.25 去除了这个功能。在当前内核中，这个限制可以设置，但没有任何效果。

RLIMIT_MEMLOCK

指定无 CAP_SYS_IPC 权限的进程通过 mlock()、mlockall()、shmctl()能够锁定的内存最大字节数。如果超出这个限制，这些调用将失败，并返回 EPERM。在实践中，有效的限制被舍入到页面的整数倍。具有 CAP_SYS_IPC 能力的进程可以锁定任意数量的页面到内存中，这个限制完全没有任何作用。在内核 2.6.9 之前，这个限制是 CAP_SYS_IPC 进程能够锁定内存的最大数量，而无权限的进程则完全不能锁定任何页面。这个限制不是

POSIX 标准，由 BSD 最早引入。

RLIMIT_MSGQUEUE

指定用户可以为 POSIX 消息队列分配的最大字节数。如果新创建的消息队列超出这个限制，mq_open()将失败，并返回 ENOMEM。这个限制也不是 POSIX 标准，在内核 2.6.8 中作为 Linux 特定限制而加入。

RLIMIT_NICE

指定进程可以降低 nice 值（提升权限）的最大值。如本章前面讨论的，普通进程只能增大自己的 nice 值（降低优先权）。这个限制允许管理员强加一个最大等级（nice 值的基准），进程可以根据这个限制合法地提升自己的权限。由于 nice 值可能为负，内核解释这个限制时使用 20 - rlim_cur。因此，如果限制设置为 40，进程能够降低 nice 最小值为-20（最高优先权）。内核 2.6.12 开始引入这个限制。

RLIMIT_NOFILE

指定进程可以保持打开的文件描述符最大数量，这个限制的值是最大数量值加 1。试图超过这个限制将导致失败，相关的系统调用返回 EMFILE。这个限制也可以通过 RLIMIT_OFILE 指定，后者是 BSD 相应限制的名称。

RLIMIT_NPROC

指定用户可以在系统中给定时间同时运行的最大进程数。试图超过这个限制将导致失败，fork()调用返回 EAGAIN。这个限制也不是 POSIX 标准，BSD 最早引入它。

RLIMIT_RSS

指定进程可以常驻内存（resident set size, RSS）的页面最大数量。只有 2.4 内核之前的版本强制这个限制。当前内核允许设置这个限制，但它不是强迫性的。这个限制同样也不是 POSIX 标准，BSD 最早引入它。

RLIMIT_RTPRIO

指定无 CAP_SYS_NICE 能力（非 root）的进程能够请求的最大实时优先级。通常非特权进程不能请求任何实时调度类型。这个限制不是 POSIX 标准定义，在内核 2.6.12 中加入，作为 Linux 特定扩展。

RLIMIT_SIGPENDING

指定当前可以排队的信号最大数量（标准和实时）。试图排列额外的信号将失败，而且系统调用（如 sigqueue()）返回 EAGAIN。注意无论这个限制怎么设置，总是可以排列第一个未排列的信号。因此，内核也总是可以发送给进程 SIGKILL 或者 SIGTERM。这个限制也不是 POSIX 标准，它是 Linux 特定的。

RLIMIT_STACK

指定进程堆栈的最大大小（字节）。超出这个限制将导致发送 SIGSEGV 信号。

内核对每个用户保存一份限制。换句话说，相同用户运行的所有进程对任何资源都拥有相同的软和硬限制。但是这个限制值本身描述的是每个进程而不是用户能够做的。例如，内核对每个用户维护一个 RLIMIT_NOFILE 值；默认情况下它是 1024。而这个限制表示的是每个进程可以打开的文件描述符最大数量，而不是当前用户能够打开的数量。注意这并不意味着限制可以对每个用户进程进行单独设置——如果某个进程改变了 RLIMIT_NOFILE 的软限制，这个改变会应用到当前用户拥有的所有进程。

默认限制

进程可用的默认限制依赖于三个因素：初始软限制、初始硬限制、和你的系统管理员。

内核指定初始软限制和初始硬限制；表 6-1 列出了所有默认值。内核对 init 进程设置这些限制，由于子进程继承父进程的限制，因此所有后续进程都继承 init 进程的软和硬限制。

表 6-1. 默认软和硬资源限制

资源限制	软限制	硬限制
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 pages	8 pages
RLIMIT_MSGQUEUE	800 KB	800 KB
RLIMIT_NICE	0	0
RLIMIT_NOFILE	1024	1024
RLIMIT_NPROC	0 (无限制)	0 (无限制)
RLIMIT_RSS	RLIM_INFINITY	RLIM_INFINITY
RLIMIT_RTPRIO	0	0
RLIMIT_SIGPENDING	0	0
RLIMIT_STACK	8 MB	RLIM_INFINITY

有两种办法可以改变这些默认限制：

- 任何进程都可以自由地增大软限制为[0 ~ 硬限制]之间的任意值，也可以减小硬限制。子进程在 fork 时将继承这些更新后的限制值。
- 特权进程可以自由地设置硬限制为任意值。子进程在 fork 时也将继承这些更新后的限制值。

通常普通进程世系中的特权进程不太可能会改变硬限制值，因此改变资源限制一般是上面第一个选项。实际上，描述进程的实际限制通常都由用户 Shell 设置，系统管理员可以提供不同的限制。例如 Bourne-again Shell(bash)，管理员通过 ulimit 命令设置限制。注意管理员不一定设置更小的值，它也可以增大软限制到硬限制值，给用户提供合适的默认值。通常使用 RLIMIT_STACK，在许多系统中它都被设置为 RLIM_INFINITY。

设置和获得限制值

我们已经解释了不同的资源限制，现在让我们看看怎样获得和设置限制值。获得资源限制非常简单：

```
struct rlimit rlim;
int ret;

/* get the limit on core sizes */
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);
```

编译并运行这个代码片断，得到下面输出：

```
RLIMIT_CORE limits: soft=0 hard=-1
```

软限制为 0，硬限制为无限（-1 表示 RLIM_INFINITY）。因此，我们可以设置新的软限制为任意值。下面例子设置最大 core 大小为 32MB：

```
struct rlimit rlim;
int ret;

rlim.rlim_cur = 32 * 1024 * 1024; /* 32 MB */
rlim.rlim_max = RLIM_INFINITY; /* leave it alone */
ret = setrlimit (RLIMIT_CORE, &rlim);
if (ret == -1) {
    perror ("setrlimit");
    return 1;
}
```

错误代码

出错时有三个可能的 errno 值：

EFAULT

rlim 指向的内存非法或者无法访问。

EINVAL

resource 表示的值非法，或者 rlim.rlim_cur 比 rlim.rlim_max 更大（setrlimit()时）。

EPERM

调用者不具备 CAP_SYS_RESOURCE 能力，但却试图增大硬限制。

第七章 文件和目录管理

在第 2、3、4 章，我们讨论了文件 I/O 的许多方法。在这一章，我们再访文件，不过这一次不再集中于读取或者写入文件，而是怎样操作和管理文件以及相关的元数据。

文件和元数据

正如第一章讨论过的，每个文件都由一个 inode 引用，inode 又通过文件系统唯一的 inode 数值进行寻址。inode 既是 Unix 系统中存放在磁盘中的物理对象，又是 Linux 内核中由数据结构描述的概念实体。inode 存储文件相关联的元数据，例如文件的访问权限、最后访问时间戳、拥有者、组、和大小，以及文件数据存储位置。

你可以通过 ls 命令的-i 标志获得文件的 inode 数值：

```
$ 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
```

例如输出显示 disk.c 的 inode 数值为 1689460。在这个特定的文件系统中，没有其它文件会使用同一 inode 值。当然在其它文件系统中，我们就不能保证这一点了。

stat 家族

Unix 提供一组函数来获取文件的元数据：

```
#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);
```

每个函数都返回某个文件的信息。stat()返回路径 path 表示的文件的信息；而 fstat()则返回文件描述符 fd 描述的文件的信息；lstat()等同于 stat()，除了在符号链接的情况下，lstat()返回链接本身而不是目标文件的信息。

每个函数都把信息存储在 stat 结构体中，结构体由用户提供。stat 结构体在<bits/stat.h>中定义，被包含在<sys/stat.h>头文件中。

```
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 */
};
```

更详细地说，各个域如下：

- st_dev 域描述文件存储的设备节点（我们在本章后面会讨论设备节点）。如果文件并不由设备后备——例如它存储在 NFS 挂载点——这个值为 0。
- st_ino 域描述文件的 inode 数值。
- st_mode 域描述文件的 mode 字节，第一章和第二章讨论了 mode 字节和权限。
- st_nlink 域描述文件指向的硬链接数目。每个文件至少有一个硬链接。
- st_uid 域描述拥有文件的用户的用户 ID。
- st_gid 域描述拥有文件的用户的组 ID
- 如果文件是一个设备节点，st_rdev 域描述了文件表示的设备。
- st_size 域描述文件的大小（字节）。
- st_blksize 域描述高效文件 I/O 首选的块大小。这个值（或者整数倍）是用户缓冲 I/O 最佳的块大小（参考第三章）。
- st_blocks 域描述文件系统分配给文件的块数目，如果文件有洞（稀疏文件），这个值会比 st_size 更小。
- st_atime 域描述文件的最后访问时间。这个值是文件最后一次被访问的时间（例如，通过 read()或者 execle()）。
- st_mtime 域描述文件的最后修改时间。也就是文件最后一次被写入的时间。
- st_ctime 域描述文件的最后改变时间。通常被误认为是文件的创建时间，实际上 Linux 或者其它 Unix 系统并不保存文件的创建时间。这个域实际上描述文件元数据（例如拥有者和权限）最后改变的时间。

成功时所有三个函数都返回 0，并存储文件的元数据到 stat 结构体中。错误时返回-1，并设置 errno 为以下之一：

EACCESS

调用进程缺少 path 指向的目录的查询权限。

EBADF

fd 非法

EFAULT

path 或者 buf 是非法指针。

ELOOP

path 包含了太多的符号链接。

ENAMETOOLONG

path 太长

ENOENT

path 中的某个组成不存在。

ENOMEM

完成请求需要的可用内存不够。

ENOTDIR

path 中的某个组成不是目录。

下面程序使用 stat()获得命令行指定的文件的大小：

```
#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;
}
```

下面是对源代码本身进行测试的结果：

```
$ ./stat stat.c
stat.c is 392 bytes
```

下面这段代码使用 fstat()检查已打开的文件是否存在于物理设备上（与网络相反）：

```
/*
* is_on_physical_device – returns a positive
* integer if 'fd' resides on a physical device,
* 0 if the file resides on a nonphysical or
* virtual device (e.g., on an NFS mount), and
* -1 on error.
```

```
*/
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);
}
```

权限

stat 调用也可以用来获得指定文件的权限值，有另外两个系统调用设置这些权限值：

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

int chmod (const char *path, mode_t mode);
int fchmod (int fd, mode_t mode);
```

chmod()和 fchmod()都设置文件的权限为 mode。使用 chmod()时，path 表示待修改文件的相对或者绝对路径。而使用 fchmod()时，文件通过文件描述符 fd 指定。

mode 的合法值由不透明的 mode_t 整数类型描述，mode_t 和 stat 结构体中的 st_mode 相同。尽管它的数值是简单的整数，但它们代表的含义是 Unix 实现特定的。因此，POSIX 定义了一组常量来描述各种权限（详细信息请参考第二章“新文件的权限”一节）。这些常量可以使用二进制 OR 操作组合到一起作为合法的 mode。例如，(S_IRUSR | S_IRGRP)设置文件的权限为拥有者和组可读。

要改变文件的权限，调用 chmod()或者 fchmod()函数的进程有效 ID 必须与文件拥有者匹配，或者进程拥有 CAP_FOWNER 能力。

成功时两个调用都返回 0；失败时都返回-1，并设置 errno 为以下错误代码之一：

EACCESS

调用进程缺乏查找路径 path 的权限（只对 chmod）。

EBADF

文件描述符 fd 无效（只对 fchmod）。

EFAULT

path 是非法指针（只对 chmod）。

EIO

文件系统发生内部 I/O 错误。这是非常严重的错误，它可能表示磁盘或者文件系统损坏。

ELOOP

解析 path 时内核遭遇太多的符号链接（只对 chmod）。

ENAMETOOLONG
path 太长（只对 chmod）。
ENOENT
path 不存在（只对 chmod）。
ENOMEM
完成请求需要的可用内存不够。
ENOTDIR
path 中的某个组成不是目录（只对 chmod）。
EPERM
调用进程的有效 ID 与文件拥有者不匹配，并且进程没有 CAP_FOWNER 能力。
EROFS
文件存在于只读文件系统。

下面代码片断设置文件 map.png 为拥有者可读和可写：

```
int ret;
/*
* Set 'map.png' in the current directory to
* owner-readable and -writable. This is the
* same as 'chmod 600 ./map.png'.
*/
ret = chmod ("./map.png", S_IRUSR | S_IWUSR);
if (ret)
    perror ("chmod");
```

下面代码片断做同样的事情，假设 fd 表示打开文件 map.png：

```
int ret;
/*
* Set the file behind 'fd' to owner-readable
* and -writable.
*/
ret = fchmod (fd, S_IRUSR | S_IWUSR);
if (ret)
    perror ("fchmod");
```

chmod()和 fchmod()在所有现代 Unix 系统中都可用。POSIX 强制要求前者，而定义后者为可选。

拥有者

在 stat 结构体中，st_uid 和 st_gid 域分别描述文件的拥有者和组。有三个系统调用允许用户改变这两个值：

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

```
int chown (const char *path, uid_t owner, gid_t group);
int lchown (const char *path, uid_t owner, gid_t group);
int fchown (int fd, uid_t owner, gid_t group);
```

chown()和 lchown()设置路径 path 指定的文件的拥有者。它们的作用相同，区别在文件是符号链接时：chown()跟随符号链接，并改变链接目标文件而不是链接本身的拥有者；而 lchown()并不跟随符号链接，而改变符号链接文件的拥有者。fchown()设置文件描述符 fd 描述的文件的拥有者。

成功时三个函数都设置文件的拥有者为 owner，设置文件的组为 group，并返回 0；如果 owner 或者 group 为-1，则不设置相应的值。只有拥有 CAP_CHOWN 能力的进程（通常是 root 进程）才可以改变文件的拥有者。文件的拥有者可以改变文件的组为用户存在的任意组；拥有 CAP_CHOWN 能力的进程可以改变文件的组为任意值。

失败时调用返回-1，并设置 errno 为以下值之一：

EACCESS

调用进程缺乏查找 path 的权限（只对 chown 和 lchown）。

EBADF

fd 无效。

EFAULT

path 无效。

EIO

内部 I/O 错误（这个很严重）。

ELOOP

内核在解析 path 时遇到太多的符号链接。

ENAMETOOLONG

path 太长。

ENOENT

文件不存在。

ENOMEM

完成请求需要的可用内存不足。

ENOTDIR

path 的某个组成不是目录。

EPERM

调用进程没有足够的权限改变拥有者或组。

EROFS

文件系统只读。

下面代码片断改变当前工作目录下的文件 manifest.txt 的组为 officers。要使代码成功运行，调用用户必须拥有 CAP_CHOWN 能力或者必须是 kidd 并且在 officers 组中：

```
struct group *gr;
int ret;
/*
* getgrnam( ) returns information on a group
* given its name.
*/
```

```
gr = getgrnam ("officers");
if (!gr) {
        /* likely an invalid group */
        perror ("getgrnam");
        return 1;
}

/* set manifest.txt's group to 'officers' */
ret = chmod ("manifest.txt", -1, gr->gr_gid);
if (ret)
        perror ("chmod");
```

在执行之前，文件的组是 crew：

```
$ ls  -l
-rw-r--r-- 1 kidd crew 13274 May 23 09:20 manifest.txt
```

执行之后，文件就是 officers 的成员了：

```
$ ls  -l
-rw-r--r-- 1 kidd officers 13274 May 23 09:20 manifest.txt
```

文件的拥有者 kidd，并没有改变，因为上面代码传递-1 给 uid。

下面函数设置 fd 描述的文件的拥有者和组为 root：

```
/*
* make_root_owner - changes the owner and group of the file
* given by 'fd' to root. Returns 0 on success and -1 on
* failure.
*/
int make_root_owner (int fd)
{
        int ret;

        /* 0 is both the gid and the uid for root */
        ret = fchown (fd, 0, 0);
        if (ret)
                perror ("fchown");

        return ret;
}
```

调用进程必须拥有 CAP_CHOWN 能力。这通常都意味着它必须属于 root。

扩展属性

扩展属性也叫 xattrs，为文件提供了一种永久性关联[键/值](key/value)对的机制。在本章前面，我们已经讨论了与文件相关联的所有 key/value 元数据：文件大小、拥有者、最后修改时间戳等等。扩展属性允许现有文件系统支持原始设计中缺少的特性，例如强制访问的安

全控制。扩展属性最有趣的地方是用户空间应用也可以任意地创建、读取、和写入 key/value 对。

扩展属性是文件系统不可知的，应用程序使用标准接口来操作它们；而接口并不特定于任何文件系统。应用因此可以使用扩展属性，而不关注文件所在的文件系统，或者文件系统内部怎样存储键和值。不过扩展属性的实现本身仍然是文件系统特定的。不同的文件系统使用不同的方式存储扩展属性，但是内核隐藏了这些差异，把它们抽象成统一的扩展属性接口。

例如 ext3 文件系统存储文件的扩展属性在文件 inode 的空闲空间中。这个特性使得读取扩展属性非常快速。因为应用访问文件时，包含 inode 的文件系统块从磁盘中读取到内存中，扩展属性也被“自动”读取到内存中，不需要任何额外开销就可以访问。

其它文件系统，例如 FAT 和 minixfs，完全不支持扩展属性。对这些文件系统中的文件调用扩展属性接口时会返回 ENOTSUP。

键和值

一个唯一的 key 标识每一个扩展属性。key 必须是合法的 UTF-8 字符。扩展属性采用 namespace.attribute 形式。每个 key 必须是完全限定的，也就是必须以合法的命名空间开始，跟随一个点，然后是 key。合法 key 的例子如 user.mime_type，这个 key 在 user 命名空间中，属性名为 mime_type。

key 可以是定义和未定义。如果 key 被定义，它的值可能是空也可能非空。也就是说未定义 key、和已定义 key 而并未赋值之间是有区别的。后面我们将看到，这意味着我们需要一个特殊的接口来移除 key（赋为空值是不够的）。

value 关联到一个 key，可以是任意字节数组。因为这个值并不一定是字符串，它并不需要以 null 字符结束，尽管存储 C 字符串为 value 时应该使用 null 结束字节数组。由于 value 不确保以 null 结束，对扩展属性的所有操作都需要 value 的大小。当读取属性时，内核提供大小；当写入属性时，你必须提供大小。

Linux 并不对 key 数量、key 长度、value 大小、以及一个文件相关联的所有 key 和 value 所能消耗的总空间做任何限制。但是文件系统却有实际的限制。这些限制通常显示为一个指定文件相关联的所有 key 和 value 的总大小。

例如 ext3 文件系统，给定文件的所有扩展属性必须能够存放在文件 inode 的空闲空间中，最多增加到一个额外的文件系统块。（老的 ext3 版本限制为一个文件系统块，没有存储在 inode 中）。在实践中这个大小通常是每个文件 1KB 到 8KB，具体依赖于文件系统的块大小。相反在 XFS 中则没有实际的限制。不过即使是 ext3，这些限制通常也不会成为问题，因为大多数 key 和 value 都是简短的文本字符。无论如何，请记住这点——在存储整个项目的修订版本历史信息到文件的扩展属性中时请三思。

存储 MIME 类型更好的方法

GUI 文件管理器，例如 GNOME 的 Nautilus，对不同文件的行为差异很大：提供独特的图标、不同的默认双击行为、特殊的执行操作列表等等。要实现这些，文件管理器必须知道每个文件的格式。要确定文件的格式，Windows 文件系统只是简单地查看文件的后缀。基于传统和安全的理由，Unix 文件系统倾向于检查文件并解析它的类型。这个过程就称为 MIME 类型嗅探。

有一些文件管理器动态地产生这个信息；另外一些则在产生这个信息之后缓存起来。使

用缓存信息的管理器一般把这些信息存放到自定义数据库中。这些文件管理器必须在数据库和文件之间保持同步，而这些信息可能在文件管理器不知道的情况下改变。一个更好的方式是摒弃自定义数据库，而存储这些元数据到扩展属性中：它更容易维护、访问速度更快、并且可以被任何应用访问。

扩展属性命名空间

扩展属性相关联的命名空间只不过是一个组织工具。内核对不同的命名空间强制执行不同的访问策略。

Linux 当前定义了四个扩展属性命名空间，未来还可能定义更多。当前的四个如下：

system

使用扩展属性来实现内核特性时将使用 system 命名空间，例如访问控制列表(ACL)。这个命名空间下的扩展属性如 system.posix_acl_access。用户是否能够读取和写入这些属性，依赖于扩展属性的安全模块。最严格的时候没有任何用户（包括 root）能够读取这些属性。

security

security 命名空间用来实现安全模块，例如 SELinux。用户空间应用是否能够访问这些属性，同样也依赖于安全模块。默认所有进程都可以读取这些属性，但只有拥有 CAP_SYS_ADMIN 能力的进程可以写入它们。

trusted

trusted 命名空间存储对用户空间受限的信息。只有拥有 CAP_SYS_ADMIN 能力的进程能够读取和写入这些属性。

user

user 命名空间是给普通进程使用的标准命名空间。内核通过普通文件权限位来控制对这个命名空间的访问。要读取一个已存在 key 的 value，进程必须拥有对指定文件的读取权限。要创建一个新 key，或者写入 value 到指定的 key，进程必须拥有对指定文件的写权限。在 user 命名空间你只能对普通文件赋予扩展属性，而不能是符号链接或者设备文件。当设计使用扩展属性的用户空间应用时，user 是你最有可能使用的命名空间。

扩展属性操作

POSIX 定义了四个操作，应用可以对指定文件的扩展属性执行这些操作：

- 给定一个文件和一个 key，返回相应的 value。
- 给定一个文件、一个 key、和一个 value，把 value 赋给 key。
- 给定一个文件，返回文件所有已赋值的扩展属性 key 的列表。
- 给定一个文件和一个 key，从文件中移除这个扩展属性。

对每个操作，POSIX 提供三个系统调用：

- 对指定路径进行操作；如果路径引用一个符号链接，将对链接指向的目标文件进行操作（通常的行为）。
- 对指定路径进行操作；如果路径引用一个符号链接，将对链接本身进行操作（标准系统调用的 l 变体）。
- 对文件描述符进行操作（标准 f 变体）。

在接下来的小节，我们将讨论所有 12 种组合。

获取扩展属性。最简单的操作就是指定一个 key，从文件中返回相应的扩展属性：

```
#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);
```

成功调用 getxattr()存储 path 文件的 key 的扩展属性到缓冲区 value 中，它的大小为 size 字节。函数返回 value 的实际大小。

如果 size 为 0，调用直接返回属性值的大小，而不把它存储到 value 中。因此，传递 0 允许应用确定缓冲区的正确大小。有了这个值，应用就可以分配或者调整缓冲区的大小。

lgetxattr()的行为和 getxattr()相同，除了在 path 是符号链接时，它会返回符号链接本身的扩展属性，而不是链接的目标文件。回忆下前面小节我们说过 user 命名空间不能对符号链接进行扩展属性操作——因此，这个调用很少使用。

fgetxattr()对文件描述符 fd 进行操作；除了这一点，它的行为和 getxattr()相同。

出错时，所有三个调用都返回-1，并设置 errno 为以下值之一：

EACCESS

调用进程缺乏查找 path 路径中的目录的权限(只对 getxattr()和 lgetxattr())。

EBADF

fd 非法(只对 fgetxattr())。

EFAULT

path、key、或者 value 是无效指针。

ELOOP

path 包含过多的符号链接(只对 getxattr()和 lgetxattr())。

ENAMETOOLONG

path 过长(只对 getxattr()和 lgetxattr())。

ENOATTR

属性 key 不存在，或者进程对属性没有访问权限。

ENOENT

path 的部分组成不存在(只对 getxattr()和 lgetxattr())。

ENOMEM

完成请求需要的可用内存不足。

ENOTDIR

path 的某个组成不是目录(只对 getxattr()和 lgetxattr())。

ENOTSUP

path 或者 fd 所在的文件系统不支持扩展属性。

ERANGE

size 太小不能存储 key 的值。如前面所说，也可以设置 size 为 0 发起调用，返回值就表示需要的缓冲区大小，然后就可以适当地调整 value 的大小。

设置扩展属性。下面三个系统调用设置指定的扩展属性：

```
#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);
```

成功调用 setxattr()设置 path 文件的扩展属性 key 的值为 value，它的大小为 size 字节。flags 域可以修改调用的行为。如果 flags 是 XATTR_CREATE，则扩展属性已经存在时调用将失败；如果 flags 是 XATTR_REPLACE，则扩展属性不存在时调用也将失败。默认行为——当 flags 为 0 时执行——允许同时创建和替换。无论 flags 为何值，其它 key 都不会受到影响。

lsetxattr()的行为和 setxattr()相同，除了 path 是符号链接时，它对链接本身设置扩展属性，而不是链接的目标文件。由于 user 命名空间不能对符号链接进行扩展属性操作，因此这个调用同样很少使用。

fsetxattr()对文件描述符 fd 进行操作；除了这一点，它的行为和 setxattr()相同。

成功时所有三个调用都返回 0；失败时返回-1，并设置 errno 为以下值之一：

EACCESS

调用进程缺乏查找 path 路径中的目录的权限。

EBADF

fd 非法。

EDQUOT

配额限制阻止了请求操作需要的空间消费。

EEXIST

flags 设置为 XATTR_CREATE，而 key 已经存在于指定文件。

EFAULT

path、key、或者 value 是无效指针。

EINVAL

flags 非法。

ELOOP

path 包含过多的符号链接。

ENAMETOOLONG

path 过长。

ENOATTR

flags 设置为 XATTR_REPLACE，而指定文件的 key 并不存在。

ENOENT

path 的某个组成部分不存在。

ENOMEM

完成请求所需的可用内存不足。

ENOSPC

文件系统没有足够的空间来存储扩展属性。

ENOTDIR

path 的某个组成部分不是目录。

ENOTSUP

path 或 fd 所在的文件系统不支持扩展属性。

列出文件的所有扩展属性。下面三个系统调用枚举指定文件的所有扩展属性 key 集合：

```
#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);
```

成功调用 listxattr()返回 path 表示的文件的所有扩展属性 key 列表。这个列表存储在 list 缓冲区中，大小为 size 字节。调用返回 list 的实际大小。

list 中返回的每个扩展属性 key 都以 null 字符结束，看起来可能像下面这样：

"user.md5_sum\0user.mime_type\0system.posix_acl_default\0"

因此尽管每个 key 都是传统的以 null 结束的 C 字符串，你仍然需要整个 list 的长度（函数的返回值），这样才能对 key 列表进行操作。要得到需要分配的缓冲区的大小，设置 size 为 0 调用任何一个列表函数，这样函数就会返回整个 key 列表的实际大小。和 getxattr()一样，应用可以使用这个功能来创建或者调整传递给 value 的缓冲区。

llistxattr()的行为和 listxattr()相同，除了当 path 是符号链接时，它枚举符号链接本身的所有扩展属性 key，而不是链接指向的目标文件。我们说过，user 命名空间不能对符号链接应用扩展属性，因此这个函数同样很少使用。

flistxattr()对文件描述符 fd 进行操作；除了这一点，它的行为和 listxattr()完全相同。

失败时所有三个调用都返回-1，并设置 errno 为以下错误代码之一：

EACCESS

调用进程缺乏查找 path 路径中的目录的权限。

EBADF

fd 非法。

EFAULT

path 或 list 是无效指针。

ELOOP

path 包含过多的符号链接。

ENAMETOOLONG

path 过长。

ENOENT

path 的某个组成部分不存在。

ENOMEM

完成请求需要的可用内存不足。

ENOTDIR

path 的某个组成部分不是目录。

ENOTSUPP

path 或 fd 所在的文件系统不支持扩展属性。

ERANGE

size 非 0，而缓冲区不够大以存储完整的 key 列表。应用可以设置 size 为 0 重新发起调用来获得列表的实际大小。程序然后可以调整缓冲区的大小并再次发起系统调用。

移除一个扩展属性。最后三个系统调用从指定文件移除指定的 key：

```
#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);
```

成功调用 removexattr()从文件 path 中移除扩展属性 key。回忆下我们前面讲过的，未定义 key，和已定义 key 但 value 为空是有区别的。

lremovexattr()的行为和 removexattr()相同，除了当 path 是符号链接时，它移除符号链接本身的扩展属性，而不是链接指向的目标文件。由于 user 命名空间不能对符号链接应用扩展属性，因此这个调用也很少使用。

fremovexattr()对文件描述符 fd 进行操作；除了这一点，它的行为和 removexattr()相同。

成功时三个系统调用都返回 0；失败时返回-1，并设置 errno 为以下值之一：

EACCESS

调用进程缺乏查找 path 路径中的目录的权限。

EBADF

fd 非法。

EFAULT

path 或 key 是无效指针。

ELOOP

path 包含过多的符号链接。

ENAMETOOLONG

path 过长。

ENOATTR

指定文件中 key 并不存在。

ENOENT

path 的某个组成部分不存在。

ENOMEM

完成请求需要的可用内存不足。

ENOTDIR

path 的某个组成部分不是目录。

ENOTSUPP

path 或 fd 所在的文件系统不支持扩展属性。

目录

在 Unix 中，目录是一个简单的概念：它包含一个文件名列表，每个都映射到一个 inode 数值。每个文件名称为目录项，每个名字－inode 映射称为一个链接。目录的内容——用户通过 ls 看到的结果——是那个目录中的所有文件名列表。当用户在指定目录中打开一个文件时，内核在目录的列表中查找文件名，以及相应的 inode 数值。内核然后传递 inode 数值给文件系统，文件系统使用 inode 值找到设备中文件的物理位置。

目录也可以包含其它目录。子目录就是在包含在另一个目录中的目录。有了这个定义，于是所有目录都是某个父目录的子目录，除了文件系统的 root 目录/。这个目录被称为根(root)目录（不要和 root 用户的 home 目录/root 混淆）。

路径名包含文件名以及一个或多个父目录。绝对路径就是以根目录开始的路径——例如/usr/bin/sextant。相对路径则是不以根目录开始的路径，例如 bin/sextant。要使相对路径有效，操作系统必须知道路径相对的那个目录。当前工作目录（下一节讨论）将被用作起始目录。

文件和目录名可以包含任意字符，除了/(它在路径中表示目录)，和 null 字符(它终止整个路径)。在路径名中使用当前 locale 下的可打印字符，或者只使用 ASCII 字符，是标准实践。不过由于内核和 C 库都不强制这个实践，只能由应用来强制只使用合法的可打印字符。

老的 Unix 系统限制文件名最多 14 个字符。今天所有的现代 Unix 文件系统都至少允许 255 字节的文件名。Linux 下的许多文件系统甚至允许更长的文件名。

每个目录包含两个特殊目录：.和..（叫做 dot 和 dot-dot）。dot 目录是对当前目录本身的引用。dot-dot 目录则是对父目录的引用。例如/home/kidd/gold/..和/home/kidd 目录相同。根目录的 dot 和 dot-dot 目录指向自己——也就是说，/、/.和/..都是同一个目录。因此从技术上讲，你可以认为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工作目录，它最初从父进程继承而来。这个目录就被称为进程的当前工作目录(cwd)。当前工作目录是内核查找相对路径名的起始位置。例如，如果进程的当前工作目录是/home/blackbeard，这时进程试图打开 parrot.jpg 文件，内核就会去打开/home/blackbeard/parrot.jpg 文件。反过来，如果进程试图打开/usr/bin/mast 文件，内核会直接打开/usr/bin/mast——当前工作目录不影响绝对路径（也就是以/开始的路径）。

进程可以获得和改变自己的当前工作目录。

获得当前工作目录

获得当前工作目录的首选方法是 getcwd()系统调用，POSIX 对它进行了标准化：

```
#include <unistd.h>

char * getcwd (char *buf, size_t size);
```

成功调用 getcwd()复制当前工作目录（绝对路径）到 buf 指向的缓冲区，并返回指向 buf 的指针，buf 的长度为 size 字节；失败时调用返回 NULL，并设置 errno 为以下值之一：

EFAULT

buf 是无效指针。

EINVAL

size 为 0，但 buf 不是 NULL。

ENOENT

当前工作目录不再有效。在当前工作目录被删除时可能发生。

ERANGE

size 太小，buf 不能容纳当前工作目录。应用需要分配一个更大的缓冲区并重试。

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);
```

POSIX 规定当 buf 为 NULL 时 getcwd()的行为未定义。Linux 的 C 库在这种情况下，会分配一个 size 大小的缓冲区，并存储当前工作目录到该缓冲区中。如果 size 为 0，C 库会分配一个足够存储当前工作目录的缓冲区。然后由应用在使用完之后，通过 free()释放这个缓冲区。由于这个行为是 Linux 特定的，可移植或者严格遵循 POSIX 的应用不应该依赖于这个功能。不过这个特性确实使得使用 getcwd()非常简单！下面是例子：

```
char *cwd;

cwd = getcwd (NULL, 0);
if (!cwd) {
    perror ("getcwd");
    exit (EXIT_FAILURE);
}
printf ("cwd = %s\n", cwd);

free (cwd);
```

Linux 的 C 库同时还提供一个 get_current_dir_name()函数，它和 getcwd()的 buf 为 NULL，size 为 0 时的行为完全一样：

```
#define _GNU_SOURCE
#include <unistd.h>

char * get_current_dir_name (void);
```

因此，下面代码片断和前面代码功能一样：

```
char *cwd;

cwd = get_current_dir_name ( );
if (!cwd) {
      perror ("get_current_dir_name");
      exit (EXIT_FAILURE);
}
printf ("cwd = %s\n", cwd);

free (cwd);
```

老的 BSD 系统也使用 getwd()调用，Linux 为了向后兼容性也提供它：

```
#define _XOPEN_SOURCE_EXTENDED /* or _BSD_SOURCE */
#include <unistd.h>

char * getwd (char *buf);
```

调用 getwd()复制当前工作目录到 buf 中，它至少是 PATH_MAX 字节。调用成功时返回 buf；失败时返回 NULL。例如：

```
char cwd[PATH_MAX];

if (!getwd (cwd)) {
      perror ("getwd");
      exit (EXIT_FAILURE);
}
printf ("cwd = %s\n", cwd);
```

由于可移植和安全的原因，应用不应该使用 getwd()，getcwd()是首选。

改变当前工作目录

当用户第一次登录到系统时，登录进程设置用户的当前工作目录为该用户的 home 目录，用户的 home 目录在/etc/passwd 文件中指定。不过有时候进程可能想修改自己的当前工作目录。例如，Shell 在用户输入 cd 命令时就需要这样做。

Linux 提供两个系统调用来改变当前工作目录，一个接受目录的路径名作为参数，另一个则接受已打开目录的文件描述符作为参数：

```
#include <unistd.h>

int chdir (const char *path);
int fchdir (int fd);
```

调用 chdir()改变当前工作目录为 path 指定的路径，它可以是绝对路径或者相对路径。类似的，调用 fchdir()改变当前工作目录为文件描述符 fd 表示的目录，它必须是一个已打开的目录。成功时两个调用都返回 0；失败时返回-1。

失败时，chdir()设置 errno 为以下错误代码之一：

EACCESS

调用进程缺乏查找 path 的某个组成目录的权限。

EFAULT

path 不是有效的指针。

EIO

发生内部 I/O 错误。

ELOOP

内核在查找路径时遇到过多符号链接。

ENAMETOOLONG

path 过长。

ENOENT

path 指向的目录不存在。

ENOMEM

没有足够的可用内存来完成请求。

ENOTDIR

path 的一个或多个组成不是目录。

fchdir()设置 errno 为以下值之一：

EACCESS

调用进程缺乏查找 fd 表示的目录的权限。如果顶层目录可读，但不可执行，这时 open()会成功，但 fchdir()却会失败。

EBADF

fd 不是打开的文件描述符。

根据不同的文件系统，其它的错误值对这两个调用也是合法的。

这两个系统调用只影响当前运行的进程。Unix 中没有提供改变其它进程的当前工作目录的机制。因此，shell 中的 cd 命令不能是单独的进程（和其它大多数命令一样），简单的以命令行第一个参数执行 chdir()，然后退出。相反 cd 必须是一个特殊的内建命令，由 shell 本身调用 chdir()，这样才能改变自己的当前工作目录。

getcwd()最常见的用法是保存当前工作目录，这样进程在稍后就可以恢复它。例如：

```
char *swd;
int ret;

/* save the current working directory */
swd = getcwd (NULL, 0);
if (!swd) {
        perror ("getcwd");
        exit (EXIT_FAILURE);
}

/* change to a different directory */
ret = chdir (some_other_dir);
if (ret) {
```

```
        perror ("chdir");
        exit (EXIT_FAILURE);
}

/* do some other work in the new directory... */

/* return to the saved directory */
ret = chdir (swd);
if (ret) {
        perror ("chdir");
        exit (EXIT_FAILURE);
}
free (swd);
```

不过更好的办法是打开当前目录，稍后再 fchdir()。这种方式更加快速，因为内核不需要在内存中存储当前工作目录的路径名；它只存储 inode。因此，无论什么时候用户调用 getcwd()，内核都必须产生路径并遍历目录结构。反过来，打开当前工作目录的代价要小，因为内核已经有了 inode，而打开文件并不需要路径名。下面代码片断就是使用这个方法：

```
int swd_fd;

swd_fd = open (".", O_RDONLY);
if (swd_fd == -1) {
        perror ("open");
        exit (EXIT_FAILURE);
}

/* change to a different directory */
ret = chdir (some_other_dir);
if (ret) {
        perror ("chdir");
        exit (EXIT_FAILURE);
}

/* do some other work in the new directory... */

/* return to the saved directory */
ret = fchdir (swd_fd);
if (ret) {
        perror ("fchdir");
        exit (EXIT_FAILURE);
}

/* close the directory's fd */
ret = close (swd_fd);
```

```
if (ret) {
        perror ("close");
        exit (EXIT_FAILURE);
}
```

这也正是 shell 实现缓存上一个目录的方法（例如在 bash 中使用 cd -）。

有些进程并不关心自己的当前工作目录——例如 daemon，通常会通过 chdir("/")把当前工作目录设置为/。面向用户和他的数据的应用，例如文本处理，通常会设置当前工作目录为用户的 home 目录，或者一个特殊的文档目录。因为当前工作目录只与相对路径有关，当前工作目录最大的用处在于 Shell 中给用户调用命令。

创建目录

Linux 提供一个系统调用来创建新目录，它由 POSIX 标准化：

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

int mkdir (const char *path, mode_t mode);
```

成功调用 mkdir()会创建 path 目录，path 可以是相对或者绝对路径，mode 指定权限（由当前 umask 进行修改），函数成功时返回 0。

当前 umask 按通常方式修改 mode 参数，加上操作系统指定的 mode 位：在 Linux 中，新创建目录的权限位是(mode & ~umask & 01777)。换句话说，umask 的作用是对进程强加 mkdir()不能超越的限制。如果新创建目录的父目录设置了组 ID 位(sgid)，或者文件系统以 BSD 组挂载，则新创建的目录将从父目录继承组(group)关系。否则，进程的有效组 ID 将应用到新目录上。

失败时，mkdir()返回-1，并设置 errno 为以下值之一：

EACCESS

父目录对当前进程不可写，或者 path 的某个组成不可查找。

EEXIST

path 已经存在（不一定是目录）。

EFAULT

path 是无效指针。

ELOOP

内核在解析 path 时遇到过多的符号链接。

ENAMETOOLONG

path 太长。

ENOENT

path 的某个组成不存在或者是无效符号链接。

ENOMEM

完成请求需要的可用内存不足。

ENOSPC

包含 path 的设备没有空间，或者用户的磁盘限额超过限制。

ENOTDIR

path 的一个或多个组成部分不是目录。

EPERM

包含 path 的文件系统不支持目录创建。

EROFS

包含 path 的文件系统以只读方式挂载。

删除目录

作为 mkdir()的配对，POSIX 标准化了 rmdir()，用来从文件系统永久删除一个目录：

```
#include <unistd.h>

int rmdir (const char *path);
```

成功时 rmdir()从文件系统删除 path，并返回 0。path 指定的目录必须为空（它只能包含 dot 和 dot-dot 目录）。没有系统调用实现 rm –r 这样的递归删除。类似的工具必须手动对文件系统执行深度优先遍历，从树的叶子开始删除所有文件和目录。rmdir()可以在删除所有文件之后删除该目录。

失败时，rmdir()返回-1，并设置 errno 为以下值之一：

EACCESS

path 的父目录不允许写入，或者 path 的某个组成部分不能被查找。

EBUSY

path 正在被系统使用，不能被删除。在 Linux 中，只有 path 是挂载点或者 root 目录时才会发生(root 目录不一定是挂载点，感谢 chroot())。

EFAULT

path 不是有效指针。

EINVAL

path 的最后一个组成部分是 dot。

ELOOP

内核在解析 path 时遇到太多的符号链接。

ENAMETOOLONG

path 过长。

ENOENT

path 的某个组成部分不存在，或者是摇摆符号链接。

ENOMEM

完成请求需要的可用内存不足。

ENOTDIR

path 的某个组成部分不是目录。

ENOTEMPTY

path 包含 dot 和 dot-dot 目录以外的项（目录非空）。

EPERM

path 的父目录设置了 sticky 位(S_ISVTX)，但进程的有效用户 ID 不是既用户 ID 也不是 path 的 ID，并且进程没有 CAP_FOWNER 能力。或者包含 path 的文件系统不允许删除目录。

EROFS

包含 path 的文件系统以只读方式挂载。

使用非常简单：

```
int ret;

/* remove the directory /home/barbary/maps */
ret = rmdir ("/home/barbary/maps");
if (ret)
    perror ("rmdir");
```

读取目录的内容

POSIX 定义了一组函数来读取目录的内容——也就是获得指定目录下所有文件的列表。这些函数在以下情况中很有用：实现 ls 命令或者图形文件保存对话框、操作指定目录下的每一个文件、查找目录下匹配指定模式的文件等等。

要读取目录的内容，你需要创建一个目录流，由 DIR 对象描述：

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

DIR * opendir (const char *name);
```

成功调用 opendir()创建 name 指定的目录流。

目录流实际上就是一个描述已打开目录的文件描述符、一些元数据、以及一个保存目录内容的缓冲区。因此，可以通过指定的目录流获得文件描述符：

```
#define _BSD_SOURCE /* or _SVID_SOURCE */
#include <sys/types.h>
#include <dirent.h>

int dirfd (DIR *dir);
```

成功调用 dirfd()返回目录流 dir 的文件描述符；出错时调用返回-1。由于目录流函数内部也使用这个文件描述符，我们的程序应该只调用那些不操作文件位置的函数。dirfd()是 BSD 扩展，并不是 POSIX 标准；宣称遵循 POSIX 标准的程序员应该避免使用 dirfd()。

读取目录流

一旦你使用 opendir()创建了目录流，你就可以开始读取目录的项。使用 readdir()，从指定的 DIR 对象中一个一个地返回目录项：

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

struct dirent * readdir (DIR *dir);
```

成功调用 readdir()返回 dir 表示的目录的下一项。dirent 结构体描述一个目录项，在<dirent.h>文件中定义。在 Linux 中它的定义是：

```
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 只要求 d_name 域，也就是目录下单个文件的文件名。其它域要么是可选的，要么是 Linux 特定的。希望移植到其它系统，或者遵循 POSIX 标准的应用应该只访问 d_name。

应用连续地调用 readdir()，获得目录下的每个项，直到找到想要的文件。或者直到整个目录流读取结束，这时候 readdir()返回 NULL。

失败时，readdir()同样返回 NULL。要区分出错和读取所有文件完成，应用必须在每次调用 readdir()之前设置 errno 为 0，然后在调用后检查返回值和 errno。readdir()唯一设置的 errno 值是 EBADF，表示 dir 无效。因此，许多应用根本不检查错误，而假设 NULL 表示所有文件已经读取完成。

关闭目录流

要关闭 opendir()打开的目录流，需要使用 closedir()：

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

int closedir (DIR *dir);
```

成功调用 closedir()关闭 dir 表示的目录流，包括 dir 背后的文件描述符，并返回 0；失败时函数返回-1，并设置 errno 为 EBADF，这是唯一的错误代码，表示 dir 不是已打开的目录流。

下面代码片断实现 find_file_in_dir()函数，使用 readdir()查找指定目录下的指定文件。如果文件存在于指定目录，函数返回 0；否则函数返回一个非 0 值：

```
/*
* find_file_in_dir - searches the directory 'path' for a
* file named 'file'.
*
* Returns 0 if 'file' exists in 'path' and a nonzero
* value otherwise.
*/
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;
}
```

读取目录内容使用的系统调用

前面讨论的读取目录内容的函数，由 POSIX 标准定义，并由 C 库提供实现。在内部，这些函数使用一个或者两个系统调用：readdir()和 getdents()，为了讨论的完整性，这里也顺带说明一下：

```
#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);
```

你不会想使用这些系统调用！它们笨重而且不可移植。实际上，用户空间应用应该使用 C 库提供的 opendir()、readdir()、和 closedir()系统调用。

链接

回忆我们对目录的讨论：目录下的每个“名字－inode”映射称为一个链接。有了这个简单的定义——链接本质上就是目录中指向一个 inode 的名字——看起来不需要任何理由解释为什么多个链接可以指向同一个 inode。例如，一个单独的 inode（也就是一个文件）可以被/etc/customs 和/var/run/ledger 同时引用。

实际的情况就是这样，不过有一个例外：由于链接映射到 inode，而 inode 数值特定于不同的文件系统，因此/etc/customs 和/var/run/ledger 必须存在于同一个文件系统中。在一个单独的文件系统中，可以有许多链接指向任意一个文件。唯一的限制是用来存储链接数量的整数数据类型的大小。在一个文件的所有链接中，没有哪个链接是“原始”或者“主要”链接。所有链接都享有相同状态，指向同一个文件。

我们把这种链接称为硬链接。文件可以有零个、一个、或者多个链接。大多数文件的链接计数都是 1——也就是它被一个单独的目录项引用——但是另一些文件则有两个或者多个链接。链接计数为 0 的文件在文件系统中没有对应的目录项。当文件的链接计数到达 0 时，文件就被标记为空闲(free)，它的磁盘块可以被重新使用。不过如果某个进程打开了这个文件，则文件会被保留在文件系统中。一旦没有任何进程打开这个文件，文件将最终被删除。

Linux 内核采用一个链接计数和一个使用计数来实现这个行为。使用计数是文件被打开的实例总数。文件在链接计数和使用计数都到达 0 之前，不会从文件系统中删除。

另一种类型的链接是符号链接，它不是文件系统的映射，而是更高层的指针抽象，并且在运行时解析。这样的链接可以跨越文件系统——我们很快会讨论它。

硬链接

link()是最早的 Unix 系统调用之一，现在由 POSIX 标准化，用来给已存在的文件创建一个新的链接：

```
#include <unistd.h>

int link (const char *oldpath, const char *newpath);
```

成功调用 link()为已存在的 oldpath 文件，在路径 newpath 下创建一个新的链接，然后返回 0。操作完成后，oldpath 和 newpath 都引用同一个文件——这时候实际上已经没有办法区分哪个是“原始”链接了。

失败时调用返回-1，并设置 errno 为以下值之一：

EACCESS

调用进程缺乏查找 oldpath 路径的权限，或者调用进程对包含 newpath 的目录没有写入权限。

EEXIST

newpath 已经存在——link()不会覆盖已经存在的目录项。

EFAULT

oldpath 或者 newpath 是无效指针。

EIO

发生内部 I/O 错误（这很糟糕）。

ELOOP

内核在解析 oldpath 或者 newpath 时遇到太多的符号链接。

EMLINK

oldpath 指向的 inode 已经拥有最大允许数量的链接。

ENAMETOOLONG

oldpath 或者 newpath 过长。

ENOENT

oldpath 或者 newpath 的某个组成部分不存在。

ENOMEM

完成请求需要的可用内存不足。

ENOSPC

包含 newpath 的设备没有足够的空间来存储目录项。

ENOTDIR

oldpath 或者 newpath 的某个组成部分不是目录。

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 存在于只读文件系统中。

EXDEV

newpath 和 oldpath 不在同一个文件系统中。（Linux 允许一个单独的文件系统挂载在多个位置，即使是这种情况下，硬链接也不能跨挂载点创建）。

下面例子创建一个新的目录项 pirate，映射到已经存在的文件 privateer 的 inode 上，两个链接都在/home/kidd 下：

```
int ret;

/*
* create a new directory entry,
* '/home/kidd/privateer', that points at
* the same inode as '/home/kidd/pirate'
*/
ret = link ("/home/kidd/privateer", /home/kidd/pirate");
if (ret)
      perror ("link");
```

符号链接

符号链接又被称为 symlinks 或者软链接，在多个链接指向同一个文件上和硬链接类似。符号链接与硬链接的区别在于它不仅仅是一个额外的目录项，符号链接本身是一种特殊类型的文件。这个特殊的文件包含另一个文件的路径，称为符号链接的目标。在运行时，内核使用目标的路径替换符号链接的路径（除非使用系统调用的 l 版本，例如 lstat()，对链接文件本身而不是目标文件进行操作）。因此，虽然无法区分相同文件的不同硬链接，要区分符号链接和目标文件却是非常容易的。

符号链接可以是相对或者绝对的，它也可以包含前面讨论过的特殊的 dot 目录，引用链

接所在的那个目录；或者 dot-dot 目录，这时候链接引用父目录。

软链接不像硬链接，它可以跨文件系统。实际上它可以指向任何地方！符号链接可以指向已经存在的文件（通常的情况），甚至是不存在的文件。后一种类型的符号链接被称为摇摆符号链接(dangling symlink)。有时候，摇摆符号链接是有害的——当目标文件被删除，而符号链接文件还没有删除——但是也有时候摇摆符号链接是故意制造的。

创建符号链接的系统调用和硬链接非常相似：

```
#include <unistd.h>

int symlink (const char *oldpath, const char *newpath);
```

成功调用 symlink()创建符号链接 newpath，指向目标文件 oldpath，然后返回 0。

出错时 symlink()返回-1，并设置 errno 为以下值之一：

EACCESS

调用进程缺乏查找 oldpath 路径的权限，或者调用进程对包含 newpath 的目录没有写入权限。

EEXIST

newpath 已经存在——symlink()不会覆盖已经存在的目录项。

EFAULT

oldpath 或者 newpath 是无效指针。

EIO

发生内部 I/O 错误（这很糟糕）。

ELOOP

内核在解析 oldpath 或者 newpath 时遇到太多的符号链接。

EMLINK

oldpath 指向的 inode 已经拥有最大允许数量的链接。

ENAMETOOLONG

oldpath 或者 newpath 过长。

ENOENT

oldpath 或者 newpath 的某个组成部分不存在。

ENOMEM

完成请求需要的可用内存不足。

ENOSPC

包含 newpath 的设备没有足够的空间来存储目录项。

ENOTDIR

oldpath 或者 newpath 的某个组成部分不是目录。

EPERM

包含 newpath 的文件系统不允许创建新的符号链接。

EROFS

newpath 存在于只读文件系统中。

下面代码片断和之前的例子一样，不过创建/home/kidd/pirate 为/home/kidd/privateer 的符号链接：

```
int ret;
/*
```

```
* create a symbolic link,
* '/home/kidd/privateer', that
* points at '/home/kidd/pirate'
*/
ret = symlink ("/home/kidd/privateer", "/home/kidd/pirate");
if (ret)
        perror ("symlink");
```

Unlinking

linking 的反面是 unlinking，从文件系统中删除 pathname。使用 unlink()系统调用完成这个任务：

```
#include <unistd.h>

int unlink (const char *pathname);
```

成功调用 unlink()从文件系统中删除 pathname，并返回 0。如果这是文件的最后一个引用，文件将从文件系统中彻底删除。不过如果有进程打开了这个文件，内核不会从文件系统中删除文件，直到进程关闭该文件。一旦没有进程打开这个文件，它才被删除。

如果 pathname 是符号链接，链接本身而不是目标文件将被销毁。

如果 pathname 指向另一种特殊类型的文件，例如设备、FIFO、或者 socket，特殊文件将从文件系统中删除，但打开该文件的进程可以继续使用它。

出错时 unlink()返回-1，并设置 errno 为以下错误代码之一：

EACCESS

调用进程没有 pathname 的父目录的写权限，或者调用进程没有查找路径 pathname 的权限。

EFAULT

pathname 是无效指针。

EIO

发生内部 I/O 错误（很糟糕）。

EISDIR

pathname 指向一个目录。

ELOOP

内核在穿越 pathname 时遇到过多的符号链接。

ENAMETOOLONG

pathname 过长。

ENOENT

pathname 的某个组成部分不存在。

ENOMEM

完成请求需要的可用内存不足。

ENOTDIR

pathname 的某个组成部分不是目录。

EPERM

系统不允许 unlinking 文件。

EROFS

pathname 所在的文件系统只读。

unlink()不能删除目录，应用需要使用 rmdir()来删除目录，前面“删除目录”一节中我们已经讨论过了。

为了删除不同类型的文件，C 语言提供了 remove()函数：

```
#include <stdio.h>

int remove (const char *path);
```

成功调用 remove()从文件系统中删除 path，并返回 0。如果 path 是一个文件，remove()调用 unlink()；如果 path 是一个目录，remove()将调用 rmdir()。

出错时，remove()返回-1，并设置 errno 为 unlink()和 rmdir()可能产生的任何错误代码。

复制和移动文件

两个最基本的文件操作就是复制和移动文件，通常通过 cp 和 mv 命令来完成。在文件系统层次，复制是在一个新路径重复指定文件内容的一个操作。这和创建一个文件的新硬链接不同，修改其中一个文件不会影响到另一个文件——也就是，在两个不同的目录项下存在两个不同的文件拷贝。反过来，移动文件则是重命名文件所在的目录项。这个动作不会导致文件的第二份拷贝产生。

复制

尽管很令人吃惊，Unix 并没有包含一个系统或者库调用来完成复制文件和目录。相反，类似于 cp 或 GNOME 的 Nautilus 管理器都手动执行复制任务。

复制 src 文件为 dst 文件，步骤如下：

1. 打开 src
2. 打开 dst，如果不存在则创建，如果存在则把它的大小截断为 0
3. 从 src 中读取一块数据到内存中
4. 把这块数据写入到 dst 中
5. 继续 34 直到 src 完全读取并写入到 dst 中
6. 关闭 dst
7. 关闭 src

如果复制一个目录，则目标目录和它的任何子目录都通过 mkdir()创建；然后单个地复制目录下的每个文件。

移动

和复制不一样，Unix 提供了一个系统调用来移动文件。ANSI C 标准定义这个调用来移动文件，POSIX 则标准化它可以移动文件和目录：

```
#include <stdio.h>

int rename (const char *oldpath, const char *newpath);
```

成功调用 rename()重命名路径 oldpath 为 newpath，文件的内容和 inode 保持不变。oldpath 和 newpath 必须在同一个文件系统中；如果不在同一个文件系统中，调用将失败。类似 mv 这样的工具必须处理这种情况，通过复制和 unlink。

成功时 rename()返回 0，文件以前由 oldpath 引用，现在由 newpath 引用；失败时调用返回-1，不会动 oldpath 和 newpath，并且设置 errno 为以下值之一：

EACCESS

调用进程缺乏 oldpath 或 newpath 父目录的写权限、缺乏 oldpath 或 newpath 的查找权限、oldpath 是目录时缺乏 oldpath 的写权限。最后一种情况之所以是问题，是因为 oldpath 是目录时 rename()必须更新..目录。

EBUSY

oldpath 或 newpath 是挂载点。

EFAULT

oldpath 或 newpath 是无效指针。

EINVAL

newpath 包含在 oldpath 中，因此，重命名 oldpath 将使 oldpath 成为自己的子目录。

EISDIR

newpath 已经存在，并且是一个目录，但 oldpath 不是目录。

ELOOP

解析 oldpath 或 newpath 时，遇到过多的符号链接。

EMLINK

oldpath 已经拥有最大数量的链接、或者 oldpath 是一个目录，而 newpath 已经拥有最大数量的链接。

ENAMETOOLONG

oldpath 或 newpath 过长。

ENOENT

oldpath 或 newpath 的组成部分不存在，或者是摇摆符号链接。

ENOMEM

完成请求需要的可用内存不足。

ENOSPC

设备没有足够的空间来完成请求。

ENOTDIR

oldpath 或 newpath 的某个组成部分不是目录（除了最后一个部分），或者 oldpath 是一个目录，而 newpath 已经存在又不是一个目录。

ENOTEMPTY

newpath 是一个目录，并且非空。

EPERM

参数指定的路径中至少一个已存在，父目录设置了 sticky 位，调用进程的有效用户 ID 既不是文件的用户 ID，也不是父进程的 ID，并且进程没有特权。

EROFS

文件系统标记为只读。

EXDEV

oldpath 和 newpath 不在同一个文件系统中。

表 7-1 回顾了在不同文件类型之间移动的结果

表 7-1. 不同类型文件移动的效果

	目标是一个文件	目标是一个目录	目标是一个链接	目标不存在
源是一个文件	目标被源文件覆盖	以 EISDIR 失败	文件被重命名，目标被覆盖	文件被重命名
源是一个目录	以 ENOTDIR 失败	如果目标为空，源重命名为目标；否则以 ENOTEMPTY 失败	目录被重命名，目标将被覆盖	目录被重命名
源是一个链接	链接被重命名，目标被覆盖	以 EISDIR 失败	链接被重命名，目标被覆盖	链接被重命名
源不存在	以 ENOENT 失败	以 ENOENT 失败	以 ENOENT 失败	以 ENOENT 失败

对上面所有情况，无论文件的类型，只要源和目标不在同一个文件系统中，调用都将失败并返回 EXDEV。

设备节点

设备节点是允许应用与设备驱动交互的特殊文件。当应用对设备节点执行普通 Unix I/O 时——打开、关闭、读取、写入等等，内核不把这些处理当作普通文件 I/O 处理。相反，内核把请求传递给设备驱动。设备驱动处理这些 I/O 操作，并返回结果给用户。设备节点提供设备抽象，这样应用就不需要关注特定的设备，以及设备的特殊接口。实际上，设备节点是 Unix 系统中访问硬件的标准机制。网络设备是一个罕见的例外，超出了 Unix 的历史，有些人认为这个例外是一个错误。原本可以使用优雅的方式统一操作机器的所有硬件，使用标准的 read()、write()和 mmap()调用。

内核是如何识别设备驱动，决定由谁来处理请求的呢？每个设备节点都分配两个数值，一个称为主数值(major number)、另一个称为副数值(minor number)。主和副数值映射到内核装载的一个特定的设备驱动。如果设备节点的主和副数值在内核中没有相应的设备驱动（由于各种原因，这偶尔也会发生），对设备节点的 open()请求将返回-1，并设置 errno 为 ENODEV。这样的设备节点指向不存在的设备。

特殊设备节点

所有 Linux 系统都有特殊设备节点。这些设备节点是 Linux 开发环境的一部分，并且它们也是 Linux ABI 的重要组成部分。

null 设备的主数值为 1，副数值为 3，它存在于/dev/null。这个设备文件的拥有者是 root，可以被所有用户读取和写入。内核默默地抛弃对该设备的所有写入请求，对该文件的所有读取请求都返回 end-of-file(EOF)。

zero 设备存在于/dev/zero，主数值为 1，副数值为 5。和 null 设备类似，内核默默地抛弃对 zero 设备的所有写入请求。从设备读取返回的是 null 字节的无穷流。

full 设备的主数值为 1，副数值为 7，存在于 dev/full。和 zero 设备一样，读取请求返回 null 字符(\0)。然而写入请求，总是触发 ENOSPC 错误，表示这个设备已满。

这些设备都有各种用途。对于测试应用怎样处理错误情况和各种问题非常有用——例如一个满的文件系统。由于 null 和 zero 设备忽略写入请求，它们也提供一个无开销的方式以抛弃不希望的 I/O 请求。

随机数生成器

内核的随机数生成器设备存在于/dev/random 和/dev/urandom。它们的主数值为 1，副数值分别是 8 和 9。

内核的随机数生成器从设备驱动和其它来源收集噪音(noise)，内核把所有 noise 连接在一起，并对连接后的 noise 使用单向的 hash。然后把结果存储在 entropy 池。内核对池中的 entropy 保持一个位数的估计。

从/dev/random 中读取将返回池中的 entropy。结果非常适合用来作为随机数生成器的种子、执行键生成、以及其它需要密码强度的 entropy 的任务。

理论上，如果能够从 entropy 池获得足够的数据，并成功破解单向 hash，也就可以获得 entropy 池的其余状态信息。尽管类似的攻击在目前仍然只有理论上的可能性——目前尚未出现成功破解——内核对每次读取请求负责消耗 entropy 池的数量。如果数量到达 0，读取将阻塞直到系统生成更多的 entropy，一般来说 entropy 的数量足以满足读取请求。

/dev/urandom 没有这个特性；即使内核的 entropy 数量不够完成请求，从这个设备读取也会成功。因为只有最安全的应用——例如为 GNU Privacy Guard 安全数据交换生成 key——才会关心密码高强的 entropy，大多数应用应该使用/dev/urandom 而不是/dev/random。如果没有 I/O 动作发生来给内核 entropy 池生成种子，从后者读取可能会阻塞很长一段时间。在无磁盘 headless 的服务器上并不少见。

Out-of-Band 通讯

Unix 的文件模型确实令人印象深刻。仅仅使用简单的 read 和 write 操作，Unix 抽象了你可能对一个对象可能执行的任何动作。然而有时候程序员与文件通讯时，可能超出了主要数据流。例如，考虑一下串口设备。从设备读取需要从串口远端的硬件读取；写入到设备也需要发送数据到硬件。进程怎样读取串口的特殊状态引脚信息，例如 data terminal ready(DTR)信号呢？或者进程又怎样设置串口的奇偶校验呢？

答案是使用 ioctl()系统调用。ioctl()代表 I/O 控制，允许 out-of-band 通讯。

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

int ioctl (int fd, int request, ...);
```

这个系统调用需要两个参数：

fd

文件的文件描述符。

request

一个特殊的请求代码，由内核和进程预先定义并达成一致，指定要对 fd 表示的文件执行的操作。

调用也可以接受一个或多个可选参数（通常是 unsigned 整数或指针）传递给内核。

下面程序使用 CDROMEJECT 请求来弹出 CD-ROM 设备的媒体盘，由用户运行程序的命令行提供第一个参数。这个程序在功能上类似于标准的 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;
    }
    /*
    * Opens the CD-ROM device, read-only. O_NONBLOCK
    * tells the kernel that we want to open the device
    * even if there is no media present in the drive.
    */
    fd = open (argv[1], O_RDONLY | O_NONBLOCK);
    if (fd < 0) {
        perror ("open");
        return 1;
    }
    /* Send the eject command to the CD-ROM device. */
    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 CD-ROM 设备驱动的一个特性。当内核接到 ioctl()请求时，内核会根据提供的文件描述符查找文件系统（实际文件）或者设备驱动（设备节点），并传递请求进行相应的处理。在上面的程序中，CD-ROM 设备驱动接收请求，并弹出设备。

本章的后面我们会查看一个 ioctl()例子，使用可选参数返回信息给请求进程。

监视文件事件

Linux 提供 inotify 接口，用来监控文件——例如查看文件何时被移动、读取、写入、或者删除。想像你正在编写一个图形界面的文件管理器，例如 GNOME 的 Nautilus。如果文件被复制到一个目录，而 Nautilus 正在显示它的内容，文件管理器的目录视图将变得不一致。

一个解决办法是持续地读取目录的内容，检测变化并更新显示。这带来了周期性的开销，也远远不是一个优雅的解决方案。更坏的是，当文件被删除或者添加到目录时，文件管理器重新读取目录内容总是存在竞争条件。

使用 inotify，内核可以在发生变化时主动地把事件通知给应用。在文件被删除的同时，内核可以通知 Nautilus。然后 Nautilus 就可以立即从目录的图形显示中去除已经删除的文件。

许多其它的应用同样关注文件事件。考虑备份工具或者数据索引工具。inotify 允许这些程序以实时方式操作：在文件创建、删除、写入的同时，这些工具可以更新备份档案或者数据索引。

inotify 替换了之前的 dnotify，dnotify 是早期的文件监控机制，基于笨重的信号接口。应用总是应该优先使用 inotify 而不使用 dnotify。inotify 在内核 2.6.13 中引入，非常灵活而且使用简单，因为程序使用 inotify 时和普通文件使用相同的操作——特别是 select()和 poll()。我们在本书中只讲解 inotify。

初始化 inotify

在进程可以使用 inotify 之前，必须先初始化它。inotify_init()系统调用初始化 inotify 并返回一个文件描述符，描述已初始化的 inotify 实例：

```
#include <inotify.h>

int inotify_init (void);
```

出错时 inotify_init()返回-1，并设置 errno 为以下值之一：

EMFILE

用户的最大 inotify 实例数量限制已经达到。

ENFILE

系统的最大文件描述符数量限制已经达到。

ENOMEM

完成请求需要的可用内存不足。

让我们来初始化 inotify，以便接下来的步骤使用：

```
int fd;

fd = inotify_init ( );
if (fd == -1) {
        perror ("inotify_init");
        exit (EXIT_FAILURE);
}
```

监视

进程初始化 inotify 之后，就需要设置监视(watches)。监视由 watch 描述符表示，是标准的 Unix 路径，和一个相关联的监视掩码，它告诉内核进程感兴趣的事件——例如读取、写入、或者两个一起。

inotify 可以监视文件和目录。如果监视一个目录，inotify 报告目录本身发生的事件，同时也报告目录下的所有文件的事件（但不包括监视目录的子目录下的文件——监视不是递归的）。

添加一个新的监视

系统调用 inotify_add_watch()向 fd 表示的 inotify 实例添加一个监视，mask 表示要对文件或目录 path 监视的事件：

```
#include <inotify.h>

int inotify_add_watch (int fd,
                       const char *path,
                       uint32_t mask);
```

成功时调用返回一个新的监视描述符；失败时 inotify_add_watch()返回-1，并设置 errno 为以下值之一：

EACCESS

没有对 path 文件的读取访问权限。调用进程要对文件添加监视必须能够读取文件。

EBADF

文件描述符 fd 无效。

EFAULT

path 指针无效。

EINVAL

watch 掩码 mask 包含非法的事件。

ENOMEM

完成请求需要的可用内存不足。

ENOSPC

用户的 inotify 监视最大数量限制已经达到。

监视掩码

监视掩码是一个或多个 inotify 事件的二进制或(OR)，事件在<inotify.h>中定义：

IN_ACCESS

文件被读取。

IN_MODIFY

文件被写入。

IN_ATTRIB

文件的元数据（例如拥有者、权限、或者扩展属性）改变。

IN_CLOSE_WRITE

文件关闭，并且之前以写入方式打开。

IN_CLOSE_NOWRITE

文件关闭，并且之前没有以写入方式打开。

IN_OPEN

文件被打开。

IN_MOVED_FROM

一个文件从被监视目录中移走。

IN_MOVED_TO

一个文件移动到被监视目录下。

IN_CREATE

一个文件在被监视目录下创建。

IN_DELETE

一个文件从被监视目录下删除。

IN_DELETE_SELF

被监视对象本身被删除。

IN_MOVE_SELF

被监视对象本身被移动。

以下事件组合一个或者多个事件到一起，同样也在<inotify.h>中定义：

IN_ALL_EVENTS

所有合法的事件。

IN_CLOSE

所有与关闭相关的事件（当前就是 IN_CLOSE_WRITE 和 IN_CLOSE_NOWRITE）。

IN_MOVE

所有与移动相关的事件（当前就是 IN_MOVED_FROM 和 IN_MOVED_TO）。

现在我们来看一下怎样向 inotify 实例中添加监视：

```
int wd;

wd = inotify_add_watch (fd, "/etc", IN_ACCESS | IN_MODIFY);
if (wd == -1) {
    perror ("inotify_add_watch");
    exit (EXIT_FAILURE);
}
```

上面例子对目录/etc 添加了所有读取和写入监视。如果/etc 下的任何文件被写入或者读取，inotify 将发送一个事件给 inotify 文件描述符 fd，并提供监视描述符 wd。我们先来看看 inotify 是如何描述这些事件的。

inotify 事件

inotify_event 结构体在<inotify.h>中定义，描述了 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 表示监视描述符，和 inotify_add_watch()获得的一样，mask 描述事件。如果 wd 表示一个目录，并且该目录下的某个文件发生了被监视的事件，则 name 就是该文件的文件名（相对）。在这种情况下，len 非 0。注意 len 和字符串 name 的长度并不一样；name 可能包含一个结束的 null 字节作为填充，以确保后续的 inotify_event 正确对齐。因此，你必须使用 len 而不是 strlen()，来计算 inotify_event 结构体数组中下一个对象的偏移量。

例如，如果 wd 表示/home/rlove，它的掩码是 IN_ACCESS，并且文件/home/rlove/canon 被读取，这时候 name 就等于 canon，并且 len 至少会是 6。相反，如果我们用相同的掩码直接监视/home/rlove/canon 文件，len 将为 0，并且 name 也会是 0 长度。

cookie 用来连接两个相关但分离的事件。我们会在随后的章节中讲解它。

读取 inotify 事件

获取 inotify 事件非常容易：你只需要从 inotify 实例相关联的文件描述符中读取即可。inotify 提供一个称为 slurping 的特性，允许你通过一个单独的 read 请求读取多个事件——多到恰好装进提供给 read()的缓冲区。由于 name 域是可变长度，这也是读取 inotify 事件最常见的方式。

我们之前的例子实例化了一个 inotify 实例，并且向它添加了一个监视。现在，让我们来读取未决的事件：

```
char buf[BUF_LEN]_attribute_((aligned(4)));
ssize_t len, i = 0;

/* read BUF_LEN bytes' worth of events */
len = read (fd, buf, BUF_LEN);

/* loop over every read event until none remain */
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");

        /* if there is a name, print it */
        if (event->len)
            printf ("name=%s\n", event->name);

        /* update the index to the start of the next event */
        i += sizeof (struct inotify_event) + event->len;
    }
```

由于 inotify 文件描述符和普通文件一样，程序可以通过 select()、poll()和 epoll()来监视它。这允许进程以一个单独的线程多路(multiplex)inotify 事件和其它文件 I/O。

高级 inotify 事件。除了标准事件，inotify 还可以产生其它事件：

IN_IGNORED

wd 描述的监视已经被移除。可能发生的原因是用户手动删除监视，或者被监视对象已经不存在。我们在随后的章节会讨论这个事件。

IN_ISDIR

受到影响的对象是一个目录。（如果没有设置，则受影响的对象是一个文件）

IN_Q_OVERFLOW

inotify 队列溢出。内核限制事件队列的大小，以防止大量消耗内核内存。一旦未决事件到达最大限制值减一，内核就会产生这个 IN_Q_OVERFLOW 事件，并将它添加到事件队列的末尾。除非队列被读取使大小低于限制，否则不会再有任何事件产生。

IN_UNMOUNT

被监视对象的设备被卸载(unmount)。因此对象不再可用；内核会移除监视，并产生 IN_IGNORED 事件。

任何监视都可能产生这些事件；用户不需要显式地设置它们。

程序员必须把 mask 当作未决事件的位掩码对待。因此，绝对不要使用直接相等(==)来检查事件：

```
/* Do NOT do this! */
if (event->mask == IN_MODIFY)
    printf ("File was written to!\n");
else if (event->mask == IN_Q_OVERFLOW)
    printf ("Oops, queue overflowed!\n);
```

正确的做法是使用位测试：

```
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");
```

连接移动事件到一起

IN_MOVED_FROM 和 IN_MOVED_TO 事件中的每一个只描述了移动的一半：前者表示从指定位置移出；而后者表示移入到新位置。因此，程序如果要有效地跟踪文件在文件系统中被移动（考虑文件索引应用不对已经移走的文件重新计算索引），进程就需要能够把两个移动事件连接到一起。

这就需要使用 inotify_event 结构体中的 cookie 域。

cookie 域如果非 0，包含唯一的值，把两个事件连接到一起。考虑某个进程监视/bin 和/sbin。假设/bin 的监视描述符为 7，并且/sbin 的监视描述符为 8。如果文件/bin/compass 被移动到/sbin/compass，内核将会产生两个 inotify 事件。

第一个事件的 wd 等于 7，mask 等于 IN_MOVED_FROM，name 为 compass。第二个事件的 wd 等于 8，mask 等于 IN_MOVED_TO，并且 name 也是 compass。在这两个事件中，cookie 是相同的——例如都为 12。

如果文件被重命名，内核仍然会产生两个事件，而且它们的 wd 也相同。

注意如果一个文件被移动到一个目录，或者从一个目录中移走，而这个目录并没有被监视，进程将不会收到其中的一个事件。程序必须自己注意第二个事件不会产生。

高级监视选项

当创建一个新的监视时，你可以向 mask 添加一个或多个选项，以控制监视的行为：

IN_DONT_FOLLOW

如果设置了这个值，并且 path 的目标，或者 path 的任何组成部分是符号链接，那么不会跟随链接，而且 inotify_add_watch()将失败。

IN_MASK_ADD

通常情况下，如果你对一个已经有监视的文件调用 inotify_add_watch()，监视掩码将被新提供的 mask 更新。如果设置了这个标志，则提供的事件将添加到已存在的掩码中。

IN_ONESHOT

如果设置了这个值，则内核会在对象产生第一个事件后自动删除该监视。因此实际上监视是“one shot”。

IN_ONLYDIR

如果设置了这个值，监视只有在提供的对象是一个目录时才会添加。如果 path 表示一个文件而不是目录，inotify_add_watch()将失败。

例如，下面代码片断只有在/etc/init.d 是一个目录时才添加监视，并且/etc 和/etc/init.d 都不能是符号链接：

```
int wd;
/*
* Watch '/etc/init.d' to see if it moves, but only if it is a
* directory and no part of its path is a symbolic link.
*/
```

```
wd = inotify_add_watch (fd,
                        "/etc/init.d",
                        IN_MOVE_SELF |
                        IN_ONLYDIR |
                        IN_DONT_FOLLOW);

if (wd == -1)
    perror ("inotify_add_watch");
```

删除 inotify 监视

你可以使用系统调用 inotify_rm_watch()从一个指定的 inotify 实例中删除一个监视：

```
#include <inotify.h>

int inotify_rm_watch (int fd, uint32_t wd);
```

成功调用 inotify_rm_watch()从 inotify 实例 fd 中删除监视描述符 wd 表示的监视，并返回 0。

例如：

```
int ret;

ret = inotify_rm_watch (fd, wd);
if (ret)
    perror ("inotify_rm_watch");
```

失败时调用返回-1，并设置 errno 为以下两个错误代码之一：

EBADF

fd 不是有效的 inotify 实例。

EINVAL

wd 不是指定 inotify 实例的合法的监视描述符。

当删除一个监视时，内核产生 IN_IGNORED 事件。内核不仅仅在手动删除监视时发送这个事件，在其它操作导致销毁监视时也会发送 IN_IGNORED 事件。例如，当一个监视文件被删除时，该文件的所有监视都会被自动删除，在这种情况下，内核也会发送 IN_IGNORED。这个行为允许应用在一个地方处理监视的删除：IN_IGNORED 的事件处理器。对于 inotify 的高级用户，和那些管理每个 inotify 监视的复杂数据结构的应用，例如 GNOME 的 Beagle 搜索工具非常有用。

获得事件队列的大小

未决事件队列的大小，可以通过对 inotify 实例的文件描述符执行 ioctl 的 FIONREAD 选项获得。请求的第一个参数接收队列的大小（字节），它是一个 unsigned 整数：

```
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);
```

注意请求按字节返回队列的大小，而不是队列中事件的个数。程序可以从字节数估计出事件的数目，使用 inotify_event 结构体的大小（通过 sizeof），和 name 域的平均大小。不过更有用的是，字节数是进程读取未决事件的理想大小。

头文件<sys/ioctl.h>定义了 FIONREAD 常量。

销毁 inotify 实例

销毁一个 inotify 实例，以及任何相关联的监视，只需要简单地关闭实例的文件描述符：

```
int ret;

/* 'fd' was obtained via inotify_init( ) */
ret = close (fd);
if (fd == -1)
    perror ("close");
```

当然，和任何文件描述符一样，内核在进程退出时，会自动关闭所有文件描述符，并清理所有相关的资源。

第八章 内存管理

内存是进程最基本也最重要的可用资源。本章讲解内存的管理：分配、操作、和最终释放内存。

术语分配(allocate)——获得内存的通常用语——有一点令人误解，因为它给人的图像是分配稀缺资源给超过提供的需要，确实许多用户喜欢更多的内存。然而在现代系统中，问题实际上并不是共享太少和太多内存，而在于正确的使用内存和保持有效使用。

在本章，你将学习在不同程序区域中分配内存的所有方法，包括每种方法的优点和缺点。我们也讲解设置和操作任意内存区域的方法，并查看如何锁定内存，使它保留在 RAM 中，使你的程序不用等待内核从交换空间中装载页面。

进程地址空间

Linux 和所有现代操作系统一样，虚拟了物理内存资源。进程不会直接寻址物理内存；相反内核关联每个进程一个唯一的虚拟地址空间。这个地址空间是线性的，地址从 0 开始，一直增长到某个最大值。

页面和分页

虚拟地址空间由页面组成。系统架构和机器类型决定页面的大小，页面的大小是固定的；典型的值包括 4KB(32 位系统)、和 8KB(64 位系统)。页面要么是有效的，要么是无效的。有效的页面与物理内存相关联，或者与某个二级存储相关联（例如交换分区或者磁盘中的文件）。无效的页面则没有与任何存储关联，并且表示一个未使用未分配的地址空间块。访问无效页面会导致 segmentation violation。地址空间并不需要是连续的。当线性寻址时，它包含大量未寻址的间隙。

程序不能使用在二级存储而不是内存中的页面，除非这个页面关联到物理内存中。当进程访问这种页面的地址时，内存管理单元(MMU)会产生一个页错误。内核然后进行干预，透明地从二级存储装载需要的页面到物理内存中。由于虚拟内存比物理内存大得多（即使是在现代系统的单个虚拟地址空间中！），内核还经常要把物理内存中的页面交换到二级存储，为更多的页面创造空间。内核试图把最不可能使用的页面交换出去，以最优化系统性能。

共享和写入时复制(copy-on-write)

即使是不同进程的不同虚拟地址空间，虚拟内存的多个页面也可能映射到一个单独的物理内存页面。这允许不同的虚拟地址空间共享物理内存的数据。共享数据可以只能读取、或者同时允许读取和写入。

当一个进程向共享可写页面写入时，可能发生两种情况。最简单的是内核允许写入发生，这时所有共享页面的进程都可以看到写入操作的结果。通常，允许多个进程读取或者写入到一个共享页面，都需要一定程度的协调和同步。

另一种情况，MMU 可能截获写入操作并抛出一个异常；然后内核为写入进程透明地创建一份页面拷贝，并使进程对新页面进行写入操作。我们把这种方式称为写入时复制(COW)。

实际上允许进程读取共享数据，这可以节省空间。当进程希望向共享页面写入时，它会即时地得到一个唯一的页面拷贝，因此进程看起来就像总是拥有自己的私有拷贝一样。由于写入时复制以每一页的方式进行，巨大的文件可以高效地被多个进程共享，并且单个进程在写入页面时会接收到自己唯一的物理页面拷贝。

内存区域

内核把共享某些特性的页面划分为块，特性包括访问权限等。这些块称为内存区域、段、或者映射。每个进程都拥有几种确定的内存区域：

- text 段包含进程的程序代码、字面字符串、常量、和其它只读数据。在 Linux 中，这个段被标记为只读，并直接从 object 文件（可执行程序或库）映射到内存中。
- stack（堆栈）包含进程的执行堆栈，随着栈深度的变化自动增大或减小。执行堆栈包含本地变量和函数返回数据。
- 数据段、或者堆(heap)，包含进程的动态内存。这个段是可写的，并且大小可以增长和收缩。它也是 malloc()返回的内存所在（下一节讨论）。
- bss 段包含未初始化的全局变量。C 标准规定这些变量包含特殊值（本质上都是 0）。Linux 以两种方式优化这些变量。首先，由于 bss 段专门给未初始化数据使用，连接器(ld)不会在目标文件中实际存储这些特殊值。这降低了二进制文件的大小。第二，当这个段被装载到内存中时，内核简单地通过写入时复制把它映射到一个 0 页面，有效地设置变量为它们的默认值。
- 多数地址空间包含多个映射文件，例如程序可执行文件本身、C 和其它连接库、以及数据文件。查看一下/proc/self/maps，或者 pmap 程序的输出，它们包含了进程映射文件的一些示例。

本章讲解 Linux 提供的内存管理接口，包括获取和返回内存、创建和销毁映射、以及两者之间的所有其它接口。

分配动态内存

内存同样分为自动和静态变量，但是任何内存管理系统的基础都是分配、使用和最终返回动态内存。动态内存在运行时分配，而不是编译期，它的分配大小可能直到分配内存之前都不确定。作为开发者，当你对需要的内存数量，或者需要的时间，在程序运行之前不能够确定时，你就需要动态内存。例如，你可能想存储一个文件的内容到内存中，或者从键盘中读取输入。由于文件的大小不确定，而用户可能输入任何数量的按键，缓冲区的大小不能够确定，缓冲区的大小是变化的，读取数据时你需要使它的大小动态。

没有 C 变量来表示动态内存。例如，C 并没有提供机制从动态内存中获取一个 struct pirate_ship。相反，C 提供一个机制，分配足够存储 pirate_ship 结构体的动态内存。程序员通过一个指针与内存交互——在我们的例子中，就是 struct pirate_ship *。

经典的 C 接口获取动态内存是 malloc()：

```
#include <stdlib.h>

void * malloc (size_t size);
```

成功调用 malloc()分配 size 字节内存，并返回一个指向新分配内存区域开始位置的指针。这段内存的内容未定义；不要期望内存会被自动置 0。失败时 malloc()返回 NULL，并设置 errno 为 ENOMEM。

malloc()的使用非常直观，分配固定数量字节如下面例子所示：

```
char *p;

/* give me 2 KB! */
p = malloc (2048);
if (!p)
    perror ("malloc");
```

下面例子分配一个结构体：

```
struct treasure_map *map;
/*
* allocate enough memory to hold a treasure_map stucture
* and point 'map' at it
*/
map = malloc (sizeof (struct treasure_map));
if (!map)
    perror ("malloc");
```

C 语言在赋值时自动把 void 指针提升为任意的类型。因此，上面的例子不需要把 malloc()的返回值类型转换为赋值左边的类型。但是 C++语言并不会执行自动的 void 指针提升。因此，C++用户需要对 malloc()的返回值进行类型转换，如下所示：

```
char *name;

/* allocate 512 bytes */
name = (char *) malloc (512);
if (!name)
    perror ("malloc");
```

有一些 C 程序员喜欢把任何函数返回的指针类型转换为 void，包括 malloc()。我不赞成这个实践，因为如果函数的返回值又转变为非 void 指针，它会隐藏一个错误。此外，如果函数没有正确地声明，这样的类型转换还隐藏了一个 bug（未声明函数默认返回 int，整数到指针的转换不是自动的，将产生一条警告信息。类型转换为 void 会禁止这条警告信息）。前一种情况对 malloc()来说没有危险，后一个则确实存在。

由于 malloc()可能返回 NULL，开发者检查并处理错误条件是至关重要的。许多程序员定义并使用一个 malloc()包装函数，当 malloc()返回 NULL 时打印一条错误消息并终止程序。按照传统，开发者通常命名这个包装函数为 xmalloc()：

```
/* like malloc( ), but terminates on failure */
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 本身也是动态时，动态内存分配可能十分复杂。一个例子就是动态数组分配，数组元素的大小可能是固定的，但需要分配的元素数目是动态的。为了简化这种情况，C 提供了 calloc()函数：

```
#include <stdlib.h>

void * calloc (size_t nr, size_t size);
```

成功调用 calloc()返回一个指针，这块内存适合保存 nr 个元素的数组，每个元素的大小是 size 字节。因此，下面两个调用请求的内存数量是相等的（两个都可能返回更多内存，但决不会更少）：

```
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;
}
```

但是二者的行为却是不一样的。malloc()不保证已分配内存的内容，而 calloc()则会把返回内存块的所有字节置为 0。因此，数组 y 中的所有 50 个整数的值都是 0，而 x 中的元素则是未定义的。除非程序立即设置所有 50 个值，程序员应该使用 calloc()来确保数组元素不被垃圾填充。注意二进制的 0 可能和浮点 0 不一样。

即使不是处理数组，用户通常也希望把动态内存“置 0”。本章后面我们会讨论 memset()，它提供一个接口来设置内存块的每个字节为指定的值。不过让 calloc()执行置 0 操作会更快，因为内核可以提供已经置 0 的内存。

失败时和 malloc()一样，calloc()返回 NULL，并设置 errno 为 ENOMEM。

为什么标准没有定义一个与 calloc()独立的“分配并置 0”的函数，这是一个谜。不过开发者可以轻易地定义自己的接口：

```
/* works identically to malloc( ), but memory is zeroed */
void * malloc0 (size_t size)
{
    return calloc (1, size);
}
```

顺便我们也可以组合这个 malloc0()和我们之前的 xmalloc()：

```
/* like malloc( ), but zeros memory and terminates on failure */
void * xmalloc0 (size_t size)
{
    void *p;

    p = calloc (1, size);
    if (!p) {
        perror ("xmalloc0");
        exit (EXIT_FAILURE);
    }
    return p;
}
```

调整分配大小

C 语言提供 realloc()接口调整（增大或者缩小）已分配的内存：

```
#include <stdlib.h>

void * realloc (void *ptr, size_t size);
```

成功调用 realloc()调整 ptr 指向的内存区域的大小为 size。它返回新调整后的内存指针，这个指针可能和 ptr 一样，也可能不一样——当扩大一个内存区域时，如果 realloc()不能在当前位置扩大内存块，函数可能会分配一个新的 size 字节的内存区域，把原来的区域复制到新内存区域中，并释放旧的内存区域。不管是增大还是缩小，其中的一部分内存区域是保持不变的。由于可能需要复制操作，realloc()扩大内存区域时的代价可能相对比较昂贵。

如果 size 为 0，效果和对 ptr 调用 free()一样。

如果 ptr 为 NULL，操作的结果和调用 malloc()一样。如果 ptr 不是 NULL，它必须是之前通过 malloc()、calloc()、或者 realloc()返回的指针。

失败时 realloc()返回 NULL，并设置 errno 为 ENOMEM。ptr 指向的内存状态不变。

让我们考虑一个缩小内存区域的例子。首先，我们使用 calloc()分配足够的内存来存储两个 map 结构体的数组：

```
struct map *p;

/* allocate memory for two map structures */
p = calloc (2, sizeof (struct map));
if (!p) {
    perror ("calloc");
    return -1;
```

```
}
/* use p[0] and p[1]... */
```

现在假设我们发现其中一个 map 不需要再使用，于是我们决定调整内存大小，把其中一半内存还给系统（这通常并不值得，但如果 map 结构体非常巨大，并且我们要保留 map 很长一段时间，这时就值得这样做）：

```
struct map *r;

/* we now need memory for only one map */
r = realloc (p, sizeof (struct map));
if (!r) {
    /* note that 'p' is still valid! */
    perror ("realloc");
    return -1;
}

/* use 'r'... */

free (r);
```

在上面例子中，p[0]在 realloc()调用之后仍然保留，原先的数据仍然在那里。如果调用失败，p 将不会受到任何影响，因此仍然是合法的。我们可以继续使用它，最终还是需要释放它。反过来，如果调用成功，我们就忽略 p，而使用 r（它可能和 p 相同，如果调整是在当前内存位置进行的话）。在我们完成操作之后，我们需要释放的是 r。

释放动态内存

自动分配在堆栈展开时会自动回收内存，动态分配则在进程地址空间中永久存在，直到它们被手动释放。程序员因此担负着返回已分配动态内存给系统的责任。（当然，静态和动态分配的内存在进程退出时都会最终消失）。

通过 malloc()、calloc()、或 realloc()分配的内存，在不使用时必须通过 free()返还给系统：

```
#include <stdlib.h>

void free (void *ptr);
```

调用 free()释放 ptr 指向的内存。参数 ptr 必须是之前由 malloc()、calloc()、或 realloc()返回的指针。也就是说，你不能使用 free()来释放部分内存（通过传递内存中间的一个指针）——比方说释放已分配内存块的一半。

ptr 可以是 NULL，这种情况下 free()直接返回。因此，常见的在调用 free()之前，检查 ptr 是否 NULL 的动作是多余的。

我们来看一个例子：

```
void print_chars (int n, char c)
{
    int i;
```

```
        for (i = 0; i < n; i++) {
            char *s;
            int j;

            /*
            * Allocate and zero an i+2 element array
            * of chars. Note that 'sizeof (char)'
            * is always 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);

            /* Okay, all done. Hand back the memory. */
            free (s);
        }
    }
```

这个例子分配了 n 个 char 数组，包含的元素个数从 2 个到 n + 1 个。然后对每个数组，循环写入字符 c 到每个字节，除了最后的'\0'作为结束字符，以字符串打印数组，最后释放动态分配的内存。

以 n = 5 调用 print_chars()，c 设置为 X，我们得到如下结果：

```
X
XX
XXX
XXXX
XXXXX
```

当然还有更高效的方法来实现这个函数。不过这里的重点是即使分配的大小和数量只有在运行时才知道，我们仍然可以动态地分配内存。

有一些 Unix 系统，例如 SunOS 和 SCO，提供一个 free()的变种，名字是 cfree()。根据不同的系统，它的行为可能和 free()相同、或者接收三个参数，镜像到 calloc()。在 Linux 中，free()可以处理从任何动态分配机制获得的内存。cfree()永远不应该使用，除非要向后兼容。Linux 版本的 cfree()和 free()完全相同。

注意如果我们不调用 free()的后果。程序将不会返还内存给系统，更严重的是，它可能丢失自己对内存的唯一引用——指针 s——使得不可能再次访问这些内存。我们把这种类型的程序错误称为内存泄漏。内存泄漏和类似的动态内存错误可能是最常见、不幸地也是最有害的 C 编程错误。由于 C 语言把所有管理内存的责任放在了程序员身上，C 程序员必须对所

有内存分配保持关注。

另一个常见的 C 编程错误是“释放后使用”，这发生在释放内存之后，再次访问该内存区域。一旦对一块内存调用了 free()，程序就不应该再次访问它的内容。程序员必须特别小心监视摇摆指针，或者指向无效内存块的非 NULL 指针。两个常见的工具 Electric Fence 和 valgrind*可以帮助你。

对齐

数据对齐指的是硬件计量内存块与它的地址之间的关系。内存中的一个变量，如果地址是类型大小的倍数，则称为自然对齐。例如，一个 32 位的变量的内存地址如果是 4 的倍数，就是自然对齐的——也就是地址的最低两位是 0。因此，2^n 字节大小类型的地址的低 n 位必须都是 0。

不同硬件的对齐规则不一样。有一些机器体系架构有非常严格的数据对齐要求。在某些机器上，装载未对齐数据将导致处理器陷阱。在其它一些系统中，访问未对齐数据则是安全的，但会导致性能的退化。要编写可移植代码，必须避免对齐问题，所有类型都应该自然对齐。

分配对齐的内存

多数情况下，编译器和 C 库能够透明地处理对齐问题。POSIX 规则 malloc()、calloc()、和 realloc()返回的内存对所有标准类型都正确对齐。在 Linux 中，32 位系统总是返回 8 字节边界对齐的内存；64 位系统总是返回 16 字节对齐的内存。

有时候程序员需要请求对齐于更大边界的动态内存，例如按页对齐。动机可能多种多样，最常见的情况是需要正确地对齐缓冲区，以使用直接块 I/O，或者其它软件到硬件的通信。POSIX 1003.1d 提供 posix_memalign()函数完成这个任务：

```
/* one or the other -- either suffices */
#define _XOPEN_SOURCE 600
#define _GNU_SOURCE
#include <stdlib.h>

int posix_memalign (    void **memptr,
                        size_t alignment,
                        size_t size);
```

成功调用 posix_memalign()分配 size 字节动态内存，并确保内存地址对齐于 alignment 的倍数。alignment 参数必须是 2 的幂，并且是 void 指针大小的倍数。memptr 指向所分配内存的地址，然后系统调用返回 0。

失败时没有内存被分配，memptr 的结果未定义，并且调用返回以下错误代码之一：

EINVAL

alignment 参数不是 2 的幂，或者不是 void 指针大小的整数倍。

ENOMEM

没有足够的内存完成请求的分配。

注意函数不会设置 errno——函数直接返回错误代码。

通过 posix_memalign()获得的内存同样使用 free()释放，使用非常简单：

```
char *buf;
int ret;

/* allocate 1 KB along a 256-byte boundary */
ret = posix_memalign (&buf, 256, 1024);
if (ret) {
    fprintf (stderr, "posix_memalign: %s\n",
    strerror (ret));
    return -1;
}

/* use 'buf'... */
free (buf);
```

老的接口。在 POSIX 定义 posix_memalign()之前，BSD 和 SunOS 分别提供以下接口：

```
#include <malloc.h>

void * valloc (size_t size);
void * memalign (size_t boundary, size_t size);
```

valloc()函数的操作和 malloc()相同，除了它分配的内存是按页边界对齐。第四章我们讨论过 getpagesize()获得系统的页大小。

memalign()函数类似，只是分配的内存按 boundary 字节边界对齐，它必须是 2 的幂。在下面例子中，两次分配都返回足够的内存保存 ship 结构体，按页边界对齐：

```
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;
}
/* use 'pirate' and 'hms'... */

free (hms);
free (pirate);
```

在 Linux 中，这两个函数获得的内存都可以通过 free()来释放，但是在其它 Unix 系统中却不一定可以，有些系统甚至没有提供相应的机制来安全地释放这两个函数分配的内存。关心可移植性的程序只能不释放这两个函数分配的内存，没有其它选择。

Linux 程序员只有在兼容老的 Unix 系统时，才应该使用这两个函数；否则 posix_memalign()是首选。只有当需要的对齐大于 malloc()提供的时候，才需要使用这三个函数。

对齐的其它关注

对齐需要的关注不仅仅是标准类型的自然对齐和动态内存分配。例如，非标准和复杂类型比标准类型有更加复杂的对齐需求。此外，当在不同类型指针之间赋值、和使用类型强制转换时，需要加倍关注对齐。

非标准类型。非标准类型和复杂数据类型的对齐要超过标准类型的自然对齐。下面是四个有用的规则：

- 结构体对齐的要求是它成员类型最大的那个。例如，如果一个结构体最大的类型是 32 位整数（按 4 字节边界对齐），则结构体也必须至少以 4 字节边界对齐。
- 结构体同时也引入了填充，确保结构体的每个成员类型都按类型自己的需要对齐。因此，如果 char（按 1 字节对齐）后面跟着一个 int（按 4 字节对齐），编译器会在这两个成员之间插入三个填充字节，以确保 int 能够按 4 字节边界对齐。程序员有时候通过对成员排序——例如降序——来最小化填充“浪费”的字节数。GCC 选项 -Wpadded 可以提供帮助，当编译器插入隐式的填充时将产生警告信息。
- union 要求按最大的联合类型对齐。
- 数组按它的类型对齐。因此，数组的对齐不需要超过它所存储的类型。这时数组的所有成员都能够实现自然对齐。

指针相关的问题。编译器能够透明地处理大多数对齐的需求，暴露潜在的问题需要一点努力。但是处理指针与类型转换时遇到对齐问题却是常见的。

通过指针访问更小对齐转换为更大对齐的数据时，可能导致处理器装载未正确对齐的数据。例如，在下面代码片断中，通过把 c 赋值为 badnews，试图以 unsigned long 来读取 c：

```
char greeting[] = "Ahoy Matey";
char *c = greeting[1];
unsigned long badnews = *(unsigned long *) c;
```

unsigned long 可能按 4 字节或 8 字节边界对齐；而 c 肯定是按 1 字节对齐的。因此，当类型转换后载入 c 时，会违反对齐规则。根据不同的机器体系架构，结果可能导致性能损失，也可能导致程序崩溃。在那些能够检测但不能够正确处理违反对齐规则的机器体系架构中，内核会向进程发送 SIGBUS 信号，并终止进程。我们在第九章讨论信号。

类似的问题可能比你想像的更加常见。实际的情况当然不会像我们的例子这样愚蠢，但他们也不会像我们的例子这样明显。

管理数据段

Unix 系统由于历史原因，提供了直接管理数据段的接口。但是大多数程序都不需要使用这些接口，因为 malloc()和其它内存分配方案更容易使用，也更加强大。为了满足那些好

奇的读者，也为了那些想要实现自己的堆内存分配机制的读者，我们在这里也讨论管理数据段的接口：

```
#include <unistd.h>

int brk (void *end);
void * sbrk (intptr_t increment);
```

这两个函数的名字来源于守旧的 Unix 系统，那时候堆和栈存在于同一个段。堆中的动态内存分配从段的底部向上增长；栈从段的顶部朝着堆向下增长。堆和栈的分界线就被称为 break 或 break point。在现代系统中，数据段存在于独立的内存映射中，我们仍然把地址映射的末尾标记为 break point。

调用 brk()设置 break point（数据段的末尾）为 end 指定的地址。成功时函数返回 0；失败时返回-1，并设置 errno 为 ENOMEM。

调用 sbrk()将数据段的末尾增加 increment，它可以是正数或负数值。sbrk()返回修改后的 break point。因此，increment 为 0 时函数返回当前的 break point：

```
printf ("The current break point is %p\n", sbrk (0));
```

POSIX 和 C 标准都没有定义这些函数。不过几乎所有 Unix 系统都支持一个或两个。可移植的程序应该使用基于标准的函数。

匿名内存映射

glibc 的内存分配使用数据段和内存映射。实现 malloc()的经典方式是把数据段划分为一系列分区（每个分区的大小都是 2 的幂），分配内存时返回满足请求的最接近大小的内存块。释放内存只是简单地把相应分区标记为“free”。如果相邻的分区都是 free，它们可以合并为一个更大的分区。如果堆完全空闲，系统可以使用 brk()来降低 break point，缩小堆的大小，并把内存返回给内核。

这个算法称为“buddy 内存分配方案”。它的优点在于速度和简单，但缺点是引入了两种类型的碎片。当分配内存时返回的内存比请求更多时，就产生了内部碎片，这导致低效的内存使用；当分配内存时返回足够的内存，但内存块被分成两个或多个不相邻的块时，就产生了外部碎片，这也将导致低效的内存使用（因为本来可以使用一个更小的内存块），或者导致随后的内存分配失败（没有足够大的可用内存块）。

此外，这个方案允许一个内存分配“钉”住另一个，阻止 glibc 返回已释放内存给内核。假设已经分配两个内存块 A 和 B，块 A 正好在 break point 上，块 B 正好在 A 的下面。即使程序释放了 B 内存，glibc 也不能调整 break point，除非 A 也被释放。照这样，一个长时期的内存分配可以“钉”住所有其它内存分配。

这个不是什么大问题，因为 glibc 不会经常返还内存给系统。通常堆不会在每次释放内存时缩小。相反，glibc 保留已释放的内存，以便随后的分配使用。只有当堆的大小比已分配内存大许多时，glibc 才会缩小数据段。一个大的内存分配，就可以防止这种缩小。

因此对于大的内存分配，glibc 不使用堆。相反 glibc 创建一个匿名内存映射，来满足内存分配请求。匿名内存映射类似于第四章讨论的基于文件的映射，除了它不使用任何文件（因此叫做匿名映射）。匿名内存映射只是一个大的、用 0 填充的内存块。我们可以把它想像成一个新的堆，全部给一次内存分配使用。由于这些映射存在于堆之外，它们不会导致数据段的碎片。

通过匿名映射分配内存有几个好处：

- 没有碎片的问题。当程序不再需要匿名内存映射时，可以解除映射，内存会立即返回给系统。
- 匿名内存映射可以调整大小，拥有可调整的权限，并且可以像普通映射那样接受建议（参看第四章）。
- 每个分配存在于独立的内存映射中。不需要管理全局堆。

使用匿名内存映射而不是堆同样也有两个缺点：

- 每个内存映射都是系统页大小的整数倍。因此分配内存不是页大小的整数倍时，会导致浪费内存空间。在小的内存分配时这是一个问题，相比于分配内存的大小，浪费的空间相对比较大。
- 创建一个新的内存映射比从堆中返回内存开销更大，后者可能完全不需要内核参与。分配的内存越小，这个缺点就越明显。

考虑这些优点和缺点，glibc 的 malloc()函数在分配小的内存时使用数据段，大的内存分配则使用匿名内存映射。临界点可以配置（本章后面“高级内存分配”一节），不同的 glibc 版本也可能有变化。当前的临界点是 128KB：从堆中分配小于或等于 128KB 内存，更大的内存分配则使用匿名内存映射。

创建匿名内存映射

如果你想对某个内存分配强制使用匿名内存映射而不是堆，或者你正在编写自己的内存分配系统，你可能希望手动创建自己的匿名内存映射——不管是哪种情况，Linux 使它变得很简单。回忆第四章创建映射使用 mmap()系统调用，销毁内存映射使用 munmap()：

```
#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);
```

创建匿名内存映射实际上比创建基于文件的映射更加简单，因为不需要打开和管理文件。这两种类型映射的主要区别在于一个特殊标志的存在，指明创建的映射是匿名的。

我们来看一个例子：

```
void *p;

p = mmap (NULL, /* do not care where */
          512 * 1024, /* 512 KB */
          PROT_READ | PROT_WRITE, /* read/write */
```

```
                MAP_ANONYMOUS | MAP_PRIVATE, /* anonymous, private */
                -1, /* fd (ignored) */
                0); /* offset (ignored) */

        if (p == MAP_FAILED)
                perror ("mmap");
        else
                /* 'p' points at 512 KB of anonymous memory... */
```

对多数匿名映射，调用 mmap()的参数都和例子相同，当然映射内存的大小（字节）是不相同的。其它参数通常如下所述：

- 第一个参数 start 设置为 NULL，表示匿名映射可以按内核的意愿从任意内存开始。也可以指定一个非 NULL 值，只要是按页对齐的，但却限制了可移植性。很少有程序关心映射在内存中的位置！
- prot 参数通常设置 PROT_READ 和 PROT_WRITE 位，使映射可读和可写。一个空映射（全 0）如果不能读取也不能写入，那就没有任何作用。另一方面，很少需要从匿名映射中执行代码，而且设置可执行也带来潜在的安全漏洞。
- flags 参数设置 MAP_ANONYMOUS 位，使映射匿名；以及 MAP_PRIVATE 位，使当前映射私有。
- fd 和 offset 参数在 MAP_ANONYMOUS 位设置时会被忽略。但是有一些老的系统，要求 fd 的值为-1，因此在关心可移植性时传递-1 是一个好主意。

通过匿名映射获得的内存看上去和从堆中获得的内存完全一样。从匿名映射中分配内存的一个好处是页面已经被 0 填充。这完全不需要任何开销，因为内核通过写入时复制(COW)映射应用的匿名页到一个 0 填充的页。因此，返回的内存不需要使用 memset()。实际上，这也是使用 calloc()，而不是 malloc()跟随一个 memset()的好处：glibc 知道匿名映射已经清零，当 calloc()从映射分配时不需要显式的清零。

系统调用 munmap()释放匿名映射，返回已分配的内存给内核：

```
        int ret;

        /* all done with 'p', so give back the 512 KB mapping */
        ret = munmap (p, 512 * 1024);
        if (ret)
                perror ("munmap");
```

映射/dev/zero

其它 Unix 系统，例如 BSD，没有 MAP_ANONYMOUS 标志。相反，它们通过映射一个特殊设备文件/dev/zero 实现类似的方案。/dev/zero 设备提供匿名内存相同的语义。映射包含复制时写入的全 0 页；因此这个行为和映射内存完全相同。

Linux 同样也有一个/dev/zero 设备，并且提供映射这个文件获得填充 0 的内存的能力。实际上，在引入 MAP_ANONYMOUS 之前，Linux 程序员就是使用这种方式。为了向后兼容老版本的 Linux，或者保持与其它 Unix 系统的可移植性，开发者仍然可以通过映射/dev/zero 来创建匿名映射。这和映射其它文件没有任何区别：

```
void *p;
int fd;

/* open /dev/zero for reading and writing */
fd = open ("/dev/zero", O_RDWR);
if (fd < 0) {
	perror ("open");
	return -1;
}

/* map [0,page size) of /dev/zero */
p = mmap (NULL, /* do not care where */
		getpagesize ( ), /* map one page */
		PROT_READ | PROT_WRITE, /* map read/write */
		MAP_PRIVATE, /* private mapping */
		fd, /* map /dev/zero */
		0); /* no offset */

if (p == MAP_FAILED) {
	perror ("mmap");
	if (close (fd))
	perror ("close");
	return -1;
}

/* close /dev/zero, no longer needed */
if (close (fd))
	perror ("close");

/* 'p' points at one page of memory, use it... */
```

按这种方式映射的内存，当然需要使用 munmap()来解除映射。

这个方法需要额外的系统调用，有打开和关闭设备文件的开销。因此，匿名内存是一个更快的方案。

高级内存分配

本章讨论的许多分配操作都由内核参数限制和控制，程序员可以修改这些参数，使用 mallopt()系统调用：

```
#include <malloc.h>

int mallopt (int param, int value);
```

调用 mallopt()设置内存管理相关的参数，param 指定要设置的参数，value 指定参数设

置的值。成功时调用返回非 0 值；失败时它返回 0，注意 mallopt()不设置 errno。它总是倾向于返回成功（乐观的函数），因此应该尽量避免从返回值获得有用的信息。

Linux 当前支持六个 param 参数值，所有都在<malloc.h>中定义：

M_CHECK_ACTION

MALLOC_CHECK_环境变量的值（下一节讨论）。

M_MMAP_MAX

系统中动态内存请求使用的映射的最大数量。当达到这个限制时，所有内存分配都将使用数据段，直到某个映射被释放。设置为 0 表示禁止动态内存分配使用匿名映射。

M_MMAP_THRESHOLD

分配请求将使用匿名映射而不是数据段的临界大小。注意小于这个临界大小的内存分配也可能通过匿名映射来实现。设置为 0 使所有内存分配都使用匿名映射，也实际上禁止了动态内存分配使用数据段。

M_MXFAST

fast bin 的最大大小（字节）。fast bin 是堆中的特殊内存块，它不会与相邻块合并，也不会返回给系统，以增加碎片的代价来换取快速内存分配。设置为 0 则禁止使用 fast bin。

M_TOP_PAD

当调整数据段大小时使用的 padding 数量（字节）。不管什么时候，glibc 使用 brk()增加数据段的大小，它都可能请求比需要更多的内存，希望借此减小不久再次调用额外的 brk()。同样，当 gblic 缩小数据段的大小时，它可以保留一些额外的内存，归还系统稍微少一点的内存。这些额外字节就是 padding。设置为 0 禁止使用 padding。

M_TRIM_THRESHOLD

数据段顶部允许的空闲内存的最小数量（字节）。如果小于这个临界值，gblic 将调用 brk()归还内存给内核。

XPG 标准宽松地定义了 mallopt()，指定了另外三个参数：M_GRAIN、M_KEEP、和 M_NLBLKS。Linux 也定义了这些参数，但是设置它们的值没有任何作用。表 8-1 是所有合法参数的完整列表、它们的默认值、以及它们能够接受的值的范围。

表 8-1. mallopt()参数

参数	来源	默认值	合法值	特殊值
M_CHECK_ACTION	Linux 特定	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 特定	64 * 1024	>= 0	0 禁止使用 mmap()
M_MMAP_THRESHOLD	Linux 特定	128 * 1024	>= 0	0 禁止使用堆
M_MXFAST	XPG 标准	64	0 - 80	0 禁止 fast bin
M_NLBLKS	XPG 标准	Linux 不支持	>= 0	
M_TOP_PAD	Linux 特定	0	>= 0	0 禁止 padding

程序必须在第一次调用 malloc()或者其它内存分配接口之前调用 mallopt()，使用非常简单：

```
int ret;

/* use mmap( ) for all allocations over 64 KB */
```

```
ret = mallopt (M_MMAP_THRESHOLD, 64 * 1024);
if (!ret)
    fprintf (stderr, "mallopt failed!\n");
```

使用 malloc_usable_size()和 malloc_trim()微调

Linux 提供几个函数，能够对 glibc 的内存分配系统进行低层控制。第一个函数允许程序询问一个指定的内存分配包含的可用字节的数量：

```
#include <malloc.h>

size_t malloc_usable_size (void *ptr);
```

成功调用 malloc_usable_size()返回 ptr 指向的内存块的实际分配大小。由于 glibc 可能使用已经存在的块或匿名映射来分配内存，所分配内存的可用空间可能比请求的更大。当然，分配不可能比请求的大小更小。下面是一个使用该函数的例子：

```
size_t len = 21;
size_t size;
char *buf;

buf = malloc (len);
if (!buf) {
    perror ("malloc");
    return -1;
}

size = malloc_usable_size (buf);

/* we can actually use 'size' bytes of 'buf'... */
```

第二个函数是 malloc_trim()，允许程序强制 glibc 立即返回所有空闲内存给内核：

```
#include <malloc.h>

int malloc_trim (size_t padding);
```

成功调用 malloc_trim()将尽可能地缩小数据段，少量的 padding 字节将被保留。然后函数返回 1；失败时函数返回 0。通常在空闲内存达到 M_TRIM_THRESHOLD 字节时，glibc 会自动执行这种缩小。它使用的 padding 大小为 M_TOP_PAD。

除了调试或者教学目的，你几乎不会使用这两个函数。它们不可移植，同时暴露了 glibc 内存分配系统的底层细节到你的应用程序中。

调试内存分配

程序可以设置环境变量 MALLOC_CHECK_，允许内存子系统的增强调试功能。额外的调

试检查导致更加低效的内存分配，但是这种开销在应用开发的调试阶段通常是值得的。

由于使用一个环境变量控制调试，我们不需要重新编译程序。例如，你可以简单地发起如下命令：

```
$ MALLOC_CHECK_=1 ./rudder
```

如果 MALLOC_CHECK_设置为 0，内存子系统默默地忽略所有错误。如果它被设置为 1，一条有用的信息将打印到 stderr。如果它被设置为 2，程序将通过 abort()立即退出。由于 MALLOC_CHECK_改变了运行程序的行为，setuid 程序会忽略这个变量。

获得统计信息

Linux 提供 mallinfo()函数，获取内存分配系统相关的统计信息：

```
#include <malloc.h>

struct mallinfo mallinfo (void);
```

调用 mallinfo()在结构体 mallinfo 中返回统计信息。结构体通过值而不是指针直接返回。它的内容同样在<malloc.h>中定义：

```
/* 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 */
};
```

使用很简单：

```
struct mallinfo m;
m = mallinfo ( );
printf ("free chunks: %d\n", m.ordblks);
```

Linux 同时也提供 malloc_stats()函数，它打印内存相关的统计信息到 stderr：

```
#include <malloc.h>

void malloc_stats (void);
```

在一个内存密集的程序中调用 malloc_stats()得到如下数字：

```
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

到目前为止，我们讨论的动态内存分配的所有机制都使用堆或内存映射来获得内存。这就是我们通常所希望的，因为堆和内存映射本质上就是动态的。程序地址空间中另一个常见的组成是堆栈，堆栈是程序自动变量的所在。

但是没有任何理由不允许程序员使用堆栈来进行动态内存分配，只要分配不会导致堆栈的溢出。在堆栈中分配非常简单，而且能够执行得非常好。要在栈中分配动态内存，可以使用 alloca()系统调用：

```
#include <alloca.h>

void * alloca (size_t size);
```

成功时 alloca()返回一个指向 size 字节大小的内存指针。这段内存存在于堆栈中，并且在调用函数返回时自动释放。有一些实现在失败时返回 NULL，但是大多数 alloca()实现都不会失败，或者无法报告失败。失败就表示堆栈已经溢出。

函数的使用和 malloc()一样，但是你不需要（实际上必须不）释放已分配的内存。下面是使用函数的一个例子：在系统配置目录打开一个指定的文件，它可能是/etc，也可能在编译期确定。函数分配一个新的缓冲区，把系统配置目录拷贝到缓冲区中，然后把缓冲区与提供的文件名连接在一起：

```
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);
}
```

在返回时，通过 alloca()分配的内存将自动释放，因为堆栈会展开并归还给调用函数。这意味着一旦调用 alloca()的函数返回，你就不能再使用这段内存。不过由于你不需要调用 free()做任何清理工作，使用 alloca()的代码会更加清晰一点。下面是使用 malloc()实现的相同函数：

```
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;
}
```

注意你不能在函数调用的参数中使用 alloca()分配的内存，因为分配的内存会存在于函数参数保留堆栈的中间。例如，下面代码是禁止使用的：

```
/* DO NOT DO THIS! */
ret = foo (x, alloca (10));
```

alloca()接口在不同的系统的情况不一样。在许多系统中，它都工作得很糟糕，或者导致未定义的行为。在那些堆栈大小固定或者堆栈很小的系统中，使用 alloca()很容易导致堆栈溢出，并杀死你的程序。在另外一些系统中，alloca()甚至不存在。充满 bug 和不统一的实现使 alloca()赢得了一个很坏的名声。

因此，如果你的程序必须保持可移植性，你应该避免使用 alloca()。然而在 Linux 中，alloca()是一个非常有用的工具，并且未得到充分使用。alloca()工作得非常好——在许多体系架构中，通过 alloca()分配内存只需要增加堆栈指针——它比 malloc()方便许多。对于 Linux 特定代码的小的分配，alloca()能够获得极佳的性能。

在堆栈中复制字符串

alloca()常见的一个用法是临时复制一个字符串。例如：

```
/* we want to duplicate 'song' */
char *dup;

dup = alloca (strlen (song) + 1);
strcpy (dup, song);

/* manipulate 'dup'... */
```

```
return; /* 'dup' is automatically freed */
```

由于经常有这样的需要，以及 alloca()提供的速度，Linux 系统提供 strdup()函数的变体，用来复制指定的字符串到堆栈中：

```
#define _GNU_SOURCE
#include <string.h>

char * strdupa (const char *s);
char * strndupa (const char *s, size_t n);
```

调用 strdupa()返回 s 的一份拷贝。调用 strndupa()最多复制字符串 s 的前 n 位。如果 s 比 n 更长，拷贝到 n 为止，函数自动在末尾添加一个 null 字节。这些函数提供 alloca()一样的好处。复制的字符串在调用函数返回时自动释放。

POSIX 没有定义 alloca()、strdupa()、和 strndupa()函数，而且它们在其它操作系统中的历史记录也有污点。如果关注可移植性，那就不应该使用这些函数。不过在 Linux 中，alloca()和其它函数工作得非常好，可以提供非常优秀的性能，仅仅需要调整堆栈指针，替代复杂的动态内存分配。

可变长度数组

C99 引入了可变长度数组(VLA)，它的长度在运行时而不是编译时设置。GNU C 之前已经支持可变长度数组有一段时间了，但是现在 C99 已经标准化可变长度数组，我们有更强的理由使用它。VLA 以 alloca()类似的方式避免了动态内存分配的开销。

使用 VLA 的方法如下：

```
for (i = 0; i < n; ++i) {
    char foo[i + 1];
    /* use 'foo'... */
}
```

在上面代码片断中，foo 是一个大小为 i + 1 的 char 数组。在每次循环中，foo 被动态创建，并在退出循环时自动清理。如果我们使用 alloca()而不是 VLA，内存就只能在函数返回时才被释放。使用 VLA 确保内存在每次循环迭代都会被释放。因此，使用 VLA 最多消耗 n 字节，而 alloca()将消耗 n * (n + 1) / 2 字节。

使用可变长度数组，我们可以如下重写 open_sysconf()函数：

```
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()和可变长度数组主要的区别是前者获得的内存在函数执行期间存在，而后者获得的内存在变量退出作用域之前存在，它可能在函数返回之前。这个特性有好处也有坏处。在我们前面讨论的循环例子中，在每次循环迭代中回收内存没有任何副作用（我们确实不需要空闲这些额外的内存）。但是如果由于某些原因我们希望在循环迭代之外保留内存，那使用alloca()就更加合适。

在一个函数中混合 alloca()和可变长度数组可能导致奇怪的行为。在一个特定的函数中只能使用其中一个。

选择内存分配机制

本章讨论了这么多内存分配选择，可能程序员想知道对特定的工作而言，哪个才是最佳方案。在所有方案中，通常 malloc()是你最佳的选择。不过有时候，不同的方式能够提供更佳的工具。表 8-2 总结了选择内存分配机制的指导方针：

表 8-2. Linux 的内存分配方式

分配方式	优点	缺点
malloc()	简单、常用	返回内存未置 0
calloc()	使分配数组简单、返回内存置 0	分配数组专用接口
realloc()	调整已分配内存大小	只有在调整已分配内存大小时有用
brk()和 sbrk()	允许直接控制堆	对多数用户太底层
匿名内存映射	使用简单、可共享、允许开发者调整保护级别和提供建议、对大的映射最佳	对小的分配不够有效、malloc()在映射最佳时自动使用匿名内存映射
posix_memalign	分配内存对齐于合理的边界	相对较新，因此可移植性可能置疑；只适应于强烈要求对齐的应用
alloca()	分配非常快速，不需要显式释放内存；对小的分配非常适用	无法返回错误，对大的分配不适用，在某些 Unix 系统中容易破碎
可变长度数组	和 alloca()一样，但是在数组超出作用域时释放内存，而不是函数返回时	只对数组有用；alloca()的释放方式在某些情况下可能更加适用；在其它 Unix 系统中可能比 alloca()更少见

最后，我们不要忘记另外一个选择：自动和静态内存分配。在堆栈中分配自动变量，或者堆中分配全局变量，通常更加容易，并且不需要程序员管理指针和释放内存。

操作内存

C 语言提供一组函数操作内存。这些函数的操作在许多方面类似于字符串操作接口，例

如 strcmp()和 strcpy()，但是它们依赖于用户提供缓冲区大小而不是假设字符以 null 结束。注意内存操作函数都不返回错误。由程序员自己防止错误的产生——传递错误的内存区域将导致段错误。

设置字节

在内存操作的所有函数中，最常用的是简单的 memset()：

```
#include <string.h>

void * memset (void *s, int c, size_t n);
```

调用 memset()设置指针 s 开始的 n 个字节为字节 c，并返回 s。常见的用法是对一块内存清零：

```
/* zero out [s,s+256) */
memset (s, '\0', 256);
```

bzero()是一个由 BSD 引入的更老的，已经被不赞成使用的接口，完成相同的任务。新的代码应该使用 memset()，不过 Linux 为了向后兼容和保持与其它系统的可移植性，也提供 bzero()接口：

```
#include <strings.h>

void bzero (void *s, size_t n);
```

下面调用和上面 memset()例子相同：

```
bzero (s, 256);
```

注意 bzero()——以及其它 b 接口——需要头文件<strings.h>而不是<string.h>。

如果你能使用 calloc()就不要使用 memset()！避免使用 malloc()分配内存，然后立即通过 memset()清零。虽然结果是一样的，使用 calloc()替换这两个函数，calloc()返回已经清零的内存，是更好的选择。你不仅可以少调用一个函数，而且 calloc()可以从内核获得已经清零的内存。在这种情况下，你避免了手动设置内存的每个字节，因此能够提高性能。

比较字节

类似于 strcmp()，memcmp()比较两块内存是否相等：

```
#include <string.h>

int memcmp (const void *s1, const void *s2, size_t n);
```

调用 memcmp()比较 s1 和 s2 的前 n 个字节，如果比较的块内存相等则返回 0，如果 s1

小于 s2 则返回一个负数，否则如果 s1 大于 s2 返回一个正数值。

BSD 再次提供一个现在已经不赞成使用的接口，完成大体上相同的任务：

```
#include <strings.h>

int bcmp (const void *s1, const void *s2, size_t n);
```

调用 bcmp()比较 s1 和 s2 的前 n 个字节，如果相等则返回 0，不相等返回非 0 值。

由于结构体的 padding（参考本章前面“其它对齐关注事项”一节），通过 memcmp()或 bcmp()来比较两个结构体是否相等是不可靠的。padding 中可能存在未初始化的垃圾字节，导致两个本来相等的结构体变得不相等。因此，如下代码是不安全的：

```
/* are two dinghies identical? (BROKEN) */
int compare_dinghies (struct dinghy *a, struct dinghy *b)
{
    return memcmp (a, b, sizeof (struct dinghy));
}
```

相反，程序员必须比较结构体的每个成员。这个方法带来一点点优化，但无疑需要比不安全的 memcmp()更多的工作。下面是等同的代码：

```
/* are two dinghies identical? */
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;

    /* and so on, for each member... */
}
```

移动字节

memmove()复制 src 的前 n 个字节到 dst，并返回 dst：

```
#include <string.h>

void * memmove (void *dst, const void *src, size_t n);
```

BSD 再次提供一个已被反对的接口，完成相同的任务：

```
#include <strings.h>
```

```
void bcopy (const void *src, void *dst, size_t n);
```

注意尽管这两个函数接受相同的参数，他们的前两个参数的顺序刚好相反。

bcopy()和 memmove()都可以安全地处理内存区域的重叠（如果 dst 的部分内存在 src 中）。这允许内存字节在指定的区域中上下移动。由于这种情况很少，而且程序员应该了解这个情况，C 标准定义了一个 memmove()变体，它不支持内存区域重叠。这个函数可能会更快：

```
#include <string.h>

void * memcpy (void *dst, const void *src, size_t n);
```

memcpy()函数的行为和 memmove()相同，除了 dst 和 src 不能够重叠以外。如果它们确实重叠，结果是未定义的。

另一个安全的复制函数是 memccpy()：

```
#include <string.h>

void * memccpy (void *dst, const void *src, int c, size_t n);
```

memccpy()函数的行为和 memcpy()一样，除了它在 src 前 n 个字节遇到字节 c 时会停止复制。调用返回指向 dst 中字节 c 之后的指针，或者当 c 未找到时返回 NULL。

最后，你可以使用 mempcpy()来穿过一段内存：

```
#define _GNU_SOURCE
#include <string.h>

void * mempcpy (void *dst, const void *src, size_t n);
```

mempcpy()函数和 memcpy()一样，除了它返回的指针指向复制字节的下一个字节。这在复制一组数据到连续的内存时很有用——不过由于它仅仅返回 dst + n，也不是那么有用。这个函数是 GNU 专有的。

查找字节

memchr()和 memrchr()函数在内存块中查找指定的字节：

```
#include <string.h>

void * memchr (const void *s, int c, size_t n);
```

memchr()函数扫描 s 指向的内存的前 n 个字节，直到找到字符 c 为止，字符 c 被解释为 unsigned char。

调用返回指向匹配 c 的第一个字节的指针，否则没找到则返回 NULL。

```
#define _GNU_SOURCE
#include <string.h>
```

```
void * memrchr (const void *s, int c, size_t n);
```

memrchr()函数和 memchr()相同，不过它从 s 指向的内存的第 n 个字节开始向后查找，而不是从 s 向前查找。和 memchr()不一样，memrchr()是 GNU 扩展，不是 C 语言的一部分。

对于更复杂的查找任务，可以使用 memmem()查找内存块中的任意字节数组（相当糟糕的命名）：

```
#define _GNU_SOURCE
#include <string.h>

void * memmem (const void *haystack,
               size_t haystacklen,
               const void *needle,
               size_t needlelen);
```

memmem()函数返回内存 haystack 中第一次出现子块 needle 的指针，haystack 的长度为 haystacklen，needle 的长度的 needlelen。如果函数没有在 haystack 中找到 needle，则返回 NULL。这个函数也是 GNU 的扩展。

Frobnicating 字节

Linux C 库提供一个接口简单地旋转字节数据：

```
#define _GNU_SOURCE
#include <string.h>

void * memfrob (void *s, size_t n);
```

调用 memfrob()模糊内存 s 开始的 n 个字节，把每个字节与数字 42 进行异或，调用返回指针 s。

调用 memfrob()的效果可以通过对相同内存再次调用 memfrob()进行反转。因此，下面代码对 secret 没有任何影响：

```
memfrob (memfrob (secret, len), len);
```

这个函数不能作为加密的替代；它的用途只是简单地模糊字符串。它是 GNU 专有的。

锁定内存

Linux 实现了按需要进行页面调度(demand paging)，这意味着在页面需要时会从磁盘中交换进来，在不需要时会交换到磁盘。这允许系统的虚拟地址空间和物理内存的实际大小没有直接关系，因为磁盘交换空间可以提供物理内存接近无限的假象。

这种交换透明进行，应用通常不需要关注（甚至根本不知道）Linux 内核的页面调度行为。但是在两种情况下，应用可能希望能够控制页面的行为：

确定性

强制时间(timing constraints)的应用需要确定的行为。如果某些内存访问导致页面错误（将导致开销很大的磁盘 I/O 操作），应用可能超过时间限制。通过确保应用需要的页面总是存在于物理内存，而且从不被交换到磁盘，应用就可以保证内存访问不会导致页面错误，因此提供一致、确定和改进的性能。

安全

如果私有的机密数据保留在内存中，机密可能最终被交换并存储未加密数据到磁盘中。例如，如果一个用户的私有密钥加密后存储在磁盘中，内存中密钥的未加密拷贝可能会最终交换到交换分区。在高安全性要求的环境中，这种行为是不可接受的。应用可以要求包含密钥的内存总是保留在物理内存中，来解决这个问题。

当然，改变内核的行为可能导致系统总体性能的负面影响。一个应用的确定性或安全性可能改善了，但是它的页面锁定在内存中，其它应用的页面就需要交换出去。如果我们相信内核的设计的话，内核总是选择最佳的页面交换出去（在未来最不可能使用的页面），因此当你改变内核的行为时，它不得不交换非最佳的页面。

锁定部分地址空间

POSIX 1003.1b-1993 定义了两个接口来“锁定”一个或多个页面到物理内存，确保它们永远不被交换到磁盘中。第一个接口锁定指定间隔的地址：

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

int mlock (const void *addr, size_t len);
```

调用 mlock()锁定 addr 开始的虚拟内存，并延伸到 len 字节的物理内存。成功时调用返回 0；失败时返回-1，并适当地设置 errno。

成功调用 mlock()锁定内存[addr, addr+len)中的所有物理页面。例如，如果调用仅仅指定一个字节，则字节所在的整个页面将被锁定到内存中。POSIX 标准规定 addr 必须按页面边界对齐。Linux 没有强制这个要求，如果需要会默认地把 addr 传入到最接近的页面。但是需要可移植性的程序应该确保 addr 正好按页面边界对齐。

合法的 errno 代码包括：

EINVAL

参数 len 为负数。

ENOMEM

调用者试图锁定超过 RLIMIT_MEMLOCK 资源限制允许的更多页面(下一节“锁定限制”)。

EPERM

RLIMIT_MEMLOCK 资源限制为 0，但是进程不具有 CAP_IPC_LOCK 能力。

子进程不会通过 fork()继承父进程已锁定的内存。不过由于 Linux 地址空间的写入时复制行为，子进程在写入页面之前，实际上能够有效地锁定内存。

下面例子假设程序在内存中保存着一个加密字符串，进程可以如下锁定包含字符串的页面：

```
int ret;

/* lock 'secret' in memory */
ret = mlock (secret, strlen (secret));
if (ret)
        perror ("mlock");
```

锁定所有地址空间

如果进程希望锁定自己的整个地址空间到物理内存中，使用 mlock()会很麻烦。对于这种目的——实时应用很常见——POSIX 定义了一个系统调用，用来锁定整个地址空间：

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

int mlockall (int flags);
```

调用 mlockall()锁定当前进程地址空间的所有页面到物理内存。flags 参数是以下两个值的按位或，控制函数的行为：

MCL_CURRENT

如果设置这个值，则表示 mlockall()锁定所有当前已映射的页面——堆栈、数据段、映射文件等等——到进程的地址空间。

MCL_FUTURE

如果设置这个值，则表示 mlockall()确保进程将来映射到地址空间的页面同样也被锁定到内存中。

多数应用同时指定这两个值。

成功时调用返回 0；失败时返回-1，并设置 errno 为以下错误代码之一：

EINVAL

参数 flags 为负数。

ENOMEM

调用者试图锁定超过 RLIMIT_MEMLOCK 资源限制允许的更多页面（下一节“锁定限制”）。

EPERM

RLIMIT_MEMLOCK 资源限制为 0，但是进程不具有 CAP_IPC_LOCK 能力。

解除内存锁定

要从物理内存解除页面锁定，再次允许内核按需要交换页面到磁盘中，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);
```

munlock()系统调用解除锁定 addr 开始的 len 字节所在的页面，它撤销 mlock()的作用。

系统调用 munlockall()则撤销 mlockall()的作用。两个调用成功时都返回 0；出错时返回-1，并设置 errno 为以下值之一：

EINVAL

参数 len 无效(仅对 munlock())。

ENOMEM

指定的某些页面无效。

EPERM

RLIMIT_MEMLOCK 资源限制为 0，但进程不具有 CAP_IPC_LOCK 能力。

内存锁不嵌套。因此一个 munlock()或 munlockall()就能够解除页面的锁定，不管之前对页面调用了多少次 mlock()或 mlockall()。

锁定限制

由于锁定内存可能影响系统总体性能——实际上，如果过多页面被锁定，内存分配将失败——Linux 对一个进程能够锁定的页面数量设置了限制。

拥有 CAP_IPC_LOCK 能力的进程可以锁定任意数量的页面到内存中。不具有这个能力的进程只能锁定 RLIMIT_MEMLOCK 字节。默认情况下这个资源限制是 32KB——对于锁定一两个机密数据到内存中已经足够大，而且又不会大到足以影响系统的性能。（第六章讨论了资源限制，以及如何获得和设置相应的值）。

页面是否在物理内存中？

为了便于调试和诊断，Linux 提供了 mincore()函数，它可以用来确定指定范围的内存是否存在于物理内存，还是已经被交换到磁盘：

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

int mincore (void *start,
             size_t length,
             unsigned char *vec);
```

调用 mincore()提供一个向量，描述在当前系统调用时映射的哪些页面在物理内存中。调用通过 vec 返回向量，描述从 start（必须对齐于页）开始的 length（不需对齐于页）字节内存。vec 中的每个字节对应于所提供的内存范围的一个页，第一个字节描述第一页，并且线性向前增长。因此，vec 至少必须足够包含(length – 1 + page size) / page size 字节。每个字节的最低位如果为 1 则表示页面驻留在物理内存中，如果为 0 则不在物理内存中。当前没有定义其它位，保留将来使用。

成功时调用返回 0；失败时返回-1，并设置 errno 为以下值之一：

EAGAIN

内核资源不足，无法完成请求。

EFAULT

vec 指向的参数是无效地址。

EINVAL

start 参数没有对齐于页边界。

ENOMEM

[address, address + 1)包含的内存不是基于文件的映射。

当前这个系统调用只能对通过 MAP_SHARED 创建的基于文件的映射正常工作，这极大地限制了函数的使用。

机会主义(opportunistic)分配

Linux 采用一种机会主义分配策略。当一个进程从内核请求额外的内存——例如扩大数据段，或者创建新的内存映射——内核直接承诺分配内存，但并不实际提供任何物理内存。只有当进程写入到新分配的内存时，内核才会把之前承诺的内存转化为实际分配的物理内存。内核以一页一页的形式完成这个操作，按需要执行页面请求调度与写入时复制。

这种行为有几个好处。首先，延迟分配内存允许内核推迟大部分工作到最后时刻——如果最后确实需要分配的话。第二，由于请求是按需要的页来满足，只有实际使用的物理内存需要消耗物理存储。最后，提交的内存数量可以远远超出实际的物理内存，甚至超出可用的交换空间。最后一个特性被称为过量使用(overcommitment)。

Overcommitting 和 OOM

overcommitting 允许系统运行更多更大的应用，如果每个请求的内存页在分配时而不是使用时就必须物理存储支持，那能够运行的应用就要少得多。没有 overcommitting，通过写入时复制映射 2GB 文件，需要内核设置 2GB 存储空间。而有了 overcommitting，映射 2GB 文件只需要进程实际写入的页面数。同样，没有 overcommitting，每个 fork()都需要足够的存储空间来复制地址空间，即使多数页面从来不会经过写入时复制。

但是如果进程试图要求比系统拥有的物理内存和交换空间更多的内存会怎么样呢？在这种情况下，一个或多个请求肯定会失败。由于内核已经承诺了内存分配——请求内存的系统调用已经返回了成功——而进程现在试图使用之前承诺的内存，内核唯一能做的就是杀死某个进程，释放可用内存。

当 overcommitting 导致内存不足以满足已提交的请求，我们称之为 out of memory(OOM)情况发生。内核对 OOM 情况的响应，是采用 OOM killer 挑选一个最“适合”的进程终止。内核设法找出最不重要而又消耗内存最多的那个进程。

OOM 情况非常少见——因此允许 overcommitting 能够带来巨大的效用。但是无可否认，OOM 情况是不受欢迎的，而且不确定的进程被 OOM killer 杀死常常也是不可接受的。

对于不希望 overcommitting 的系统，内核允许禁止 overcommitting，通过文件/proc/sys/vm/overcommit_memory，以及类似的 sysctl 参数 vm.overcommit_memory。

这个参数的默认值是 0，指示内核执行启发式的 overcommitting 策略，有理由的 overcommitting 内存，但不允许不正常的内存提交。参数值为 1 允许所有内存提交成功，不考虑任何后果。有些内存密集的应用，例如科学领域的应用，倾向于请求比需要多得多的内存，这种情况下设置为 1 是合理的。

参数值为 2 完全禁止 overcommitting，同时允许 strict accounting。在这个模式下，内存提交限制为交换区域的大小，加上可配置的物理内存百分比。可配置的百分比通过文件

/proc/sys/vm/overcommit_ratio 设置，或者类似的 sysctl 参数 vm.overcommit_ratio。默认值是 50，它限制内存提交的大小为交换区域加上物理内存的一半。由于物理内存也包含内核、页表、系统保留页面、锁定页面等等，只有一部分是可以交换和确保能用来分配内存的。

请小心 strict accounting！许多系统设计者，被 OOM killer 的概念击败，认为 strict accounting 是万灵药。然而应用通常执行许多不必要的分配，这需要 overcommitting 支持，允许这个行为也是虚拟内存的主要动机。

第九章 信号

信号是软件中断，提供一种处理异步事件的机制。这些事件可能来自系统外部——例如用户产生中断字符(通常 Ctrl-C)；或者来自程序内部或者内核——例如进程执行代码除以零。作为进程间通信(IPC)的基础构成，一个进程也可以向其它进程发送信号。

关键点不仅仅在于事件异步发生（例如用户可以在程序执行的任何时候按下 Ctrl-C），程序处理信号也是异步的。信号处理函数在内核中注册，内核在信号发出时异步地调用函数。

信号在 Unix 的早期就已经是 Unix 的一部分。不过随着时间的过去，信号也得到了很大的进步——最显著的变化是可靠性（信号在过去可能会丢失），以及功能（信号现在可以携带用户定义的数据）。一开始不同的 Unix 系统对信号进行了许多不兼容的改变。感谢 POSIX 前来解救并标准化了信号处理。这个标准正是 Linux 提供的，也是本章将讨论的主题。

在这一章，我们从信号的简介开始，讨论它们的用法和错误用法。然后我们讨论各种 Linux 管理和操作信号的接口。

多数不平凡的应用都需要与信号交互。即使你故意设计应用，不依赖于信号的通信功能（通常是一个好主意）——你仍然不得不在某些情况下与信号一起工作，例如当处理程序终止时。

信号的概念

信号有一个非常精确的生命周期。首先，信号被抛出（有时候也叫做发送和产生）。然后内核存储信号，直到准备好能够把信号交付出去。最后，一旦交付信号，内核会适当地处理信号。内核根据进程的要求，可能执行以下三种动作：

忽略信号

不执行任何动作。有两个信号不能被忽略：SIGKILL 和 SIGSTOP。原因是系统管理员必须能够杀死或停止进程，如果进程能够选择忽略 SIGKILL 和 SIGSTOP 信号，则进程将无法杀死和停止。

捕获并处理信号

内核挂起进程的当前代码执行路径，跳转到先前注册的信号处理函数，然后进程执行这个处理函数。一旦进程从函数返回，它会跳回原先捕获信号的地方继续执行。

SIGINT 和 SIGTERM 是两个经常被捕获的信号。进程捕获 SIGINT 来处理用户产生的中断字符——例如终端可能捕获这个信号并立即返回给主程序。进程捕获 SIGTERM 信号在进程终止之前执行必要的清理工作，例如断开网络，或者移除临时文件。SIGKILL 和 SIGSTOP 信号不能被捕获。

执行默认动作

这个动作依赖于被发送的具体信号。默认动作通常是终止进程。例如 SIGKILL 就是这种情况。不过许多信号具有特殊目的，程序员只有在特殊情况下才会关注它，这些信号的默认动作是忽略，因为多数程序对它并不感兴趣。我们马上会查看不同的信号以及它们的默认动作。

传统上，当一个信号被发送时，处理信号的函数对特定信号的产生，以及发生了什么没有任何信息。今天内核可以向程序员提供信号的大量上下文信息，而且信号也可以传递用户

定义的数据，和之后更高级的 IPC 机制一样。

信号标识符

每个信号有一个符号名，以前缀 SIG 开始。例如 SIGINT 是用户按下 Ctrl-C 时发送的信号，SIGABRT 是进程调用 abort()函数发送的信号，SIGKILL 是进程被强制终止时发送的信号。

信号全部在头文件<signal.h>中定义。信号实际上就是简单的正整数的预处理定义（每个信号都与一个整数整数标识符相关联）。名字－整数的映射是实现相关的，在不同的 Unix 系统中不一样，虽然最前面的十几个信号通常都映射到相同的整数（例如 SIGKILL 是信号 9）。一个好的程序员总是使用信号的可读名字，而决不使用信号的整数值。

信号数值从 1 开始（通常是 SIGHUP），线性向上增长。总共大约有 31 个信号，不过多数程序只与其中少数信号打交道。没有哪个信号的值是 0，它是一个特殊值，称为 null 信号。null 信号其实没有任何重要的地位，它实际上不值得这样一个特殊的名字，不过有一些系统调用(例如 kill())使用 0 表示特殊情况。

你可以生成当前系统支持的信号列表，通过 kill –l 命令。

Linux 支持的信号

表 9-1 列出了 Linux 支持的所有信号。

表 9-1. 信号

信号	描述	默认动作
SIGABRT	abort()发送	core dump 终止
SIGALRM	alarm()发送	终止
SIGBUS	硬件或对齐错误	core dump 终止
SIGCHLD	子进程终止	忽略
SIGCONT	进程停止后继续执行	忽略
SIGFPE	算术异常	core dump 终止
SIGHUP	进程的控制终端关闭(最经常的是用户登出)	终止
SIGILL	进程试图执行非法指令	core dump 终止
SIGINT	用户产生中断字符(Ctrl-C)	终止
SIGIO	异步 I/O 事件	终止(a)
SIGKILL	不可捕获的进程终止	终止
SIGPIPE	进程写入管道，但是管道没有读者	终止
SIGPROF	Profiling 定时器到期	终止
SIGPWR	电源故障	终止
SIGQUIT	用户产生退出字符(Ctrl-\)	core dump 终止
SIGSEGV	内存访问违规	core dump 终止
SIGSTKFLT	协处理器堆栈错误	终止(b)
SIGSTOP	暂停进程的执行	停止
SIGSYS	进程试图执行非法的系统调用	core dump 终止
SIGTERM	可捕获的进程终止	终止

SIGTRAP	遇到断点	core dump 终止
SIGTSTP	用户产生暂停字符(Ctrl-Z)	停止
SIGTTIN	后台进程从控制终端读取	停止
SIGTTOU	后台进程向控制终端写入	停止
SIGURG	紧急未决 I/O	忽略
SIGUSR1	进程定义信号	终止
SIGUSR2	进程定义信号	终止
SIGVTALRM	以 ITIMER_VIRTUAL 标志调用 setitimer()时产生	终止
SIGWINCH	终端终端窗口大小改变	忽略
SIGXCPU	超出处理器资源限制	core dump 终止
SIGXFSZ	超出文件资源限制	core dump 终止

(a) 在其它 Unix 系统(例如 BSD)上，默认动作是忽略该信号。
(b) Linux 内核不再产生这个信号，保留仅仅是为了向后兼容。

还有其它几个信号值存在，但是 Linux 对它们的定义等于其它值：SIGINFO 定义为 SIGPWR，SIGIOT 定义为 SIGABRT，SIGPOLL 和 SIGLOST 定义为 SIGIO。

现在我们有了一个快速参考信号表，下面让我们详细地讨论每一个信号：

SIGABRT

abort()函数向调用进程发送这个信号。进程然后终止并产生一个 core 文件。在 Linux 中，assert()断言在失败时调用 abort()函数。

SIGALRM

当 alarm 到期时，alarm()和 setitimer()函数（ITIMER_REAL 标志）向调用进程发送这个信号。第十章讨论它们以及相关的函数。

SIGBUS

当进程发生内存保护之外的硬件错误时，内核抛出这个信号，内存保护错误时抛出的信号是 SIGSEGV。在传统的 Unix 系统中，这个信号表示许多不可恢复的错误，例如未对齐内存访问。但是 Linux 内核能够自动修复多数类似错误，而不抛出信号。内核在进程不正确地访问通过 mmap()创建的内存区域时，确实会抛出这个信号（第八章讨论内存映射）。除非信号被捕获，内核会终止进程，并产生一个 core dump。

SIGCHLD

无论什么时候进程终止或停止时，内核向父进程发送这个信号。由于 SIGCHLD 默认被忽略，进程如果关心子进程的生命周期，必须显式地捕获并处理信号。这个信号的处理器通常调用 wait()（第五章讨论），来确定子进程的 pid 和 exit 代码。

SIGCONT

当进程在停止后又继续执行时，内核向进程发送这个信号。信号默认被忽略，但是如果进程希望在继续之后执行某个动作，可以捕获它并相应处理。这个信号通常被终端或编辑器使用，它们希望能够刷新屏幕。

SIGFPE

这个信号描述算术异常，而且不仅仅是浮点操作相关的异常。这些异常包括溢出、下溢、以及除以 0。默认动作是终止进程并产生 core 文件，但是进程可以捕获并处理这个信号。注意如果进程选择继续运行，则进程的行为以及操作的结果是未定义的。

SIGHUP

当会话的终端断开时，内核向会话领导者发送这个信号。在会话领导者终止时，内核还将向前台进程组的每个进程发送这个信号。默认动作是终止进程，这是有意义的——信号暗示用户已经登出。Daemon 进程通过指示它们重新装载配置文件的机制来“重载”这个信号。例如发送 SIGHUP 信号给 Apache，就指示它重新读取 httpd.conf 文件。 这样使用 SIGHUP 是一个通用的惯例，但不是强制的。这样的实践是安全的，因为 daemon 并不拥有控制终端，因此不会收到这个信号。

SIGILL

当进程试图执行非法机器指令时，内核发送这个信号。默认动作是终止进程，并产生一个 core dump。进程可以选择捕获并处理 SIGILL 信号，但是在发生 SIGILL 信号之后的行为是未定义的。

SIGINT

当用户输入中断字符（通常 Ctrl-C）时，这个信号会发送给前台进程组中的所有进程。默认行为是终止；但是进程可以选择捕获并处理这个信号，通常的做法是在进程终止前执行相应的清理工作。

SIGIO

当 BSD 风格的异步 I/O 事件产生时发送这个信号。在 Linux 中很少使用 BSD 风格的 I/O。

SIGKILL

通过 kill()系统调用发送这个信号；它提供系统管理员一个无条件地杀死进程的方法。这个信号不能被捕获或忽略，它的结果总是终止进程。

SIGPIPE

如果进程写入到管道，但是管道的读者已经终止，内核将抛出这个信号。默认动作是终止进程，但是这个信号可以被捕获和处理。

SIGPROF

setitimer()函数使用 ITIMER_PROF 标志时，在 profiling 定时器到期时产生这个信号。默认动作是终止进程。

SIGPWR

这个信号是系统依赖的。在 Linux 中，它表示电池能量低（例如 UPS）。UPS 监控 daemon 发送这个信号给 init，然后 init 响应信号，并执行清理工作以及关闭系统——希望在电源耗尽之前能够完成。

SIGQUIT

当用户输入终端退出字符（通常 Ctrl-\）时，内核向前台进程组的所有进程发送这个信号。默认动作是终止进程，并产生 core dump。

SIGSEGV

这个信号的名字来源于 segmentation violation，当进程试图访问非法内存时，内核发送这个信号。这包括访问未映射内存、从不可读内存中读取、在不可执行内存中执行代码、或者写入到不可写内存中。进程可以捕获和处理这个信号，但是默认动作是终止进程并产生 core dump。

SIGSTOP

这个信号只有 kill()能发送。它无条件地停止一个进程，并且不能被捕获和忽略。

SIGSYS

当进程试图调用非法的系统调用时，内核发送这个信号。当二进制文件在新版本操作系统中构建（拥有新的系统调用），而在老版本的系统中运行时，就会发生这种情况。通过 glibc 正确构建二进制来调用系统调用，就不会收到这个信号。相反，非法的系统调用会返回-1，并且设置 errno 为 ENOSYS。

SIGTERM

这个信号只会由 kill()发送；它允许用户温和地终止一个进程（默认动作）。进程可以选择捕获这个信号，在终止前执行清理工作，但是捕获这个信号被认为是粗鲁的，而且进程因此不能够快速地终止。

SIGTRAP

当进程穿过断点时，内核向进程发送这个信号。通常调试器捕获这个信号，其它进程忽略它。

SIGTSTP

当用户输入暂停字符（通常 Ctrl-Z）时，内核向前台进程组的所有进程发送这个信号。

SIGTTIN

当后台进程试图从控制终端读取时，内核将发送这个信号给进程。默认动作是停止进程。

SIGTTOU

当后台进程试图写入到控制终端时，内核将发送这个信号给进程。默认动作是停止进程。

SIGURG

当 out-of-band(OOB)数据到达 socket 时，内核向进程发送这个信号。out-of-band 数据超出了本书的范围。

SIGUSR1 和 SIGUSR2

这两个信号是用户定义使用的；内核从不抛出它们。进程可以按任何目的和方式来使用 SIGUSR1 和 SIGUSR2。典型的用法是指示 daemon 执行不同的行为。信号的默认动作是终止进程。

SIGVTALRM

当通过 ITIMER_VIRTUAL 标志创建的定时器到期时，setitimer()函数发送这个信号。第十章讨论定时器。

SIGWINCH

当进程的控制终端窗口大小改变时，内核向前台进程组的所有进程发送这个信号。默认情况下，进程忽略这个信号，但是如果进程希望知道控制终端窗口大小，可以选择捕获并处理这个信号。例如 top 程序捕获这个信号，试试运行 top 程序，并调整窗口大小，观察程序如何响应。

SIGXCPU

当进程超出处理器软限制时，内核抛出这个信号。内核会每秒持续抛出这个信号，直到进程退出，或者超出处理器硬限制。一旦硬限制达到，内核会发送 SIGKILL。

SIGXFSZ

当进程超出文件大小限制时，内核抛出这个信号。默认动作是终止进程，但是如果信号被捕获或忽略，那些可能导致超出文件大小限制的系统调用将返回-1，并设置 errno 为 EFBIG。

基本信号管理

信号管理最简单也最古老的接口是 signal()函数，由 ISO C90 标准定义，只标准化了最少的公共部分信号支持，这个系统调用非常基础。Linux 通过其它接口提供比 signal()更多的控制，我们在本章后面会详细讨论。由于 signal()最基本，也感谢 ISO C 的标准定义，它的使用很常见，我们首先讲解它：

```
#include <signal.h>
```

```
typedef void (*sighandler_t)(int);

sighandler_t signal (int signo, sighandler_t handler);
```

成功调用 signal()移除 signo 信号的当前动作，把它替换为 handler 指定的信号处理器。signo 是上一节讨论的信号中的一个，例如 SIGINT 或 SIGUSR1。由于进程不能捕获 SIGKILL 和 SIGSTOP，因此为它们设置信号处理器没有任何意义。

handler 函数必须返回 void，这是明智的，因为函数不知道会返回到程序的什么地方（和普通函数不一样）。处理函数有一个整型参数，它是被处理的信号标识符（例如 SIGUSR2），这允许一个处理函数处理多个信号。处理函数的原型如下：

```
void my_handler (int signo);
```

Linux 使用 typedef sighandler_t 来定义这个原型。其它 Unix 系统直接使用函数指针；有一些系统则二者都有，但不一定命名为 sighandler_t。关注可移植性的程序不应该直接引用类型定义。

当发送信号给已经注册信号处理器的进程时，内核暂停程序的正常指令执行流程，并调用信号处理器。处理器函数被传递信号的值，它就是提供给 signal()的 signo 参数。

你可以使用 signal()来指示内核对当前进程忽略指定的信号，或者重置信号为默认行为。这通过两个特殊的 handler 参数值来实现：

SIG_DFL

设置 signo 指定的信号为默认行为。例如 SIGPIPE，进程将终止。

SIG_IGN

忽略 signo 指定的信号。

signal()函数返回信号先前的行为，它可能是指向信号处理器的指针、SIG_DFL、SIG_IGN；出错时，函数返回 SIG_ERR，函数不设置 errno。

等待信号，任何信号

POSIX 定义了 pause()系统调用睡眠进程，直到进程接收到信号，信号要么被处理，要么终止进程。parse()对调试和编写演示性代码非常有用：

```
#include <unistd.h>

int pause (void);
```

pause()只有接收到被捕获的信号后才会返回，这种情况下信号被处理，pause()返回-1，并设置 errno 为 EINTR。如果内核抛出一个被忽略的信号，进程不会被唤醒。

在 Linux 内核中，pause()是最简单的系统调用之一，它仅仅执行两个动作。首先，内核把进程置为可中断睡眠状态；然后，内核调用 schedule()来调用 Linux 进程调度器找到另一个进程来执行。由于进程实际上并没有等待任何东西，内核不会唤醒它，除非进程接收到信号。所有这些只需要两行代码。

例子

我们来看几个简单的例子。第一个例子为 SIGINT 信号注册处理器，简单地打印一条信息并终止程序（正如 SIGINT 信号要做的一样）：

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

/* handler for SIGINT */
static void sigint_handler (int signo)
{
	/*
	 * Technically, you shouldn't use printf( ) in a
	 * signal handler, but it isn't the end of the
	 * world. I'll discuss why in the section
	 * "Reentrancy."
	 */
	printf ("Caught SIGINT!\n");
	exit (EXIT_SUCCESS);
}

int main (void)
{
	/*
	* Register sigint_handler as our signal handler
	* for SIGINT.
	*/
	if (signal (SIGINT, sigint_handler) == SIG_ERR) {
		fprintf (stderr, "Cannot handle SIGINT!\n");
		exit (EXIT_FAILURE);
	}

	for (;;)
		pause ( );

	return 0;
}
```

在下面例子中，我们给 SIGTERM 和 SIGINT 信号注册相同的处理器。同时重置 SIGPROF 的默认行为（终止程序），并忽略 SIGHUP 信号：

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

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

/* handler for SIGINT */
static void signal_handler (int signo)
{
	if (signo == SIGINT)
		printf ("Caught SIGINT!\n");
	else if (signo == SIGTERM)
		printf ("Caught SIGTERM!\n");
	else {
		/* this should never happen */
		fprintf (stderr, "Unexpected signal!\n");
		exit (EXIT_FAILURE);
	}

	exit (EXIT_SUCCESS);
}

int main (void)
{
	/*
	* Register signal_handler as our signal handler
	* for SIGINT.
	*/
	if (signal (SIGINT, signal_handler) == SIG_ERR) {
		fprintf (stderr, "Cannot handle SIGINT!\n");
		exit (EXIT_FAILURE);
	}

	/*
	* Register signal_handler as our signal handler
	* for SIGTERM.
	*/
	if (signal (SIGTERM, signal_handler) == SIG_ERR) {
		fprintf (stderr, "Cannot handle SIGTERM!\n");
		exit (EXIT_FAILURE);
	}

	/* Reset SIGPROF's behavior to the default. */
	if (signal (SIGPROF, SIG_DFL) == SIG_ERR) {
		fprintf (stderr, "Cannot reset SIGPROF!\n");
		exit (EXIT_FAILURE);
	}
```

```
        /* Ignore 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

当进程最早执行时，所有信号都重置为默认动作，除非父进程（执行新进程的那个）忽略某些信号，在这种情况下，新创建的进程也同样忽略这些信号。换一种说法，任何被父进程捕获的信号在子进程中都将被重置为默认动作，所有其它信号保持不变。由于新执行的进程没有共享父进程的地址空间，因此任何已注册的信号处理器都可能不存在。

进程执行的行为有一个值得注意的用法：当 shell 执行一个后台进程（或者当一个后台进程执行另一个后台进程时），新创建的进程应该忽略中断和退出信号。

因此，在 shell 执行一个后台进程之前，它应该设置 SIGINT 和 SIGQUIT 的动作为 SIG_IGN。处理这两个信号的程序，首先检查信号是否被忽略。例如：

```
/* handle SIGINT, but only if it isn't ignored */
if (signal (SIGINT, SIG_IGN) != SIG_IGN) {
        if (signal (SIGINT, sigint_handler) == SIG_ERR)
                fprintf (stderr, "Failed to handle SIGINT!\n");
}

/* handle SIGQUIT, but only if it isn't ignored */
if (signal (SIGQUIT, SIG_IGN) != SIG_IGN) {
        if (signal (SIGQUIT, sigquit_handler) == SIG_ERR)
                fprintf (stderr, "Failed to handle SIGQUIT!\n");
}
```

设置信号行为之前需要检查信号的行为突出了 signal()接口的不足。稍后我们会学习一个函数，它没有这个缺点。

正如你预料的，fork()与信号的行为不一样。当进程调用 fork()时，子进程继承父进程的信号，这同样也是有道理的，因为子进程和父进程共享地址空间，因此父进程的信号处理器在子进程中也存在。

映射信号数值到字符串

在之前的例子中，我们硬编码信号的名字。但是有时候你需要转换信号数值为信号名的

字符串描述，这会非常方便（甚至是必须）。有几个方法来完成这个任务，其中一个是从静态定义列表中取得字符串：

```
extern const char * const sys_siglist[];
```

sys_siglist 是一个字符串数组，保存了系统支持的所有信号名，使用信号数值索引。

另一个方法是使用 BSD 定义的 psignal()接口，它相当常见，因此 Linux 也支持它：

```
#include <signal.h>

void psignal (int signo, const char *msg);
```

调用 psignal()打印你提供的 msg 参数到 stderr，跟随一个冒号，一个空格，然后是 signo 指定的信号名。如果 signo 非法，打印信息也会提示信号非法。

一个更好的接口是 strsignal()。它没有被标准化，但是 Linux 和许多非 Linux 系统都支持它：

```
#define _GNU_SOURCE
#include <string.h>

char *strsignal (int signo);
```

调用 strsignal()返回一个字符串指针，指向 signo 表示的信号描述。如果 signo 非法，返回的描述通常也会说明这一点（有一些 Unix 系统可能返回 NULL）。函数返回的字符串只有在下一次调用 strsignal()之前才是合法的，因此这个函数不是线程安全的。

使用 sys_siglist 通常是你的最佳选择。使用这个方法，我们可以重写之前的信号处理器：

```
static void signal_handler (int signo)
{
    printf ("Caught %s\n", sys_siglist[signo]);
}
```

发送信号

kill()系统调用从一个进程向另一个进程发送信号，它是 kill 通用工具的基础：

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

int kill (pid_t pid, int signo);
```

在 kill()的正常用法中（pid 大于 0），它向 pid 标识的进程发送 signo 信号。

如果 pid 为 0，signo 将发送给调用进程的进程组中的所有进程。

如果 pid 为-1，signo 将发送给调用进程有权限发送信号的所有进程，除了进程本身和 init。我们在下一节会讨论发送信号的权限。

如果 pid 小于-1，signo 将发送给进程组-pid。

成功时 kill()返回 0。只要有一个信号被发送出去，调用就认为是成功的；失败时（没有信号发送）调用返回-1，并设置 errno 为以下值之一：

EINVAL

signo 指定的信号非法。

EPERM

调用进程缺乏足够的权限，来发送信号给请求的进程。

ESRCH

pid 表示的进程或进程组不存在，或者是僵尸进程。

权限

要向另一个进程发送信号，发送进程需要适当的权限。拥有 CAP_KILL 能力的进程（通常是 root 用户拥有）可以向任何进程发送信号。没有这个能力，发送进程的有效或者实际用户 ID 必须等于接收进程的实际或者已保存用户 ID。简单地说，用户只能向用户自己拥有的进程发送信号。

Unix 系统对 SIGCONT 信号定义了一个例外：进程可以向相同会话的所有其它进程发送这个信号。用户 ID 不需要匹配。

如果 signo 为 0——前面提到过的 null 信号——调用不会发送信号，但是它仍然会执行错误检查。这对于测试一个进程是否有合适的权限向指定的进程发送信号非常有用。

例子

下面代码向进程 1722 发送 SIGHUP 信号：

```
int ret;
ret = kill (1722, SIGHUP);
if (ret)
        perror ("kill");
```

这个代码片断与 kill 工具的如下调用等同：

```
$ kill -HUP 1722
```

要检查我们是否有权限向 1722 发送信号，可以这样做：

```
int ret;
ret = kill (1722, 0);
if (ret)
        ; /* we lack permission */
else
        ; /* we have permission */
```

向自己发送信号

进程使用 raise()函数向自己发送信号：

```
#include <signal.h>
```

int raise (int signo);

下面调用：

raise(signo);

等同于下面调用：

kill(getpid ( ), signo);

调用成功时返回 0；失败时返回非零值，函数不设置 errno。

向整个进程组发送信号

另一个便利的函数是 killpg()，使得向指定进程组的所有进程发送信号很简单，而不需要使用 kill()和进程组 ID 的负数：

#include <signal.h>

int killpg (int pgrp, int signo);

下面调用

killpg (pgrp, signo);

等同于下面调用：

kill (-pgrp, signo);

如果 pgrp 为 0，上面两个调用也是相同的，这种情况下 killpg()向调用进程组中的所有进程发送 signo 信号。

成功时 killpg()返回 0；失败时返回-1，并设置 errno 为以下值之一：

EINVAL

signo 指定的信号非法。

EPERM

调用进程缺乏足够的权限，来向请求的进程发送信号。

ESRCH

pgrp 指定的进程组不存在。

可重入

当内核抛出一个信号时，进程可能执行到任何地方。例如，它可能正在一个重要的操作过程中间，如果中断，会使进程处于不一致的状态（比如只更新数据结构的一半、仅仅执行部分计算）。进程甚至可能正在处理另一个信号。

当信号抛出时，信号处理器不知道进程代码当前执行到哪里；处理器可能在任何代码的中途运行。因此每个处理器都必须非常小心它执行的动作，以及它涉及的数据，这个非常重要。信号处理器必须不对进程中断时所做的事情做任何假设。特别地，处理器在修改全局数据时必须特别小心。通常，处理器不接触任何全局数据是一个好主意。不过在后面的小节中，我们仍然会讨论临时阻塞信号发送，允许安全地操作信号处理器与进程共享的数据。

系统调用和其它库函数又怎么样呢？如果进程正在写入文件或者分配内存的中途，而信

号处理器也向同一个文件写入或者同样调用 malloc()，情况又会怎样？或者如果进程正在调用一个使用静态缓冲区的函数，例如 strsignal()，这时有信号发送过来会怎么样？

有一些函数明显不是可重入的。如果程序正在执行一个非可重入函数，这时产生信号，而信号处理器然后又调用同一个非重入的函数，就会产生混乱。可重入函数是可由自身安全调用（或者相同进程的不同线程并发调用）的一种函数。如果想成为一个可重入函数，就不能操作静态数据，必须仅仅操作堆栈分配的数据，或者由调用方提供的数据，而且不能调用任何非可重入函数。

确保可重入的函数

当编写信号处理器时，你必须假设被中断进程处于执行非可重入函数中（或者其它任何可能）。因此信号处理器必须只使用可重入函数。

不同的标准规定了信号安全的函数列表（也就是可重入），这些函数可以在信号处理器中安全使用。最显著的是 POSIX.1-2003 和 Single UNIX Specification 规定的函数列表，这些函数在所有遵从标准的平台都确保可重入和信号安全。表 9-2 列出了所有函数。

表 9-2. 确保信号安全使用的可重入函数

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( )
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( )

许多其它函数也是安全的，但是 Linux 和其它 POSIX 兼容的系统只保证上面这些函数的可重入性。

信号集

本章后面我们将讨论的几个函数需要操作信号集，例如进程阻塞的信号集、或者进程的未决信号集。信号集操作函数管理这些信号集：

```
#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()初始化信号集 set，把它标记为空（信号集不包括任何信号）。sigfillset()初始化信号集 set，标记为满（包括所有信号）。两个函数都返回 0。你应该在进一步使用信号集之前，调用其中的一个函数初始化信号集。

sigaddset()添加 signo 为信号集 set 中，而 sigdelset()则从信号集 set 中删除 signo。两个函数在成功时都返回 0；出错时返回-1，并设置 errno 为错误代码 EINVAL，表示 signo 是一个非法的信号标识符。

如果 signo 在信号集 set 中，sigismember()函数返回 1，否则返回 0，出错时返回-1。出错的情况下设置 errno 为 EINVAL，表示 signo 非法。

更多信号集函数

前面的函数都由 POSIX 标准化，并且在任何现代 Unix 系统上都支持。Linux 也提供另外几个非标准的信号集函数：

```
#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 指定的信号集为空，sigisemptyset()函数返回 1，否则返回 0。

sigorset()把 left 和 right 信号集的联合（OR）放到 dest 信号集中。

sigandset()把 left 和 right 信号集的交集（AND）放到 dest 信号集中。两个函数成功时都返回 0；出错时返回-1，并设置 errno 为 EINVAL。

这些函数很有用，但是需要完全遵循 POSIX 标准的程序应该避免使用它们。

阻塞信号

前面我们讨论了可重入，和信号处理器异步运行带来的问题。我们讨论了不要在信号处理器中调用的函数，因为它们不是可重入的。

但是如果你的程序需要在信号处理器和程序其它地方共享数据，该怎么做呢？或者程序的某个执行过程不希望被中断，包括不被信号处理器中断。我们把程序的这种部分称为临界区，并且通过临时暂停信号发送来保护它们。这种信号就被称为已阻塞。任何被抛出的信号如果阻塞，在解除阻塞之前不会被处理。进程可以阻塞任意数量的信号；被阻塞的信号集就称为信号掩码。

POSIX 定义并且 Linux 实现了管理进程信号掩码的函数：

```
#include <signal.h>

int sigprocmask (int how,
                 const sigset_t *set,
                 sigset_t *oldset);
```

sigprocmask()的行为依赖于 how 的值，有如下标志：

SIG_SETMASK

调用进程的信号掩码变成 set。

SIG_BLOCK

set 中的信号添加到调用进程的信号掩码中。换句话说，进程的信号掩码变为当前掩码与 set 的并集(OR)。

SIG_UNBLOCK

set 中的信号将从调用进程的信号掩码中移除。换句话说，进程的信号掩码变为当前掩码与 set 求反的交集。对一个未阻塞的信号进行解除阻塞是非法的。

如果 oldset 非 NULL，函数把之前的信号集放在 oldset 中。

如果 set 为 NULL，函数忽略 how，并且不改变信号掩码，但是函数仍然会把信号掩码放到 oldset 中。换句话说，给 set 传递 NULL 值是获取当前信号掩码的方法。

成功时调用返回 0；失败时返回-1，并设置 errno 为 EINVAL，表示 how 非法；或者 EFAULT，表示 set 或 oldset 是无效指针。

阻塞 SIGKILL 和 SIGSTOP 是不允许的。sigprocmask()默默地忽略阻塞这两个信号的请求。

获取未决信号

当内核抛出一个阻塞信号时，它不会被发送，我们称之为未决信号。当一个未决信号解除阻塞时，内核把它传递给进程处理。

POSIX 定义了一个函数获得未决信号集：

```
#include <signal.h>

int sigpending (sigset_t *set);
```

成功调用 sigpending()把未决信号集放到 set 中，并返回 0；失败时返回-1，并设置 errno 为 EFAULT，表示 set 是无效指针。

等待信号集

POSIX 定义的第三个函数是 sigsuspend()，允许进程临时改变信号掩码，等待信号直到进程被信号终止，或者信号被处理：

```
#include <signal.h>

int sigsuspend (const sigset_t *set);
```

如果信号终止了进程，sigsuspend()不会返回。如果信号被抛出并处理，sigsuspend()在信号处理器结束执行之后返回-1，并设置 errno 为 EINTR。如果 set 是无效指针，errno 设置为 EFAULT。

sigsuspend()的一个常见用途是在程序执行临界区中获得已经到达并被阻塞的信号。进程首先使用 sigprocmask()阻塞信号集，在 oldset 中保存老的掩码。在退出临界区之后，进程调用 sigsuspend()，使用 oldset 作为参数。

高级信号管理

本章开头学习的 signal()函数非常基础。由于它是标准 C 库的一部分，因此对于操作系统的信号支持只能做最小的假设，函数只能提供信号管理功能的最小公分母。作为另一种选择，POSIX 标准化 sigaction()系统调用，提供更多的信号管理能力。你可以使用它在处理器执行时阻塞特定的信号，获得信号抛出时系统和进程的大量数据信息：

```
#include <signal.h>

int sigaction (int signo,
               const struct sigaction *act,
               struct sigaction *oldact);
```

调用 sigaction()改变 signo 指定的信号的行为，signo 可以是除 SIGKILL 和 SIGSTOP 之外的任何信号。如果 act 不为 NULL，系统调用改变信号的当前行为到 act 中。如果 oldact 不为 NULL，调用存储之前的（如果 act 为 NULL 则为当前）行为到 oldact 中。

sigaction 结构体允许信号的精确控制。结构体在头文件<sys/signal.h>中定义，由<signal.h>头文件引入：

```
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 域指定接收到信号的处理动作，和 signal()一样，这个域可以是 SIG_DFL，表示默认动作；也可以是 SIG_IGN，指示内核忽略该信号；也可以是一个信号处理器函数指针。函数的原型和 signal()使用的信号处理器原型相同：

```
void my_handler (int signo);
```

如果 sa_flags 中设置了 SA_SIGINFO，则 sa_sigaction 而不是 sa_handler 将成为信号处理函数。这个函数的原型稍微有些不同：

```
void my_handler (int signo, siginfo_t *si, void *ucontext);
```

函数接收信号数值作为第一个参数，第二个参数是 siginfo_t 结构体，第三个参数是 ucontext_t 结构体（转换为 void 指针），函数没有返回值。siginfo_t 结构体给信号处理器提供了大量信息；我们马上会讨论它。

注意在某些机器体系架构中（也可能有些 Unix 系统），sa_handler 和 sa_sigaction 在一个联合中，你最好不要对这两个域同时赋值。

sa_mask 域提供阻塞信号集，在信号处理器执行过程中这些信号将会被阻塞。这允许程序员在多个信号处理器间执行适当的可重入保护。当前正在被处理的信号同样也被阻塞，除非 sa_flags 中设置了 SA_NODEFER 标志。你不能阻塞 SIGKILL 和 SIGSTOP，sigaction()调用忽略 sa_mask 中的这两个信号。

sa_flags 域是多个标志的位掩码，改变 signo 信号的处理行为。我们已经说过了 SA_SIGINFO 和 SANODEFER 标志；其它 sa_flags 标志如下：

SA_NOCLDSTOP

如果 signo 是 SIGCHLD，这个标志指示系统在子进程停止或者继续时不提供通知。

SA_NOCLDWAIT

如果 signo 是 SIGCHLD，这个标志允许自动子进程回收：子进程在终止时不会转化为僵尸进程，父进程不需要（也不能）对它调用 wait()。请参考第五章对子进程、僵尸进程、和 wait()的讨论。

SA_NOMASK

这个标志已经废弃，与 POSIX 标准 SA_NODEFER 相同。使用 SA_NODEFER 代替 SA_NOMASK，但是在老的代码中你可能还会看见它。

SA_ONESHOT

这个标志已经废弃，与 POSIX 标准 SA_RESETHAND 相同。使用 SA_RESETHAND 代替这个标志，但是在老的代码中你仍然可能看见它。

SA_ONSTACK

这个标志指示系统在 alternative signal stack 中调用信号处理器，由 sigaltstack()提供。如果你不提供 alternative stack，则使用默认堆栈（也就是你不设置 SA_ONSTACK 标志时的

行为）。alternative signal stack 很少见，尽管在某些 pthread 应用中很有用（小线程堆栈可能被某些信号处理器使用）。本书不讨论 sigaltstack()。

SA_RESTART

这个标志允许重新启动被信号中断的系统调用，这是 BSD 风格。

SA_RESETHAND

这个标志允许“one-shot”模式。指定信号的行为在信号处理器返回后将重置为默认。

sa_restorer 域已经废弃，Linux 中不再使用。而且它也不是 POSIX 标准定义。假装没看见它，并且决不碰它。

sigaction()成功时返回 0；失败时返回-1，并设置 errno 为以下错误代码之一：

EFAULT

act 或 oldact 是无效指针。

EINVAL

signo 是非法信号、SIGKILL 或 SIGSTOP。

siginfo_t 结构体

siginfo_t 结构体同样也在<sys/signal.h>中定义，如下：

```
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 */
};
```

这个结构体包含大量传递给信号处理器的信息（如果你使用 sa_sigaction 代替 sa_sighandler 的话）。在现代计算中，许多人认为 Unix 信号模型是执行 IPC 是很糟糕的方法。可能问题就在于他们使用 signal()，而实际上应该使用 sigaction()加 SA_SIGINFO。sigaction_t 结构体为编写多功能信号程序打开了一扇门。

sigaction_t 结构体中有许多有用的数据，包括发送信号的进程信息，以及信号的原因。下面是每个域的详细描述：

si_signo

信号的信号数值。信号处理器的第一个参数也提供了这个信息（并且避免了指针解引用）。

si_errno

如果非 0，是信号相关联的错误代码。这个域对所有信号都有效。

si_code

解释为什么和进程从哪里收到信号(例如，kill())。我们在下一节会讨论所有可能的值，这个域对所有信号有效。

si_pid

对于 SIGCHLD，它是终止进程的 pid。

si_uid

对于 SIGCHLD，它是终止进程的拥有者 UID。

si_status

对于 SIGCHLD，它是终止进程的退出状态代码。

si_utime

对于 SIGCHLD，它是终止进程消耗的用户时间。

si_stime

对于 SIGCHLD，它是终止进程消耗的系统时间。

si_value

si_int 和 si_ptr 的联合。

si_int

对于通过 sigqueue()发送的信号（本章后面“携带 Payload 发送信号”一节），提供的 payload 为整型。

si_ptr

对于通过 sigqueue()发送的信号（本章后面“携带 Payload 发送信号”一节），提供的 payload 为 void 指针。

si_addr

对于 SIGBUS、SIGFPE、SIGILL、SIGSEGV、SIGTRAP，这个 void 指针包含错误的地址。例如 SIGSEGV，这个域包含内存访问违规的地址（因此经常是 NULL!）。

si_band

对于 SIGPOLL，它是 si_fd 文件描述符的 out-of-band 和优先权信息。

si_fd

对于 SIGPOLL，它是操作完成的文件描述符。

si_value、si_int 和 si_ptr 是特别复杂的主题，因为进程可以使用它们传递任何数据给另一个进程。因此你可以使用它们发送简单的整数或者指向数据结构的指针（注意如果进程间不共享地址空间，那传递指针没太大用处）。这些域会在下一节“携带 payload 发送信号”中讨论。

POSIX 只保证前三个域对所有信号有效，其它域只有处理适当的信号时才应该访问。例如只有信号是 SIGPOLL 时，你才能访问 si_fd 域。

si_code 的精彩世界

si_code 域指示信号的原因。对于用户发送的信号，这个域指示信号是怎样发送的；对于内核发送的信号，这个域指示信号为什么发送。

下列 si_code 值对所有信号都有效。它们指示怎样/为什么信号被发送：

SI_ASYNCIO

发送信号的原因是异步 I/O 完成（第五章）。

SI_KERNEL
信号由内核抛出。
SI_MESGQ
发送信号的原因是 POSIX 消息队列（本书不讨论）状态改变。
SI_QUEUE
信号由 sigqueue()发送（下一节）。
SI_TIMER
发送信号的原因是 POSIX 定时器到期（第十章）。
SI_TKILL
信号由 tkill()或 tgkill()发送。这些系统调用被线程库使用，本书未讲解。
SI_SIGIO
发送信号的原因是 SIGIO 排队。
SI_USER
信号由 kill()或 raise()发送。

以下 si_code 值只对 SIGBUS 有效。它们指示发生的硬件错误的类型：
BUS_ADRALN
进程引起对齐错误（参见第八章对齐的讨论）。
BUS_ADRERR
进程访问非法物理地址。
BUS_OBJERR
进程引发其它硬件错误。

对于 SIGCHLD，以下值指示子进程做了什么，导致产生信号发送给父进程。
CLD_CONTINUED
子进程停止后又继续执行。
CLD_DUMPED
子进程异常终止。
CLD_EXITED
子进程通过 exit()正常终止。
CLD_KILLED
子进程被杀死。
CLD_STOPPED
子进程停止。
CLD_TRAPPED
子进程击中陷阱。

以下值只对 SIGFPE 有效。它们解释发生的算术错误的类型：
FPE_FLTDIV
进程执行浮点操作，引起除以 0 错误。
FPE_FLTOVF
进程执行浮点操作，引起溢出。
FPE_FLTINV
进程执行非法浮点操作。

FPE_FLTRES
进程执行浮点操作，导致不精确或无效结果。
FPE_FLTSUB
进程执行浮点操作，导致超出范围的下标。
FPE_FLTUND
进程执行浮点操作，导致下溢。
FPE_INTDIV
进程执行整数操作，引起除以 0 错误。
FPE_INTOVF
进程执行整数操作，导致溢出。

以下 si_code 值只对 SIGILL 有效。它们解释非法指令执行的种类：
ILL_ILLADR
进程试图进入非法地址模式。
ILL_ILLOPC
进程试图执行非法操作码。
ILL_ILLOPN
进程试图执行非法操作数。
ILL_PRVOPC
进程试图执行特权操作码。
ILL_PRVREG
进程试图在特权寄存器中执行。
ILL_ILLTRP
进程试图进入非法陷阱。

对于上面所有值，si_addr 都指向违规操作的地址。

对于 SIGPOLL，以下值指示产生信号的 I/O 事件：
POLL_ERR
发生 I/O 错误。
POLL_HUP
设备挂起或 socket 断开。
POLL_IN
文件有数据可读取。
POLL_MSG
有可用消息。
POLL_OUT
文件能够被写入。
POLL_PRI
文件有高优先权数据可以读取。

以下代码只对 SIGSEGV 有效，描述非法内存访问的两种类型：
SEGV_ACCERR
进程以非法方式访问合法内存区域——也就是进程违反内存访问权限。

SEGV_MAPERR
进程访问非法内存区域。

对于上面这两个值，si_addr 包含内存地址。

对于 SIGTRAP，下面两个 si_code 值指示击中陷阱的类型：
TRAP_BRKPT
进程命中断点。
TRAP_TRACE
进程命中跟踪陷阱。

注意 si_code 是值域而不是位域。

携带 payload 发送信号

正如我们在前一节看到的，通过 SA_SIGINFO 标志注册的信号处理器，会被传递一个 siginfo_t 参数。这个结构体包含一个 si_value 域，它是可选的 payload，由信号产生者发送给信号接收者。

POSIX 定义了 sigqueue()函数，允许进程携带 payload 发送信号：

```
#include <signal.h>

int sigqueue (pid_t pid,
              int signo,
              const union sigval value);
```

sigqueue()和 kill()类似。成功时 signo 信号在 pid 标识的进程或进程组中排队，函数返回 0；信号的 payload 由 value 指定，它是一个整数和一个 void 指针的联合：

```
union sigval {
    int sival_int;
    void *sival_ptr;
};
```

失败时调用返回-1，并设置 errno 为以下值之一：
EINVAL
signo 指定的信号非法。
EPERM
调用进程缺乏足够的权限来发送信号给请求的进程。发送信号的权限和 kill()一样。
ESRCH
pid 指示的进程或进程组不存在，或者是僵尸进程。

和 kill()一样，你可以给 signo 传递 null 信号来测试权限。

例子

下面例子向 1722 进程发送 SIGUSR2 信号，payload 为整数值 404：

```
sigval value;
int ret;
value.sival_int = 404;

ret = sigqueue (1722, SIGUSR2, value);
if (ret)
      perror ("sigqueue");
```

如果 1722 进程通过 SA_SIGINFO 标志处理信号，它会发送 signo 设置为 SIGUSR2，si->si_int 设置为 404，并且 si->si_code 设置为 SI_QUEUE。

总结

信号在许多 Unix 程序员中拥有一个非常坏的名声。信号是古老、陈旧的内核向用户通信的一种机制，至多算是 IPC 的原始形式。在现代多线程编程和事件循环的世界里，信号通常都不合适。

无论如何，不管好坏，我们确实需要信号。信号是从内核接收许多通知的唯一方法（例如执行非法操作码错误通知）。此外，信号是 Unix（和 Linux）终止进程和管理父子进程关系的途径。因此，我们必须使用信号。

信号毁誉的一个主要原因是很难编写一个正确的对可重入安全的信号处理器。如果你保持处理器简单，并且仅仅使用表 9-2 列出的函数，它就是安全的。

信号的另一个问题是许多程序员仍然使用 signal()和 kill()，而不是 sigaction()和 sigqueue()，来进行信号的管理。正如最后两节所展示的，SA_SIGINFO 风格的信号功能要强大得多，而且富于表现力。尽管我自己不是信号的狂热爱好者——我很希望看到信号被基于文件描述符可 poll 机制代替，这实际上也是 Linux 内核未来版本正在考虑的事情，使用 Linux 的高级信号接口减轻了很多的痛苦（如果不是消除的话）。

第十章 时间

时间在现代操作系统中提供各种作用，许多程序都与时间相关。内核按三种不同的方式度量时间的流逝：

真实时间(Wall time)

这是真实世界的实际日期和时间——也就是你能在墙上钟表上读到的时间。进程使用 wall time 与用户交互，或者标记事件的时间戳。

进程时间

这是进程消耗的时间，要么直接在用户空间代码中，要么间接通过内核对进程的工作。进程测量和统计时需要关注这个时间——例如测量指定操作花费的时间。wall time 对于测量进程时间是错误的，因为 Linux 的多任务天性，某个操作的进程时间可能比 wall time 少得多。进程同样还可能花费大量周期在等待 I/O 上（特别是键盘输入）。

Monotonic 时间

这个时间是严格线性增长的。大多数操作系统（包括 Linux）使用系统的运行时间（从系统启动开始）。wall time 可以改变——例如用户设置它，或者系统持续地调整时间，并且由于闰秒等因素，可能引入额外的不精确。反过来系统的运行时间，是确定并且不可改变的时间。monotonic 时间最重要的方面不在于当前时间，而在于确保时间严格线性增长，因此对于计算两个时间差非常有用。

monotonic 时间适合用来计算相对时间，而 wall 时间适合测量绝对时间。

这三种时间测量可以被两种格式描述：

相对时间

这是一个基于某个基准的值，例如从现在开始的 5 秒，或者 10 分钟以前。

绝对时间

不依赖于基准的时间描述，例如 noon on 25 March 1968。

相对和绝对时间有各自的用处。进程可能需要在 500 毫秒后取消一个请求，每秒刷新屏幕 60 次，或者从某个操作开始后的 7 秒钟，所有这些都是相对时间的计算。反过来，日期程序可能保存用户的聚会为二月八号，文件系统在文件创建时会写入完整的日期和时间（而不是“5 秒之前”），用户的时钟显示 Gregorian 日期，而不是从系统启动到现在的秒数。

Unix 系统按 epoch 开始消逝的秒数来描述绝对时间，epoch 定义为 1970 年 1 月 1 号 00:00:00 UTC。UTC 大体上就是 GMT（格林威治）或者 Zulu 时间。这意味着在 Unix 中，即使是绝对时间，在某种程度上也是相对的。Unix 引入一个特殊的数据类型存储“自从 epoch 的秒数”，我们在下一节会讨论。

操作系统通过软件时间跟踪时间的前进，由内核维护软件时间。内核实例化一个周期性的定时器，称为系统定时器，按特定的频率跳动。当定时器周期结束，内核增加已消逝的时间一个单元，称为 tick 或 jiffy。已消逝时间的计数器被称为 jiffy 计数器。以前是 32 位的值，在 Linux2.6 内核它是 64 位的计数器。

在 Linux 中，系统定时器的频率称为 HZ（和处理器的描述相同）。HZ 的值是体系架构相关的，并且不是 Linux ABI 的一部分（也就是说程序不能依赖或者期望任何值）。历史上 X86 架构使用 100，意味着系统定时器每秒运行 100 次（也就是系统定时器的频率为 100 赫兹）。这样每个 jiffy 就是 0.01 秒(1/HZ 秒)。在 Linux2.6 内核发布版中，内核开发者增加 HZ 的值为

1000，每个 jiffy 的值为 0.001 秒。然而在 2.6.13 及之后的版本中，HZ 的值为 250，这样每个 jiffy 的值为 0.004 秒。对于 HZ 的值有一个内在的权衡：更高的值提供更高的分辨率，但是引入更高的时间开销。

尽管进程不应该依赖于具体的 HZ 值，POSIX 还是定义了一个机制在运行时获得系统的定时器频率：

```
long hz;

hz = sysconf (_SC_CLK_TCK);
if (hz == -1)
    perror ("sysconf"); /* should never occur */
```

当程序希望确定系统定时器的分辨率时很有用，但转换系统时间值时并不需要它，因为多数 POSIX 接口输出的时间已经是转换过的，或者相对于固定的频率，与 HZ 无关。固定频率和 HZ 不一样，它是系统 ABI 的组成部分：在 X86 中，固定频率值为 100。返回时钟滴答的 POSIX 函数使用 CLOCKS_PER_SEC 来描述固定频率。

有时候事件导致计算机关掉，有时候计算机未通电，但是在启动后，它仍然拥有正确的时间。这是因为多数计算机有一个电池供电的硬件时钟，在计算机关闭期间存储日期和时间。当内核启动时，它从硬件时钟初始化当前时间。同样，当用户关闭系统时，内核写入当前时间到硬件时钟。系统管理员可以通过 hwclock 命令同步硬件时钟。

Unix 系统管理时间消耗需要完成几个任务，进程只关心其中的几个：包括设置和获得当前 wall time，计算时间流逝，睡眠指定数量的时间，执行高精度时间测量，以及控制定时器。本章讲解所有这些时间相关的主题，我们从 Linux 描述时间的数据结构开始。

时间数据结构

随着 Unix 系统的进化，并实现自己的管理时间接口，有多个数据结构用来描述看上去简单的时间概念。这些数据结构从最简单的整数到许多域的结构体都有。我们在深入实际的接口之前先讲解它们。

原始描述

最简单的数据结构是 time_t，在头文件<time.h>中定义。time_t 的意图是成为一个不透明的类型，然而在多数 Unix 系统中（包括 Linux），time_t 只是 C long 类型的 typedef：

```
typedef long time_t;
```

time_t 描述从 epoch 到当前经过的秒数。“不能持续多久就会溢出”是大家典型的反应。实际上，它比你想像的要更长久，但在很多 Unix 系统未来仍然在使用的时候就会溢出。在 32 位 long 类型中，time_t 至多可以描述从 epoch 开始的 2,147,483,647 秒。这意味着我们会再次遇到 Y2K 问题（在 2038 年）。幸运的是，到 2038 年 1 月 18 号星期一 22:14:07，多数系统和软件将会是 64 位的。

微秒精度

time_t 的另一个问题是一秒钟能够发生太多事情了！timeval 结构体扩展了 time_t，增加微秒精度。头文件<sys/time.h>如下定义该结构体：

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

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

tv_sec 表示秒，tv_usec 表示微秒。令人莫名其妙的 suseconds_t 通常是整型的 typedef。

纳秒精度

Unix 并不满足于微秒精度，timespec 结构体提高到纳秒精度，头文件<time.h>中如下定义该结构体：

```
#include <time.h>

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

有了这个选择，接口更加倾向于使用纳秒而不是微秒。因此，自从引入 timespec 结构体，多数时间相关的接口都切换到 timespec，因此获得更高的精度。不过正如我们将看到的，一个很重要的函数仍然使用 timeval。

在实践中，由于系统定时器没有提供纳秒甚至微秒精度，这两个结构体通常都无法提供规定的精度。无论如何，在接口中使用尽可能高的精度是首选，这样就能获得系统能够提供的最高精度。

分解时间

我们后面要讨论的几个函数在 Unix 时间和字符串间转换，或者由程序创建指定日期的描述。C 标准提供 tm 结构体描述“分解”后的时间，以一种更加可读的方式。这个结构体同样在<time.h>中定义：

```
#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 结构体使得判断 time_t 值（例如 314159）是星期天还是星期六更加简单。从空间来看，它明显是一个糟糕的描述日期和时间的选择，但是它对于转换基于用户的值非常方便。

各个域如下：

tm_sec

分钟之后的秒数。这个值通常范围是[0, 59]，但是它可以高到 61 来表示 2 个闰秒。

tm_min

小时之后的分钟数。这个值的范围是[0, 59]。

tm_hour

午夜之后的小时数。这个值的范围是[0, 23]。

tm_mday

月份中的天数。这个值的范围是[0, 31]。POSIX 没有指定值 0；不过 Linux 使用它表示上个月的最后一天。

tm_mon

一月之后的月数。这个值的范围是[0, 11]。

tm_year

从 1900 年开始的年数。

tm_wday

从星期天开始的天数。这个值的范围是[0, 6]。

tm_yday

从 1 月 1 号开始的天数。这个值的范围是[0, 365]。

tm_isdst

一个特殊值指示是否日光节约时间(DST)对其它域表示的时间有效。如果它的值为正数，则 DST 有效；如果它的值为 0，则 DST 无效；如果它的值为负数，DST 的状态未知。

tm_gmtoff

当前时区相对格林威治时间的秒数偏移量。这个域只有当定义_BSD_SOURCE 时才存在。

tm_zone

当前时区的缩写——例如 EST。这个域只有当定义_BSD_SOURCE 时才存在。

进程时间类型

clock_t 类型描述时钟滴答。它是整数类型，通常是 long。根据不同的接口，clock_t 表示系统的实际定时器频率(HZ)或者 CLOCKS_PER_SEC。

POSIX 时钟

本章讨论的几个系统调用使用 POSIX 时钟，它是实现和描述时间源的标准。类型 clockid_t 描述一个特定的 POSIX 时钟，Linux 支持其中的四个：

CLOCK_MONOTONIC

一个单调递增的时钟，不能被任何进程设置。它描述从某个起点开始总共流逝的时间，例如系统启动。

CLOCK_PROCESS_CPUTIME_ID

处理器提供的对每个进程的高精度时钟。例如在 i386 架构中，这个时钟使用 timestamp counter(TSC)寄存器。

CLOCK_REALTIME

系统级的实际时间(wall time)时钟。设置这个时钟需要特殊的权限。

CLOCK_THREAD_CPUTIME_ID

类似于单个进程的时钟，但是它对进程中的每个线程唯一。

POSIX 定义了所有四个时间源，但是只强制 CLOCK_REAL_TIME。虽然 Linux 可靠地实现了所有四个时钟，可移植代码应该只依赖于 CLOCK_REAL_TIME。

时间源精度

POSIX 定义函数 clock_getres()来获得指定时间源的精度：

```
#include <time.h>

int clock_getres (clockid_t clock_id,
                  struct timespec *res);
```

成功调用 clock_getres()存储 clock_id 指定的时间源精度到 res 中，并返回 0；失败时函数返回-1，并设置 errno 为以下错误代码之一：

EFAULT

res 是无效指针。

EINVAL

clock_id 不是该系统的有效时间源。

下面例子输出前面讨论的四个时间源的精度：

```
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 系统中，得到类似如下输出：

```
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 纳秒是 4 毫秒，也就是 0.004 秒。因此 0.004 秒是 HZ 为 250 的 x86 系统的精度，正如我们在本章第一节中讨论的那样。CLOCK_REALTIME 和 CLOCK_MONOTONIC 都与 jiffy 捆绑，并且由系统定时器提供精度。反过来，CLOCK_PROCESS_CPUTIME_ID 和 CLOCK_THREAD_CPUTIME_ID 使用更高精度的时间源 TSC（在 x86 机器上），它提供纳秒级精度。

在 Linux（以及多数 Unix 系统）中，所有使用 POSIX 时钟的函数都需要将目标文件链接 librt。例如把前面代码编译为可执行程序，你需要使用如下命令：

```
$ gcc -Wall -W -O2  –lrt -g -o snippet snippet.c
```

获得当前日期和时间

应用程序有几个理由需要获得当前日期和时间：向用户显示、计算相对或流逝时间、事件时间戳等等。历史最悠久也最简单的获取当前时间的方法是 time()函数：

```
#include <time.h>

time_t time (time_t *t);
```

调用 time()返回当前时间，描述从 epoch 到现在经过的秒数。如果参数 t 不为 NULL，函数还将当前时间写入到所提供的指针。

出错时函数返回-1(转换为 time_t)，并适当地设置 errno。唯一可能的错误是 EFAULT，表示 t 是无效指针。

例如：

```
time_t t;

printf ("current time: %ld\n", (long) time (&t));
printf ("the same value: %ld\n", (long) t);
```

一种幼稚的时间表示方法

time_t 描述“自从 epoch 经过的秒数”，它并不是实际经过的秒数。Unix 假设闰年是能被 4 整队的所有年份，并且完全忽略闰秒。time_t 描述的时间点并不那么精确，但是始终统一。

更好的接口

函数 gettimeofday()扩展 time()，提供微秒精度：

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

int gettimeofday (struct timeval *tv,
                  struct timezone *tz);
```

成功调用 gettimeofday()把当前时间存储在 tv 指向的 timeval 结构体中，并返回 0；结构体 timezone 和参数 tz 已经废弃，两个都不应该使用，总是给 tz 传递 NULL。

失败时函数返回-1，并设置 errno 为 EFAULT，这是唯一可能的错误，表示 tv 或 tz 是无效指针。

例如：

```
struct timeval tv;
int ret;

ret = gettimeofday (&tv, NULL);
if (ret)
    perror ("gettimeofday");
else
    printf ("seconds=%ld useconds=%ld\n",
            (long) tv.sec, (long) tv.usec);
```

timezone 结构体由于内核并不管理时区而废弃，并且 glibc 拒绝使用 timezone 结构体的 tz_dsttime 域。我们在后面的小节会讨论如何操作时区。

高级接口

POSIX 提供 clock_gettime()接口来获得指定时间源的时间。不过更有用的是，这个函数可以获得纳秒精度：

```
#include <time.h>

int clock_gettime (clockid_t clock_id,
                   struct timespec *ts);
```

成功时调用返回 0，并存储 clock_id 指定的时间源的当前时间到 ts 中；失败时函数返回-1，并设置 errno 为以下值之一：

EFAULT

ts 是无效指针。

EINVAL

clock_id 在该系统中是无效时间源。

下面例子获得所有四个标准时间源的当前时间：

```
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);
}
```

获得进程时间

times()系统调用获得当前运行进程和它的子进程的进程时间（时钟滴答）：

```
#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);
```

调用成功时把当前进程和子进程消耗的进程时间填充到 buf 指向的 tms 结构体。报告的时间分为用户和系统时间。用户时间是在用户空间执行代码花费的时间；系统时间则是执行内核空间代码花费的时间（例如系统调用、或者页面错误）。只有在子进程终止，并且父进程对子进程调用 waitpid()之后才会包含子进程的进程时间。调用返回单调递增的时钟滴答数，从任意的起点开始算起。以前这个引用点是系统启动，因此 times()函数返回系统的启动时间。但是现在引用点变为系统启动前大约 429 百万秒。内核开发者对实现的改变，是为了捕获内核代码不能处理系统启动时间 wrap around 并重新归零。因此函数返回的绝对数值无意

义，再次调用之间的相对值才有价值。

失败时函数返回-1，并适当地设置 errno。在 Linux 中，唯一可能的错误代码是 EFAULT，表示 buf 是无效指针。

设置当前日期和时间

前一节讲的是如何获取时间，应用有时候同样也需要设置当前日期和时间。这通常都使用专门的工具来实现，例如 date。

相对 time()的时间设置函数是 stime()：

```
#define _SVID_SOURCE
#include <time.h>

int stime (time_t *t);
```

成功调用 stime()设置系统时间为 t 指定的值，并返回 0。调用进程必须拥有 CAP_SYS_TIME 能力。通常只有 root 用户才有这个能力。

失败时函数返回-1，并设置 errno 为 EFAULT，表示 t 是无效指针；或者 EPERM，表示调用进程没有 CAP_SYS_TIME 能力。

使用非常简单：

```
time_t t = 1;
int ret;

/* set time to one second after the epoch */
ret = stime (&t);
if (ret)
    perror ("stime");
```

我们会在后面小节讲解转换 time_t 为可读形式的接口。

精确地设置时间

gettimeofday()对应的函数是 settimeofday()：

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

int settimeofday (const struct timeval *tv ,
                  const struct timezone *tz);
```

成功调用 settimeofday()设置系统时间为 tv 指定的值，并返回 0。和 gettimeofday()一样，给 tz 传递 NULL 是最佳实践；失败时调用返回-1，并设置 errno 为以下值之一：

EFAULT

tv 或 tz 指向无效内存区域。

EINVAL

所提供的结构体的某个域无效。

EPERM

调用进程缺乏 CAP_SYS_TIME 能力。

下面例子设置当前时间为 1979 年 12 月中的星期六：

```
struct timeval tv = { .tv_sec = 31415926,
                      .tv_usec = 27182818 };
int ret;

ret = settimeofday (&tv, NULL);
if (ret)
    perror ("settimeofday");
```

设置时间的高级接口

正如 clock_gettime()提高了 gettimeofday()的精度，clock_settime()也废弃了 settimeofday()：

```
#include <time.h>

int clock_settime (clockid_t clock_id,
                   const struct timespec *ts);
```

成功时调用返回 0，并且 clock_id 指定的时间源被设置为 ts 指定的时间；失败时调用返回-1，并设置 errno 为以下值之一：

EFAULT

ts 是无效指针。

EINVAL

clock_id 在该系统中是无效时间源。

EPERM

进程缺乏设置指定时间源需要的权限，或者指定的时间源不能被设置。

在多数系统中，唯一能够设置的时间源是 CLOCK_REALTIME。因此，这个函数比 settimeofday()的唯一优点是它提供纳秒精度（还有它不用处理毫无用处的 timezone 结构体）。

Playing with Time

Unix 系统和 C 语言提供一组函数来转换分解时间（时间的 ASCII 字符描述）和 time_t。asctime()转换 tm 结构体为 ASCII 字符串：

```
#include <time.h>
char * asctime (const struct tm *tm);
char * asctime_r (const struct tm *tm, char *buf);
```

第一个函数返回静态分配字符串的指针。随后调用任何时间函数都可能覆盖这个字符串；因此 asctime()不是线程安全的。

多线程程序（以及憎恨糟糕接口设计的程序员）应该使用 asctime_r()。这个函数使用

buf 提供的字符串，而不是静态分配字符串，buf 的长度最少必须为 26。

出错时两个函数都返回 NULL。

mktime()函数同样转换 tm 结构体，不过它把 tm 转换为 time_t：

```
#include <time.h>
time_t mktime (struct tm *tm);
```

mktime()还通过 tzset()根据 tm 设置时区。出错时函数返回-1（类型转换为 time_t）。

ctime()转换 time_t 为 ASCII 描述：

```
#include <time.h>
char * ctime (const time_t *timep);
char * ctime_r (const time_t *timep, char *buf);
```

失败时函数返回 NULL，例如：

```
time_t t = time (NULL);
printf ("the time a mere line ago: %s", ctime (&t));
```

注意代码没有添加换行。ctime()在返回字符串后面自动添加换行符，这点有时候可能不太方便。

和 asctime()一样，ctime()返回静态字符串指针。它不是线程安全的，多线程程序应该使用 ctime_r()，它对 buf 提供的缓冲区进行操作。缓冲区至少必须 26 个字符。

gmtime()转换 time_t 为 tm 结构体，按 UTC 时区表示：

```
#include <time.h>
struct tm * gmtime (const time_t *timep);
struct tm * gmtime_r (const time_t *timep, struct tm *result);
```

失败时函数返回 NULL。

gmtime()再次返回静态分配的结构体，因此不是线程安全的。多线程程序应该使用 gmtime_r()，它对 result 提供的缓冲区进行操作。

localtime()和 localtime_r()分别执行 gmtime()和 gmtime_r()相同的功能，但是它按指定的用户时区转换 time_t：

```
#include <time.h>
struct tm * localtime (const time_t *timep);
struct tm * localtime_r (const time_t *timep, struct tm *result);
```

和 mktime()一样，调用 localtime()同时调用 tzset()，并初始化时区。localtime_r()是否执行这个步骤是未定义的。

difftime()返回两个 time_t 值之间的秒数差，转换为 double：

```
#include <time.h>
double difftime (time_t time1, time_t time0);
```

在所有 POSIX 系统中，time_t 都是算术类型，并且 difftime()等同于下面代码，忽略相减时的溢出检查：

```
(double) (time1 - time0)
```

在 Linux 中，由于 time_t 是整数类型，因此没有必要转换为 double。不过为了保持可移植性，还是应该使用 difftime()。

调整系统时钟

wall time 巨大和突然的跳跃可能会破坏依赖于绝对时间的应用的操作。考虑 make 的例子，它根据一个 Makefile 文件来构建软件项目。每次调用 make 不会重建整个源代码树，如果它这样做，那大项目改变一个文件可能就需要几个小时的重新构建。相反，make 查看源代码文件的修改时间戳，并与相应的目标文件对比。如果源代码文件（或者它需要的任何文件）比目标文件更新，make 就会重新编译该源文件为目标文件。如果源文件不比目标文件新，则不进行任何动作。

考虑下如果用户发现自己的时钟关闭了几个小时，并使用 date 更新系统时钟。然后用户更新并保存源文件 wolf.c，我们就有麻烦了。如果用户向后调整当前时间，wolf.c 看上去会比 wolf.o 更老（尽管事实不是这样），于是不会发生重新构建。

为了防止类似的问题，Unix 提供 adjtime()函数，它逐渐地向指定的方向调整当前时间。它的意图在于支持网络时间协议(NTP)daemon，NTP 经常性地纠正时间偏差，使用 adjtime()最小化它们对系统的影响：

```
#define _BSD_SOURCE
#include <sys/time.h>
int adjtime (const struct timeval *delta,
             struct timeval *olddelta);
```

成功调用 adjtime()指示内核慢慢地调整 delta 指定的时间量，并返回 0。如果 delta 指定的时间是正数，内核会加快系统时钟直到完全纠正时间。如果 delta 指定的时间为负数，内核会减慢系统时钟直到完全纠正时间。内核通过这种调整，保证时钟总是单调递增，而不会产生时间的剧变。即使是负数的 delta 值，调整也不会使时钟向后移动；相反，时钟会减慢直到系统时钟等于正确的时间。

如果 delta 不是 NULL，内核会停止处理任何之前已注册的纠正。但是已经做的那部分纠正则保留。如果 olddelta 不是 NULL，任何之前注册并尚未应用的纠正会写入到所提供的 timeval 结构体。传递 NULL 给 delta，以及合法的 olddelta 获得所有正在进行的纠正。

adjtime()应用的纠正可能非常小——理想的使用场景是 NTP，如前所述，NTP 应用小的纠正（几秒）。Linux 维持最小和最大的纠正，往不同方向的几千秒。

出错时 adjtime()返回-1，并设置 errno 为以下值之一：

EFAULT

delta 或 olddelta 是无效指针。

EINVAL

delta 指定的调整太大或太小。

EPERM

调用进程没有 CAP_SYS_TIME 能力。

RFC1305 定义了一个比 adjtime()的逐渐调整方式更加强大和复杂的时钟调整算法。Linux 通过 adjtimex()系统调用提供了这个算法的实现：

```
#include <sys/timex.h>
int adjtimex (struct timex *adj);
```

调用 adjtimex()读取内核时间相关的参数到 adj 指向的 timex 结构体中。根据结构体中的 modes 域，系统调用可能额外地设置适当的参数。

头文件<sys/timex.h>定义了 timex 结构体：

```
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 域是以下标志的按位 OR：

ADJ_OFFSET

通过 offset 设置时间偏移。

ADJ_FREQUENCY

通过 freq 设置频率偏移。

ADJ_MAXERROR

通过 maxerror 设置最大误差。

ADJ_ESTERROR

通过 esterror 设置估计误差

ADJ_STATUS

通过 status 设置时钟状态。

ADJ_TIMECONST

通过 constant 设置 phase-locked loop(PLL)时间常量。

ADJ_TICK

通过 tick 设置滴答值。

ADJ_OFFSET_SINGLESHOT

通过 offset 设置偏移，使用类似 adjtime()的简单算法。

如果 modes 为 0，则不设置任何值。只有具有 CAP_SYS_TIME 能力的用户才能提供非 0 的 modes 值；任何用户都可以为 modes 提供 0，来获得所有参数，但是不设置它们。

成功时 adjtimex()返回当前时钟状态，它是以下值之一：

TIME_OK

时钟已同步。

TIME_INS
　　将插入一个闰秒。
TIME_DEL
　　将删除一个闰秒。
TIME_OOP
　　闰秒正在进行。
TIME_WAIT
　　刚刚发生闰秒。
TIME_BAD
　　时钟不同步。

　　失败时 adjtimex()返回-1，并设置 errno 为以下错误代码之一：
EFAULT
　　adj 是无效指针。
EINVAL
　　modes、offset、tick 无效。
EPERM
　　modes 非 0，但是调用进程不具有 CAP_SYS_TIME 能力。

　　adjtimex()系统调用是 Linux 特有的。关注可移植性的应用应该使用 adjtime()。
　　RFC1305 定义了一个相当复杂的算法，因此对 adjtimex()的完整讨论超出了本书的范围。更多的信息请参考 RFC。

睡眠与等待

　　一些函数允许进程睡眠（暂停执行）指定的时间。第一个是 sleep()，把调用进程睡眠 seconds 指定的秒数：

```
#include <unistd.h>

unsigned int sleep (unsigned int seconds);
```

　　调用返回没有睡眠的秒数。因此调用成功时返回 0，但是函数可能返回 0 和 seconds 之间的其它值（例如信号中断了睡眠）。sleep()函数不设置 errno。多数用户并不关心 sleep()实际睡眠了多久，因此，不要检查 sleep()的返回值：

```
sleep (7); /* sleep seven seconds */
```

　　如果确实需要睡眠指定时间，你可以继续以返回值调用 sleep()，直到返回 0：

```
unsigned int s = 5;

/* sleep five seconds: no ifs, ands, or buts about it */
while ((s = sleep (s)))
    ;
```

微秒精度睡眠

以整秒的粒度来睡眠是相当差劲的。在现代系统中一秒简直就是永恒，因此程序通常希望以更高精度来睡眠，使用 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);
```

成功调用 usleep()睡眠调用进程 usec 微秒。不幸的是，BSD 和 SUS 规范对函数的原型定义不一致。BSD 接受一个 unsigned long，并且没有返回值；而 SUS 则定义 usleep()接受一个 useconds_t 类型，并且返回 int。如果定义了_XOPEN_SOURCE，Linux 遵循 SUS 规范；如果_XOPEN_SOURCE 没有定义，或者设置为小于 500，则 Linux 跟随 BSD。

SUS 的 usleep()在成功时返回 0；失败时返回-1。合法的 errno 值是 EINTR，表示睡眠被信号中断；或者 EINVAL，表示 usecs 太大（在 Linux 中，useconds_t 类型的整个范围都是合法的，因此这个错误不会发生）。

根据规范，useconds_t 类型是无符号整数，它的值至少能保存 1,000,000。

由于这两个函数原型之间的差异，而且某些 Unix 系统可能只支持其中一个，因此在代码中显式地使用 useconds_t 类型是不明智的。要获得最大的可移植性，可以假设参数是 unsigned int，并且不依赖于 usleep()的返回值：

```
void usleep (unsigned int usec);
```

使用如下：

```
unsigned int usecs = 200;
usleep (usecs);
```

这样对函数的两个版本都适用，而且检查错误仍然是可以的：

```
errno = 0;
usleep (1000);
if (errno)
    perror ("usleep");
```

不过多数程序并不关心 usleep()是否返回错误。

纳秒精度睡眠

Linux 不推荐使用 usleep()函数，用 nanosleep()代替了它，提供纳秒精度，而且接口更加整洁统一：

```
#define _POSIX_C_SOURCE 199309
#include <time.h>
```

int nanosleep (const struct timespec *req,
struct timespec *rem);

成功调用 nanosleep()睡眠调用进程 req 指定的时间，并返回 0；出错时返回-1，并适当地设置 errno。如果信号中断了睡眠，调用会在到达睡眠指定时间前返回。这种情况下，nanosleep()返回-1，并设置 errno 为 EINTR。如果 rem 不是 NULL，函数把睡眠剩余的时间存放在 rem 中。程序可以重新发起调用，传递 rem 给 req 参数。

下面是其它可能的 errno 值：

EFAULT

req 或 rem 是无效指针。

EINVAL

req 的某个域非法。

使用很简单：

```
struct timespec req = { .tv_sec = 0,
                        .tv_nsec = 200 };
/* sleep for 200 ns */
ret = nanosleep (&req, NULL);
if (ret)
    perror ("nanosleep");
```

下面例子使用第二个参数在中断的情况下继续睡眠：

```
struct timespec req = { .tv_sec = 0,
                        .tv_nsec = 1369 };
struct timespec rem;
int ret;

/* sleep for 1369 ns */
retry:
ret = nanosleep (&req, &rem);
if (ret) {
    if (errno == EINTR) {
        /* retry, with the provided time remaining */
        req.tv_sec = rem.tv_sec;
        req.tv_nsec = rem.tv_nsec;
        goto retry;
    }
    perror ("nanosleep");
}
```

最后是持续睡眠的另外一种实现方法（也许更加高效，但是不那么可读）：

```
struct timespec req = { .tv_sec = 1,
                        .tv_nsec = 0 };
struct timespec rem, *a = &req, *b = &rem;
```

```
/* sleep for 1s */
while (nanosleep (a, b) && errno == EINTR) {
        struct timespec *tmp = a;
        a = b;
        b = tmp;
}
```

nanosleep()比 sleep()和 usleep()有几个优点：

- 纳秒比秒和微秒精度更高。
- 由 POSIX.1b 标准化。
- 不是通过信号实现（信号实现的缺点在后面会讨论）。

尽管 Linux 不推荐使用另外两个，但多数程序仍然喜欢使用 usleep()而不是 nanosleep()——值得庆幸的是，越来越少的程序使用 sleep()。由于 nanosleep()是 POSIX 标准，并且没有使用信号，新的程序应该使用它（或者下一节讨论的接口），而不是 sleep()和 usleep()。

高级睡眠方法

除了我们已经学习过的几个睡眠函数，POSIX 时钟家族还定义了一个最高级的睡眠接口：

```
#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()。实际上下面调用：

```
ret = nanosleep (&req, &rem);
```

等同于这个调用：

```
ret = clock_nanosleep (CLOCK_REALTIME, 0, &req, &rem);
```

区别在于 clock_id 和 flags 参数。clock_id 指定度量的时间源。多数时间源是合法的，但是你不能指定调用进程的 CPU 时钟（例如 CLOCK_PROCESS_CPUTIME_ID）；这样做没有意义，因为调用暂停进程的执行，因此进程时间实际上已经停止增长。

指定哪个时间源依赖于你睡眠的目的。如果睡眠直到某个绝对时间值，CLOCK_REALTIME 最适合；如果睡眠相对数量时间，CLOCK_MONOTONIC 无疑是最佳选择。

flags 参数是 TIMER_ABSTIME 或 0。如果是 TIMER_ABSTIME，req 指定的值就是绝对值，而不是相对值。这解决了一个潜在的竞争条件。要解释这个参数的作用，假设进程在时间 T0，希望睡眠直到时间 T1。在 T0 时进程调用 clock_gettime()获得当前时间(T0)。然后用 T1 减去 T0，获得 Y 并传递给 clock_nanosleep()。但是在获得当前时间，和进程开始睡眠之间会有一定的时间花费。更坏的情况是，如果进程刚好被调离 CPU、产生页面错误、或者其它类似的事情发生。在获得当前时间、计算时间差、和实际睡眠之间总是存在潜在的竞争条件。

TIMER_ABSTIME 标志解除了这个竞争条件，允许进程直接指定 T1。内核暂停进程直到指定的时间源到达 T1。如果指定的时间源当前已经超过了 T1，调用立即返回。

让我们同时看一下相对和绝对睡眠。下面例子睡眠 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");
```

反过来，下面例子睡眠直到到达指定的绝对时间——它就是 clock_gettime()指定的时间源 CLOCK_MONOTONIC 返回时间之后的一秒钟：

```
struct timespec ts;
int ret;

/* we want to sleep until one second from NOW */
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");
```

多数程序只需要相对睡眠，因为它们的睡眠不需要非常精确。但是某些实时进程有非常精确的时间要求，需要绝对睡眠来避免潜在的竞争条件危害。

可移植睡眠方法

回忆我们在第二章讲过的 select()：

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

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

在第二章我们提到，select()提供一种可移植的微秒精度睡眠方式。在很长一段时间里，可移植的 Unix 程序都不得不使用 sleep()来实现睡眠：usleep()不是特别广泛可用；nanosleep()尚未被编写。开发者发现给 select()参数 n 传递 0，其它三个 fd_set 都传递 NULL，指定 timeout

作为睡眠时间，是可移植而且高效的进程睡眠方法：

```
struct timeval tv = { .tv_sec = 0,
                      .tv_usec = 757 };
/* sleep for 757 us */
select (0, NULL, NULL, NULL, &tv);
```

如果需要与老的 Unix 系统保持可移植，使用 select()可能是你的最佳选择。

Overruns

本节讨论的所有接口都保证至少睡眠请求的时间（除非出错时提前返回）。它们不会在请求的时间过去前返回成功。不过比请求时间睡眠更久是可能的。

这个现象的原因可能是简单的调度行为——请求时间过去时，内核可能及时地唤醒进程，但调度器却可能选择另一个进程运行。

但是还存在另一个潜在的更加危害的情况：定时器 overrun。在定时器的粒度比请求时间更粗糙时就会发生这种情况。例如假设系统定时器滴答为 10ms，而进程请求睡眠 1ms。系统只能按 10ms 间隔来测量时间，以及响应时间相关的事件（例如唤醒进程）。如果进程发起睡眠请求时，定时器刚好离时钟滴答 1ms，那所有事情都会很好——1ms 后，请求的睡眠时间过去，内核可以唤醒进程；但是如果定时器正好在进程请求睡眠时命中，那么在 10ms 内将不会再有定时器滴答，结果是进程将睡眠额外的 9ms！也就是会有九个 1ms 的 overrun。平均来说，时间为 X 的定时器会有 X/2 的 overrun 比率。

使用高精度的时间源（例如 POSIX 时钟所提供的），以及更高的 HZ 值，能够最小化定时器 overrun。

睡眠的替代选择

如果可能，你应该尽量避免睡眠。通常你不能完全避免，这也是可以接受的——特别是你的代码睡眠少于 1 秒的情况。但是加入睡眠的代码，如果只是为了“忙等(busy-wait)”事件，通常是糟糕的设计。代码阻塞在某个文件描述符，允许内核处理睡眠和唤醒进程，是更好的设计。相对于进程在循环中睡眠等待事件到来，内核可以直接阻塞进程的执行，并在需要时及时地唤醒它。

定时器

定时器提供一种机制，当指定的时间过去时通知进程。在定时器期满之前的时间数量称为延迟，或满期。定时器期满时内核怎样通知进程依赖于具体的定时器。Linux 提供多种类型。我们将学习所有方式。

有几个理由使定时器非常有用。常见的例子如每秒刷新屏幕 60 次、如果 500 毫秒之后事务仍然没有完成则取消事务等等。

简单的 Alarm

alarm()是最简单的定时器接口：

```
#include <unistd.h>

unsigned int alarm (unsigned int seconds);
```

调用这个函数在 seconds 秒实时时间过去后，向调用进程发送 SIGALRM 信号。如果之前预定的信号未决，调用取消 alarm，并用新请求的 alarm 替换它，然后返回前一个 alarm 剩余的秒数。如果 seconds 为 0，之前的 alarm（如果有）被取消，并且不会有新的 alarm 预定。

成功使用这个函数需要对 SIGALRM 信号注册处理器（信号和处理器在前一章讨论）。下面代码片断注册 SIGALRM 处理器 alarm_handler()，并设置一个 5 秒的 alarm：

```
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

时间间隔定时器最早在 4.2BSD 中出现，现在已经由 POSIX 标准化，提供比 alarm()更多的控制：

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

int getitimer ( int which,
                struct itimerval *value);
int setitimer ( int which,
                const struct itimerval *value,
                struct itimerval *ovalue);
```

时间间隔定时器和 alarm()的操作类似，但是可以自动地重新装备自己，并且可以按以下三种不同模式进行：

ITIMER_REAL

计量实时时间。当指定数量的实时时间过去时，内核向进程发送 SIGALRM 信号。

ITIMER_VIRTUAL

只有当进程的用户空间代码执行时才增加。当指定数量的进程时间过去时，内核向进程

发送 SIGVTALRM。

ITIMER_PROF

当进程代码执行、和内核为了进程而执行代码时（例如完成系统调用）增加。当指定数量的时间过去时，内核向进程发送 SIGPROF 信号。这个模式通常加上了 ITIMER_VIRTUAL，因此程序可以分别计量进程的用户和内核时间。

ITIMER_REAL 和 alarm()计量相同的时间；另外两个模式对于分析很有用。

itimerval 结构体允许用户指定直到定时器过期的时间数量，以及期满时间，在定时器到期时可以重新装备它。

```
struct itimerval {
    struct timeval it_interval; /* next value */
    struct timeval it_value; /* current value */
};
```

回忆我们之前讲过的 timeval 结构体，提供微秒精度：

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

setitimer()装备一个 which 类型的定时器，it_value 指定期满时间。一旦 it_value 指定的时间过去，内核将以 it_interval 提供的时间重新装备定时器。因此，it_value 是当前定时器剩余的时间。一旦 it_value 到达 0，它将被设置为 it_interval。如果定时器到期并且 it_interval 为 0，定时器不会重装。类似的，如果定时器的 it_value 设置为 0，定时器将停止而不重装。

如果 ovalue 不为 NULL，which 类型的间隔定时器之前的值将返回。

getitimer()返回 which 类型定时器的当前值。

两个函数在成功时都返回 0；出错返回-1，并设置 errno 为以下值之一：

EFAULT

value 或 ovalue 是无效指针。

EINVAL

which 不是合法的间隔定时器类型。

下面代码设置一个 SIGALRM 信号处理器，然后以 5 秒初始过期时间，以及 1 秒的后继间隔装备定时器：

```
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 ( );
    }
```

有一些 Unix 系统通过 SIGALRM 实现 sleep()和 usleep()，而 alarm()和 setitimer()明显也使用了 SIGALRM。因此，程序员必须特别小心不要重叠这些函数的调用，否则结果是未定义的。对于简短的等待，程序员应该使用 nanosleep()，POSIX 规定它不使用信号。对于定时器，程序员应该使用 setitimer()或 alarm()。

高级定时器

最强大的定时器接口，无疑来自 POSIX 时钟家族。

在 POSIX 时钟定时器中，实例化、初始化、和最终的删除定时器被分离到三个不同的函数：time_create()创建定时器、timer_settime()初始化定时器、timer_delete()销毁定时器。

POSIX 时钟定时器接口无疑是最高级，但也是最新（最低可移植性），使用最复杂的接口。如果简单或可移植是主要因素，setitimer()很可能是更好的选择。

创建定时器

使用 timer_create()创建定时器：

```
#include <signal.h>
#include <time.h>

int timer_create (clockid_t clockid,
                  struct sigevent *evp,
                  timer_t *timerid);
```

成功调用 timer_create()创建一个新的定时器，与 POSIX 时钟 clockid 关联，并存储一个唯一的定时器标识到 timerid 中，然后返回 0。这个调用仅仅设置运行定时器的条件，在定时器被装备之前不会发生任何事情，下一节讨论装备定时器。

下面例子以 CLOCK_PROCESS_CPUTIME_ID 时钟创建一个新的定时器，并存储定时器 ID 到 timer 中：

```
timer_t timer;
int ret;
```

```
ret = timer_create (CLOCK_PROCESS_CPUTIME_ID,
                    NULL,
                    &timer);
if (ret)
    perror ("timer_create");
```

失败时调用返回-1，timerid 未定义，并且设置 errno 为以下值之一：

EAGAIN

系统缺乏足够的资源完成请求。

EINVAL

clockid 指定的 POSIX 时钟非法。

ENOTSUP

clockid 指定的 POSIX 时钟合法，但是系统不支持使用该时钟作为定时器。POSIX 保证所有实现都支持 CLOCK_REALTIME 时钟作为定时器。其它时钟是否支持取决于具体的实现。

evp 参数（如果不为 NULL）定义定时器到期时发生的异步通知。头文件<signal.h>定义了此结构体。它的内容被认为是对程序员不透明的，但是它至少包含以下域：

```
#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;
};
```

POSIX 时钟定时器对内核怎样通知进程定时器到期有非常多的控制，允许进程指定内核发送哪个信号；甚至允许内核创建一个线程，在定时器到期时执行一个函数作为响应。进程通过 sigev_notify 指定定时器到期时的行为，它必须是以下三个值之一：

SIGEV_NONE

“null”通知，当定时器到期时，不发生任何事情。

SIGEV_SIGNAL

当定时器到期时，内核向进程发送 sigev_signo 指定的信号。在信号处理器中，si_value 被设置为 sigev_value。

SIGEV_THREAD

当定时器到期时，内核创建一个线程（当前进程内），并让线程执行 sigev_notify_function，传递 sigev_value 作为该函数的参数。当函数返回时线程终止。如果 sigev_notify_attributes 不为 NULL，则提供的 pthread_attr_t 结构体定义新创建的线程的行为。

如果 evp 为 NULL，正如我们前面的例子，定时器到期通知行为被设置成：sigev_notify 为 SIGEV_SIGNAL，sigev_signo 为 SIGALRM，并且 sigev_value 为定时器 ID。因此默认情况下，这个定时器的行为类似于 POSIX 时间间隔定时器。不过通过定制，时钟定时器能够做多得多的工作。

下面例子创建一个 CLOCK_REALTIME 时钟定时器。当定时器到期时，内核会发送 SIGUSR1 信号，并设置 si_value 为定时器 ID 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");
```

装备定时器

通过 timer_create()创建的定时器是未装备的。要关联定时器到期满时间，并开始定时，使用 timer_settime()：

```
#include <time.h>

int timer_settime (timer_t timerid,
                   int flags,
                   const struct itimerspec *value,
                   struct itimerspec *ovalue);
```

成功调用 timer_settime()装备 timerid 指定的定时器，期满时间为 value，它是一个 itimerspec 结构体：

```
struct itimerspec {
    struct timespec it_interval; /* next value */
    struct timespec it_value; /* current value */
};
```

和 setitimer()一样，it_value 指定当前定时器期满时间。当定时器到期时，it_value 将使用 it_interval 的值刷新。如果 it_interval 为 0，定时器就不是时间间隔的，一旦 it_value 到期就解除装备。

回忆之前我们学习过的 timespec 结构体，提供纳秒精度：

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

如果 flags 为 TIMER_ABSTIME，value 指定的时间就是绝对时间，而不是默认的相当时间。这个行为防止了获得当前时间、计算时间相对差异、想要的未来时间、以及装备定时器之间的竞争条件。详细信息请参考之前 “睡眠的高级方式”一节的讨论。

如果 ovalue 不为 NULL，则前一个定时器期满时间将被存储在所提供的 itimerspec 中。如果定时器之前被解除装备，结构体的成员都将被设为 0。

下面例子使用之前 timer_create()创建的 timer，创建一个每秒期满的定时器：

```
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");
```

获得定时器期满时间

通过 timer_gettime()，你不需要重新设置定时器就可以获得它的期满时间：

```
#include <time.h>

int timer_gettime (timer_t timerid,
                   struct itimerspec *value);
```

成功调用 timer_gettime()把 timerid 指定的定时器的期满时间存储在 value 指定的结构体中，并返回 0；失败时调用返回-1，并设置 errno 为以下值之一：

EFAULT
value 是无效指针。
EINVAL
timerid 是无效定时器。

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);
    }
```

获得定时器 overrun

POSIX 定义了一个接口获得指定的定时器发生的 overrun 时间：

```
#include <time.h>

int timer_getoverrun (timer_t timerid);
```

成功时 timer_getoverrun()返回定时器的 overrun，也就是定时器期满与通知进程之间的时间差。例如，在我们之前的例子中，当 1ms 的定时器运行 10ms 时，这个调用将返回 9。

如果 overrun 大于或等于 DELAYTIMER_MAX，调用返回 DELAYTIMER_MAX。

失败时函数返回-1，并设置 errno 为唯一的错误代码 EINVAL，表示 timerid 指定的定时器无效。

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);
```

删除定时器

删除定时器非常简单：

```
#include <time.h>

int timer_delete (timer_t timerid);
```

成功调用 timer_delete()销毁 timerid 关联的定时器，并返回 0；失败时调用返回-1，并设置 errno 为唯一的错误代码 EINVAL，表示 timerid 是无效的定时器。

附录 GCC 对 C 语言的扩展

GNU 编译器集(GCC)对 C 语言提供了许多扩展，其中一些已经证明对系统程序员特别有价值。我们在本附录中讨论的多数 C 语言扩展都提供一种方式，让程序员向编译器提供代码行为和使用的额外信息。然后编译器就可以利用这些信息产生更加优化的机器代码。其它一些扩展则填补 C 语言的缺陷，特别是在底层。

GCC 提供的几个扩展现在已经在最新的 C 标准 ISO C99 中存在。有一些扩展则类似于 C99 标准所定义，但是 ISO C99 对其它扩展的实现则完全不同。新代码应该使用 ISO C99 特性。我们不会在这里讨论这种扩展，只讨论 GCC 特有的扩展。

GNU C

GCC 支持的 C 通常称为 GNU C。在 1990 年代，GNU C 弥补了 C 语言的几个缺陷，提供类似复杂变量、0 长度数组、内联函数、和命名初始化块等特性。但是在经过大概十年之后，C 语言最终得到了进化，并且有了 ISO C99 标准，GNU C 扩展变得不那么重要。无论如何，GNU C 继续提供有用的特性，并且许多 Linux 程序员在他们的 C90 或 C99 代码中仍然使用 GNU C 扩展的子集（通常只是一两个扩展）。

GCC 特定代码的显著例子是 Linux 内核，它严格按照 GNU C 编写。不过最近 Intel 已经投入工程努力，允许 Intel C 编译器(ICC)理解内核使用的 GNU C 扩展。因此，许多扩展现在已经不那么 GCC 专有了。

内联函数

编译器复制内联函数的所有代码到函数被调用的地方，而不是把函数存储在外部并在调用时跳转到函数，它直接运行函数的内容。这个行为节省了函数调用的开销，并允许潜在的优化，因为编译器可以对调用方和函数一起进行优化。后者在函数的参数是常量时特别有效。但是很显然，把函数代码复制到每一个调用点对代码大小有不利的影响。因此，只有当函数很小而且简单时，或者不被许多地方调用时，才应该使用内联。

GCC 已经支持 inline 关键字很多年了，指示编译器内联指定的函数。C99 正式定义了这个关键字：

```
static inline int foo (void) { /* ... */ }
```

不过技术上讲，这个关键字仅仅是一个提示——建议编译器考虑内联指定的函数。GCC 进一步提供指示编译器总是内联指定函数的扩展：

```
static inline __attribute__ ((always_inline)) int foo (void) { /* ... */ }
```

内联函数最明显的候选人是预处理宏定义。GCC 中的内联函数和宏执行完全一样，并且提供额外的类型检查。例如，以下宏定义：

```
#define max(a,b) ({ a > b ? a : b; })
```

可以使用相应的内联函数替换：

```
static inline max (int a, int b)
{
```

```
    if (a > b)
        return a;
    return b;
}
```

程序员倾向于过度使用内联函数。函数调用开销在多数现代体系架构中非常非常小(特别是 x86)。只有对最值得内联的函数才应该考虑。

禁止内联

在最激进的优化模式下，GCC 自动选择适合的函数并执行内联。这通常是一个好主意，但是有时候程序员确定某个函数执行内联时将执行错误。一个可能的例子是当使用__builtin_return_address（后面讨论）时。使用 noinline 关键字禁止内联：

```
__attribute__ ((noinline)) int foo (void) { /* ... */ }
```

纯函数

“纯”函数是对参数和全局变量没有影响，并且返回值只反应函数参数或非 volatile 全局变量的函数。任何参数或全局变量的访问都必须是只读的。循环优化和子表达式消除可以应用到这种函数上。函数通过 pure 关键字标记为纯函数：

```
__attribute__ ((pure)) int foo (int val) { /* ... */ }
```

一个常见的例子是 strlen()。给定相同的输入，函数的返回值在多次调用不变，因此它可以从循环中提出来，并且只调用一次。例如，考虑以下代码：

```
/* character by character, print each letter in 'p' in uppercase */
for (i = 0; i < strlen (p); i++)
    printf ("%c", toupper (p[i]));
```

如果编译器不知道 strlen()是纯函数，它就只能在每次循环迭代时都调用函数！

聪明的程序员会编写如下代码，如果 strlen()标记为 pure，聪明的编译器也会产生类似如下代码：

```
size_t len;
len = strlen (p);
for (i = 0; i < len; i++)
    printf ("%c", toupper (p[i]));
```

顺带插一句，更加聪明的程序员（例如本书的读者）可能会这样写：

```
while (*p)
    printf ("%c", toupper (*p++));
```

纯函数返回 void 是非法的，实际上也没有意义，因为返回值是纯函数的唯一要点。

常函数

“常”函数是比纯函数更加严格的变种。这种函数不能访问全局变量，并且不能把指针作为参数。因此，常函数的返回值只反应按值传递进来的参数。在纯函数优化之上的额外优化可以应用到常函数中。数学函数例如 abs()，是常函数的一个例子（假设它不保存状态，或者使用其它优化技巧）。程序员通过 const 关键字标记常函数：

```
__attribute__ ((const)) int foo (int val) { /* ... */ }
```

和纯函数一样，常函数返回 void 也没有意义。

无返回的函数

如果一个函数不返回——可能它调用 exit()——程序员可以用 noreturn 关键字标记函数，告诉编译器这个事实：

```
__attribute__ ((noreturn)) void foo (int val) { /* ... */ }
```

编译器知道函数在任何情况下都不返回时，就可以执行额外的优化。这种函数返回除 void 之外的值是无意义的。

分配内存的函数

如果一个函数返回一个指针，指向之前不存在的内存（函数分配新内存并返回指针），程序员可以通过 malloc 关键字标记函数，编译器据此可以执行适当的优化：

```
__attribute__ ((malloc)) void * get_page (void)
{
    int page_size;

    page_size = getpagesize ( );
    if (page_size <= 0)
        return NULL;

    return malloc (page_size);
}
```

强制调用者检查返回值

warn_unused_result 属性指示编译器在函数返回值没有被存储，或者在条件语句中使用时，产生一条警告信息。它不是一个优化，而是一个编程援助：

```
__attribute__ ((warn_unused_result)) int foo (void) { /* ... */ }
```

这允许程序员在函数返回值特别重要时，确保所有调用都检查并处理了函数的返回值。

拥有非常重要的返回值，但却经常被忽略的函数(例如 read)，是这个属性的最佳候选者。这种类型的函数不能返回 void。

标记函数为 Deprecated

Deprecated 属性指示编译器在函数被调用时产生一条警告信息：

```
__attribute__ ((deprecated)) void foo (void) { /* ... */ }
```

这可以帮助程序员反对和废弃接口。

标记函数为已使用

有时候编译器可见的代码都没有调用某个特定的函数。使用 used 属性标记函数，告诉编译器程序已经使用了这个函数，不管表面上函数是否被引用：

```
static __attribute__ ((used)) void foo (void) { /* ... */ }
```

编译器由此可以输出结果汇编语言，并不显示未使用函数的警告。这个属性在 static 函数只被硬编码的汇编代码调用时非常有用。通常如果编译器不知道函数被调用时，它会产生一条警告信息，并可能移除这个函数。

标记函数或参数为未使用

unused 属性告诉编译器指定的函数或函数参数未使用，并指示它不要产生相应的警告：

```
int foo (long __attribute__ ((unused)) value) { /* ... */ }
```

当你希望捕获未使用函数参数，使用-W 或-Wunused 编译，但是你的某些函数又必须匹配预定义的函数签名时（在事件驱动 GUI 编程或信号处理器中很常见），这个属性很有用。

包装结构体

packed 属性告诉编译器某个类型或变量应该打包为使用可能的最小内存空间，潜在地忽视对齐需求。如果对 struct 或 union 指定，所有成员都将被打包。如果只对一个变量指定，则只有这个特定的变量被打包。

下面代码会打包结构体中的所有变量为最小空间：

```
struct __attribute__ ((packed)) foo { ... };
```

举个例子，一个结构体包含一个 char，紧跟着一个 int，通常会把 int 对齐于不紧接 chra 的内存地址，例如三个字节之后。编译器通过插入填充字节来对齐变量。一个 packed 结构体没有这种填充，可以消耗更少内存，但是未能满足架构的对齐需求。

增加变量的对齐

GCC 允许对变量进行 packed，也允许程序员指定变量的最小对齐。然后 GCC 最小会按这个值来对齐变量，而不是体系架构和 ABI 指定的对齐。例如，下面语句声明 beard_length 变量最小对齐于 32 字节（而不是 32 位机器上典型的 4 字节）：

```
int beard_length __attribute__ ((aligned (32))) = 0;
```

通常只有处理硬件对齐需求大于体系架构本身；或者你混合 C 和汇编代码，并希望使用需要特定对齐值的指令时，对类型强制对齐才是有用的。使用这个对齐功能的一个例子是存储经常使用的变量到处理器缓存，优化缓存行为。Linux 内核使用了这个技术。

除了指定最小对齐值，你可以请求 GCC 对齐指定的类型到还没有被任何类型使用的最大的[最小对齐值]。例如下面代码指示 GCC 对齐 parrot_height 到它已经使用的最大对齐值，它可能是 double 类型的对齐值：

```
short parrot_height __attribute__ ((aligned)) = 5;
```

这个决定通常涉及到空间/时间权衡：按这种方式对齐的变量消耗更多的空间，但是复制它们（以及其它复杂操作）可能会更快，因为编译器可以发起处理最大数量内存的机器指令。

体系架构或系统工具链的许多因素都可能对变量对齐强加最大的限制。例如在某些 Linux 体系架构中，链接器不能识别超过很小默认值的对齐值。在这种情况下，使用这个关键字提供的对齐值会被减小为最小允许的对齐值。例如，如果你请求对齐为 32，但系统链接器只支持对齐为 8 字节，变量最终会对齐于 8 字节边界。

放置全局变量到寄存器

GCC 允许程序员把全局变量放到指定的机器寄存器，这个变量在程序执行期间一直定居在寄存器中。GCC 把这种变量称为全局寄存器变量。

语法要求程序员指明具体的机器寄存器。下面例子使用 ebx：

```
register int *foo asm ("ebx");
```

程序员必须选择那些不是函数-clobbered 的变量：也就是选择的变量必须只对本地函数可用，在函数被调用时保存并恢复；并且不被体系架构或操作系统 ABI 指定为任何特殊用途。如果选择的寄存器不恰当，编译器会产生一个警告信息。如果寄存器是合适的（例子中的 ebx，在 x86 体系下就很好），编译器自身就会停止使用该寄存器。

如果变量被频繁使用，这种优化可以提供巨大的性能提高。一个很好的例子是虚拟机。把保存虚拟堆栈帧的指针存放到寄存器中，可以获得大量好处。反过来，如果体系架构本身就缺乏寄存器（例如 x86 架构），这个优化就没有太大意义。

全局寄存器变量不能在信号处理器中使用，也不能被多个线程执行。它们同时还不能有初始值，因为执行文件没有办法给寄存器提供默认值。全局寄存器变量声明必须在所有函数定义之前。

分支注解

GCC 允许程序员注解一个表达式的预期值——例如，告诉编译器一个条件语句可能是 true 还是 false。然后 GCC 就可以执行块重排以及其它优化，来提高条件分支的性能。

GCC 分支注解的语法丑陋得可怕。为了使分支注解看上去更加简单，我们使用预处理宏：

```
#define likely(x) __builtin_expect (!!(x), 1)
#define unlikely(x) __builtin_expect (!!(x), 0)
```

程序员可以通过 likely()或 unlikely()标记一个表达式可能或不太可能为 true。

下面例子标记一个分支不太可能为 true（也就是很可能为 false）：

```
int ret;
ret = close (fd);
if (unlikely (ret))
    perror ("close");
```

反过来，下面例子标记分支可能为 true：

```
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");
```

和内联函数一样，程序员倾向于过度使用分支注解。一旦你开始注解表达式，你可能就会标记所有表达式。但是请小心，只有你预先知道表达式在几乎所有情况下(99%)是 true 或 false 时，才应该标记该表达式。很少发生的错误是 unlikely()很好的候选者。但是请记住，一个错误的预测比没有预测更糟糕。

获得表达式的类型

GCC 提供 typeof()关键字来获得指定表达式的类型。从语义上讲，typeof()的操作和 sizeof()类似。例如下面表达式返回 x 指向的类型：

```
typeof(*x)
```

我们可以使用这个类型来定义数组：

```
typeof (*x) y[42];
```

typeof()的一个流行用法是编写“安全”的宏，它可以操作任何算术值，并且对参数只计算一次：

```
#define max(a,b) ({ \
    typeof (a) _a = (a); \
    typeof (b) _b = (b); \
```

```
        _a > _b ? _a : _b; \
    })
```

获得类型的对齐

GCC 提供关键字__alignof__来获得指定对象的对齐。这个值是体系架构和 ABI 相关的。如果当前体系架构没有对齐要求，__alignof__返回 ABI 推荐的对齐值。其它情况下，关键字返回最小必需对齐值。

语法和 sizeof()相同：

```
__alignof__(int)
```

根据不同的体系架构，它可能返回 4，32 位整数通常对齐于 4 字节边界。

这个关键字也可以用于左值。在这种情况下，返回的对齐是指定类型的最小对齐，而不是指定左值的实际对齐。如果最小对齐已经通过 aligned 属性改变，这个改变也会反应在__alignof__中。

例如，考虑下面结构体：

```
struct ship {
    int year_built;
    char canons;
    int mast_height;
};
```

以及下面代码片断：

```
struct ship my_ship;
printf ("%d\n", __alignof__(my_ship.canons));
```

__alignof__将返回 1，尽管结构体填充可能导致 canons 消耗四个字节。

结构体成员偏移量

GCC 提供一个内建关键字__builtin_offsetof 获得结构体中某个成员的偏移量。头文件<stddef.h>中定义的 offsetof()宏，是 ISO C 标准的一部分。多数类似的定义都很恐怖，引入可恶的指针运算以及不适当的代码。GCC 扩展更加简单，可能还会更快一些：

```
#define offsetof(type, member) __builtin_offsetof (type, member)
```

调用返回 type 类型中 member 的偏移量，也就是从结构体开始到该成员的字节数。例如，考虑下面结构体：

```
struct rowboat {
    char *boat_name;
    unsigned int nr_oars;
    short length;
};
```

实际的偏移量依赖于变量的大小，以及体系架构的对齐需求和填充行为，但是在 32 位机器上，我们可以预料对结构体 orwboat 和 boat_name，nr_oars，以及 length 调用 offsetof()

将分别返回 0，4，和 8。

在 Linux 系统中，offsetof()宏应该已经使用 GCC 关键字来定义，因此不需要重新定义。

获得函数返回地址

GCC 提供一个关键字获得当前函数的返回地址，或者当前函数的调用者的返回地址：

```
void * __builtin_return_address (unsigned int level)
```

level 参数指定需要返回的调用链中的函数地址。值为 0 请求当前函数的返回地址；值为 1 请求返回当前函数的调用者的返回地址；值为 2 请求返回调用者的调用者的返回地址，依此类推。

如果当前函数是内联函数，则返回调用函数的地址。如果这不可接受，可以使用 noinline 关键字来强制编译器不内联函数。

__builtin_return_address 关键字有几个用途。其一是调试和报告，另外一个是展开调用链，实现自省(introspection)，崩溃 dump 工具，调试器等等。

注意某些体系架构只能返回调用函数的地址。在这些架构下，非 0 参数可能导致随机返回值。因此任何非 0 的参数都是不可移植的，只应该在调试用途中使用。

case 范围

GCC 允许 case 语句标签指定一个范围的值。通用的语法如下：

```
case low ... high:
```

例如：

```
switch (val) {
case 1 ... 10:
    /* ... */
    break;
case 11 ... 20:
    /* ... */
    break;
default:
    /* ... */
}
```

这个功能对 ASCII 范围也很有用：

```
case 'A' ... 'Z':
```

注意在…之前和之后必须有一个空格，否则编译器就糊涂了，特别是整数范围时。总是像下面这样：

```
case 4 ... 8:
```

而不是这样：

```
case 4...8:
```

void 指针和函数指针运算

在 GCC 中，对 void 指针和函数指针进行加减操作是允许的。通常 ISO C 不允许对这种指针进行运算，因为“void”的大小没有概念，并且依赖于指针实际指向的对象。为了方便这种运算，GCC 把指针引用的对象当作一个字节。因此下面代码片断增加一字节：

```
a++;        /* a is a void pointer */
```

GCC 的-Wpointer-arith 选项在程序使用了这些扩展时会产生一条警告信息。

更加可移植和更加优美

坦白地说，__attribute__语法并不优美。本章我们讨论的某些扩展本质上需要预处理宏来使得它们的使用更加可移植，但是打扮整齐对所有扩展的使用都有帮助。

只需要一点点预处理魔法，这就可以实现。此外，相同的动作还使 GCC 扩展变得可移植，通过在非 GCC 编译器中移除它们。

把下面代码片断放到头文件中，并在你的源代码中包含这个头文件：

```
#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
```

例如下面代码使用我们的定义标记纯函数：

```
    __pure int foo (void) { /* ... */
```

如果使用的是 GCC，这个函数将使用 pure 属性标记；如果没有使用 GCC，预处理宏将替换__pure 标记为空。注意你可以把多个属性放到一个定义中，因此在一个定义中使用多个属性是没有问题的。

更简单，更优美，而且更加可移植！

Bibliography

This bibliography presents recommended reading related to system programming, broken down into four subcategories. None of these works are required reading. Instead, they represent my take on the top books on the given subject matter. If youfind yourself pining for more information on the topics discussed here, these are myfavorites.

Some of these books address material with which this book assumes the reader isalready conversant, such as the C programming language. Other texts included makegreat supplements to this book, such as the works covering *gdb*, Subversion (*svn)*, oroperating system design. Still others handle topics that are beyond the scope of thisbook, such as multithreading of sockets. Whatever the case, I recommend them all. Of course, these lists are certainly not exhaustive—feel free to explore otherresources.

Books on the C Programming Language

These books document the C programming language, the *lingua franca* of system programming. If you do not code C as well as you speak your native tongue, one or more of the following works (coupled with a lot of practice!) ought to help you in that direction. If nothing else, the first title—universally known as *K&R*—is a treat to read. Its brevity reveals the simplicity of C.

- *The C Programming Language*, 2nd ed. Brian W. Kernighan and Dennis M. Ritchie. Prentice Hall, 1988.

  This book, written by the author of the C programming language and his thencoworker, is the bible of C programming.
- *C in a Nutshell*. Peter Prinz and Tony Crawford. O’Reilly Media, 2005.

  A great book covering both the C language and the standard C library.
- *C Pocket Reference*. Peter Prinz and Ulla Kirch-Prinz. Translated by Tony Crawford. O’Reilly Media, 2002.

  A concise reference to the C language, handily updated for ANSI C99.
- *Expert C Programming*. Peter van der Linden. Prentice Hall, 1994.

  A wonderful discussion of lesser-known aspects of the C programming language, elucidated with an amazing wit and sense of humor. This book is rife with nonsequitur jokes, and I love it.
- *C Programming FAQs: Frequently Asked Questions*, 2nd ed. Steve Summit. Addison-Wesley, 1995.

  This beast of a book contains more than 400 frequently asked questions (with answers) on the C programming language. Many of the FAQs beg obvious answers in the eyes C masters, but some of the weightier questions and answers should impress even the most erudite of C programmers. You are a true C ninja if you can answer all of these bad boys! The only downside is that the book has not been updated for ANSI C99, and there have definitely been some changes (I handmade the corrections in my copy). Note there is an online version that has likely been more recently updated.

Books on Linux Programming

The following texts cover Linux programming, including discussions of topics not covered in this book (sockets, IPC, and *pthreads*), and Linux programming tools (CVS, GNU Make, and Subversion).

- *Unix Network Programming, Volume 1: The Sockets Networking API*, 3rd ed. W. Richard Stevens et al. Addison-Wesley, 2003.

  The definitive tome on the socket API; unfortunately not specific to Linux, but fortunately recently updated for IPv6.
- *UNIX Network Programming, Volume 2: Interprocess Communications*, 2nd ed. W. Richard Stevens. Prentice Hall, 1998.

  An excellent discussion of interprocess communication (IPC).
- *PThreads Programming: A POSIX Standard for Better Multiprocessing*. Bradford Nichols et al. O’Reilly Media, 1996.

  A review of the POSIX threading API, pthreads.
- *Managing Projects with GNU Make*, 3rd ed. Robert Mecklenburg. O’Reilly Media, 2004.

  An excellent treatment on GNUMake, the classic tool for building software projects on Linux.
- *Essential CVS*, 2nd ed. Jennifer Versperman. O’Reilly Media, 2006.

  An excellent treatment on CVS, the classic tool for revision control and source code management on Unix systems.
- *Version Control with Subversion*. Ben Collins-Sussman et al. O’Reilly Media, 2004.

  A phenomenal take on Subversion, the proper tool for revision control and source code management on Unix systems, by three of Subversion's own authors.
- *GDB Pocket Reference*. Arnold Robbins. O’Reilly Media, 2005.

  A handy pocket guide to *gdb*, Linux’s debugger.
- *Linux in a Nutshell*, 5th ed. Ellen Siever et al. O’Reilly Media, 2005.

  A whirlwind reference to all things Linux, including many of the tools comprising Linux’s development environment.

Books on the Linux Kernel

The two titles listed here cover the Linux kernel. Reasons for investigating this topic are threefold. First, the kernel provides the system call interface to user space, and is thus the core of system programming. Second, the behaviors and idiosyncrasies of a kernel shed light on its interactions with the applications it runs. Finally, the Linux kernel is a wonderful chunk of code, and these books are fun.

- *Linux Kernel Development*, 2nd ed. Robert Love. Novell Press, 2005.

  This work is ideally suited to system programmers who want to know about the design and implementation of the Linux kernel (and naturally, I would be remiss not to mention my own treatise on the subject!). Not an API reference, this book offers a great discussion of the algorithms used and decisions made by the Linux kernel.
- *Linux Device Drivers*, 3rd ed. Jonathan Corbet et al. O’Reilly Media, 2005.

This is a great guide to writing device drivers for the Linux kernel, with excellent API references. Although aimed at device drivers, the discussions will benefit programmers of any persuasion, including system programmers merely seeking more insight into the machinations of the Linux kernel. A great complement to my own Linux kernel book.

Books on Operating System Design

These two works, not specific to Linux, address operating system design in the abstract. As I've stressed in this book, a strong understanding of the system you code on only improves your output.

- *Operating Systems*, 3rd ed. Harvey Deitel et al. Prentice Hall, 2003.

  A *tour de force* on the theory of operating system design coupled with top-notch case studies putting that theory to practice. Of all the textbooks on operating system design, this is my favorite: it's modern, readable, and complete.
- *UNIX Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programming*. Curt Schimmel. Addison-Wesley, 1994.

  Despite being only modestly related to system programming, this book offers such an excellent approach to the perils of concurrency and modern caching that I recommend it even to dentists.