

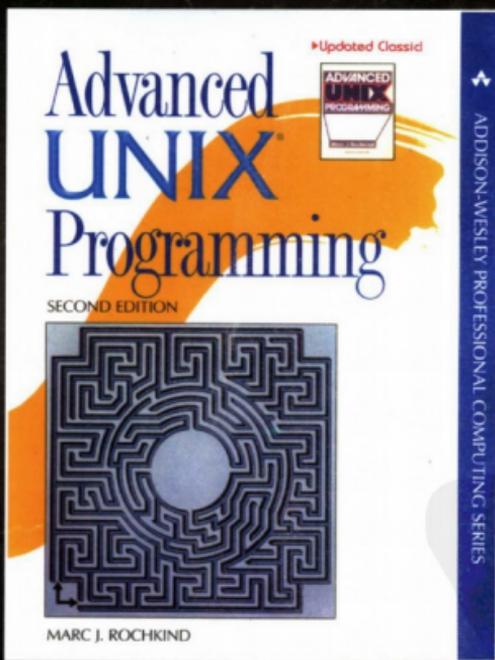


计 算 机 科 学 丛 书

原书第2版

高级UNIX编程

(美) Marc J. Rochkind 著 王嘉祯 杨素敏 张斌 等译



Advanced UNIX Programming
Second Edition



机械工业出版社
China Machine Press

如果你对UNIX系统程序设计感兴趣，那一定要读这本书。

——Ed Schaefer (Unix Review)

对UNIX程序设计新手和那些有经验的C程序员来说，本书是一个极好的起点。

——TechBookReport

本书第1版自1985年出版以来，历经20年畅销不衰，许多有经验的程序员都求助于它，作者Marc J. Rochkind被誉为UNIX先行者。当前，UNIX规范中有1100多个函数，要掌握这些函数确实是一件让人望而却步的事。第2版几乎完全重写，当中建议了如何可靠地使用关键函数，深入讲解了UNIX操作系统家族最新的、必用的系统调用函数（多达307个），涉及POSIX、FreeBSD、Solaris、Linux等几大主流系统实现。

全书包括：

- 基本概念，进程通信，网络（套接字），伪终端，I/O流，高级信号，实时处理和线程。
- 数千行示例代码，包括一个Web浏览程序，一个击键记录程序/播放器，用管道、重定向写的shell程序，以及相关的后台进程程序。
- 每章末的练习。一些是简单的程序设计问题，还有一些则可以作为每学期的UNIX程序设计项目。

配套网站www.basepath.com/aup提供了丰富的学习资源，包括：书中所有示例的源代码、作者的博客、书评、相关文献和图书、标准UNIX函数概要，等等。

作者简介

Marc J. Rochkind

UNIX程序设计先行者，20世纪70年代任职于Bell实验室，那时UNIX正处于起步阶段。他对UNIX的首要贡献便是开发了源代码控制系统。这次修订主要来自他多年在Bell实验室应用系统开发积累的经验。



www.PearsonEd.com



ISBN 7-111-18521-8



9 787111 185215



华章图书

上架指导：计算机科学/程序设计

华章网站 <http://www.hzbook.com>

网上购书：www.china-pub.com

投稿热线：(010) 88379604

购书热线：(010) 68995259, 68995264

读者信箱：hzjsj@hzbook.com

ISBN 7-111-18521-8/TP · 4673

定价：59.00 元

计

算

机



工院 106202969270

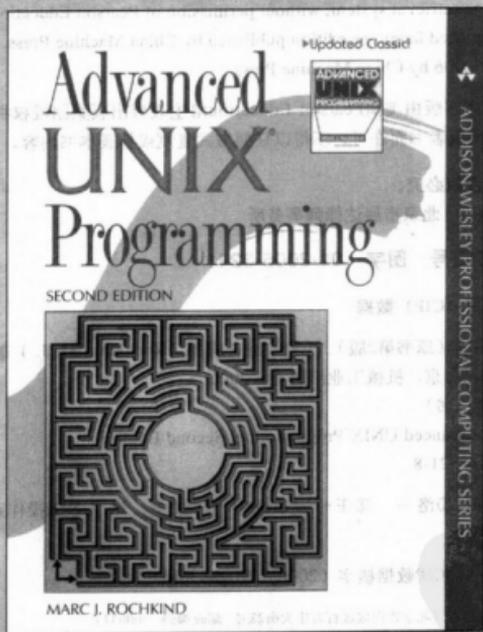
丛

书

原书第2版

高级UNIX编程

(美) Marc J. Rochkind 著 王嘉祯 杨素敏 张斌 等译



Advanced UNIX Programming
Second Edition



机械工业出版社
China Machine Press

本书以当前UNIX规范为基础,详细介绍了UNIX系统函数的用法,并用大量的代码和示例程序进行演示,对实际编程具有指导意义。全书共9章,内容包括:基本概念、基本文件I/O、高级文件I/O、终端I/O、进程与线程、基本进程间通信、高级进程间通信、网络技术与套接字,以及信号与定时器等。涉及POSIX、FreeBSD、Solaris、Linux等几大主流系统实现。每章末都给出了一些练习,一些是简单的程序设计问题,还有一些可以作为学期的UNIX程序设计项目。

本书适合广大UNIX和C程序员、研究人员、高校相关专业师生学习和参考。

Authorized translation from the English language edition entitled *Advanced UNIX Programming*, Second Edition by Marc J. Rochkind, published by Pearson Education, Inc., publishing as Addison-Wesley (ISBN0-13-141154-3), Copyright © 2004 Pearson Education, Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanic, including photocopying, recording, or by any information storage retrieval system, without permission of Pearson Education, Inc.

Chinese simplified language edition published by China Machine Press.

Copyright © 2006 by China Machine Press.

本书中文简体字版由美国Pearson Education培生教育出版集团授权机械工业出版社独家出版。未经出版者书面许可,不得以任何方式复制或抄袭本书内容。

版权所有,侵权必究。

本书法律顾问 北京市展达律师事务所

本书版权登记号: 图字: 01-2005-3241

图书在版编目(CIP)数据

高级UNIX编程(原书第2版)/(美)洛奇肯德(Rochkind, M. J.)著;王嘉楨,杨素敏,张斌等译.-北京:机械工业出版社,2006.5

(计算机科学丛书)

书名原文:Advanced UNIX Programming, Second Edition

ISBN 7-111-18521-8

I. 高… II. ①洛… ②王… ③杨… ④张… III. UNIX操作系统-程序设计 IV. TP316.81

中国版本图书馆CIP数据核字(2006)第028636号

机械工业出版社(北京市西城区百万庄大街22号 邮政编码 100037)

责任编辑:朱起飞 冯春丽

北京牛山世兴印刷厂印刷·新华书店北京发行所发行

2006年5月第1版第1次印刷

787mm × 1092mm 1/16 · 31.25印张

定价:59.00元

凡购本书,如有倒页、脱页、缺页,由本社发行部调换
本社购书热线:(010) 68326294



出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅肇划了研究的范畴，还揭开了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短、从业人员较少的现状下，美国等发达国家在其计算机科学发展的几十年间积淀的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章图文信息有限公司较早意识到“出版要为教育服务”。自1998年开始，华章公司就将工作重点放在了遴选、移译国外优秀教材上。经过几年的不懈努力，我们与Prentice Hall, Addison-Wesley, McGraw-Hill, Morgan Kaufmann等世界著名出版公司建立了良好的合作关系，从它们现有的数百种教材中甄选出Tanenbaum, Stroustrup, Kernighan, Jim Gray等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及度藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专程为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍，为进一步推广与发展打下了坚实的基础。

随着学科建设的初步完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都步入一个新的阶段。为此，华章公司将加大引进教材的力度，在“华章教育”的总规划之下出版三个系列的计算机教材：除“计算机科学丛书”之外，对影印版的教材，则单独开辟出“经典原版书库”；同时，引进全美通行的教学辅导书“Schaum's Outlines”系列组成“全美经典学习指导系列”。为了保证这三套丛书的权威性，同时也为了更好地为学校和老师服务，华章公司聘请了中国科学院、北京大学、清华大学、国防科技大学、复旦大学、上海交通大学、南京大学、浙江大学、中国科技大学、哈尔滨工业大学、西安交通大学、中国人民大学、北京航空航天大学、北京邮电大学、中山大学、解放军理工大学、郑州大学、湖北工学院、中国国家信息安全测评认证中心等国内重点大学和科研机构在计算机的各个领域的著名学者组成“专家指导委员会”，为我们提供选题意见和出版监督。

这三套丛书是响应教育部提出的使用外版教材的号召，为国内高校的计算机及相关专业的教学度身订造的。其中许多教材均已为M. I. T., Stanford, U.C. Berkeley, C. M. U. 等世界名牌大学所采用。不仅涵盖了程序设计、数据结构、操作系统、计算机体系结构、数据库、编译原理、软件工程、图形学、通信与网络、离散数学等国内大学计算机专业普遍开设的核心课程，而且各具特色——有的出自语言设计者之手、有的历经三十年而不衰、有的已被全世界的几百所高校采用。在这些圆熟通博的名师大作的指引之下，读者必将在计算机科学的宫殿中由登堂而入室。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证，但我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。教材的出版只是我们的后续服务的起点。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

电子邮件：hzsj@hzbook.com

联系电话：(010) 68995264

联系地址：北京市西城区百万庄南街1号

邮政编码：100037



译者序

UNIX的系统调用是UNIX内核与用户程序之间的接口，是UNIX程序员编写程序时必须掌握的重要内容。

但是，UNIX规范中大约有1108个系统调用，要想通过规范来学习这些系统调用，不但枯燥无味，而且也是不可能的。本书向读者详细地介绍主要的系统调用，并通过示例代码说明它们的使用方法，再进一步通过练习来实践这些内容。本书是深入学习UNIX、编写应用程序不可多得的优秀教材。

早在1985年，作者就出版了《高级UNIX编程》的第1版，本书是第2版，两版间隔近20年的时间。这期间，UNIX环境发生了很大变化，出现了POSIX、Solaris、Linux、FreeBSD以及Darwin (Mac OS X) 等类UNIX系统。第2版中包括了对以上内容的讨论；除了包含第1版大约70个系统调用之外，又增加了200多个，共有300多个系统调用。这300多个系统调用包括：进程间通信、网络（套接字）、伪终端、高级信号量、实时处理和线程等内容；同第1版一样，本书的第2版还包括了几千行示例代码，其中大多数来源于实际程序（比如shell、全屏菜单系统、Web服务器和实时输出记录器），只是在实际程序的基础上进行了条件简化。这些例子都是用C语言编写的，以帮助读者更好地理解系统调用的含义和具体的应用；此外，在本书的每一章后面都有精心设计的习题，用于巩固所学的内容。这些习题的难度不同，有的习题比较简单，而有的习题则是让读者运用所学的知识进行综合性的练习，有一定难度。

本书的作者Marc J. Rochkind自20世纪70年代开始就在Bell实验室工作，长期致力于UNIX系统的开发研究，积累了许多开发应用系统的经验，本书的第1版出版后就一直为应用程序的编程人员所青睐，现在重新修订的第2版又对UNIX操作系统家族最新的、必需的系统调用进行了系统深入的讲解和示范。

参加本书翻译工作的有：王嘉祯、杨素敏、张斌、冯兵、党辰、彭德云、徐波和胡建理等；王嘉祯、杨素敏等对全书进行了校阅、统稿；中国人民解放军军械工程学院米东教授等为本书的翻译工作提出了许多宝贵的意见和建议，对他付出的辛勤劳动表示深切的谢意。由于书中涉及的知识面比较广泛，再加上译者的水平有限，书中错误和不妥之处在所难免，恳请广大读者批评指正。

王嘉祯
2006年2月



前 言

本书是对1985年版《Advanced UNIX Programming》的更新，覆盖了过去18年来UNIX所发生的一些变化。也许“一些”并不确切！“更新”同样不太确切。的确，除了散落各处的个别句子，本书基本是全新的。第1版大约包含了70个系统调用；本版大约包含了300个。本版所讨论的UNIX标准和实现 [POSIX、Solaris、Linux、FreeBSD以及Darwin (Mac OS X)] 在1985年左右还未出现。但是我仍然可以在1985年版的前言中找到一些无需修改便可用在这里的句子：

本书的主题是UNIX系统调用——即UNIX内核与在其上层运行的用户程序之间的接口。对于那些仅使用命令与系统交互（比如shell、文本编辑器以及其他应用程序）的人来讲，或许不必对系统调用有太多了解，但对于UNIX程序员来说，对系统调用的彻底了解是至关重要的。系统调用是访问诸如文件系统、多任务机制以及进程间通信原语等内核功能的唯一途径。

系统调用定义了UNIX系统到底是什么。所有的一切（除了子程序和命令行）都是建立在这个基础之上的。尽管这些高层程序的许多新颖之处为UNIX赢得了不少名声，但它们也同样可以在任何现代操作系统上实现。当人们说UNIX系统是个精致的、简单的、高效的、可靠的和可移植的操作系统时，指的不是其命令（其中一些并不怎么样），而是其内核。

以上所说仍然正确，只是有一点令人遗憾：现在内核的编程接口不再那么雅致、简单了。事实上，由于在过去几十年中UNIX的发展分裂为几个分支，同时因为最初的标准组织（The Open Group）将几乎所有已有的函数都集合了起来（一共1108个函数），所以导致接口变得笨拙、矛盾、冗余，容易出错和混淆。但它依然高效、可靠并可移植，这就是为什么UNIX和类UNIX系统如此成功的原因。的确，UNIX系统调用接口是迄今我们所拥有的唯一——一个具有广泛可移植性的接口，而且这种状况可能在我们有生之年不会改变。

为了理清问题，拥有全部的文档是不够的，就像仅仅拥有黄页并不能找到好的饭店或宾馆一样。我们需要一位向导，能够告诉我们什么是好的、什么是坏的，而不仅仅是告诉我们有哪些东西。这就是本书的目的所在，也是本书与其他UNIX编程书的不同之处。本书不仅要指导读者如何使用系统调用，还要告诉他们不要使用哪些系统调用，因为那些系统调用都不是必需的，它们有的过时了，有的未被正确地实现，有的设计得很糟糕。

下面简单介绍一下本书的大致内容：开篇将介绍单一UNIX规范第3版中定义的1108个函数，但其中去掉了大约590个标准C函数和其他不属于内核接口层的库函数、大约90个POSIX线程函数（保留了其中十多个最为重要的）、大约25个审计登录函数、大约50个跟踪函数、大约15个晦涩废旧的函数以及大约40个用于调度和其他不大有用的函数。本书真正要介绍的只有307个。（见附录D的列表。）不是说这307个全是好的函数——有的也没什么用处，有的甚至还是危险的。但这307个函数都是读者需要了解的。

本书没有包括以下内容：内核实现（除了一些基本的）、设备驱动程序、C程序设计（有些间接的除外）、UNIX命令（shell、vi、emacs等）和系统管理。

全书共有9章：基本概念、基本文件I/O、高级文件I/O、终端I/O、进程和线程、基本的进程间通信、高级进程间通信、网络和套接字以及信号和定时器。先通读第1章，而后就可以自由跳跃浏览了。其中有许多交叉参考，能避免在阅读中迷失。

同第1版一样，这本新书包括了几千行示例代码，其中大多数来源于实际程序（比如shell、全屏菜单系统、Web服务器和实时输出记录器），并进行了简化。这些例子都是用C语言编写的，但在本书的附录B和附录C中给出了其他语言的接口，所以如果你喜欢，就可以采用C++、Java或Jython（Python的变体）来编程。

文字和示例代码仅仅是种资源；实际上还要通过练习来学习UNIX编程。为了提供练习，在每章的末尾都有练习题。这些练习难度不一，有的只需要简单地编写几行代码，有的则是一学期的课程设计。

我选了4种UNIX系统作为详细研究之用，并用来测试例子：Solaris 8、SuSE Linux 8（2.4内核）、FreeBSD 4.6和Darwin（Mac OS X内核）6.8。我将源码保存在FreeBSD系统上，然后用NFS或Samba把代码安装到其他系统上。[⊖]

我在Windows系统上用TextPad编辑代码，使用Telnet、SSH（PuTTY）或者X Window系统（XFree86和Cygwin）访问4个测试系统。在同一显示屏上打开文本编辑器和4个Telnet/SSH/Xterm窗口十分方便，因为从写代码到在4个系统上测试只需要几分钟时间。另外，我常常使用一个浏览器窗口打开单一UNIX规范，一个浏览器窗口打开Google，一个浏览器窗口运行Microsoft Word写书。除了Word对于像书之类的大型文档（破折号、混合样式，弱交叉引用，古怪的文档组合）有些糟糕之外，所有的工具都很好用。[⊗]我使用Perl和Python做了不同的事情，比如抽取代码样本和维护系统调用的数据库。

所有的示例代码（免费公开代码）、勘误表和更多的内容都在本书的Web站点 www.basepath.com/aup 上。

我要感谢那些审阅了草稿或者以其他方式提供了技术支持的人：Tom Cargill、Geoff Clare、Andrew Gierth、Andrew Josey、Brian Kernighan、Barry Margolin、Craig Patridge和David Schwartz。另外还要特别感谢那些专心细致地审阅了草稿但要求匿名的人。当然，这些人不需要为我在本书中找到的错误受到谴责——我信任他们。

我还要感谢我的编辑——Mary Franz，是她在一年前提议编写该书的。幸运的是，她正好是在我深入浏览了Linux并再一次为UNIX而兴高采烈的时候找到了我。这使我回想起1972年的时候……

我真心希望您能从本书中得到快乐！如果您发现了错误，或者您将代码移植到了新的系统中，或者您只是想分享您的想法，那么请给我发邮件：aup@basepath.com。

Marc J. Rochkind

Boulder, Colorado

2004年4月

⊖ 这4个系统运行在我多年收集的不同的旧PC和一台我花了200美元在eBay上购买的Mac上。使用SuSE并没有特别的原因，我已经在那台机器上安装了RedHat 9。

⊗ 我可以采用任何一个系统作为基础系统。Windows用起来很方便，因为我的宽LCD显示器连接在这个系统上，而且我喜欢用TextPad（www.textpad.com）。有关PuTTY的信息请参见站点 www.chiark.greenend.org.uk/~sgtatham/putty/。（如果这个链接不管用的话，Google一下“PuTTY”。）

专家指导委员会

(按姓氏笔画顺序)

尤晋元
石教英
张立昂
邵维忠
周克定
郑国梁
高传善
裘宗燕

王 珊
吕 建
李伟琴
陆丽娜
周傲英
施伯乐
梅 宏
戴 葵

冯博琴
孙玉芳
李师贤
陆鑫达
孟小峰
钟玉琢
程 旭

史忠植
吴世忠
李建中
陈向群
岳丽华
唐世渭
程时端

史美林
吴时霖
杨冬青
周伯生
范 明
袁崇义
谢希仁



目 录

出版者的话	
专家指导委员会	
译者序	
前言	
第1章 基本概念	1
1.1 UNIX和Linux一览	1
1.2 UNIX的版本	9
1.3 使用系统调用	12
1.4 错误处理	14
1.5 UNIX标准	24
1.6 共享头文件	35
1.7 日期和时间	36
1.8 关于示例代码	44
1.9 必要的资源	45
练习	45
第2章 基本文件I/O系统调用	47
2.1 概述	47
2.2 文件描述符及打开文件描述	48
2.3 文件权限位符号	49
2.4 open和creat系统调用	50
2.5 umask系统调用	56
2.6 unlink系统调用	57
2.7 创建临时文件	57
2.8 文件偏移量和O_APPEND	59
2.9 write系统调用	60
2.10 read系统调用	63
2.11 close系统调用	64
2.12 用户缓冲I/O	64
2.13 lseek系统调用	69
2.14 pread和pwrite系统调用	71
2.15 readv和writev系统调用	72
2.16 同步I/O	75
2.17 truncate和ftruncate系统调用	79
练习	80
第3章 高级文件I/O	81
3.1 概述	81
3.2 磁盘特殊文件和文件系统	81
3.3 硬链接和符号链接	90
3.4 路径名	94
3.5 访问和显示文件元数据	96
3.6 目录	105
3.7 改变信息节点	121
3.8 其他的文件处理调用	124
3.9 异步I/O	127
练习	134
第4章 终端I/O	135
4.1 概述	135
4.2 从终端读取数据	135
4.3 会话和进程组(作业)	149
4.4 ioctl系统调用	155
4.5 设置终端属性	155
4.6 其他终端控制系统调用	164
4.7 终端识别系统调用	165
4.8 全屏应用程序	167
4.9 流I/O	171
4.10 伪终端	171
练习	186
第5章 进程和线程	187
5.1 概述	187
5.2 环境	187
5.3 exec系统调用	191
5.4 实现shell(版本1)	197
5.5 fork系统调用	200
5.6 实现shell(版本2)	203
5.7 exit系统调用和进程终止	203
5.8 wait、waitpid和waitid系统调用	205
5.9 信号、终止和等待	211
5.10 实现shell(版本3)	212
5.11 获得用户ID和组ID	213
5.12 设置用户ID和组ID	214

5.13 获得进程ID	215	7.15 性能比较	350
5.14 chroot系统调用	216	练习	351
5.15 获得并设置优先级	216	第8章 网络和套接字	352
5.16 进程限制	218	8.1 套接字基础	352
5.17 线程介绍	222	8.2 套接字地址	362
5.18 阻塞问题	237	8.3 套接字选项	369
练习	242	8.4 简单套接字接口	373
第6章 基本的进程间通信	244	8.5 SMI套接字实现	384
6.1 概述	244	8.6 无连接套接字	386
6.2 管道	244	8.7 带外数据	393
6.3 dup和dup2系统调用	250	8.8 网络数据库函数	394
6.4 一个真正的shell	254	8.9 其他系统调用	406
6.5 非重定向管道的双向通信	264	8.10 高性能方面的考虑	409
6.6 用双向管道进行双向通信	270	练习	409
练习	272	第9章 信号和定时器	411
第7章 高级进程间通信	274	9.1 信号的基本概念	411
7.1 概述	274	9.2 等待信号	426
7.2 FIFO或命名管道	274	9.3 其他信号系统调用	434
7.3 抽象的简单消息接口 (SMI)	280	9.4 不赞成使用的信号系统调用	434
7.4 System V IPC	290	9.5 实时信号扩展	436
7.5 System V 消息队列	293	9.6 全局跳转	443
7.6 POSIX IPC	299	9.7 时钟和定时器	446
7.7 POSIX消息队列	301	练习	456
7.8 关于信号量	309	附录A 进程属性	458
7.9 System V信号量	311	附录B Ux: 一个对标准UNIX函数进行 包装的程序	461
7.10 POSIX信号量	318	附录C Jtux: 标准UNIX函数的Java/ Jython接口	463
7.11 文件锁	323	附录D 函数字母速查表及其分类表	469
7.12 关于共享内存	330	参考文献	487
7.13 System V共享内存	330		
7.14 POSIX 共享内存	342		

第1章 基本概念

1.1 UNIX和Linux—览

本节将快速浏览UNIX和Linux内核提供的各种功能。这里不会涉及那些通常和UNIX一起提供的用户程序（命令），如ls、vi和grep。对它们的讨论已经超出了本书范围。同时，本书也不会过多涉及有关系统内核的问题（如文件系统是如何实现的）。（从现在起，在书中只要提及UNIX一词，其概念均包含Linux，除非另有说明。）

本节的目的在于进行一次复习。因为这里假设读者已对某些概念（如进程等概念）有了粗略了解，所以在使用这些概念时不会先对其进行定义。如果读者对所提及的概念有些生疏，那就需要在阅读本书前先了解一下UNIX系统（如果还不知道“进程”是什么，那就必须先热热身！）。目前，有许多UNIX的入门书籍可帮助读者启蒙，其中有两本不错的书：《UNIX开发环境》（The UNIX Programming Environment）[Ker 1984]和《UNIX速成》（UNIX for the Impatient）[Abr 1996]（第2章是一个非常好的入门介绍）。^①

1.1.1 文件

UNIX文件包括以下几种：常规文件、目录、符号链接、特殊文件、命名管道（FIFO）和套接字（socket）文件。这里先介绍前4种文件，后两种将在1.1.7节中介绍。

1.1.1.1 常规文件

常规文件（Regular file）包含以线性数组组织的数据字节。任何字节或字节序列都可被读或写。读或写时的开始字节位置由文件偏移量（file offset）确定，后者可设定为任意数值（甚至可以超出文件末尾）。另外，常规文件存储在磁盘上。

不能在文件中插入字节（向两端扩展文件）或删除字节（向中间收缩文件），字节将被写入到文件的末尾，每次一个字节，文件长度随之增加。文件可以缩短或增加到任意长度，并可以删除字节或添加零字节。

同一文件可同时被两个或两个以上的进程并发读写。其结果依赖于每个I/O请求发生的次序，且往往不可预见。维护次序的机制可通过文件锁功能和信号量（semaphore）来实现，它们是进程可以检测和设置的系统范围内的标志（详见1.1.7节）。

文件没有名称，只有被称为索引节号（i-number）的数字。一个索引节号是一批信息节点（i-node）的索引，这些信息节点存储在每个包含UNIX文件系统的磁盘区域的前部。每一个信息节点包含关于一个文件的重要信息。有趣的是，这个信息既不包括文件名称，也不包括数据字节。它包括以下内容：文件类型（常规、目录、套接字等）、链接数量（稍后给出解释）、文件所有者ID和组ID、三组访问权限（用户权限、组权限和其他权限）、字节数、最后访问时间、最后修改时间、状态改变时间（信息节点最后被修改的时间），当然还有指向包含文件内容的磁盘块的指针。

^① 本书在末尾提供了参考文献。

1.1.1.2 目录和符号链接

由于使用索引节号标识文件不方便，所以引入目录（directory）来使用名称标识文件。在实际中，几乎总是用目录来进行文件访问。

概念上，每个目录包含一个两列表，一列是文件名称，另一列是对应的索引节号。名称/信息节点对被称为链接（link）。当UNIX内核通过名称访问文件时，它会自动查询目录来寻找索引节号，然后获得对应的、包含文件更多信息的信息节点（如谁有权访问该文件）。如果允许访问数据本身，信息节点会给出磁盘上的存储位置。

目录和常规文件非常类似，也占用一个信息节点并拥有数据。因此，在一个目录中，对应于一个特定名称的信息节点也可以是另一个目录中的信息节点。这样就允许用户按照UNIX用户所熟悉的层次结构来安排文件。例如，memo/july/smith的路径（path）指示内核获取当前目录的信息节点以确定其数据字节的存储位置，然后从这些数据字节中寻找memo，获取对应的索引节号，并获取其信息节点以确定memo目录的数据字节位置，再从这些数据字节中找出july，获取对应的索引节号，并获取信息节点以确定july目录的数据字节位置，最终找到smith，并获取对应的信息节点，该信息节点就是和memo/july/smith相关联的信息节点。

在对相对路径（relative path）（从当前目录开始的路径）进行追溯时，内核怎样知道应该从何处开始呢？只需要简单地地为每个进程保存其当前目录的索引节号即可。当进程改变其当前目录时，它必须提供新目录的路径，该路径对应的索引节号作为新的当前目录的索引节号来保存。

以“/”为开头的绝对路径（absolute path），其起点在根（root）目录。内核为根目录简单地保留了一个索引节号（比方说是2），这在文件系统初次设立时就已建立。通过系统调用可以改变进程的根目录（变为不是2的其他索引节号）。

由于目录的两列结构直接由内核使用（这是内核关心文件内容的少量特例之一），同时因为一个无效目录的存在可能很容易使整个UNIX系统崩溃，所以程序（即使以超级用户权限来执行）不能像常规文件那样写目录。要想对目录进行操作，程序就必须使用一套特定的系统调用。总而言之，合法的写操作仅包括链接的增加和去除。

可以使用同一个索引节号来标识相同或不同目录中的两个或两个以上的链接。这意味着同一文件可能拥有一个以上的名称。但在通过给定路径访问文件时不会产生多义性，因为只会找到一个索引节号。当然，也可以通过其他路径获取索引节号，但结果也是一样的。然而，当在目录中删除某个链接时，不能立刻明确其信息节点以及关联的数据字节是否也可被丢弃。这就是信息节点中包含链接计数的原因。去除指向信息节点的某个链接，只会减少其链接计数；当计数值为零时，内核就删除该文件。

对于为什么不像常规文件一样存在目录的多重链接这一问题，并没有结构上的原因。然而，它会使得扫描整个文件系统命令的编程变得更为复杂，因此，大多数内核将其排除在外。

利用索引节号指向文件的多重链接只在这些链接都属于同一文件系统时才起作用，因为索引节号的唯一性仅限于一个文件系统内部。为避免这种情况带来的后果，还可以采用符号链接（symbolic link），它将要链接的文件路径保存在一个真实文件的数据部分中。这种方式比在某处制作一个第二目录的开销要大，但是它更为常用。用户不需要读和写这些符号链接文件，而仅需使用为符号链接所准备的专门系统调用。

1.1.1.3 特殊文件

特殊文件（special file）通常是某种类型的设备（如CD-ROM驱动或通信链路等）。[⊖]

⊖ 有时，命名管道也被当成是特殊文件，但本书将其另归为一类。

设备特殊文件有两种基本类型：块和字符。块特殊文件（block special file）遵循以下特定模型：设备包含固定长度块（如每块4096字节）的数组，并使用内核缓冲区（buffer）池作为高速缓冲存储器来加速I/O操作。字符特殊文件（character special file）根本不用遵循任何规则，它们进行I/O操作时可以传输很小的块（字符），也可以传输很大的块（磁盘磁道），因此，它们不太规则，不适合使用缓冲区缓存。

同一个物理设备可以同时具有块和字符特殊文件，事实上，磁盘就是这样的物理设备。内核中的文件系统代码通过一个块特殊文件访问常规文件和目录时，可从缓冲区缓存中获益。有时，尤其是高性能的应用程序，需要进行更直接的访问。例如，一台数据库管理器可以完全跳过文件系统，使用一个字符特殊文件访问磁盘（但不是文件系统使用的区域）。大多数UNIX系统拥有这种类型的字符特殊文件，它可以使用直接存储器访问（direct memory access, DMA）——一种可以成数量级提高性能的方式，直接在进程的地址空间和磁盘之间传输数据、进行存取。其另一个好处是错误检测的鲁棒性更好，因为缓冲区缓存的间接性会使得错误检测难以实现。

一个特殊文件有一个信息节点，但在磁盘上没有任何指向该信息节点的数据字节。相反，在信息节点的那部分数据字节中包含了一个设备号（device number），它是一个列表的索引，该列表是内核用来查找一种叫设备驱动程序（device driver）的子程序集合的。

当执行系统调用对特殊文件进行操作时，将调用相应的设备驱动程序。接着发生的事情完全取决于设备驱动程序的设计者；因为驱动程序是在内核中运行的，而不是作为用户进程运行的，所以它能够访问（甚至修改）任意内核部分、任意用户进程以及计算机本身的任意注册变量或内存。因为向内核中添加新的驱动程序相对简单，所以采用该方式提供一个钩子，不仅可实现与新型的I/O设备进行交互，而且还可做许多事情。一种最流行的办法是，让UNIX做一些它的设计者从未打算让它做的事。想像用认可的方法去做一些很疯狂事情的感觉。

1.1.2 程序、进程和线程

程序（program）是指在磁盘常规文件中存储的指令（instruction）和数据（data）的集合。在它的信息节点中，文件被标记为可执行的，文件的内容是按照内核建立的规则进行安排的。（这是内核关注文件内容的另一个例子。）

程序员可以选用任何方法创建可执行文件。只要文件内容遵循规则并将文件标记为可执行的，程序就能运行。在实际中，常按如下步骤进行：首先，将使用某种程序设计语言的源程序（如C或C++）输入到常规文件中，通常称为文本文件（text file），因为它是按照文本行的方式安排的。其次，创建一个称作目标文件（object file）的常规文件，其中包含源程序的机器语言的翻译结果。这项任务由编译器或汇编程序（它们本身也是程序）来完成。如果这个目标文件是完备的（没有缺少子程序），就标识为可执行的，认为其可以运行。如果不完备，就使用链接器（linker）（在UNIX的术语中有时叫装载器）来链接这一目标文件和其他那些先前产生的文件，那些先前产生的文件取自一个称为库（library）的目标文件的集合。除非链接器找不到它所寻找的东西，它的输出才是完全的、可执行的。^①

为了运行一个程序，首先要请求内核产生一个新进程，它是程序运行的环境，由三个部分组成：指令段（instruction segment）^②、用户数据段（user data segment）和系统数据段

^① 这不像解释性语言如Java、Perl、Python和shell脚本那样工作。对它们来说，可执行的是解释器，即使将程序编译成某种中间代码，也仅仅是为解释器提供数据，并且UNIX内核决不会见到或关心它们。内核的客户是解释器。

^② 在UNIX的专门术语中，将指令段叫做“文本段”，这里是为了避免术语的混淆。

(system data segment)。程序是用来初始化指令和用户数据的。初始化完成后，进程开始脱离运行的程序。虽然现代程序员通常并不修改指令，但会修改数据。另外，进程可能会请求程序没有声明的资源（更多的内存、打开的文件等）。

当进程运行时，内核将记录所有那些对进程数据的相同部分进行读和写的线程。每一个线程 (thread) 是一个单独的指令控制流。（实际上，每一个线程都有自己的堆栈。）在编程时，除非执行特殊的系统调用来创建其他线程，否则只能从一个线程开始。因此，初学者可以认为进程是单线程的。^①

几个并发运行的进程可以由同一个程序初始化。然而，在这些进程之间不存在功能性的关系。通过安排这样的进程来共享指令段，内核也许能节省内存，但参与的进程并不能察觉这样的共享。相反，在同一进程的线程之间具有很强的功能性关系。

进程的系统数据 (system data) 包括诸如当前目录、打开文件的描述符、累计CPU时间等属性。进程不能直接访问或修改它的系统数据，因为它处于进程的地址空间之外。然而，可以通过多种系统调用对这些属性进行访问或修改。

由内核在当前运行进程的基础上产生的新进程称为原进程的子进程 (child process)，原进程称为父进程 (parent process)。子进程继承了父进程的大多数系统数据属性。例如，如果父进程拥有打开的文件，这些文件对子进程同样处于打开状态。这种类型的继承是UNIX操作的本质，正如本书所要展示的。这不同于创建新线程的线程，同一进程中的线程在大多数方面都是平等的，不存在继承关系。所以线程都平等访问全部数据和资源，而不是它们的副本。

1.1.3 信号

内核可以向进程发送信号 (signal)。信号可以由内核自身产生，从进程自身发送，从其他进程发送或以用户的名义发送。

例如：段式违例信号是一个由内核发出的信号，是在进程试图访问其地址空间以外的内存时发出的；一个进程发送给自身的信号例子是abort信号，由abort函数发出，用来终止一个主存储器信息转储的进程；一个由进程发送给其他进程的信号例子是termination信号，由几个相关进程之一在决定终止所有相关进程时发出；一个由用户发出的信号例子是中断信号，当用户键入Ctrl-c时，发送给由该用户创建的所有进程。

信号的类型约有28种（某些版本的UNIX可能有所出入）。除kill信号和stop信号外，进程对收到的其他信号，能控制响应行为。它既可以接受默认的行为，通常结果是终止进程；也可以忽略信号；还可以在收到信号时，捕获信号并执行某个函数（称为信号处理程序）。信号的类型（如SIGALRM）是作为参数传递给处理程序的。然而，处理程序无从得知是谁发送的信号。^②当信号处理程序返回时，进程从断点继续运行。有两种信号没有被内核使用，这可能是应用程序用于其自身的。

1.1.4 进程ID、进程组和会话

每一个进程都有自己的进程ID (process-ID)，该ID为正整数，在任何时刻都确保是唯一的一个。除了一个进程之外，其他每个进程都有其父进程。

在进程的系统数据中，同样保留其父进程ID (parent-process-ID)，即其父进程的进程ID。

^① 并不是每一个UNIX版本都支持多线程。多线程是POSIX线程（缩写为“pthread”）选项特征的一部分，它们是在20世纪90年代中期被引入的。关于POSIX更多的内容见1.5节，有关线程的内容见第5章。

^② 对于28种基本信号来说是这样的。实时信号选项为某些可用的信息添加了一些信号。详见第9章。

如果子进程的父进程在子进程结束前终止，导致其子进程成为“孤儿”，那么其父进程ID将变为1（或其他固定值）。该值是在启动时创建的初始化进程的进程ID，是所有其他进程的“祖先”。换句话说，初始化进程收留所有的“孤儿”。

在实现子系统时，程序员有时会选用一组相关进程来实现，而不是单个进程。UNIX内核允许将这些相关进程组织成进程组（process group），进程组可以进一步组织成会话（session）。

会话的成员之一是会话领导者（session leader），每个进程组的成员之一是进程组领导者（process-group leader）。对于每一个进程，UNIX都会记录其进程组领导者和会话领导者的进程ID，这样，每个进程都能在这种层次结构中找到自己的位置。

内核提供的系统调用是用于向进程组的每个成员发送信号的。通常被用于终止整个组，但任何信号都可以通过这种方式广播。

UNIX外壳程序（shell）一般会为每个登录创建一个会话。它们通常为流水线进程创建一个单独的进程组；在此上下文中，进程组也可称为任务（job）。但这只是外壳处理问题的方式；内核允许更灵活的处理方式，使得每个单独的登录都可以建立任意数量的会话，每个会话所拥有的进程组都可以用它喜欢的方式来建立，只是需要保持其层次结构。

其工作方式如下：任何进程都可以脱离原先所在的会话，并成为自身会话（只包含一个进程的进程组）的领导者。然后它可创建子进程并扩充为新进程组。在这个会话中，还可建立其他进程组，并将进程分配到不同的进程组中（这种分配也可改变）。因此，一个单独的用户可能会运行这样一个会话：比如该会话由10个进程构成，而这10个进程又被包含于3个进程组中。

会话和它的进程可以拥有控制终端（controlling terminal）——由会话领导者打开的第一个终端设备；这时会话领导者成为控制进程（controlling process）。通常，用户进程的控制终端是用户登录所用的终端。当建立新会话时，新会话中的进程不再拥有控制终端，除非已存在一个打开的终端。在一些所谓的后台程序（daemon）^①（Web服务器、cron工具等）中，并不存在控制终端，因为这些后台程序一旦启动，就会和启动它们的终端分开运行。

通过组织可以实现会话中仅有一个进程组（即前台任务）可以访问终端，并且也可以在前台和后台之间来回移动进程组，这称为任务控制（job control）。在试图访问终端的后台进程组还没有被移至前台之前，先将其挂起。这样可以阻止其非法获取来自前台任务的输入或修改前台进程组的输出。

如果不采用任务控制（一般只有shell使用它），那么会话中的所有进程则都是有效的前台进程，都可以平等地自由访问终端，即使灾难就在前方（随之而来的就是灾难）。

终端设备驱动程序可以发出中断、退出和挂机信号，这些信号从该终端发向以其为控制终端的前台进程组中的每个进程。例如，除非采取预防措施，否则挂断终端（例如，关闭一个telnet连接）会终止该组中的所有用户进程。为避免此类情况，可设置进程忽略挂断信号。

另外，无论什么原因，只要会话领导者（控制进程）终止，所有前台进程组中的进程都会收到一个挂断信号。因此，简单的注销通常会终止用户的所有进程，除非采取特殊措施，或者使它们成为后台进程，或者使它们免受挂断信号的影响。

总之，每个进程都有4个相关的进程ID号：

- 进程ID：能唯一标识这个进程的正整数；
- 父进程ID：其父进程的进程ID；
- 进程组ID：进程组领导者的进程ID。如果和进程ID相同，则该进程即为组的领导者；

^① “demon”和“daemon”是同一个单词的两种拼写，但前者的意思是魔鬼，而后者的意思是介于神和人之间的一种超自然的精灵。行为不端的daemon可以称为demon。

- 会话领导者ID: 会话领导者的进程ID。

1.1.5 权限

用户ID (user-ID) 是和password文件 (/etc/passwd) 中用户的登录名称 (login name) 相关联的一个正整数。当用户登录后, login命令会将此ID作为已创建的第一个进程 (login shell) 的用户ID。其他进程将从shell继承这个用户ID。

用户也会被划分到组 (不要和进程组混淆) 中, 这些组也会有ID, 称为组ID (group-ID)。用户的登录组ID被当成其登录外壳 (login shell) 的组ID。

组是在组文件 (/etc/group) 中定义的。登录后, 用户可以转到他所隶属的其他组中。这样将改变处理请求 (通常是shell, 通过newgrp命令) 的进程的组ID, 并被所有派生进程所继承。由于一个用户可能是多个组的成员, 所以进程也拥有一个补充组ID列表。对大多数场合来说, 在进程的组权限存在争议的情况下, 将会检查该列表, 使得用户不必经常手工转换组。

这两个用户和组的登录ID被称为实际用户ID (real user-ID) 和实际组ID (real group-ID), 因为它们代表了实际用户, 即登录系统的人员。另两种和进程相关的ID是有效用户ID (effective user-ID) 和有效组ID (effective group-ID), 它们通常和对应的实际ID相同, 但也可能不同, 就像稍后所看到的那样。

有效ID经常用于决定权限, 真实ID用于审计以及用户到用户的通信。一个表明了用户的权限, 另一个表明了用户的身份。

每个文件 (常规文件、目录、套接字文件等) 的信息节点中都具有一个所有者用户ID (owner user-ID) (简称所有者) 和一个所有者组ID (owner group-ID) (简称组)。同时, 信息节点中还包含有三组权限位, 每组三位 (共计9位)。每组包含一个读权限位 (read permission bit)、一个写权限位 (write permission bit) 和一个执行权限位 (execute permission bit)。位为1时代表允许权限, 为0时代表拒绝权限。三组权限中, 一组用于所有者, 一组用于组, 一组用于其他用户 (不在前两类中)。表1-1表明了位的分配 (位0是最右位)。

表1-1 权限位

位	含 义	位	含 义
8	所有者读	3	组执行
7	所有者写	2	其他用户读
6	所有者执行	1	其他用户写
5	组读	0	其他用户执行
4	组写		

权限位经常用八进制数来表示。例如, 八进制数775表示具有所有者及组的读、写和执行权限, 但同时只具有其他用户的读和执行权限。ls命令显示其权限组合为rwxrwxr-x; 其二进制表示为111111101, 直接转化为八进制就是775。(转化为十进制是509, 但这种表示通常是没有用的, 因为这几个数字和权限位之间没有直观的联系。)

权限系统能够决定指定进程能否对指定文件执行某种预期行为 (读、写或执行)。对常规文件来说, 所有这三个行为的含义都是明显的。对目录来说, 读行为的含义是明显的。目录的“写”权限的含义是指使用系统调用来修改目录 (增加或删除链接) 的能力。“执行”权限意味着在路径中使用该目录的能力, 有时又称为“搜索”权限。对特殊文件来说, 读和写权限意味着执行读或写系统调用的能力。其具体含义取决于设备驱动程序的设计者。特殊文件的执行权限是没有任何意义的。

权限系统采用以下算法来决定是否可以授予某种权限：

- 1) 如果有效用户ID是0，则立即授予权限（有效用户是超级用户）；
- 2) 如果进程的有效用户ID和文件的用户ID相符，则使用所有者位组来判断该行为是否被允许执行；
- 3) 如果进程的有效组ID或一个补充组ID和文件的组ID相符，则用组权限位组进行判断；
- 4) 如果与用户ID和组ID都不符合，则进程属于“其他用户”，此时使用第三组权限位组进行判断。

以上步骤是按顺序执行的，因此，如果在步骤3中访问被拒绝（例如由于组的写权限被拒绝），则该进程不能执行写，虽然依照其他用户权限（如步骤4）可能允许写。组的权限比其他用户的权限更为严格，这种情况可能不常见，但这确实是有效的方法。（假设要给一组雇员召开“惊喜聚会”——当事人事先不知道该聚会，那么除这组雇员外，参加聚会的其他人都应该能读到该聚会的邀请函。）

还有其他一些称为“改变信息节点”的行为，只有所有者或超级用户可以进行。这些行为包括改变文件的用户ID或组ID、改变文件权限和改变文件的访问或修改时间。作为特例，文件的写权限允许将其访问和修改时间改为当前时间。

有时，可能需要用户暂时性地取得其他用户的特权。例如，当运行passwd命令改变口令时，可能希望有效用户ID是超级用户，因为只有root才可以写入口令文件。这可以这样实现：将passwd命令的所有者设为root（超级用户的登录名称），然后打开passwd命令的信息节点中的另一个权限位，称为设置用户ID（set-user-ID）位。运行打开此位的程序，会将进程的有效用户ID改变为包含该程序的文件的所有者的用户ID。由于是使用有效用户ID而不是实际用户ID来确定权限，所以这允许用户临时取得其他用户的权限。设置组ID（set-group-ID）位的使用与之相似。

由于两种用户ID（实际用户ID和有效用户ID）都从父进程继承到子进程，因此使用设置用户ID功能，可以在很长的时间内运行某个有效用户ID。su命令就是这么做的。

这里存在一个潜在的漏洞。假设进行以下操作：将sh命令文件复制到用户自己的目录（因此用户就成为复制文件的所有者）；然后使用chmod打开设置用户ID位，并使用chown命令将文件的所有者改为root。现在，运行用户的sh拷贝，就会取得root的特权！幸运的是，这个漏洞在很久以前就被堵住了。如果不是超级用户，改变文件的所有者，会自动清除设置用户ID位和设置组ID位。

1.1.6 进程的其他属性

进程的系统数据段中还记有其他几种有趣的属性。

进程打开的每个文件（常规文件、特殊文件、socket文件或命名管道文件）都有一个打开文件描述符（file descriptor）（从0到大约1000的整数），而进程产生的每个未命名管道（见1.1.7节）都有两个打开文件描述符。子进程并不从其父进程继承打开文件描述符，而是复制它们。虽然如此，它们都索引至相同的全系统打开文件表，再加上其他因素，可意味着父进程和子进程共享相同的文件偏移（下一次读或写的字节位置）。

进程的优先级由内核调度程序使用。任何进程都可通过系统调用nice来降低其优先级；通过相同的系统调用，超级用户进程可以提高其优先级（即，不用nice）。从技术上讲，nice有一个属性叫做nice值，该属性仅有一个参数用于计算实际的优先级。

进程的文件大小限制可能会（通常就是这样的）小于全系统的限制；这是为了防止初学者写入超出其控制以外的文件。超级用户进程可以扩大限制范围。

在本书描述相关系统调用的部分中，还会讨论更多的进程属性。

1.1.7 进程间通信

在最早期（1980年以前）的UNIX系统中，进程之间可以通过共享文件偏移、信号、进程跟踪（process tracing）、文件和管道（pipe）进行通信。后来，又加入了命名管道（named pipe）（FIFO），以及信号量、文件锁（file lock）、消息（message）、共享存储器（shared memory）网络套接字。如同我们在本书中将要看到的那样，上述11种机制中，没有一种能完全满足要求。这就是目前存在11种机制的原因！由于信号量、消息和共享存储器都存在两种版本，因此这样算来可能还要更多。从非常古老的、来自AT&T的System V UNIX的“SYSTEM V IPC”，到非常新的、来自20世纪90年代初期成立的一个实时标准团体的“POSIX IPC”系统，几乎每个版本的UNIX（包括FreeBSD和Linux）都含有这11种最古老的机制，而主要商业版本的UNIX（如Sun的Solaris或HP的HP/UX）则包含所有这14种机制。

共享文件偏移（shared file offset）很少用于进程间通信。理论上，如果一个进程将文件偏移量定位到文件的某个位置，那么第二个进程就可以找到该位置。该位置（比如介于0~100间的某个数）就是通信数据。由于进程必须涉及共享文件偏移量，因此也可以使用管道。

在进程仅需要指向其他进程时，有时使用信号。例如，每当打印文件假脱机时，打印假脱机程序就向实际打印进程发出信号。但是，对大多数应用程序来说，信号不能传递足够的信息。另外，信号会中断接收进程，与在接收者准备好后再获得通信的方式相比，其编程变得更为复杂。信号的主要用途只是终止进程或指出异常事件。

利用进程跟踪，父进程可以控制其子进程的运行。由于父进程可以读写子进程的数据，所以两者可以自由通信。进程跟踪仅由调试器使用，因为对于一般应用来说，这过于复杂并且不安全。

文件是进程间通信最常见的形式。例如，可以通过某个运行vi的进程写文件，然后通过另一个运行python的进程运行该文件。然而，如果两个进程是并发运行的，使用文件就不太方便了，原因有两个：首先，读进程可能会超过写进程，当读到文件末尾时，读进程可能会认为通信已经结束。（这可以通过一些复杂的编程进行控制。）其次，两个进程的通信越长，文件就会越大。有时进程间通信可能会持续几天或几周，传输几十亿字节的数据，这会很快耗尽文件系统。

信号量使用空文件，也是一种传统的UNIX技术。它利用了UNIX系统创建文件方式的某些特点。更详细的内容见2.4.3节。

最后，提出以下建议：管道解决了文件的同步问题。管道不是常规文件，虽然它有信息节点，但不存在指向它的链接。管道的读写类似于文件的读写，但存在一些显著的不同：如果读进程超过了写进程，读进程将被阻塞（停止运行一会儿），直到有了更多的数据。如果写进程远远超过了读进程，写进程将被阻塞直到读进程有机会赶上来，这样内核不会有太多的排队数据。最后，一旦字节被读取，它就永久消失，因此通过管道连接的进程，长时间运行也不会填满文件系统。

对shell用户而言，管道是众所周知的，可以输入如下的命令行：

```
ls | wc
```

来查看拥有多少文件。然而，正如在第6章将要看到的那样，内核工具远比shell提供的工具通用。

但是，管道有三个主要的缺点：第一，通过管道通信的进程之间必须是相关的，典型的有父子关系或兄弟关系。对于许多应用程序而言，这种约束太严格了，诸如这样的应用程序：一个进程是数据库管理器，而另一个进程是需要访问数据库的应用程序。第二个不足是，无

法保证超过本地设置最大值（比如4096字节）的写入的原子性，当存在多个写入进程时，必须禁止使用管道，否则它们的数据可能会相互混合。第三个不足之处是管道可能会很慢。必须首先将数据从写入用户进程复制到内核，然后再复制到读进程。虽然没有执行实际的磁盘I/O，但对某些关键应用程序来说，复制过程本身花费的时间也相当长。由于这些缺点的存在，引入了空想家方案（fancier scheme）。

命名管道，也称为FIFO（FIFO代表“先进先出”），用以解决管道的第一个缺点。命名管道作为特殊文件而存在，任何具有权限的进程都可以打开它进行读写。命名管道也易于编程，正如将在第7章中所述的那样。

命名管道有什么不足？它们不能消除管道的第二个缺点和第三个缺点：即大量写入时可能会发生交叉存取，有时可能会很慢。对于大多数关键应用程序，都可以使用更新的进程间通信特性（例如消息或共享存储器），但难度会更大一些。

信号量（在计算机领域中）是一个用于防止两个或更多的进程在同一时间访问同一资源的计数器。如上文所述，文件也可用作信号量，但对许多应用程序来说其开销太大。UNIX系统有两类完全不同的信号量机制，一类属于SYSTEM V IPC，另一类属于POSIX IPC系统。（在1.5节中将介绍POSIX。）

文件锁，实际上是一种具有特殊目的的信号量，可以防止两个及两个以上的进程访问文件的同一部分。通常只在进程强制检查的情况下有效；其较弱的形式称为建议锁，较强的形式称为强制锁。后者无论进程是否检查都是有效的，但它并不是标准的，而且仅在某些UNIX系统中可用。

消息是一种可以发送到信息队列的少量（如500字节）数据。消息可以是不同类型的。任何拥有适当权限的进程都可以从队列中接收消息。有多种选择：可以是第一条消息，也可以是指定类型的第一条消息，还可以是一组类型的第一条消息。和信号量一样，也存在SYSTEM V IPC消息系统调用和完全不同的POSIX IPC系统调用。

共享存储器潜在地提供了最快的进程间通信。可将同一存储器映射到两个或两个以上进程的地址空间中。一旦数据被写入了共享存储器，就可立即被读取。可用信号量或消息来同步读进程和写进程。确切地说，共享存储器存在两种版本，细心的读者可以猜出来是哪两种。

最后也是最好的一种方法是：使用一组称为套接字的系统调用（该组中的其他系统调用的名称分别为bind、connect和accept）的网络进程间通信。和上文提到的其他机制不同，通过socket通信的进程不必运行在同一台机器中。它可以在另一台机器上，或者局域网或Internet的什么地方。另一台机器甚至不必运行UNIX，可以运行Windows或其他操作系统。在某种意义上，它甚至可以是一台网络打印机或无线设备。

为某个特定的应用程序选择IPC机制不是件容易的事，因此在第7章和第8章中，花费了大量篇幅对它们进行了比较。

1.2 UNIX的版本

1969年，在AT&T贝尔实验室，Ken Thompson和Dennis Ritchie开始将UNIX作为一个研究项目，此后很快在AT&T的内部系统（如自动电话修理呼叫中心）中得到广泛使用。那时，AT&T还未开展商业计算机生产或商业软件销售的业务，但在20世纪70年代早期，它将UNIX提供给一些大学用于教学目的，其源代码仅透露给那些持有其许可的其他大学。到了70年代末期，AT&T也开始将源代码许可给商业销售者。

许多（如果不是绝大多数）获得许可者自己对UNIX进行了改进，以使其能够适应新的硬

件、加入设备驱动或仅仅是对其进行调整。其中有两个改进可能称得上是最重大的改变，它们都来自加州大学伯克利（Berkeley）分校：网络系统调用（“socket”）和对虚拟存储器的支持。结果是，遍及全世界的大学和研究工作实验室都使用BSD系统，加州大学伯克利分校软件（Berkeley Software Distribution, BSD），即使它们仍然要从AT&T获得许可。一些商业销售者，如知名的Sun Microsystems，同样是从BSD UNIX开始的。Sun的创办者之一，Bill Joy，就曾是一位BSD的主要开发者。

贝尔实验室同样在继续开发工作，到80年代中期，两个系统已经岔开很远了。（AT&T的系统那时称为系统V。）二者都支持虚拟存储器，但其网络系统调用已经完全不同了。系统V的进程间通信设施（消息、共享存储器和信号量）是由BSD系统以不同的方式实现的，BSD除几乎改变了所有的命令外，还添加了许多命令，其中以vi最为著名。

另外，还有大量其他的UNIX变体，包括少许使用未经AT&T许可的源代码的克隆产品。但是，大多数UNIX世界都可以划分为两大阵营，即BSD和系统V。前者几乎包括了所有的学术界和一些重要的工作站制造者，后者则包括了AT&T自己和一些商业销售者。几乎所有的计算机科学专业的学生都学习BSD系统。（当他们来到贝尔实验室时，他们带来了自己喜爱的命令和所有的UNIX知识。）令人啼笑皆非的是，无论AT&T还是加州大学，事实上都不想介入商业软件的业务！（AT&T是这么说的，但事实并非如此。）

电气和电子工程师协会（Institute of Electrical and Electronics Engineers, IEEE）在20世纪80年代中期开始尝试对两种UNIX系统调用及命令进行标准化，继续接手过去称为/usr/group的工业团体遗留的工作。1988年IEEE发布了它的第一个系统调用标准，其官方名称为IEEE Std 1003.1-1988，本书缩写为POSIX1988。^①第一个命令（如vi和grep）标准颁布于1992年（IEEE Std 1003.2-1992）。在经过少量修正后，POSIX1988被国际标准化组织（International Organization for Standardization, ISO）采用，作为国际标准，称为POSIX1990。

上述标准仅对应用程序接口（Application Program Interface, API）的语法和语义进行了标准化，并不涉及底层实现。因此，无论其原始结构或内部结构如何，任何具有足够功能性的系统都能够遵从POSIX1990或者它的后续标准，甚至像Microsoft Windows或Digital（现在的HP）的VMS之类的系统同样可以做到。如果应用程序遵守操作系统API标准和编写语言（如C++）的标准，同时其目标系统也遵从相关标准，那么应用程序的源代码就能够移植，无需改变就可以编译和运行。在实际中，应用程序、编译器和操作系统中都存在bug，而且几乎每个严谨的应用程序都不得使用某些非标准的API，因此，进行移植时确实需要做一些工作。遵从标准，可以将移植工作量减少到易于处理的程度，和应用标准前相比要容易得多。

POSIX1990固然是一种极大的成就，但还不足以统一系统V和BSD阵营，因为它不包含重要的API，比如BSD socket和系统V中的消息、信号量和共享存储器。同样，为了UNIX的新应用程序，甚至有更新的API仍不断地被引入，主要有实时程序（和真实世界交互的应用程序，如汽车引擎管理）和线程。幸运的是，IEEE也包含为实时和线程工作的标准工作组（1.5节详述了POSIX1990之后的标准）。

在1987年，AT&T和SUN（使用了BSD派生的UNIX）认为它们能够通过共同开发一个新系统来统一UNIX的两个主要版本。这震动了业界的其他公司，它们建立了开放软件基金会（Open Software Foundation, OSF），并最终制作出了它们自己的UNIX版本（然而仍然包含AT&T的许可代码）。因此，在20世纪90年代早期，存在有BSD、AT&T/Sun UNIX、pre-OSF UNIX版本以及OSF版本。

^① POSIX是Portable Operating System Interface的缩写，其发音是pahz-icks。

当Microsoft Windows一统天下的时候，UNIX却好像是起了内讧。由于X/Open公司在将重要的API收录进标准方面比IEEE更为积极，这促使业界都支持该公司。在其最新的标准中，居然定义了1108条功能接口！

到20世纪90年代中期，所有的事情都慢慢开始趋于稳定，就在这时，Linux出现了。1991年在Usenet的新闻组中，悄悄出现了一个名为“Free minix-like[Ⓞ] kernel sources for 386-AT”的帖子，张贴者是一位名叫Linus Torvalds的研究生。帖子的开头是这样的：

你渴望使用minix-1.1的那些好日子吗？那个时候自己写设备驱动的年代？你手头是否没有好的项目，因而极度渴望想开始学习一种可以自由修改以满足自己需要的操作系统？当你发现一切都工作在minix之上时会不会非常沮丧？你想不用再整夜不眠就能获得一个工作得很漂亮的程序吗？那么，这个帖子就是为你预备的：-)

正如我一个月(?)前提到的，我一直在为AT-386计算机开发一种类似minix的自由版本而工作。这项工作终于到了可以使用(虽然可能和你想要的还不太一样)的阶段，我想公开其源代码，使之可以得到更广泛的传播。它目前的版本仅为0.02 [+1 因为已经打了(很少量的)补丁]，但是在其上本人已成功运行了bash/gcc/gnu-make/gnu-sed/compress等。[Ⓞ]

到20世纪90年代末，Linux已经发展成为一个严肃的操作系统，在很大程度上，这要感谢数百名开发者在其自由和开放的源代码上付出的辛勤工作。目前已经有了基于Intel的PC机的Linux二进制发行版的商业销售商(如Red Hat、SuSE和Mandrake)，以及装配Linux计算机的硬件制造商(如Dell、IBM甚至Sun)。他们都按照Linux许可的要求提供源代码，这和AT&T的许可完全相反！

在此期间，虽然伯克利的程序员已经对内核进行了15年的大量改变，但是他们开发的系统中仍然包含有AT&T的许可代码。他们开始移去AT&T的代码，但在完成之前就用了经费。在20世纪90年代早期，他们发布了一个虽不完整但已不包含AT&T许可代码的系统，称为4.4BSD-Lite。(如果他们早点完成这项工作，Torvalds就会从它而不是Minix开始了。)马上就有几个团体开始进行充实4.4BSD-Lite的工作，这些工作成就了今天的FreeBSD(为Intel CPU或少量其他处理器而设计的)、NetBSD/OpenBSD(可移植的)、WinRiver的BSD/OS(商业支持的)和Darwin(Mac OS X中的UNIX)。唉，伯克利小组已经不在其中了。

到今天，UNIX系统有以下三种主要的变种：

- 商业的、非开放的系统，其历史基于AT&T的SYSTEM V或BSD(Solaris、HP/UX、AIX等)；
- 基于BSD的系统，其中FreeBSD最为著名，Darwin的知名度迅速增长；
- Linux。

它们中哪一种才是“UNIX”呢？拥有UNIX商标的AT&T的UNIX开发组织，已经在1993年卖给了Novell。Novell又马上把UNIX商标转给了X/Open。1996年，OSF和X/Open合并，成立了Open Group。该团体目前开发了一个易于理解的、称为Single UNIX Specification[SUS2002]的标准。只要系统满足Open Group的要求，任何系统都可以合法地使用“UNIX”商标。主要的硬件销售商都拥有带UNIX商标的系统——甚至连IBM的大型机MVS(现在称为OS/390)都带有UNIX商标。但是，开源系统Linux和FreeBSD都没有该商标，即使它们声称遵

Ⓞ Minix是阿姆斯特丹自由大学的Andrew Tanenbaum开发的一个小的指令OS，第一次发行是在1987年。它不使用AT&T代码。

Ⓞ 原始消息和1981年以来其他7亿Usenet消息都被归档在Google Group中了(www.google.com/grph)。

循了某种POSIX标准。方便起见，可以把它们叫做“类UNIX”系统。

1.3 使用系统调用

本节将解释如何利用C或其他语言来使用系统调用，并对其正确用法提出了一些建议。

1.3.1 C (和C++) 绑定

C或C++程序员实际上是怎样使用某个系统调用的？答案是：就和任何其他函数调用一样。例如，可采用如下方式调用read：

```
amt = read(fd, buf, numbyte);
```

函数read的实现和UNIX实现不同。通常，它是小型函数，通过运行某些特定代码，它可以将控制从用户进程转换到内核，然后返回结果。实际工作是在内核中完成的，这就是它叫做系统调用而不是库程序的原因，后者仅在用户空间完成全部工作，如qsort或strlen。

然而请记住，由于系统调用涉及了两次上下文转换（从用户到内核再转回来），所以和进程自身地址空间内的简单函数调用相比，其花费的时间更长。因此，应避免过多地使用系统调用。在2.12节中，当我们探究缓冲I/O时，还会强调这一点。

每个系统调用都需要在头文件中进行定义，在执行调用前，必须确保已包含了正确的头文件（有时需要包含一个以上的文件）。例如，调用read需要包含头文件unistd.h，形式如下：

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

通常，在一个头文件中，会对多个系统调用进行定义（头文件unistd.h中定义了约80个），因此，对于典型的程序来说，仅需要包含几个头文件即可。即便程序中还含有大量的标准C[⊖]函数（例如string.h），其数量仍然易于处理。将实际上不需要的头文件包含在内并不会带来害处，因此可将最常用的头文件收集起来包含到一个主控头文件中，以后仅需包含该文件即可。本书的主控头文件，是1.6节中提出的defs.h。在本书的示例代码中，虽然没有列出包含该头文件的语句，但读者应假定它被包含在了本书的每个C文件中。

很不幸，由于各种标准中存在的多义性或实现者方面的错误理解，可能会造成头文件冲突，所以，有时可能需要变换包含的顺序或对C预处理程序使些技巧，以便能够全部正常输出。虽然你能在defs.h中看出一点儿这样的情况，但无论如何你都察觉不到包含在代码中的一些奇特的事情。

对于给定的功能，是应该采用系统调用来实现，还是应该采用库函数来实现，并没有什么标准。因此，在本书介绍或在例子程序中展示所使用的某种实现时，并不特意区分二者。同时，请不要过分严格地按照字面意思来理解本书中对术语“系统调用”的使用，它仅意味着该功能通常采用这种实现方式。

1.3.2 其他语言绑定

虽然主要的UNIX标准文档（POSIX和SUS）是用C来描述接口的，但系统开发者考虑到C只是许多程序设计语言中的一种，所以他们也定义了其他标准化语言（如Fortran和Ada）的绑定。像read这样的简单函数是没有问题的，只要程序设计语言中提供了传递整数和缓冲区地址以及返回整数的方法即可，对此，大多数程序设计语言都可以办到（但对Fortran来说是一

[⊖] 本书所说的标准C是指：1989年制定的最初标准、1995年进行了更新的版本或者称作C99的最新版本。当特指C99时，会专门指出。

种挑战)。处理结构(如stat使用的)和链表(如getaddrinfo使用的)会更困难一些。尽管如此,语言专家通常还是能够找到办法。^①

许多语言,如Java、Perl和Python,都具有可以支持某些POSIX功能的标准功能。例如,这里有一个Python程序,它使用了UNIX系统调用fork来创建子进程。(本书将在第5章介绍fork;现在只需要知道:它可以创建子进程,然后用不同的值返回子进程和父进程,因此if和else为真时,分别代表在父进程和子进程中。)

```
import os
pid = os.fork()
if pid == 0:
    print 'Parent says, "HELLO!'"
else:
    print 'Child says, "hello!'"
```

输出如下:

```
Parent says, "HELLO!"
Child says, "hello!"
```

在本书中,除了附录C,不再举更多的使用C和C++以外语言的例子。在附录C中将介绍一些使用Java和Jython(Python的一种运行于Java环境的版本)的例子。

1.3.3 库函数调用指南

以下是一些调用库函数(C或C++)的一般性指南,可应用于系统调用或其他任意函数:

- 包含必需的头文件(如上文所述)。
- 一定要知道函数指示错误和检查错误的方法,除非有足够的理由(见1.4节),否则请对错误进行检查。如果不想检查,则请将返回值设为void,如下所示:

```
(void)close(fd);
```

printf一贯违反该指南,但是下面的例子程序中还有少量的其他例外。

- 除非绝对必需,否则请不要使用cast,因为它们可能会隐含错误。应当避免以下做法:

```
int n;
struct xyz *p;
...
free((void *)p); /* gratuitous cast */
```

问题在于,如果你将p误写为n,那么cast将会掩盖一条编译器警告。当函数原型指定为void *时,不需要使用cast。如果不能肯定要使用的类型是否是正确的,那么请不要使用cast,这样编译器就可以告诉你了。这可应用于这样的情况:诸如某函数要调用int时,你想提供给它的却是pid_t(进程ID的类型)。当存在编译警告时,得到警告通知总比没有警告要好得多。如果确实获得了警告,并且已经确定cast是唯一的解决办法,那时就可以使用cast了。

- 如果获得了超过一个的线程,请弄清楚要调用的函数是否是线程安全的。也就是说,是否能正确处理多线程程序的全局变量、互斥体(信号量)以及信号等。有许多函数是不行的。
- 尽量写入标准接口而不是特定的系统,这会带来许多好处:代码会具有更强的可移植性,标准化的功能性可能通过了更全面的测试,其他程序员理解起来可能会更加容易,代码可能会和下一版本的操作系统更兼容。实际上,所能做的只是维持一个对Open Group Web站点(见1.9节并参阅[SUS2002])开放的浏览器窗口,该站点有一个非常好的按字

^① 例如,附录C描述的Jtux是Java Native Interface(JNI)的扩展应用。

母排序的所有标准函数（包括标准C库）和所有头文件的列表。这种方法比在本地UNIX系统中使用联机资料更好。当然，有时确实需要为你的系统寻找特例，偶尔还需要使用某些不标准的东西。但是请把这些看作某种特例而不要将其视为规则，并用注释来标明——它是不标准的（这里所说的“标准”将在1.5节中详细阐述）。

1.3.4 函数对照表

本节将对每个系统调用或函数进行正式介绍。每个介绍都包括一个简单的、包含必需的头文件、参数和错误报告方法的对照表。以下是关于一个标准C函数`atexit`的例子。该函数将在1.4.2节中使用：

```
atexit——当进程退出时注册已调用的函数

#include <stdlib.h>

int atexit(
    void (*fcn)(void)    /* function to be called */
);
/* Returns 0 on success, non-zero on error (errno not defined) */
```

（如果对`errno`还不熟悉，可以在下一节中找到相关讲解。）

`atexit`的工作方式是，首先进行函数声明：

```
static void fcn(void)
{
    /* work to be done at exit goes here */
}
```

然后进行注册：

```
if (atexit(fcn) != 0) {
    /* handle error */
}
```

这时，当进程退出时，会自动调用该函数。可以注册1个以上的函数（最多32个），它们将按照和注册相反的顺序依次被调用。请注意，正如对照表中所述，在`if`语句中测试的是非零值，它不测试1、比零大的值、-1、`false`、`NULL`或其他什么值。这是唯一能安全完成任务的方式。此外，这使得日后进行bug查找和试图比较文档与代码的工作更为容易。比较二者时，不必费丝毫脑力。

1.4 错误处理

对从系统调用中返回的错误进行测试需要一定技巧，发现错误后处理错误更具有技巧性。本节将解释该问题并给出一些实用的解决方案。

1.4.1 错误检测

大多数系统调用都会返回值。在`read`的例子中（见1.3.1节），返回的是已读取的字节数。为了表示错误，系统调用通常会返回一个不会和有效数据混淆的数值，以-1最为常见。因此，该例的编码应当进行如下修改：

```
if ((amt = read(fd, buf, numbyte)) == -1) {
    fprintf(stderr, "Read failed!\n");
    exit(EXIT_FAILURE);
}
```

请注意`exit`也是一个系统调用，但它不返回错误，因为它不执行返回操作。符号`EXIT_FAILURE`是属于标准C的。

在本书所涵盖的系统调用中，大约有60%在发生错误时返回-1，20%返回其他值，如`NUL`、0或类似`SIG_ERR`的特定符号；还有20%根本不报告错误。因此，不应假定它们都是按照相同方式工作的，必须阅读每一个系统调用的文档。在本书介绍每个系统调用时，会提供此类信息。

返回错误指示系统调用失败的原因有很多种。其中80%的情况下，整数符号`errno`包含的代码可以指明原因。要得到`errno`，应包含头文件`errno.h`。虽然`errno`不一定非是个整数变量，但可像整数那样使用它。如果使用线程，那么`errno`用起来会像函数调用，因为不同的线程不可能全部都能可靠地使用同一个全局变量。因此，不要自己声明`errno`（这可能是你正打算要做的），而应使用头文件中的定义，如下所示（未显示其他头文件）：

```
#include <errno.h>

if ((amt = read(fd, buf, numbyte)) == -1) {
    fprintf(stderr, "Read failed! errno = %d\n", errno);
    exit(EXIT_FAILURE);
}
```

如果文件描述符错误，其输出应为：

```
Read failed! errno = 9
```

通常情况下，仅在首先执行了错误检测的前提下，才可使用`errno`的值，不能仅检测`errno`查看是否发生了错误。因为，只有函数被专门用来返回错误的时候才会设定`errno`的值。因此，以下代码是错误的：

```
amt = read(fd, buf, numbyte);
if (errno != 0) { /* wrong! */
    fprintf(stderr, "Read failed! errno = %d\n", errno);
    exit(EXIT_FAILURE);
}
```

可以在使用`errno`之前，将它的值设为0：

```
errno = 0;
amt = read(fd, buf, numbyte);
if (errno != 0) { /* bad! */
    fprintf(stderr, "Read failed! errno = %d\n", errno);
    exit(EXIT_FAILURE);
}
```

但这依然不是一个好方法，因为：

- 如果以后要修改代码，想在调用`read`之前加入其他系统调用，或加入其他最后会执行系统调用的函数时，`errno`的值可能会被那个调用所设定。
- 不是所有的系统调用都会设置`errno`的值，应该养成严格遵循函数规范来检查错误的习惯。因此，对几乎所有系统调用来说，都只能在已经确认错误发生之后，再检查`errno`。

上面已经对单独使用`errno`检查错误进行了警告。对于UNIX来说，必须说还存在少量确实要依靠改变的`errno`值来表示错误的特例（如`sysconf`和`readdir`），但就连它们也会返回一个特定的值，告诉人们去检查`errno`。因此，一个不错的规则是在检查返回值之前不要检查`errno`，这条规则对于大多数例外情况也适用。

`errno`取值为9时，并不是标准化的，只会显示一些没有多少意义的段落。因此最好使用标准C函数`perror`，如下所示：

```

if ((amt = read(fd, buf, numbyte)) == -1) {
    perror("Read failed!");
    exit(EXIT_FAILURE);
}

```

现在的输出是:

```
Read failed!: Bad file number
```

另一个有用的标准C函数是strerror，它不像perror那样显示信息，而仅提供一个字符串信息。

但是，虽然消息“Bad file number”足够清晰，但同样不是标准化的，因此仍然存在问题：在系统调用及其他使用errno的函数的官方文档中，当谈及各种错误时，使用的是EBADF之类的符号引用，而不是文本消息。例如，以下是摘自SUS的有关read的条目：

[EAGAIN]

为文件描述符设置O_NONBLOCK标志，同时进程将被延迟。

[EBADF]

files参数不是为读操作打开的、有效的文件描述符。

[EBADMSG]

文件是一个流（STREAM）文件，它被设置成一般控制模式，并且等待读取的消息包含一个控制部分。

尽管文本中没有显示出那些确切的词语，但“Bad file number”与EBADF是直接匹配的，然而对于那些更为模糊的错误来说就不是这样了。我们真正想要和文本信息一同得到的是实际的符号，但并不存在提供这种信息的标准C或SUS函数。因此，可以自己编写能将号码翻译成符号的函数。因为许多符号是和具体系统相关的，因此本书在根据Linux、Solaris和BSD的errno.h文件得来的代码中建立了符号列表。对于读者所用的系统而言可能必须调整这个代码。篇幅所限，这里没有列出所有的符号，但可以从AUP网站获得全部代码（见1.8节并参阅[AUP2003]）。

```

static struct {
    int code;
    char *str;
} errcodes[] =
{
    { EPERM, "EPERM" },
    { ENOENT, "ENOENT" },
    ...
    { EINPROGRESS, "EINPROGRESS" },
    { ESTALE, "ESTALE" },
#ifdef BSD
    { ECHRNG, "ECHRNG" },
    { EL2NSYNC, "EL2NSYNC" },
    ...
    { ESTRPIPE, "ESTRPIPE" },
    { EDQUOT, "EDQUOT" },
#endif
#ifdef SOLARIS
    { EDOTDOT, "EDOTDOT" },
    { EUCLEAN, "EUCLEAN" },
    ...
    { ENOMEDIUM, "ENOMEDIUM" },
    { EMEDIUMTYPE, "EMEDIUMTYPE" },
#endif
#ifdef
    { 0, NULL }
};

```



```

const char *errsymbol(int errno_arg)
{
    int i;

    for (i = 0; errcodes[i].str != NULL; i++)
        if (errcodes[i].code == errno_arg)
            return errcodes[i].str;
    return "[UnknownSymbol]";
}

```

下面是用新函数对read进行错误检查的代码：

```

if ((amt = read(fd, buf, numbyte)) == -1) {
    fprintf(stderr, "Read failed!: %s (errno = %d; %s)\n",
            strerror(errno), errno, errsymbol(errno));
    exit(EXIT_FAILURE);
}

```

现在输出是完整的：

```
Read failed!: Bad file descriptor (errno = 9; EBADF)
```

编写一个能够格式化错误信息的更具实用性的函数是很方便的，因此可以将它用于下节将要编写的代码中：

```

char *syserrmsg(char *buf, size_t buf_max, const char *msg, int errno_arg)
{
    char *errmsg;

    if (msg == NULL)
        msg = "???";
    if (errno_arg == 0)
        snprintf(buf, buf_max, "%s", msg);
    else {
        errmsg = strerror(errno_arg);
        snprintf(buf, buf_max, "%s\n\t\t*** %s (%d: \"%s\") ***", msg,
                errsymbol(errno_arg), errno_arg,
                errmsg != NULL ? errmsg : "no message string");
    }
    return buf;
}

```

我们将这样使用syserrmsg：

```

if ((amt = read(fd, buf, numbyte)) == -1) {
    fprintf(stderr, "%s\n", syserrmsg(buf, sizeof(buf),
        "Call to read function failed", errno));
    exit(EXIT_FAILURE);
}

```

其输出如下：

```
Call to read function failed
*** EBADF (9: "Bad file descriptor") ***
```

那么，对于其他20%仅报告错误但不设置errno的调用，该怎样处理呢？在它们当中，大约有20%会采用其他方法报告错误，通常是简单返回错误代码（即，非零返回值表明发生了错误，同时也表明代码的含义）；其余的将完全不提供具体的原因。在本书中，将提供每个函数（共计300个左右）的详细内容。下面是一个直接返回错误代码的例子（该代码的功能目前并不重要）：

```
struct addrinfo *infop;
```

```

if ((r = getaddrinfo("localhost", "80", NULL, &info)) != 0) {
    fprintf(stderr, "Got error code %d from getaddrinfo\n", r);
    exit(EXIT_FAILURE);
}

```

`getaddrinfo`是不设置`errno`的函数之一，不能将它返回的错误代码传递给`strerror`，因为`strerror`函数只对`errno`值有效。在[SUS2002]或系统手册中定义了那些不设置`errno`的函数所返回的不同错误代码，因此当然可以为那些函数编写某个版本的`errsymbol`（见上文）。但困难的是，不能指定某个函数的符号拥有与其他函数符号相区别的值。这就意味着不能编写一个仅处理并查询某个错误代码的函数，像`errsymbol`一样，同样必须传入函数的名称。（这样做可以从`gai_strerror`中获得好处，它是专为`getaddrinfo`定作的`strerror`版本）。

本书中大约有20多个函数，在标准[SUS2002]中没有定义任何`errno`的值，或者即使设置了`errno`的值，但在实现的时候也许还需要设定`errno`。在这些函数的对照表中将显示短语“`errno not defined`”。

开始感觉头疼了吗？UNIX的错误处理真是一团乱麻。很不幸，要构造能使系统调用出错的测试实例来检测错误控制代码是很困难的，这种矛盾使得每次使其恢复正常都很困难。但是目前无法对其进行改善（受标准所限），所以必须适应这种状况。不过要小心！

1.4.2 C错误检测宏

将每个系统调用加入到`if`语句，并在其后编写显示错误信息以及退出或返回的代码是很单调乏味的。当需要做清理工作时，事情将会变得更糟，如下例所示：

```

if ((p = malloc(sizeof(buf))) == NULL) {
    fprintf(stderr, "%s\n", syserrmsg(buf, sizeof(buf),
        "malloc failed", errno));
    return false;
}
if ((fdin = open(filein, O_RDONLY)) == -1) {
    fprintf(stderr, "%s\n", syserrmsg(buf, sizeof(buf),
        "open (input) failed", errno));
    free(p);
    return false;
}
if ((fdout = open(fileout, O_WRONLY)) == -1) {
    fprintf(stderr, "%s\n", syserrmsg(buf, sizeof(buf),
        "open (output) failed", errno));
    (void)close(fdin);
    free(p);
    return false;
}

```

继续下去，清理代码将会越来越长，难以编写、阅读和维护。有些程序员会使用一个`goto`语句，使得清理代码只需编写一次。通常应当避免使用`goto`语句，但这里似乎值得一用。注意必须很仔细地初始化含有清理代码和对文件描述符的值进行测试的变量，以保证无论处于何种不完整的状态，清理代码都能正确执行。

```

char *p = NULL;
int fdin = -1, fdout = -1;

if ((p = malloc(sizeof(buf))) == NULL) {
    fprintf(stderr, "%s\n", syserrmsg(buf, sizeof(buf),
        "malloc failed", errno));
}

```

```
        goto cleanup;
    }
    if ((fdin = open(filein, O_RDONLY)) == -1) {
        fprintf(stderr, "%s\n", syserrmsg(buf, sizeof(buf),
            "open (input) failed", errno));
        goto cleanup;
    }
    if ((fdout = open(fileout, O_WRONLY)) == -1) {
        fprintf(stderr, "%s\n", syserrmsg(buf, sizeof(buf),
            "open (output) failed", errno));
        goto cleanup;
    }
    return true;

cleanup:
    free(p);
    if (fdin != -1)
        (void)close(fdin);
    if (fdout != -1)
        (void)close(fdout);
    return false;
}
```

编写所有那些if、fprintf和goto代码，仍然很痛苦。系统调用本身的代码几乎被淹没了！

可以使用某些宏来简化检测错误、显示错误信息和从函数获取信息等任务。下文将首先说明其使用方法，然后说明其实现方法（该内容只使用了标准C编码，与系统调用及UNIX没什么特别的联系，但由于它将被用于本书随后的所有例子，所以在此包含了这个内容）。

下面使用这些错误检测（“ec”）宏，重新编写上一个例子。在此没有显示其上下文，但该代码包含在名为fcb的函数中：

```
char *p = NULL;
int fdin = -1, fdout = -1;

ec_null( p = malloc(sizeof(buf)) )
ec_negl( fdin = open(filein, O_RDONLY) )
ec_negl( fdout = open(fileout, O_WRONLY) )

return true;

EC_CLEANUP_BGN
free(p);
if (fdin != -1)
    (void)close(fdin);
if (fdout != -1)
    (void)close(fdout);
return false;
EC_CLEANUP_END
```

以下是对该函数的调用。由于它处于main函数中，因此在出现错误时退出是有意义的。

```
ec_false( fcb() )

/* other stuff here */

exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
exit(EXIT_FAILURE);
EC_CLEANUP_END
```



上述调用的执行过程是：宏`ec_null`、`ec_neg1`和`ec_false`分别检测其参数表达式是否为`NULL`、`-1`和`false`，存储错误信息，并转向宏`EC_CLEANUP_BGN`位置的标签。然后，运行和上文相同的清理代码。在`main`中，对`fcfn`返回值的测试同样会跳转到`main`中的相同标签，接着程序退出。用`atexit`（见1.3.4中的介绍）装载的函数可以显示所有积累的错误信息：

```
ERROR: 0: main [/aup/c1/errorhandling.c:41] fcn()
       1: fcn [/aup/c1/errorhandling.c:15] fdin = open(filein, 0x0000)
          *** ENOENT (2: "No such file or directory") ***
```

在最后一行中，可以看到`errno`符号、值和描述文本。在它之前是错误返回的反向跟踪。每行跟踪显示了层次、函数名、文件名、行号和返回错误所指示的代码。这类信息不是给最终用户看的，但在开发阶段它非常有用。以后，可以改变这些宏命令（很快就可以看到是如何改变的）以将上述内容存储到某个日志文件中，使得用户能看到一些对他们更有意义的东西。

在运行时，因为要给应用程序开发者最大的自由度，用其认为合适的方式来处理错误，所以是将错误信息累积起来而不是显示出来。对库中的函数来说，只是将错误信息写到`stderr`中实在是没有什么用。错误信息可能没有出现在合适的位置，对应用程序的最终用户来说，其措辞也可能不太恰当。当然，我们最终会把它们显示出来，如果在实际应用中使用这些宏，那项决定将能够很容易地被改变。

综上所述，这些宏可以提供以下功能：

- 简单易读的错误检测
- 自动跳转以清理代码
- 提供完整的错误信息和回溯追踪

宏的缺点是句法有些奇怪（句末没有分号），而且控制流中存在着某些程序员认为是非常不好的隐式跳转。如果你认为利大于弊，就可以使用宏（就像本书这样）。否则，你就应该自己去设计（可能会使用显式的`goto`语句而不是隐式的`goto`语句），或者完全忽略它们。

以下是实现错误检测宏的头文件（`ec.h`）的大部分内容（忽略了一些函数声明和少量其他的次要细节）：

```
extern const bool ec_in_cleanup;

typedef enum {EC_ERRNO, EC_EAI} EC_ERRTYPE;

#define EC_CLEANUP_BGN\
    ec_warn();\
    ec_cleanup_bgn:\
    {\
        bool ec_in_cleanup;\
        ec_in_cleanup = true;
#define EC_CLEANUP_END\
    }

#define ec_cmp(var, errrtn)\
    {\
        assert(!ec_in_cleanup);\
        if ((intptr_t)(var) == (intptr_t)(errrtn)) {\
            ec_push(__func__, __FILE__, __LINE__, #var, errno, EC_ERRNO);\
            goto ec_cleanup_bgn;\
        }\
    }

#define ec_rv(var)\
    {\
```



```

int errrtn;\
assert(!ec_in_cleanup);\
if ((errrtn = (var)) != 0) {\
    ec_push(__func__, __FILE__, __LINE__, #var, errrtn, EC_ERRNO);\
    goto ec_cleanup_bgn;\
}\
}\

#define ec_ai(var)\
(\
    int errrtn;\
    assert(!ec_in_cleanup);\
    if ((errrtn = (var)) != 0) {\
        ec_push(__func__, __FILE__, __LINE__, #var, errrtn, EC_EAI);\
        goto ec_cleanup_bgn;\
    }\
)\

#define ec_neg1(x) ec_cmp(x, -1)
#define ec_null(x) ec_cmp(x, NULL)
#define ec_false(x) ec_cmp(x, false)
#define ec_eof(x) ec_cmp(x, EOF)
#define ec_nzero(x)\
(\
    if ((x) != 0)\
        EC_FAIL\
)\

#define EC_FAIL ec_cmp(0, 0)

#define EC_CLEANUP goto ec_cleanup_bgn;
#define EC_FLUSH(str)\
(\
    ec_print();\
    ec_reinit();\
)\

```

在解释宏之前，我们必须首先讨论一个问题并提出其解决方案。该问题是如果你在清理代码中调用了错误检测宏（例如`ec_neg1`）并且出现了错误，那么很可能会形成一个无限循环，因为宏还会跳转到清理代码中！下面正是这样一个例子：

```

EC_CLEANUP_BGN
free(p);
if (fdin != -1)
    ec_neg1( close( fdin ) )
if (fdout != -1)
    ec_neg1( close( fdout ) )
return false;
EC_CLEANUP_END

```

看上去程序员是在非常小心地检测`close`返回的错误，但其做法会带来非常严重的后果。其真正糟糕的是只有在某错误后存在错误清理时才会发生循环，而这是一种在测试时很难捕获的非常不常见的情况。应谨防这种情况——错误检测宏应当增加可靠性，而不是降低可靠性。

解决方案是在清理代码中加入一个局部变量`ec_in_cleanup`，并将其值设为`true`，就像在宏`EC_CLEANUP_BGN`的定义中所看到的那样。针对它的测试存在于宏`ec_cmp`（如设置了该值，`assert`就会启动，从而就知道出错了）中。

（`bool`类型及其值`true`和`false`，是在C99中新引入的。如果缺少它们，就只能将以下

代码

```
typedef int bool;
#define true 1
#define false 0
```

粘贴到某个头文件中。)

为防止在清理代码之外调用`ec_cmp`时阻止断言启动(例如某正常调用),引入了全局变量,将其同样命名为`ec_in_cleanup`,并永久性地设为`false`。仅在很少的特例中可以在(本质上,实际上)使用局部变量来隐藏全局变量。

为什么要用局部变量?为什么在清理代码的开头不将全局变量设为`true`,并在代码末尾改回`false`呢?答案是,如果从恰好合法使用`ec_cmp`宏的清理代码中调用某函数,上述方法将不起作用。它将会发现全局变量被设成了`true`,从而认为它在自身的清理代码中,但实际并不是这样。因此,每个函数(就是说,每个单独的清理代码部分)都需要一个私有的保护变量。

下面将逐一介绍宏:

- `EC_CLEANUP_BGN`包括清理代码的标签(`ec_cleanup_bgn`),在其前面的函数调用仅输出一条控制流入标签的警告。这可以避免忘记在标签之前放置一条`return`语句而导致在没有错误时跳入清理代码之类的常见错误。(这是我为了查找不存在的错误而浪费了一个小时得到的经验教训。)接着是上文讲述过的`ec_in_cleanup`。
- `EC_CLEANUP_END`只是起到一个封闭大括号的作用。需要这些大括号来创建局部上下文。
- `ec_cmp`做了大部分的工作:确保没有处于清理代码中,检测错误,调用`ec_push`(下文很快就要讲到)把位置信息(`_FILE`等)压入堆栈,并跳到清理代码。类型`intptr_t`是C99新加入的:它是一个能够容纳指针的整数类型。如果缺少该类型,把`typedef`定为`long`也可以。如果要更强的安全性,可以在程序的某处粘贴一些用于测试`sizeof(void *)`等于`sizeof(long)`的代码。(如果不熟悉符号`#var`,请研读C语言——它把无论什么`var`都扩大到一个字符串。)
- `ec_rv`和`ec_cmp`相似,但是它用于返回一个非零错误代码以指示错误,而不使用`errno`本身的函数。然而,返回的代码是`errno`值,因此可以直接传递给`ec_push`。
- `ec_ai`和`ec_rv`相似,但是它所处理的错误代码不是`errno`值。为表明这种情况,传递给`ec_push`的最后一个参数是`EC_EAI`(仅有第8章出现的两个函数使用这种方法)。
- 宏`ec_negl`、`ec_null`、`ec_false`和`ec_eof`使用恰当的函数调用`ec_cmp`,而`ec_nzero`自己进行检测。它们适用于大多数情况,对其他情况可以仅直接使用`ec_cmp`。
- 当由于没有使用上段介绍的宏,而出现测试错误时,使用`EC_FAIL`。
- `EC_CLEANUP`仅用于需要跳入清理代码的情况。
- `EC_FLUSH`仅用于需要显示错误信息而不需要等待退出的情况。这对于需要保持持续运行的交互式程序来说是很方便的。

这里没有列出宏调用的各种服务函数,因为它们对UNIX系统调用说明得不多(它们仅使用标准C),你可以到AUP的网站[AUP2003]对其进行浏览,网站中还有关于它们工作方法的解释。概括如下:

- `ec_push`将传送给它(例如由`ec_cmp`宏传来的)的错误及其上下文信息压入堆栈中。
- 用`atexit`注册的函数,在程序退出时显示堆栈中的信息:

```
static void ec_atexit_fcn(void)
{
    ec_print();
}
```

- `ec_print`遍历堆栈以显示跟踪和错误信息。
- `ec_reinit`清空堆栈内容，使得错误检测可以开始全新的跟踪。
- 如果碰巧陷入其中，则从`EC_CLEANUP_BGN`代码中调用`ec_warn`。

所有这些函数都是线程安全的，因此可以在多线程程序中使用。本书在5.17节中对此进行了更详细的介绍。

1.4.3 使用C++异常

在花费很多的时间和精力开始决定是否喜欢前节的“`ec`”宏和提出改进之前，最好问心自问是否使用C编程。现在你更有可能想使用C++。毕竟，几乎本书中的所有内容都能很好地适用于C++程序。对于嵌入式系统、操作系统（如Linux）、编译器和其他相对低级的软件而言，C仍然是很好的编程语言，但是这些系统更趋向于采用其自带的、高度专用的错误处理机制。

C++提供了用异常处理错误的机会，异常是C++语言内建的，而不是像C语言所使用的`goto`和`return`语句的组合物。异常自身也有缺陷，但如果小心使用，它们会比“`ec`”宏更容易使用，且更为可靠。例如，“`ec`”宏不会在想使用`ec_neg1`却误用了`ec_null`的时候提供保护。

由于包含系统调用包代码的库通常都只提供C版本的，所以除非特别制作了C++版本的，否则它将不会抛出异常。因此，要使用异常，就需要进行另一层次的封装，正如以下`close`系统调用所显示的那样：

```
class syscall_ex {
public:
    int se_errno;

    syscall_ex(int n)
        : se_errno(n)
    { }
    void print(void)
    {
        fprintf(stderr, "ERROR: %s\n", strerror(se_errno));
    }
};

class syscall {
public:
    static int close(int fd)
    {
        int r;
        if ((r = ::close(fd)) == -1)
            throw(syscall_ex(errno));
        return r;
    }
};
```

然后，只需要调用`syscall::close`而不是直接调用`close`，它会在发生错误时抛出一个异常。可能并不需要键入其他大约1100个UNIX函数的代码，而仅仅只需要键入应用程序中所使用的那些。

如果想要在异常信息中包含诸如文件和行号等本地信息，就需要定义另一个包（这次是

一个宏)用来捕获预处理的数据(例如via_LINE_),[⊖]因此有如下两个更为奇特的类:

```
class syscall_ex {
public:
    int se_errno;
    const char *se_file;
    int se_line;
    const char *se_func;

    syscall_ex(int n, const char *file, int line, const char *func)
        : se_errno(n), se_file(file), se_line(line), se_func(func)
    { }
    void print(void)
    {
        fprintf(stderr, "ERROR: %s [%s:%d %s()]\n",
            strerror(se_errno), se_file, se_line, se_func);
    }
};

class syscall {
public:
    static int close(int fd, const char *file, int line, const char *func)
    {
        int r;
        if ((r = ::close(fd)) == -1)
            throw(syscall_ex(errno, file, line, func));
        return r;
    }
};

#define Close(fd) (syscall::close(fd, __FILE__, __LINE__, __func__))
```

这次调用的是Close而不是close。

如果愿意,还可以通过逐个调用追踪使其膨大,就像对“ec”宏所做的那样,也许比它还要大。

在本书的附录B中,描述了一个叫作Ux的C++包的例子,其中封装了所有的系统调用。

1.5 UNIX标准

事实上,在现实世界中,大多数商业UNIX销售商都使用Open Group的品牌(见1.2节),而开源软件发行人声称只遵从POSIX1990(虽然存在少量例外),但实际上他们中的大部分并没有进行过认证检测(现在该检测已经可以自由开展了,因此从现在看,未来的认证很可能会实现)。

1.5.1 API标准的演化

要列举所有不同的POSIX和Open Group的标准、指南和规范,会非常复杂令人困惑。在大多数情况下,所有相关的发展都可以被简单地认为是与UNIX API相关的标准的发展,其中仅有表1-2中列出的8个标准,才是本书要重点讲述的。

实际上,还有一个POSIX2001(IEEE标准POSIX.1-2001),但由于它已包含在SUS3中,所以不需要单独列出。

[⊖] 我们需要该宏是因为如果仅把_LINE_和其他的预处理数据作为syscall_ex构造体的直接参数,就会在class syscall定义中得到一个错误的位置。

表1-2 POSIX和Open Group API标准

名称	标准	解释
POSIX 1988	IEEE标准1003.1-1988 (198808L)	第一个标准
POSIX1990	IEEE标准1003.1-1990/ISO9945-1; 1990 (199009L)	POSIX1988的较少更新
POSIX1993	IEEE标准1003.1b-1993 (199309L)	POSIX1990+实时
POSIX1996	IEEE标准1003.1-1996/ISO 9945-1: 1996(199506L)	POSIX1993+线程+修正的实时
XPG3	X/Open Portability Guide	第一个分布广泛的X/Open准则
SUS1	单一UNIX规范, 版本1	POSIX1990+BSD、AT&T系统V和OSF所使用的一般API; 还有著名的1170规范; 标有UNIX950*的已认证系统
SUS2	单一UNIX规范, 版本2	更新POSIX1996的SUS1+64位、大文件、增强的多字节和Y2K; 标有UNIX98商标
SUS3	单一UNIX规范, 版本3 (2002112L)	更新SUS2; API部分符合IEEE标准1003.1-2001 (POSIX与Open Group完全融合); 标有UNIX 03商标

* 这个巨大的数是IEEE通过的年和月, 它们将被用在下一节解释的特性检测中。

† 1170规范的命名来自API、头文件和命令的总数。

即便是从这张过于简化的列表中也还是可以看出来, 不严谨声明OS“遵从POSIX”是毫无意义的, 如果发表了这种不严谨的声明, 那么表明发布声明的人要么没有真正了解POSIX (或SUS), 要么假定听众不懂。POSIX指的是什么? 同时, 由于一些新特性, 如实时和线程等是可选的, 选项是什么? 一般来说, 以下是对声明内容的一种解释:

- 如果所听到的都是说POSIX, 它可能意味着POSIX1990。这是大多数人掌握起来很困难的第一个也是最后一个标准。
- 除非知道某操作系统已经通过了标准化组织制定的认证测试, 否则所能确定的只是OS开发者仅在某种程度上考虑了POSIX标准。
- 必须单独研究可选特性的状态 (稍后会更详细地讲述这些)。

1.5.2 告诉系统你想要什么

本节和下节将描述根据标准应当做什么, 以便保证应用程序适应环境并检测OS遵从的标准版本。然后可以决定需要做多少工作, 可以忽略多少工作。

首先, 通过在包含任何POSIX头文件 (如unistd.h) 之前定义一个预处理符号, 应用程序可以表明它希望的标准版本是什么。这限定了标准头文件 (如stdio.h, unistd.h, errno.h), 使得我们只能使用那些已经在标准中定义了的符号。为了把自己限制在标准的POSIX1990中, 可以这样做:

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

对于较新的POSIX标准, 必须更具体一些, 可以像下面这样另外增加一个符号:

```
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 199506L
#include <sys/types.h>
#include <unistd.h>
```

注意，与 `_POSIX_SOURCE` 不同，`_POSIX_C_SOURCE` 有一个通常由标准通过时（例如 1995年6月）的年和月构成的长整数类型的值。另一个可能的长整数值是 200112L。但对于 `POSIX1990`，应将它设为 1，而不是 199009L。[⊖]

如果使用商业 UNIX 系统，那么很可能使你感兴趣的不是 POSIX，而是 SUS，因为它有一个专门针对商业 UNIX 系统的符号 `_XOPEN_SOURCE`，这个符号有两个特殊的值：500 针对 SUS2，600 针对 SUS3。因此，如果要使用 SUS2 的内容——号称 UNIX98 的系统，应进行如下工作：

```
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 199506L
#define _XOPEN_SOURCE 500
#include <sys/types.h>
#include <unistd.h>
```

对于 SUS1，定义 `_XOPEN_SOURCE` 时不用定义它的值，同时将 `_XOPEN_SOURCE_EXTENDED` 的值定义为 1：

```
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 2
#define _XOPEN_SOURCE
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>
```

从技术角度说，如果真有 SUS2 系统，就不需要定义头两个 POSIX 符号了，因为它们是自动定义的。但是，如果编写的源代码有可能会用在更老的系统上，则需要加入上述定义，用 POSIX 可以明确理解的方式告诉它。`_POSIX_C_SOURCE` 的值是由 `_XOPEN_SOURCE` 决定的，当后者的值没有定义时，前者的值应该为 2；当后者的值为 500 时，前者的值应为 199506L；为 600 时，应为 200112L。

感到迷惑了吗？简直太糟糕了，因为它越来越糟了。为了能有所帮助，以下给出待办工作的总结：

- 决定所写程序期望达到的标准化级别。
- 如果仅是 POSIX1990，那么就只定义 `_POSIX_SOURCE`。
- 如果是 POSIX1993 或是 POSIX1996，那么就同时定义 `_POSIX_SOURCE` 和 `_POSIX_C_SOURCE`，并使用 1.5.1 节表 1-2 中的适当数值。
- 如果是 SUS1、SUS2 或 SUS3，那么就定义 `_XOPEN_SOURCE`（无值、500 或 600），并使用那两个带有合适数值的 POSIX 符号。
- 对于 SUS1，同时须将 `_XOPEN_SOURCE_EXTENDED` 的值定义为 1。
- 忘了 XPG3（还有本书没提及的 XPG4）吧——你的麻烦已经够多了。

在包含某个头文件之前，很容易编写一个这样的头文件，该头文件设置了多个基于单独定义的请求符号。以下是头文件 `suvreq.h`[⊗]；本书将在下一节中介绍它的用法。

```
Header to request specific standard support. Before including it, one
of the following symbols must be defined (1003.1-1988 isn't supported):
```

```
SUV_POSIX1990    for 1003.1-1990
SUV_POSIX1993    for 1003.1b-1993 - real-time
SUV_POSIX1996    for 1003.1-1996
```

⊖ 如果也需要来自 1003.2-1992 的 C 绑定，可以将它设置为 2。

⊗ 其发音是“S-U-V wreck”。SUV 代表标准 UNIX 版本。

```

SUV_SUS1      for Single UNIX Specification, v. 1 (UNIX 95)
SUV_SUS2      for Single UNIX Specification, v. 2 (UNIX 98)
SUV_SUS3      for Single UNIX Specification, v. 3 (UNIX 03)
*/
#ifdef(SUV_POSIX1990)
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 1

#elif defined(SUV_POSIX1993)
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 199309L

#elif defined(SUV_POSIX1996)
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 199506L

#elif defined(SUV_SUS1)
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 2
#define _XOPEN_SOURCE
#define _XOPEN_SOURCE_EXTENDED 1

#elif defined(SUV_SUS2)
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 199506L
#define _XOPEN_SOURCE 500
#define _XOPEN_SOURCE_EXTENDED 1

#elif defined(SUV_SUS3)
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 200112L
#define _XOPEN_SOURCE 600
#define _XOPEN_SOURCE_EXTENDED 1
#endif

```

如果试图写一个高可移植性程序，最好使用标准的東西，这样有助于确保不会偶尔使用到非标准的東西。但是，对其他应用，这样限制就太死了。例如，虽然FreeBSD声称仅遵从POSIX1990，但是它的确也实现了SUS1级的System V消息。如果需要消息或者任何其他FreeBSD提供的POSIX1990之后的内容，就不需要定义`_POSIX_SOURCE`。可以不定义它，因为它是可选的。我必须做的是得到本书中运行在FreeBSD[®]上的示例代码。

我们解释的这些符号和可笑的数字可能会使你头疼，下面要跟你你说的是，你可能不需要使用它们。欢迎来到标准的世界！

1.5.3 系统包含的内容

仅仅对使用`_POSIX_SOURCE`、`_POSIX_C_SOURCE`和`_XOPEN_SOURCE`符号问了一些问题，这并不意味着那是你想得到的。如果系统仅是POSIX1993，那就是系统所能提供的全部信息。如果那刚好是所需要的就好了。如果不是，就需要拒绝编译（使用`#error`命令），使应用的某个选项特性失效，或使用其他的实现方法。

在包含`unistd.h`之后，发现OS必须提供内容的方式是检测`_POSIX_VERSION`符号，如果在SUS系统上，那么还要检测`_XOPEN_UNIX`和`XOPEN_VERSION`符号。上节中使用的4个“SOURCE”符号，是告诉系统的，现在系统返回给你。

① 另外，大多数编译器和操作系统都具有能够引入有用的非标准特性的其他符号。例如`gcc`的`_GNU_SOURCE`或者Solaris的`_EXTENSIONS`，可以查看系统文档。

当然，这些符号甚至可能都没有定义，尽管这对于 POSIX_VERSION 来说是奇怪的，因为已经成功地包含了 unistd.h 文件。因此，必须仔细测试，使用预处理命令来测试的方法如下：

```
#if _POSIX_VERSION >= 199009L
/* we have some level of POSIX */
#else
#error "Can be compiled only on a POSIX system!"
#endif
```

如果没有定义 POSIX_VERSION，它将被替换为 0。

_POSIX_VERSION 从上一节的表中取值，如 199009L。如果系统的标准不低于 SUS1，那么 _XOPEN_UNIX 将被自动定义（值为空），但更好的方法是检查 _XOPEN_VERSION，其值将会等于 4 500 或 600，也许将来还会是某个更大的数。

下面编写一个小程序。在请求 SUS2 兼容性之后，它将输出头文件的一些信息（头文件 suvreq.h 在前一节中）。

```
#define SUV_SUS2

#include "suvreq.h"
#include <unistd.h>
#include <stdio.h>

int main(void)
{
    printf("Request:\n");
#ifdef _POSIX_SOURCE
    printf("\t_POSIX_SOURCE defined\n");
    printf("\t_POSIX_C_SOURCE = %ld\n", (long)_POSIX_C_SOURCE);
#else
    printf("\t_POSIX_SOURCE undefined\n");
#endif

#ifdef _XOPEN_SOURCE
    #if _XOPEN_SOURCE + 0 == 0
        printf("\t_XOPEN_SOURCE defined (no value)\n");
    #else
        printf("\t_XOPEN_SOURCE = %d\n", _XOPEN_SOURCE);
    #endif
#else
    printf("\t_XOPEN_SOURCE undefined\n");
#endif

#ifdef _XOPEN_SOURCE_EXTENDED
    printf("\t_XOPEN_SOURCE_EXTENDED defined\n");
#else
    printf("\t_XOPEN_SOURCE_EXTENDED undefined\n");
#endif

    printf("Claims:\n");
#ifdef _POSIX_VERSION
    printf("\t_POSIX_VERSION = %ld\n", _POSIX_VERSION);
#else
    printf("\tNot POSIX\n");
#endif

#ifdef _XOPEN_UNIX
    printf("\tX/Open\n");
    #ifdef _XOPEN_VERSION
        printf("\t_XOPEN_VERSION = %d\n", _XOPEN_VERSION);
    #endif
#endif
}
```



```
#else
    printf("\tError: _XOPEN_UNIX defined, but not "
           "_XOPEN_VERSION\n");
#endif
#else
    printf("\tNot X/Open\n");
#endif
return 0;
}
```

在Solaris 8系统中运行时，其输出如下：

```
Request:
    _POSIX_SOURCE defined
    _POSIX_C_SOURCE = 199506
    _XOPEN_SOURCE = 500
    _XOPEN_SOURCE_EXTENDED defined
Claims:
    _POSIX_VERSION = 199506
    X/Open
    _XOPEN_VERSION = 500
```

因此，Solaris 8可被认为是符合SUS2的。但是如果将代码的第一行改为SUV_SUS1，则其输出如下：

```
Request:
    _POSIX_SOURCE defined
    _POSIX_C_SOURCE = 2
    _XOPEN_SOURCE defined (no value)
    _XOPEN_SOURCE_EXTENDED defined
Claims:
    _POSIX_VERSION = 199506
    X/Open
    _XOPEN_VERSION = 4
```

这意味着Solaris可同样被设置为像SUS1系统那样运行，隐藏所有比SUS1系统更新的功能。

在FreeBSD 4.6系统中运行时，其输出如下：

```
Request:
    _POSIX_SOURCE defined
    _POSIX_C_SOURCE = 2
    _XOPEN_SOURCE defined (no value)
    _XOPEN_SOURCE_EXTENDED defined
Claims:
    _POSIX_VERSION = 199009
    Not X/Open
```

这意味着无论我们请求什么，FreeBSD系统都会自称只是符合POSIX1990的。

如上文所述，对FreeBSD系统而言，你可能不想进行此项请求，因为这会去除其已经包含的太多内容，其中许多内容遵循的可能是新于POSIX1990的某些标准。当根本没有头文件suvreq.h后，重新运行程序的结果如下：

```
Request:
    _POSIX_SOURCE undefined
    _XOPEN_SOURCE undefined
    _XOPEN_SOURCE_EXTENDED undefined
Claims:
    _POSIX_VERSION = 199009
    Not X/Open
```

毫无疑问，仍遵从POSIX1990。有趣的是，在Solaris系统上运行时，运行结果表明它将

会遵从XPG3，因此可称其为默认级别。对于号称UNIX98的Solaris系统来说，为什么默认级别不是SUS2？这还是一个谜。以下是Solaris系统的输出：

```
Request:
  _POSIX_SOURCE undefined
  _XOPEN_SOURCE undefined
  _XOPEN_SOURCE_EXTENDED undefined

Claims:
  _POSIX_VERSION = 199506
  X/Open
  _XOPEN_VERSION = 3
```

对于SuSE Linux 8.0 (Linux 2.4版, glibc 2.95.3版), 当定义SUV_SUS2时得到的结果是:

```
Request:
  _POSIX_SOURCE defined
  _POSIX_C_SOURCE = 199506
  _XOPEN_SOURCE = 500
  _XOPEN_SOURCE_EXTENDED defined

Claims:
  _POSIX_VERSION = 199506
  X/Open
  _XOPEN_VERSION = 500
```

因此, 可认为Linux是SUS2系统。但其默认是SUS1, 因为当根本不包含头文件svreq.h时得到的结果是:

```
Request:
  _POSIX_SOURCE defined
  _POSIX_C_SOURCE = 199506
  _XOPEN_SOURCE undefined
  _XOPEN_SOURCE_EXTENDED undefined

Claims:
  _POSIX_VERSION = 199506
  X/Open
  _XOPEN_VERSION = 4
```

将该输出与不包含头文件时的Solaris系统的输出进行比较, 这一次Linux走在了前面, 并且, 当包含头文件unistd.h时, Linux为我们定义了_POSIX_SOURCE。这可能会有意义, 因为在POSIX需求中, 经常要对其进行定义。

其实Darwin才是最谨慎的, 它仅声称符合POSIX1988 (包含头文件svreq.h):

```
Request:
  _POSIX_SOURCE defined
  _POSIX_C_SOURCE = 199506
  _XOPEN_SOURCE = 500
  _XOPEN_SOURCE_EXTENDED defined

Claims:
  _POSIX_VERSION = 198808
  Not X/Open
```

不管怎样, 除根本就不做任何请求的FreeBSD和Darwin两种系统外, 本书中的所有例子代码都在包含头文件svreq.h之前定义了SUV_SUS2。这又带来了一个问题: 如何知道运行的系统是FreeBSD还是Darwin呢? (答案在1.5.8节)。

1.5.4 检查选项

上文讲到POSIX1996 (IEEE标准1003.1-1996) 中包含实时和线程的内容, 但这并不意味着遵循它的系统必须支持实时程序和线程, 因为那些是可选项。这意味着如果不支持, 这些

系统必须采用某种标准的方式对此进行说明，如下面将要解释的那样。

这些和符号体系一样都非常复杂，而对选项的内容介绍得太少。所以，已经将大量附加符号整个地定义为选项组（例如线程、POSIX消息等），其中的一些还定义为子选项。这里对它们不做一一解释，仅指出如何检查，在一般情况下，还会指出是否支持某个选项。由于选项检查规则非常复杂，其涉及范围从POSIX1990到SUS1直至SUS3，所以这里的解释经过了一定的简化。（对选项更深入的讨论请参阅网页[Dre2003]，文中对相关的函数进行了交叉索引。）

在实际操作中，我遇到了以下令人不愉快的情况：

- 主要商业系统（如Solaris）支持所有的选项。由于支持所有的选项，尽管可以不用检查就能工作，但是它们还是提供了能正确处理选项检查符号的功能。
- 开源系统包括Linux、FreeBSD和Darwin，趋向于不支持许多选项，因此进行选项检查是十分重要的。然而，它们并不能正确地处理选项检查，因此所写的检查代码经常不起作用。

很明显，如果OS开发者不遵循当前标准，想要改善这种状况，单靠标准制定团体是毫无办法的。

不管怎样，对每一个选项，在头文件unistd.h中都定义了一个预处理符号。对该符号进行测试，可以判断是否支持该选项。例如，对异步I/O选项，定义了_POSIX_ASYNCHRONOUS_IO，其中包含了aio_read、aio_write和少量其他系统调用（详细介绍见第3章）。如果所编写的程序中使用了该选项，首先应该对其符号进行测试：

```
#if _POSIX_ASYNCHRONOUS_IO <= 0
    /* substitute code, or message, or compile failure with #error */
#else
    /* code that uses the feature */
#endif
```

这段代码意味着，如果未定义符号或其值定义为0或更小的值，将不能使用那些系统调用，而且不能包含相关的头文件（aio.h）。如果定义的符号值大于0，则可以假定总是支持该特性。[⊖]

某些选项取决于所使用的文件，因此其规则是不同的。例如：如果支持_POSIX_ASYNCHRONOUS_IO，那么必须检查其子选项_POSIX_ASYNC_IO以确定对于将要使用的文件而言是否支持异步I/O。子选项的规则随选项的不同而略有不同，但在大多数情况下，如果选项是与文件相关的，则首先应检查是否定义了子选项。如果定义了，则值-1表示不支持任何文件，其他的值表示支持所有文件。如果没有定义，则必须调用pathconf或fpathconf进行测试（见1.5.6节）。

以上对文件相关性选项规则的讲解适用于SUS2。对于SUS2之前的POSIX1993和POSIX1996来说，其规则是不同的：如果没有定义顶层选项（如前例中的_POSIX_ASYNCHRONOUS_IO），则仅需检查文件；如果定义的值不是-1，则可以假定该选项支持所有文件。

Linux的情况略为混乱：它声称遵循SUS2，但遵循的却是SUS2之前的规则。

解决这种选项混乱的方法之一是：对要使用的每组可选系统调用，将其选项检查封装为一个函数。例如，以下是一个检测是否支持异步I/O函数的例子，并针对Linux进行了调整和

⊖ 如上所编写的检测代码过于简单，因为它没有把未定义的情况或值为0的情况当作特殊情况。对SUS的一些符号和版本可以得到头文件和函数存根，所以可以在运行时用sysconf检测。但是有一些系统，如著名的Linux系统，忽略了未定义情况中的存根。因此，和显示的检测结果一样，不能解释符号是0的系统，但sysconf会报告说它支持该特性。

完善（对pathconf的解释见1.5.6节）：

```
typedef enum (OPT_NO = 0, OPT_YES = 1, OPT_ERROR = -1) OPT_RETURN;

OPT_RETURN option_async_io(const char *path)
{
    #if _POSIX_ASYNCHRONOUS_IO <= 0
        return OPT_NO;
    #elif _XOPEN_VERSION >= 500 && !defined(LINUX)
        #if !defined(_POSIX_ASYNC_IO)
            errno = 0;
            if (pathconf(path, _PC_ASYNC_IO) == -1)
                if (errno == 0)
                    return OPT_NO;
                else
                    EC_FAIL
            else
                return OPT_YES;
        #endif
        EC_CLEANUP_BGN
        return OPT_ERROR;
        EC_CLEANUP_END
    #elif _POSIX_ASYNC_IO == -1
        return OPT_NO;
    #else
        return OPT_YES;
    #endif /* _POSIX_ASYNC_IO */
    #elif _POSIX_VERSION >= 199309L
        return OPT_YES;
    #else
        errno = EINVAL;
        return OPT_ERROR;
    #endif /* _POSIX_ASYNCHRONOUS_IO */
}
```

注意，使用该函数无法找出使用某个选项的代码能否编译。由于必须使用预处理程序，所以此后也必须直接使用选项符号（如_POSIX_ASYNCHRONOUS_IO）。因此，仍然需要进行如下工作：

```
#if _POSIX_ASYNCHRONOUS_IO <= 0
    /* substitute code, or message, or compile failure with #error */
#else
    /*
        code that calls option_async_io and uses the
        option if it returns OPT_YES
    */
#endif
```

为防止读者混淆，这里重新表述如下：选项_POSIX_ASYNCHRONOUS_IO是用来测试系统是否整体上支持异步I/O的；对于SUS2系统，为找出其是否支持想要使用的特殊文件，必须接下来检查_POSIX_ASYNC_IO，这可能需要在运行时调用pathconf或fpathconf。

请不要试图掌握所有选项所对应的符号——它们的数量太多了，其中大多数都是根本用不着的，而且其应用过程中存在太多的不规范。因此，对于本书中涉及的大多数可选系统调用，将会指明必须检查哪些符号。^①但是，在本书示例程序中通常不显示那些实际的选项检查。

① 不要把可选项和非标准弄混了，因为可选项是标准的。本书中只包含了一小部分非标准调用。

1.5.5 sysconf系统调用

`sysconf` 系统调用，不仅可以用来在运行时检查是否支持可选特性（见1.5.4节），还可以用来查看那些与各种实现相关的限制，如一次可以打开的文件数目。通过联机资料或[SUS2002]可以看到这些内容的列表。

sysconf —— 得到系统选项或限制值

```
#include <unistd.h>

long sysconf(
    int name           /* option or limit name */
);
/* Returns option/limit value or -1 (sets errno on error) */
```

假如在代码编译时使用了`sysconf`，那么唯一可能产生的错误就是：使用了某个它所不了解的符号。在这种情况下，它将返回-1并将`errno`设为`EINVAL`。否则，如果没有设置`errno`，则返回值-1仅意味着不支持该选项，或者如果是在测试某个限制，则返回值-1意味着没有该限制。因此，在调用`sysconf`之前，必须把`errno`设置为0。同样，不要使用`ec_neg1`宏，因为它认为所有的返回值-1都是错误的。在下面检测某个限制的示例中，自己进行检测并使用`EC_FAIL`来捕获错误。（“`ec`”相关的解释见1.4.2节。）

```
long value;

#if defined(_SC_ATEXIT_MAX)
    errno = 0;
    if ((value = sysconf(_SC_ATEXIT_MAX)) == -1)
        if (errno == 0)
            printf("max atexit registrations: unlimited\n");
        else
            EC_FAIL
    else
        printf("max atexit registrations: %ld\n", value);
#else
    printf("_SC_ATEXIT_MAX undefined\n");
#endif
```

在Linux系统上的输出为：

```
max atexit registrations: 2147483647
```

这是`long`类型所能容纳的最大值（和标准C的`LONG_MAX`符号相同）。这意味着Linux可能将该注册信息保存在了某个链表中，而不是固定大小的数组中。但在FreeBSD上运行时，得到的结果是这样的：

```
_SC_ATEXIT_MAX undefined
```

这也是正常的，因为该符号不属于POSIX1990，正如FreeBSD所声明遵循的那样。这里只是演示检测符号有否定义的重要性。（由于标准C规定`atexit`的最小值是32，所以可以假定FreeBSD中的这个数为32，虽然这是从C标准中得到的，而不是从`sysconf`。）

1.5.6 pathconf系统调用和fpathconf系统调用

`sysconf`只用于检查系统范围内的选项和限制。其他的选项和限制依赖于正在处理的文件，为取得这些选项和限制，可使用两种非常近似的函数：`pathconf`和`fpathconf`。

pathconf——通过路径得到系统选项或限制

```
#include <unistd.h>

long pathconf(
    const char *path,      /* pathname */
    int name               /* option or limit name */
);
/* Returns option/limit value or -1 (sets errno on error) */
```

fpathconf——通过文件描述符得到系统选项或限制

```
#include <unistd.h>

long fpathconf(
    int fd,               /* file descriptor */
    int name              /* option or limit name */
);
/* Returns option/limit value or -1 (sets errno on error) */
```

这两个函数的第二个参数以及返回结果都是相同的。当已经存在一个打开的文件时，可使用fpathconf；当存在一个有路径名的文件时，不管这个文件打开与否，都可以使用pathconf。可以在[SUS2002]或系统的联机资料中找到name的有效值。

对返回值的解释和上节中的sysconf的解释是相同的。特别的是，仅在errno的值变为（即之前的值不是-1）-1时才意味着发生了错误，因此必须在执行调用前将errno置为0。本书在1.5.4节中列出了使用pathconf的示例代码。

1.5.7 confstr系统调用

confstr的用法和sysconf类似，但它是用于字符串值的，而不是数值：

confstr——得到配置字符串

```
#include <unistd.h>

size_t confstr(
    int name,             /* option or limit name */
    char *buf,           /* returned string value */
    size_t len           /* size of buf */
);
/* Returns size of value or 0 on error (sets errno on error) */
```

执行此调用时，将参数len设置成buf参数传入的缓冲区大小；在输出时，它用以NUL结束的字符串填充缓冲区，如果没有足够的空间放下整个字符串，将会对其进行截取。返回值是整个字符串所需要的大小。如果返回值为0并且改变了errno，则表明出现了错误。如果没有改变errno，则意味着名字是无效的。因此，和前面几个函数相似，在调用前必须将errno置为0。

1.5.8 检测特定的OS

虽然通过POSIX/SUS方式进行特征检测的方法是最好的，但有时确实需要知道所运行的平台是Solaris还是HP/UX、AIX、FreeBSD、Darwin、Linux还是其他什么系统，甚至有时还需要知道其主次版本号。在上文中已有一个示例：对于FreeBSD或Darwin系统不需要定义

`_POSIX_SOURCE`。其他的例子包括某个OS存在漏洞，或具有（如System V消息）超出了它们所声称遵循的标准特性（见下节）。

对OS的检测不存在捷径，并且对某些系统（如Solaris）来说，连非捷径的方法也没有。对大多数系统而言，一旦包含了某个特定的头文件，就会设置某个符号，但我们想要在很早的时候就进行OS检测，甚至在包含头文件`unistd.h`（它通常是POSIX或标准C所包含的第一个头文件）之前。因此，我们在编译阶段设置了一个符号，并确保对于各种不同的系统来说，其值是不一样的，如下所示：

```
$ gcc -DSOLARIS -c xyz.c
```

实际的命令行要比上面复杂一些，并包含在make文件中，但不难理解。

1.5.9 额外特性

虽然像FreeBSD和Darwin之类的系统在宣称其遵循POSIX的标准方面是非常保守的，但是它们远比纯粹的POSIX 1988或POSIX 1990系统更加完善，后者既不包含套接字形式的网络，也不包含System V IPC。FreeBSD具有的这些特征，其中大多数都达到了设想的标准，而且甚至更好。^①

对OS来说，想要通过设置POSIX级别和选项符号来标明其额外特性是行不通的，因为其一：某些特性，例如上文已列出的那些，在SUS3（如果愿意，可称其为POSIX2001）之前的POSIX标准中从未出现过；其二：在SUS3中，它们不是可选的，因此也没有对应的符号。FreeBSD不能通过宣称其遵循某个标准（如SUS1）来解决这个问题，原因即在此。

同时，当然不想将FreeBSD符号的测试遍布在程序的代码中，因为这并不是我们想要的：我们想要的是“套接字”和“System V IPC”。不必要的OS依赖性，会增加代码（尤其是来自不同编程者的代码）移植的难度，因为移植者不清楚原代码中进行OS依赖性测试的目的。那些将代码移植到其他平台（如Linux）的人可能是Linux的专家，但同时也可能对于FreeBSD的特性一无所知。

因此，为调用额外特性，需要建立更多的符号：它们在某处可能是标准，但并不包含在系统所声明遵循的标准中。本书将陆续介绍这些内容，它们可能会在某个例子程序中出现，如下面的代码行所示：

```
#if (defined(_XOPEN_SOURCE) && _XOPEN_SOURCE >= 4) || defined(BSD_DERIVED)
#define HAVE_SYSVMSG
#endif
```

在共享头文件（见下节）中，如果定义了FREEBSD或DARWIN，则要定义BSD_DERIVED，对此的解释见1.5.8节。

人们可以认为上述工作是一种“片面遵循”观点的形式化，虽然这种观点使标准制定者倒退，但却使OS实现者和应用程序编写者找到了本质。

1.6 共享头文件

本书中的所有示例都包含了共享头文件`defs.h`，它包括了对SUS_SUS2的请求和本书1.5.3节中论述的头文件`suvreq.h`。同时，它还包含了我们通常需要的许多标准头文件；如果程序用不到它们，也不必担心包含了多余的内容。在某些较为罕见的情况下，还需要包含另外一些

^① 尽管Mac OS X 10.2.6中的版本Darwin 6.6有其他的System V IPC特性，但没有System V消息。

要使用的头文件。defs.h还包含了几个像syserrmsg（见1.4.1节）这样的功能性函数的原型，对此这里没有显示。以下是defs.h的绝大部分代码，或至少也是其编写时的内容；最新版本在网站上[AUP2003]。

```
#if defined(FREEBSD) || defined(DARWIN)
#define BSD_DERIVED
#endif

#if !defined(BSD_DERIVED) /* _POSIX_SOURCE too restrictive */
#define SUV_SUS2
#include "suvreq.h"
#endif

#ifdef __GNUC__
#define _GNU_SOURCE /* bring GNU as close to C99 as possible */
#endif

#include <unistd.h>

#ifdef __cplusplus
#include <stdbool.h> /* C99 only */
#endif
#include <sys/types.h>
#include <time.h>
#include <limits.h>
#ifdef SOLARIS
#define _VA_LIST /* can't define it in stdio.h */
#endif
#include <stdio.h>
#ifdef SOLARIS
#undef _VA_LIST
#endif
#include <stdarg.h> /* this is the place to define _VA_LIST */
#include <stdlib.h>
#include <stddef.h>
#include <string.h>
#include <ctype.h>
#include <errno.h>
#include <fcntl.h>
#include <assert.h>
#include "ec.h"
```

本例代码三分之二处的#ifdef SOLARIS就是一个展示为什么有时需要OS相关代码的示例。对于“va”宏（用于C中的变量参数列表）究竟应属于stdio.h还是stdarg.h，目前还存在争议。对此问题的处理，Solaris上的gcc和其他系统是不同的。因此，为了有效禁用stdio.h中的定义，这里使用了一点技巧。尽管这种方法并不很完美，但在实在没有别的办法可行而又必须继续进行工作的情况下，使用这种方法以达到禁用的目的是值得的。

本例include的顺序可能有些奇怪，本可以按照字母排序的，但是本例没有这么做，因为在少数情况下，为了在所有要尝试的系统中都能获得干净利落的编译结果，需要变戏法似地按依赖顺序排列，所以本例做了调查，上面看到的的就是其最终的调整结果。虽然头文件可能包含它所依赖的其他头文件，不应该是按依赖顺序排列的，但事实却不是这样的。

1.7 日期和时间

通常在UNIX系统中使用两种类型的时间：日历时间和执行时间。本节将介绍获取和处理

时间的系统调用和函数。此外，还有可以设置的内部定时器，这一内容将在9.7节中进行讨论。

1.7.1 日历时间

日历时间被用于文件的访问、修改和状态改变次数，具有记录用户登录时间、显示当前日期和时间等一系列功能。

日历时间通常有以下4种表示形式：

- 算术类型 `time_t`，这是一个从UTC（通用协调时间）^① 1970年1月1日午夜开始计数的秒数值。将时间存储在文件中并在函数中传递是一种好方法，因为这种方法占空间较小并且是时区无关的。
- 结构类型 `struct timeval`，其时间包含秒和微秒。（既可用于日历时间，也可用于执行时间。）
- 结构类型 `struct tm`，其时间可分解为年、月、日、时、分、秒及其他部分。
- 字符串，例如：Tue Jul 23 09:44:17 2002。

有一整套库函数可用于在时间的三种形式之间进行转换，其中大部分来自标准C。这些函数的命名十分古怪（而不是简单的命名为如 `cvt_tm_to_time`），从意思上根本讲不通。图1-1给出了9种转换函数的应用映射。第10种函数，`time`，用来获取当前时间。

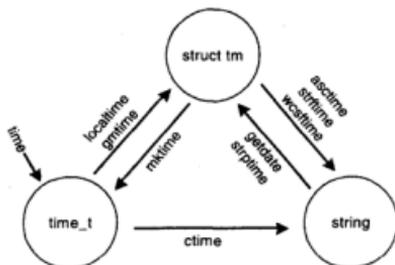


图1-1 时间转换函数

下面列出了 `tm` 结构和 `timeval` 结构^②，紧跟其后列出的是几个主要的日历时间函数的对照表：^③

struct tm —— 分散时间的结构

```
struct tm {
    int tm_sec;           /* second [0,61] (up to 2 leap seconds) */
    int tm_min;          /* minute [0,59] */
    int tm_hour;         /* hour [0,23] */
    int tm_mday;         /* day of month [1,31] */
    int tm_mon;          /* month [0,11] */
    int tm_year;         /* years since 1900 */
    int tm_wday;         /* day of week [0,6] (0 = Sunday) */
    int tm_yday;         /* day of year [0,365] */
    int tm_isdst;        /* daylight-savings flag */
};
```

struct timeval —— gettimeofday 的结构

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

① 通用协调时间原来称作GMT（即格林尼治标准时间）。

② 有一个旧结构 `timeb`，它以秒和微秒为单位表示时间，可以用旧函数 `ftime` 来填充它，而不使用 `gettimeofday`。

③ 一些标准C函数也有重入形式（例如 `ctime_r`），这种形式不使用静态缓冲区，但本书不包含这部分内容。

time——得到当前日期和时间作为time_t

```
#include <time.h>

time_t time(
    time_t *t           /* NULL or returned time */
);
/* Returns time or -1 on error (errno not defined) */
```

gettimeofday——得到当前日期和时间作为timeval

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

int gettimeofday(
    struct timeval *tvalbuf, /* returned time */
    void *dummy             /* always NULL */
);
/* Returns 0 on success or -1 (maybe) on error (may set errno) */
```

localtime——把time_t转换成本地分散时间

```
#include <time.h>

struct tm *localtime(
    const time_t *t /* time */
);
/* Returns broken-down time or NULL on error (sets errno) */
```

gmtime——把time_t转换成UTC分散时间

```
#include <time.h>

struct tm *gmtime(
    const time_t *t /* time */
);
/* Returns broken-down time or NULL on error (sets errno) */
```

mktime——把本地分散时间转换成time_t

```
#include <time.h>

time_t mktime(
    struct tm *tmbuf /* broken-down time */
);
/* Returns time or -1 on error (errno not defined) */
```

ctime——把time_t转换成本地时间字符串

```
#include <time.h>

char *ctime(
    const time_t *t /* time */
);
/* Returns string or NULL on error (errno not defined) */
```

asctime——把分散时间转换成本地时间字符串[⊖]

```
#include <time.h>

char *asctime(
    const struct tm *tmbuf /* broken-down time */
);
/* Returns string or NULL on error (errno not defined) */
```

⊖ 当SUS出错时，asctime不返回NULL，但无论如何，最好还是进行检测。

strftime —— 把分散时间转换成带格式的字符串

```
#include <time.h>

size_t strftime(
    char *buf,           /* output buffer */
    size_t bufsize,     /* size of buffer */
    const char *format,  /* format */
    const struct tm *tmbuf /* broken-down time */
);
/* Returns byte count or 0 on error (errno not defined) */
```

wcsftime —— 把分散时间转换成带格式的宽字符串

```
#include <wchar.h>
size_t wcsftime(
    wchar_t *buf,       /* output buffer */
    size_t bufsize,    /* size of buffer */
    const wchar_t *format, /* format */
    const struct tm *tmbuf /* broken-down time */
);
/* Returns wchar_t count or 0 on error (errno not defined) */
```

getdate —— 用某些规则把字符串转换成分散时间

```
#include <time.h>

struct tm *getdate(
    const char *s          /* string to convert */
);
/* Returns broken-down time or NULL on error (sets getdate_err) */
```

strptime —— 把字符串转换成带格式的分散时间

```
#include <time.h>

char *strptime(
    const char *s,        /* string to convert */
    const char *format,  /* format */
    struct tm *tmbuf     /* broken-down time (output) */
);
/* Returns pointer to first unparsed char or NULL on error (errno not defined) */
```

由上面对照表可以看出，在这些函数中都没有设置 `errno`，并且其中大部分甚至都不返回出错指示，尽管测试任何返回的指针是否为 `NULL` 是一个好主意。`getdate` 更为奇怪：它在错误发生时将（从不在其他地方使用的）变量或宏 `getdate_err` 的值设置成一个从 1 到 8 的数值，用以表明问题的种类；有关细节请参阅 [SUS2002]。

对于 `time_t`，大多数 UNIX 系统都用 `long` 整型来实现，通常只包含 32 位（带符号的）；因此按秒计最大可到 2038 年 1 月 19 日。^① 某些时候，它需要更多的位数，那么需要将它改成 64 位的。^② 为了预防这种情况，应始终使用 `time_t` 类型（而不是其他类型，如 `long` 类型），而不用假定 `time_t` 类型是用什么实现的。对历史时间来说，`time_t` 已经不够用了，因为在许多系统中它最早（取负数时）只能到 1901 年。

由于没有假定 `time_t` 类型是用什么实现的，所以会给两个这种类型的时间相减带来困难。而这种时间相减又是很常用的，因此提供了以下专用函数：

① 如果遵循 19 年一个版本循环的话，到那时，本书的第 4 版应该出来了。

② 很明显，64 位是太大了。但是对计算机来说，32 后的数字经常就是 64。

difftime——计算两个time_t值的差

```
#include <time.h>

double difftime(
    time_t time1,      /* time */
    time_t time0      /* time */
);
/* Returns time1 - time0 in seconds (no error return) */
```

和time相比，gettimeofday的精度更高，但其他函数没有把struct timeval作为参数。然而，它们使用tv_sec字段，因为它是time_t类型的，可以单独处理微秒。对于tv_usec类型来说，所能做出的全部安全假设就是，它是某种大小的带符号整数。然而，它没有理由超过32位，因为每秒钟只包含1 000 000微秒，因此为tv_usec字段分配一个长整数类型就可以了。要获取某个运行时间段的间隔时间，可以两次调用gettimeofday，然后对tv_sec字段使用difftime，对tv_usec字段只使用减法。

某些实现(FreeBSD、Darwin和Linux，不包括Solaris)使用gettimeofday的第二个参数返回时区信息，但这不是标准的做法。大多数实现方案以返回值-1表示出错，甚至还会设置errno。其他方案通常返回0，因此在所有情况下，都可以安全地检测到返回的错误。

time_t通常用UTC来表示，而断开的时间(struct tm)和字符串则既可以用UTC表示，也可以用当地时间表示。在输入或输出时间时使用当地时间是很方便的，但这会使事情变得复杂。这要求必须采用某些措施，使得计算机了解用户所在的时区以及当前是标准时间还是夏时制时间。更糟的是，当输出过去的时间时，计算机必须推算那时是标准时间还是夏时制时间。通过一些和当地时间相关的函数，可以很好地处理这一问题。

用户的时区信息保存在环境变量TZ中，如果没有设置TZ，那么将作为系统默认值。为了提供移植性，提供了一个函数和三个全局变量来处理时区和夏时制时间设置：

tzset——设置时区信息

```
#include <time.h>

extern int daylight;      /* DST? (not in FreeBSD/Darwin) */
extern long timezone;    /* timezone in secs (not in FreeBSD/Darwin) */
extern char *tzname[2];  /* timezone strings (not in FreeBSD/Darwin) */

void tzset(void);
```

应用程序可以调用tzset设法得到时区(从TZ或其他地方)并设置三个全局变量，如下所示：

- 如果该时区中不应用夏时制时间，daylight置为0(false)，否则置为非0值(true)。转换周期按已定义的实现方式内建于库函数中。
- timezone被置为UTC和当地标准时间之间的秒数差值。(除以3 600可以得到小时数的差值。)
- tzname是包含两个字符串的数组，tzname[0]用于在标准时间中指明本地的时区，tzname[1]用于指明夏时制时间。

这三个全局变量在比POSIX2002更早的POSIX中是找不到的，在FreeBSD或Darwin中也没有进行定义。但通常并不需要调用tzset或直接读取全局变量，仅使用相关的标准函数就已经足够了。

下面是一个示例:

```
tzset();
printf("daylight = %d; timezone = %ld hrs.; tzname = %s/%s\n",
       daylight, timezone / 3600, tzname[0], tzname[1]);
```

其输出为:

```
daylight = 1; timezone = 7 hrs.; tzname = MST/MDT
```

至于如何使用日历时间函数的详细内容,尤其是涉及格式化字符串的内容,请参阅[SUS2002]或浏览一本关于标准C的好书(比如[Har2002])。以下是需要记住的几点:

- 要忽略返回错误的可能性,可将`ctime(t)`定义为`asctime(localtime(t))`。
- 为了允许偶尔漏掉的秒数, `tm`结构中的`tm_sec`字段的取值可达到61(而不是只到59)。
- `tm`结构中的`tm_year`字段的取值为与1900年的差值,它完全不存在千年虫问题,只是看起来特别一些(2002年被表示为102)。
- 同样特别的是, `ctime`和`asctime`在输出字符串的最后都加入了一个换行符。
- 在几个函数中都包含有指向`time_t`的指针,虽然按值调用`time_t`同样会工作得很好。这是一种从最早期的UNIX遗留下来的现象,那时C语言还不存在长整数类型,因此必须输入由两个16位整数组成的数组。
- `time`中的`time_t`指针参数是完全不需要的,应当设置为NULL。看起来像是为`time`提供了一个可供使用的缓冲区,但实际上并不需要缓冲区,因为返回的`time_t`是一个算术类型。
- 在大多数实现中,上面提及的返回指向`tm`结构指针的函数,其内部都使用静态缓冲区。因此在调用其他的此类函数之前,请先使用或者复制其缓冲区的内容。如果需要,可以在多线程程序中使用它们带有`_r`后缀的重用形式。
- 使用格式的函数,如`strftime`、`wcsftime`、`getdate`和`strptime`,是非常复杂的,但也是非常有用的,因此应该掌握。前两种在标准C中,后两种在SUS中。在FreeBSD或Darwin中没有`getdate`,然而在GNU C库中有它的一个版本,如果需要可以进行安装。
- 当获取用户输入的日期和时间时可使用`getdate`,当读取包含日期的数据文件时可使用`strptime`或`getdate`。原因是`strptime`只能接受单一格式的输入,而对于用户来说,保证日期和时间的格式正确实在太难了。`getdate`使用了一个完整的格式列表,并通过逐项比较来获得匹配。

1.7.2 执行时间

执行时间是用来测量时间间隔、进程执行的时间以及审计记录的。内核自动记录每个进程的执行时间。间隔时间以秒的各种子级为单位,其范围从十亿分之一秒到百分之一秒。

和日历时间一样,执行时间的类型和函数也容易造成混淆。先看一下其主要类型:

- `clock_t`, 这是一个标准C中的运算类型(通常是一个长整型,但并不保证都是)。这是一个以`CLOCKS_PER_SEC`或时钟滴答声为单位的时间间隔。
- 结构`timeval`, 以秒和毫秒为单位的时间间隔(也可以用于日历时间,详细论述见上一节)。
- 结构`timespec`, 以秒和十亿分之一秒为单位的时间间隔。

以下是`timespec`的结构(前面已经列出了`timeval`),其定义位于头文件`<time.h>`中:

struct timespec ——以秒和纳秒为单位的时间的结构

```

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

```

主要有三个函数可用于测量时间间隔：`gettimeofday`、`clock`和`times`。其中，`gettimeofday`在上节中已经详细讨论过，后两者及`times`中使用的结构如下：

clock ——得到执行时间

```

#include <time.h>

clock_t clock(void);
/* Returns time in CLOCKS_PER_SEC or -1 on error (errno not defined) */

```

times ——得到进程和子进程的执行时间

```

#include <sys/times.h>

clock_t times(
    struct tms *buffer /* returned times */
);
/* Returns time in clock ticks or -1 on error (sets errno) */

```

struct tms ——times系统调用的结构

```

struct tms {
    clock_t tms_utime; /* user time */
    clock_t tms_cutime; /* user time of terminated children */
    clock_t tms_stime; /* sys time */
    clock_t tms_cstime; /* sys time of terminated children */
};

```

`times`返回从过去某个时间点（通常是系统启动时间）开始已消失的时间，因此可称为“挂钟”时间或“实时”时间。这和上节中函数返回的时间是不同的，后者返回的是从新纪元（通常是1970年1月1日）开始的时间。

`times`使用时钟滴答声作为单位；使用`sysconf`（见1.5.5节）得到每秒中时钟报时滴答声的方法如下：

```
clock_ticks = sysconf(_SC_CLK_TCK);
```

另外，`times`用更多的特定信息加载`tms`结构：

- `tms_utime`（用户时间）是执行进程的用户代码指令所花费的时间。它仅包括CPU时间，不包括等待运行的时间。
- `tms_stime`（系统时间）是代表进程执行系统调用所花费的CPU时间。
- `tms_cutime`（子用户时间）是用户CPU时间的总和，包括已经终止的所有进程的子进程所用的时间以及父进程发出`wait`系统调用所用的时间（详见第5章）。
- `tms_cstime`（子系统时间）是所有用来终止和等待子进程的系统CPU时间的总和。

虽然`clock`的返回类型和`times`相同，但其值是进程启动以来所使用的CPU时间，而不是真实时间，单位由宏`CLOCKS_PER_SEC`确定，而不是时钟滴答声。（除单位不同外）它等于`tms_utime + tms_stime`。

在SUS中, CLOCKS_PER_SEC被定义为1 000 000 (即微秒), 但非SUS系统可以使用不同的值, 因此总是使用宏。(FreeBSD将CLOCKS_PER_SEC定义为128; Darwin定义为100。)

如果clock_t就像常见的那样, 是一个32位的带符号整数, 并以微秒为单位, 虽然36分钟左右就循环一遍, 但至少在进程启动时clock才开始计时。times按时钟滴答声进行工作, 可以对更长的时间进行计时, 但同时它启动得更早, 即如果UNIX系统已经运行了数周或数月 (这通常很常见), 那就更早了! 但是, 32位带符号的clock_t如果按时钟滴答声进行工作, 那么就算250天也循环不了一遍, 因此如果能将系统设置为每半年重新启动一次, 就能解决这个问题。

然而, 还不能假设clock_t就是一个32位的带符号整数, 或者不能假定是一个带符号的数, 甚至也不能假定是一个整数。它可能是一个无符号长整数型的, 甚至可能会是一个双精度型的。

由于times可以一次给出实际时间和用户及系统的CPU时间, 所以在本书中将用它对各种程序进行计时。下面是要使用的两个方便的函数, 一个用来开始计时, 另一个用来停止计时并输出结果:

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

static struct tms tbuf1;
static clock_t real1;
static long clock_ticks;
void timestart(void)
{
    ec_negl( real1 = times(&tbuf1) )
    /* treat all -1 returns as errors */
    ec_negl( clock_ticks = sysconf(_SC_CLK_TCK) )
    return;
}

EC_CLEANUP_BGN
    EC_FLUSH("timestart");
EC_CLEANUP_END
}

void timestop(char *msg)
{
    struct tms tbuf2;
    clock_t real2;

    ec_negl( real2 = times(&tbuf2) )
    fprintf(stderr, "%s:\n\t\"Total (user/sys/real)\", %2f, %2f, %2f\n"
        "\t\"Child (user/sys)\", %2f, %2f\n", msg,
        ((double)(tbuf2.tms_utime + tbuf2.tms_cutime) -
         (tbuf1.tms_utime + tbuf1.tms_cutime)) / clock_ticks,
        ((double)(tbuf2.tms_stime + tbuf2.tms_cstime) -
         (tbuf1.tms_stime + tbuf1.tms_cstime)) / clock_ticks,
        (double)(real2 - real1) / clock_ticks,
        (double)(tbuf2.tms_cutime - tbuf1.tms_cutime) / clock_ticks,
        (double)(tbuf2.tms_cstime - tbuf1.tms_cstime) / clock_ticks);
    return;
}

EC_CLEANUP_BGN
    EC_FLUSH("timestop");
EC_CLEANUP_END
}
```

如下例所示, 在实际计时的时候, 只需要在间隔开始时调用timestart, 并在结束时调用timestop就可以了。为了比较本例中还包括了对gettimeofday和clock的调用:

```

#define REPS 1000000
#define TV_SUBTRACT(t2, t1)\
    ((double)(t2).tv_sec + (t2).tv_usec / 1000000.0 -\
    ((double)(t1).tv_sec + (t1).tv_usec / 1000000.0))

int main(void)
{
    int i;
    char msg[100];
    clock_t c1, c2;
    struct timeval tv1, tv2;
    snprintf(msg, sizeof(msg), "%d getpid's", REPS);
    ec_negl( c1 = clock() )
    gettimeofday(&tv1, NULL);
    timestart();
    for (i = 0; i < REPS; i++)
        (void)getpid();
    (void)sleep(2);
    timestop(msg);
    gettimeofday(&tv2, NULL);
    ec_negl( c2 = clock() )
    printf("clock(): %.2f\n", (double)(c2 - c1) / CLOCKS_PER_SEC);
    printf("gettimeofday(): %.2f\n", TV_SUBTRACT(tv2, tv1));
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}

```

Linux系统上的输出如下 (Solaris、FreeBSD和Darwin上的输出与之相似):

```

1000000 getpid's:
    "Total (user/sys/real)", 0.72, 0.44, 3.19
    "Child (user/sys)", 0.00, 0.00
clock(): 1.16
gettimeofday(): 3.19

```

可以看出, `clock`的计时结果是1.16, [⊖]其恰好是0.72和0.44之和。同时, `gettimeofday`的测量结果和`times`相同。这简直太棒了, 否则还要做更多的解释。

如果需要的精度比`times`所能提供的高, 则可以使用`getrusage` (见5.16节)。

1.8 关于示例代码

本书中有许多示例代码, 只要按照常规, 比如在手册及图书的关于对话框或声明页中进行了版权声明, 就可以在自学或商业产品中自由地使用它们。详细信息请看网页 www.basepath.com/aup/copyright.htm。

为了节省空间, 有时对本书的示例进行了一些删节 (例如本章开始时的错误处理代码和`defs.h`的代码), 但其完整文件可以在www.basepath.com/aup/ [AUP2003]找到。(那里还有许多其他信息, 如勘误表等。)网上的代码随时都在更新, 比如对出版本书后发现的一些漏洞进行了更新。因此, 如果对本书中某些地方是否有误出现迷惑, 请首先查看网上较新的代码。然后, 如果仍然认为有错误, 请发电子邮件到aup@basepath.com来告诉我。如果对本书中所讲述的内容持有不同的观点或认为有更好的解决办法, 或者只是想告诉我这是你读过的最好的

⊖ 在本书的1985年的版本中, 1000条`getpids`需要1.43秒。

书，也欢迎来信。

本书的例子是用C99编写的，由于可以自己定义宏和typedef，所以即便你的编译器级别不够，这些较新的内容（如intptr_t、bool）也不会出现问题。大部分代码已经使用gcc编译器进行了测试，测试平台包括在Intel CPU上运行的SuSE Linux 8.0（基于Linux 2.4）、FreeBSD 4.6和Solaris 8，以及在基于PowerPC的iMac上运行的Darwin 6.6（Mac OS X 10.2.6的UNIX部分）。随着时间的推移，可以在更多系统上进行更多的测试，如果有什么改动，会在网站上公布。

1.9 必要的资源

下面是一些除本书之外必要资源的列表，在编程时可能会用到：

- 所用语言的优秀参考书。对C来说，最好的书是：Harbison和Steele编写的《a reference manual》[Har2002]；对C++来说，也许是Bjarne Stroustrup编写的《The C++ Programming Language》[Str2000]。最好是有一本包含了所有答案的书，即使找到这样的书较困难。但当得到这样的书后，就不再需要找其他的书了，因此一本书与多本书相比不仅节省了大量时间，也节省了空间。
- Open Group SUS（单一UNIX规范），网址为：www.unix.org/version3，这是一个极好的站点。在左边框架点击函数名，右边框架就显示说明。也可以在CD或书中获得这些说明，但网站上的版本是免费的。
- Usenet新闻组comp.unix.programmer。可以张贴任何UNIX问题，无论多困难，都会得到答案，这个答案也许不正确，但通常至少可以指出正确的方向。（为了进入新闻组，需要一个能够处理新闻组和新闻信息包的邮件客户端。ISP可能会提供，如果没有提供可以向www.surpernews.com或www.giganews.com付费购买。或者通过位于<http://groups.google.com/grphp>的Google Group访问。）也有一些Web论坛，例如www.unix.com，但是它们不如comp.unix.programmer好。只允许张贴UNIX（或者Linux或者FreeBSD）问题，如果张贴的是C或C++问题，则会被立刻删除。对那些题目有单独的新闻组（成千上万的新闻组在一起），Google Group也允许通过Usenet档案文件查看20年前父辈们是否有人提过相同的问题。
- Google，Web搜索站点。假设sigwait系统调用在FreeBSD上出错了，或者不能确信Linux是否实现了异步I/O，那么可以按照次序得到答案（例如从FreeBSD或Linux站点），也可以通过搜索获得答案。大约有一半时间，我想查找的资料都不是从官方获得的，而是从Google碰巧检索到的一些无名论坛张贴或私人站点得到的。
- GNU C库，一个很大的编码示例资源库，可以从www.gnu.org/software/libc上下载得到。
- 系统的联机资料。尽管查找起来有些不太方便（apropos命令可以提供一些帮助），然而一般情况下，一旦找到正确的记录，就会受益匪浅。最好是多花费点时间掌握一个有效使用它们的方法，以便在用到的时候能够得心应手。

练习

- 1.1 使用头文件defs.h编写一个可以显示“Hello World”的C程序，其中至少要使用一个错误检测宏（例如ec_neg1）。该练习的目的是为了让读者在系统上安装阅读本书所需的所有代码，以确保C编译器能够正常运行，并且确保安装了所需要的工具，例如文本编辑器和make工具。如果没有权限访问UNIX或Linux系统，那么处理这个问题也是该练习的一部分。

- 1.2 编写一个程序，用来显示系统遵从何种POSIX和SUS版本，并且要满足不同的请求（例如，SUS_SUS2）。
- 1.3 编写一个程序，用来显示系统所有的sysconf值。构思在不必逐个输入符号的情况下如何生成包含sysconf调用的所有行。
- 1.4 和练习1.3一样，但要求显示的是pathconf，该程序以路径为参数。
- 1.5 和练习1.3一样，但要求显示的是confstr。
- 1.6 编写一个不带命令行选项的date命令程序。为了扩大实用性，其控制的功能最好尽量多。依据SUS或POSIX中的某个标准，不要抄袭GNU源代码或其他类似示例。



第2章 基本文件I/O系统调用

2.1 概述

本章将讨论普通文件的基本I/O。对于高级文件、特殊文件、管道、命名管道以及套接字的I/O系统将分别在第3章、第4章、第6章、第7章和第8章进行介绍。

在开始讲解之前，首先举一个简单的例子，其中使用了大家熟悉的4个系统调用：`open`、`read`、`write`及`close`。此函数完成了文件复制功能（类似于`cp`命令）：

```
#define BUFSIZE 512

void copy(char *from, char *to) /* has a bug */
{
    int fromfd = -1, tofd = -1;
    ssize_t nread;
    char buf[BUFSIZE];

    ec_negl( fromfd = open(from, O_RDONLY) )
    ec_negl( tofd = open(to, O_WRONLY | O_CREAT | O_TRUNC.
        S_IRUSR | S_IWUSR) )
    while ((nread = read(fromfd, buf, sizeof(buf))) > 0)
        if (write(tofd, buf, nread) != nread)
            EC_FAIL
    if (nread == -1)
        EC_FAIL
    ec_negl( close(fromfd) )
    ec_negl( close(tofd) )
    return;

    EC_CLEANUP_BGN
    (void)close(fromfd); /* can't use ec_negl here! */
    (void)close(tofd);
    EC_CLEANUP_END
}
```

根据1.4.1节中的提示，试着找出此程序中的bug。如果找不到，可以查看2.9节。

下面简单介绍此函数，在以后的章节中还将对其做详细的论述。第一个`open`调用以读方式打开输入文件（`O_RDONLY`标志），并返回一个文件描述符用于随后的系统调用。第二个`open`调用实现如下功能：如果文件不存在，就创建一个新文件（`O_CREAT`标志），否则将文件长度截短为0（`O_TRUNC`标志）。不论发生哪种情况，都以可写方式打开文件，并返回一个文件描述符。`open`的第三个参数规定了新建文件的使用权限（一般只希望文件所有者才有读写文件的权利）。`read`将按照第三个参数给定的字节数把数据读入第二个参数指定的缓冲区，返回结果为读入的字节数、文件结尾标志0或者错误标志-1。`write`从第二个参数指定的缓冲区中写入第三个参数给定的字节数，返回结果为写入的字节数，如果返回的字节数与要求写入的字节数不相等，则认为写入出错。最后，`close`关闭文件描述符。

这里没有使用`if`语句，`fprintf`调用以及`goto`等处理错误，而是使用了方便的宏`ec_negl`和`EC_FAIL`，当`ec_negl`的参数是-1时，对函数进行错误处理，而`EC_FAIL`则总

是按错误处理函数。(实际上,它们是跳转到清除代码,这些代码通过EC_CLEANUP_BGN和EC_CLEANUP_END界定,在此程序中没有定义任何内容。)1.4.2节对以上内容作了介绍。

2.2 文件描述符及打开文件描述

每一个UNIX进程都有一个文件描述符范围,其大小为0到N,N标志文件描述符的最大数。N的大小取决于UNIX的版本以及系统配置,但一般最小为16,目前UNIX系统中此数值更大,为了发现系统运行的实际N值,可以调用带有参数_SC_OPEN_MAX sysconf(见1.5.5节),其代码如下:

```
printf("_SC_OPEN_MAX = %ld\n", sysconf(_SC_OPEN_MAX));
```

在Linux系统中,N值为1024,在FreeBSD系统中,N值为957,在Solaris系统中,N值为256。目前除了嵌入式操作系统外,大概没有N值为16的系统了。

2.2.1 标准文件描述符

按照惯例,进程开始运行时前三个文件描述符就已经打开了。文件描述符0是标准输入,文件描述符1是标准输出,文件描述符2是标准错误输出,并且文件描述符2对控制终端常常是打开的。除了使用数字之外,最好使用符号常数STDIN_FILENO、STDOUT_FILENO和STDERR_FILENO。

UNIX过滤程序从STDIN_FILENO文件中读取数据,并向STDOUT_FILENO文件写入数据;shell命令在管道操作中使用此种方式。因为写入STDOUT_FILENO的任何内容都有可能转入管道或者文件,而且从来不会发现输出重定向。(通过shell实现输出重定向是最常见的,因此STDERR_FILENO应该用于保存重要信息。)

这些标准文件描述符中任何一个都可打开文件、管道、FIFO、设备甚至是套接字。最好以一种不依赖于目的或源的类型的方式编程,但有的时候不能实现。如对于屏幕编辑器来说,如果标准输出不是终端设备,它也许不能正常工作。

调用read和write时可以立即使用这三个标准文件描述符。用于文件、管道等其他的文件描述符可以通过进程本身获得。父进程传递给子进程的不仅是这三个标准文件描述符,具体内容会在第6章讨论管道和进程的关系时予以阐述。

2.2.2 应用文件描述符

通常情况下,UNIX可以像操作文件一样操作文件描述符,因此可以对其进行读操作、写操作或者读写操作。文件描述符不能应用于与文件无相似方面的通信机制,如消息队列,因为不能对它们进行读写操作(对此有专门的调用)。

只有几种方法可以得到新的打开文件描述符,对此我们现在不做深入的研究,但知道以下内容是有益的:

- **open**: 用于打开大多数具有路径名的文件,包含普通文件、特殊文件以及命名管道(FIFO)。
- **pipe**: 用于创建和打开非命名管道(第6章)。
- **socket**、**accept**和**connect**: 用于网络操作(第8章)。

在C中没有规定文件描述符的类型(像进程ID一样),因此仅使用普通的int。[⊖]

[⊖] 我最初想在本书中介绍fid_t类型,以使书中的例子更具可读性,但后来我决定不这么做,因为在实际的代码中可以使用普通的int类型,因此不必熟悉它。

2.2.3 打开文件描述和共享

如第1章所述，文件描述符只是对每个进程表的索引。进程表中每个记录项指向一个全系统的打开文件描述（也就是众所周知的文件表记录项），接着文件描述指向文件数据（通过信息节点的内存复制）。如图2-1所示，多个文件描述符，甚至来自不同进程的文件描述符都可以指向同一个文件描述。

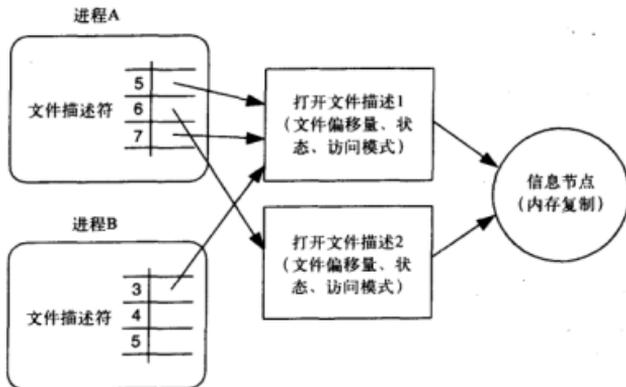


图2-1 文件描述符、打开文件描述符和信息节点

每一个open或pipe系统调用都会创建一个新的打开文件描述和新的文件描述符。在图中进程A两次打开了同一个文件，得到了文件描述符5和6，并创建了打开文件描述1和2。接着通过文件描述符复制机制（通过dup、dup2和fork系统调用实现），进程A得到了文件描述符5的复制品文件描述符7，这意味着它指向的打开文件描述与文件描述符5的是相同的。进程B是进程A的子进程，并且文件描述符3同样是文件描述符5的复制品。

我们一再引用图2-1，因为通过它可以了解很多信息。例如，在2.8节讲解文件偏移量时，我们会看到，由于进程A的文件描述符5和7及进程B的文件描述符3共享同一个打开文件描述，它们将共享同一个文件偏移量。

2.3 文件权限位符号

回顾1.15节，可以知道文件具有9种权限位：其包括三种用户类型（所有者、组和其他用户）和三种访问权限（读、写和执行文件）的组合。在ls命令的输出中，总可以看到这9个文件权限位：

```
-rwxr-xr-x 1 marc users 29808 Aug 4 13:45 hello
```

所有人都认为这9个文件权限位应该在一起，并有一定的顺序（所有者、组和其他用户），但实际上并不必如此，只要有这9个权限位即可。因此从POSIX1988时起，就有了权限位符号，用于替代传统的八进制数表示。这些符号的形式为S_Ipwww，其中，P代表访问权限（R、W或X），WWW代表谁操作（USR、GRP或OTH）。这就表示出了全部的9个符号。

例如，对于上述文件，不用八进制的755，可用符号表示如下：

```
S_IRUSR | S_IWUSR | S_IXUSR | S_IRGRP | S_IXGRP | S_IROTH | S_IXOTH
```

当USR、GRP或OTH拥有所有三种访问权限时，它们将使用单独的符号，这些符号具有如下形式：S_IRWXw，这里w是“whom”的第一个字母，可以是U或G，也可以是O。因此可以按如下方式写文件访问权限：

```
S_IRWXU | S_IRGRP | S_IXGRP | S_IROTH | S_IXOTH
```

尽管此种描述方式比八进制数^①的可读性要差，并且出现错误的概率更高，但操作者可以自由安排权限位的位置。因为在编写程序时可能仅使用一些组合（例如，可能一次只用其创建的1个或2个数据文件，或者是创建的某个目录），所以最好是一次性就定义好所需的宏，而不是在所有的位置都使用长的S_I*符号序列。在本书中，我们只使用了下列定义，它们被包含在了defs.h文件中（见1.6节）：

```
#define PERM_DIRECTORY S_IRWXU
#define PERM_FILE      (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
```

注意，要根据所使用的情況定义宏，而不是依据权限位。因此采用PERM_FILE方式定义宏，而不再采用PERM_RWURGO形式定义文件权限位，因为这样仅需改变一次宏，就可以改变整个应用程序的访问权限策略。

2.4 open和creat系统调用

open —— 打开或创建文件

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

int open(
    const char *path,      /* pathname */
    int flags,             /* flags */
    mode_t perms           /* permissions (when creating) */
);
/* Returns file descriptor or -1 on error (sets errno) */
```

我们可以使用open打开一个已经存在的文件（普通文件、特殊文件或命名管道），或创建一个新文件，但它只能创建普通文件。创建特殊文件需使用mknod（见3.8.2节），命名管道使用mkfifo（见7.2.1节）。文件一旦打开，read、write、lseek、close以及其他调用就可以使用返回的文件描述符值，至于其他调用将在以后的章节中进行讨论。

2.4.1 打开已存在文件

首先讨论如何打开一个由path指定的已经存在的文件。如果flags的值是O_RDONLY，就以只读方式打开文件；如果是O_WRONLY，就以只写方式打开文件；如果是O_RDWR，就以读写方式打开文件。^②

使用1.1.5节中说明的算法，进程需要读、写或读写以及打开文件的许多种权限。例如，如果进程的有效用户ID与文件的所有者相匹配，就一定要设置文件所有者的读、写或读写权限位。

对于一个已经存在的文件，参数perms是没有用的，通常将其省略，因此调用open时只有2个参数。

① 一个技术评论员指出，即使采用八进制数，内核也可以将其映射为内部正在使用的文件系统。

② 为什么有3个参数？我们可以忽略O_RDWR参数，而只使用O_RDONLY或O_WRONLY吗？不行，因为实现已经将O_RDONLY定义为0，而不是某个位。

文件偏移量（读和写文件用到的）位于文件的第一个字节。更多详细内容见2.8节。

下面是打开一个已存在文件的程序代码：

```
int fd;
ec_negl( fd = open("/home/marc/oldfile", O_RDONLY) )
```

`open`失败的原因很多，大多数情况下系统将告诉用户错误所在。当一个路径指向一个不存在的文件时（`ENOENT`）会出现错误，此时所需的解决方案与访问权限（`EACCES`）出现错误时所需的是不同的。我们的普通错误检测和错误报告对此控制得很好（宏“`ec`”见1.4.2节）。

`open`成功时返回的文件描述符是一个可用的最小整数，正常情况下用户不必关心此数。但有时这个数是有用的，当需要重定向某个标准文件描述符0、1、2时（见2.2.1节），用户首先关闭需要重定向的文件，随后打开该文件时就需要刚才得到的这个整数（例如1）。

2.4.2 创建新文件

当文件不存在时，如果用OR操作向`open`里的`flags`中加入标志`O_CREAT`，那么`open`将创建一个新文件。当然这样可以创建一个新的只读文件，但没有任何意义，因为如上创建的新文件没有任何可读内容。因此一般需要`O_CREAT`与`O_WRONLY`或`O_RDWR`一起使用。现在用带着参数`perms`了，如下例所示：

```
ec_negl( fd = open("/home/marc/newfile", O_RDWR | O_CREAT, PERM_FILE) )
```

参数`perms`仅在创建新文件时有效。对于一个已经存在的文件，它没有任何作用。

新建文件的权限位是系统调用中的权限位和进程文件方式创建屏蔽字的补码进行与操作的结果，典型情况是登录时设置（通过`umask`命令），或者`umask`系统调用（见2.5节）。“对补码进行与操作”是指文件方式屏蔽字中设置的权限位将清除文件相应的权限位。因此，即使调用带有标志`S_IWOTH`的`open`，`002`屏蔽字也会致使`S_IWOTH`位（其他用户写权限）清除。然而作为一个程序员，通常不必关心此屏蔽字，因为它是用户限制访问权限时使用的。

如果用带有标志`O_WRONLY`或`O_RDWR`的`open`创建文件，但权限位不允许写会怎么样？因为它是一个新文件，所以仍可以进行写操作，然而，当下次打开它时，它已经存在，这时权限位将按照上节讲述的那样控制访问。

用户有时需要一个新的、没有任何数据的文件。也就是说，如果文件已经存在，需要将其所有数据清除，并置文件偏移量为0。标志`O_TRUNC`可以实现此功能：

```
ec_negl( fd = open("/home/marc/newfile", O_WRONLY | O_CREAT | O_TRUNC,
    PERM_FILE) )
```

因为`O_TRUNC`能够破坏数据，所以只要进程具有写权限，就可以清除已存在文件的数据，因为它是写形式的一种。但对于具有`O_RDONLY`标志的文件，它就不起作用了。

对于一个新文件（即用`O_CREAT`创建的），因为需要为新文件创建一个新链接，所以需要在父目录中为其设置写权限。对于一个已存在文件来说，目录中的权限已无关紧要了，它依赖的是文件的权限。你可能要自问，“如何完成这些操作呢？”

也许还需要说一下，有时需要在路径（即`home`和`marc`）的中间目录上搜索（执行）文件权限。然而通常也可应用到路径的任何地方，以后不再重复说明。

`O_TRUNC`不必与`O_CREAT`同时使用，其实际上是当文件存在时，将文件长度截为0，如果文件不存在，则操作失败（如创建了日志文件，则可以具有日志特征；如果没有日志文件，将不存在日志功能）。

`O_WRONLY|O_CREAT|O_TRUNC`这个组合是很常见的（“创建或截短一个具有只写权限

的文件”)，以致于具有专门的系统调用实现此功能：

```

creat —— 创建或清空文件以便写入

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

int creat(
    const char *path,      /* pathname */
    mode_t perms          /* permissions */
);
/* Returns file descriptor or -1 on error (sets errno) */

```

采用open打开一个已经存在的文件只需要第一个参数和第二个参数 (path和flags)；使用creat创建新文件仅需要第一个参数和第三个参数。实际上，creat仅仅是一个宏：

```
#define creat(path, perms) open(path, O_WRONLY | O_CREAT | O_TRUNC, perms)
```

为什么不能忽略creat，而仅使用open呢？这样不就不用记忆两个系统调用，而且标志也总能描述得很清楚了吗？这是个好主意，因此这本书将只使用open。^①

在创建新文件时，我们跳过了一个重要的内容：谁是新文件的所有者？回顾1.1.5节，可以知道每一个文件都有一个所有者用户ID和一个所有者组ID，我们简称所有者和组。下面是如何为新文件设置所有者和组的方法：

- 所有者通过进程的有效用户ID设置。
- 组可以被设置成父目录的组ID，也可以被设置成进程的有效组ID。

尽管可以通过stat系统调用（见3.5.1节）寻找组ID使用的方法，但是应用程序不能设定组ID使用哪种方法，也不能使用chown系统调用（见3.7.2节）强制组ID使用它需要的方法。根本上很少有应用程序关心组ID。

还有另一个标志O_EXCL，它和O_CREAT标记一起使用时，如果文件存在，则创建文件操作将失败。如果没有使用O_CREAT标志的open是“文件存在则打开文件，文件不存在则打开失败”的话，那么使用了O_CREAT|O_EXCL标志的open的执行结果就会正好相反，“文件不存在，则创建文件，否则失败”。

O_EXCL的一个有趣用法是将文件当作锁，下一节将对此做以介绍。另一个用法是针对应用程序退出时要删除的临时文件的。如果应用程序发现临时文件已经存在，那么这意味着前面的调用已经异常终止了，因此需要采取一定的清理或补救措施。这种情况需要创建失败，这正好是O_EXCL参数可以完成的工作：

```

int fd;

while ((fd = open("/tmp/apptemp", O_RDWR | O_CREAT | O_TRUNC | O_EXCL,
    PERM_FILE)) == -1) {
    if (errno == EEXIST) {
        if (cleanup_previous_run())
            continue;
        errno = EEXIST; /* may have been reset */
    }
    EC_FAIL /* some other error or can't cleanup */
}
/* file is open; go ahead with rest of app */

```

在调用open时，我们不想使用ec_negl宏，因为我们想亲自调查errno。值EEXIST是

① 如果你对历史感兴趣，就会知道，creat实际是一个很老的系统调用，在早期的open仅有两个参数时，它是很重要的。

专门针对O_EXCL情况的。调用某个名为cleanup_previous_run的函数（没有给出实现代码）循环检测清理工作是否已经完成，这就是存在while循环的原因。如果没有完成，注意我们重置了errno选项，由于其变化很大，而且可能包含了上千条代码，所以很难知道cleanup_previous_run函数所做的工作。（我们本可以仅使用有两次迭代的for循环来实现，而不使用while，当函数cleanup_previous_run的返回结果为true时，执行循环，但当链接文件失败时，退出循环，因而能够明白程序的执行情况。）

如果有两个不同的用户并发运行上面那个例子所示的应用程序，那么就会出现混乱。这时，创建临时文件的意义很大，解除与临时文件的链接将是错误的。如果允许并发运行应用程序，则需要重新设计程序，以便每个执行过程都有一个唯一的临时文件，2.7节将对此进行介绍。如果想彻底阻止并发运行应用程序，则需要采用加锁机制，下节将对此进行介绍。

2.4.3 用文件当锁

当某些进程需要以独占方式访问系统资源时，将遵循以下协议：在访问资源前，使用一个一致同意的文件命名约定，用O_EXCL创建一个文件。它们中只有一个能成功创建文件，其他进程的open操作都将失败。那些失败进程要么等待随后再次尝试，要么放弃。当成功创建文件的进程用完资源后，将释放链接。其中一个先前没有成功访问的进程将再次进行open操作，并可以安全进行。

为了完成这项工作，检查文件是否存在（例如用access，见3.8.1节）以及创建文件的操作必须是原子操作（不可分），中间不允许插入其他任何进程，或者说只有在第一个进程完成了对文件的检查之后才能创建该文件。因此不能如下这样操作：

```
if (access( ... ) == 0)      /* file does not exist */
    ... open(...) ...      /* create it */
```

而是需要采用更可靠的方式保证原子操作。

一个像这样的简单相互排斥机制称做互斥（mutex）（相互排斥的简称）、二元信号量（计数只能到1）或锁。本书中将多次提到这些内容，摘要内容见1.1.7节。在UNIX领域中，“互斥”一词经常被用于描述线程间的关系，“信号量”一词常被用于UNIX的信号量系统调用中，因此在本节只把这个机制称为“锁”。

最好将这个协议封装成两个函数——lock和unlock，使用方法如下：

```
if (lock("accounts")) {
    ... manipulate accounts ...
    unlock("accounts");
}
else
    ... couldn't obtain lock ...
```

锁名“accounts”是抽象的，它不必与某个实际文件相关。如果两个或多个进程同时执行这段代码，锁将阻止它们同时执行被保护段（无论accounts指的是什么，都“对accounts加锁”）。记住如果进程没有调用lock函数，那么将不对其进行保护。它们是建议性锁，而非强制性锁。（7.11.5节将详细介绍两者之间的区别。）

下面是lock、unlock以及lockpath函数的代码：

```
#define LOCKDIR "/tmp/"
#define MAXTRIES 10
#define NAPLENGTH 2

static char *lockpath(char *name)
{
```

```

static char path[100];
if (snprintf(path, sizeof(path), "%s%s", LOCKDIR, name) > sizeof(path))
    return NULL;
return path;
}

bool lock(char *name)
{
    char *path;
    int fd, tries;

    ec_null( path = lockpath(name) )
    tries = 0;
    while ((fd = open(path, O_WRONLY | O_CREAT | O_EXCL, 0)) == -1 &&
        errno == EEXIST) {
        if (++tries >= MAXTRIES) {
            errno = EAGAIN;
            EC_FAIL
        }
        sleep(NAPLENGTH);
    }
    if (fd == -1)
        EC_FAIL
    ec_negl( close(fd) )
    return(true);

    EC_CLEANUP_BGN
    return false;
    EC_CLEANUP_END
}

bool unlock(char *name)
{
    char *path;

    ec_null( path = lockpath(name) )
    ec_negl( unlink(path) )
    return true;

    EC_CLEANUP_BGN
    return false;
    EC_CLEANUP_END
}

```

lockpath函数生成了一个锁的实际文件名。我们把这个文件名放在/tmp目录下，因为每个UNIX系统都具有该目录，并且任何人都可对该目录进行写操作。注意即便snprintf函数返回的数字过大，也不会溢出所给定的缓冲区，此返回值代表了可能发生的事情。

根据以上章节介绍的内容，我们知道，lock会通过尝试用O_EXCL创建文件的方式来设置锁。我们区别了EEXIST与其他错误，如代码所示。

我们试图进行MAXTRIES次创建文件操作，两次尝试之间的间隔时间为NAPLENGTH秒。（关于sleep，既可以参考标准C，也可以见9.7.2节。）由于对写入文件的实际内容不感兴趣，所以关闭open返回的文件描述符时，只关心文件存在与否，甚至不带任何访问权限创建文件。此外，为增强功能，可以把进程数和时间写入文件，以便其他等待此文件的进程清楚谁正在等待以及工作的时间。如果想实现这项功能，需要对文件具有读权限。

unlock函数所做的工作是删除文件，这样下次尝试创建文件才能成功。2.6节将介绍系统调用unlink。

下面也是一个有趣的小测试程序:

```
void testlock(void)
{
    int i;

    for (i = 1; i <= 4; i++) {
        if (lock("accounts")) {
            printf("Process %ld got the lock\n", (long)getpid());
            sleep(rand() % 5 + 1); /* work on the accounts */
            ec_false( unlock("accounts") )
        }
        else {
            if (errno == EAGAIN) {
                printf("Process %ld tired of waiting\n", (long)getpid());
                ec_reinit(); /* forget this error */
            }
            else
                EC_FAIL /* something serious */
        }
        sleep(rand() % 5 + 5); /* work on something else */
    }
    return;
}
EC_CLEANUP_BGN
    EC_FLUSH("testlock")
EC_CLEANUP_END
}
```

该程序循环4次完成“请求/工作/释放”模式，这里的“工作”指休眠1秒到5秒之间的随机数的时间。如果没有得到锁，它将打印报告并继续尝试创建，接着休眠5秒到9秒之间的某个时间，然后再次循环。printf调用通过系统调用getid（该系统调用的解释见5.13节）得到进程ID。现在立刻运行3次这个小程序：

```
$ tst & tst & tst &
```

输出结果如下：

```
Process 9232 got the lock
Process 9233 got the lock
Process 9234 got the lock
Process 9232 got the lock
Process 9233 got the lock
Process 9232 got the lock
Process 9233 got the lock
Process 9234 got the lock
Process 9232 got the lock
Process 9233 got the lock
Process 9234 got the lock
Process 9234 got the lock
Process 9234 got the lock
```

刚开始时以可预测的方式运行，到第6行以后才变得更有意思。最后，进程9234等待一段时间，并在其他进程返回后才得到锁。

下面讨论用文件当锁的优点和缺点。优点是：它们易于编码生成文件（目前我们仍然位于本书第2章的前几节），作为文件它们可以包含一些数据，而且只要文件需要，就可得到这些数据，当需要永久锁时这是有用的。但最后一点在以下情况中也会出现缺点：如果进程在没有释放锁的情况下中断执行，即使重新启动，也不能释放锁，除非删除/tmp目录。同样，因为创建文件进程执行很慢，即使创建失败，操作量也很大，对于每个应用程序的执行来说，少数几次也许能成功，但对于需要快速锁定的操作，如数据库或实时程序来说就不能满足要求了。

然而我们并没有提起最坏的情况：当进程无法得到锁时，进程将保持休眠状态，并一直尝试得到锁，这种行为称为轮询（polling）。仅仅为了回答一个简单问题：“到我了么？”，CPU就需要做很多的工作。然而当进程正在休眠时，可能会有空闲，但在进程醒来之前，是不会发现它的，这是一种浪费。

幸运的是，所有嵌入UNIX内部的、需要锁的设备都使用了阻塞（blocking），阻塞是指进程将一直处于休眠状态，直到所需事件出现。7.11节将对此进行介绍。

2.4.4 open标志一览

除了以上介绍的open标志外，open还有许多标志，在以后适当的章节进行介绍将更加有意义。例如，O_NOCTTY与终端有关，因此将在第4章对其进行介绍。表2-1列出了SUS3^①定义的所有标志，关于它们的详细介绍，可以参考相关章节。

表2-1 open标志

标 志	解 释
O_RDONLY [*]	只读方式打开（见2.4.1节）。
O_WRONLY	只写方式打开（见2.4.1节）。
O_RDWR	读写方式打开（见2.4.1节）。
O_APPEND	每次写都追加到文件的尾端（见2.8节）。
O_CREAT	若此文件不存在则创建文件（见2.4.2节）。
O_DSYNC	设置同步I/O方式（见2.16.3节）。
O_EXCL	如果文件已经存在，则出错；必须与O_CREAT一起使用（见2.4.2节）。
O_NOCTTY	不将此设备作为控制终端（见4.10.1节）。
O_NONBLOCK [†]	不等待命名管道或特殊文件准备好（见4.2.2节和7.2节）。
O_RSYNC	设置同步I/O方式（见2.16.3节）。
O_SYNC	设置同步I/O方式（见2.16.3节）。
O_TRUNC	将其长度截短为0（见2.4.2节）。

^{*}头三个标志中只能使用其中一个。

[†]原来称为O_NDELAY，含义略有不同。

2.5 umask系统调用

在2.4.2节提到了进程文件方式创建屏蔽字。umask系统调用可以对其进行设置，除了umask命令，一般不使用其他命令设置文件方式创建屏蔽字。

umask——设置和得到文件模式的创建掩码

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

mode_t umask(
    mode_t cmask           /* new mask */
);
/* Returns previous mask (no error return) */
```

因为每个进程都对应一个屏蔽字，并且9个权限位的每一种组合都是合法的，所以umask从不返回错误。它总是返回旧屏蔽字。为了找出没有更改的旧屏蔽字，需要调用两次umask：第一次可以使用任何参数得到旧屏蔽字，第二次按照要求重新设置。

① 单一UNIX规范，版本3；见1.5.1节。

2.6 unlink系统调用

unlink —— 删除目录项

```
#include <unistd.h>

int unlink(
    const char *path      /* pathname */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

unlink系统调用能从目录中删除链接，并将信息节点所引用的文件链接数减1。如果链接数减到0，文件系统将删除这个文件，它所占用的磁盘空间可再次被利用（添加到“自由表”中），信息节点也可被重用。但在包含此链接的目录中进程必须具有写权限。

可以解链任何类型的文件（普通文件、套接字、命名管道以及特殊文件等），但只有超级用户才可以解链目录，在某些系统中，即使是超级用户，也不可以解链目录。无论如何，解链目录都应该使用rmdir系统调用（见3.6.3节），而不是unlink。

当链接数为0，而某些进程仍有打开文件时，为了避免中断正在运行的进程，文件系统将延迟删除文件，直到文件关闭。这种特性常用于程序运行时创建的临时文件，程序代码如下：

```
ec_neg1( fd = open("temp", O_RDWR | O_CREAT | O_TRUNC | O_EXCL, 0) )
ec_neg1( unlink("temp") )
```

这种技术有两个优点：第一，如果由于某种原因进程终止，系统将删除文件。例如为了确保文件解链，不必使用atexit（见1.3.4节）注册函数。第二，因为通过unlink可以立即从当前目录删除链接，进一步减少了第二个进程使用同一临时文件以及因为带有O_EXCL标志时open出错的危险。但是如果第二个进程在第一个进程的open和unlink之间执行open，仍有可能出现以上危险。

补救这个问题的一个方法是采用锁（见2.4.3节）：

```
ec_false( lock("opentemp") )
ec_neg1( fd = open("temp", O_RDWR | O_CREAT | O_TRUNC | O_EXCL, 0) )
ec_neg1( unlink("temp") )
ec_false( unlock("opentemp") )
```

补救方法虽然很完美，但仍有一个缺点：锁仅是建议性的，并且临时文件的命名相当不规范。因此，很可能有另一个进程（没有使用锁）使用了同一个名字，而其中一个进程将不能得到它的临时文件（因为带有O_EXCL标志的open出错）。更好的补救方法是使临时文件名唯一，但是需要一点技巧，2.7节将对临时文件做更加详细的介绍。

也许你想到了另一个问题：因为文件名总是temp，两个进程同时对同一个临时文件进行读写，不会造成混乱吗？回答是：不会。如果我们今天命名了一个名叫myfile的文件，并删除了它，明天创建相同文件名的文件也不会指向同一个文件，因为第一个进程使用的信息节点与第二个进程将使用的信息节点是截然不同的，所以即便第一个进程（数据完整无缺）仍然有打开的文件也没关系。可以按以下方式考虑：即使文件仍然是打开的，unlink也删除了目录入口项，结果信息节点变成了匿名的，因此与新进程的访问完全隔离了。

2.7 创建临时文件

在前一节中，创建临时文件的方法是使用固定的命名（temp），并使用锁防止两个进程同时执行相同的代码，这种方法很笨拙，因此在UNIX中更常用的方法是通过确保命名唯一来

避免产生冲突。标准C函数`tmpnam`似乎能够满足这个要求:

```
char *pathname;

ec_null( pathname = tmpnam(NULL) )
ec_negl( fd = open(pathname, O_RDWR | O_CREAT | O_TRUNC | O_EXCL, 0) )
ec_negl( unlink(pathname) )
```

因为确信没有此名字的文件存在,所以能确保`tmpnam`的返回值是唯一的。但是只有执行了`open`,才能创建该文件,因此有较少可能是另一个同时执行`tmpnam`的进程创建了具有相同名字的文件。(必须清楚地认识到,两行代码之间的执行是任意的,其他进程也可以执行`tmpnam`。)因为`O_EXCL`标志,其中一个不能创建和打开,因此不存在I/O混淆的危险,而且也比使用固定命名好。虽然冲突的可能性降低了,但还不够,因为没有降到0。

下面是解决该问题的方法:

mkstemp——用唯一的名字建立和打开文件

```
#include <stdlib.h>

int mkstemp(
    char *template /* template for file name */
);
/* Returns open file descriptor or -1 on error (may not set errno) */
```

`mkstemp`绝对确保了所创建的文件具有独一无二的命名,没有竞态条件问题。给定一个以6个X结尾的名称模板,只要能使命名唯一,6个X可替换成任何字母组合。`mkstemp`比`tmpnam`的作用更强,实际上,它创建和打开的文件都具有读写权限。尽管大多数实现可能仅允许所有者(`S_IRUSR|S_IWUSR`)读写,但因为标准(SUS3)没有规定,所以不能假定使用的权限。

对于一个可移植程序,如果`mkstemp`返回-1,系统将不知道如何处理`errno`。标准没有规定任何错误代码,但是可以肯定所有的工具都将返回一个有效的`errno`。因此可以选择使用`ec_negl`宏(回忆可知,它记录了`errno`),即使标准没有设置`errno`。如果没有为错误设置`errno`,那么为了避免产生误导错误消息,在调用函数之前,应尽量将其设置为0。

`mkstemp`存在于SUS1、Linux、FreeBSD和Darwin(其起源于BSD)中,因此几乎可以随处使用。

下面举例说明;为了说明问题,我们输出文件名:

```
char pathname[] = "/tmp/dataXXXXXX";

errno = 0; /* mkstemp may not set it on error */
ec_negl( fd = mkstemp(pathname) )
ec_negl( unlink(pathname) )
printf("%s\n", pathname);
```

输出结果:

```
/tmp/dataKdBy0u
```

用户不必立刻解链文件,但必须安排以后解除(如利用已注册的`atexit`函数),否则将被遗忘。当然,如果需要向程序其他部分或外部程序传递路径名,则不能立即解链,如下例所示:

```
int status;
char cmd[100];

ec_negl( fd = mkstemp(pathname) )
/* code to write text lines to fd (not shown) */
snprintf(cmd, sizeof(cmd), "sort %s", pathname);
ec_negl( status = system(cmd) )
```

将解释语句 (“code to write...”) 替换成写这个文件的代码。system是调用外部程序的标准C函数；第5章将对其进行介绍。

顺便说一下，你可能想要查找一个名为tmpfile的标准C函数，它的工作过程与mkstemp相同，但是它返回的是FILE指针，而不是文件描述符。

一个听得更多的函数是mktemp，此函数与tmpnam的功能接近程度高于与mkstemp的接近程度，因为该函数返回结果为文件名，而不创建文件。此函数与tmpnam一样具有很多问题，并且不包含在标准C中，因此不要使用它。

小结：

好的函数：mkstemp和tmpfile。

差的函数：mktemp和tmpnam。

2.8 文件偏移量和O_APPEND

本节将介绍O_APPEND标志，这个标志首先出现在2.4.4节，然后介绍read、write、lseek、pread和pwrite系统调用的一些特性，对这些函数的详细介绍见以后章节。

对于普通文件来说，文件偏移量标识的是下一次读或写文件的位置。这是文件偏移量的唯一目的。其他类型的文件，包括目录、套接字、命名管道和符号链接都没有文件偏移量。特殊文件是否有文件偏移量，要取决于它们的实现（见3.2节）。

在UNIX出现之前（提示：甲壳虫乐队还没有解散），大部分操作系统都包含“顺序数据文件”和“随机数据文件”。UNIX只有一种数据文件类型，通过可移动的文件偏移量控制随机访问。即使今天看起来很平常了，但那时是很重大的改革。

如图2-1所示，每次打开一个文件时，因为能得到一个新的打开文件描述，所以可以得到一个独立的文件偏移量。这意味着在下面的例子中：

```
int fd1, fd2, fd3;

ec_neg1( fd1 = open("myfile", O_WRONLY | O_CREAT | O_TRUNC, PERM_FILE) )
ec_neg1( fd2 = open("myfile", O_RDONLY) )
ec_neg1( fd3 = open("yourfile", O_RDWR | O_CREAT | O_TRUNC, PERM_FILE) )
```

fd1和fd2拥有各自的文件偏移量，因此对fd1的写以及对fd2的读是独立的，但对fd3文件的读写只有一个文件偏移量。

在缺少O_APPEND标志的情况下，对于新打开的文件，其文件偏移量为0，读操作或写操作会自动获取该偏移量，并通过read或write的大小自动完成读或写操作。因此，除非有意改变了文件的偏移量，否则read和write操作是有顺序的。用户可以读一些数据，接着再用read读下面的数据，依此类推。同样，可以实现write操作。

假设文件偏移量从0开始，如果往fd1中写入100个字节，接着从fd2中读出100个字节，那么读出的字节会是刚刚写入的那100个字节。但是如果往fd3中写入，接着从中读出，那么得到的将是写入数据之后的内容，如果没有数据，将返回文件结束标志，因为read和write使用的是同一个文件描述而每个文件描述又只有一个唯一的偏移量。

通过lseek（见2.13节）可以找出文件偏移量在文件描述符中的位置，并且（或者）为其设置新值。此新偏移量值将影响下一次在此文件描述符中的读或写。

在第6章中，我们将讲解如何复制一个打开文件描述符，这里所说的“复制”不是像

```
fd1=fd2;
```

这样的拷贝，而是采用系统调用如dup实现的复制。但无论怎样，最重要的一点是一个文件描述符是另一个的复制品，因为共享同一个打开文件描述，所以两者共享同一个文件偏移量。

当文件以O_APPEND标志打开时，所有通过write系统调用进行的写操作，都将通过隐含的lseek将文件偏移量设置在文件的结尾，因此写操作都自动在文件结尾进行。即使几个进程同时对用O_APPEND标志打开的文件进行写操作，每个写操作也都会立即在文件结尾进行，而且彼此之间也不会覆盖或者混淆它们的数据。但用户无法用跟有写操作的lseek来实现同样的功能（不设置O_APPEND标志），因为如我们所看到的其他情况一样，两个系统调用之间有间隙，这个间隙能导致如下后果：

- 1) 进程A将文件偏移量定位在文件结尾（假定位置在1000）。
- 2) 进程B将文件偏移量也定位在文件结尾（位置同样是1000）。
- 3) 进程B写入200字节（位置是1000）。
- 4) 进程A写入200字节（在位置1000，覆盖了进程B写入的内容）。哎！

可以通过锁（见2.4.3节）补救该问题，但解决此问题更好的方法是：当进程A和进程B打开文件时，如果设置了O_APPEND标志，则必须保证可以完成如下功能：

- 1) 进程A将文件偏移量定位在文件结尾（假定位置是1000），并进行写操作。
- 2) 进程B将文件偏移量定位在文件结尾（位置是1200），并进行写操作。

因此当需要从几个进程累加输出量时，对日志文件或其他情况而言，通过设置O_APPEND标志来实现是很合适的。

也可以仅仅通过规定系统调用本身的位置来实现文件的读写，不用先调用lseek；pread和pwrite就可以实现此功能（见2.14节）。这两个函数不使用文件偏移量，也不改变它的位置。

现在对读写操作已经有了很好的了解，如果喜欢的话，可以跳到2.13节了解lseek的功能以及相应的示例，然后返回来继续2.9节的内容。

2.9 write系统调用

write——向文件描述符写

```
#include <unistd.h>

ssize_t write(
    int fd,           /* file descriptor */
    const void *buf, /* data to write */
    size_t nbytes    /* amount to write */
);
/* Returns number of bytes written or -1 on error (sets errno) */
```

前面已经多次提到write，现在到了对其进行全面介绍的时候了。

write将buf所指缓冲区的n字节写入fd所描述的打开文件中。写操作从文件偏移量的当前位置开始执行，并且在完成之后，文件偏移量将增加所写入的字节。若写入成功，返回值为已写的字节数，出错则为-1。

前面说到，如果设置了O_APPEND标志，写之前文件偏移量会自动定位到文件的结尾。

write也用于向管道、特殊文件和套接字写入数据，但是在这些情况中语义略有不同。一个重要的差别是，这些写操作可以阻塞，这意味着它们正在等待某个不可预知的事件，例如等待可用数据。如果阻塞了写操作，那么到达的信号会中断其操作（见9.14节），在这种情

况下写操作将返回值-1，并将errno设置成EINTR。现在只介绍普通文件的写操作，对于其他文件类型的写操作，将在第4、6、8章进行。

write给人的假象是很简单。它似乎只是写入数据，接着返回结果，但一个小小的实验就可以使人相信实际并非如此，即没那么快。那是骗人的！

实际上，那确实是骗人的。当用户调用write系统调用时，并不执行写操作，接着返回数据，它仅仅是将数据传递给内核中的缓冲区，除了声明如下内容之外，不做其他事情：

我注意到了你的请求，接下来会保证你的文件描述符可以使用。我已经成功地复制了你的数据，磁盘空间是充足的。以后，我方便的时候，如果我仍是激活的，我会设法把你的数据放到磁盘中。如果发现错误，我会设法在控制台输出错误，但我不会告诉你这些的（实际上，你那时可能已经终止运行了）。如果在我写出这些数据之前，你或者其他进程试图读这些数据，那么我将从缓冲区为你读这些数据，因此，如果一切顺利，你不会知道我什么时候完成的请求，也不会知道我是否完成了你的请求。你可以进一步提出要求。相信我并感谢我的快速回答吧——我认为那正是你所关心的。

如果一切顺利，那么延迟写就是假想的。语义和实际发生写操作时相同，但非常快。然而如果有磁盘错误，或者由于某种原因内核停止了，那么就全完了。你会发现要写的数据根本没有写到磁盘上。

除了不能确定什么时候发生物理写操作之外，对于延迟写还有其他两个问题。第一个问题：一个调用写操作的进程没有得到写错误的通知。实际上，文件系统缓存并不被任一单个进程所有，如果多个进程同时向同一文件的同一块写数据，数据将被传送到相同的缓存。当然，我们可以构思一个方案，其中“写错误”信号可以发送到每一个向特定缓存写数据的进程，在较短的时间，应当处置它的进程在做什么呢？并且内核如何通知已经终止了的进程呢？

第二个问题是物理写操作的顺序是无法控制的。顺序经常引发问题。例如，当在文件中更新链表结构时，较好的方法是写一个新记录，然后更新那个指向它的指针，相反则不好，因为没有指针指向的记录比不指向任何内容的指针产生的问题少。即使write系统调用是按照某种特定顺序调用的，也不能保证缓冲区的数据能按此顺序写入磁盘，因此除了上述例子之外，小心的替换（careful replacement）技术也不能像其设定的那样起作用，而只能保证进程不会在不合适的位置中断，但它们不能保证不出现磁盘错误和内核崩溃。

幸运的是，可以采用强制同步写方式，将在2.16节介绍相关内容。

这些与write相关的问题不必被过分强调。现代计算机的可靠性很高，而且UNIX实现的可靠性通常也很高，因此出现内核崩溃的几率很少。大多数用户只会发现缓冲区的速度之快，永远不会发现内核错误。

现在，我们再次看一下本章开始时的文件复制的例子。核对写错误的缺陷是：

```
if (write(tofd, buf, nread) != nread)
    EC_FAIL
```

如果write返回的计数比请求的计数少，那并不算是错误。计数少也许是因为磁巧管道已满，或者是达到了普通文件规定的极限。下一次调用写操作时将产生错误。[⊖]

因此缺陷是EC_FAIL宏记录了一个没有任何意义的errno，这个值仅在返回值为-1时设定。可以重新编写这个函数，接受部分写操作，并保持尝试直到真正的错误发生：

[⊖] 就算当时能发现什么原因造成了计数的减少，例如是空间的短缺，并能在下次调用write之前纠正错误，不出现返回错误，但是也没有100%可靠的方法可以查出为什么部分write或read的计数少了。

```

#define BUFSIZE 512

void copy2(char *from, char *to)
{
    int fromfd = -1, tofd = -1;
    ssize_t nread, nwrite, n;
    char buf[BUFSIZE];

    ec_negl( fromfd = open(from, O_RDONLY) )
    ec_negl( tofd = open(to, O_WRONLY | O_CREAT | O_TRUNC,
        S_IRUSR | S_IWUSR) )
    while ((nread = read(fromfd, buf, sizeof(buf))) > 0) {
        nwrite = 0;
        do {
            ec_negl( n = write(tofd, &buf[nwrite], nread - nwrite) )
            nwrite += n;
        } while (nwrite < nread);

        if (nread == -1)
            EC_FAIL
        ec_negl( close(fromfd) )
        ec_negl( close(tofd) )
        return;
    }

    EC_CLEANUP_BGN
    (void)close(fromfd); /* can't use ec_negl here! */
    (void)close(tofd);
    EC_CLEANUP_END
}

```

实际上，对于普通文件而言，这段代码未免有些过于麻烦了，尽管我们以后对终端和管道进行I/O操作时要使用的技术与此相似，但终端和管道有时仅需多次尝试，问题便可解决。因此当向普通文件写数据时，下面这个简单方案也许意义更大：

```

if ((nwrite = write(tofd, buf, nread)) != nread) {
    if (nwrite != -1)
        errno = 0;
    EC_FAIL
}

```

或者可能有人喜欢这样写：

```

errno = 0;
ec_false( write(tofd, buf, nread) == nread )

```

这里将errno设为0，以便错误报告显示错误（并显示代码行号），而不是显示容易令人误解的错误代码。因此可以确定不需要做的事情是完全忽略少的计数：

```

ec_negl( write(tofd, buf, nread) ) /* wrong */

```

下面介绍一个方便实用的函数writeall，它封装了前面copy2的例子中用到的“保持尝试”的方法。在本书后面的章节（见4.10.2节和8.5节）中，当需要确保写入所有的内容时，将使用此函数。注意，因为“ec”宏是直接替换写操作，所以不使用它：

```

ssize_t writeall(int fd, const void *buf, size_t nbyte)
{
    ssize_t nwritten = 0, n;

    do {
        if ((n = write(fd, &((const char *)buf)[nwritten],
            nbyte - nwritten)) == -1) {

```

```

        if (errno == EINTR)
            continue;
        else
            return -1;
    }
    nwritten += n;
} while (nwritten < nbytes);
return nwritten;
}

```

因为EINTR错误实际上不是错误，因此对它进行了特殊处理。EINTR错误只是说明了在写入过程中，某个信号中断了写操作，因此可以忽略它，而继续进行写操作。9.1.4节对信号和如何处理中断系统调用做了详细的讲解。下一节有一个类似的readall。

2.10 read系统调用

read——从文件描述符中读入

```

#include <unistd.h>

ssize_t read(
    int fd,           /* file descriptor */
    void *buf,       /* address to receive data */
    size_t nbytes    /* amount to read */
);
/* Returns number of bytes read or -1 on error (sets errno) */

```

read系统调用与write相反，它是从fd所描述的打开文件中读取buf所指缓冲区中的n字节。read从当前文件偏移量开始读数据，并且完成读操作后，文件偏移量将增加所读字节数。read的返回值是所读字节数、文件结束标志0或者错误标志-1。读操作不受O_APPEND标志的影响。

与write不同，read系统调用不会轻易被传递数据和随后读取的数据欺骗。如果数据已经不在缓冲区中（由于以前的I/O操作），进程必须等待内核从磁盘得到数据。有时，当内核注意到访问方式暗示是从连续磁盘块中顺序读取数据时，它会设法加快读取速度，提前读取所需要的数据。如果系统能够轻易地加载足够的的数据，而且能在缓冲区保持一会，并且读操作是顺序的，则提前读是很有效的。

与部分写一样，也存在部分读的问题：因为返回的计数值少于所读字节不是错误，所以errno是无效的，用户必须推测问题所在。如果需要读所有的数据，最好是通过循环调用read，如readall函数所实现的功能（可以与上一节的writeall函数做比较）：

```

ssize_t readall(int fd, void *buf, size_t nbytes)
{
    ssize_t nread = 0, n;

    do {
        if ((n = read(fd, &((char *)buf)[nread], nbytes - nread)) == -1) {
            if (errno == EINTR)
                continue;
            else
                return -1;
        }
        if (n == 0)
            return nread;
        nread += n;
    } while (nread < nbytes);
}

```

```

    return nread;
}

```

在8.5节将讲述readall函数的使用。

与write一样，从管道、特殊文件或套接字中读数据时read可以阻塞，这种情况下读操作可能会被信号中断（见9.1.4节），结果返回值-1，并把errno设置成EINTR。

2.11 close系统调用

close——关闭文件描述符

```

#include <unistd.h>

int close(
    int fd          /* file descriptor */
);
/* Returns 0 on success or -1 on error (sets errno) */

```

通过对close进行分析，应该明白最重要的一点是它没有做任何工作。它没有刷新任何内核缓冲区，而仅仅使文件描述符可重用。当指向一个打开文件描述的最后文件描述符关闭时（见2.2.3节），也将删除打开文件描述。同样，相应地，当删除指向内存信息节点的最后一个打开文件描述时，也将删除内存信息节点。进一步说：如果移去了实际信息节点的所有链接，系统将删除磁盘上的信息节点和它的所有数据（见2.6节）。

因为close不刷新缓存，也不会加速刷新，所以从写入了数据的文件中读数据前不需要关闭文件，它可以保证用户能读出所写入的数据。也可以这样理解，内核缓冲决不会影响read、write、lseek或者其他系统调用的语义。

实际上，如果不再需要文件描述符，根本没有必要调用close，因为进程终止时将收回文件描述符。然而，最好的办法是释放内核结构，并且向程序的读取进程表明用户已经完成了对文件的操作。如果时常检查错误，则可以避免偶尔使用文件描述符带来的问题。

调用close，对管道和其他的不规则文件也会带来负面作用，当讨论这些文件类型时，还会详细讨论这些问题。

2.12 用户缓冲I/O

到目前为止，我们已经讨论了内核缓冲区，通过它可以在快速内存中实现预先读、延迟写以及保持频繁的数据访问。本节还涉及了一个类型截然不同的缓存，用户执行进程使用的缓存，该缓存与内核缓存根本无关。为了与内核缓存区分，称这种缓存为用户缓存。

2.12.1 用户缓存与内核缓存

第1章已经介绍，UNIX文件系统是建立在块特殊文件之上的，因此所有的内核I/O操作和所有的缓冲操作都是以块为单位的。块的大小可以是任何数字，但一般是512字节的倍数，1024、2048和4096都是常用的数字。所有设备的块尺寸可以不同，并可以根据磁盘分区的大小而改变。下面将进行简要的说明。

读写大的数据块比小的数据块要快。为了演示效果，现在重新编译2.9节的copy2程序，通过如下改动将用户缓存大小变为1字节：

```
#define BUFSIZE 1
```

在表2-2中列出了两个版本下复制4MB文件^①所使用的时间（时间以秒为单位）。

表2-2 以块为单位的I/O和以字符为单位的时间

方法	用户	系统	总计
512字节缓存	0.07	0.58	0.65
1字节缓存	18.43	204.98	223.41

（这些时间是在Linux系统上运行的结果；在FreeBSD系统中具有相同的结果）

用户时间是用户进程执行指令所使用的时间。系统时间是进程在内核中执行指令所使用的时间。

对于普通文件来说，这种小的缓存所造成的I/O性能恶化是极其剧烈的，所以根本没有人这样做，除非程序是为了临时、意外或者是在极不寻常的情况下使用（例如向后读文件，见2.13节）。缓存大时也会出现性能恶化的系统调用（当缓存是512字节时）。为了检查I/O缓存选择不当造成的性能恶化，选择了两个不同的BUFSIZE进行比较，1024字节的缓存，效果较好；1100字节的缓存，虽然数值较大，但效果不好。在Linux系统中，所用时间差别较少，但1100字节缓存所用时间仍高75%（总时间是7.4秒对4.25秒）。在FreeBSD和Solaris系统中，相差更少，仅10%~20%左右。

但是问题是在实际系统中使用与程序所需大小相符合的块字节是很难的，程序行数的变化以及多样化的结构是很普遍的。因此解决的方法是在用户空间中将临时数据打包成块，当用户空间满时才写入块中。输入数据时采用相反操作：读数据的过程中将块解包。那就是说，除了使用内核缓存外，还要使用用户缓存。因为一个数据片可能会跨越一个块，所以会给编程带来困难，本书将在下节介绍如何处理这个问题。

首先，一个恼人的问题：如何知道块的大小呢？答案是用户可以使用打算要容纳所要处理的文件的文件系统的实际大小作为块的大小，^②但一些实验显示，合理的情况下与文件大小相比，较大的缓存比较小的缓存效果好。原因如下：如果缓存比实际的块小，但均匀分割文件后，内核就可以非常有效地将数据打包到缓存，并且当缓存满时，可以对被写的缓存进行协调。如果数值较大并且是缓存的整数倍，那么对内核来说将数据装入缓存仍是非常有效的，并且使用write系统调用的次数也会较少，一般write是影响速度的决定性因素。read的情况也一样。

因此，目前最有效的做法是使用标准C规定的宏BUFSIZ，这个宏是标准I/O函数（如fputs函数）使用的。但是由于它是常数，所以对于实际的文件系统来说它不是最佳的，但实验效果显示影响不大。如果空间不足，用户甚至可以使用512字节的，感觉也还可以。

2.12.2 用户缓存函数

在任何函数调用单元，只要需要都可以很方便地使用读、写以及查找函数等。这些子程序自动控制缓存，从不偏离块模型。一个特别好的这种程序包是“标准I/O库”，这个标准库在很多C语言书中都有讲述，例如[Har2002]。

为了显示用户缓存包的规则，下面举个简化的例子，这个例子的名字叫BUFIO，它支持单个字符的读和写，但不支持查找。首先是包的用户必须包含头文件bufio.h（原型未显示）：

```
typedef struct {
    int fd; /* file descriptor */
```

① 本书的1985版本使用了一个4000字节的文件，并用了差不多的时间！

② 用户可以使用stat系统调用（见3.5.1节）。

```

char dir; /* direction: r or w */
ssize_t total; /* total chars in buf */
ssize_t next; /* next char in buf */
unsigned char buf[BUFSIZ]; /* buffer */
} BUFIO;

```

下面是包的实现 (bufio.c):

```

BUFIO *Bopen(const char *path, const char *dir)
{
    BUFIO *b = NULL;
    int flags;

    switch (dir[0]) {
        case 'r':
            flags = O_RDONLY;
            break;
        case 'w':
            flags = O_WRONLY | O_CREAT | O_TRUNC;
            break;
        default:
            errno = EINVAL;
            EC_FAIL
    }
    ec_null( b = calloc(1, sizeof(BUFIO)) )
    ec_neg1( b->fd = open(path, flags, PERM_FILE) )
    b->dir = dir[0];
    return b;

EC_CLEANUP_BGN
    free(b);
    return NULL;
EC_CLEANUP_END
}

static bool readbuf(BUFIO *b)
{
    ec_neg1( b->total = read(b->fd, b->buf, sizeof(b->buf)) )
    if (b->total == 0) {
        errno = 0;
        return false;
    }
    b->next = 0;
    return true ;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

static bool writebuf(BUFIO *b)
{
    ssize_t n, total;

    total = 0;
    while (total < b->next) {
        ec_neg1( n = write(b->fd, &b->buf[total], b->next - total) )
        total += n;
    }
    b->next = 0;
    return true ;
}

```



```

    EC_CLEANUP_BGN
        return false;
    EC_CLEANUP_END
}

int Bgetc(BUFIO *b)
{
    if (b->next >= b->total)
        if (!readbuf(b)) {
            if (errno == 0)
                return -1;
            EC_FAIL
        }
    return b->buf[b->next++];

    EC_CLEANUP_BGN
        return -1;
    EC_CLEANUP_END
}

bool Bputc(BUFIO *b, int c)
{
    b->buf[b->next++] = c;
    if (b->next >= sizeof(b->buf))
        ec_false( writebuf(b) )
    return true;

    EC_CLEANUP_BGN
        return false;
    EC_CLEANUP_END
}

bool Bclose(BUFIO *b)
{
    if (b != NULL) {
        if (b->dir == 'w')
            ec_false( writebuf(b) )
        ec_neg1( close(b->fd) )
        free(b);
    }
    return true;

    EC_CLEANUP_BGN
        return false;
    EC_CLEANUP_END
}

```

最后使用新包重新编写文件复制函数:

```

#include "bufio.h"

bool copy3(char *from, char *to)
{
    BUFIO *stfrom, *stto;
    int c;

    ec_null( stfrom = Bopen(from, "r" ) )
    ec_null( stto = Bopen(to, "w" ) )
    while ((c = Bgetc(stfrom)) != -1)
        ec_false( Bputc(stto, c) )
    if (errno != 0)

```



```

        EC_FAIL
        ec_false( Bclose(stfrom) )
        ec_false( Bclose(stto) )
        return true;

EC_CLEANUP_BGN
    (void)Bclose(stfrom);
    (void)Bclose(stto);
    return false;
EC_CLEANUP_END
}

```

可以注意到BUFIO和标准I/O库的子集非常相似。下面是使用库函数的文件复制的版本:

```

bool copy4(char *from, char *to)
{
    FILE *stfrom, *stto;
    int c;

    ec_null( stfrom = fopen(from, "r") )
    ec_null( stto = fopen(to, "w") )
    while ((c = getc(stfrom)) != EOF)
        ec_eof( putc(c, stto) )
    ec_false( !ferror(stfrom) )
    ec_eof( fclose(stfrom) )
    ec_eof( fclose(stto) )
    return true;

EC_CLEANUP_BGN
    (void)fclose(stfrom);
    (void)fclose(stto);
    return false;
EC_CLEANUP_END
}

```

为了显示使用用户缓存后的优点,表2-3以秒为单位列出了分别使用BUFIO、标准I/O库以及直接的系统调用方法(见copy2)进行文件复制的时间。

表2-3 缓存和非缓存I/O 的比较

方 法	用 户*	系 统	总 计
BUFIO			
Solaris	1.00	0.51	1.51
Linux	1.00	0.28	1.28
FreeBSD	1.00	0.45	1.45
标准I/O			
Solaris	0.57	0.24	0.81
Linux	11.32	0.15	11.48
FreeBSD	1.02	0.20	1.22
BUFSIZ缓存			
Solaris	0.00	0.52	0.52
Linux	0.00	0.23	0.23
FreeBSD	0.01	0.37	0.38

*所有时间以秒为单位,并且在系统中进行归一化处理,以便系统上用户BUFIO时间是1.00。

通过使用用户缓存,几乎可以得到最完美的结果:可以随意处理数据,哪怕一次只有1字

节，其系统时间和使用BUFSIZ缓存方法的大致相同。因此用户缓存无疑是最好的方法。除了Linux系统，标准I/O的时间是最好的；除了速度较快和更加灵活外，它实现了BUFIO函数的功能。标准I/O用户时间在Linux系统中是11.32秒，此值非常突出。一些研究表明，当在所有实验中都使用gcc编译器时，Linux系统使用的是stdio.h中的gcc版本，FreeBSD使用的是BSD上的版本，Solaris使用的是系统V上的版本。看起来需要注意的是gcc版本。^①

标准I/O库的广泛接收是可笑的，尽管在任何单元对普通文件进行I/O操作已经成为UNIX内核的显著特征，然而实际上，这个特征通常效率太低无法使用。

2.13 lseek系统调用

`lseek`^②仅仅用于设置文件偏移量，这个偏移量是接下来进行`read`、`write`或者`lseek`所使用的。实际上没有执行I/O操作，并且没有命令发往磁盘控制器（记住，无论如何通常都有一个高速缓存）。

`lseek`— 设置和得到文件偏移量

```
#include <unistd.h>

off_t lseek(
    int fd,           /* file descriptor */
    off_t pos,       /* position */
    int whence       /* interpretation */
);
/* Returns new file offset or -1 on error (sets errno) */
```

参数`whence`可以取下面的任意一个值：

`SEEK_SET` 将该文件偏移量设置到`pos`参数。

`SEEK_CUR` 将该文件偏移量设置为其当前值加`pos`参数，`pos`可为正数、负数或0，0是查找当前文件偏移量的方法。

`SEEK_END` 将该文件偏移量设置为文件长度加`pos`参数，`pos`可为正数、负数或0，0是将文件偏移量设置为文件结尾的方法。

通常文件偏移量的返回值是非负整数，甚至比文件大。如果比文件大，下一次写操作时将把文件延长到需要的大小，并将中间的字节填充为0。文件偏移量位于或超过结尾时，`read`会产生0（文件结尾）返回。对于`write`操作超过结尾而延长的文件，如果`read`成功，则会返回0字节。

当`write`超过了文件的结尾时，大部分UNIX操作系统实际上并不存储其间填充的都是0的块。因此对于磁盘可能造成这样的结果，可以容纳3 000 000块的磁盘，其实际容纳的字节超过了3 000 000块。当对文件进行备份然后恢复时，这种情况可能会产生严重的问题；当必须读取文件并将其传输到备份设备时，所写出和写回的数据会超过3 000 000块！创建许多带有空穴文件的用户经常能从管理员那儿听到类似事件，除非备份系统有能力识别这些空穴。

使用`lseek`的方式很多，但最常用的是以下三种，第一种，可以用`lseek`查找文件的某个绝对位置：

① 这个问题是在每次`putc`时都要检查线程的锁，但是其他系统能智能地发现不是多线程的。到用户读这些的时候，这个问题也许已经解决。

② 在C语言以前，称之为“seek”，并且为`long`数据类型（这可以向后追忆），并且为了得到超过65 535的字节，需要先用一个`seek`函数定位块，然后用第二个`seek`函数定位块中的字节。新系统调用多了一个额外的字母，如`creat`是少了一个字母。

```
ec_negl( lseek(fd, offset, SEEK_SET) )
```

第二种，可以用lseek查找文件的结尾：

```
ec_negl( lseek(fd, 0, SEEK_END) )
```

第三种，可以用lseek查找文件偏移量的当前位置：

```
off_t where;
ec_negl( where = lseek(fd, 0, SEEK_CUR) )
```

其他方式很少应用。

据2.8节的介绍，可以知道，内核的大部分查找工作都是隐含的，不是显式调用lseek实现的。当调用open时，内核查找第一个字节。当调用read或write时，内核将文件偏移量增加读或写的字节数。当采用O_APPEND标志打开文件时，每次写之前都要查找文件的结尾。

为了解释lseek的使用方法，下面给出一个backward函数，它实现了按逆序输出文件的功能，每次一行。例如，如果文件包含以下内容：

```
dog
bites
man
```

那么backward函数将输出以下内容：

```
man
bites
dog
```

以下是函数代码：

```
void backward(char *path)
{
    char s[256], c;
    int i, fd;
    off_t where;

    ec_negl( fd = open(path, O_RDONLY) )
    ec_negl( where = lseek(fd, 1, SEEK_END) )
    i = sizeof(s) - 1;
    s[i] = '\0';
    do {
        ec_negl( where = lseek(fd, -2, SEEK_CUR) )
        switch (read(fd, &c, 1)) {
            case 1:
                if (c == '\n') {
                    printf("%s", &s[i]);
                    i = sizeof(s) - 1;
                }
                if (i <= 0) {
                    errno = E2BIG;
                    EC_FAIL
                }
                s[--i] = c;
                break;
            case -1:
                EC_FAIL
                break;
            default: /* impossible */
                errno = 0;
                EC_FAIL
        }
    } while (where > 0);
}
```



```

printf("%s", &s[i]);
ec_negl( close(fd) )
return;

EC_CLEANUP_BGN
    EC_FLUSH("backward");
EC_CLEANUP_END
)

```

在此函数中需要注意两个棘手的问题：第一，因为read是隐含地向前查找文件，为了读前面的字节，必须向后查找2个字节，为了读文件，必须将文件偏移量设置在文件结尾的最后一个字节；第二，在read中，因为有文件结束返回值，没有文件开始返回值，所以必须观察文件偏移量（变量where），并且在读第一个字节后停止。换句话说，要等lseek将文件指针变为负数。但是这种做法是不明智的：因为错误代码（EINVAL）也能用于指示其他无效参数的情况，而且一般原则是，最好不使用错误返回弥补算法的不足。

2.14 pread和pwrite系统调用

pread —— 在偏移处从文件描述符读

```

#include <unistd.h>

ssize_t pread(
    int fd,                /* file descriptor */
    void *buf,            /* address to receive data */
    size_t nbytes,        /* amount to read */
    off_t offset           /* where to read */
);
/* Returns number of bytes read or -1 on error (sets errno) */

```

pwrite —— 在偏移处向文件描述符写

```

#include <unistd.h>

ssize_t pwrite(
    int fd,                /* file descriptor */
    const void *buf,       /* data to write */
    size_t nbytes,        /* amount to write */
    off_t offset           /* where to write */
);
/* Returns number of bytes written or -1 on error (sets errno) */

```

pread和pwrite与在lseek之前调用read和write的功能相似，除了以下情况：

- 没有使用文件偏移量，因为读和写文件的位置是由offset参数明确给定的。
- 没有设置文件偏移量，实际上是完全忽略了此参数。

O_APPEND标志（见2.8节）确实影响pwrite，使其行为与write完全相同（根据以上所说可知，此时该函数忽略了offset参数）。

调用一次肯定比调用两次方便，但是更重要的是，pread和pwrite避免了文件偏移量改变的问题，这个改变可能是另一个进程或线程在lseek和read或write之间更改的。回顾可知这种情况是存在的，如线程可能会使用同一个文件描述符，以及进程可能复制文件偏移量一样，这些内容曾在2.2.3节中介绍过。这也是O_APPEND标志所避免的问题（见2.8节）。因为pread和pwrite不会使用文件偏移量，所以它们不会出现文件位置的错误。

为了说明pread如何工作，下面给出backward函数（上节介绍的函数）的另一个版本。这个版本更加简明（写和调试都很容易），因为去掉了调用lseek减少文件偏移量的代码。

```

void backward2(char *path)
{
    char s[256], c;
    int i, fd;
    off_t file_size, where;

    ec_negl( fd = open(path, O_RDONLY) )
    ec_negl( file_size = lseek(fd, 0, SEEK_END) )
    i = sizeof(s) - 1;
    s[i] = '\0';
    for (where = file_size - 1; where >= 0; where--)
        switch (pread(fd, &c, 1, where)) {
            case 1:
                if (c == '\n') {
                    printf("%s", &s[i]);
                    i = sizeof(s) - 1;
                }
                if (i <= 0) {
                    errno = E2BIG;
                    EC_FAIL
                }
                s[--i] = c;
                break;
            case -1:
                EC_FAIL
                break;
            default: /* impossible */
                errno = 0;
                EC_FAIL
        }
    printf("%s", &s[i]);
    ec_negl( close(fd) )
    return;
}
EC_CLEANUP_BGN
    EC_FLUSH("backward2");
EC_CLEANUP_END
}

```

2.15 readv和writev系统调用

readv ——非连续的读

```

#include <sys/uio.h>

ssize_t readv(
    int fd, /* file descriptor */
    const struct iovec *iov, /* vector of data buffers */
    int iovcnt /* number of elements */
);
/* Returns number of bytes read or -1 on error (sets errno) */

```

writev ——非连续的写入

```

#include <sys/uio.h>

ssize_t writev(
    int fd, /* file descriptor */
    const struct iovec *iov, /* vector of data buffers */
    int iovcnt /* number of elements */
);
/* Returns number of bytes written or -1 on error (sets errno) */

```

除了不使用数据地址的情况之外，`readv`和`writenv`与`read`和`write`相似，它们可以同时从多个内存地中读取数据，也可以将数据同时写入到多个内存地址中，有时称它们为散布读和聚集写。数据在文件、管道、套接字或者其他任何目的打开的fd中仍是连续的，即它可以分散的进程内存。

换句话说，如果有三个结构数据要写，那么无需使用三次`write`来完成，而只需使用一次`writenv`就可以完成；然而，看一下怪异的第二个参数，你很快就能看出，这两个函数不太好用，因此最好在必要时才用它们，对吗？稍后将研究这个问题，这里首先说明如何使用它们。

在调用这两个函数之前，必须设置`iovec`数组（具有`iovcnt`个元素），以便每一个元素都包含一个指向数据和大小的指针，实际上相当于`read`或`write`调用的第二个和第三个参数。可以把`struct iovec`定义成这样（也可能有额外的、非标准的成员）：

struct iovec —— `readv`和`writenv`的结构[⊖]

```
struct iovec {
    void *iov_base;    /* base address of data */
    size_t iov_len;   /* size of this piece */
};
```

在程序中，只要有足够的内存，并且正确进行了初始化，用户就可以声明一个`struct iovec`类型的数组，或者使用`malloc`函数分配内存，或者做其他任何喜欢的事情。数组元素个数的最大值与所用系统有关，但是至少是16。在SUS系统上，如果定义了`IOV_MAX`符号，就可以知道实际的最大值；如果没有定义该符号，那么调用带参数`_SC_IOV_MAX`的`sysconf`（见1.5.5节）也可以。但是，实际上无论如何设计`readv`和`writenv`，16都足够了。

下面的几段代码来自一个例程，该例程利用`writenv`写了一个头结构和两个数据结构。首先，结构声明如下：

```
#define VERSION 506
#define STR_MAX 100

struct header {
    int h_version;
    int h_num_items;
} hdr, *hp;
struct data {
    enum {TYPE_STRING, TYPE_FLOAT} d_type;
    union {
        float d_val;
        char d_str[STR_MAX];
    } d_data;
} d1, d2, *dp;
struct iovec v[3];
```

（这个版本只是识别这个头类型的一些数字）

接下来，我们初始化头、两个数据结构以及向量：

```
hdr.h_version = VERSION;
hdr.h_num_items = 2;
d1.d_type = TYPE_STRING;
strcpy(d1.d_data.d_str, "Some data to write");
d2.d_type = TYPE_FLOAT;
d2.d_data.d_val = 123.456;
```

⊖ `sendmsg`和`recvmsg`也采用了相同的结构；见8.6.3节。

```
v[0].iov_base = (char *)&hdr; /* iov_base is sometimes char **/
v[0].iov_len = sizeof(hdr);
v[1].iov_base = (char *)&d1;
v[1].iov_len = sizeof(d1);
v[2].iov_base = (char *)&d2;
v[2].iov_len = sizeof(d2);
```

然后利用一次writev调用写入所有3个结构:

```
ec_negl( n = writev(fd, v, sizeof(v) / sizeof(v[0])) )
```

注意, 在iov_base的初始化中, 因为SUS将其设置为void指针, 因此我们通常不需要强制类型转换。但是FreeBSD (以及其他系统上也可能) 将其定义为char指针。强制类型转换对两者都适用。

为了说明数据在文件中确实是连续的, 读回数据, 并将其输出, 但不把数据读到原来的结构中, 而是一个大的匿名缓存中, 这个缓存采用合适类型的指针 (hp和dp) 指向。注意, 这里是利用lseek函数绕回到用O_RDWR (未显示) 打开的文件的开头:

```
ec_null( buf = malloc(n) )
ec_negl( lseek(fd, 0, SEEK_SET) )
ec_negl( read(fd, buf, n) )
hp = buf;
dp = (struct data *) (hp + 1);
printf("Version = %d\n", hp->h_version);
for (i = 0; i < hp->h_num_items; i++) {
    printf("#%d: ", i);
    switch (dp[i].d_type) {
        case TYPE_STRING:
            printf("%s\n", dp[i].d_data.d_str);
            break;
        case TYPE_FLOAT:
            printf("%.3f\n", dp[i].d_data.d_val);
            break;
        default:
            errno = 0;
            EC_FAIL
    }
}
ec_negl( close(fd) )
free(buf);
```

输出如下:

```
Version = 506
#0: Some data to write
#1: 123.456
```

除了显而易见的, 即除了其他函数需要几次调用才能完成的工作, 该函数一次就能完成之外, readv和writev的价值到底是多少? 表2-4列出了一些计时测试数据, 这些数据表明了writev和write之间写入数据时间的区别, writev写16个数据项, 每个200字节, 这样做50 000次, 用16次write替代writev再这样做。如此, 每次测试写一个160MB的文件。因为不同的UNIX系统运行在不同的硬件上, 所以我们将时间统一了一下, 将显示writev的系统时间设成了50秒, 这正好是Linux机器上的规定。[⊖]

⊖ 在看此表时, 记着你不能对操作系统 (比如Linux与FreeBSD) 的相对速度下任何结论, 因为它们每个是分别进行归一化的。在某种程度上, 我们只能关心每一个系统中writev对write的优势。

表2-4 write与writev的速度

系统调用	用户	系统
Solaris		
writev	1.35	50.00
write	8.45	67.61
Linux		
writev	0.17	50.00
write	1.83	27.67
FreeBSD		
writev	0.39	50.00
write	4.90	209.89

从表中可以很清楚地看到，在Solaris系统，尤其在FreeBSD系统上，writev明显优于write。Linux上系统时间实际上更差。仔细读Linux代码就能知道原因，代码表明，对于文件，Linux系统仅仅是通过向量循环来实现，对向量的每个元素都调用write。对于套接字（这是readv和writev设计的目的）Linux做得要好一些，它把向量一直传送到了某个非常低级的代码上。尽管没有计时测试报告，但从代码上可以很明显地看到，在套接字上writev的确更有优势。

2.16 同步I/O

本节讲解如何绕过2.12.1节介绍过的内核缓存。并且如果不想深入研究内核缓存，当缓存被刷新至磁盘时，如何控制缓存。

2.16.1 已同步的与同步的比较

在英语中“已同步的”（synchronized）和“同步的”（synchronous）这两个词意思是非常近似的，但是在UNIX I/O中，它们的含义是不同的：

- 已同步的I/O：是指对write的调用（或者是它的同类pwrite和writev）等到数据被刷新至输出设备（主要是磁盘）才返回。如2.9节所述，write是非已同步的（unsynchronized），通常write将数据保留在内核缓存中就返回。如果计算机在此间崩溃，数据将丢失。
- 同步的I/O：是指read（以及同类函数）等到能获得数据才返回，而write（以及同类函数）至少要等到数据被写到内核缓存中才返回，并且当I/O也是同步时，数据将被写到设备。到目前为止，这里讲述的read、pread、readv、writev、pwrite以及writev，所有这些操作都是同步的。

因此，通常UNIX I/O是非已同步的（unsynchronized）和同步的。实际上，在某种程度上读和写都是异步的，因为这里高速缓存起作用；然而，一旦把写操作设置为已同步的（每一次调用强迫缓存输出），实际等待一次写操作的返回就会大大降低程序的速度，那时就真的需要异步写了。你想说：“初始化写操作后，在写的过程中我可以离开去做有用的事，以后在需要的时候再来询问发生了什么。”读，尤其是非顺序读时，某种程度上总是已同步的（如果数据不在缓存，内核不会伪造），也不必等待它们。

使read和write已同步化，需要设置open标志，或者执行系统调用刷新内核缓存，这就是本节的主题。异步I/O采用了一组完全不同的系统调用（例如aio_write），本书将在

3.9节对其进行讲述，到时将进一步清楚地阐述已同步的和同步的概念。

2.16.2 刷新缓存系统调用

最老的、最著名的，也是最低效的已同步I/O调用是sync:

sync —— 调度缓存刷新

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

sync所做的是告诉内核刷新缓存，这样内核会立即将事件加入待做事情的相关列表中。但是由于sync立刻返回，因此刷新是迟后的。所以仍不能确定什么时候缓存开始刷新。sync也是比较笨的——所有已经写过的缓存都要刷新，而不仅仅只是所关心的相关文件缓存的刷新。

这个系统调用的主要作用是实现sync命令，它在UNIX当机时或者可移动设备解挂之前运行。对应用程序来说，有更好的选择。

下一个调用是fsync，其行为最小程度上与sync相似，但它只对某个特定文件相关的写缓存起作用。如果定义了POSIX_FSYNC选项符号，SUS2系统以及早期的系统就能支持它。

fsync —— 对某个文件执行调度或强制刷新缓存操作

```
#include <unistd.h>
int fsync(
    int fd          /* file descriptor */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

这里说“在最小程度上”，是因为当支持已同步的I/O选项（_POSIX_SYNCHRONIZED_IO）时（见1.5.4节），保护性能会更强：直到缓存中的数据已经物理上写入了设备控制器或者检测到了某种错误后才返回。[⊖]

下面的函数option_sync_io可用来检查是否支持该选项。见1.5.4节对其所做的解释。

```
OPT_RETURN option_sync_io(const char *path)
{
    #if _POSIX_SYNCHRONIZED_IO <= 0
        return OPT_NO;
    #elif _XOPEN_VERSION >= 500 && !defined(LINUX)
        #if !defined(_POSIX_SYNC_IO)
            errno = 0;
            if (pathconf(path, _PC_SYNC_IO) == -1)
                if (errno == 0)
                    return OPT_NO;
                else
                    EC_FAIL
            else
                return OPT_YES;
        #endif
    #endif
    EC_CLEANUP_BGN
}
```

⊖ 数据仍有可能还在控制器的缓存中，但通常即使UNIX崩溃或死机，只要硬件仍在加电和起作用，那么就可以从控制器的缓存中将数据极其迅速地写到存储介质中。

```

        return OPT_ERROR;
    EC_CLEANUP_END
    #elif _POSIX_SYNC_IO == -1
        return OPT_NO;
    #else
        return OPT_YES;
    #endif /* _POSIX_SYNC_IO */
    #elif _POSIX_VERSION >= 199309L
        return OPT_YES;
    #else
        errno = EINVAL;
        return OPT_ERROR;
    #endif /* _POSIX_SYNCHRONIZED_IO */
}

```

最后一个同步调用是 `fdatasync`，它比 `fsync` 执行速度稍快，因为它只操作实际数据，而对对象文件修改时间等信息不予控制。对大部分关键应用来说，这就够用了，一般高速缓存写操作会关心稍后的控制信息。`fdatasync` 仅是已同步 I/O 选项的一部分（也就是说，没有类 `sync` 行为）。

fdatasync—对某个文件的数据执行强制刷新缓存操作

```

#include <unistd.h>

int fdatasync(
    int fd          /* file descriptor */
);
/* Returns 0 or -1 on error (sets errno) */

```

就已同步 I/O 选项而论，`fsync` 和 `fdatasync` 都能够返回真实的 I/O 错误，它们的 `errno` 被设置为 `EIO`。

对这些函数需要指出的最后一点是：如果不支持已同步 I/O，当调用 `sync` 或者 `fsync` 时，实现上不必执行任何操作（不引入 `fdatasync`）。它们也许是空操作，或仅当作请求。但是如果支持该选项，那么尽管实际所发生的已同步要取决于设备驱动和设备，但也要求实现提供更高级别的数据完整性。

2.16.3 用于已同步的 open 标志

当应用程序必须知道数据已经完成写入工作，例如在向用户报告数据库事务已经完成提交工作之前，一般就需要调用 `fsync` 或者 `fdatasync`。但是对于更关键的应用，可以通过 `open` 标志来实现，即对每一次 `write`、`pwrite` 和 `writew` 都隐含调用 `fsync` 或者 `fdatasync`。

在 2.4.4 节的表中第一次提到了标志 `O_SYNC`、`O_DSYNC` 和 `O_RSYNC`，这些标志只是在支持已同步的 I/O 的系统上才可以得到，下面是它们的功能：

`O_SYNC`：每次调用 `write` 后隐含调用 `fsync`（全部更新）。

`O_DSYNC`：每次调用 `write` 后隐含调用 `fdatasync`。

`O_RSYNC`：同步进行 `read` 和 `write`，必须和 `O_SYNC` 或者 `O_DSYNC` 标志一起使用。

（这里虽然以 `write` 函数举例，但同样适用于 `pwrite` 和 `writew`，此外 `read` 也与之类似）。

`O_RSYNC` 标志实际完成的工作是确保节点的访问时间以同步方式更新，这意味着 `O_DSYNC` 标志可能什么也不用做。即使使用 `O_SYNC` 标志，访问时间也很少紧迫到需要同步

更新。同样，在某些系统上，即使带有O_RSYNC标志，也可能会使提前读无效。

下面是一个使用O_DSYNC标志的例程，在此例程中对已同步写和非已同步写进行了比较：

```
#define SYNCREPS 5000
#define PATHNAME "tmp"

void synctest(void)
{
    int i, fd = -1;
    char buf[4096];

#ifdef !_POSIX_SYNCHRONIZED_IO || _POSIX_SYNCHRONIZED_IO == -1
    printf("No synchronized I/O -- comparison skipped\n");
#else
    /* Create the file so it can be checked */
    ec_negl( fd = open("tmp", O_WRONLY | O_CREAT, PERM_FILE) )
    ec_negl( close(fd) )
    switch (option_sync_io(PATHNAME)) {
        case OPT_YES:
            break;
        case OPT_NO:
            printf("sync unsupported on %s\n", PATHNAME);
            return;
        case OPT_ERROR:
            EC_FAIL
    }

    memset(buf, 1234, sizeof(buf));

    ec_negl( fd = open(PATHNAME, O_WRONLY | O_TRUNC | O_DSYNC) )
    timestart();
    for (i = 0; i < SYNCREPS; i++)
        ec_negl( write(fd, buf, sizeof(buf)) )
    ec_negl( close(fd) )
    timestop("synchronized");

    ec_negl( fd = open(PATHNAME, O_WRONLY | O_TRUNC) )
    timestart();
    for (i = 0; i < SYNCREPS; i++)
        ec_negl( write(fd, buf, sizeof(buf)) )
    ec_negl( close(fd) )
    timestop("unsynchronized");
#endif
    return;
}

EC_CLEANUP_BGN
    EC_FLUSH("backward");
    (void)close(fd);
EC_CLEANUP_END
}
```

函数timestart和timestop用于得到时间，在1.7.2节已讲过这两个函数。注意那个用于检查路径名的调用option_sync_io，它要求首先创建要使用的文件。三个open调用中的选项有点不平常：第一个使用了O_CREAT而没有使用O_TRUNC标志，因为只想确保那个文件一定存在（之后立刻关闭了它）；后两个使用了O_TRUNC标志而没有使用O_CREAT标志，因为知道文件已经存在。

在Linux上运行的结果如表2-5所示（在Solaris上得到的时间与此相似；FreeBSD上不支持此选项）。

表2-5 已同步I/O与非已同步I/O

测 试	用户时间 [*]	系统时间	实际时间
已同步	0.03	5.57	266.45
非已同步	0.02	1.13	1.15

^{*} 时间以秒为单位

(实际时间是指所有流失的时间, 包括I/O完成任务时的等待时间。)

从表中可以看到, 已同步操作的时间是相当长的, 这就是为什么要使用异步的原因, 具体如何使用将在3.9节中讲述。[⊖]

2.17 truncate和ftruncate系统调用

truncate——通过路径截短或加长文件

```
#include <unistd.h>

int truncate(
    const char *path, /* pathname */
    off_t length /* new length */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

ftruncate——通过文件描述符截短或加长文件

```
#include <unistd.h>

int ftruncate(
    int fd, /* file descriptor */
    off_t length /* new length */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

扩展文件很容易(这里所说的扩展是指, 无需实际写入数据就可使文件变大), 前面已经讲述了如何完成扩展文件: `lseek`可以定位到超过文件结尾的某个位置, 并写入数据。`truncate`和`ftruncate`也能实现这项功能, 但是它们最大的用处是截断文件(缩短文件), 过去在UNIX系统上无法实现截断文件, 直到它们出现。(过去常常是必须得写一个全新的文件, 并更名为旧文件名。)

下面是相当精心设计的例子, 在此例子中写操作使用了不寻常的错误检验方法, 在2.9节末尾已对此内容进行了解释:

```
void ftruncate_test(void)
{
    int fd;
    const char s[] = "Those are my principles.\n"
        "If you don't like them I have others.\n"
        "\t--Groucho Marx\n";

    ec_negl( fd = open("tmp", O_WRONLY | O_CREAT | O_TRUNC, PERM_FILE) );
    errno = 0;
    ec_false( write(fd, s, sizeof(s)) == sizeof(s) );
    (void)system("ls -l tmp; cat tmp");
    ec_negl( ftruncate(fd, 25) );
}
```

⊖ 如果这段内容你看不懂, 可以重读2.16.1节。

```
(void)system('ls -l tmp; cat tmp');
ec_neg1( close(fd) )
return;

EC_CLEANUP_BGN
    EC_FLUSH('ftruncate_test');
EC_CLEANUP_END
}
```

下面是输出:

```
-rw-r--r--  1 marc      sysadmin    80 Oct  2 14:03 tmp
Those are my principles.
If you don't like them I have others.
    --Groucho Marx
-rw-r--r--  1 marc      sysadmin    25 Oct  2 14:03 tmp
Those are my principles.
```

`ftruncate`也常用于减少共享内存的大小, 7.14节将介绍此内容。

练习

- 2.1 改变2.4.3节中的`lock`, 以便将登录名保存到锁文件 (使用`getlogin`系统调用, 见3.5.2节)。当`lock`返回值为`false`时, 添加适当的参数, 以便提供需要锁的用户的登录名。
- 2.2 写一段程序, 实现打开文件, 并使用`O_APPEND`标志向文件写入一行文本。同时运行几个并发进程执行此段程序, 并保证文本行不混淆。接着不使用`O_APPEND`标志重新编写程序, 并在每次写之前, 使用`lseek`函数定位到文件的结尾。重新运行那些并发进程, 检查文本是否会混淆。
- 2.3 使用大小分别为2、57、128、256、511、513和1024缓存重新运行2.12.2节的缓存I/O计时测试程序。如果感兴趣, 可以实验其他数据。如果可以得到UNIX的不同版本, 那么请在不同版本上运行此实验, 并将结果绘制到一张表格内。
- 2.4 编写打开文件的函数以实现文件的读写操作, 并将其加入到`BUFIO`包中 (见2.12.2节)。
- 2.5 向`BUFIO`包中添加`Bseek`函数。
- 2.6 写一个不带任何选项的`cat`命令。为了增加信任度, 也可以带尽可能多的选项来实现, 但要符合SUS的规范。
- 2.7 和练习2.6的要求一样, 编写`tail`命令。



第3章 高级文件I/O

3.1 概述

本章将对第2章遗留的内容进行介绍。首先将已介绍的I/O系统调用扩展到磁盘特殊文件上，以便观察文件系统的内部结构。然后介绍可以链接到已存在的文件的、附加的系统调用；创建、删除和读目录；获得或者修改文件状态信息。

本章使用较多的是第2章的系统调用，而不是本章介绍的高级特征。但是通过明白如何使用这些系统调用，就可以对文件I/O如何工作有一个更加完整的理解。

本章的程序示例比以前介绍的程序内容更加广博，因为这些程序说明了文字所无法表述的内涵，所以值得细心研究。

3.2 磁盘特殊文件和文件系统

本节将讲述如何对磁盘特殊文件进行I/O，这些I/O可以实现对UNIX文件系统内部的访问。另外，还将介绍如何安装和卸载文件系统。

3.2.1 磁盘特殊文件的I/O

到现在为止，通过内核文件系统完成了独占I/O，它使用了相对高层的抽象，如文件、目录和信息节点。和在2.12节讨论的一样，文件系统是在使用高速缓存^①的块I/O系统之上实现的。访问块I/O系统是通过块特殊文件，或者与磁盘直接接口的块设备实现的。磁盘被看作是扇区大小的倍数的一系列块构成的，一般扇区大小是512。

也许有几个物理磁盘，每一种物理磁盘可以被分成片，每一个片被称作卷（volume）、分区（partition）或者文件系统（file system）。（文件系统的概念出现混淆，因为它也是内核的一部分。通过上下文可以对本书中所使用的这个术语所要表达的意思有更清楚的了解。）

每一个卷对应于一个特殊文件，一般情况下特殊文件名由设备名构成，如“hd”，其后跟随数字和字母，表明它占用哪一个物理磁盘的哪个扇区。尽管这些特殊文件不需要链接到/dev目录，但通常都这么做。例如，在Linux上，文件/dev/hdb3表示第二个物理硬盘的第三个分区。

原则上通过I/O系统调用可以像普通文件一样操作磁盘特殊文件，可以对它进行打开（为了读和/或写数据）、读或者写入数据（在任意单元中）、定位（对任意字节范围），以及关闭操作。磁盘特殊文件使用高速缓存（因为是块特殊文件），但是在卷中没有目录、文件、信息节点、访问权限、所有者、大小、时间等内容。仅需要处理一个巨大的、已编号块的数组。

但实际上，因为没有访问权限，大多数用户不能执行这些操作。读取包含其他用户文件的磁盘会危及它们的保密性，尽管当磁盘被当作特殊文件看待时，看起来很随意（没有按照明显顺序将块分配给UNIX文件和目录）。不通过内核文件系统向磁盘写数据将造成更严重的破坏。

^① 更新的系统使用的是虚拟内存系统而不是高速缓存，但是对内核如何控制文件来说，高速缓存仍是个有用的抽象模型。

另一方面，如果卷的使用者只是某一个用户，不被其他的用户文件和目录所使用，那么将不会有冲突。实现数据库管理者或者数据获取系统的用户确实希望保留一个卷，这样就可以像块特殊文件一样访问它。当然，限制是仅可提供这么多的磁盘特殊文件。通常如果一个应用程序使用的是一个磁盘特殊文件，那么这个应用程序可能是这个计算机上仅有的一个，或者肯定是主要的一个。

可能比块磁盘特殊文件更快的是原始（raw）磁盘特殊文件，这些设备驱动器处理磁盘的同一区域。然而，这些原始磁盘特殊文件是字符设备，不是块设备。那意味着它们不遵从块模型，并且不使用高速缓存。这样处理起来会更好。

当对一个原始特殊文件初始化读或者写时，进程将被锁定在内存中（防止交换），因此没有物理数据地址的更改。另外，如果硬件和驱动器支持原始特殊文件，那么磁盘将接收指令通过DMA传输数据。数据直接在进程数据段和磁盘控制器之间流动，根本不经过内核。一次传输的数据多半大于一个块。

通常，原始设备上I/O的灵活性比普通文件和块特殊文件差，需要在成倍的磁盘扇区上进行I/O操作。DMA硬件可能需要进程的缓存地址处在特定的范围内，定位也可能只在块的范围內实现。这些限制不会使面向数据库文件系统的设计者感到麻烦，因为他们发现将磁盘当作固定大小的页无论如何是很方便的。

到目前为止，我们已经看到UNIX特征随着版本的不同而略有不同，然而也随着硬件、设备驱动器甚至安装而变化。因为计算机上的I/O硬件发生了巨大的变化，所以设备驱动器也随之发生了变化。同样，实现成果的目标也影响到原始I/O速度的重要性，例如在通用桌面系统上，它的重要性已经大打折扣了。

像已同步I/O（见2.16节）一样，原始I/O也有一个显著的优点和缺点。显著的优点是，因为进程要等待写完成，又因为哪个进程拥有数据是没有疑问的，所以可以通过返回值-1和errno代码将物理写错误告知给进程。尽管有一个定义的代码（EIO），但是究竟是否传递取决于设备驱动程序的实现代码，因此必须检查系统规范文档，或者甚至要看设备驱动程序的代码才能确定。

原始I/O的显著缺点是，因为进程要等待读或者写的完成，又因为UNIX设计允许进程一次只发送一个单独的read或write系统调用，所以进程执行I/O速度很快，但不是经常这样。例如，对于一个带有一个中心数据库管理进程的多用户数据库应用程序（常见的安排），如果应用原始I/O，因为每次仅有一个进程使用I/O，所以将造成巨大的I/O流量拥塞。解决方法是使用多个数据库进程、多线程（见5.17节）或者异步I/O（见3.9节）。

原始I/O的另一个不太显著的缺点（我们可以马上解决的）是，当某个进程被锁定在内存中时，由于没有内存空间把其他进程交换进来，所以其他进程将无法运行。这有点遗憾，因为DMA并不使用CPU，并且实际上这些进程可能以另外的方式运行。解决方法只是添加更多的内存。根据目前市场价格，没有足够的内存避免大多数交换（即使不是全部）不能成为一个借口，特别是在运行一个足够完成原始I/O的重要应用程序的计算机上。

表3-1概括了在普通文件、块磁盘设备和原始磁盘设备之间的功能区别。

表3-1 I/O特性比较

特性	普通文件	块磁盘设备	原始磁盘设备
目录、文件、信息节点、权限等	是	不	不
高速缓存	是	是	不
I/O错误返回值，DMA	不	不	是

为了说明速度间的差异，在Solaris系统上，进行10 000次读操作，每次读65 536字节。其中读了普通文件、块磁盘设备（/dev/dsk/c0d0p0）和原始磁盘设备（/dev/rdisk/c0d0p0）。和第2章类似，在表3-2中，列出了系统时间、用户时间以及实际时间，以秒为单位。因为读原始磁盘设备时的某些特征与带有O_RSYNC标志读普通文件有些相似（详细解释见2.16.3节），所以也包含了带有O_RSYNC标志时读普通文件的时间。

表3-2 I/O时间比较

I/O类型	用户时间	系统时间	实际时间
普通文件	0.40	21.28	441.75
带有O_RSYNC O_DSYNC标志读普通文件	0.25	25.50	412.05
块磁盘设备	0.38	22.50	562.78
原始磁盘设备	0.10	2.87	409.70

如看到的一样，在Solaris上原始I/O的速度优势是很大的，在Linux系统上，得到的结论是相同的。我不厌其烦地对写操作进行比较，因为我知道使用高速缓存的两种文件类型优势比较明显，也因为我没有一个可乱写的空卷。当然，因为带有高速缓存的时间没有包含物理I/O时间，而且包含的原始I/O时间太少，所以这种比较也不公平。

3.2.2 对文件系统的低级访问

通过像磁盘设备一样读出文件系统，可以查看文件系统的内部结构（假设用户具有读权限），这是有启发的。在应用程序中，用户可能从来不做这个事情，但是那是低级文件工具如fsck做的工作。

以下系统的文件系统设计都比UNIX系统多：Solaris上的UNIX文件系统（UNIX File System, UFS）、FreeBSD上的快速文件系统（Fast File System, FFS）（也称作UFS），以及Linux系统上的ReiserFS和Ext2fs。这里并没有严格地遵循早期的（20世纪70年代）UNIX文件系统和FFS。

原始磁盘设备被当作固定大小的块序列，比如2048字节。（对于这个讨论来说实际大小并不重要。）大约第一个块保留给启动程序（如果磁盘是可启动的）、磁盘标号以及其他类似的管理信息。

文件系统准确地从带有超级块（superblock）磁盘头的固定偏移量处开始，它包含多种结构信息，如信息节点的数目、卷中包含块的总数以及自由块的链接表头等。每个文件（普通文件、目录、特殊文件等）使用一个信息节点，并且必须至少有一个目录链接此文件。磁盘上有数据的文件（普通文件、目录和符号链接）也有由信息节点指向的数据块。信息节点是从超级块指向的位置开始的。

信息节点0和1是没用的^①，信息节点2是为根目录（/）保留的，因此当给定内核一个绝对路径时，内核可以从一个已知的位置开始。其他的索引节号没有特殊的意义，可以根据需要将其分配给文件。

下面这个专门针对FreeBSD上的FFS实现的程序给出了一种通过读磁盘设备/dev/ad0s1g访问超级块和信息节点的方法，这里/dev/ad0s1g恰巧是包含/user目录树的原始磁盘设备。这可以通过执行mount命令看到（FFS已知为“usf”类型）：

^① 信息节点从1开始编号，允许使用0是指“没有信息节点”（例如空目录）。历史上，信息节点1用于收集坏的磁盘块。

```

$ mount
/dev/ad0s1a on / (ufs, NFS exported, local)
/dev/ad0s1f on /tmp (ufs, local, soft-updates)
/dev/ad0s1g on /usr (ufs, local, soft-updates)
/dev/ad0s1e on /var (ufs, local, soft-updates)
procfs on /proc (procfs, local)

```

下面是程序:

```

#ifndef FREEBSD
#error "Program is for FreeBSD only."
#endif

#include "defs.h"
#include <sys/param.h>
#include <ufs/ffs/fs.h>
#include <ufs/ufs/dinode.h>

#define DEVICE "/dev/ad0s1g"

int main(int argc, char *argv[])
{
    int fd;
    long inumber;
    char sb_buf[((sizeof(struct fs) / DEV_BSIZE) + 1) * DEV_BSIZE];
    struct fs *superblock = (struct fs *)sb_buf;
    struct dinode *d;
    ssize_t nread;
    off_t fsbo, fsba;
    char *inode_buf;
    size_t inode_buf_size;

    if (argc < 2) {
        printf("Usage: inode n\n");
        exit(EXIT_FAILURE);
    }
    inumber = atol(argv[1]);
    ec_negl( fd = open(DEVICE, O_RDONLY) )
    ec_negl( lseek(fd, SBLOCK * DEV_BSIZE, SEEK_SET) )
    switch (nread = read(fd, sb_buf, sizeof(sb_buf))) {
    case 0:
        errno = 0;
        printf("EOF from read (1)\n");
        EC_FAIL
    case -1:
        EC_FAIL
    default:
        if (nread != sizeof(sb_buf)) {
            errno = 0;
            printf("Read only %d bytes instead of %d\n", nread,
sizeof(sb_buf));
            EC_FAIL
        }
    }
    printf("Superblock info for %s:\n", DEVICE);
    printf("\tlast time written = %s", ctime(&superblock->fs_time));
    printf("\tnumber of blocks in fs = %ld\n", (long)superblock->fs_size);
    printf("\tnumber of data blocks in fs = %ld\n",
(long)superblock->fs_ds_size);
    printf("\tsize of basic blocks in fs = %ld\n",
(long)superblock->fs_bs_size);
}

```

```

printf("\tsize of frag blocks in fs = %ld\n",
      (long)superblock->fs_fsize);
printf("\tname mounted on = %s\n", superblock->fs_fsmnt);

inode_buf_size = superblock->fs_bsize;
ec_null( inode_buf = malloc(inode_buf_size) );

fsba = ino_to_fsba(superblock, inumber);
fsbo = ino_to_fsbo(superblock, inumber);

ec_negl( lseek(fd, fsbtodb(superblock, fsba) * DEV_BSIZE, SEEK_SET) )
switch (nread = read(fd, inode_buf, inode_buf_size)) {
case 0:
    errno = 0;
    printf("EOF from read (2)\n");
    EC_FAIL
case -1:
    EC_FAIL
default:
    if (nread != inode_buf_size) {
        errno = 0;
        printf("Read only %d bytes instead of %d\n",
              nread, inode_buf_size);
        EC_FAIL
    }
}

d = (struct dinode *)&inode_buf[fsbo * sizeof(struct dinode)];
printf("\ninumber %ld info:\n", inumber);
printf("\tmode = 0%o\n", d->di_mode);
printf("\tlinks = %d\n", d->di_nlink);
printf("\towner = %d\n", d->di_uid);
printf("\tmod. time = %s", ctime((time_t *)&d->di_mtime));
exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}

```

下面是sb_buf的复杂声明:

```
char sb_buf[((sizeof(struct fs) / DEV_BSIZE) + 1) * DEV_BSIZE];
```

要将其大小上舍入为偶数扇区，因为这是在FreeBSD上读原始磁盘设备的一个要求。

第一个lseek函数定位到SBLOCK*DEV_BSIZE，这是超级块的位置。SBLOCK恰好是16，DEV_BSIZE是扇区大小为512字节，因为大部分磁盘的扇区都是这个数。超级块总是从文件系统起始部分的第16个扇区开始。第一组printf函数输出的是较长的fs结构中的一些字段，输出结果如下:

```

Superblock info for /dev/ad0s1g:
  last time written = Mon Oct 14 15:25:25 2002
  number of blocks in fs = 1731396
  number of data blocks in fs = 1704331
  size of basic blocks in fs = 16384
  size of frag blocks in fs = 2048
  name mounted on = /usr

```

因为这个算法在FFS磁盘上很复杂，所以接下来程序使用了某个头文件定义的宏来查找作为其参数(argv[1])被提供的信息节点。通过带有i选项执行ls命令，得到了文件的索引节号:

```
$ ls -li x
383642 -rwxr-xr-x 1 marc marc 11687 Sep 19 13:29 x
```

下面是信息节点383642的其余输出信息:

```
inumber 383642 info:
mode = 0100755
links = 1
owner = 1001
mod. time = Thu Sep 19 13:29:30 2002
```

因为实际希望的是解释普通文件和特殊文件间有怎样的联系，所以不打算在低级磁盘访问上花太多的时间。如果你有FreeBSD系统的话，也许会喜欢把程序稍作修改以显示该系统上的其他信息，或将其移植到其他UNIX系统上。

3.2.3 statvfs和fstatvfs系统调用

在块或原始磁盘设备上找到超级块后，通过read读取超级块是很有意义的，但是在SUS1兼容系统上，有更好的方法，它们使用了statvfs和fstatvfs系统调用：

statvfs——通过路径得到文件系统信息

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

int statvfs(
    const char *path, /* pathname */
    struct statvfs *buf /* returned info */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

fstatvfs——通过文件描述符得到文件系统信息

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

int fstatvfs(
    int fd, /* file descriptor */
    struct statvfs *buf /* returned info */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

通常可以使用statvfs，它返回关于文件系统的信息包含有第一个参数给出的路径。如果文件是打开的，则可以使用fstatvfs，也可以得到同样的信息。

标准中定义了statvfs结构的字段，但实现时不要求支持所有这些字段。大多数实现也支持附加的字段和f_flag字段的附加标志。用户只能通过查看系统的联机资料了解相关规定。即使实现不支持标准字段，字段也照样存在，只是不包含有意义的信息。

struct statvfs——statvfs和fstatvfs的结构

```
struct statvfs {
    unsigned long f_bsize; /* block size */
    unsigned long f_frsize; /* fundamental (fblock) size */
    fsblkcnt_t f_blocks; /* total number of fblocks */
    fsblkcnt_t f_bfree; /* number of free fblocks */
    fsblkcnt_t f_bavail; /* number of avail. fblocks */
    fsfilcnt_t f_files; /* total number of i-numbers */
    fsfilcnt_t f_ffree; /* number of free i-numbers */
    fsfilcnt_t f_favail; /* number of avail. i-numbers */
    unsigned long f_fsid; /* file-system ID */
    unsigned long f_flag; /* flags (see below) */
    unsigned long f_namemax; /* max length of filename */
};
```

下面是对此结构的一些解释:

- 为了加快访问,大部分文件系统为文件分配相当大的块空间,其大小由字段f_bsize定义,但为了避免浪费空间,以较小的碎片块结束文件。碎片块大小称做基本块大小,其大小由字段f_frsize定义,一般简称为fblock。fsblkcnt_t类型字段以fblock为单位,因此为了得到总空间字节数,可以将f_frsize与f_blocks相乘得到。
- fsblkcnt_t和fsfilcnt_t是无符号类型,然而在别的方面却可以定义。典型情况下,它们是long型或者long long型,如果需要显示一个类型并且有C99编译器,可把它强制转换为uintmax_t类型,使用printf的格式%ju;否则,强制转换为unsigned long long型,并使用%llu格式。
- SUS标准仅为f_flag字段规定了两个标志,它们指出了文件系统如何安装:如果是只读方式,则设置ST_RDONLY标志,并且如果在可执行的文件上忽略了set-user-ID-on-execution和set-group-ID-on-execution位,则设置ST_NOSUID(安全预防)。
- 将术语“可获得的”(available)应用在f_bavail和f_favail字段上,是指“对非超级用户进程可用”。相对于“空闲”字段,它比较小,以便当空闲空间紧张时,为性能受损的系统保留一个最少量的空闲空间。

SUS的前身FreeBSD不支持statvfs和fstatvfs函数,但是它有一个相似的statfs函数,它的结构名不同,大部分字段也不相同。对基本信息同时采用statvfs与statfs,在某种程度上讲是可能的,像下面的程序示例那样。

```
#if _XOPEN_SOURCE >= 4
#include <sys/statvfs.h>
#define FCN_NAME statvfs
#define STATVFS 1

#elif defined(FREEBSD)
#include <sys/param.h>
#include <sys/mount.h>
#define FCN_NAME statfs

#else
#error "Need statvfs or nonstandard substitute"
#endif

void print_statvfs(const char *path)
{
    struct FCN_NAME buf;

    if (path == NULL)
        path = ".";
    ec_negl( FCN_NAME(path, &buf) )
#ifdef STATVFS
    printf("block size = %lu\n", buf.f_bsize);
    printf("fundamental block (fblock) size = %lu\n", buf.f_frsize);
#else
    printf("block size = %lu\n", buf.f_iosize);
    printf("fundamental block size = %lu\n", buf.f_bsize);
#endif
    printf("total number of fblocks = %llu\n",
           (unsigned long long)buf.f_blocks);
    printf("number of free fblocks = %llu\n",
           (unsigned long long)buf.f_bfree);
    printf("number of avail. fblocks = %llu\n",
           (unsigned long long)buf.f_bavail);
}
```

```

printf("total number of i-numbers = %llu\n",
(unsigned long long)buf.f_files);
printf("number of free i-numbers = %llu\n",
(unsigned long long)buf.f_ffree);
#ifdef STATVFS
printf("number of avail. i-numbers = %llu\n",
(unsigned long long)buf.f_favail);
printf("file-system ID = %lu\n", buf.f_fsid);
printf("Read-only = %s\n",
(buf.f_flag & ST_RDONLY) == ST_RDONLY ? "yes" : "no");
printf("No setuid/setgid = %s\n",
(buf.f_flag & ST_NOSUID) == ST_NOSUID ? "yes" : "no");
printf("max length of filename = %lu\n", buf.f_namemax);
#else
printf("Read-only = %s\n",
(buf.f_flags & MNT_RDONLY) == MNT_RDONLY ? "yes" : "no");
printf("No setuid/setgid = %s\n",
(buf.f_flags & MNT_NOSUID) == MNT_NOSUID ? "yes" : "no");
#endif
printf("\nFree space = %.0f%%\n",
(double)buf.f_bfree * 100 / buf.f_blocks);
return;

EC_CLEANUP_BGN
EC_FLUSH("print_statvfs");
EC_CLEANUP_END
}

```

下面是在Solaris系统上包含/home/marc/aup目录的文件系统的输出结果（Linux系统具有相似的结果）：

```

block size = 8192
fundamental block (fblock) size = 1024
total number of fblocks = 4473046
number of free fblocks = 3683675
number of avail. fblocks = 3638945
total number of i-numbers = 566912
number of free i-numbers = 565782
number of avail. i-numbers = 565782
file-system ID = 26738695
Read-only = no
No setuid/setgid = no
max length of filename = 255

```

Free space = 82%

下面是在FreeBSD系统上目录/user/home/marc/aup的输出结果：

```

block size = 16384
fundamental block size = 2048
total number of fblocks = 1704331
number of free fblocks = 1209974
number of avail. fblocks = 1073628
total number of i-numbers = 428030
number of free i-numbers = 310266
Read-only = no
No setuid/setgid = no

```

Free space = 71%



`statvfs` (或者`statfs`) 系统调用是众所周知的`df` (“磁盘空闲”) 命令的核心 (见练习3.2)。下面是它在FreeBSD系统上的报告:

```
$ df /usr
Filesystem 1K-blocks  Used   Avail Capacity  Mounted on
/dev/ad0s1g 3408662 988696 2147274    32%   /usr
```

所报告的1k-blocks值 3408662等于示例程序`print_statvfs`中显示的2k-blocks值1704331。

同样在信息节点中也有读取信息的标准方式, 因此没有必要按照上节中的方式读取设备文件, 在3.5节将讲述这种方式。

3.2.4 安装和卸载文件系统

一个UNIX系统可以拥有多个磁盘文件系统 (硬盘、软盘、CD-ROM、DVD等), 但是它们都可以从根目录开始的单一目录树中得到。将文件系统连接到一个已存在的层次结构叫做安装, 断开连接叫做卸载。图3-1显示了一个大的文件系统, 包含根目录 (信息节点2, 带有目录项x和y) 和一个较小的未连接的文件系统, 也有自己的根目录, 也编号为2。(以前讲到2通常是根目录的索引节号。)

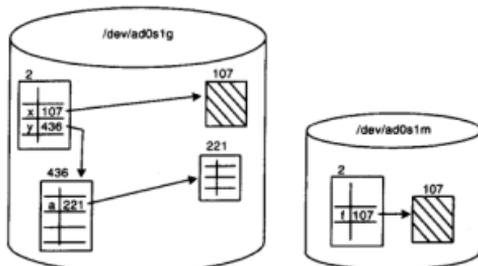


图3-1 两个断开连接的文件系统

为了安装第二个文件系统, 需要两个东西: 设备名, 这里用的是`/dev/ad0s1m`; 你想连接到的目录, 这里选择`/y/a` (信息节点221)。下面是创建图3-2中树的命令:

```
# mount /dev/ad0s1m /y/a
```

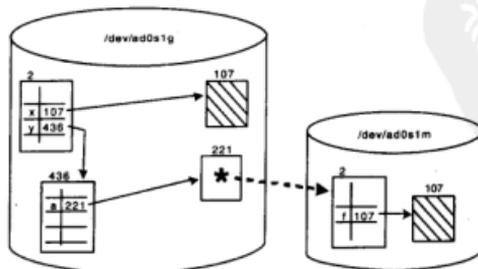


图3-2 已安装的文件系统

目录/y/a中以前的内容现在被隐藏,所有指向那个目录的内容都自动指向了节点ad0s1m:2, ad0s1m:2表示的是“ad0s1m设备上的信息节点2”。因此,现在可以用/y/a/f访问节点ad0s1m:107。当卸载ad0s1m时,其内容将不可访问,这时ad0s1g:221中以前的内容将再现。^①

每个UNIX系都有一个只有超级用户才能调用的系统调用mount和它的相反调用umount(有时称作unmount),但是它们的参数和确切的功能随系统的不同而不同。像其他仅有超级用户才能调用的函数一样,它们没有被标准化。实际上,这些系统调用常用于实现mount和umount命令,并且几乎从不直接调用。举一个例子,下面是Linux系统的一个对照表,对其也不准备详细讲述:

mount——安装文件系统(非标准的)

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

int mount(
    const char *source,      /* device */
    const char *target,     /* directory */
    const char *type,       /* type (e.g., ext2) */
    unsigned long flags,    /* mount flags (e.g., MS_RDONLY) */
    const void *data        /* file-system-dependent data */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

umount——卸载文件系统(非标准的)

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

int umount(
    const char *target      /* directory */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

正常情况下,当文件或目录正在使用时,umount会出现EBUSY错误。通常有一个可替代的系统调用umount2,该函数的第二个参数实现强制执行卸载,有时当系统必须关闭时,该参数是必需的。

从应用编程的观点来看,除了一种情况之外,只要有工作的路径名,一般不关心将要访问的给定的文件或者目录是否在与其它文件或目录分开安装的文件系统上。实际上,也许是在引导序列期间,所有可访问的文件系统(甚至是目录)必须同时安装。有一种情况与链接有关,下一节将介绍这个内容。

3.3 硬链接和符号链接

包含名字和索引节号的目录记录项称作硬链接(hard link)。其他种类的链接称作符号链接(symbolic link),马上将讲述这个内容。图3-3图解了这两种链接。(圆括号内的十六进制数是设备ID,以后将对其进行介绍)。

3.3.1 创建硬链接(link系统调用)

无论创建什么类型的文件(包括目录),都可以得到一个硬链接。通过link系统调用可得到非目录的^②附加的硬链接:

① 实际上目录中没有内容,唯一的目的是当作安装点,但是系统函数有时允许使用它隐藏目录。

② 在某些系统上,超级用户可以链接到已存在目录,但是进行这种操作的结果是,可能会导致目录结构不再是树结构,会使系统管理复杂化。

link——建立一个硬链接

```
#include <unistd.h>

int link(
    const char *oldpath, /* old pathname */
    const char *newpath  /* new pathname */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

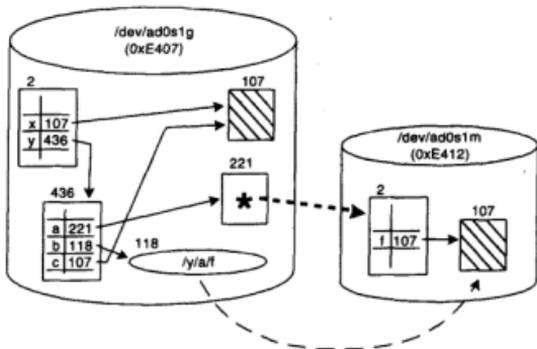


图3-3 硬链接和符号链接

第一个参数oldpath必须是已经存在的链接，它提供所要使用的索引节号。第二个参数newpath表示新链接的名字。在所有方式下这些链接都是平等的，因为UNIX没有主链接和次链接的概念。进程必须对包含新链接的目录具有写权限。第二个参数指定的链接必须是不存在的，link不能更改已经存在的链接。如果出现这种情况，必须首先通过unlink（见2.6节）移去旧链接，或通过rename系统调用更改链接名（见3.3.2节）。

3.3.2 重命名文件或目录（rename系统调用）

乍一看，重命名文件很简单，仿佛是指“更改目录记录项中的名字即可”。即只需要通过link函数创建第二个硬链接，并通过unlink函数移去旧链接，但是有许多复杂的情况：

- 如果文件是目录，就不能为其创建第二个硬链接。
- 有时需要将新名字放在不同的目录中，如果两者在同一个文件系统，这没有问题，但当二者不在同一文件系统时，会出现很大的问题，因为link函数只能工作在同一文件系统中。当然也可以在另一个目录中创建一个符号链接（下一节介绍），但是那不是大多数人认为的“rename”含义。
- 在两个文件系统之间如果将要“更名”（实际上是移动）的是个目录，用户必须移动该目录下的所有子目录。仅移动空目录是很受限制的。
- 如果有多个硬链接，并且在同一个文件系统中更名文件，这是允许的，因为它们参考的索引节号不会更改。但当设法将文件移动到另一个文件系统时，旧链接会失效。因为必须复制文件然后再解链最初的链接，所以可能发生的情况是：除了被移动的硬链接指向该旧文件外，旧文件将保留与其他硬链接的联系，出现这种情况是很糟糕的。
- 如果符号链接与路径相联，而用户改变了那个路径，则该符号链接将成为死点，这样也

很糟糕。

如果仔细地思考该问题，有可能会发现更多的复杂问题，但是你要明白，这些就已经够头疼了。那就是为什么实现“更名”文件和目录的UNIX系统mv命令复杂的原因。但是它仍然不能处理最后的两个问题。

无论如何，mv命令的功能已经很好了，但是需要以下替代机制之一来移动一个文件系统中的目录：

- 总是复制目录和它的子目录，跟着删除旧目录，即使只在它的父目录中重命名旧目录。
- link系统调用可以对目录操作，即使只限于超级用户。（可以通过设置用户ID位临时取得超级用户的权限来运行mv命令。）
- 使用可以在一个文件系统中更名目录的新的系统调用。

第一种替代不好，因为，如果用户想做的只是改变目录名中的几个字符的话，那么这样做工作量就太大了！POSIX系统取消了第二种替代，而支持第三种替代——即使用新的rename系统调用。实际上，rename并不新——它在标准C和BSD中已经存在了，这是无需完全复制或断开链接就可更名目录的唯一方法。

rename —— 重命名文件

```
#include <stdio.h>

int rename(
    const char *oldpath,      /* old pathname */
    const char *newpath      /* new pathname */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

rename函数大概按如下顺序工作：

- 1) 如果newpath已经存在，使用unlink或rmdir删除它。
- 2) link (oldpath, newpath)，即使oldpath是个目录。
- 3) 使用unlink或rmdir删除oldpath。

rename带来一些额外的特性和规则：

- 如上所述，步骤2可应用于目录，即使不是超级用户运行该进程。（但是需要对newpath的父目录具有写权限。）
- 如果newpath已经存在，newpath和oldpath必须同为文件或者同为目录。
- 如果newpath已经存在，并且是目录的话，其必须是空目录。（rmdir具有同样的规则。）在步骤3中，oldpath如果是目录，即使不空，也要将其删除。因为它的内容已经在newpath中存在。
- 如果rename某步失败，文件或目录将保持不变。

因此，你能看出虽然步骤1、2、3好像是可以写成库函数，但是无法让它像系统调用那样工作。

mv命令能做到rename系统调用做不到的一些事情，如在文件系统之间移动文件和目录，将文件组和目录组移动到一个新的父目录下。rename仅能提供mv的一小部分而且是最基本的功能。

最后一点应该注意的是：如果oldpath是符号链接，rename操作的是符号链接，而非符号链接所指向的对象。因此，本来应该称做lrename函数，但是那样会与标准C函数混淆。

3.3.3 创建符号链接 (symlink系统调用)

如图3-3所示, 如果想创建从目录/y到文件/x (ad0slg:107)的第二个链接, 可以执行如下调用:

```
ec_negl( link("/x", "/y/c") )
```

这使得/x和/y/c完全是等同的, ad0slg:107不能同时“存在于”两个目录记录项中。

现在UNIX文件安装机制的透明性破坏了: 因为目录记录项仅仅有标识被链接到对象的索引节号, 而非设备: 索引节号的组合, 而索引节号通常认为是唯一的, 因此不可能创建一个从/y到/y/a/f(ad0slm:107)的硬链接。(在图中, 要使两个完全不同的文件的索引节号都为107就会引起麻烦。)如果试图执行:

```
ec_negl( link("/y/a/f", "/y/b") )
```

将产生一个EXDEV错误。

每一个有经验的UNIX用户都知道, 解决的方式是采用符号链接。在硬链接中, 用户要链接的索引节号直接存于目录中, 和硬链接不同的是, 符号链接是包含用户想链到的对象的路径文本的小文件。我们需要的链接显示在图3-3的椭圆形中, 符号链接有自己的信息节点(ad0slg:118), 但是通常任何访问/y/b的尝试都会使内核去尝试访问/y/a/f, 而/y/a/f位于另一个文件系统上。

一个符号链接能引用另一个符号链接。例如, 通常, 当一个符号链接传递给open时, 内核会保持重引用符号链接, 直到发现它不是符号链接。硬链接不会出现这种情况, 因为硬链接直接引用信息节点, 而信息节点不能是硬链接。(尽管它可以是一个包含硬链接的目录。)换句话说, 如果路径指向的是硬链接, 那么随后跟随的肯定是路径, 但如果指向的是符号链接, 那么随后的实际路径名要取决于符号链接所引用的内容, 直到达到最后的链路结尾。

可以用带有-s选项的ln命令创建符号链接, 但是实际起作用的是symlink系统调用:

symlink —— 建立符号链接

```
#include <unistd.h>

int symlink(
    const char *oldpath,      /* old pathname */
    const char *newpath      /* possible new pathname */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

大多数情况下, symlink的功能与link类似。在这两种系统调用中, 都是由newpath给定所创建硬链接的路径, 但是用symlink, 硬链接指向的是包含由newpath给出的字符串的一个符号链接文件。

在对照表的解释中写得是“可能的”, 因为根本没有有效的newpath——根本不能保证它是一个有效的路径, 还是其他有效的东西。甚至可以这样做:

```
ec_negl( symlink("lunch with Mike at 11:30", "reminder") )
```

接着用ls命令可以看到上面设置的reminder (根据页的大小输出):

```
$ ls -l reminder
lrwxrwxrwx 1 marc sysadmin 24
Oct 16 12:04 reminder -> lunch with Mike at 11:30
```

这种相当松散的行为是有目的的, 它引出了符号链接的一个有用的特征: 引用的对象不

必存在。或者，如果存在，它所在的文件系统可以是未被安装的。

内核对符号链接目标对象是否存在不感兴趣的反面是：当目标没有被链接时，对符号链接什么都不做。通过“解链”，当然是指“解除硬链接”，因为计数到零的硬链接仍是内核决定是否不再需要文件（可回收信息节点和数据空间）的依据。因为一部分符号链接可以在未安装的文件系统上，所以不可能去调整符号链接。对硬链接也是不可能的，因为它们在内核的内部。

那么如何去掉符号链接呢？用`unlink`（参考图3-3）：

```
ec_negl( unlink("/y/b") )
```

这只能解链符号链接`/y/b`，而对其所指的文件`/y/a/f`没有作用。可以把`unlink`当作移去直接由参数规定的硬链接的系统调用。

好，那么如何解链通过符号链接所指的文件呢？可以使用另一个路径，因为文件必须硬链接到某个目录。但是如果拥有的仅是符号链接路径，就可以利用`readlink`系统调用读取它的内容。

readlink —— 读取符号链接

```
#include <unistd.h>

ssize_t readlink(
    const char *path,          /* pathname */
    char *buf,                /* returned text */
    size_t bufsize            /* buffer size */
);
/* Returns byte count or -1 on error (sets errno) */
```

`readlink`在UNIX系统调用中很少使用，因为不能假定返回的字符串是以NUL结尾的。必须保证`buf`指向的空间足够容纳符号链接的内容加上一个NUL字节，然后将空间大小减1作为第三个参数传递。返回值可能需要强加上一个NUL字节，如下所示：

```
ssize_t n;
char buf[1024];

ec_negl( n = readlink("/home/marc/mylink", buf, sizeof(buf) - 1) )
buf[n] = '\0';
```

如果需要，也可以如下移去`/home/marc/mylink`所链接的内容和符号链接本身：

```
ec_negl( unlink(buf) )
ec_negl( unlink("/home/marc/mylink") )
```

当需要处理的是符号链接本身而不是它所引用的内容时，`readlink`不是唯一的选择。对于得到信息节点的信息来说，可以选择另外一个系统调用`stat`（见3.5.1节），它有一个不跟随称作`lstat`的符号链接的变体。

`readlink`示例代码的一个问题是常量1024，该常数代表了一个足以容纳最长路径名的大小。我们的程序不必担心路径名较长的问题，因为我们对它十分小心。当路径名称较长时简单地截断并返回路径名，这还是不太好。

但如果目录中的文件名受限制，比如255字节，而对嵌套的目录深度没有限制，那么要多大的空间才够用呢？当然答案不是1024，因为该数只够处理4层嵌套！寻找答案很麻烦，因此拿出一节进行介绍。请继续往下读。

3.4 路径名

本节将解释如何确定路径名的最大长度，以及如何检索当前目录的路径名。

3.4.1 路径名长度大小

如果对路径名的长度有一个固定的限制，假定是常量 `_POSIX_MAX_PATH`，那么对嵌套的目录的深度也将有一个限制。如果数很大，也会产生问题，因为对UNIX系统来说，通过一些设施，如NFS文件系统将文件系统有效地安装到其他计算机上是常有的事。因此，这个限制必须在运行期间动态确定，并且必须能随文件系统而变化。

这正是 `pathconf` 和 `fpathconf` 所支持的（见1.5.6节）。在遵循POSIX1990的系统上（基本上它们都遵循），可以如下调用它们：

```
static long get_max_pathname(const char *path)
{
    long max_path;

    errno = 0;
    max_path = pathconf(path, _PC_PATH_MAX);
    if (max_path == -1) {
        if (errno == 0)
            max_path = 4096; /* guess */
        else
            EC_FAIL
    }
    return max_path + 1;
}
EC_CLEANUP_BGN
return -1;
EC_CLEANUP_END
}
```

这里将 `pathconf` 的返回值加1，是因为我所看过的文档没有明确说明这个大小是否包含NUL字节。我倾向于假设它没有，而且OS的实现者也如我一样不能确定。

在三个测试系统上，对本地已安装的文件系统和已安装NFS的文件系统调用此函数，在FreeBSD和Solaris上，得到的是1024，在Linux系统上，得到的是4096。但是这些与这些系统的版本和相应的配置文件有关。在你的代码中需要使用 `pathconf` 函数得到这些数字，而不能依靠我的数字。

从技术上讲，对于 `_PC_PATH_MAX`，`pathconf` 从它的参数路径中返回的是最长的相对于路径名的大小，而不是最长的绝对路径。因此，为了完全准确，你应该利用你感兴趣的文件系统的根目录来调用 `pathconf`，然后加上从这个根目录到那个文件系统的根目录的路径的大小。但那种可能不存在，几乎每个人都仅用带有“/”或者“.”参数来调用它。

以前是按照POSIX和SUS标准。实际上，即使系统得到了 `pathconf` 和 `fpathconf` 返回的某个数，系统也不能将这个数当作创建目录的限制，但当试图把一个过长的路径名传递给以路径为参数的系统调用时，系统就会用这个数来强制限制，例如 `open`（错误是 `ENAMETOOLONG`）。在大部分系统上，如果给了足够大的缓存，`getcwd`（见下一节）可以返回比 `pathconf` 或 `fpathconf` 返回的最大值还要长的字符串。

3.4.2 getcwd系统调用

有一个能直接得到当前目录路径的系统调用。和 `readlink`（见3.3.3节）一样，唯一棘手的问题是需知道给定的缓存的大小。

getcwd —— 得到当前目录路径名

```
#include <unistd.h>

char *getcwd(
    char *buf,           /* returned pathname */
    size_t bufsize      /* size of buf */
);
/* Returns pathname or NULL on error (sets errno) */
```

下面的这个函数可以使调用getcwd更加简单，因为它自动分配缓存来容纳路径串。带有true参数的调用会告诉该函数释放缓存。

```
static char *get_cwd(bool cleanup)
{
    static char *cwd = NULL;
    static long max_path;

    if (cleanup) {
        free(cwd);
        cwd = NULL;
    }
    else {
        if (cwd == NULL) {
            ec_negl( max_path = get_max_pathname(".") )
            ec_null( cwd = malloc((size_t)max_path) )
        }
        ec_null( getcwd(cwd, max_path) )
        return cwd;
    }
    return NULL;

    EC_CLEANUP_BGN
        return NULL;
    EC_CLEANUP_END
}
```

下面这段代码的功能与标准pwd命令一样：

```
char *cwd;

ec_null( cwd = get_cwd(false) )
printf("%s\n", cwd);
(void) get_cwd(true);
```

在Solaris系统上运行时，得到如下结果：

```
/home/marc/aup
```

到此为止，用讲过的函数已经可以自己编写getcwd，本书将在3.6.4节给出它的实现代码。getwd的功能与getcwd相似，但它已经过时了，在新程序中很少使用。

3.5 访问和显示文件元数据

本节将解释如何检索文件元数据（例如所有者或者修改时间）以及如何显示它。

3.5.1 stat、fstat和lstat系统调用

一个信息节点包含一个文件元数据（metadata）——所有除了文件名（实际上不属于该文

件)和文件数据(信息节点所指向的)以外的信息。

从磁盘中直接读取信息节点,如3.2.2节所介绍的那样,实在是太笨拙了。幸运的是,有三个实现读取信息节点数据的标准系统调用——stat、fstat和lstat以及一个众所周知的命令ls。

stat——通过路径得到文件信息

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

int stat(
    const char *path,      /* pathname */
    struct stat *buf       /* returned information */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

lstat——不遵循符号链接,通过路径得到文件信息

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

int lstat(
    const char *path,      /* pathname */
    struct stat *buf       /* returned information */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

fstat——通过文件描述符得到文件信息

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

int fstat(
    int fd,                /* file descriptor */
    struct stat *buf       /* returned information */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

stat获取路径作为参数,并通过它查找信息节点;fstat获取打开文件描述符作为参数,并从内核中活动的信息节点表中查找信息节点。lstat与stat相同,除了路径指向一个符号链接时,元数据是针对符号链接本身的,而非符号链接所链接的对象。^①对三者而言,来自信息节点的相同的元数据被重新排列,并放置到提供的stat结构中。

下面是stat的结构,但要注意实现对添加或者重新安排字段没有限制,因此要确保包含本地系统的sys/stat.h头文件。

struct stat——stat, fstat和lstat的结构

```
struct stat {
    dev_t st_dev;           /* device ID of file system */
    ino_t st_ino;          /* i-number */
    mode_t st_mode;        /* mode (see below) */
    nlink_t st_nlink;      /* number of hard links */
    uid_t st_uid;          /* user ID */
    gid_t st_gid;          /* group ID */
    dev_t st_rdev;         /* device ID (if special file) */
    off_t st_size;         /* size in bytes */
    time_t st_atime;       /* last access */
    time_t st_mtime;       /* last data modification */
    time_t st_ctime;       /* last i-node modification */
    blksize_t st_blksize;  /* optimal I/O size */
    blkcnt_t st_blocks;    /* allocated 512-byte blocks */
};
```

① 因为没有办法得到打开符号链接的文件描述符,所以没有flstat。任何使用open中符号链接的企图都会打开被链接的文件,而不是符号链接。并且,如果有那样的方式,以致于文件描述符已经可以指定正确对象时,那么用fstat就行了。

下面是对此结构的解释:

- 设备ID (dev_t类型) 是一个唯一标识已安装文件系统的数字, 即使当设备以NFS文件系统安装时。因此st_dev和st_ino唯一标识这个信息节点。本质上和3.2.4节定义的ad0slg:221符号相同。重新观察图3-3, 在图中每一个设备名下给定了一个十六进制设备ID。

该标准没有规定如何分解一个设备ID, 但本质上, 所有的实现都将其当作主设备和次设备号的组合看待, 其中, 主设备号标识驱动器, 而次设备号标识实际设备, 因为同一个设备驱动器可以与同类型的所有设备接口。一般情况下, 次设备号是最右边的字节。

- 字段st_dev是包含信息节点的设备; 字段st_rdev是特殊文件表示的设备, 仅用于特殊文件。例如, 特殊文件/dev/ad0slm在根文件系统中, 因此它的st_dev是根文件系统所在的设备。但st_rdev是所指的设备, 可以是完全不同的磁盘。
- 根据信息节点类型和实现的不同, 字段st_size具有不同的解释。对于一个代表磁盘数据的信息节点, 即普通文件、目录和符号链接, 它是数据的大小(对符号链接是路径)。对于共享内存对象, 它是内存大小。对于管道, 它是管道中的数据量, 但那不是标准的。
- 无论何时读何种类型文件, 系统都将更新访问时间(st_atime), 但当查找出现在路径中的目录时不更新访问时间。
- 当写文件及从目录中添加或移去硬链接时, 将更新数据修改时间(st_mtime)。
- 当向文件写数据, 或者信息节点隐含修改时(例如更改其所有者或链接数), 将更新信息节点修改时间(st_ctime), 也称作状态修改时间, 但仅因为读取文件引起的访问时间更改时不更新该时间。
- 字段st_blksize位于stat结构中, 那么实现根据文件可以更改它, 如果实现选择这么做。大多数情况下, 该值与超级块中的值相同(见3.2.3节)。
- 如果由于超过文件尾部查找和写入数据时文件产生了空穴, 那么st_blocks*512的值可能小于st_size。
- 当拥有的文件描述符不是来自打开的路径(例如非命名管道或者套接字)时, fstat会特别有用。因为字段st_mode、st_ino、st_dev、st_uid、st_gid、st_atime、st_ctime以及st_mtime都要求有有效的值, 但是其他字段是否有有效值依赖于实现。通常对命名管道和未命名管道来说, st_size字段包含管道中未读取字节的数。

st_mode字段由指示文件类型(普通文件、目录等)、访问权限和其他几个特征的位构成。除了假定一些特定位, 一个可移植的应用程序应该使用宏。首先介绍文件类型的宏。

st_mode —— 文件类型的位掩码和值

```
S_IFMT          /* all type-of-file bits */
S_IFBLK        /* block special file */
S_IFCHR        /* character special file */
S_IFDIR        /* directory */
S_IFIFO        /* named or un-named pipe */
S_IFLNK        /* symbolic link */
S_IFREG        /* regular file */
S_IFSOCK       /* socket */
```

宏S_IFMT定义文件类型位, 其他的宏是这些位的值, 而不是位掩码。因此, 比如说对于套接字的测试代码一定不能写成:

⊖ 实际上, S_IFIFO (虽然是正确的) 是拼写错误; 应该称作S_IPFIPO。

```
if ((buf.st_mode & S_IFSOCK) == S_IFSOCK) /* wrong */
```

而是要写成:

```
if ((buf.st_mode & S_IFMT) == S_IFSOCK)
```

或者, 用户可以使用这些检测宏中的一个, 当结果为真时, 每个都返回非零值, 当结果为假时, 每个都返回0, 因此可以像C布尔表达式一样使用:

st_mode —— 文件类型测试宏

```
S_ISBLK(mode)      /* is a block special file */
S_ISCHR(mode)      /* is a character special file */
S_ISDIR(mode)      /* is a directory */
S_ISFIFO(mode)     /* is a named or un-named pipe */
S_ISLNK(mode)      /* is a symbolic link */
S_ISREG(mode)      /* is a regular file */
S_ISSOCK(mode)     /* is a socket */
```

套接字的检测代码如下:

```
if (S_ISSOCK(buf.st_mode))
```

模式中的某处有9位代表访问权限, 2.3节已经介绍了这些位的宏, 因为open和其他一些系统调用也使用了同样的宏, 这些宏也具有S_Ipwww的形式, 其中p是访问权限(R、W或X), www是所有者(USR、GRP或者OTH)。

现在可以把这些位用作掩码了, 下面是检测组的读权限和写权限的代码:

```
if ((buf.st_mode & (S_IRGRP | S_IWGRP)) == (S_IRGRP | S_IWGRP))
```

为了使解释更加清楚(也可能会使问题更糟糕——对不起!), 可以用如下代码检测组的读权限或写权限:

```
if ((buf.st_mode & S_IRGRP) == S_IRGRP ||
    (buf.st_mode & S_IWGRP) == S_IWGRP)
```

如果需要的话, 也可以采用如下方式编写读和写的情况:

```
if ((buf.st_mode & S_IRGRP) == S_IRGRP &&
    (buf.st_mode & S_IWGRP) == S_IWGRP)
```

在st_mode字段中也有一些其他位, 这里将重复讲述访问权限位, 以便以后返回本页时能够一目了然:

st_mode —— 权限和其他位掩码

```
S_Ixwww          /* x = R|W|X, www = USR|GRP|OTH */
                  /* examples: S_IRUSR, S_IWOTH */
S_ISUID           /* set-user-ID on execution */
S_ISGID           /* set-group-ID on execution */
S_ISVTX           /* directory restricted-deletion */
```

在1.1.5节, 已经讲述了设置用户ID和设置组ID。如果设置了S_ISVTX标志, 那么意味着只有超级用户、目录的所有者或者文件的所有者才可以从目录中解链文件。如果没有设置该标志(一般不设置), 那么在目录中拥有写权限就可以了。[⊖]

⊖ 那是SUS定义。历史上, 称该位为粘性位(sticky bit), 它应用在可执行程序文件上, 以便在交换设备上保持经常使用程序的指令段(例如, shell)。在目前的、面向页的UNIX系统中不常使用。(字母“SVTX”来自“SaVe TeXt”, 正因为如此, 这些指令也被称作“text”。)

一个显示如何使用模式宏的好方法就是像ls命令那样来显示这些模式，即使用带有10个字母的序列（例如：drwxr-xr-x）。简单地说，ls的使用规则如下：

- 第一个字母标识文件的类型。
- 紧接着的9个字母分为3组，每组三个，标识所有者、组和其他用户，如果设置了访问权限位，通常是r、w或者x，如果没有设置访问权限位，则是破折号。
- 如果设置了用户ID和组ID，同时也设置了执行位，那么所有者和组执行字母是s（小写），如果设置了设置位，但没有设置执行位，则所有者和组执行字母是S（大写）。第二种情况的组合是可能的，但没有意义，除非结合设置组ID并不设置组执行位，但这会在一些系统上引发强制文件锁，7.11.5节将做解释。
- 如果设置了限制删除位（I_SVTX），并且也设置了执行（搜索）位，则其他的执行字母是t，否则，当设置了限制删除位，而没有设置执行位时，则其他字母是T。

下面的函数将帮助弄清楚这个算法：

```
#define TYPE(b) ((statp->st_mode & (S_IFMT)) == (b))
#define MODE(b) ((statp->st_mode & (b)) == (b))

static void print_mode(const struct stat *statp)
{
    if (TYPE(S_IFBLK))
        putchar('b');
    else if (TYPE(S_IFCHR))
        putchar('c');
    else if (TYPE(S_IFDIR))
        putchar('d');
    else if (TYPE(S_IFIFO)) /* sic */
        putchar('p');
    else if (TYPE(S_IFREG))
        putchar('-');
    else if (TYPE(S_IFLNK))
        putchar('l');
    else if (TYPE(S_IFSOCK))
        putchar('s');
    else
        putchar('?');
    putchar(MODE(S_IRUSR) ? 'r' : '-');
    putchar(MODE(S_IWUSR) ? 'w' : '-');
    if (MODE(S_ISUID)) {
        if (MODE(S_IXUSR))
            putchar('s');
        else
            putchar('S');
    }
    else if (MODE(S_IXUSR))
        putchar('x');
    else
        putchar('-');
    putchar(MODE(S_IRGRP) ? 'r' : '-');
    putchar(MODE(S_IWGRP) ? 'w' : '-');
    if (MODE(S_ISGID)) {
        if (MODE(S_IXGRP))
            putchar('s');
        else
            putchar('S');
    }
    else if (MODE(S_IXGRP))
        putchar('x');
```



```

else
    putchar('-');
putchar(MODE(S_IROTH) ? 'r' : '-');
putchar(MODE(S_IWOTH) ? 'w' : '-');
if (MODE(S_IFDIR) && MODE(S_ISVTX)) {
    if (MODE(S_IXOTH))
        putchar('t');
    else
        putchar('T');
}
else if (MODE(S_IXOTH))
    putchar('x');
else
    putchar('-');
}

```

此函数不会使用新行中断输出，以后将介绍其原因，下面是检测代码：

```

struct stat statbuf;

ec_negl( lstat("somefile", &statbuf) )
print_mode(&statbuf);
putchar('\n');
ec_negl( system("ls -l somefile") )

```

这是上列检测代码的输出，希望这些能使大家相信所讲述的一切：

```

prw--w--w-
prw--w--w-  1 marc      sysadmin      0 Oct 17 13:57 somefile

```

注意，这里调用的是`lstat`，而不是`stat`，因为如果参数是符号链接，那么需要的是符号链接本身的信息，而不是链接所指的对象。

下面是输出链接数的函数，这是比输出模式更加简单的方法。（你能明白我们朝向哪里吗？）

```

static void print_numlinks(const struct stat *statp)
{
    printf("%5ld", (long)statp->st_nlink);
}

```

为什么要强制转换成`long`型？因为除了知道`nlink_t`是整数类型外，实际上不知道`nlink_t`的类型，所以需要把它强制转换成一个具体的类型以便与`printf`中的格式匹配。同样`stat`结构中其他类型也采用了相同的方式，稍后在本章中可以看到对应的内容。

下一步，需要输出属主和组名，但是为实现这个功能，需要另外两个库函数。

3.5.2 getpwuid、getgrgid和getlogin系统调用

得到所有者ID和组ID很容易，因为`stat`结构中有对应的字段`st_uid`和`st_gid`，但我们想得到是它们的名字。`getpwuid`和`getgrgid`函数可以完成这个功能，这两个函数并不是真正的系统调用，因为它们需要的信息在口令文件和组文件中，任何一个进程只要不怕麻烦，本身都可以读取这些信息。尽管它的问题是文件布局没有标准化。

下面是Solaris系统上的口令文件的“marc”记录项，它是相当典型的情况：

```

$ grep marc /etc/passwd
marc:x:100:14::/home/marc:/bin/sh

```

（口令并不在口令文件中，它存在于一个只有超级用户才可以读取的加密文件中。）

在这个系统中，“marc”是一些组的成员，但是组14是它的登录组：

```
$ grep 14 /etc/group
sysadmin::14:
```

getpuid——得到口令文件入口

```
#include <pwd.h>

struct passwd *getpuid(
    uid_t uid /* user ID */
);
/* Returns pointer to structure or NULL on error (sets errno) */
```

struct passwd——getpuid的结构

```
struct passwd {
    char *pw_name; /* login name */
    uid_t pw_uid; /* user ID */
    gid_t pw_gid; /* group ID */
    char *pw_dir; /* login directory */
    char *pw_shell; /* login shell */
};
```

getgrgid——得到组文件入口

```
#include <grp.h>

struct group *getgrgid(
    gid_t gid /* group ID */
);
/* Returns pointer to structure or NULL on error (sets errno) */
```

struct group——getgrgid的结构

```
struct group {
    char *gr_name; /* group name */
    gid_t gr_gid; /* group ID */
    char **gr_mem; /* member-name array (NULL terminated) */
};
```

如以前讲述的一样，这里所讲的两个结构都是标准字段。实现上可能具有更多的字段，因此需要按照头文件定义的那样使用结构。

现在使用两个查找函数输出登录名和组名，在不知道名字的情况下，仅输出其ID。当文件系统通过网络挂载时，找不到名字是很常见的情况，因为一个系统上的用户在另一个系统上可能没有登录名。

```
static void print_owner(const struct stat *statp)
{
    struct passwd *pwd = getpuid(statp->st_uid);

    if (pwd == NULL)
        printf("%-8ld", (long)statp->st_uid);
    else
        printf("%-8s", pwd->pw_name);
}

static void print_group(const struct stat *statp)
{
    struct group *grp = getgrgid(statp->st_gid);

    if (grp == NULL)
```



```

    printf(" %-8ld", (long)statp->st_gid);
else
    printf(" %-8s", grp->gr_name);
}

```

现在已经明白了，下面是得到已登录用户名字的函数：

getlogin——得到登录名

```

#include <unistd.h>
char *getlogin(void);
/* Returns name or NULL on error (sets errno) */

```

3.5.3 显示文件元数据的更多信息

继续讨论stat结构，下面是输出文件大小的函数。对于特殊文件的情况，因为文件大小没有意义，所以输出的是主设备号和次设备号。根据3.5节的内容可知，虽然设备ID的编码没有标准化，但是最右边的8位通常表示的是次设备号：

```

static void print_size(const struct stat *statp)
{
    switch (statp->st_mode & S_IFMT) {
    case S_IFCHR:
    case S_IFBLK:
        printf("%4u,%4u", (unsigned)(statp->st_rdev >> 8),
            (unsigned)(statp->st_rdev & 0xFF));
        break;
    default:
        printf("%9lu", (unsigned long)statp->st_size);
    }
}

```

接下来是输出文件的数据修改日期和时间，为了完成这项艰巨的任务，使用的是标准C函数strptime。函数time和difftime也包含在标准C中（这三个函数的介绍见1.7.1节）。通常不输出年，除非时间已过6个月，目的是为了节省空间：[⊖]

```

static void print_date(const struct stat *statp)
{
    time_t now;
    double diff;
    char buf[100], *fmt;

    if (time(&now) == -1) {
        printf(" ??????????????");
        return;
    }
    diff = difftime(now, statp->st_mtime);
    if (diff < 0 || diff > 60 * 60 * 24 * 182.5) /* roughly 6 months */
        fmt = "%b %e %Y";
    else
        fmt = "%b %e %H:%M";
    strptime(buf, sizeof(buf), fmt, localtime(&statp->st_mtime));
    printf(" %s", buf);
}

```

⊖ 一般情况下，我们的策略是检测所有错误，除了定义格式（为了输出）和输出时，difftime也不例外，它是没有错误返回的函数之一。

最后一件事是输出文件名，当然文件名并不在stat结构中，因为它不在信息节点中。然而这个问题有点棘手，因为如果名字是个符号链接，那么需要同时输出符号链接及其所包含的内容：

```
static void print_name(const struct stat *statp, const char *name)
{
    if (S_ISLNK(statp->st_mode)) {
        char *contents = malloc(statp->st_size + 1);
        ssize_t n;

        if (contents != NULL && (n = readlink(name, contents,
            statp->st_size)) != -1) {
            contents[n] = '\0'; /* can't assume NUL-terminated */
            printf(" %s -> %s", name, contents);
        }
        else
            printf(" %s -> [can't read link]", name);
        free(contents);
    }
    else
        printf(" %s", name);
}
```

在3.3.3节，介绍readlink时曾指出，为了规定缓存的大小，需要知道最长文件名的字节数。这个函数提供了更加明了的方式，因为路径的准确大小（不包括NUL字节）在符号链接的st_size字段中。^①注意当时分配的缓存是st_size + 1个字节，但是把st_size的值传给了readlink函数，以便保证有NUL字节的空间，以防readlink没有多提供这个字节，因为readlink没有要求这么做。

现在，正如所等待的，下面的程序将所有的输出函数都集中在了一起：

```
int main(int argc, char *argv[])
{
    int i;
    struct stat statbuf;

    for (i = 1; i < argc; i++) {
        ec_negl( lstat(argv[i], &statbuf) )
        ls_long(&statbuf, argv[i]);
    }
    exit(EXIT_SUCCESS);

    EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
    EC_CLEANUP_END
}

static void ls_long(const struct stat *statp, const char *name)
{
    print_mode(statp);
    print_numlinks(statp);
    print_owner(statp);
    print_group(statp);
    print_size(statp);
    print_date(statp);
    print_name(statp, name);
    putchar('\n');
}
```

① 现代的文件系统在信息节点的右边保存的是短符号链接，而不是数据块，但仍用st_size字段给定大小。

下面是简化的ls命令的结果:

```
$ aups /dev/tty
crw-rw-rw-  1 root    root      5,   0 Mar 23  2002 /dev/tty
$ aups a.tmp a.out util
lrwxrwxrwx  1 marc    sysadmin  5 Jul 29 13:30 a.tmp -> b.tmp
-rwxr-xr-x  1 marc    sysadmin  8392 Aug  1  2001 a.out
drwxr-xr-x  3 marc    sysadmin  512 Aug 28 12:26 util
```

从最后一行可以看出, 该函数不知道如何列出目录——它只列出了由参数给出的名字。为了添加这个特性, 需要明白如何从链接中读取目录, 下一节将马上讨论这个问题。

3.6 目录

本节介绍如何读取目录、删除目录、更改当前目录和遍历目录树。

3.6.1 读取目录

下面, 除了在信息节点有特殊的设置, 以及内核不允许写目录之外, UNIX系统几乎总是以普通文件方式来操作目录。在一些系统上, 允许用户通过read读目录, 但是POSIX和SUS标准不要求这样, 并且也没有规定目录信息的内部格式。但是探究一下也有意义, 因此写了一段小程序读取当前目录的头96字节, 并以字符形式(如果内容可输出)和十六进制形式输出:

```
static void dir_read_test(void)
{
    int fd;
    unsigned char buf[96];
    ssize_t nread;

    ec_negl( fd = open(".", O_RDONLY) )
    ec_negl( nread = read(fd, buf, sizeof(buf)) )
    dump(buf, nread);
    return;
}

EC_CLEANUP_BGN
    EC_FLUSH("dir_read_test");
EC_CLEANUP_END
}

static void dump(const unsigned char *buf, ssize_t n)
{
    int i, j;

    for (i = 0; i < n; i += 16) {
        printf("%4d ", i);
        for (j = i; j < n && j < i + 16; j++)
            printf(" %c", isprint((int)buf[j]) ? buf[j] : ' ');
        printf("\n      ");
        for (j = i; j < n && j < i + 16; j++)
            printf(" %.2x", buf[j]);
        printf("\n\n");
    }
    printf("\n");
}
}
```

下面是在Linux系统上的输出结果(减去了“ec”宏跟踪的内容):

```
*** EISDIR (21: "Is a directory") ***
```

虽然结果不那么理想，但是在FreeBSD（和Solaris系统）上有好的结果：

```

0      .          V
60 d8 05 00 0c 00 04 01 2e 00 00 00 56 d8 05 00

16      b
0c 00 04 02 2e 2e 00 00 62 d8 05 00 0c 00 08 02

32      m 2      y      k      C
6d 32 00 c0 79 d8 05 00 0c 00 08 01 6b 00 43 c6

48      p c s y n c _ s
b3 da 05 00 18 00 08 0c 70 63 73 79 6e 63 5f 73

64      i g . o
69 67 2e 6f 00 00 00 00 e3 e3 05 00 10 00 08 06

80      t i m e . o      r
74 69 6d 65 2e 6f 00 c6 72 d8 05 00 0c 00 08 02

```

第一眼看起来比较乱，但仔细查看，可以看到一些有意义的内容。可以看到6个名字：`..`、`...`、`m2`、`k`、`pcsync_sig.o`以及`time.o`。同时相关的索引节号也一定在里面，为了用十六进制显示它们，使用了带有`-i`选项的`ls`命令和`dc`（“桌面计算器”）将十进制转换成了十六进制（后面的斜体为解释内容）：[⊖]

```

$ ls -ldif . . . m2 k
383072 drwxr-xr-x 2 marc marc 2560 Oct 18 11:02 .
383062 drwxr-xr-x 9 marc marc 512 Oct 14 18:05 ..
383074 -rwxrwxrwx 1 marc marc 55 Jul 25 11:14 m2
383097 -rwxr--r-- 1 marc marc 138 Sep 19 13:28 k
$$$ dc
16o          make 16 the output radix
383072p     push the i-number onto the stack and print it
5D860      dc printed i-number in hex (radix 16)
383062p     - ditto ...
5D856
q          quit

```

果真，5D860和5D856显示在了结果的第二行（第一个十六进制行）上。同样在每一个记录项都有一些数字，类似字符串大小的东西，但是实际上不必进一步挖掘里面的含义。读取目录的一个比较好的标准方式是使用专为此目的设计的系统调用：

opendir——打开目录

```

#include <dirent.h>

DIR *opendir(
    const char *path      /* directory pathname */
);
/* Returns DIR pointer or NULL on error (sets errno) */

```

closedir——关闭目录

```

#include <dirent.h>

int closedir(
    DIR *dirp             /* DIR pointer from opendir */
);
/* Returns 0 on success or -1 on error (sets errno) */

```

⊖ 带有`-ldif`选项的`ls`的功能是显示长列表但不遍历目录，显示信息节点数但不进行排序。

readdir —— 读目录

```
#include <dirent.h>

struct dirent *readdir(
    DIR *dirp          /* DIR pointer from opendir */
);
/* Returns structure or NULL on EOF or error (sets errno) */
```

struct dirent —— readdir的结构

```
struct dirent {
    ino_t d_ino;      /* i-number */
    char d_name[];   /* name */
};
```

rewinddir —— 返绕目录

```
#include <dirent.h>

void rewinddir(
    DIR *dirp          /* DIR pointer from opendir */
);
```

seekdir —— 搜索目录

```
#include <dirent.h>

void seekdir(
    DIR *dirp,         /* DIR pointer from opendir */
    long loc           /* location */
);
```

telldir —— 得到目录位置

```
#include <dirent.h>

long telldir(
    DIR *dirp          /* DIR pointer from opendir */
);
/* Returns location (no error return) */
```

这些函数的功能和希望的一样：以`opendir`开始，它给出了一个指向DIR的指针（与调用标准C中的`fopen`得到一个指向FILE的指针类似）。然后该指针可作为其他5个函数的参数。可以循环调用`readdir`，直到返回NULL为止，这意味着如果`readdir`没有改变`errno`，则表示已经到达了文件的末尾，如果`readdir`改变了`errno`，则表示出现了错误。（因此为了不出现混淆，在调用`readdir`函数之前应该将`errno`设置为0。）`readdir`函数返回了一个指向`dirent`结构的指针，该结构包含了索引节号和在该目录记录项中的名字。当不用DIR时，可以使用`closedir`关闭它。

`readdir`返回的`d_ino`字段中的索引节号并没有特别的用途，因为如果目录记录项是挂载点，那么它将是挂载前的索引节号，而不是反映当前目录树的索引节号。例如，在图3-3中（见3.3节），`readdir`给出的目录记录项`y/a`的信息节点为221，但是因为其是挂载点，所以有效的信息节点是`ad0slm:2`（简单一个2并不能说明问题，因为设备`ad0slg`中也包含一个2）。信息节点221甚至是不可访问的，因此，当沿着目录树读取目录时，必须通过`stat`函数才能得到正确的目录记录项的索引节号，不能从`d_ino`字段中获取索引节号，在3.6.4节中将讨论如何得到当前目录的路径名。

当需要返回到目录的起始位置时，可能会用到`rewinddir`，使用这个函数可以直接再次

读一个目录，无需关闭后再打开目录。很少会用到`seekdir`和`tellidir`。`seekdir`的参数`loc`需要通过`tellidir`得到，但不能假设它就是目录记录项的数目，也不能像以前见到的情况那样假设目录占用固定宽度的位置。

`readdir`属于这样一类函数：此类函数使用返回指针所指向的单个静态分配的结构。虽然这种方式很方便，但是对多线程程序来说却不是很好，因此改用`readdir`的无结构变种，此变种使用的是传入的内存单元：

readdir_r —— 读目录

```
#include <dirent.h>

int readdir_r(
    DIR *restrict dirp,          /* DIR pointer from opendir */
    struct dirent *entry,       /* structure to hold entry */
    struct dirent **result      /* result (pointer or NULL) */
);
/* Returns 0 on success or error number on error (errno not set) */
```

程序必须传递一个指向`dirent`结构的指针给`readdir_r`，并且`readdir_r`至少要能容纳`NAME_MAX+1`个元素。可以通过调用`pathconf`和`fpathconf`得到`NAME_MAX`的值，利用目录作为参数，与3.4节得到最大路径长度的值的做法一样。`readdir_r`通过`result`参数返回结果，其含义与`readdir`的返回值相同，不同的是不能检测`errno`——因为错误号（或者0）是函数的返回值。但可以使用`ec_rv`宏（见1.4.2节）来检测`errno`，如下面例子所示，该例子列出了当前目录中的名称和索引号。

```
static void readdir_r_test(void)
{
    bool ok = false;
    long name_max;
    DIR *dir = NULL;
    struct dirent *entry = NULL, *result;

    errno = 0;
    /* failure with errno == 0 means value not found */
    ec_negl( name_max = pathconf(".", _PC_NAME_MAX) )
    ec_null( entry = malloc(offsetof(struct dirent, d_name) +
        name_max + 1) )
    ec_null( dir = opendir(".") )
    while (true) {
        ec_rv( readdir_r(dir, entry, &result) )
        if (result == NULL)
            break;
        printf("name: %s; i-number: %ld\n", result->d_name,
            (long)result->d_ino);
    }
    ok = true;
    EC_CLEANUP

    EC_CLEANUP_BGN
    if (dir != NULL)
        (void)closedir(dir);
    free(entry);
    if (!ok)
        EC_FLUSH("readdir_r_test");
    EC_CLEANUP_END
}
```



代码的解释如下:

- 当`errno = 0`时, `pathconf`命令的返回值为-1时表示对目录中的名字没有限制, 这种情况是不允许的。但我们希望代码能处理这种情况, 因此采用这样的捷径: 将`errno`设置为0, 并将所有返回值-1当作错误处理。本条解释适用于所有实际上`errno = 0`的错误消息。这种思想是: 在不将问题复杂化的情况下, 保证代码能处理不可能的情况, 因为无法为不可能的事件建立检测代码, 所以无论如何不可能检测它们。
- 使用`malloc`分配空间是个很棘手的问题。关于`dirent`结构, 能够确信的是字段`d_ino`位于该结构的某个地方 (也许有其他非标准的字段), `d_name`是最后一个字段。因此`d_name`的偏移量加上需要的大小是计算总大小的唯一的安全方式, 同时也要考虑到结构中的空隙 (标准C中允许) 和隐藏字段。
- `EC_CLEANUP`跳转到了清理代码, 如1.4.2节中的一样。布尔值`ok`表示是否有错误。

这些问题真让人头疼! 但如果能够确定不是多个线程, 而仅有一个线程读取某个目录, 那么使用`readdir`就相对容易一些:

```
static void readdir_test(void)
{
    DIR *dir = NULL;
    struct dirent *entry;

    ec_null( dir = opendir(".") )
    while (errno = 0, (entry = readdir(dir)) != NULL)
        printf("name: %s; i-number: %ld\n", entry->d_name,
              (long)entry->d_ino);
    ec_nzero( errno )
    EC_CLEANUP

    EC_CLEANUP_BGN
        if (dir != NULL)
            (void)closedir(dir);
        EC_FLUSH("readdir_test");
    EC_CLEANUP_END
}
```

这里棘手的问题是将`readdir`加入到`while`循环检测中: 逗号表达式的第一部分是将`errno`设置为0, 第二部分求出整个表达式的值, 这是`while`循环检测的内容。也可以不用写得这么紧凑, 但不能写成这样:

```
errno = 0;
while ((entry = readdir(dir)) != NULL) { /* wrong */
    /* process entry */
}
```

因为在目录记录项的处理过程中, 可能会复位`errno`。每次调用`readdir`时都应将`errno`设置为0。^①

总之, 下面是输出结果的头几行 (来自例子之一):

```
name: .; i-number: 383072
name: ..; i-number: 383062
name: m2; i-number: 383074
name: k; i-number: 383097
```

① 也许你正在想: UNIX系统编程的主要困难就是处理`errno`。非常正确! 但是别怨我, 我只是一个带路者! 附录B给出了更加简单的方式。

既然可以读取目录，就可以把readdir循环与ls_long函数（在3.5.3节的末尾出现过）放在一起，得到一个可以列出目录的ls命令：

```
int main(int argc, char *argv[]) /* has a bug */
{
    bool ok = false;
    int i;
    DIR *dir = NULL;
    struct dirent *entry;
    struct stat statbuf;

    for (i = 1; i < argc; i++) {
        ec_negl( lstat(argv[i], &statbuf) )
        if (!S_ISDIR(statbuf.st_mode)) {
            ls_long(&statbuf, argv[i]);
            ok = true;
            EC_CLEANUP
        }
        ec_null( dir = opendir(argv[i]) )
        while (errno = 0, ((entry = readdir(dir)) != NULL)) {
            ec_negl( lstat(entry->d_name, &statbuf) )
            ls_long(&statbuf, entry->d_name);
        }
        ec_nzero( errno )
    }
    ok = true;
    EC_CLEANUP

    EC_CLEANUP_BGN
    if (dir != NULL)
        (void)closedir(dir);
    exit(ok ? EXIT_SUCCESS : EXIT_FAILURE);
    EC_CLEANUP_END
}
```

当用这个程序列出当前目录时，该程序运行良好，和结果显示的一样（仅显示了结果的头几行）：

```
$ aupls .
drwxr-xr-x  2 marc  marc      2560 Oct 18 12:20 .
drwxr-xr-x  9 marc  marc      512 Oct 14 18:05 ..
-rwxrwxrwx  1 marc  marc       55 Jul 25 11:14 m2
-rwxr--r--  1 marc  marc      138 Sep 19 13:28 k
```

但当试图在/tmp目录上运行时，得到的结果却是这样的：

```
$ aupls /tmp
drwxr-xr-x  2 marc  marc      2560 Oct 18 12:20 .
drwxr-xr-x  9 marc  marc      512 Oct 14 18:05 ..
ERROR: 0: main [/aup/c3/aupls.c:422] lstat(entry->d_name, &statbuf)
*** ENOENT (2: "No such file or directory") ***
```

症状是readdir循环调用lstat时找不到名字，即使从目录读取也不行。原因是试图带有路径“auplog.tmp”调用lstat时，如果/tmp在当前目录中，那么程序会运行良好，但问题是在不在。[⊖]当首先调用cd命令时，其工作的情况如下（仅显示头几行）：

```
$ cd /tmp
```

⊖ 仅列出了前两个记录项，因为.和..存在于每一个目录中。然而输出的结果仍然是不正确的。

```

$ /usr/home/marc/aup/aupls .
drwxrwxrwt   3 root   wheel   1536 Oct 18 12:20 .
drwxr-xr-x  18 root   wheel   512 Jul 25 08:01 ..
-rw-r--r--   1 marc   wheel  189741 Aug 30 11:22 auplog.tmp
-rw-----   1 marc   wheel    0 Aug  5 13:23 ed.7GHahk

```

补救的方法是在调用readir循环前使用chdir系统调用，下节将对此系统调用作以介绍。

3.6.2 chdir和fchdir系统调用

大家都知道shell程序中的cd命令的功能，下面是晚于它开发的系统调用：

chdir——根据路径改变当前目录

```

#include <unistd.h>

int chdir(
    const char *path          /* pathname */
);
/* Returns 0 on success or -1 on error (sets errno) */

```

fchdir——根据文件描述符改变当前目录

```

#include <unistd.h>

int fchdir(
    int fd                   /* file descriptor */
);
/* Returns 0 on success or -1 on error (sets errno) */

```

和大家希望的一样，chdir的参数可以是绝对路径，也可以是相对路径，并且无论它指向的信息节点是什么，如果是一个目录，那么它就能使这个目录变成新的当前目录。

fchdir把向目录打开的文件描述符当作参数，但等一下！3.6.1节中不是讲过打开目录是非标准的吗？我没有说过；我说过读取目录是非标准的。打开目录时确实有一个标准，那就是用fchdir得到可用的文件描述符，尽管必须带有O_RDONLY标志打开目录，但不允许为写操作打开。

既然不能方便地读取目录，为什么还需要fchdir——为什么不一直使用带路径名的chdir呢？因为fchdir和open是配对使用的，所以能够很好地标识所处的位置，并将其返回。比较这两种技术：

- | | |
|---------------------|------------------------|
| 1) 得到当前目录的路径名 | 1) 打开当前目录 |
| 2) 改变当前目录 | 2) 改变当前目录 |
| 3) chdir使用第一步得到的路径名 | 3) fchdir使用第一步得到的文件描述符 |

左边的技术较差，原因是：

- 不太容易得到当前目录的路径名，见3.4.2节。
- 该过程很耗时，在后面的3.6.4节中，当我们亲自做这项工作时就能明白这个问题了。

在某些情况下，如果仅运行到下一层目录，那么可以通过下面的代码返回到上一级目录：

```
ec_neg1( chdir("../") )
```

此时并不需要有路径名，但是最好还是使用open/fchdir函数对，因为该函数对不需要知道程序是如何遍历目录的。然而当用户对目录没有读权限时，是不能使用该函数对的。

现在修正上一节的aupls的版本，读取之前将其转到一个目录。我们确实需要返回到原来的地方，因为有多参数要处理，并且每个都假设未改变当前目录。下面是修改后的中间

部分:

```
ec_null( dir = opendir(argv[i]) )
ec_negl( fd = open(".*", O_RDONLY) )
ec_negl( chdir(argv[i]) )
while (errno = 0, ((entry = readdir(dir)) != NULL)) {
    ec_negl( lstat(entry->d_name, &statbuf) )
    ls_long(&statbuf, entry->d_name);
}
ec_nzero( errno )
ec_negl( fchdir(fd) )
```

但仍然有一个问题。如果你想试着自己找到这个问题，那么请不要往下读了。

问题是如果在`chdir`和`fchdir`调用之间出现错误跳转到清理代码，那么将不执行对`fchdir`的调用，并且不能恢复当前目录。在这个程序中出现这种错误并不碍事，因为这些错误将终止进程，并且对每一个进程来说，当前目录都是唯一的（不影响shell的当前目录）。但是，如果错误处理在某点被改变，或者如果把这段代码复制并粘贴到另一个程序中，那么情况就会变得很糟糕。

好的修复方法（这里没有列出代码）是在清理代码中第二次调用`fchdir`，从而无论出现任何情况，它都会执行。可以将`fd`初始化为-1，并检测该值，除非它是一个有效值时才使用它。

3.6.3 mkdir和rmdir系统调用

创建目录和删除目录的两个系统调用分别是：

mkdir——建立目录

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

int mkdir(
    const char *path,          /* pathname */
    mode_t perms              /* permissions */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

rmdir——删除目录

```
#include <unistd.h>

int rmdir(
    const char *path          /* pathname */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

对`mkdir`来说，访问权限、如何与文件创建屏蔽字交互以及新目录的所有权和`open`的选项都是相同的（见2.4节），并能自动创建特殊的`.`和`..`链接。

`rmdir`的功能和`unlink`类似，但`unlink`不能对目录进行操作。[⊖]一个很大的限制是要删除的目录必须是空的（除`.`和`..`之外）。如果目录非空，那么必须首先从分支树的底部开始删除链接，也许需要多次调用`unlink`和`rmdir`。如果这就是想做的工作，那么这样写更容易实现：

```
ec_negl( system("rm -rf somedir") )
```

因为`rm`知道如何遍历目录树。

⊖ 在一些系统上，虽然超级用户可以在目录上使用它，但是那是非标准的。

下面是上面所讨论的一个示例:

```
void rmdir_test(void)
{
    ec_negl( mkdir("somedir", PERM_DIRECTORY) )
    ec_negl( rmdir("somedir") )
    ec_negl( mkdir("somedir", PERM_DIRECTORY) )
    ec_negl( close(open("somedir/x", O_WRONLY | O_CREAT, PERM_FILE)) )
    ec_negl( system("ls -ld somedir; ls -l somedir") )
    if (rmdir("somedir") == -1)
        perror("Expected error");
    ec_negl( system("rm -rf somedir") )
    ec_negl( system("ls -ld somedir") )
    return;
}

EC_CLEANUP_BGN
    EC_FLUSH("rmdir_test");
EC_CLEANUP_END
}
```

在1.1.5节中已经介绍了PERM_DIRECTORY和PERM_FILE。创建文件的代码行有点古怪，但是当仅为了创建文件时，却很方便。唯一的缺点是，如果open失败，那么所报告的errno值将传给close（得到的参数是-1），而代码编写者就只能猜测为什么不能创建文件了。

执行此函数的结果如下:

```
drwx-----  2 marc    users          72 Oct 18 15:32 somedir
total 0
-rw-r--r--  1 marc    users           0 Oct 18 15:32 x
Expected error: Directory not empty
ls: somedir: No such file or directory
```

3.6.4 实现getcwd（向上遍历目录树）

说实话当想起3.4.2节的getcwd时就没有兴趣了，因为它面临着定义缓存大小的麻烦。现在的目的是实现它!

基本思想是从当前目录开始，得到其索引节号，接着到其父节点查找这个信息节点的记录项，从中得到子节点的名字。重复这个过程，直到不能上溯为止。

“不能上溯”的含义是什么？它的含义是或者:

```
chdir("../")
```

返回-1并带有ENOENT错误，或者执行成功，但是仍在同样的目录中（即在原目录中，没有动），这两种行为都有可能发生，两个的含义都表示在根目录。

当沿树向上走时，我们会把找到的路径成分累积添加到链表中，这样得到的链表的顶部是最新的（即最高层）记录项。因此如果当前目录的名字是grandchild，那么接下来找到父节点会是child，再然后找到的child的父节点会是parent，此时就找到根目录了，从而结束链接。如图3-4所示。



图3-4 路径名链接表

下面是每个链接表节点的结构和函数，该函数实现了创建新节点和将其放到链表头的功能：

```

struct pathlist_node {
    struct pathlist_node *c_next;
    char c_name[1]; /* flexible array */
};

static bool push_pathlist(struct pathlist_node **head, const char *name)
{
    struct pathlist_node *p;

    ec_null( p = malloc(sizeof(struct pathlist_node) + strlen(name)) )
    strcpy(p->c_name, name);
    p->c_next = *head;
    *head = p;
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

```

每个pathlist_node的大小都是可变的；结构中有足够的空间用以容纳终止字符串的NUL字节，但当分配节点时，必须为该名字增加此空间，可以参照malloc的参数。注意到在每个链接表的头端都放置了一个节点，这样遇到这个节点后，就可以以相反顺序查找链接表。这个顺序恰好是累积成路径名的顺序，如get_pathlist函数所示：

```

static char *get_pathlist(struct pathlist_node *head)
{
    struct pathlist_node *p;
    char *path;
    size_t total = 0;

    for (p = head; p != NULL; p = p->c_next)
        total += strlen(p->c_name) + 1;
    ec_null( path = malloc(total + 1) )
    path[0] = '\0';
    for (p = head; p != NULL; p = p->c_next) {
        strcat(path, "/");
        strcat(path, p->c_name);
    }
    return path;

EC_CLEANUP_BGN
    return NULL;
EC_CLEANUP_END
}

```

该函数遍历了两次链表，第一次只计算了所需路径的总空间（循环中“+1”是为了“/”，第二次才建立路径字符串。

最后路径处理函数释放链接表空间，大概会在get_pathlist函数之后调用：

```

static void free_pathlist(struct pathlist_node **head)
{
    struct pathlist_node *p, *p_next;

    for (p = *head; p != NULL; p = p_next) {
        p_next = p->c_next;

```

```

    free(p);
}
*head = NULL;
}

```

这里，使用p_next变量保存了指向下一个节点的指针，因为p本身一经释放将会变成无效变量。下面是创建图3-4所示链接表的代码：

```

struct pathlist_node *head = NULL;
char *path;

ec_false( push_pathlist(&head, "grandchild" ) )
ec_false( push_pathlist(&head, "child" ) )
ec_false( push_pathlist(&head, "parent" ) )
ec_null( path = get_pathlist(head) );
free_pathlist(&head);
printf( "%s\n", path );
free(path);

```

以下是输出结果：

```
/parent/child/grandchild
```

这种处理路径的方式最方便的特性是不需要预先为路径分配空间，就像调用标准getcwd函数一样，见3.4.2节。

现在讨论getcwdx，我们自己的getcwd版本。getcwdx是向上遍历树，在每一层中，一旦识别出一个记录项名是一个子节点的对应项，那么就调用push_pathlist，直到根节点。首先通过宏检测两个stat结构是否代表同一信息节点，方法是通过检测设备ID和索引符号来实现：

```

#define SAME_INODE(s1, s2) ((s1).st_dev == (s2).st_dev &&\
(s1).st_ino == (s2).st_ino)

```

下面将在getcwdx函数中两次调用该宏：

```

char *getcwdx(void)
{
    struct stat stat_child, stat_parent, stat_entry;
    DIR *sp = NULL;
    struct dirent *dp;
    struct pathlist_node *head = NULL;
    int dirfd = -1, rtn;
    char *path = NULL;

    ec_neg1( dirfd = open(".", O_RDONLY) )
    ec_neg1( lstat(".", &stat_child) )
    while (true) {
        ec_neg1( lstat("...", &stat_parent) )

        /* change to parent and detect root */
        if (((rtn = chdir("..")) == -1 && errno == ENOENT) ||
            SAME_INODE(stat_child, stat_parent)) {
            if (head == NULL)
                ec_false( push_pathlist(&head, "") )
            ec_null( path = get_pathlist(head) )
            EC_CLEANUP
        }
        ec_neg1( rtn )

        /* read directory looking for child */

```

```

ec_null( sp = opendir(".") )
while (errno == 0, (dp = readdir(sp)) != NULL) {
    ec_negl( lstat(dp->d_name, &stat_entry) )
    if (SAME_INODE(stat_child, stat_entry)) {
        ec_false( push_pathlist(&head, dp->d_name) )
        break;
    }
}
if (dp == NULL) {
    if (errno == 0)
        errno = ENOENT;
    EC_FAIL
}
stat_child = stat_parent;
}

EC_CLEANUP_BGN
if (sp != NULL)
    (void)closedir(sp);
if (dirfd != -1) {
    (void)fchdir(dirfd);
    (void)close(dirfd);
}
free_pathlist(&head);
return path;
EC_CLEANUP_END
}

```

该函数的解释如下:

- 使用open/fchdir技术得到和复位当前目录, 因为该函数遍历树时会改变当前目录。
- 在循环中, stat_child是试图得到的子节点的记录项的名字, stat_entry是从readdir得到的每个记录项, stat_parent是父节点, 当向上追溯时该父节点又会变成子节点。
- 在本节开始部分解释了如何检测是否到达根节点: chdir函数或者失败, 或者不做任何事情。
- 如果到达根节点时, 链接表为空, 则将空名字压入堆栈以使路径名为/。
- 在readdir循环中, 虽然已经跳过了非目录的记录项, 但是实际上不必这么做, 因为SAME_INODE检测既快又准确。

最后, 下面是一段使用上述函数实现的小程序, 该程序的功能类似于标准的pwd命令:

```

int main(void)
{
    char *path;
    ec_null( path = getcwdx() )
    printf("%s\n", path);
    free(path);
    exit(EXIT_SUCCESS);

    EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
    EC_CLEANUP_END
}

```

3.6.5 实现ftw (向下遍历目录树)

对于标准库函数ftw (“file tree walk”的缩写), 这里不做特别的描述, 该函数提供了一

个递归处理目录树中记录项的方法。给ftw一个指向函数的指针，遇到每个对象时都会调用这个函数，但是更有趣的是依据本章所学知识可以实现自己的目录树遍历函数。

第一件要弄清的事情是目录结构是否是真正的树，因为递归算法对此有要求。如果有循环，那么程序将会变成死循环，并且如果两次链接同一个目录（使用.和..之外的记录项），那么将会不只一次地访问同一个目录（及其分支）。需要考虑以下两类问题：

1) 符号链接链接到目录是很平常的，并且至少会创建两个链接，因为每个目录还有一个硬链接，但没有针对符号链接建立循环的保护措施。

2) 在一些系统上，超级用户通过link系统调用可以为目录创建第二个硬链接。虽然这种情况几乎不会出现，但程序要考虑它的可能性。

只要不跟随符号链接，就可以容易地解决第一类问题。尽管练习3.5涉及了第二类问题，但我们现在不谈它。因此，为了达到目的，通过硬链接形成的的确是个树。

我们希望aupls程序（3.5.3节中所述程序的扩展）能够表现出ls的功能，其中-R参数使其递归，-d参数告诉该函数仅列出目录的信息，而不是目录的内容。如果不带有这两个参数中的任何一个，那么aupls程序的功能就和以前的早期版本一样了。

带有-R参数时，aupls程序（如ls命令一样）将首先列出目录的路径，以及同层上的所有记录项，包括目录，接着对每个目录递归地做同样的工作，如下示例：

```
$ aupls -R /aup/common

/aup/common:
-rwxr-xr-x  1 root  root    1145 Oct  2 10:21 makefile
-rwxr-xr-x  1 root  root    171 Aug 23 10:41 logf.h
-rwxr-xr-x  1 root  root   1076 Aug 26 15:24 logf.c
drwxr-xr-x  1 root  root   4096 Oct  2 12:20 cf
-rwxr-xr-x  1 root  root    245 Aug 26 15:29 notes.txt

/aup/common/cf:
-rwxr-xr-x  1 root  root   1348 Oct  3 13:52 cf-ec.c-ec_push.htm
-rwxr-xr-x  1 root  root    576 Oct  3 13:52 cf-ec.c-ec_print.htm
-rwxr-xr-x  1 root  root    450 Oct  3 13:52 cf-ec.c-ec_reinit.htm
-rwxr-xr-x  1 root  root    120 Oct  3 13:52 cf-ec.c-ec_warn.htm
```

从代码中你将看到每级（一个readdir循环）使用两个传递，对于每个目录第一个传递（PASS1）输出“stat”的信息，然后在第二个传递（PASS2）中，接着我们利用此目录中的PASS1进行递归，以此类推。

在这个程序中，以结构来表现传递给每一个函数的遍历信息比使用全局变量或者长参数列表方便。

```
typedef enum {SHOW_PATH, SHOW_INFO} SHOW_OP;

struct traverse_info {
    bool ti_recursive;                /* -R option? */
    char *ti_name;                    /* current entry */
    struct stat ti_stat;               /* stat for ti_name */
    bool (*ti_fcn)(struct traverse_info *, SHOW_OP); /* callback fcn */
};
```

下面是要使用的回调函数：

```
static bool show_stat(struct traverse_info *p, SHOW_OP op)
{
    switch (op) {
```

```

    case SHOW_PATH:
        ec_false( print_cwd(false) )
        break;
    case SHOW_INFO:
        ls_long(&p->ti_stat, p->ti_name);
    }
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

```

把op设置成SHOW_PATH调用show_stat时，输出的是路径名的头；把op设置成SHOW_INFO调用show_stat时，输出的是明细行。ls_long来自于3.5.3节，print_cwd的代码与3.4.2节的几乎完全相同：

```

static bool print_cwd(bool cleanup)
{
    char *cwd;

    if (cleanup)
        (void)get_cwd(true);
    else {
        ec_null( cwd = get_cwd(false) )
        printf("\n%s:\n", cwd);
    }
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

```

接下来在总清理代码中将会看一个print_cwd(true)调用。

现在跳转到main函数的顶层来观察程序是如何初始化的，列表是如何开始工作的：

```

#define USAGE "Usage: aupls [-Rd] [dir]\n"

static long total_entries = 0, total_dirs = 0;

int main(int argc, char *argv[])
{
    struct traverse_info ti = (0);
    int c, status = EXIT_FAILURE;
    bool stat_only = false;

    ti.ti_fcn = show_stat;
    while ((c = getopt(argc, argv, "dR")) != -1)
        switch(c) {
            case 'd':
                stat_only = true;
                break;
            case 'R':
                ti.ti_recursive = true;
                break;
            default:
                fprintf(stderr, USAGE);
                EC_CLEANUP
        }
}

```



```

switch (argc - optind) {
case 0:
    ti.ti_name = ".";
    break;
case 1:
    ti.ti_name = argv[optind];
    break;
default:
    fprintf(stderr, USAGE);
    EC_CLEANUP
}
ec_false( do_entry(&ti, stat_only) )
printf("\nTotal entries: %ld; directories = %ld\n", total_entries,
total_dirs);
status = EXIT_SUCCESS;
EC_CLEANUP
}

EC_CLEANUP_BGN
    print_cwd(true);
    exit(status);
EC_CLEANUP_END
}

```

全局变量`total_entries`和`total_dirs`用于保存总的计数，有趣的是在程序的最后显示了它们的内容，立刻就可以看到它们在何处增加。

`getopt`是一个标准库函数（POSIX/SUS的组成部分，不是标准C的内容）。其第三个参数是一个可允许选项的字母列表。它完成选项后，会将全局变量`optind`设置为需要处理的下一个参数的索引，在例子中是一个可选的路径名。如果没有给定任何参数，则假定为当前目录。

`do_entry`完成的实际工作如下：

```

static bool do_entry(struct traverse_info *p, bool stat_only)
{
    bool is_dir;

    ec_negl( lstat(p->ti_name, &p->ti_stat) )
    is_dir = S_ISDIR(p->ti_stat.st_mode);
    if (stat_only) {
        total_entries++;
        if (is_dir)
            total_dirs++;
        ec_false( p->ti_fcn)(p, SHOW_INFO) )
    }
    else if (is_dir)
        ec_false( do_dir(p) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

```

可以按以下两种方式之一使用`do_entry`：如果`stat_only`参数为`true`或者当前记录项不是目录，就增加全局计数器，然后以`SHOW_INFO`模式调用回调函数。只有当规定了`-d`参数以及在`aupls`命令行或者在目录列表的`PASS1`中规定了非目录时才这么做，否则，将调用包含`readdir`循环的函数`do_dir`处理目录记录项：

```

static bool do_dir(struct traverse_info *p)
{
    DIR *sp = NULL;
    struct dirent *dp;
    int dirfd = -1;
    bool result = false;

    ec_negl( dirfd = open(".", O_RDONLY) )
    if ((sp = opendir(p->ti_name)) == NULL || chdir(p->ti_name) == -1) {
        if (errno == EACCES) {
            fprintf(stderr, "%s: Permission denied.\n", p->ti_name);
            result = true;
            EC_CLEANUP
        }
        EC_FAIL
    }
    if (p->ti_recursive)
        ec_false( (p->ti_fcn)(p, SHOW_PATH) )
    while (errno = 0, ((dp = readdir(sp)) != NULL)) {
        if (strcmp(dp->d_name, ".") == 0 ||
            strcmp(dp->d_name, "..") == 0)
            continue;
        p->ti_name = dp->d_name;
        ec_false( do_entry(p, true) )
    }
    if (errno != 0)
        syserr_print("Reading directory (Pass 1)");
    if (p->ti_recursive) {
        rewinddir(sp);
        while (errno = 0, ((dp = readdir(sp)) != NULL)) {
            if (strcmp(dp->d_name, ".") == 0 ||
                strcmp(dp->d_name, "..") == 0)
                continue;
            p->ti_name = dp->d_name;
            ec_false( do_entry(p, false) )
        }
        if (errno != 0)
            syserr_print("Reading directory (Pass 2)");
    }
    result = true;
    EC_CLEANUP
}
EC_CLEANUP_BGN
if (dirfd != -1) {
    (void)fchdir(dirfd);
    (void)close(dirfd);
}
if (sp != NULL)
    (void)closedir(sp);
return result;
EC_CLEANUP_END
}

```

do_dir是递归发生的地方。它向当前目录打开dirfd，以便在清理代码中可以恢复。然后用opendir打开目录，并对其进行修改以便能处理相对于父目录的目录记录项。出现EACCES错误是很平常的，因为目录是不可读的（opendir失败），或者不可查找的（chdir失败），所以出现这些情况时仅需要输出信息，而不用终止处理。

接下来，如果设定了-R参数，那么告诉回调函数输出当前目录的路径，这表示可以把所有的常规输出都定位到一个回调函数。

然后进行PASS1的readdir循环（其中调用了do_entry函数，并将stat_only设置成了true），如果设定了-R参数，则rewinddir系统调用将进行PASS2的readdir循环（其中调用了do_entry函数，并将stat_only设置成了false）。因为PASS2的do_entry可能会递归调用do_dir，因此产生递归。注意两者的readdir循环都跳过了.和..记录项，默认情况下ls命令也忽略。

对于来自readdir的错误，我们决定把它当作非致命的，继续运行程序，而不是用ec_nzero宏跳出程序，该宏解释了对syserr_print的两次调用，其中syserr_print的代码如下：

```
void syserr_print(const char *msg)
{
    char buf[200];

    fprintf(stderr, "ERROR: %s\n", syserrmsg(buf, sizeof(buf), msg,
        errno, EC_ERRNO));
}
```

syserrmsg见1.4.1节。

这就是我们自己开发的ls的完整的递归版本！

3.7 改变信息节点

系统调用的stat族（见3.5.1节）是从信息节点中检索信息的，介绍这一族时还解释了在操作文件时信息节点中的各种数据字段是如何改变的。其中的一部分字段也可以通过本节讨论的系统调用直接更改。表3-3列出了能实现此功能的系统调用。符号“fixed”是指字段是根本不可更改的（如果想改变该字段的内容，则必须创建一个新的信息节点），符号“side-effect”是指仅在操作其他函数而间接影响到该字段时，该字段才是可以更改的，例如使用link函数创建了一个新链接，但这个新链接是不能直接更改的。

表3-3 改变信息节点字段

信息节点字段	描述	改变属性
st_dev	文件系统的设备ID	fixed
st_ino	索引符号	fixed
st_mode	模式	chmod, fchmod
st_nlink	硬链接数	side-effect
st_uid	用户ID	chown, fchown, lchown
st_gid	组ID	chown, fchown, lchown
st_rdev	设备ID（如果是特殊文件）	fixed
st_size	字节数	side-effect
st_atime	最后存取时间	utime
st_mtime	最后数据修改时间	utime
st_ctime	最后信息节点修改时间	side-effect
st_blksize	最佳I/O大小	fixed
st_blocks	分配的512字节块	side-effect

3.7.1 chmod和fchmod系统调用

chmod——根据路径改变文件模式

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

int chmod(
    const char *path,      /* pathname */
    mode_t mode           /* new mode */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

fchmod——根据文件描述符改变文件模式

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

int fchmod(
    int fd,                /* file descriptor */
    mode_t mode           /* new mode */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

chmod系统调用可以改变已存在的任何文件的模式。但是它并不能改变文件类型本身，而是仅改变S_ISUID、S_ISGID和S_ISVTX标志或者权限位。要如stat结构规定的那样使用这些宏（见3.5.1节）。fchmod的功能与chmod相似，但它利用的仅是打开的文件描述符而不是路径。

调用者的有效用户ID必须与文件的用户ID匹配，或者调用者必须是超级用户。仅有写权限或者仅满足调用者的有效组ID与文件的用户ID匹配是不够的。另外，如果设置了S_ISGID，那么有效的组ID也必须匹配（除了超级用户）。

除非打算设置整个模式，否则首先必须调用某个stat函数得到已经存在的模式，接着设置或清除需要改变的位，最后执行chmod修改模式。

在程序中不常调用chmod，因为创建文件时通常已设置了模式。然而用户常常使用chmod命令。

3.7.2 chown、fchown和lchown系统调用

chown——根据路径改变文件的所有者和文件的组

```
#include <unistd.h>

int chown(
    const char *path,      /* pathname */
    uid_t uid,            /* new user ID */
    gid_t gid             /* new group ID */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

fchown——根据文件描述符改变文件的所有者和文件的组

```
#include <unistd.h>

int fchown(
    int fd,                /* file descriptor */
    uid_t uid,            /* new user ID */
    gid_t gid             /* new group ID */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

lchown —— 通过路径改变符号链接的所有者和组

```
#include <unistd.h>

int lchown(
    const char *path,      /* pathname */
    uid_t uid,            /* user ID */
    gid_t gid             /* group ID */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

chown可以改变文件的用户ID和组ID。只有所有者（进程的有效用户ID与文件的用户ID相同）或者超级用户才能执行它。如果uid或者gid为-1，将不处理对应的ID。

fchown与之类似，但利用的是打开文件描述符而不是路径。lchown也相似，但它是直接作用于路径参数的，如果是符号链接，那么它并不作用于符号链接所指向的内容。

除非调用者是超级用户，否则这些系统调用将清除设置用户ID位和设置组ID位，这是为了禁止十分明显的非法闯入：

```
$ cp /bin/sh mysh          [get a personal copy of the shell]
$ chmod 4700 mysh         [turn on the set-user-ID bit]
$ chown root mysh        [make the superuser the owner]
$ my sh                   [become superuser]
```

如果想使用自己的超级用户shell，则必须颠倒chmod和chown的顺序，但是除非已经是超级用户，否则是不允许执行chmod的。这样做就堵住了漏洞。

如果设置了宏_POSIX_CHOWN_RESTRICTED（使用pathconf和fpathconf检测），则要对某些UNIX系统进行配置，以便用略有不同的规则进行操作。

只有超级用户可以改变所有者（用户ID），但是所有者可以将组ID改变为进程的有效组ID或者所提供的组ID中的一个。换句话说，所有者不能放弃文件——最多可以将组ID改成与进程相关的组ID中的一个。

因为通常可以通过chown命令（调用chown系统调用）来更改所有权，所以这些规则通常不会直接影响应用程序。但它们会影响系统管理及用户使用系统的方式。

3.7.3 utime系统调用

utime —— 设置文件访问时间和修改时间

```
#include <utime.h>

int utime(
    const char *path,      /* pathname */
    const struct utimbuf *timbuf /* new times */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

struct utimbuf —— utime的结构

```
struct utimbuf {
    time_t actime;        /* access time */
    time_t modtime;      /* modification time */
};
```

utime可以改变任何类型文件的访问时间和修改时间。[⊖]结构中的time_t类型是纪元以

⊖ 在一些系统上，有一个类似的系统调用utimes，但它已经过时了。

来的秒数，和1.7.1节中定义的一样。只有超级用户或所有者才能通过`timbuf`参数改变时间。

如果`timbuf`参数等于`NULL`，则访问时间和修改时间被设置为当前时间。这样做可以强制文件更新到当前时间，而不需要重写文件，但重要的是为了`touch`命令的利益。任何具有写权限的用户都可以在允许写操作的文件上做这个事情，因为写操作也会改变文件的时间。

除了`touch`外，当从清除磁带中恢复文件和通过网络接收文件时，也常常使用`utime`，将时间复位为伴随文件而来的初始值。因为信息节点没有移动到其他位置——新创建的位置，所以不能复位状态改变时间是合适的。

3.8 其他的文件处理调用

本节将讨论其他文件处理系统调用，它们不适合前面章节的内容。

3.8.1 access系统调用

与处理权限位的其他系统调用不同，`access`检测的是实际用户ID或者组ID，而不是有效ID。

access——确认文件的访问

```
#include <unistd.h>

int access(
    const char *path,      /* pathname */
    int what               /* permission to be tested */
);
/* Returns 0 if allowed or -1 if not or on error (sets errno) */
```

参数`what`使用下列标志，其中头三个标志可以一起执行或操作：

```
R_OK          /* read permission */
W_OK          /* write permission */
X_OK          /* execute (search) permission */
F_OK          /* test for existence */
```

如果进程的实际用户ID与其路径的匹配，则检测所有者权限位；如果进程的实际组ID与其路径的匹配，则检测组权限位；如果两者都不是，则检测其他的权限位。

`access`主要有以下两个用途：

- 检测实际用户或者组对文件是否有操作权限，以防设置了设置用户ID位或者设置组ID位。例如，在执行过程中，可能会有一个命令将用户ID设置为超级用户，但是做这个工作之前，需要检测实际用户是否有把文件从目录断开的权限。不能只看`unlink`失败时是否带有`EACCESS`错误，因为当有效用户ID是超级用户时，`unlink`不会再因为那种原因而失败。
- 检测文件是否存在。尽管可以使用`stat`，但是`access`更简单。（实际上，整个`access`系统调用可以写作调用`stat`的一个库函数，在某些实现中，甚至可能已经实现了那种方式。）

如果`path`是符号链接，那么跟随此链接，直到发现一个非符号链接。[⊖]

调用`access`时，可能不想把它放在`ec_neg1`宏中，因为，当返回值为-1时，需要从其他错误中区分是`EACCESS`错误（对于`R_OK`宏、`W_OK`宏、和/或`X_OK`宏）还是`ENOENT`错误（对于`F_OK`宏）。采用如下的方式：

[⊖] 实际情况是，除非以字母I开始，几乎所有的系统调用都以路径名为参数。`unlink`和`rename`是两个例外。

```

if (access("tmp", F_OK) == 0)
    printf("Exists\n");
else if (errno == ENOENT)
    printf("Does not exist\n");
else
    EC_FAIL

```

3.8.2 mknod系统调用

mknod——建立文件

```

#include <sys/stat.h>

int mknod(
    const char *path,          /* pathname */
    mode_t perms,             /* mode */
    dev_t dev,                 /* device-ID */
);
/* Returns 0 on success or -1 on error (sets errno) */

```

mknod可以创建普通文件、目录、特殊文件或者命名管道（FIFO）。如果没有mkfifo的话（见7.2.1节），那么唯一的优点（唯一不需要是超级用户的）是创建命名管道。创建普通文件（可以使用open创建）或者目录（可以使用mkdir创建）不需要它，也不能用它来创建符号链接（可以使用symlink创建）或者套接字（可以使用bind创建）。

因此，mknod的主要用途是创建特殊文件，通常在/dev目录中，该特殊文件常用于访问设备。因为通常只有当安装新设备驱动器时，才使用mknod命令，它会执行该系统调用。

perms参数和3.5.1节定义的stat结构使用相同的位和宏，那里dev被定义为设备ID。如果安装了新设备驱动器，就可以知道该设备ID是什么了。

3.8.3 fcntl系统调用

在2.2.3节中（在继续向下看之前你可能需要重新读一下），当时解释说几个打开文件描述符能共享同一个打开文件描述，该文件描述含有文件偏移量、状态标志（例如O_APPEND）以及访问模式（例如O_RDONLY）。通常，可以在打开文件时设置文件的状态标志和访问模式，但是也可以通过fcntl系统调用在任何时间得到和设置状态标志，并可以用它得到（但不设置）访问模式。

fcntl——控制打开的文件

```

#include <unistd.h>
#include <fcntl.h>

int fcntl(
    int fd,                    /* file descriptor */
    int op,                    /* operation */
    ...                        /* optional argument depending on op */
);
/* Returns result depending on op or -1 on error (sets errno) */

```

总共有10种操作，但这里仅打算介绍其中的4种，其他的操作将在其他地方介绍，这10种操作的说明如表3-4所示。

表3-4 fcntl操作

操 作	用 途	所在章节
F_DUPFD	复制文件描述符	6.3节
F_GETFD	得到文件描述符标志	本节
F_SETFD	设置文件描述符标志 (使用第三个int参数)	本节
F_GETFL	得到文件描述符状态标志和访问模式	本节
F_SETFL	设置文件描述符状态标志 (使用第三个int参数)	本节
F_GETOWN	和套接字一起使用*	8.7节
F_SETOWN	和套接字一起使用*	8.7节
F_GETLK	得到一个锁	7.11.4节
F_SETLK	设置或者清除一个锁	7.11.4节
F_SETLKW	设置或者清除一个锁	7.11.4节

* 因为太复杂, 而无法在此概括。

对于所有“get”和“set”操作, 应该首先得到当前值, 设置或清除需要修改的标志, 然后才能进行设置, 即使只是定义一个标志。即使以后要增加多个标志, 或者不依赖于实现并且有不了解的非标准参数, 用这种方法实现的代码也仍然有效。例如, 这样设置O_APPEND标志是错误的:

```
ec_negl( fcntl(fd, F_SETFL, O_APPEND) ) /* wrong */
```

正确的方法是:

```
ec_negl( flags = fcntl(fd, F_GETFL) )
ec_negl( fcntl(fd, F_SETFL, flags | O_APPEND) )
```

如果需要清除O_APPEND标志, 则第二行代码可以这样改:

```
ec_negl( fcntl(fd, F_SETFL, flags & ~O_APPEND) )
```

(规则是: 用或操作来实现设置, 用与操作来实现清除。)

唯一定义的标准文件描述符标志是执行关闭(close-on-exec)标志FD_CLOEXEC, 该标志表示: 在执行exec系统调用时, 是否会关闭文件描述符。5.3节中对执行关闭将进行更详细的介绍, 对其可以进行F_GETFD和F_SETFD操作。这是fcntl的重要用途, 因为打开文件时没有设置该参数。

对文件描述符状态标志和访问模式使用F_GETFL和F_SETFL操作, 访问模式可以使用O_RDONLY、O_WRONLY或者O_RDWR三个标志中的一个, 但只能获得不能设置。因为这些都是标志值, 不是位掩码, 所以必须用O_ACCMODE从返回值中取出这些标志, 像这样:

```
ec_negl( flags = fcntl(fd, F_GETFL) )
if ((flags & O_ACCMODE) == O_RDONLY)
    /* file is opened read-only */
```

下面两个if语句都是错误的:

```
if (flags & O_RDONLY) /* wrong */
if ((flags & O_RDONLY) == O_RDONLY) /* still wrong */
```

在2.4.4节表2-1中所列的状态标志更有意义。可以得到和设置O_APPEND、O_DSYNC、O_NOCTTY、O_NONBLOCK、O_RSYNC和O_SYNC标志; 可以得到O_CREAT、O_TRUNC和O_EXCL的标志值, 但设置这些标志的值是没有意义的, 因为这些标志只对open系统调用起作用(调用函数后再设置这些参数就迟了)。前面给出过一个使用O_APPEND标志的例子。

在4.2.2节和7.2节中还有几个关于O_NONBLOCK标志的例子。

3.9 异步I/O

本节将介绍如何初始化I/O操作，以便程序不必等待I/O工作完成，就可以离开去做其他事情，并在随后的时间中检测I/O操作的状态。

3.9.1 进一步比较已同步的与同步的

在阅读本节之前，一定要先阅读2.16.1节，那一节中解释了已同步的（与非已同步的对比）。本节要讨论的是异步I/O，即系统调用（例如aio_read）初始化I/O操作后，在I/O操作完成之前该系统调用已经返回，完成工作是分别进行的。也就是说：

- 已同步的是指I/O操作与物理I/O是同时完成的。非已同步的是指进程和缓存之间的I/O完成就可以了。
- 同步的是指仅当I/O完成后，系统调用才返回，如前面段落定义的一样。异步的是指初始化I/O完成之后系统调用立刻返回，然后用其他系统调用来检测I/O操作是否完成。

一句话就是：同步的/异步的是指调用是否需要等待I/O完成，已同步的/非已同步的强调的是“完成”了什么。

我们假设读者对信号的工作原理已经有所了解，这里阅读本节内容的前提，如果还不了解，那么在阅读本节之前请阅读第9章，或者至少参照第9章的部分内容，特别是9.5.6节。

如2.9节讲述的，在UNIX中大部分write操作某种程度上都已经是异步操作了，因为数据被传递给缓存后系统调用都会立即返回，之后才完成实际的输出。但是如果设置了O_SYNC或者O_DSYNC标志（见2.16.3节），则write操作要到完成了实际输出之后才返回。相反，一般情况下read操作通常包含等待实际输入，因为直到物理上读取了数据，才能进行read操作，除非进行了预读取或者数据恰好是在缓冲区中。

对于异步I/O来说（AIO），进程不必等待read，或者已同步的（设置了O_SYNC或者O_DSYNC标志的）write。初始化I/O操作后，AIO调用会立刻返回，随后进程可以调用aio_error来检测操作是否已经完成，或者通过信号或创建新线程（见5.17节）通知它操作什么时间完成。

这里使用的“已经完成”仅指read和write所操作的，并不是指I/O是已同步的，除非将文件描述符设置为同步，如2.16.3节解释的那样。因此有4种情况，见表3-5。

表3-5 已同步的与同步的读写操作比较

	同步的	异步的
非已同步的	read/write:清除O_SYNC和O_DSYNC	aio_read/aio_write:清除O_SYNC和O_DSYNC
已同步的	read/write:设置O_SYNC或O_DSYNC	aio_read/aio_write:设置O_SYNC或O_DSYNC

如果可以对应用程序进行组织，以便应用程序在需要数据之前可以初始化read，随后还可以进行其他有用的工作，那么无论read是已同步的还是非已同步的，AIO都可以增加其性能。但是如果在此期间进程没有其他工作可以做，那么也可以阻塞read，让另一个进程或者线程运行。AIO对已同步的write有用，对非已同步的write没有太大用处，因为缓存已经做了同样的事情。

不要将异步I/O与通过O_NONBLOCK标志设置的非阻塞I/O混淆（见2.4.4节和4.2.2节）。非

阻塞是指如果阻塞，则调用返回，但它不做工作。[⊖]异步是指初始完成后则返回。

AIO函数是异步输入/输出选项的一部分，如_POSIX_ASYNCHRONOUS_IO宏代表的内容。可以在运行时通过pathconf来检测，1.5.4节给出了完成这项工作的示例代码。

本节中的一个系统调用lio_listio既可以用于同步I/O操作，也可以用于异步I/O操作。

3.9.2 AIO控制块

所有的AIO系统调用都使用控制块来跟踪操作的状态，不同的操作必须使用不同的控制块。当然，操作完成时，控制块可以被重用。

struct aiocb —— AIO控制块

```
struct aiocb {
    int aio_fildes;           /* file descriptor */
    off_t aio_offset;        /* file offset */
    volatile void *aio_buf;  /* buffer */
    size_t aio_nbytes;       /* size of transfer */
    int aio_reqprio;         /* request priority offset */
    struct sigevent aio_sigevent; /* signal information */
    int aio_lio_opcode;      /* operation to be performed */
};
```

前4个成员与pread和pwrite中的相似（见2.14节），也就是说，包括read和write的三个参数以及隐含的lseek的文件偏移量。

在9.5.6节中详细解释了sigevent结构，当操作完成时，可以利用该结构重新产生信号，或者开始一个线程。可以向信号句柄或线程传递任意一个整数或者指针值，通常这将是一个指向控制块的指针。如果不需要信号或者线程，那么可以将成员aio_sigevent.sigev_notify设置成SIGEV_NONE。

aio_reqprio成员用于影响操作的优先权，并且只有在支持其他两个POSIX选项（_POSIX_PRIORITIZED_IO和_POSIX_PRIORITY_SCHEDULING）时，才可以使用该成员。关于该成员的更多特征，请参阅[SUS2002]。

最后一个成员aio_lio_opcode是与lio_listio系统调用一起使用的，具体解释见3.9.9节。

3.9.3 aio_read和aio_write

基本的AIO函数是aio_read和aio_write。与pread和pwrite一样，aio_offset成员决定着文件中I/O发生的位置。当文件描述符打开了一个不允许检索的设备（例如套接字），或者对aio_write设置了O_APPEND标志（见2.8节）时，它是无效的。缓冲区和大小位于控制块中，不是作为参数传递的。

aio_read —— 从文件中异步读

```
#include <aio.h>

int aio_read(
    struct aiocb *aiocbp    /* control block */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

⊖ connect是例外；见8.12节。

aio_write——异步写入文件

```
#include <aio.h>

int aio_write(
    struct aiocb *aiocbp    /* control block */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

从这些函数成功返回仅仅意味着操作已被初始化，当然并不意味着I/O是成功的，也不意味着控制块设置的值是对的。总是需要通过（下一节介绍）来检测结果。执行程序也许会立刻报告像无效偏移量那样的错误，由或者返回值-1，或者返回值0并随后报告坏消息。

3.9.4 aio_error和aio_return**aio_error**——为异步I/O操作检索错误状态

```
#include <aio.h>

int aio_error(
    const struct aiocb *aiocbp    /* control block */
);
/* Returns 0, errno value, or EINPROGRESS (does not set errno) */
```

如果操作完成，即操作是成功的，则会返回0，或者返回errno值，该errno值与read、write、fsync或fdatasync可能会返回的值相同（见2.9节、2.10节和2.16.2节）。如果操作没有完成，则返回EINPROGRESS。如果知道操作已经完成（例如得到了通知已完成的信号），则可以像对待其他错误一样对待EINPROGRESS，并使用ec_rv宏，如3.6.1节介绍的那样：

```
ec_rv( aio_error(aiocbp) )
```

尽管如此，还是要提及一下，异步I/O调用通常都被用于相当高级的应用程序，在这种情况下，简单的“ec”错误检测可能就不再适用了，如1.4.2节提到的。但在开发过程中，ec_rv仍是有用的，因为它提供了函数调用跟踪的能力。

即使aio_error报告了操作是成功的，也还是需要从等价的read或write函数中得到实际传输字节数的返回值时，可以利用aio_return：

aio_return——检索异步I/O操作的返回状态

```
#include <aio.h>

ssize_t aio_return(
    struct aiocb *aiocbp    /* control block */
);
/* Returns operation return value or -1 on error (sets errno) */
```

只有当aio_error报告成功的情况下，才可以调用aio_return。当aio_return返回-1时是指调用出错，而不是指该操作返回了错误。同样，在每次操作中只能调用一次aio_return，因为检索完毕后，系统将立即丢弃返回值。

3.9.5 aio_cancel

可以通过函数取消未完成的异步操作:

aio_cancel ——取消异步I/O请求

```
#include <aio.h>

int aio_cancel(
    int fd,                /* file descriptor */
    struct aiocb *aiocbp  /* control block */
);
/* Returns result code or -1 on error (sets errno) */
```

如果参数是NULL, 则该调用会尝试取消文件描述符fd上的所有异步操作。没有取消掉的操作仍会按照正常方式报告完成情况, 但已取消的操作会通过报告错误代码ECANCELED。

如果参数不是NULL, 则仅会尝试取消以这个控制块开始的操作。在这种情况下, fd参数必须与控制块中的aio_fildes成员相同。

如果成功, 则返回下面结果代码中的一个:

AIO_CANCELED 取消了所有的请求操作。

AIO_NOTCANCELED 因为请求的操作已经进行, 所以其中的一个或多个 (也许是所有的) 操作不能被取消。必须在每个操作上调用来找出哪些操作被取消了。

AIO_ALLDONE 因为所有操作都已经完成, 所以没有一个请求操作被取消。

3.9.6 aio_fsync

除了read和write的等价物之外, 还有第三种可以异步工作的I/O: 缓存的刷新, 可以利用fsync或fdatasync来实现 (2.16.2节介绍了两者的区别)。这两种刷新方式由同一调用控制:

aio_fsync ——为某个文件初始化缓存刷新

```
#include <aio.h>

int aio_fsync(
    int op,                /* O_SYNC or O_DSYNC */
    struct aiocb *aiocbp  /* control block */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

与其他调用相比, 控制块的使用有点不同: 不仅仅是刷新那些以控制块开始的、与操作相关的缓存, 而且要刷新所有由aio_fildes成员给定的、与文件描述符相关的缓存, 即使仅有一个缓存。同步的请求 (如刷新缓冲区) 是异步的: 同步没有立刻发生, 仅仅是被初始化了, 可以采用普通方式检测其是否完成 (如使用aio_error或信号)。因此, 在使用aio_read和aio_write时, 返回值为0仅表示初始化成功了。

3.9.7 aio_suspend

当I/O完成时, 不用通过信号或者线程异步通知, 而是可以通过简单地等待其完成来实现同步:

aio_suspend——等待异步I/O操作请求

```
#include <aio.h>

int aio_suspend(
    const struct aiocb *const list[], /* array of control blocks */
    int cbcnt, /* number of elements in array */
    const struct timespec *timeout /* max time to wait */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

如果timeout为NULL，那么aio_suspend会取得一个cbcnt控制块的数组并阻塞，直到其中的一个控制块完成。每一个块都必须已经用于初始化异步I/O操作。在调用期间，只要有一个块已经完成，aio_suspend就会立即返回。

为了重新使用同一个数组，可以将list中的一个元素设置为NULL，但是这个元素应该是包含在cbcnt中。

如果timeout不为NULL，则aio_suspend至多阻塞那段时间，然后返回-1，并把errno设置成EAGAIN。timespec的结构见1.7.2节。

像其他大多数阻塞系统调用一样，信号能中断aio_suspend，有关的进一步解释见9.1.4节。这又产生了一个很棘手的问题：根据如何创建控制块，aio_suspend等待的结果是产生一个信号，该信号可能会造成aio_suspend中断，这又造成返回值-1，并将error设置为EINTR。在某些情况下，返回值没有错误，但问题是：EINTR返回值可能是由其他某个信号引起的，而不正好是完成的I/O请求引起的。按照以下两种方式之一，可以避免异步完成信号中断aio_suspend：

- 利用信号指出已完成或调用aio_suspend，但只能用一种。
- 为信号设置SA_RESTART标志（见9.16节）。

或者是，让信号中断aio_suspend，接着为传递给aio_suspend列表的每个元素调用aio_error，如果有事情发生，当其返回时可以查看发生了什么事情，如果没有事情发生，则重新发出aio_suspend（也许在循环中）。

3.9.8 比较同步I/O和异步I/O的示例

本节将举例说明如何使用AIO系统调用，并列举一些优于同步I/O的优点。

首先，下面给出一个名为sio的程序，该程序使用传统的同步I/O方式读取文件。每读8000次，该程序也可以读取标准输入。

```
#define PATH "/aup/c3/datafile.txt"
#define FREQ 8000

static void synchronous(void)
{
    int fd, count = 0;
    ssize_t nread;
    char buf1[512], buf2[512];

    ec_negl( fd = open(PATH, O_RDONLY) )
    timestart();
    while (true) {
        ec_negl( nread = read(fd, buf1, sizeof(buf1)) )
        if (nread == 0)
            break;
    }
}
```



```

    if (count % FREQ == 0)
        ec_negl( read(STDIN_FILENO, buf2, sizeof(buf2)) );
    count++;
}
timestop("synchronous");
printf("read %d blocks\n", count);
return;

EC_CLEANUP_BGN
    EC_FLUSH("synchronous")
EC_CLEANUP_END
}

```

(timestart和timestop见1.7.2节)

标准输入与管道是相连的，管道是通过名为feed的程序填充的：

```
$ feed | sio
```

feed是一个很难处理的写操作——每20秒才完成一次写操作：

```

int main(void)
{
    char buf[512];

    memset(buf, 'x', sizeof(buf));
    while (true) {
        sleep(20);
        write(STDOUT_FILENO, buf, sizeof(buf));
    }
}

```

这种利用管道方式实现的写操作，sio花费了0.12秒用户CPU时间，0.73秒系统CPU时间，从开始运行到完成总共花费了200.05秒。显然其中很多时间都浪费在了等待管道输入上。

这是个人为的例子，但是尽管如此，却可以通过异步I/O来减少所花费的时间。如果可以异步读取管道，则在此期间就可以读文件，重写代码如下：

```

static void asynchronous(void)
{
    int fd, count = 0;
    ssize_t nread;
    char buf1[512], buf2[512];
    struct aiocb cb;
    const struct aiocb *list[1] = { &cb };
    memset(&cb, 0, sizeof(cb));
    cb.aio_fildes = STDIN_FILENO;
    cb.aio_buf = buf2;
    cb.aio_nbytes = sizeof(buf2);
    cb.aio_sigevent.sigev_notify = SIGEV_NONE;
    ec_negl( fd = open(PATH, O_RDONLY) );
    timestart();
    while (true) {
        ec_negl( nread = read(fd, buf1, sizeof(buf1)) );
        if (nread == 0)
            break;
        if (count % FREQ == 0) {
            if (count > 1) {
                ec_negl( aio_suspend(list, 1, NULL) );
                ec_rv( aio_error(&cb) );
            }
            ec_negl( aio_read(&cb) );
        }
    }
}

```



```

    }
    count++;
}
timestop("asynchronous");
printf("read %d blocks\n", count);
return;

EC_CLEANUP_BGN
    EC_FLUSH("asynchronous")
EC_CLEANUP_END
}

```

注意控制块的建立方法：首先将其清零，以保证其他任何依赖于实现的成员都为0，每读8000次（前面定义的FREQ），程序调用aio_read，接着在aio_suspend中等待控制块为下一次调用做准备。然而当aio_read工作时，它会继续读文件，再不必为8000多次的读挂起。这次因为用户这段时间有很多工作要做，所以用户CPU所花费的时间变长——0.18秒。系统时间基本没变（0.7秒），但是总的花费时间却减少到了180.58秒。

使用AIO时，可能总是看不到其太大的优越性，但是有时可以发现一个比这个例子更好的示例，关键是：

- 当I/O异步工作时，程序必须有一些事情可做。
- 受益必须超过增加的开销和系统调用次数。也可能需要考虑增加的程序复杂性和并不是所有的系统上都支持AIO的事实。

3.9.9 lio_listio

控制块列表还有一些其他的用途：将I/O请求打包到一起，用单个调用对其进行初始化：

lio_listio——列出定向的I/O

```

#include <aio.h>

int lio_listio(
    int mode,                /* LIO_WAIT or LIO_NOWAIT */
    struct aiocb *const list[], /* array of control blocks */
    int cbcnt,              /* number of elements in array */
    struct sigevent *sig    /* NULL or signal to generate */
);
/* Returns 0 on success or -1 on error (sets errno) */

```

和aio_suspend一样，lio_listio也取得一个cbcnt控制块的链表，忽略了链表中的NULL元素，并且不按照特定顺序初始化请求。由控制块的aio_lio_opcode成员规定每个操作：

LIO_READ 类似于调用了aio_read或pread。

LIO_WRITE 类似于调用了aio_write或pwrite。

LIO_NOP 空操作；忽略控制块。

mode参数决定了lio_listio初始化的I/O是同步的还是异步的，如表3-6所示，该表与表3-5相似（见3.9.1节）。

表3-6 已同步的与同步的lio_listio比较

	同 步 的	异 步 的
非已同步的	LIO_WAIT; 清除O_SYNC和O_DSYNC	LIO_NOWAIT; 清除O_SYNC和O_DSYNC
已同步的	LIO_WAIT; 设置O_SYNC或O_DSYNC	LIO_NOWAIT; 设置O_SYNC或O_DSYNC

因此，与“aio”系统调用不同的是，`lio_listio`不仅只用于异步I/O，也可以用于需要打包I/O调用的任何时间。

对异步`lio_listio`来说，当模式是`LIO_NOWAIT`时，可以通过信号或通过带有`sig`参数启动的线程来请求通知，这恰好与控制块中的`aio_sigevent`成员的作用一样。但是这里不同的是，通知是指链表中的所有已经完成的请求，仍可以通过个体控制块中的`aio_sigevent`成员获得要求完成的通知。

`LIO_WAIT`并不使用`sig`参数。

对于部分其他AIO调用来说，`lio_listio`的成功返回仅指该调用运行良好。该返回值对通过`aio_error`检测的I/O操作没有任何作用。

练习

- 3.1 依据3.2.2节中最后一段建议的那样修改该节中的程序。
- 3.2 编写标准`df`命令的实现代码。
- 3.3 依据3.6.2节中最后一段建议的那样修改该节中的`aupls`命令。
- 3.4 修改3.6.4节中的`getcwdx`函数，以便不更改当前目录。
- 3.5 修补3.6.5节开头描述的问题2。
- 3.6 修改3.6.5节中的`aupls`命令，以便可以通过名字排序列表。
- 3.7 修改3.6.5节中的`aupls`命令，以便可以通过`-t`选项修改时间和名字排序。
- 3.8 修改3.6.5节中的`aupls`命令，以便利用其他的标准选项（可选择的）。
- 3.9 写一段可以复制整个目录树和文件的程序。该程序应该带有两个参数：树的根节点（例如`/usr/marc/book`）和复制的根节点（例如`/usr/marc/backup/book`）。不必处理符号链接或硬链接（也就是说生成多个复制是可以的），也不必保持所有权、权限位或者时间。
- 3.10 与练习3.9类似，但为了增加可行性，要保持所有权、权限位和时间。使程序“可行性”依靠该命令是否作为超级用户运行。
- 3.11 与练习3.10类似，但是要保持符号链接结构和硬链接结构。
- 3.12 为什么没有`lchmod`系统调用？
- 3.13 把`access`写成函数。可以使用除了`access`之外的任何系统调用。
- 3.14 用`lio_listio`实现`readv`和`writew`（见2.15节）。以这种方式实现有什么优缺点？
- 3.15 可以如练习3.14那样实现`read`、`write`、`pread`、`pwrite`、`fsync`、`fdatasync`、`aio_read`、`aio_write`和`aio_fsync`吗？如果可以实现，那么请实现它们，并说出该方式有什么优点或缺点？
- 3.16 列出通过`aio_suspend`、信号、启动的线程或者用`aio_error`轮询来检测AIO完成的优缺点。（需要的相关信息在第5章和第9章中。）

第4章 终端I/O

4.1 概述

终端I/O很复杂，因此需要单独拿出一章来介绍。复杂之处并不在于一般的终端I/O，它甚至比文件I/O还要简单，而在于终端属性的多种可变性。本章将讨论终端如何与会话和进程组通信，并讨论如何创建伪终端，该伪终端能实现一个进程控制其他终端进程。

在UNIX系统中，终端I/O将终端看成是老式的硬拷贝电传打字机，该模型带有大量的能产生可拍击的打字盒，现在只能在博物馆或老式电影中看到了。内核对显示器（字符或图形）、功能性键盘、鼠标或其他点击设备并没有专门的支持，一些较新的现代设备是靠调用标准设备驱动程序的库函数控制的，最著名的支持字符显示的包是Curses，现在是[SUS2002]的一部分。所谓的UNIX图形用户接口（GUI）通常运行在X窗口系统上，也许会用到一个系统工具包，如Motif、Qt、KDE或者Gnome。

因为本章的大部分内容都与设备驱动程序有关，而与类似文件系统的内核部分无关，所以UNIX版本的不同会造成具体特性的很大差异。有时甚至个别的UNIX版本也需要根据终端设备驱动程序而进行修改。记住，并不是所有的终端I/O属性都是由UNIX系统本身产生的。随着智能终端、局域网、前端处理器的广泛应用，在UNIX内核查找字符流之前，以及按其要求方式发送它之后，这种字符流的处理已变得更容易了。这种前处理和后处理的详细情况变化太大，无法一一列举。本书一般只讨论终端I/O的标准特性，读者可以以此为基础，依据系统手册来构建自己的系统。

4.2 从终端读取数据

本节将解释如何从终端读取数据，包括在终端没有准备好读取数据时为何不阻塞的原因。

4.2.1 标准终端I/O

现在开始解释标准终端是如何进行输入和输出工作的；也就是说，当第一次登录时和使用stty命令定制自己的需求之前，终端是如何工作的。在接下来的几节中，将讨论如何使用fcntl和tcsetattr系统调用来改变标准终端属性。

对于输入或输出，一般有三种访问终端的方式：

- 1) 可以打开符号特殊文件/dev/tty进行读、写或者读写。这个特殊文件是进程控制终端的同义词（见4.3.1节）。
- 2) 如果知道特殊文件的实际名称（例如/dev/tty04），就可以用名字打开文件。但是如果仅想控制终端，那么使用普通的名字/dev/tty就不好了。在应用程序中，实际名称主要是用于访问终端，而不是控制终端。
- 3) 传统上，每个进程都会继承三个打开的文件描述符：标准输入、标准输出和标准错误输出。通常，不论这些文件描述符对控制终端开放与否，控制终端都可以随时使用它们，因为如

果这些文件对普通文件或管道是开放的，那么用户或父进程就可以决定重定向输入或输出。

适用于终端的基本I/O系统调用是open、read、write和close。调用creat没有太大意义，因为特殊文件必须是已经存在的。lseek对终端也没有作用，因为没有文件偏移量，这也意味着pread和pwrite也不适用。

除非规定了O_NONBLOCK标志，否则必须等待设备与终端建立连接之后才能打开终端设备，对于阻塞在等待用户连接的open进程来说，该特性是很重要的。然后从open得到返回值，接着调用登录进程以便实现用户登录。完成登录后，调用密码文件中规定的用户shell，用户开始工作。

默认情况下，read系统调用对终端的作用方式与文件也不一样：它的返回值不会超过一行输入，在整行输入准备好之前也没有字符返回，即使在read请求仅需要单个字符时也是如此。这是因为在用户完成一行输入之前，不能假定它是最终形式，即用户可能还会修改或完全删除字符以修改或完全取消输入。read返回的计数值可以用于确定实际已经读取的字符数。（以上描述的是规范的终端输入，但不能使其无效，见4.5.9节。）

可以用以下两种方式之一结束一行。最常用的方式是，换行符就是结束符。当按下回车键时，设备驱动程序会将回车符（八进制15）转换成一个换行（八进制12）。换句话说，通过按下Ctrl-d键，用户可以产生一个文件结束符（EOF）。在这种情况下，虽然没有换行结束符，但read仍可读取该行。一种重要的特殊情况是：如果用户在一行开始产生了一个EOF，那么read将返回0，因为EOF结束的行是空的。这看起来像是文件结尾，这也是为什么Ctrl-d可以作为文件结束“符”的原因。

下面的函数可以通过标准输入文件描述符STDIN_FILENO（定义为0）来读终端，如果出现终止换行字符，该函数会删除它，并添加NULL字符，将此行变成C字符串。

```
bool getln(char *s, ssize_t max, bool *iseof)
{
    ssize_t nread;

    switch (nread = read(STDIN_FILENO, s, max - 1)) {
    case -1:
        EC_FAIL
    case 0:
        *iseof = true;
        return true;
    default:
        if (s[nread - 1] == '\n')
            nread--;
        s[nread] = '\0';
        *iseof = false;
        return true;
    }
}

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

返回值用于指示错误，因此第三个参数指示的是EOF，如下面这个调用示例所示：

```
ec_false( getln(s, sizeof(s), &iseof) )
if (iseof)
    printf("EOF\n");
```

```
else
    printf("Read: %s\n", s);
```

对于终端设备来说，`getln`是有效的，因为该函数的一次系统调用就可以读取整行字符。而且该函数不必查询结束符，它仅读取`read`返回值。但该函数不适用于文件和管道，因为没有使用行限制符，所以`read`读取的数据太多。与`getln`读取一整行不同的是，`read`只读取紧接着的`max-1`个字符（假定出现很多字符）。

一个更普遍的`getln`版本可能会忽略终端设备的唯一特性——至多读取一行。它只简单地检测每个字符以寻找换行符：

```
bool getln2(char *s, ssize_t max, bool *iseof)
{
    ssize_t n;
    char c;

    n = 0;
    while (true)
        switch (read(STDIN_FILENO, &c, 1)) {
            case -1:
                EC_FAIL
            case 0:
                s[n] = '\0';
                *iseof = true;
                return true;
            default:
                if (c == '\n') {
                    s[n] = '\0';
                    *iseof = false;
                    return true;
                }
                if (n >= max - 1) {
                    errno = E2BIG;
                    EC_FAIL
                }
                s[n++] = c;
        }

    EC_CLEANUP_BGN
        return false;
    EC_CLEANUP_END
}
```

该版本将任何位置键入的`Ctrl-d`都当作是表示EOF，这与`getln`的功能是不同的（`getln`只将行开头的`Ctrl-d`当作EOF）。记住，除了终端，其他的都没有`Ctrl-d`，文件的结束只是简单的文件结尾或者是没有打开写文件描述符的管道。

尽管`getln2`能正确地读取终端、文件和管道，但它读原代码很慢，因为没有像2.12节所说的那样缓存输入。比较容易的补救方式是修改`getln2`，使其调用`Bgetc`——它是2.12节介绍的`BUFIO`包中的一部分。`Bopen`已经可以用于打开终端特殊文件（例如`/dev/tty`）了。然而，为了在标准输入中使用`Bgetc`，必须添加调用`Bfdopen`（练习4.2）的函数，它是从已经打开的文件描述符而不是从路径中初始化`BUFIO`指针，接着采用如下方式从标准输入中读取字符，无论是终端、管道还是文件：

```
ec_null( stin = Bfdopen(STDIN_FILENO, "r") )
while ((c = Bgetc(stin)) != -1)
    /* process character */
```

现在，对每种情况都采用了尽可能快的读取方式：对文件或管道一次读取一块，而对于终端，一次读取一行。

BUFIO包的实现不允许同一个BUFIO指针同时指向输入和输出。因此，如果向终端发送输出，则必须使用文件描述符STDOUT_FILENO（定义为1）打开第二个BUFIO。

UNIX系统标准I/O库提供了三个预定义的、已经打开的访问终端的FILE指针：stdin、stdout和stderr，因此函数fdopen就像Bfdopen一样，通常不必用于终端调用。

因为不做删除处理，所以向终端输出数据比输入更直接。与利用write输出许多字符一样，不管换行符是否出现，向终端输出，字符会立刻进行排队。

向终端打开的文件描述符close不比关闭文件做的事情多。它只是使文件描述符可再次重用；但由于文件描述符经常是0、1或2，所以不容易发现明显的重用，因此不必在程序结束时关闭这些文件描述符。[⊖]

4.2.2 非阻塞输入

如前所说，当read从终端读取数据时，如果无法获得数据行，则read在返回之前会等待数据。因为进程在此期间不做任何事情，所以被称为阻塞（blocking）。而对于文件来说就不会出现类似的情况：或者得到数据，或者到达文件结尾。也许以后会有另一个进程向该文件写入数据，但问题是当执行read时到底结尾在哪里。

open和fcntl设置的O_NONBLOCK标志可以使读操作非阻塞。如果得不到数据行，那么read会立刻返回值-1，并将error设置成EAGAIN。[⊖]

我们经常希望能随意地关闭和打开阻塞，因此下面将编写能适当调用fcntl的函数setblock。（采用的技术与3.8.3节中设置O_APPEND标志时所用的相同。）

```
bool setblock(int fd, bool block)
{
    int flags;

    ec_negl( flags = fcntl( fd, F_GETFL ) )
    if ( block )
        flags &= ~O_NONBLOCK;
    else
        flags |= O_NONBLOCK;
    ec_negl( fcntl( fd, F_SETFL, flags ) )
    return true;

    EC_CLEANUP_BGN
        return false;
    EC_CLEANUP_END
}
```

下面是setblock的检测程序。它关闭了阻塞，接着在循环中读取数据行。如果没有数据，则在继续运行之前休眠5秒钟。在提示信息中包含了循环开始以来的时间，使用的是1.7.1节介绍的time系统调用。

```
static void test_setblock(void)
```

⊖ 在第6章中，当两个进程与一个管道连接时，将关闭它们。

⊖ 在一些UNIX版本中，有类似的标志，称为O_NDELAY，如果设置了该标志，但没有数据，则read的返回值为0，此值与文件结束的返回值相同。因此最好使用O_NONBLOCK。

```

{
    char s[100];
    ssize_t n;
    time_t tstart, tnow;

    ec_negl( tstart = time(NULL) )
    ec_false( setblock(STDIN_FILENO, false) )
    while (true) {
        ec_negl( tnow = time(NULL) )
        printf("Waiting for input (%.0f sec.) ...\n",
            difftime(tnow, tstart));
        switch(n = read(STDIN_FILENO, s, sizeof(s) - 1)) {
            case 0:
                printf("EOF\n");
                break;
            case -1:
                if (errno == EAGAIN) {
                    sleep(5);
                    continue;
                }
                EC_FAIL
            default:
                if (s[n - 1] == '\n')
                    n--;
                s[n] = '\0';
                printf("Read \"%s\"\n", s);
                continue;
        }
        break;
    }
    return;
}

EC_CLEANUP_BGN
    EC_FLUSH("test_setblock")
EC_CLEANUP_END
}

```

下面是运行一次的输出结果。在键入“hello”之前等了一会，之后在键入Ctrl-d之前等待的时间更长：

```

Waiting for input (0 sec.) ...
Waiting for input (5 sec.) ...
Waiting for input (10 sec.) ...
hello
Waiting for input (15 sec.) ...
Read "hello"
Waiting for input (15 sec.) ...
Waiting for input (20 sec.) ...
Waiting for input (25 sec.) ...
^DEOF

```

休眠5秒的方法是在频繁地执行read（浪费CPU时间）和长时间等待（我们不能立刻处理用户的输入）之间的一个妥协。其实，可以看到在键入hello和程序最后读取输入并回送之间浪费了几秒钟的时间。因此一般来说，关闭阻塞和在read/sleep循环中得到输入是愚蠢的方法。下一节将介绍比这种方法更好的方法。

如上所述，当使用open打开终端特殊文件时也可以设置O_NONBLOCK标志。在这种情况下，O_NONBLOCK标志既影响open，也影响read：如果没有建立连接，open不等待就返回。

非阻塞输入的一个应用是监控多个终端。终端设备可能是通过终端端口与UNIX计算机相连的实验室仪器。字符被零散地发送，但需要将字符按出现的先后顺序累积。因为无法预测某个给定的终端何时会发送字符，所以不能使用阻塞I/O的方式，那样可能会导致这种结果：忽略了有数据传送的终端，而等待的却是一个没有任何数据传送的终端。然而使用非阻塞I/O，就可以按顺序轮询每个终端；如果终端没有准备好数据，read返回-1（errno设置成EAGAIN），接着继续询问下一个终端。如果完全循环一次后没有发现任何已准备好的数据，那么下次循环之前先休眠1秒，以便不独占CPU。

该算法可以通过readany进行示例。该函数的前两个参数分别是：文件描述符数组fds和数组中的计数值nfds。该函数直到这些文件描述符其中一个的read返回一个字符时才返回。接着通过第三个参数（whichp）返回读取字符的文件描述符fds的下标。字符本身是函数的值；0是指文件结尾；-1指错误。readany的调用程序假定以有用的方式累积得到的数据，以便数据日后的处理。^①

```
int readany(int fds[], int nfds, int *whichp)
{
    int i;
    unsigned char c;

    for (i = 0; i < nfds; i++)
        setblock(fds[i], false); /* inefficient to do this every time */
    i = 0;
    while (true) {
        if (i >= nfds) {
            sleep(1);
            i = 0;
        }
        c = 0; /* return value for EOF */
        if (read(fds[i], &c, 1) == -1) {
            if (errno == EAGAIN) {
                i++;
                continue;
            }
            EC_FAIL
        }
        *whichp = i;
        return c;
    }
}

EC_CLEANUP_BGN
return -1;
EC_CLEANUP_END
}
```

程序中关于调用setblock低效的注释是指，实际上不必在每次调用readany时都调用它。如果模块较小，那么让调用者对它负责效率会更高。

下面是readany的检测函数。它打开了两个终端：一个是控制终端（运行在网络其他地方的telnet应用程序）/dev/tty，如4.2.1节的解释那样，另一个是/dev/pts/3，它是显示器上与计算机直接相连的xterm窗口，该窗口运行的是SuSE Linux系统。（使用tty命令可以发现/dev/pts/3的名字。）

① 假定实验室仪器正输入带有终止符的新行。在4.5.9节，将介绍如何不等待行准备好即可读取数据。

```

static void readany_test(void)
{
    int fds[2] = {-1, -1}, which;
    int c;
    bool ok = false;

    ec_negl( fds[0] = open("/dev/tty", O_RDWR) )
    ec_negl( fds[1] = open("/dev/pts/3", O_RDWR) )
    while ((c = readany(fds, 2, &which)) > 0)
        printf("Got %c from terminal %d\n", isprint(c) ? c : '?', which);
    ec_negl( c )
    ok = true;
    EC_CLEANUP

    EC_CLEANUP_BGN
        if (fds[0] != -1)
            (void)close(fds[0]);
        if (fds[1] != -1)
            (void)close(fds[1]);
        if (!ok)
            EC_FLUSH("readany_test1")
    EC_CLEANUP_END
}

```

下面是在控制终端上得到的输出，是由printf函数语句输出的：

```

$ readany_test
dog
Got d from terminal 0
Got o from terminal 0
Got g from terminal 0
Got ? from terminal 0
Got c from terminal 1
Got o from terminal 1
Got w from terminal 1
Got ? from terminal 1

```

下面是执行过程：首先，紧跟控制终端的回车输入“dog”，响应“dog”结果后接着显示了4行。（问号表示新行。）然后我又在Linux计算机上，输入了“cow”和回车，得到的输出如下：

```
cow: command not found
```

在控制终端没有显示任何内容。问题是在xterm窗口正在运行一个shell时，由于等待输入而阻塞于read状态中。对被废弃的终端来说，这样处理是正常的。两个进程同时读同一个终端，shell首先得到字母，因此当然当作命令解释。接着又采用如下方式重新实验：

```
$ sleep 10000
cow
```

sleep命令运行在前台，它使shell等待了很长时间，这次输入的数据传给了readany_test命令，它输出了输入的最后4行。这说明终端驱动程序不属于任何一个特殊进程；仅仅因为打开它的进程没有读数据，并不能说明另一进程不可以读数据。

该检测程序的另一个解释：来自任何输入终端的EOF都可以终止程序的执行，这也是我决定编写它的原因。在实际的应用中可能需要做不同的工作。

4.2.3 select系统调用

调用sleep在花费时间方面有两个缺点：首先，如以前提起的，在处理输入的字符之前

大约要延迟1秒的时间，这对于时间性要求很强的应用来说是个问题。其次，在准备好字符之前，我们可能会多次唤醒并执行其他无用的循环。函数希望的最好的方式也许是：“在任意一个文件描述符准备好字符前，让我休眠吧。”如果这种情况真能实现，那么我们甚至不必使read非阻塞，因为若数据能准备好的话，read就不会阻塞了。

我们要找的这个系统调用叫作select（稍后我们就能接触到相似的pselect）：

select —— 等待I/O准备好

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

int select(
    int nfds,                /* highest fd + 1 */
    fd_set *readset,        /* read set or NULL */
    fd_set *writese,        /* write set or NULL */
    fd_set *errorset,       /* error set or NULL */
    struct timeval *timeout /* time-out (microseconds) or NULL */
);
/* Returns number of bits set or -1 on error (sets errno) */
```

pselect —— 等待I/O准备好

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

int pselect(
    int nfds,                /* highest fd + 1 */
    fd_set *readset,        /* read set or NULL */
    fd_set *writese,        /* write set or NULL */
    fd_set *errorset,       /* error set or NULL */
    const struct timespec *timeout, /* time-out (nanoseconds) or NULL */
    const sigset_t *sigmask /* signal mask */
);
/* Returns number of bits set or -1 on error (sets errno) */
```

与readany不同，它传递的是文件描述符的数组，用select可以设置想要检测的每个文件描述符的fd_set参数中某个的位[⊖]。处理集合的宏有4个：

FD_ZERO —— 清除整个fd_set

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

void FD_ZERO(
    fd_set *fdset /* fd_set to clear */
);
```

FD_SET —— 设置fd_set文件描述符

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

void FD_SET(
    int fd,                /* file descriptor to set */
    fd_set *fdset /* fd_set */
);
```

FD_CLR —— 清除fd_set文件描述符

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

void FD_CLR(
    int fd,                /* file descriptor to clear */
    fd_set *fdset /* fd_set */
);
```

[⊖] 虽然通常是位，但不是必须是位，这里使用这个术语想表达的只是模型，不是实现。

FD_ISSET——测试fd_set文件描述符

```
#include <sys/select.h>
int FD_ISSET(
    int fd,          /* file descriptor to test */
    fd_set *fdset   /* fd_set */
);
/* Returns 1 if set or 0 if clear (no error return) */
```

根据所等待的内容（读、写或者错误）的不同，可以用0、1、2或者3设置select。当所有fd_set参数都为NULL时，select就会进入阻塞，直到超过设定时间或者被信号中断；这不是特别有用的。

可以使用FD_ZERO初始化集合，接着按照如下方式使用FD_SET为每个文件描述符设置感兴趣的位：

```
fd_set set;

FD_ZERO(&set);
FD_SET(fd1, &set);
FD_SET(fd2, &set);
```

接着调用select，此时它会阻塞（即使设置了O_NONBLOCK），直到一个或多个文件描述符准备好读或写，或者出现错误。它会修改所传递给它的集合，这次为每个准备好的文件描述符设置一个位，必须使用如下代码检测每一个文件描述符，看是否设置了位：

```
if (FD_ISSET(fd1, &set)) {
    /* do something with fd1, depending on which fd_set */
}
```

无法得到文件描述符的列表；必须分别询问每个文件描述符。通过检测位的值可以知道文件描述符准备做什么。因此，即使输入集合是相同的，也不能让多个参数使用相同的集合，为了知道文件描述符想做什么，需要不同的输出集合。

即使对应文件描述符已经准备好了，只有在输入设置了某位时，输出时才必须设置该位。也就是说，只能得到所问问题的答案。

select运行成功，则返回所有集合中的位集合的总数，除了用于调用FD_ISSET（可能在循环中）时对遇到的文件描述符数进行双检查以外，该数几乎没什么用。

第一个参数nfds不是输入位集合的个数。它是select应该考虑的集合的长度。也就是说，对每个输入集合，设置的所有位都必须在编号为0到nfds-1的文件描述符范围内，也包含0和nfds-1。通常，可以计算出最大文件描述符个数，然后加1。如果不想那么做，那么可以使用常量FD_SETSIZE，它是一个集合所能拥有的最大文件描述符个数。但是因为该数一般很大（比如为1024），所以对于select来讲足够了。

select的最后一个参数是超时间隔，使用了1.7.1节解释的timeval结构。在这段时间间隔中，如果没有任何事情发生，那么select会返回值0——没有错误，也没有位集合。注意，select可能会修改该结构，因此再次调用select之前应重新设置timeval。

如果timeout为NULL，那么就没有超时；它等同于无限时间间隔。如果不是NULL，而是时间为0，那么select执行检测后会立即返回；可以通过这种方式实现轮询，代替等待。当程序有其他工作要做时，这种方式是有意义的：因为程序正在高效使用CPU，所以在进行其他工作时，可以不断地进行轮询；之后，当工作完成时，进行阻塞（时间不为0，或者

timeout为NULL),除了等待不做其他任何工作。

下面是3种没有正确使用select的方式:

- 将第一个参数nfd设置成编号最大的文件描述符,而不是该数加1。
- 再次调用select时,使用了与更改过的集合相同的集合,而没有意识到返回值中仅对准备好的文件描述符进行了设置。每次调用前,都必须重新初始化输入集合。
- 没有理解“准备好”的含义。它不是指数据准备好,而仅是指清除O_NONBLOCK设置时,read或write不阻塞。返回计数为0(EOF)、错误或者数据都是可能的。文件描述符是否设置了O_NONBLOCK无关紧要,该设置仅意味着文件描述符决不会阻塞;仅在清除O_NONBLOCK设置后仍不阻塞的假设情况下,select才认为是已经准备好了。

下面是更好的readany实现:

```
int readany2(int fds[], int nfd, int *whichp)
{
    fd_set set_read;
    int i, maxfd = 0;
    unsigned char c;

    FD_ZERO(&set_read);
    for (i = 0; i < nfd; i++) {
        FD_SET(fds[i], &set_read);
        if (fds[i] > maxfd)
            maxfd = fds[i];
    }
    ec_negl( select(maxfd + 1, &set_read, NULL, NULL, NULL) )
    for (i = 0; i < nfd; i++) {
        if (FD_ISSET(fds[i], &set_read)) {
            c = 0; /* return value for EOF */
            ec_negl( read(fds[i], &c, 1) )
            *whichp = i;
            return c;
        }
    }

    /* "impossible" to get here */
    errno = 0;
    EC_FAIL

    EC_CLEANUP_BGN
        return -1;
    EC_CLEANUP_END
}
```

该版本的效率是很高的。read保持阻塞,没有轮询循环,也没有休眠。所有的等待都在select中,当它返回时,我们可以读字符,并立即带有字符返回。

如果碰巧另一个线程和阻塞read读取的是同一个文件描述符,那么在代码执行到read之前,select原来说的已经准备好的数据可能已经不见了,这意味着将永远阻塞。修补这个漏洞的最容易的方法就是使read非阻塞,当返回值为-1并设置errno为EAGAIN时,循环返回到select。

担心终端的write阻塞是不正常的。即使由于驱动程序的缓冲区已满而出现这种情况,也可以很快地清空缓冲区。通常对其他输出(如管道或套接字)而言,出现写阻塞是严重的,在以后的章节中会介绍这些内容。特别是,在8.1.3节给出了以典型方式使用select的例子,

该方式使用了接受多个客户连接的套接字。

如果设置了读集合位，则可以读；如果设置了写集合位，则可以写。如果设置了错误集合位，那么可以做什么呢？答案依赖于文件描述符为何打开。对套接字，错误比“异常情况”要好，异常将转到已经准备好的带外数据（见8.7节）。对本节讲到的终端，没有标准的异常情况或错误情况，尽管对执行程序来说出现非标准是可能的，但必须查看对应系统的文档。

`pselect`是`select`的一个很有魔力的变体；其与`select`之间的区别是：

- 对`pselect`来说，超时是`timespec`结构的，以纳秒为单位，不能修改的。
- 在调用`pselect`时需设置第6个参数——信号掩码（见9.1.5节），并在返回时恢复旧的掩码值。鉴于第9章说明的原因，如果期望由信号来中断程序运行，那么最好使用`pselect`。而不是`select`。（如果`sigmask`参数为`NULL`，那么就信号而言，`pselect`和`select`是相同的。）

`pselect`是SUS版本3的新功能，因为刚刚出现，可能有的系统还没有它。

4.2.4 poll系统调用

poll——等I/O准备好

```
#include <poll.h>

int poll(
    struct pollfd fdinfo[], /* info on file descriptors to be tested */
    nfds_t nfds,           /* number of elements in fdinfo array */
    int timeout            /* time-out (milliseconds) */
);
/* Returns number of ready file descriptors or -1 on error (sets errno) */
```

struct pollfd——poll的结构

```
struct pollfd {
    int fd;                /* file descriptor */
    short events;          /* event flags (see table, below) */
    short revents;         /* returned event flags (see table, below) */
};
```

`poll`最初是为在AT&T系统V（见4.9节）上使用流I/O设备而设计的，但是，与`select`一样，它也可以用于打开任何文件类型的文件描述符。尽管它的名字`poll`像`select`，但它既可以用于等待，也可以用于轮询。除了Darwin 6.6之外，大部分系统有`poll`。

使用`select`可以对文件描述符设置位掩码，而使用`poll`可以设置`pollfd`结构，该结构包括了你想了解的每个文件描述符内容。对每一个文件描述符，在`events`字段中为每一个需被监控的事件（例如，读、写）设置标志。当`poll`返回时，每个结构的`revents`字段中都包含指示哪些事件发生了的位集合。因此，与`select`不同，该调用不干扰输入，所以你可以重用输入。

如果试图用很大的值检测文件描述符，那么`poll`比`select`的效率高。为了说明原因，假定你想要检测的文件描述符是1000。`select`需要检测所有从0到1000的位，但你可以建立只有一个元素的`poll`数组。`select`的另一个问题是，它的集合大小是根据潜在的文件描述符的数量设定的，一些内核被配置以允许很大的数。具有10 000个位的位掩码的效率是相当低的。

与select和pselect不同的是，timeout参数以毫秒为单位，而不是timeval或timespec结构。如果timeout是-1，poll会阻塞，直到至少有一个要求检测的事件发生在文件描述符上。如果timeout是0，则与select和pselect相同，poll仅检测文件描述符，并返回，返回值可能为0，那表示没有事件发生。如果timeout是正数，poll将顶多阻塞这么长时间。（因此，与select和pselect相同，0和正数这两种情形实际上是相同的。）

表4-1列出了poll的事件标志。为了弄清这些标志的意思，需要知道poll在“普通”数据、“优先级”数据和“高优先级”数据之间的区别。根据文件描述符打开的目的不同，这些术语的含义是不同的。

表4-1 poll系统调用的事件标志

标志	含义
POLLRDNORM	准备读普通数据
POLLRDBAND	准备读优先级数据
POLLIN*	与POLLRDNORM POLLRDBAND相同
POLLPRI*	准备读高优先级数据
POLLWRNORM	准备写普通数据
POLLOUT*	与POLLWRNORM相同
POLLWRBAND	可以写优先级数据
POLLERR*	出现I/O错误；仅在revents中设置（即不在输入中设置）
POLLHUP*	断开设备（不再可写，但可读）；仅在revents中设置
POLLNVAL*	无效的文件描述符；仅在revents中设置

*大部分是用于非数据流；通常POLLPRI仅用于套接字。

除了非正常情况，可以认为POLLIN|POLLPRI与select的读是等价的，POLLOUT|POLLWRBAND与select的写操作是等价的。注意，使用poll时，不必特意询问异常情况——无论输入标志是什么，如果异常情况要求设置表中的最后3个标志，那么就可以设置它们。

如果已经创建了pollfd数组，但却想禁止它对文件描述符进行检测，那么可以通过将fd字段设置为-1来实现。

下面是使用poll实现的readany（来自4.2.2节）。（4.2.3节已经有了一个使用select实现的版本。）

```
#define MAXFDS 100

int readany3(int fds[], int nfd, int *whichp)
{
    struct pollfd fdinfo[MAXFDS] = { { 0 } };
    int i;
    unsigned char c;

    if (nfd > MAXFDS) {
        errno = E2BIG;
        EC_FAIL
    }
    for (i = 0; i < nfd; i++) {
        fdinfo[i].fd = fds[i];
        fdinfo[i].events = POLLIN | POLLPRI;
    }
}
```



```

    }
    ec_negl( poll(fdinfo, nfds, -1) )
    for (i = 0; i < nfd; i++) {
        if (fdinfo[i].revents & (POLLIN | POLLPRI)) {
            c = 0; /* return value for EOF */
            ec_negl( read(fdinfo[i].fd, &c, 1) )
            *whichp = i;
            return c;
        }
    }
    /* "impossible" to get here */
    errno = 0;
    EC_FAIL
}

EC_CLEANUP_BGN
return -1;
EC_CLEANUP_END
)

```

对readany3的解释如下:

- 将结构全都初始化为0比较好, 这是在fdinfo数组的定义中进行的工作, 以防实现定义了其他我们不了解的字段。同样, 我们以前在结构上必须使用调试程序, 对没有明确初始化的字段(例如revents), 将字段定义为0, 而不采用无用数据会使显示信息的可读性更好。
- 程序仅能处理100个文件描述符。通过改变此值来实现动态分配数组, 对其进行改进将会很容易。
- 严格地说, 对revents进行检测时假设标志使用的是不同的位, SUS中似乎并没有这么说, 但对poll来说这是个安全的假设。
- 我们一般不检测POLLERR、POLLHUP或者POLLNVAL标志, 但是有时应该检测。
- 在实际的应用中, 可能不需要对每个调用都设置pollfd结构的数组, 因为文件描述符并不经常改变该结构。
- 如果另一个线程正在读相同的文件描述符, 阻塞read将是个问题。select例子中的解决方式(见4.2.3节)也可以用在这里。

4.2.5 测试和读单个输入数据

在4.2.2节中, 给出了使用select和poll处理多个输入数据的例子。有时尽管只有一个数据, 但是在没有读取它的情况下也需要知道字符是否已经准备好了; 需要把检测数据操作与读取数据操作分开。通过把超时时间设置为0, 可以使用select或者poll将它们分开。但是, 如果只有一个输入, 那么在设置了O_NONBLOCK标志的情况下, 和单独使用read几乎一样简单。使用两个函数可以完成这个工作, 用cready检测字符是否已经准备好, 用cget读取数据。

使用cready的麻烦是, 当字符准备好时, 读取它的是read, 但我们想让cget读取它。简单的解决办法是把先前读入的数据放到缓冲区中, 接着cget可以先在缓冲区中查找。如果字符在缓冲区中, 就返回该字符; 否则, 打开阻塞, 并调用read。在下面的cready和cget的实现中使用了以上方案:

```
#define EMPTY '\0'
```

```
static unsigned char cbuf = EMPTY;
typedef enum {CR_READY, CR_NOTREADY, CR_EOF} CR_STATUS;

bool cready(CR_STATUS *statusp)
{
    if (cbuf != EMPTY) {
        *statusp = CR_READY;
        return true;
    }
    setblock(STDIN_FILENO, false);
    switch (read(STDIN_FILENO, &cbuf, 1)) {
    case -1:
        if (errno == EAGAIN) {
            *statusp = CR_NOTREADY;
            return true;
        }
        EC_FAIL
    case 0:
        *statusp = CR_EOF;
        return true;
    case 1:
        return true;
    default: /* "impossible" case */
        errno = 0;
        EC_FAIL
    }
}

EC_CLEANUP_BGN
return false;
EC_CLEANUP_END
}

bool cget(CR_STATUS *statusp, int *cp)
{
    if (cbuf != EMPTY) {
        *cp = cbuf;
        cbuf = EMPTY;
        *statusp = CR_READY;
        return true;
    }
    setblock(0, true);
    switch (read(STDIN_FILENO, cp, 1)) {
    case -1:
        EC_FAIL
    case 0:
        *cp = 0;
        *statusp = CR_EOF;
        return true;
    case 1:
        *statusp = CR_READY;
        return true;

    default: /* "impossible" case */
        errno = 0;
        EC_FAIL
    }
}

EC_CLEANUP_BGN
return false;
EC_CLEANUP_END
}
```



当操作成功时，两个函数都返回true，错误时返回false。参数statusp给出了cready的结果：CR_READY、CR_NOTREADY或者CR_EOF。对cget来说，有可能取的值是CR_READY和CR_EOF。

这里我们预先占有NUL字节作为空缓冲区指示符（EMPTY），它意味着将忽略cready读取的NUL字节。如果这是不允许的，那么单个bool变量也可以用于表示空缓冲区。

注意cready和cget在阻塞输入和非阻塞输入之间来回移动的方式。因为只读取了一个输入流，所以当检测字符是否可用时没有必要停留在非阻塞状态，和在4.2.2节中第一个低效的readany版本中必须做的一样。返回到阻塞状态，并调用read，毕竟等待字符是阻塞的意思。

当程序有一些可随意处理的工作时，cready和cget是最有用的，只要字符准备好了，随意的工作就可暂停。使用Curses的程序是较好的示例（4.8节）。Curses工作的方式是，所有的屏幕输出都保留在缓冲区，直到调用refresh将缓存的数据发送到显示器上。这个过程很费时，因为refresh需要比较屏幕当前的内容与新图像的内容，以便最小化传送的字符数。速度快的打字员容易超过更新的时间，尤其在屏幕需要对每一个键盘敲击都更新时，因为必须使用屏幕编辑程序。但是如果输入没在等待，那么简洁的解决办法是从输入例行程序中调用refresh。如果输入正在等待，那么用户显然在屏幕没有更新之前就已经打字了，因此忽略了屏幕更新。只有等下次执行输入例行程序时才能更新屏幕，除非有一个字符也在等待。在用户停止键入后，屏幕将成为当前状态，另一方面，如果程序处理字符的速度比打字员键入的速度快，那么屏幕会随着每次的击键而更新。

看起来很复杂，但是使用前面开发的函数，仅需几行代码就能实现：

```
ec_false( cready(&status) )
if (status == CR_NOTREADY)
    refresh();
ec_false( cget(&status, &c) )
```

4.3 会话和进程组（作业）

本节将讨论会话和进程组（也称为作业），这些主要对shell有用。

4.3.1 术语

当用户登录时，就会建立一个新的会话，它包含一个新进程组，并且该进程组包含一个运行登录shell的单个进程。该进程是进程组头（process-group leader），它的进程ID（例如73056）是进程组ID。该进程也是会话头（session leader）；进程ID也是会话ID。^①用户登录的终端会成为会话的控制终端。会话头也是控制进程。这一层关系显示在了图4-1的右半部分，进程组的标识是FG，即前台。

如果shell允许作业控制，那么命令或者命令管道可以在后台运行，像这样：

```
$ du -a | grep tmp >out.tmp&
```

这样形成了一个新进程组，它里面有两个进程，如图4-1左半部分所示。后台的进程组标识为

^① 许多标准使用短语“会话头的进程组ID”，但我更喜欢只把它称为会话ID。

BG。这两个新进程中的一个就是这个新进程组的头。

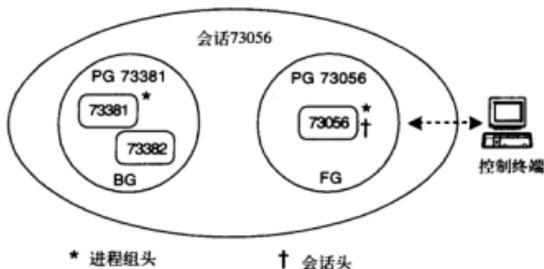


图4-1 会话、进程组、进程和控制终端

两个进程组（73056和73381）在同一个会话中，并且有同样的控制终端。用作业控制可以为每个命令行创建一个新进程组，每个这样的进程组也称为作业。

仅仅是因为进程组运行在后台，而不是说标准文件描述符不能从终端来回定向。在前面的管道例子中，`du`的标准输出是通过管道连接到`grep`的标准输入中的，`grep`的标准输出重定向到文件，但是那是用户那样特别设置的，而不是因为运行在后台上。

作业控制是指，如果某些信号从控制终端产生，例如中断（`SIGINT`）、终止（`SIGQUIT`）或者挂起（`SIGTSTP`），这些仅发送到前台进程组，而不理任何后台进程组。“控制”的含义是指，用户可以执行shell命令，以实现在前台和后台之间来回移动进程组（作业）。通常，如果进程组正在前台运行，那么可用`Ctrl-z`向其发送一个`SIGTSTP`信号，这样shell命令就可以将该进程组移动到后台，而将某个后台进程组移动到前台。或者使用shell命令`fg`将特定的后台进程组移动到前台，同时也将原来的前台进程组发送到后台。

如果不能激活作业控制，那么管道例子中的两个新进程（正在运行的`du`和`grep`）就不会运行在它们各自的进程组中，而是与第一个进程（shell）一起运行在73056进程组中，如图4-2所示。`du`和`grep`进程仍然运行在后台，这意味着该shell没有等待这两个进程完成。因为只有一个进程组，所以无法在前台和后台之间移动进程组，控制终端产生的任何信号都会进入会话中的唯一进程组的所有进程。在终端键入`Ctrl-c`将向所有三个进程发送`SIGINT`，包含两个后台进程，并且，除非这些进程已经设置了放弃或忽略该信号，否则该信号将结束这些进程，这是这个特殊信号的默认动作。（第9章对信号将有更多的介绍。）

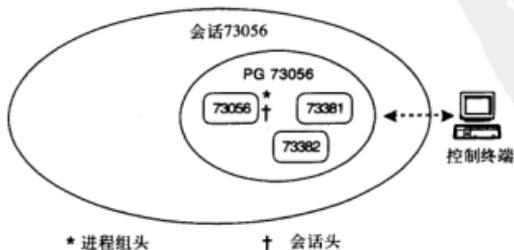


图4-2 没有作业控制；每个会话只有一个进程

无论有无作业控制，当控制终端挂起或者断开连接时，SIGHUP信号都会被发送给控制进程，默认情况下该信号终止控制进程。对于典型会话，该信号将终止登录shell，并注销该用户。控制终端将不再是会话的控制终端，因此会话中将没有控制终端。任何标准都没有规定，会话中的其他进程是否可以继续访问该终端，但大多数执行程序可能都会禁止这样的访问，例如通过从read或write返回错误。

然而，一旦由于某种原因控制进程终止（不是仅接收了SIGHUP），前台进程组中的每个进程（或者仅一个）都会得到SIGHUP，默认情况下该信号会终止这些进程。除非已经安排捕获或忽略该信号，否则这些进程将无法访问前面的控制终端。

当后台中的进程组企图访问控制终端时会发生什么问题呢？不管控制进程是否在运行，基本原则是后台中的进程企图从控制终端读数据时都将被发送给一个SIGTTIN信号，任何向控制终端写数据的企图都将被发送给一个SIGTTOU信号。这里的“写”也指使用终端控制系统调用——tcsetattr、tcdrain、tcflow、tcflush和tcsendbreak（见4.5节和4.6节）。

这些信号的默认行为是挂起进程，这个意义很大：需要访问控制终端的后台进程将停止运行，当你准备与这些进程交互时，你可以再将这些进程移动到前台。

除这个基本原则之外的例外情况：

- 后台进程企图用忽略的或者阻塞的SIGTTIN信号进行读操作，以替代从read得到EIO错误。
- 孤立的后台进程（进程组头已经终止的进程）也会从read得到EIO错误。
- 如果清除了TOSTOP终止属性（见4.5.6节），也允许进程进行写操作。
- 如果设置了TOSTOP，但忽略或阻塞了SIGTTOU信号，也允许进程进行写操作。
- 如果设置了TOSTOP，孤立的后台进程会从write得到EIO错误。

因此，这些规则概括如下：任何情况下，后台进程都不能读控制终端；后台进程将得到错误或者被停止。如果孤立的后台进程企图向终端写数据，将得到错误信息。如果清除了TOSTOP信号，或者忽略或阻塞了SIGTTOU，那么非孤立的后台进程可以进行写操作；否则停止进程。

读操作和写操作之间不对称的原因在于：两个进程同时进行读字符操作是没有意义的，因为那样很危险。（想象一下，在键入rm*.o时，shell仅得到了rm*。）两个进程同时进行写操作只会出现混乱，但如果那是用户实际要做的，也可以那样做。

4.3.2 会话系统调用

setsid——建立会话和进程组

```
#include <unistd.h>

pid_t setsid(void);
/* Returns process-group ID or -1 on error (sets errno) */
```

getsid——得到会话ID

```
#include <unistd.h>

pid_t getsid(
    pid_t pid          /* process ID or 0 for calling process */
);
/* Returns session ID or -1 on error (sets errno) */
```

前面说过，每个登录都会建立一个新会话，但是会话从何而来？实际上，任何已经不是会话头的进程都可以调用setsid而成为新会话的头，而且是那个会话中唯一一个进程组的进程组头。重要的是，该新会话没有控制终端。这点对后台进程来说是很重要的，因为后台进程不需要控制终端，而且在会话需要创建与登录时创建的控制终端不同的控制终端时，这一点也是很重要的。新会话打开的第一个终端设备将成为该会话的控制终端。

4.3.3 进程组系统调用

setpgid —— 设置或建立进程组

```
#include <unistd.h>

int setpgid(
    pid_t pid,           /* process ID or 0 for calling process */
    pid_t ppid          /* process-group ID */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

getpgid —— 得到进程组ID

```
#include <unistd.h>

pid_t getpgid(
    pid_t pid           /* process ID or 0 for calling process */
);
/* Returns process-group ID or -1 on error (sets errno) */
```

通过调用setpgid，可以使已经不是进程组头的进程成为新进程组的头。如果两个参数是相同的，而且具有那个ID的进程组不存在，那么这种情况就会发生。或者，如果第二个参数指定了一个已经存在的进程组ID，那么pid参数指示的进程组将会改变。然而有一些限制：

- pid必须是正在调用的进程，或者是正在调用进程的子进程——仍然没有完成“exec”系统调用（见5.3节的解释）。
- pid必须与正在调用的进程处在同一个会话中。
- 如果ppid存在，那么它必须和正在调用的进程处于同一个会话中。

实际上，限制没有那么多。一般情况是，shell在管道中为每一个命令创建子进程，选择一个作进程组头，使用setpgid创建新进程组，接着通过其他的setpgid系统调用将管道中的其他命令放到那个组中。一旦所有这些都创建完成，进程组的分配方案就不会变了，尽管理论上可以改变。

虽然两个较旧的调用setpgrp和getpgrp也可以用于设置和得到进程组ID，但它们的功能比setpgid和getpgid的功能弱，并且已经过时。

4.3.4 控制终端系统调用

tcsetpgrp —— 设置前台进程组ID

```
#include <unistd.h>

int tcsetpgrp(
    int fd,             /* file descriptor */
    pid_t ppid         /* process-group ID */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

tcgetpgrp——得到前台进程组ID

```
#include <unistd.h>

pid_t tcgetpgrp(
    int fd          /* file descriptor */
);
/* Returns process-group ID or -1 on error (sets errno) */
```

tcgetsid——得到会话ID

```
#include <termios.h>

pid_t tcgetsid(
    int fd          /* file descriptor */
);
/* Returns session ID or -1 on error (sets errno) */
```

tcsetpgrp系统调用可以将进程组放到前台，这意味着它接收了控制终端产生的信号。前台上使用的所有内容移到后台。进程组必须和正在调用的进程在同一个会话中。

shell命令fg使用**tcsetpgrp**将所被请求的进程放到前台。键入Ctrl-z不会直接把前台进程移到后台；实际发生的是Ctrl-z向进程发送了SIGTSTP信号，SIGTSTP信号停止了该进程，接着父进程（shell）从waitpid系统调用得到返回值，此waitpid系统调用会告诉父进程发生了什么。shell接着执行**tcsetpgrp**并将其本身移动到前台。

tcgetsid仅在SUS系统中存在，在FreeBSD系统中得不到它。然而，可以从向控制终端打开的文件描述符中得到会话ID。首先要通过调用**tcgetpgrp**得到前台进程组ID（因为该进程组ID必须是会话进程中某个进程的进程ID），接着通过调用**getsid**得到会话ID。

4.3.5 使用与会话相关的系统调用

下面的函数用前面的系统调用输出了许多与会话和进程组相关的信息：

```
#include <termios.h>

static void showpgrpinfo(const char *msg)
{
    int fd;

    printf("%s\n", msg);
    printf("\tprocess ID = %ld; parent = %ld\n",
        (long)getpid(), (long)getppid());
    printf("\tsession ID = %ld; process-group ID = %ld\n",
        (long)getsid(0), (long)getpgid(0));
    ec_negl( fd = open("/dev/tty", O_RDWR) )
    printf("\tcontrolling terminal's foreground process-group ID = %ld\n",
        (long)tcgetpgrp(fd));
    #if _XOPEN_VERSION >= 4
    printf("\tcontrolling-terminal's session ID = %ld\n",
        (long)tcgetsid(fd));
    #else
    printf("\tcontrolling-terminal's session ID = %ld\n",
        (long)getsid(tcgetpgrp(fd)));
    #endif
    ec_negl( close(fd) )
    return;
}
```

```

EC_CLEANUP_BGN
    EC_FLUSH("showpginfo")
EC_CLEANUP_END
}

```

当进程启动和进程得到SIGCONT信号时，我们希望调用showpginfo，以便当进程在后台运行时我们能够了解进展情况。在第9章之前，本书没有过多地讨论信号捕获，目前你需要了解的是该结构的初始化，以及当SIGCONT信号到达时，调用sigaction为catchsig的执行做准备。^①之后，主程序进入休眠状态。如果因为信号到达，主程序从sleep返回，那么会再次进入休眠状态。

```

int main(void)
{
    struct sigaction act;

    memset(&act, 0, sizeof(act));
    act.sa_handler = catchsig;
    ec_negl( sigaction(SIGCONT, &act, NULL) )
    showpginfo("initial call");
    while (true)
        sleep(10000);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}

static void catchsig(int signo)
{
    if (signo == SIGCONT)
        showpginfo("got SIGCONT");
}

```

与pginfo交互的示例如下：

```

$ echo $$
5140
$ pginfo
initial call
    process ID = 6262; parent = 5140
    session ID = 5140; process-group ID = 6262
    controlling terminal's foreground process-group ID = 6262
    controlling-terminal's session ID = 5140

^Z[1]+ Stopped                  pginfo
$ bg %1
[1]+ pginfo &
got SIGCONT
    process ID = 6262; parent = 5140
    session ID = 5140; process-group ID = 6262
    controlling terminal's foreground process-group ID = 5140
    controlling-terminal's session ID = 5140

```

首先可以看到shell的进程ID是5140。开始时pginfo在前台运行，并位于自己的进程组中，

^① 在9.1.7节将讨论对系统调用在信号句柄中所做事情的限制，以及showpginfo所做的一些技术上不合法的事情。虽然在第4章对这些内容不了解，但不会对理解这个例子有多大妨碍。

该进程组的ID是6262，这与pginfo的进程ID相同。这意味着pginfo是进程组头，会话ID和shell进程ID相同，这意味着pginfo和shell位于同一个会话中。然后当用户按下Ctrl-z时，shell得到了其子进程pginfo的状态，并在停止运行后进行汇报，接着将它恢复到前台，并允许用户向shell——bg键入其他命令，该shell在后台重启了pginfo。后台向pginfo发送SIGCONT信号，这使得pginfo再次输出信息。除了这次前台进程组是5140之外，其他与上述的shell相同。

大多数与会话和进程组相关的系统调用由shell而不是其他应用程序所使用。然而，当要把进程的控制终端切换到伪终端时，setsid系统调用对应用程序来说十分重要，见4.10.1节所述。

4.4 ioctl系统调用

回顾以前讨论的内容，可以知道UNIX有两种设备：块设备和字符设备。在第3章已经讨论了块设备，本章将讨论一种用于终端的重要的字符设备。有一个控制所有类型字符设备的通用系统调用ioctl：

ioctl ——控制字符设备

```
#include <...>

int ioctl(
    int fd,           /* file descriptor */
    int req,         /* request */
    ...              /* arguments that depend on request */
);
/* Returns -1 on error (sets errno); some other value on success */
```

由于POSIX和SUS标准没有详细说明这些设备，所以除了下面将要列出的那两种例外情况之外，包含文件、各种请求和与ioctl使用相关的第三个参数都是依赖于实现的，一般在你试图控制的设备驱动程序文档中可以找到相关的详细内容。

两种例外情况是：

- 通过ioctl对终端所做的每件事情都有自己的标准函数，主要是因为这样的话，可以在编译时检测参数类型。这些函数是tcgetattr、tcsetattr、tcdrain、tcflow、tcflush和tcsendbreak，下一节将讨论这些函数。
- SUS确实详细说明了ioctl是如何用于流的，我会在4.9节中简短地对其进行阐述。除了4.9节之外，本书将不再进一步讨论ioctl。

4.5 设置终端属性

用于控制终端的两个重要系统调用是tcgetattr和tcsetattr，前者用以得到控制终端的当前属性，后者用于设置新属性。其他四个函数tcdrain、tcflow、tcflush和tcsendbreak将在4.6节讨论。

4.5.1 tcgetattr和tcsetattr的基本用法

tcgetattr —— 得到终端属性

```
#include <termios.h>

int tcgetattr(
    int fd,                /* file descriptor */
    struct termios *tp     /* attributes */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

tcsetattr —— 设置终端属性

```
#include <termios.h>

int tcsetattr(
    int fd,                /* file descriptor */
    int actions,          /* actions on setting */
    const struct termios *tp /* attributes */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

struct termios —— 终端控制函数的结构

```
struct termios {
    tcflag_t c_iflag;      /* input flags */
    tcflag_t c_oflag;      /* output flags */
    tcflag_t c_cflag;      /* control flags */
    tcflag_t c_lflag;      /* local flags */
    cc_t c_cc[NCCS];      /* control characters */
};
```

终端信息结构 (termios) 包含50个标志位, 这些标志位将告诉驱动程序如何处理字符的出入, 如何设置通信线路参数, 如波特率等。该结构也定义了几个控制字符, 如删除字符 (通常是Del和Backspace), 删除行 (通常是Ctrl-u) 和EOF (通常是Ctrl-d)。

在改变这些属性之前, 需要调用tcgetattr初始化该结构。因为该结构很复杂, 也可能加载了一些已实现的实现标志, 所以最好不要从零开始初始化该结构。得到该结构后, 可以按需调整标志和字段, 接着调用tcsetattr实现这个改变。第二个参数action控制如何和何时改变生效; 使用如下符号之一:

TCSANOW 根据该结构中的信息, 立刻设置终端。

TCSADRAIN 与TCSANOW相似, 但首先要等待所有待解决的输出发送完成。当实现影响输出的改变时才使用, 以确保在写入时进行有效地处理。

TCSAFLUSH 与TCSADRAIN相似, 但除了要等待所有待解决的输出排空外, 也要刷新输入队列。当开始新的交互模式 (如开始新的屏幕编辑) 时, 这种方式将是最安全的, 因为它能防止以前输入的字符造成不良的后果。

下面的小节描述了通常使用的大部分标志, 以及在一般应用中如何将这些标志混合 (例如“原操作”模式)。关于标准化标志的完整列表, 可以查阅[SUS2002]。或者, 也可以执行man termios或者man termio来查阅实现中所使用的标志。

4.5.2 字符大小和奇偶性

c_cflag中的标志代表字符大小 (CS7代表7位; CS8代表8位)、停止位的个数 (CSTOPB代表2个, 清除该标志代表1个)、是否应该检测奇偶性 (设置PARENB) 以及奇偶位

(设置PARODD代表奇数, 清除PARODD代表偶数)。这里没有创新; 一般比较好的方式是寻找有效标志的组合。

当奇偶校验失败时, 也可以得到更多关于字符的信息。如果设置了c_iflag中的PARMRK标志, 那么错误字符前面就会有0377和0两个字符。否则, 奇偶性不正确的字符将以NUL字节输入。或者可以将c_iflag的标志设置成IGNPAR, 以完全忽略奇偶性不正确的字符。

4.5.3 速度

现在大部分终端的运行速度都很快, 但是仍然有一个定义各种不同标准速度的符号(不是整数)列表, 包括一些很低的速度。其名字具有Bn的形式, 其中n可以取50、75、110、134、150、200、300、600、1200、1800、2400、4800、9600、19200或者38400。单位是比特/秒, 在UNIX文档和标准中称为“波特”。

一些实现在c_cflag字段中设定速度, 但是为了移植, 经常使用4个函数得到或者设置输入和输出的速度, 而不是直接处理termios结构。这些函数仅处理该结构——首先必须调用tcgetattr配置该结构, 接着还必须调用tcsetattr使速度改变生效。同样, 记住speed_t唯一的标准值是Bn符号。尽管一些执行程序可能允许以简单整数形式传递, 但也不可能方便地实现。

cfgetispeed —— 从termios结构得到输入速率

```
#include <termios.h>

speed_t cfgetispeed(
    const struct termios *tp /* attributes */
);
/* Returns speed (no error return) */
```

cfgetospeed —— 从termios结构得到输出速率

```
#include <termios.h>

speed_t cfgetospeed(
    const struct termios *tp /* attributes */
);
/* Returns speed (no error return) */
```

cfsetispeed —— 在termios结构中设置输入速率

```
#include <termios.h>

int cfsetispeed(
    struct termios *tp, /* attributes */
    speed_t speed /* speed */
);
/* Returns 0 on success or -1 on error (may set errno) */
```

cfsetospeed —— 在termios结构中设置输出速率

```
#include <termios.h>

int cfsetospeed(
    struct termios *tp, /* attributes */
    speed_t speed /* speed */
);
/* Returns 0 on success or -1 on error (may set errno) */
```

速度B0是特殊的: 如果将其设置为输出速度, tcsetattr将断开与终端的连接; 如果将

其设置为输入速度，则意味着输出速度和输入速度是相等的。

4.5.4 字符映射

`c_oflag`中的标志`INLCR`和`ICRNL`可以控制输入时的回车符和换行符之间的映射；通常第一个是清除，第二个是设置。对于输入回车符和换行符的终端，当按下回车键时，将忽略回车（`IGNCR`）。

在输出时，通常需要将换行映射为一个回车符和一个换行符；`c_oflag`中的标志`ONLCR`可以完成这项工作。其他标志可以将回车符改成换行符（`OCRNL`），并且停留在0列（`ONOCR`）。如果换行也引起了回车，那么有一个标志（`ONLRET`）可以将这个情况告诉给终端驱动程序，以便终端能跟踪该列（如对于`Tab`键和`Backspace`键来说）。

驱动程序也能控制仅使用大写字母的终端，与以前相比这类终端更不常见。如果设置了`c_lflag`中的标志`XCASE`、`c_iflag`中的`IUCLC`以及`c_oflag`中的`OLCUC`，那么输入时的大写字母将映射为小写，输出时的小写字符将映射为大写字符。因为在UNIX系统中使用小写字母的时候多于大写字母，所以这是很有用的。为了输入或者输出大写字母，可以在字母前面加上“`\`”符号。

如果设置了`c_iflag`中的`ISTRIP`标志，那么输入字符将缩短成7位（在奇偶校验完成后）。否则，输入都是8位，而一些终端使用7位编码的ASCII，因此在这些终端设备上这种缩短通常是很有用的。然而现在的设备和程序，如`xterm`、`telnet`或者`ssh`，可能可以传输完整的8位。输出时发送进程中所写入的所有位。如果8个数据位要被发送给终端，那么必须关闭奇偶生成器，这可以通过清除`c_cflag`中的`PARENB`标志来实现。

4.5.5 延迟和制表符

过去常见的终端是机械的，而且缺乏大的缓冲区，因此在执行不同的动作时需要时间，如返回车架。可以通过设置`c_oflag`中的标志来调整换行、回车、退格、水平制表、垂直制表以及构架反馈的延迟，但现在可能不需要这些工作了。

另一个标志`TAB3`可以把输出制表符替换成适量的空格。这对于有些终端来说是有用的，如没有自己的制表符的终端，或者是太麻烦而不能设置制表符的终端，或者是终端实际上为另一个计算机，该计算机正在下载输出，并且仅需要一些空格。

4.5.6 流控制

当用户按下`Ctrl-s`或者通过进程调用`tcflow`（见4.6节）时，都可以立即停止向终端输出。按下`Ctrl-q`或者通过调用`flow`都可以重新启动流。

如果设置了`c_iflag`中的`IXANY`标志，那么用户键入任何一个字符都可以重新启动流，不仅仅是`Ctrl-q`。如果清除了`IXON`标志，那么用户就得不到任何输出流控制，`Ctrl-s`和`Ctrl-q`没有任何特殊含义。

终端驱动程序也支持输入流控制。如果设置了`IXOFF`标志，那么输入队列满时，驱动程序将向终端发送`Ctrl-s`以中止输入。当由于进程读取队列中的一些字符而使队列长度减少时，驱动程序将向终端发送`Ctrl-q`来恢复输入。当然，这个特性只能用在支持它的终端上。

如果设置了`c_lflag`中的`TOSTOP`标志，且后台进程组中的某个进程企图向控制终端写数据，那么驱动程序会发送`SIGTTOU`信号，原因和4.3.1节讲解的相同。

4.5.7 控制字符

通过设置termios结构中c_cc数组的一些元素，可以改变几个控制字符的默认值。对每个可设置的控制字符，表4-2列出了其在c_cc中的下标以及默认值。在数组中字符用它的内部值代替。

表4-2 c_cc下标

下 标	含 义	默 认 值
VEOF	文件结尾	Ctrl-d
VEOL	替换的行结尾（很少使用）	未定义
VERASE	删除字符	Ctrl-?*
VINTR	中断；产生SIGINT信号	Ctrl-c
VKILL	删除行	Ctrl-u
VQUIT	退出；产生SIGQUIT信号	Ctrl-\
VSUSP	中断进程；产生SIGTSTP*	Ctrl-z
VSTART	恢复输入或者输出	Ctrl-q
VSTOP	挂起输入或者输出	Ctrl-s

* 尽管调换名字VSUSP和VSTOP是有意义的，但该表还是正确的。

* 因为当按Backspace键时，有些终端产生的是Ctrl-? (ASCII DEL) 而不是Ctrl-h (ASCII BS)。

ASCII控制字符如下：Ctrl-a到Ctrl-z对应数1~26，Ctrl-[、Ctrl-\、Ctrl-^]、Ctrl-^和Ctrl-_对应数27~31；Ctrl-? (Del)对应127；Ctrl-@的值是0。数32~126对应95个可打印字符，这些字符不能用于控制字符。标准US英文键盘中无法产生大于127的数，但是在其他类型的键盘上可能可以。使用功能键产生多个字符序列还没有标准的UNIX方式。

大部分实现都为某些行为（如删除单词、重新打印和放弃输出）定义了额外的字符；然而表4-2中仅列出了一些标准的字符。数组中元素的总个数是NCCS。

由c_cc下标VINTR、VQUIT和VSUSP规定的字符产生的信号，如4.3节讨论的一样。默认情况下，SIGINT终止进程，SIGQUIT使用信息转储终止进程，SIGTSTP将在收到SIGCONT信号之后停止进程。第9章对信号还要进行更多的讨论。

为了禁用控制字符，可以将控制字符设置为_POSIX_VDISABLE。或者，也可以通过清除c_lflag中的ISIG标志来禁用中断、退出和停止（c_cc[VSUSP]），该标志能禁止信号产生，如果_POSIX_VDISABLE不在头文件unistd.h中，那么可以从pathconf或者fpathconf（见1.5.6节）中得到其值。通常该变量被定义为0。

4.5.8 应答

终端通常运行在全双工状态，即在通信线路中数据可以同时双向流动。结果是，计算机（不是终端）应答键入字符，并提供字符已经正确得到的验证。通常使用输出字符映射完成这个工作，例如，将回车符映射为换行符，当应答时，换行符映射为换行符和回车符。

为了关闭应答，可以清除c_lflag中的ECHO标志。一般用于以下情况：一种是保密，如当键入密码时；另一种是因为进程本身必须决定是否回送、回送什么以及在那里回送，如当运行屏幕编辑时。

当删除字符和删除行时，可以使用两种特殊的应答。可以设置ECHOE标志，以便删除字符

应答，如退格-空格-退格一样。从CRT屏幕上清除被删除字符可以得到令人满意的效果（对终端硬拷贝没有任何作用，除非使键入的元素不稳定）。删除行后，设置标志ECHO可以对换行符进行应答（可能映射为回车符和换行符），这样用户可以在新行进行工作。

4.5.9 准时输入与规范输入

通常情况下，输入的字符要进行排队，直到完成一行（如通过换行符或者EOF指示）。只有这时才能进行read，尽管可能仅请求得到一个字符。在许多应用程序中，如屏幕编辑和格式输入系统，当键入字符时，该进程就需要这些字符，并不等字符装配成行。实际上“行”的概念没有任何意义。

如果清除了c_lflag中的ICANON标志（“规范的”），那么在读取输入字符之前，输入字符并没有装配成行，因此不能进行删除字符和删除行操作。删除字符和删除行失去了各自特定的含义。参数MIN和TIME决定什么时间满足read。当队列中的字符长度达到MIN，或者接到一个字节后时间已经过去了十分之TIME秒时，可以得到队列中的字符。TIME当作字节间的定时器，只有在收到一个字节时才重设置字节间的定时器，在接收到第一个字节之前它是不启动的，因此在没有收到字节时不会超时。（除非MIN设置为0。）

c_cc数组中的下标VMIN和VTIME保存了MIN和TIME。因为下标VMIN和VTIME的位置与VEOF和VEOL的相同，所以必须确保明确地设置了MIN和TIME，否则得到的可能是当前EOF和EOL所有完成的工作，这将造成非常奇怪的结果。如果你曾经发现每键入4个字符进程就完成一次输入，那么也许就是上面的原因造成的。（通常EOF字符Ctrl-d是4。）

MIN和TIME隐含的意义是，在不失去单次read读取多个字符的好处下，允许进程在键入字符的同时或稍后得到字符。然而，除非编写了使用缓冲输入的输入程序，否则最好将MIN设置为1（那么TIME的数值就无关紧要了，只要大于0就行），因为read系统调用无论如何只能读取一个字符。

建议将MIN和TIME的值都设置成大于0，如果两者之一或都为0，会出现如下特殊情况之一：

- 如果TIME为0，则定时器关闭，此时使用MIN，这里假设MIN的值大于0。
- 如果MIN的值为0，TIME将不再充当字节间定时器，调用read后，定时器会立即启动。当读取一个字节或定时器超时，无论哪种情况先发生，read会返回。因此在这种情况下，read可能会返回值为0的计数值。
- 在两者都为0的情况下，如果规定了所请求字节的最小值（read的第三个参数规定的值），并且该数在输入队列中有效，那么read将返回。如果没有可用字节，那么read会返回计数值0，因此这与非阻塞read操作相似。

当设置了O_NONBLOCK且MIN和/或TIME不为0时，标准中没有准确地说明会发生什么情况。有可能O_NONBLOCK优先（如果没有可用字符，则立即返回），也有可能MIN/TIME优先。为了避免发生这种不确定性情况，当清除ICANON时，不应当设置O_NONBLOCK。

下面是称作tc_keystroke的输入程序，该程序是缓冲的，不是实时的。因为它会改变终端，所以我们也提供了配对程序tc_restore，用以将终端内容恢复到在第一次调用tc_keystroke之前，在终止之前该调用程序必须调用tc_restore。

```
static struct termios tbufsave;
```

```

static bool have_attr = false;

int tc_keystroke(void)
{
    static unsigned char buf[10];
    static ssize_t total = 0, next = 0;
    static bool first = true;
    struct termios tbuf;

    if (first) {
        first = false;
        ec_negl( tcgetattr(STDIN_FILENO, &tbuf) );
        have_attr = true;
        tbufsave = tbuf;
        tbuf.c_lflag &= ~ICANON;
        tbuf.c_cc[VMIN] = sizeof(buf);
        tbuf.c_cc[VTIME] = 2;
        ec_negl( tcsetattr(STDIN_FILENO, TCSAFLUSH, &tbuf) );
    }
    if (next >= total)
        switch (total = read(0, buf, sizeof(buf))) {
            case -1:
                syserr("read");
            case 0:
                fprintf(stderr, "Mysterious EOF\n");
                exit(EXIT_FAILURE);
            default:
                next = 0;
        }
    return buf[next++];

EC_CLEANUP_BGN
    return -1;
EC_CLEANUP_END
}

bool tc_restore(void)
{
    if (have_attr)
        ec_negl( tcsetattr(STDIN_FILENO, TCSAFLUSH, &tbufsave) );
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

```

注意我们已经清除了ICANON标志（当然也设置了MIN和TIME）。典型情况下还需要清除更多参数、关闭echo、输出符号映射等。下一节将讨论这些内容。在特殊的应用程序中，需要用实验决定MIN和TIME的值。我们曾使用过10个字符和0.2秒，使用0.2这个值的效果似乎还不错。

下面是一段在循环中调用tc-keystroke进行测试的代码：

```

setbuf(stdout, NULL);
while (true) {
    c = tc_keystroke();
    putchar(c);
    if (c == 5) { /* ^E */
        ec_false( tc_restore() );
        printf("\n%s\n", "Exiting...");
        exit(EXIT_SUCCESS);
    }
}

```

```

    }
}

```

该测试代码中的重点是：当键入字符时由终端驱动程序应答字符，接着tc_keystroke返回字符时再由putchar函数回送字符。这样我们可以看到MIN和TIME的效果，如在下面的实验中，当键入“slow”时字母的回送速度很慢，键入“fast”时字母的回送速度很快。下划线表示键入的字母。

```

slllqowwfaalfast^E
Exiting...

```

因此MIN和TIME对tc_keystroke如何操作没有影响，因为它仍是准时的（仍在0.2秒内，即设置的TIME时间内），然而进行read操作时，所用时间却减少了十分之一（或者其他我们所设置的MIN值）。

4.5.10 原始终端I/O

前一节描述的准时输入（或者称为非规范输入）是有用的，但有时既需要关闭回送，又需要做许多其他的事情。该模式通常称作原始（raw）模式，即没有特殊的输入处理或者输出处理，字符也立即可读，不必等待装配成行。没有单独的设置原始模式的标志；然而必须一个一个地设置许多属性。（stty命令具有raw选项。）

尽管没有必须遵守的规定，但我们需要原始终端具有如下属性，你可以根据自己的目的对如下属性稍作改动：

- 1) 准时输入。清除ICANON并设置MIN和TIME。
- 2) 不使用字符映射。清除OPOST以关闭输出处理。对输入来说是清除INLCR和ICRNL。将字符大小设置成CS8。清除ISTRIP以便得到所有的8个数据位，清除INPCK和PARENB以便关闭奇偶校验。清除IXTEN以便关闭扩展字符处理。
- 3) 不使用流控制。清除IXON。
- 4) 不使用控制字符。清除BRKINT和ISIG，并且设置禁用所有的控制字符，也包括具体实现定义的控制字符。
- 5) 不回送。清除ECHO。

这些操作都封装在了函数tc_setraw中。注意它保存了旧的termios结构以便tc_restore（见前一节）使用。

```

bool tc_setraw(void)
{
    struct termios tbuf;
    long disable;
    int i;

#ifdef _POSIX_VDISABLE
    disable = _POSIX_VDISABLE;
#else
    /* treat undefined as error with errno = 0 */
    ec_neg1( (errno = 0, disable = fpathconf(STDIN_FILENO, _PC_VDISABLE)) )
#endif
    ec_neg1( tcgetattr(STDIN_FILENO, &tbuf) )
    have_attr = true;
    tbufsave = tbuf;
    tbuf.c_cflag &= ~(CSIZE | PARENB);
    tbuf.c_cflag |= CS8;
}

```



```

tbuf.c_iflag &= ~(INLCR | ICRNL | ISTRIP | INPCK | IXON | BRKINT);
tbuf.c_oflag &= ~OPOST;

tbuf.c_lflag &= ~(ICANON | ISIG | IEXTEN | ECHO);
for (i = 0; i < NCCS; i++)
    tbuf.c_cc[i] = (cc_t)disable;
tbuf.c_cc[VMIN] = 5;
tbuf.c_cc[VTIME] = 2;
ec_negl( tcsetattr(STDIN_FILENO, TCSAFLUSH, &tbuf) )
return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

```

下面是tc_setraw的测试程序，该程序使用stty命令显示终端设置：

```

setbuf(stdout, NULL);
printf("Initial attributes:\n");
system("stty | fold -s -w 60");
printf("\r\nRaw attributes:\n");
tc_setraw();
system("stty | fold -s -w 60");
tc_restore();
printf("\r\nRestored attributes:\n");
system("stty | fold -s -w 60");

```

下面是在Solaris系统上得到的输出内容。注意tc_setraw关闭了所有的c_cc字符，不仅是已知的标准字符。但是它没有关闭所有可能会使终端成为非原始终端的所有标志，如imaxbel。[⊖]不过你可能会发现，必须针对代码所要植入的每个系统的具体情况来调整tc_setraw，要不然就必须使用Curses库（见4.8节），它可以把终端设置成原始模式。

```

Initial attributes:
speed 38400 baud; evenp
rows = 40; columns = 110; ypixels = 0; xpixels = 0;
swtch = <undef>;
brkint -inpck icrnl -ixany imaxbel onlcr
echo echoe echok echoctl echoke iexten

Raw attributes:
speed 38400 baud; -parity
rows = 40; columns = 110; ypixels = 0; xpixels = 0;
min = 5; time = 2;
intr = <undef>; quit = <undef>; erase = <undef>; kill =
<undef>; eof = ^e; eol = ^b; swtch = <undef>; start =
<undef>; stop = <undef>; susp = <undef>; dsusp = <undef>;
rprnt = <undef>; flush = <undef>; werase = <undef>; lnext =
<undef>;
-inpck -istrip -ixon imaxbel -opost
-isig -icanon -echo echoe echok echoctl echoke

Restored attributes:
speed 38400 baud; evenp
rows = 40; columns = 110; ypixels = 0; xpixels = 0;
swtch = <undef>;
brkint -inpck icrnl -ixany imaxbel onlcr

```

⊖ 当输入行太长时会发出响铃，该标志控制响铃长短的非标准标志。

```
echo echoe echok echoctl echoke iexten
```

有时在调试程序期间（或者之后！），将终端置于原始模式的程序可能在恢复原始设置之前就中止了。用户有时会认为是计算机崩溃，或者锁定了终端——他们甚至不使用EOF 终止联接主机的操作。但从原始模式恢复是可能的。首先，回顾可知，清除了ICRNL设置；这意味着必须使用Ctrl-j来结束输入，而不是使用回车。[⊖]其次，因为关闭了ECHO，所以你也看不到键入的内容。如果以键入几个换行符作为开始；那么应该能看到一系列shell提示符，但它们并没有位于左侧空白上，因为关闭了输出处理（OPOST）。接着，紧随一个换行符键入stty sane，所有内容就应该能正常显示了。

4.6 其他终端控制系统调用

如4.5.1节所看到的，当使用tcsetattr设置终端属性时，在利用TCSADRAIN动作完成设置之前，可以清除（等待）输出队列，另外，也可以使用TCSAFLUSH动作刷新（扔掉）输入队列中的任何字符。在不用设置属性的情况下，利用两个系统调用就可以分别控制清除和刷新操作。

tcdrain——耗尽（等待）终端输出

```
#include <termios.h>

int tcdrain(
    int fd          /* file descriptor */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

tcflush——刷新（丢弃）终端输入、输出或者两者

```
#include <termios.h>

int tcflush(
    int fd,         /* file descriptor */
    int queue      /* queue to be affected */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

这些函数只对队列起作用——即从终端接收到的字符还没有被任何进程读取，或者一个进程写入的字符还没有向终端传送。

tcflush函数的第二个参数（queue）可以取下列的一个值：

TCIFLUSH刷新输入队列。

TCOFLUSH刷新输出队列。

TCIOFLUSH同时刷新输入和输出队列。

因此在设置属性之前，tcsetattr的参数TCSADRAIN的作用与调用tcdrain相同，tcsetattr的参数TCSAFLUSH的作用与带有队列参数TCIFLUSH的情况下同时调用tcdrain和tcflush相同。

下一个系统调用是tcflow，它允许应用程序挂起或者重新启动终端的输入或者输出：

⊖ 对shell来说，一个不错的增强措施是将回车符当作行结束符和换行符。（脚注首次出现在1985年——到目前为止仍没有改变。）

tcflow ——挂起或重启终端输入流或输出流

```
#include <termios.h>

int tcflow(
    int fd,           /* file descriptor */
    int action        /* direction and suspend/restart */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

action参数可以取下列值之一:

TCOOFF挂起输出。

TCOON重新启动已挂起的输出。

TCIOFF发送STOP字符, 目的是为了使终端挂起输入。

TCION发送START字符, 以便使终端重新启动已经挂起的输入。

准备向终端写入多页的应用程序时, 在输入每页后可以使用**TCOOFF**挂起输出, 以使用户在从键盘输入START字符之前, 可以读取数据, 默认情况下用Ctrl-q来继续输入(见4.5.7节)。带有**TCOOFF**调用**tcflow**之后, 应用程序可以继续写输出页, 当终端队列满时, 下次write操作将阻塞, 直到队列释放一些数据。

然而这种方法并不是用户所期望的, 而且不是一次一页(page-at-a-time)的应用程序运行的典型方式。像more和man这样的命令输出一页后, 会提示用户键入一个普通字符(如空格或者回车符)来得到下一页。在内部, 这样的应用程序在输出一页后, 会输出提示符, 并且阻塞read(典型情况下会使用规范输入和关闭回来阻塞), 等待用户继续输入字符。

在输入方, 历史上有应用**TCIOFF**和**TCION**防止终端溢出输入队列的例子, 但是如今这完全由驱动程序和/或硬件来处理。今天, **tcflow**可能仅用于特殊的设备而不是实际的终端。

同类系统调用还有**tcsendbreak**, 它向终端发送中断信号:

tcsendbreak ——发送中断到终端

```
#include <termios.h>

int tcsendbreak(
    int fd,           /* file descriptor */
    int duration      /* duration of break */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

“中断”是向终端发送一段时间的0比特流。这种方法常用来将终端置为一种特殊的“注意”模式。

如果duration参数为0, 则时间段将在1/4秒到1/2秒之间。如果该参数不为0, 其含义是随实现而定。但是不必担心其可移植性——因为你可能永远都没有机会向终端发送中断信号。

4.7 终端识别系统调用

如4.2.1节所述, 通用路径名/dev/tty提供了访问进程控制终端的方式, 无需知道控制终端的名字。然而, 在SUS2系统之前, 没有那个要求, 但却有实现类似功能的系统调用。

ctermid——为控制终端得到路径名

```
#include <stdio.h>

char *ctermid(
    char *buf          /* buffer of size L_ctermid or NULL */
);
/* Returns pathname or empty string on error (errno not defined) */
```

可以使用NULL参数调用ctermid,这时它会使用一个具有合适大小的静态缓冲区。或者,你也可以提供缓冲区的大小l_ctermid(包含NUL字节的空间),以避免当多个线程调用该函数时发生冲突。注意,当ctermid出错时,返回的是空字符串,而不是NULL。对ctermid来说,仅需要返回字符串/dev/tty,这比返回其他内容更有用,在大多数(如果不是全部的话)系统上那是其全部的工作。

一定程度来说,更加有用的是能够返回实际终端名字的系统调用,ttyname和ttyname_r恰好能完成上述功能,只是必须给它们一个打开的文件描述符作为输入:

ttyname——查找终端的路径名

```
#include <unistd.h>

char *ttyname(
    int fd             /* file descriptor */
);
/* Returns string or NULL on error (sets errno) */
```

ttyname_r——查找终端的路径名

```
#include <unistd.h>

int ttyname_r(
    int fd,           /* file descriptor */
    char *buf,       /* buffer for pathname */
    size_t bufsize   /* size of buffer */
);
/* Returns 0 on success or error number on error (errno not set) */
```

同我们见过的其他函数一样,_r后缀是指ttyname_r是可重入函数——它使用的不是静态存储器而是你传递给它的缓冲区。同样,和我们见过的其他函数一样,设定缓冲区大小是很棘手的事情:如果(limits.h中)定义了宏TTY_NAME_MAX,那么就可以使用该宏,如果没有定义该宏,那么就必须调用带有_SC_TTY_NAME_MAX参数的sysconf系统调用(见1.5.5节)。否则,只能始终调用sysconf。如果你知道仅有一个线程在调用ttyname,那么到目前这两种使用方法是较容易的。

下面是本人设计的tty命令,它实现的功能是:输出与标准输入连接的终端名字,或者当不是终端时,输出“not a tty”:

```
int main(void)
{
    char *ctty;

    if ((ctty = ttyname(STDIN_FILENO)) == NULL) {
        printf("not a tty\n");
        exit(1);
    }
}
```

```

    printf("%s\n", ctty);
    exit(0);
}

```

这里使用1和0代替了符号EXIT_FAILURE和EXIT_SUCCESS，因为SUS清楚地说明了返回代码应该是1或者0。(POSIX仅规定了EXIT_FAILURE不为0，并没说它是1)。

如果确实需要控制终端的名字，即使某种程度上来说，标准输入是向不同的终端或非终端输入开放的，那么你可能也会认为可以打开/dev/tty，并向ttyname传递那个文件描述符。哎，但是在我实验的系统上ttyname并不是按照这种方式工作的，它仅仅返回了/dev/tty，并没有返回实际的名字。实际上，似乎并没有一个可以方便地得到控制终端名字的方法，尽管标准输入是终端，而且可能是控制终端。

可以使用isatty系统调用检测文件描述符是否是向终端打开的：

isatty——测试终端

```

#include <unistd.h>

int isatty(
    int fd          /* file descriptor */
);
/* Returns 1 if a terminal and 0 if not (may set errno on 0 return) */

```

如果isatty的返回结果为0，那么就不能依赖于所设置的errno来判断出错原因，但是在大多数情况下，如果返回结果不是1，可以不必关心其原因。

4.8 全屏应用程序

传统的UNIX命令有时是交互的（如dc、more），但是它们仍然会读写文本行。至少有3个其他有意义的应用程序分类，这些应用程序更充分地利用了显示屏幕和与它们相关的输入设备的能力（如mice）：

- 面向字符的、运行在廉价终端（目前几乎不制造了）上的全屏应用程序，以及终端仿真程序，如telnet和xtrem。最著名的这种应用程序可能是vi文本编辑器。
- 图形用户接口（GUI）应用是为X Window 系统写的应用程序，可能使用了较高级的工具包，如Motif、Gnome、KDE或者Tcl/Tk。（也有除X之外的其他GUI系统。）
- Web应用程序，在这类应用程序中，用户接口所采用的技术有HTML、JavaScript和Java等。

仅有第一组使用了本章讨论的终端的功能，这也是本章的主要内容。X应用程序通过网络与所谓的X服务器对话，GUI处理实际上是在服务器上运行的，该服务器可能运行在基于UNIX系统的计算机上，也可能没运行在这类计算机上。如果是运行在基于UNIX系统的计算机上，那么X应用程序将直接通过设备驱动程序与显示设备和输入设备通话，而不过于字符终端驱动程序。同样，Web应用程序通过网络与浏览器进行通信（见8.4.3节），浏览器将处理用户的交互过程。如果浏览器运行在UNIX系统上，那么它通常会是一个X应用程序。

下面给出一个非常简单的、面向字符的、全屏应用程序的例子，该示例完成了清除屏幕和创建一个如下菜单的功能：

```

What do you want to do?
1. Check out tape/DVD
2. Reserve tape/DVD
3. Register new member
4. Search for title/actor
5. Quit

(Type item number to continue)

```

图4-3 菜单

用户键入的数字不回送，并且也不必键入回车键。在此小示例中，如果键入了3，屏幕上就会显示图4-4所示的内容。

```

You typed 3

(Press any key to continue)

```

图4-4 响应

下面给出这个应用程序的两种实现。第一种仅针对ANSI终端并且能在可以仿真VT100终端的设备上运行良好。ANSI终端（和VT100）有许多控制序列，这里我们仅使用两种：

- 序列 ϵ r; cH将光标定位在了第r行、第c列。（ ϵ 是转义字符。）
- 序列 ϵ [2J清屏。

以下两个函数使用了这两个控制序列；注意clear也将光标移动到了屏幕的左上角：

```

#define ESC "\033"

bool mvaddstr(int y, int x, const char *str)
{
    return printf(ESC "[%d;%dH%s", y, x, str) >= 0;
}

bool clear(void)
{
    return printf(ESC "[2J") >= 0 &&
        mvaddstr(0, 0, "");
}

```

如果想在用户键入错误字符时让终端响铃，那么可以使用ASCII代码来实现：

```

#define BEL "\007"

int beep(void)
{
    return printf(BEL) >= 0;
}

```

假设通过调用tc_setraw而使终端处于原始模式，则可以用getch读字符：

```
int getch(void)
{
    char c;

    switch(read(STDIN_FILENO, &c, 1)) {
    default:
        errno = 0;
        /* fall through */
    case -1:
        return -1;
    case 1:
        break;
    }
    return c;
}
```

这些就是我们所需要的所有用户接口服务程序，完整的应用程序在main函数中。显然，如果需要做某些事情，其代码会更长。

```
int main(void)
{
    int c;
    char s[100];
    bool ok = false;

    ec_false( tc_setraw() )
    setbuf(stdout, NULL);
    while (true) {
        ec_false( clear() )
        ec_false( mvaddstr(2, 9, "What do you want to do?") )
        ec_false( mvaddstr(3, 9, "1. Check out tape/DVD") )
        ec_false( mvaddstr(4, 9, "2. Reserve tape/DVD") )
        ec_false( mvaddstr(5, 9, "3. Register new member") )
        ec_false( mvaddstr(6, 9, "4. Search for title/actor") )
        ec_false( mvaddstr(7, 9, "5. Quit") )
        ec_false( mvaddstr(9, 9, "(Type item number to continue)") )
        ec_negl( c = getch() )
        switch (c) {
        case '1':
        case '2':
        case '3':
        case '4':
            ec_false( clear() )
            snprintf(s, sizeof(s), "You typed %c", c);
            ec_false( mvaddstr(4, 9, s) )
            ec_false( mvaddstr(9, 9, "(Press any key to continue)") )
            ec_negl( getch() )
            break;
        case '5':
            ok = true;
            EC_CLEANUP
        default:
            ec_false( beep() )
        }
    }
}

EC_CLEANUP_BGN
(void)tc_restore();
(void)clear();
exit(ok ? EXIT_SUCCESS : EXIT_FAILURE);
```

```
EC_CLEANUP_END
}
```

无论某类终端如何通用，可能也不会仅为它编写代码。这就需要使用一个称作Curses的标准库，该标准库在4.1节中介绍过了。它包含了每种终端的控制序列的数据库，因此在运行期间可以切换到合适的控制序列。

因为我巧妙地以Curses库中的函数名命名了我的程序中出现的、具有相同功能的函数mvaddstr、clear、beep和getch。所以我们必须给出main函数的Curses版本：

```
#include <curses.h>

/* "ec" macro for ERR (used by Curses) */
#define ec_ERR(x) ec_cmp(x, ERR)

int main(void)
{
    int c;
    char s[100];
    bool ok = false;

    (void)initscr();
    ec_ERR( raw() )
    while (true) {
        ec_ERR( clear() )
        ec_ERR( mvaddstr( 2, 9, "What do you want to do?" ) )
        ec_ERR( mvaddstr( 3, 9, "1. Check out tape/DVD" ) )
        ec_ERR( mvaddstr( 4, 9, "2. Reserve tape/DVD" ) )
        ec_ERR( mvaddstr( 5, 9, "3. Register new member" ) )
        ec_ERR( mvaddstr( 6, 9, "4. Search for title/actor" ) )
        ec_ERR( mvaddstr( 7, 9, "5. Quit" ) )
        ec_ERR( mvaddstr( 9, 9, "(Type item number to continue)" ) )
        ec_ERR( c = getch() )
        switch (c) {
            case '1':
            case '2':
            case '3':
            case '4':
                ec_ERR( clear() )
                snprintf(s, sizeof(s), "You typed %c", c);
                ec_ERR( mvaddstr( 4, 9, s ) )
                ec_ERR( mvaddstr( 9, 9, "(Press any key to continue)" ) )
                ec_ERR( getch() )
                break;
            case '5':
                ok = true;
                EC_CLEANUP
            default:
                ec_ERR( beep() )
        }
    }

    EC_CLEANUP_BGN
    (void)clear();
    (void)refresh();
    (void)endwin();
    exit(ok ? EXIT_SUCCESS : EXIT_FAILURE);
    EC_CLEANUP_END
}
```

有关这个程序的一些注释如下:

- 当出错时,大多数Curses函数都会返回ERR,因此这里只为它们定义了ec_ERR宏。Curses文档没有规定使用error,但是不用管它,因为知道当确实得到一个错误消息时,该消息的值可能是无意义的。
- Curses要求以initscr开始,以endwin结束。
- tc_setraw的Curses版本是raw。

本书中没有过多地介绍Curses,因为Curses不是内核服务。[⊖]Curses是SUS2和SUS3中的一部分,因此可以在网站www.unix-systems.org/version2/online.html上在线阅读该规范。在大多数系统上,也可以键入man curses或man ncurses(Curses的免费版本)来查看该规范。

4.9 流I/O

流是在一些UNIX系统实现上的一种机制,该机制允许字符设备驱动程序以模块化方式执行。在一定程度上,应用程序也可以将不同的流模块链接起来,以使那类驱动程序具有所需的性能。这有点类似于在shell层上运行的UNIX过滤程序的工作方式;例如who命令可能没对其输出进行排序的选项,但是如果需要可以将其导入sort。

在下节我们将讨论一个基于流的伪终端的示例。那里需要完成的工作是打开设备文件得到伪终端,接着使用ioctl将两个流模块“压”到伪终端(ldterm和ptem),以使它的行为与终端相似。与shell管道不同,在shell管道中有许多过滤程序可供选择,并可以创造性地将几百个过滤程序合并起来,流模块一般是以食谱方式工作的——你可以按照文档的要求压模块,和烹调一样,如果仅仅将东西乱糟糟地压入,不会得到好的味道。(不应该将流与fopen、fwrite等使用的标准I/O流概念混淆,两者是完全无关的。)

流特征是SUS1和SUS2要求的,而不是SUS3要求的。将_XOPEN_VERSION设置为500(表明它是SUS2)的Linux不支持流,在这点上是不一致的。

除了下节的伪终端和本书中的一些其他讨论会用到流之外,本书并没有更全面地讲解流内容。从SUS站点或者Sun的站点www.sun.com上可以很容易地找到与流相关的内容,其中Sun的站点上提供了STREAMS Programming Guide。

4.10 伪终端

一般的终端设备驱动程序是将进程与实际的终端相连接,如图4-5a所示,在该图中用户正在终端运行vi程序。就交互(从)进程(vi)而言,伪终端驱动程序的作用和终端是一样的,但是伪终端驱动程序的另一端连接的是主进程,不是实际的设备。这允许主进程将输入添加到从进程,尽管输入来自实际的终端,并且允许捕获从进程的输出,尽管输出数据是要输出到实际的终端。就像平常工作一样,从进程可以使用tcgetattr、tcsetattr和其他仅用于终端的系统调用,并且它们都如期望的那样正确工作。

读者可能会认为将伪终端和进程相连接与使用shell重定向输入和输出是相似的,只是伪终端使用的是一种终端,而不是管道。实际上,vi无法使用管道进行工作:

[⊖] 另外,如果要介绍Curses,就必须介绍X,但那样内容太多了,会累死我们。(这里所说的“我们”是指“你和我”。)

```
$ vi >tmp
ex/vi: Vi's standard input and output must be a terminal
```

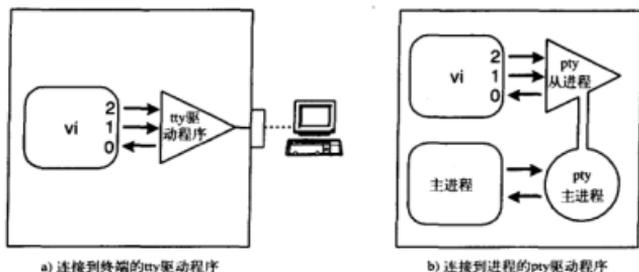


图4-5 终端驱动程序和伪终端驱动程序

也许伪终端的大多数普通应用都要通过telnetd服务器来实现，如图4-6所示。telnetd服务器通过网络与telnet客户端进行通信，通过伪终端与telnet用户进行通信。实际上，telnet会话通常以shell开始，与实际的终端会话相同；在图中，因为用户向shell键入了vi命令，所以vi正在运行。实际上，与使用实际终端相比，现在使用得更广泛的是telnet或者xterm，因为大多数用户是通过网络与UNIX计算机相连接的，而不是使用直接拨号。（用户可能会拨号连接到网络ISP，但那仅仅是为了上因特网。）这里并不打算讲述如何实现telnetd服务器（见练习4.6），我们仅讨论图中与伪终端有关的内容。

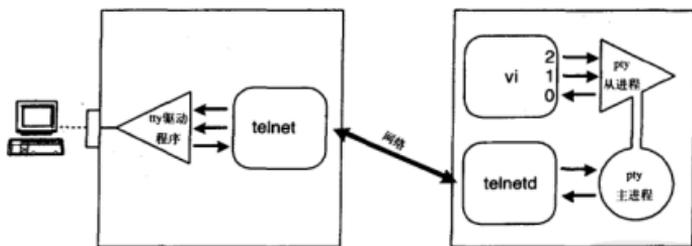


图4-6 telnet服务器 (telnetd) 使用的伪终端

在伪终端的主侧，主进程得到了从进程正在写入的内容。对于vi和其他面向屏幕的程序，这个数据包含转义序列（如清屏），关于该序列见4.8节。终端驱动程序和伪终端都不关心这些序列是什么。另一方面，控制操作（如tcsetattr所做的工作）都完全由驱动程序控制，并且到达不了主进程。因此在图4-6中，vi产生的转义序列自始至终能获得运行在左边计算机上的telnet进程。如果telnet进程运行在全屏方式，它仅将转义序列直接发送到实际终端。更常见的是，telnet进程根本没被连接到终端驱动程序，而是运行在Window系统下。在这种情况下，必须知道如何将转义序列翻译成Window系统显示字符时所要求的方式。这些程序称作终端仿真程序。因此记住终端仿真程序和伪终端是两种不同的东西：第一个处理转义序列，而第二个代替了终端驱动程序，以便将I/O重新路由到主进程。

4.10.1 伪终端库

因为本节使用了第5章的一些系统调用（例如fork），所以在阅读本节之前可能需要先阅读第5章。

和其他设备一样，伪终端（从现在开始称作“pty”）是以特殊文件表示的，文件名随系统不同而不同。不幸的是，将进程与pty连接不像完成下面工作那样直接明了：

```
$ vi </dev/pty01 >/dev/pty01
```

将进程与pty连接是很复杂的，这是因为：必须得到pty的名字，执行一些系统调用为其做准备，并使其成为控制终端。SUS1（见1.5.1节）及其之后的标准对相关的系统调用进行了标准化，但像FreeBSD这样的在SUS之前出现的系统采用了完全不同的处理方式。更糟的是，使用流实现pty系统需要一些额外的步骤，并且在SUS2和SUS3之间还有一些变化。

本书打算将系统之间的差异封装成一个小函数库，每个库的名字都以pt_开头。接着将使用这个pt库实现一个记录/重放应用的例子。

下面是使用pty的总体方案，该方案包括9个步骤：

- 1) 为读和写打开pty的主侧。
- 2) 准备访问pty（下面进行了解释）。
- 3) 从主侧的名字或文件描述符得到从侧的名字，但不打开它。
- 4) 执行fork系统调用（见5.5节）来创建子进程。
- 5) 在子进程中，调用setsid（见4.3.2节）使子进程放弃它的控制终端。
- 6) 在子进程中，打开pty的从侧，使其成为新的控制终端。在支持流的系统（如Solaris）上，必须建立流。在BSD系统上，必须使用以下命令使其成为控制终端：

```
ec_negl( ioctl( fd, TIOCSCTTY ) )
```

- 7) 重新将子进程的标准输入、标准输出以及错误文件描述符定向到pty。
- 8) 执行execvp系统调用（见5.3节），以便让子进程运行所需要的程序（例如vi）。（可以使用“exec”系统调用的6个变体之中的任何一个；在下面的例子中使用的是execvp。）
- 9) 此时，父进程可以使用第1步得到的文件描述符读和写pty的主侧。与通常一样（它对执行过程的上文一无所知），子进程从标准输入中读数据，向标准输出和标准错误输出写数据，并且那些文件描述符就像向终端设备打开一样工作。

以上过程见图4-5b。

有3种方法可以打开pty的主侧（第1步）：

A. 在SUS3系统上只能调用posix_openpty函数（见下面内容）。

B. 在大多数SUS1和SUS2系统上（包括Solaris和Linux），打开克隆文件/dev/ptmx，它会提供一个唯一的pty，尽管你不知道该实际文件的名字，甚至即使有一个但这就足够了，因为你所需要的就是这个打开文件描述符。

C. 在基于BSD的系统中，有一个特殊文件的集合，这些文件的名字形如/dev/ptyXY，其中X和Y是数字或者字母。必须做的事情是尝试打开所有的特殊文件，直到找到能够打开的那个为止。（这不是开玩笑！）

我们可以使用_XOPEN_VERSION宏来区别方法A和方法B，我们将为那些使用方法C的系统调用定义MASTER_NAME_SEARCH宏。在第6步（上面列出的9个步骤中的第6步），我们将

为那些需要建立流的系统调用定义NEED_STREAM_SETUP宏，为需要ioctl调用的系统调用定义NEED_TIOCSCTTY宏。下面是pt库中为Solaris和FreeBSD创建这些宏的代码；Linux不需要这些设置：

```
#if defined(SOLARIS) /* add to this as necessary */
#define NEED_STREAM_SETUP
#endif

#if defined(FREEBSD) /* add to this as necessary */
#define NEED_TIOCSCTTY
#endif

#ifndef _XOPEN_UNIX
#define MASTER_NAME_SEARCH
#endif
```

如果你认为该代码不能满足系统要求，那么可以在此基础上扩充代码。

下面是posix_openpt的对照表，仅在遵循SUS3的系统上可用：

posix_openpt —— 打开pty

```
#include <stdlib.h>
#include <fcntl.h>

int posix_openpt(
    int oflag           /* O_RDWR optionally Ored with O_NOCTTY */
);
/* Returns file descriptor on success or -1 on error (sets errno) */
```

万一进程已经没有控制终端，可以使用标志O_NOCTTY来防止主侧变为控制终端。（回忆以前讲过的内容可以知道，第一个为没有控制终端的进程打开的第一个终端设备将成为控制终端。）尽管我们的确需要从侧成为控制终端（第5步），但通常我们并不希望主侧成为控制终端，因此使用标志O_NOCTTY。

现在，接着讨论pt库代码，它需要一些未包含在defs.h中的代码：

```
#ifndef _XOPEN_UNIX
#include <stropts.h> /* for STREAMS */
#endif
#define NEED_TIOCSCTTY
#include <sys/ttycom.h> /* for TIOCSCTTY */
#endif
```

所有的pt库函数都对PTINFO结构进行操作，该结构包含主侧文件描述符、从侧文件描述符以及它们的路径名，如果知道：

```
#define PT_MAX_NAME 20

typedef struct {
    int pt_fd_m;           /* master file descriptor */
    int pt_fd_s;           /* slave file descriptor */
    char pt_name_m[PT_MAX_NAME]; /* master file name */
    char pt_name_s[PT_MAX_NAME]; /* slave file name */
} PTINFO;

#define PT_GET_MASTER_FD(p) ((p)->pt_fd_m)
#define PT_GET_SLAVE_FD(p) ((p)->pt_fd_s)
```

调用pt库的应用程序使用这两个宏，因为它们需要文件描述符。

稍后要给出的函数pt_open_master可以分配PTINFO结构，并返回指向该结构的指针，以便其他库函数使用该指针，就像标准I/O库使用FILE结构一样。从内部讲，pt_open_master调用find_and_open_master实现了第1步，使用了前面概述的3个方法中的一个：

```

#define defined(MASTER_NAME_SEARCH)
#define PTY_RANGE \
    "0123456789ABCDEFGHIJKLMNQRSTUUVWXYZabcdefghijklmnopqrstuvwxyz"
#define PTY_PROTO "/dev/ptyXY"
#define PTY_X 8
#define PTY_Y 9
#define PTY_MS 5 /* replace with 't' to get slave name */
#endif /* MASTER_NAME_SEARCH */

static bool find_and_open_master(PTINFO *p)
{
    #if defined(_XOPEN_UNIX)
    #if _XOPEN_VERSION >= 600
        p->pt_name_m[0] = '\0'; /* don't know or need name */
        ec_negl( p->pt_fd_m = posix_openpt(O_RDWR | O_NOCTTY) )
    #else
        strcpy(p->pt_name_m, "/dev/ptmx"); /* clone device */
        ec_negl( p->pt_fd_m = open(p->pt_name_m, O_RDWR | O_NOCTTY) )
    #endif
    #elif defined(MASTER_NAME_SEARCH)
        int i, j;
        char proto[] = PTY_PROTO;

        if (p->pt_fd_m != -1) {
            (void)close(p->pt_fd_m);
            p->pt_fd_m = -1;
        }
        for (i = 0; i < sizeof(PTY_RANGE) - 1; i++) {
            proto[PTY_X] = PTY_RANGE[i];
            proto[PTY_Y] = PTY_RANGE[0];
            if (access(proto, F_OK) == -1) {
                if (errno == ENOENT)
                    continue;
                EC_FAIL
            }

            for (j = 0; j < sizeof(PTY_RANGE) - 1; j++) {
                proto[PTY_Y] = PTY_RANGE[j];
                if ((p->pt_fd_m = open(proto, O_RDWR)) == -1) {
                    if (errno == ENOENT)
                        break;
                }
                else {
                    strcpy(p->pt_name_m, proto);
                    break;
                }
            }
            if (p->pt_fd_m != -1)
                break;
        }
        if (p->pt_fd_m == -1) {
            errno = EAGAIN;
            EC_FAIL
        }
    #endif
}

```

```

    }
    #else
        errno = ENOSYS;
        EC_FAIL
    #endif
    return true;

    EC_CLEANUP_BGN
    return false;
    EC_CLEANUP_END
}

```

`find_and_open_master`中的大部分代码是用于愚蠢的名字查找——多达3844个名字！为了使速度稍微快一点，我们使用了`access`（见3.8.1节），用它检测形如`/dev/ptyX0`的每个名字是否存在，如果不存在，就不必查找`/dev/ptyX1`、`/dev/ptyX2`和该系列中的其他59个名字了，直接移到下一个X。

第2步是访问`pty`。在SUS系统上，必须调用`grantpt`来访问从侧，并且当`pty`被锁定时，必须使用`unlockpt`解锁：

grantpt ——取得`pty`从面的访问权限

```

#include <stdlib.h>

int grantpt(
    int fd          /* file descriptor */
);
/* Returns 0 on success or -1 on error (sets errno) */

```

unlockpt ——解锁`pty`

```

#include <stdlib.h>

int unlockpt(
    int fd          /* file descriptor */
);
/* Returns 0 on success or -1 on error (sets errno) */

```

步骤3是为了得到从侧的名字。SUS1系统具有此功能的系统调用：

ptsname ——得到`pty`从面的名字

```

#include <stdlib.h>

char *ptsname(
    int fd          /* file descriptor */
);
/* Returns name or NULL on error (errno not defined) */

```

在BSD系统上（定义了`MASTER_NAME_SEARCH`），通过以“t”替换`/dev/ptyXY`中的“p”可以从主侧的名字得到该名字。即如果发现可以用`/dev/ptyK4`打开主侧，则相应从侧的名字即为`/dev/ttyK4`。

下面是函数`pt_open_master`的代码，该函数调用了前面讨论的步骤1给出的函数`find_and_open_master`，接着进行了步骤2和步骤3，以指定的从侧结束，但该从侧未打开：

```

PTINFO *pt_open_master(void)
{

```

```

PTINFO *p = NULL;
char *s;

ec_null( p = calloc(1, sizeof(PTINFO)) )
p->pt_fd_m = -1;
p->pt_fd_s = -1;
ec_false( find_and_open_master(p) )
#ifdef _XOPEN_UNIX
ec_negl( grantpt(p->pt_fd_m) )
ec_negl( unlockpt(p->pt_fd_m) )
ec_null( s = ptsname(p->pt_fd_m) )
if (strlen(s) >= PT_MAX_NAME) {
    errno = ENAMETOOLONG;
    EC_FAIL
}

strcpy(p->pt_name_s, s);
#elif defined(MASTER_NAME_SEARCH)
strcpy(p->pt_name_s, p->pt_name_m);
p->pt_name_s[PTY_MS] = 't';
#else
errno = ENOSYS;
EC_FAIL
#endif
return p;

EC_CLEANUP_BGN
if (p != NULL) {
    (void)close(p->pt_fd_m);
    (void)close(p->pt_fd_s);
    free(p);
}
return NULL;
EC_CLEANUP_END
}

```

第4步是调用fork创建子进程，该步骤是通过使用库的程序完成的——因为没有库函数能够完成该功能。

第5步调用子进程的setsid（见4.3.2节），使子进程放弃它的控制终端，因为需要pty成为控制终端。第6步打开从侧，也在子进程完成，我们已经有了该从侧的名字。pt_open_slave完成了这两个步骤的工作：

```

bool pt_open_slave(PTINFO *p)
{
    ec_negl( setsid() )
    if (p->pt_fd_s != -1)
        ec_negl( close(p->pt_fd_s) )
    ec_negl( p->pt_fd_s = open(p->pt_name_s, O_RDWR) )
#ifdef defined(NEEDED_TIOCSCTTY)
    ec_negl( ioctl(p->pt_fd_s, TIOCSCTTY, 0) )
#endif
#ifdef defined(NEEDED_STREAM_SETUP)
    ec_negl( ioctl(p->pt_fd_s, I_PUSH, "ptem" ) )
    ec_negl( ioctl(p->pt_fd_s, I_PUSH, "ldterm" ) )
#endif
/*
    Changing mode not that important, so don't fail if it doesn't
    work only because we're not superuser.
*/
}

```



```

    if (fchmod(p->pt_fd_s, PERM_FILE) == -1 && errno != EPERM)
        EC_FAIL
    return true;

    EC_CLEANUP_BGN
    return false;
    EC_CLEANUP_END
}

```

同样，也必须在子进程中调用`pt_open_slave`，不能在父进程中进行。（一会儿我们将看到一个示例。）定义了`NEED_STREAM_SETUP`的系统，例如Solaris，需要在流上压入两个模块：`ptem`和`ldterm`，前者是伪终端仿真程序，后者是普通的终端模块。[⊖]（`ioctl`系统调用和`I_PUSH`命令已经被标准化，而不是用于创建`pty`的模块名。）

在`pt_open_slave`的结尾，我们改变了`pty`的模式，因为在一些系统上可能没有适合的权限。然而，因为进程的用户ID可能没有所有权（与具体的系统相关），所以调用`fchmod`（见3.7.1节）可能会失败，在这种情况下，我们还是希望能继续运行。

步骤7、8和9没有使用`pt`库，使用的是第5、6章介绍的系统调用。在本节末尾的示例中将给出这部分代码。

当子（从）进程和父（主）进程独立运行时，在开始读写第9步的`pty`之前，父进程必须保证子进程已经完全建立。因此，`pt`库中包含一个父进程使用的调用，该调用直到安全运行才返回。

```

bool pt_wait_master(PTINFO *p)
{
    fd_set fd_set_write;

    FD_ZERO(&fd_set_write);
    FD_SET(PT_GET_MASTER_FD(p), &fd_set_write);
    ec_negl( select(PT_GET_MASTER_FD(p) + 1, NULL, &fd_set_write, NULL,
        NULL) )
    return true;

    EC_CLEANUP_BGN
    return false;
    EC_CLEANUP_END
}

```

我们所能做的就是使用`select`（见4.2.3节）等待，直到`pty`可读。

下一个函数是`pt_close_master`，当不再需要`pty`时，父进程会调用它来关闭文件描述符，并释放`PTINFO`结构。

```

bool pt_close_master(PTINFO *p)
{
    ec_negl( close(p->pt_fd_m) )
    free(p);
    return true;

    EC_CLEANUP_BGN
    return false;
    EC_CLEANUP_END
}

```

⊖ 为了在Solaris系统（也可能是其他执行流的系统）上读取这些流，可以执行`man ptem`和`man ldterm`。

还有一个`pt_close_slave`，但通常并不使用它，因为第8步中另一个程序已经覆盖了子进程。同样，因为我们重定向了文件描述符，所以此时pty的子进程的末端是以标准文件描述符(0、1和2)表示的，当进程终止时，这些文件描述符几乎总能自动关闭。代码如下：

```
bool pt_close_slave(PTINFO *p)
{
    (void)close(p->pt_fd_s); /* probably already closed */
    free(p);
    return true;
}
```

如果你发现自己的代码中调用了`pt_close_slave`，那么你的代码可能出错了。

注意`pt_close_master`和`pt_close_slave`都释放了PTINFO结构。本来就应该如此，因为当执行这两个函数时，父进程和子进程都分配了该结构。阅读5.5节中对`fork`的解释后会更加明白。

将以上内容综合起来，可以得到下面pty调用的总体框架：

```
PTINFO *p = NULL;
bool ok = false;

ec_null( p = pt_open_master() ) /* Steps 1, 2, and 3 */
switch (fork()) { /* Step 4 */
case 0:
    ec_false( pt_open_slave(p) ) /* Steps 5 and 6 */
    /*
     * Redirect fds and exec (not shown) - steps 7 and 8
     */
    break;
case -1:
    EC_FAIL
}
ec_false( pt_wait_master(p) ) /* Synchronize before step 9 */
/*
 * Parent (master) now reads and writes pty as desired - step 9
 */
ok = true;
EC_CLEANUP

EC_CLEANUP_BGN
if (p != NULL)
    (void)pt_close_master(p);
/*
 * Other clean-up goes here
 */
exit(ok ? EXIT_SUCCESS : EXIT_FAILURE);
EC_CLEANUP_END
```

这就是所讨论的应用程序的样子。

4.10.2 记录和重放示例

下面将使用pty建立记录/重放系统，它能记录命令的输出然后回放所记录的输出，就好像命令正在运行一样。即不仅屏幕按原样写，而且以同样的速度重放。为了说明我的意思，下面是显示时间、等待5秒并重新显示时间的脚本：

```
$ cat >script1
```

```
date
sleep 5
date
$ chmod +x script1
$ script1
Wed Nov 6 10:46:31 MST 2002 [5 second pause]
Wed Nov 6 10:46:36 MST 2002
```

当然，如从时间所看到的那样，在date的第1次执行和第2次执行之间有5秒暂停。（括号中的注释不是输出的一部分。）

但是如果只是在文件上捕获输出，并显示该文件，那么文件中的文本会以非常快的速度显示。当然，文件中时间完全是date命令写的时间，但是在cat命令打印输出时没有暂停：

```
$ script1 >tmp [5 second pause]
$ cat tmp
Wed Nov 6 10:55:24 MST 2002 [no pause]
Wed Nov 6 10:55:29 MST 2002
```

不过，这不是我们所需要的。我们需要的是记录输出，以便它重放的速度能和记录一样，记录时我们将采用下面名为record的命令（record命令的-p选项可以重放记录）：

```
$ record script1
Wed Nov 6 10:57:18 MST 2002 [5 second pause]
Wed Nov 6 10:57:23 MST 2002
$ record -p
Wed Nov 6 10:57:18 MST 2002 [5 second pause]
Wed Nov 6 10:57:23 MST 2002
```

不能在此页上查看到它，但是在显示第1行之后，显示第2行之前，有5秒暂停，该速度与记录输出的速度相同。

因此，record最低程度上需要完成的事情是保存事件：显示字符的时间，以使重放部分知道以怎样的速度将字符写到标准输出。并且也想让它用交互方式完成该工作，因此不能通过管道捕获输出——因为感兴趣的命令，如vi、emacs和其他全屏应用程序只能工作在终端，所以必须用pty建立命令。记录程序、记录以及所记录的命令如图4-7所示那样连接。无论record从标准输入读取了什么，它都会写到pty，并且无论从pty读了什么，它也都会写到标准输出和事件的文件。

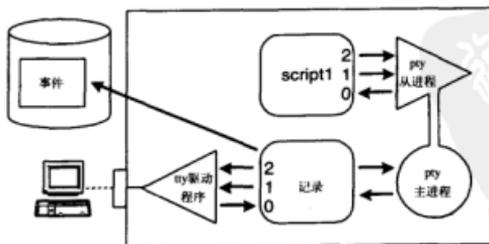


图4-7 用script1实现记录交互

首先，我会给出数据结构和函数，其中函数把命令的输出记录成了一系列事件。然后给出读事件的重放函数，该函数会在合适的时间显示事件中的数据。因为命令实际上没有运行，

所以重放不包含pty。最后，我会给出记录程序，它遵循了前一节介绍的使用了pt库的框架。

每次被记录的进程写数据时，pty的主侧都会读取该数据，并将其当作事件对待。pty仅保存记录开始以来的相对时间和event结构中的数据长度。接着向名为recording.tmp的文件中写入该结构和数据本身。图4-8显示了该文件中的三个此类事件。

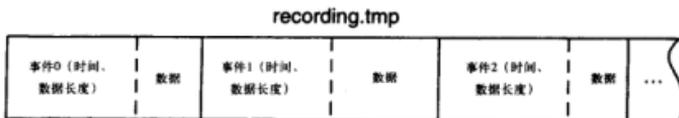


图4-8 记录文件的结构

下面是event结构，以及读写数据所需的全局文件描述符和用于文件名和缓冲区大小的宏：

```
#define EVFILE "recording.tmp"
#define EVBUFSIZE 512

struct event {
    struct timeval e_time;
    unsigned e_dataalen;
};
static int fd_ev = -1;
```

timeval结构在1.7.1节中介绍过了。

记录程序使用ev_creat打开正在记录的文件（如果需要的话创建该文件），ev_write向文件中写事件，并使用ev_close关闭文件：

即使采用了多次调用write，writeall也是写满缓冲区的简单方式；writeall见2.9节。

```
static bool ev_creat(void)
{
    ec_negl( fd_ev = open(EVFILE, O_WRONLY | O_CREAT | O_TRUNC,
        PERM_FILE) );
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

static bool ev_write(char *data, unsigned dataalen)
{
    struct event ev = { { 0 } };

    get_rel_time(&ev.e_time);
    ev.e_dataalen = dataalen;
    ec_negl( writeall(fd_ev, &ev, sizeof(ev)) );
    ec_negl( writeall(fd_ev, data, dataalen) );
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

static void ev_close(void)
{
    (void)close(fd_ev);
```



```

    fd_ev = -1;
}

```

函数 `get_rel_time` 完成的工作是：调用 `gettimeofday`（见 1.7.1 节）得到当前时间，使用另一个函数 `timeval_subtract` 从开始时间（第一次调用保存它）减去当前时间。

```

static void get_rel_time(struct timeval *tv_rel)
{
    static bool first = true;
    static struct timeval starttime;
    struct timeval tv;

    if (first) {
        first = false;
        (void)gettimeofday(&starttime, NULL);
    }
    (void)gettimeofday(&tv, NULL);
    timeval_subtract(&tv, &starttime, tv_rel);
}

static void timeval_subtract(const struct timeval *x,
                             const struct timeval *y, struct timeval *diff)
{
    if (x->tv_sec == y->tv_sec || x->tv_usec >= y->tv_usec) {
        diff->tv_sec = x->tv_sec - y->tv_sec;
        diff->tv_usec = x->tv_usec - y->tv_usec;
    }
    else {
        diff->tv_sec = x->tv_sec - 1 - y->tv_sec;
        diff->tv_usec = 1000000 + x->tv_usec - y->tv_usec;
    }
}

```

注意，函数 `timeval_subtract` 假定 $x > y$ 。

以上是创建记录所需要的所有服务函数。重放记录需要一个有如下功能的函数：打开记录的文件，读取 `event` 结构和在文件中紧随其后的数据：

```

static bool ev_open(void)
{
    ec_negl( fd_ev = open(EVFILE, O_RDONLY) )
    return true;
}

EC_CLEANUP_BGN
return false;
EC_CLEANUP_END
}

static bool ev_read(struct event *ev, char *data, unsigned datalen)
{
    ssize_t nread;

    ec_negl( nread = read(fd_ev, ev, sizeof(*ev)) )
    if (nread != sizeof(*ev)) {
        errno = EIO;
        EC_FAIL
    }
    ec_negl( nread = read(fd_ev, data, ev->e_datalen) )
}

```



```

    if (nread != ev->e_dataalen) {
        errno = EIO;
        EC_FAIL
    }
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

```

另外一个事情是：一旦重放程序有了事件，在显示该事件之前需要等待合适的时间。这是实时重放和仅仅转储数据之间的基本区别。函数`ev_sleep`从`event`结构中获得时间，计算等待的时间，接着休眠该时间间隔。当`timeval`以秒和微妙保存该时间时，对于以秒保存的可以用标准C函数中的`sleep`，对于用微妙保存的可以用`usleep`系统调用（见9.7.3节）。

```

static bool ev_sleep(struct timeval *tv)
{
    struct timeval tv_rel, tv_diff;

    get_rel_time(&tv_rel);
    if (tv->tv_sec > tv_rel.tv_sec ||
        (tv->tv_sec == tv_rel.tv_sec && tv->tv_usec >= tv_rel.tv_usec)) {
        timeval_subtract(tv, &tv_rel, &tv_diff);
        (void)sleep(tv_diff.tv_sec);
        ec_negl( usleep(tv_diff.tv_usec) )
    }
    /* else we are already running late */
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

```

这就是需要记录和重放的所有事件。假设已经完成了记录，下面是重放的函数：

```

static bool playback(void)
{
    bool ok = false;
    struct event ev;
    char buf[EVBUFFSIZE];
    struct termios tbuf, tbufsave;

    ec_negl( tcgetattr(STDIN_FILENO, &tbuf) )
    tbufsave = tbuf;
    tbuf.c_lflag &= ~ECHO;
    ec_negl( tcsetattr(STDIN_FILENO, TCSAFLUSH, &tbuf) )
    ec_false( ev_open() )
    while (true) {
        ec_false( ev_read(&ev, buf, sizeof(buf)) )
        if (ev.e_dataalen == 0)
            break;
        ev_sleep(&ev.e_time);
        ec_negl( writeall(STDOUT_FILENO, buf, ev.e_dataalen) )
    }
    ec_negl( write(STDOUT_FILENO, "\n", 1) )
}

```



```

ok = true;
EC_CLEANUP

EC_CLEANUP_BGN
(void)tcdrain(STDOUT_FILENO);
(void)sleep(1); /* Give the terminal a chance to respond. */
(void)tcsetattr(STDIN_FILENO, TCSAFLUSH, &tbufsave);
ev_close();
return ok;
EC_CLEANUP_END
}

```

在该函数中，事件的读取和向STDOUT_FILENO写入都非常直接明了。写入终端的某些转义序列会引起终端响应，但是在我们不关心重放期间的响应时（无论响应的内容如何都已经捕获在记录中），我们必须关闭回送，以便保证响应不会与输出混淆。在结尾，我们调用tcdrain以发送最后的输出，等待1秒钟时间以使终端对其进行处理，然后在恢复终端的属性时，刷新了所有剩余的输入。函数tcgetattr、tcsetattr和tcdrain见4.5.1节和4.6节。

现在，我们可以使用前一节给出的pt库和“ev”程序进行记录了。（你可能需要复习一下前面一节结尾所讲的框架。）下面分几块来讨论main函数；开头如下：

```

int main(int argc, char *argv[])
{
    bool ok = false;
    PTINFO *p = NULL;

    if (argc < 2) {
        fprintf(stderr, "Usage: record cmd ...\n      record -p\n");
        exit(EXIT_FAILURE);
    }
    if (strcmp(argv[1], "-p") == 0) {
        playback();
        ok = true;
        EC_CLEANUP
    }
}

```

如果使用-p选项调用该函数，那么它将只调用已经讨论过的playback函数并退出。否则，记录时遵循如下框架：

```

ec_null( p = pt_open_master() )
switch (fork()) {
case 0:
    ec_false( pt_open_slave(p) )
    ec_false( exec_redirected(argv[1], &argv[1], PT_GET_SLAVE_FD(p),
        PT_GET_SLAVE_FD(p), PT_GET_SLAVE_FD(p)) )
    break;
case -1:
    EC_FAIL
}
ec_false( ev_creat() )
ec_false( pt_wait_master(p) )

```

因为打算到第5章和第6章才一起讨论与重定向文件描述符和执行命令有关的内容，因此这里将这些内容都放在了函数exec_redirected中。下面将给出exec_redirected的代码，但不对其进行解释。如果需要，可以先阅读那些章节，回头再来看这个函数的代码。

目前，main中的代码已经到了第9步（前一节介绍了该步骤），该步需要读写pty主侧的文

件描述符。记住主命令（`script1`、`vi`或者其他命令）已经在运行了，并且可能会因为`read`等待一些认为是终端的输入，而使主命令阻塞。下面是`main`函数的剩余部分：

```
tc_setraw();
while (true) {
    fd_set fd_set_read;
    char buf[EVBUFFER_SIZE];
    ssize_t nread;

    FD_ZERO(&fd_set_read);
    FD_SET(STDIN_FILENO, &fd_set_read);
    FD_SET(PT_GET_MASTER_FD(p), &fd_set_read);
    ec_negl( select(FD_SETSIZE, &fd_set_read, NULL, NULL, NULL) )
    if (FD_ISSET(STDIN_FILENO, &fd_set_read)) {
        ec_negl( nread = read(STDIN_FILENO, &buf, sizeof(buf)) )
        ec_negl( writeall(PT_GET_MASTER_FD(p), buf, nread) )
    }
    if (FD_ISSET(PT_GET_MASTER_FD(p), &fd_set_read)) {
        if ((nread = read(PT_GET_MASTER_FD(p), &buf,
            sizeof(buf))) > 0) {
            ec_false( ev_write(buf, nread) )
            ec_negl( writeall(STDOUT_FILENO, buf, nread) )
        }
        else if (nread == 0 || (nread == -1 && errno == EIO))
            break;
        else
            EC_FAIL
    }
}
ec_false( ev_write(NULL, 0) )
fprintf(stderr,
    "EOF or error reading stdin or master pseudo-terminal; exiting\n");
ok = true;
    EC_CLEANUP

EC_CLEANUP_BGN
    if (p != NULL)
        (void)pt_close_master(p);
    tc_restore();
    ev_close();
    printf("\n");
    exit(ok ? EXIT_SUCCESS : EXIT_FAILURE);
EC_CLEANUP_END
}
```

对`main`函数的最后部分的解释如下：

- 因为`record`大多用于面向屏幕的命令，所以我们使用`tc_setraw`（见4.5.10节）将终端设置为原始模式。清除代码时，通过`tc_restore`恢复了属性。因为我们只是想关闭回送，所以没有使用`playback`中的那些函数。
- `record`有两个输入：用户输入的内容和`pty`返回的内容。如上节图4-5所示。`record`使用`select`等待，直到两个输入中的一个有了一些数据，接着处理这些数据，并循环返回到`select`。注意因为`select`需要改变位来报告结果，所以`record`每次都需要建立`fd_set_read`。
- 当到达文件结尾，或者出现来自`pty`的而不是`record`的标准输入I/O错误时，将退出循环。别忘了，毕竟到了主命令，而不是我们决定做什么的时间。文件结尾或I/O错误意

意味着pty的从侧已经关闭了它所有的文件描述符；即主命令已经终止。具体用哪一个指示项要取决于所使用的系统，因此我们任取了两者之一。

最后给出exec_redirected的代码，若你已经阅读了第6章，那么一看就能明白：

```
bool exec_redirected(const char *file, char *const argv[], int fd_stdin,
int fd_stdout, int fd_stderr)
{
    if (fd_stdin != STDIN_FILENO)
        ec_negl( dup2(fd_stdin, STDIN_FILENO) )
    if (fd_stdout != STDOUT_FILENO)
        ec_negl( dup2(fd_stdout, STDOUT_FILENO) )
    if (fd_stderr != STDERR_FILENO)
        ec_negl( dup2(fd_stderr, STDERR_FILENO) )
    if (fd_stdin != STDIN_FILENO)
        (void)close(fd_stdin);
    if (fd_stdout != STDOUT_FILENO)
        (void)close(fd_stdout);
    if (fd_stderr != STDERR_FILENO)
        (void)close(fd_stderr);
    ec_negl( execvp(file, argv) )

    EC_CLEANUP_BGN
        return false;
    EC_CLEANUP_END
}
```

在书中，很难真正地理解record的活动——通过视频会使其功能更加明了。自己亲手做一做！当以与原记录相同的效果重放记录时——就像观看键盘上的克隆一样，你一定会满意的。

练习

- 4.1 修改getln以返回以EOF结尾的行，如4.2.1节所讨论的那样。
- 4.2 实现Bfdopen，如4.2.1节所讨论的那样。
- 4.3 实现一个stty命令的简化版本，仅输出当前终端的状态。
- 4.4 实现一个stty命令的版本，仅允许用户规定10个最常用的操作数，你必须决定这10个操作数。
- 4.5 实现一个简单的屏幕编辑器。决定一个允许合理编辑，但不引起过多警报的函数的小选择集。在内存中进行文本编辑，使用Curses（或者等效函数）更新屏幕，读取键盘。如果可以，使用联机资料宏写一些文档。
- 4.6 实现telnetd服务器。对于它应该做到的完整细节见[RFC854]。但是，在本次练习中，只要能从现在开始运行telnet的另一台机器上连接就可以实现要求，接着通过你的服务器，执行一些面向行的命令和vi或者emacs。

第5章 进程和线程

5.1 概述

现在离开输入和输出这个主题，开始研究UNIX的多任务特性。本章将讲解使用exec、fork、wait以及相关的系统调用来调用程序和进程的技术。下一章将介绍使用管道进行简单的进程间通信。第7章和第8章将继续介绍更高级的进程间通信机制。

本主题的介绍是通过实现一个相当完整的命令解释器或shell来组织的。从一个很少使用的、功能有限的shell开始，然后逐步地增加特性，直到在下一章能够实现一个可以处理I/O重定向、管道、后台进程、引用参数和环境变量的shell为止。

5.2 环境

首先讨论大多数UNIX用户都已熟知的shell级别的环境。当执行一个UNIX程序时，程序会从调用它的进程处接收到两个数据集：参数和环境。对于C程序而言，它们形式上都是字符指针的数组，除了数组的最后一个字符指针指向一个NULL终止字符串外。最后一个指针是NULL，参数的长度也会被传递。其他语言使用的是不同的接口，但是这里仅关注C和C++。

C或C++程序以下面两种方式之一开始：[⊖]

main —— C或C++程序入口点

```
int main(  
    int argc,           /* argument count */  
    char *argv[]       /* array of argument strings */  
)  
  
int main(void)
```

计数参数argc不包括终止argv数组的NULL指针。如果程序没有使用参数，那么可以省略计数参数和数组，如第二种形式一样。本书中已经给出了这两种形式的示例。

另外，全局变量environ指向的是环境字符串数组，该数组也是以NULL结束的（没有相关的计数变量）：

environ —— 环境字符串

```
extern char **environ; /* environment array (not in any header) */
```

每个参数字符串本质上都可以是任意符号，只要是以NULL结束的就可以。环境字符串的限制更多一些，每个环境字符串都要具有name=value的形式，并且每个值后都应带有NULL终止符。当然名字不包括“=”这个字符。

在5.3节中将讲述环境是怎样被传递给main的（使environ被设置）；在这里只讨论在environ中检索值和修改environ的问题。

[⊖] 一些实现允许有第三个包含环境变量的字符串数组参数，但这既不标准也不必要。

获得环境的方法之一是直接访问`environ`，代码如下：

```
extern char **environ;

int main(void)
{
    int i;

    for (i = 0; environ[i] != NULL; i++)
        printf("%s\n", environ[i]);
    exit(EXIT_SUCCESS);
}
```

下面是Solaris中输出的部分结果（为了节约空间，省略了其余的输出部分）：

```
HOME=/home/marc
HZ=100
LC_COLLATE=en_US.ISO8859-1
LC_CTYPE=en_US.ISO8859-1
LC_MESSAGES=C
LC_MONETARY=en_US.ISO8859-1
LOGNAME=marc
MAIL=/var/mail/marc
```

一般不要求列出所有的环境，通常程序需要的是某个特定变量的值，标准C函数`getenv`可以实现此功能：

getenv——得到环境变量值

```
#include <stdlib.h>

char *getenv(
    const char *var      /* variable to find */
);
/* Returns value or NULL if not found (errno not defined) */
```

`getenv`仅仅返回了环境变量的值，即“=”号的右侧部分，如下例所示：

```
int main(void)
{
    char *s;

    s = getenv("LOGNAME");

    if (s == NULL)
        printf("variable not found\n");
    else
        printf("value is \"%s\"\n", s);
    exit(EXIT_SUCCESS);
}
```

其输出结果如下：

```
value is "marc"
```

更新环境并不像读环境那样简单。尽管其占用的内存段具有进程独占的特性，并且可以被任意修改，但仍不能保证为任意新变量或较长的值提供超额的内存空间。所以除非更新很少，否则必须重新创建一个全新的环境。如果使指针`environ`指向了这个新环境，那么它会被传递到随后调用的任何程序中（见5.3节），并且也会被随后对`getenv`的调用所使用。任何更新都不会影响到其他任何进程，包括调用进程进行更新的shell（或诸如此类）在内。因此，如

果想修改shell的环境，那么必须得把命令建立到shell中。

与其直接干预环境，不如使用一些标准函数：

putenv —— 改变或加入环境变量

```
#include <stdlib.h>

int putenv(
    char *string          /* string of form name=value */
);
/* Returns 0 on success or non-zero on error (sets errno) */
```

setenv —— 改变或加入环境变量

```
#include <stdlib.h>

int setenv(
    const char *var,      /* variable to be changed or added */
    const char *val,     /* value */
    int overwrite        /* overwrite? */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

unsetenv —— 删除环境变量

```
#include <stdlib.h>

int unsetenv(
    const char *var      /* variable to be removed */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

所有的函数都修改environ所指的存储空间，如果指针数组不够长，也可以把environ设置成一个新值。如果用户自己修改了environ或者环境指向的任何部分，那么这些函数的行为会变成未定义的，所以用户要决定是自己亲自修改还是使用函数，但不要混用这两种方法。

putenv将形如name=value的完整环境字符串当成了参数，并且让environ数组中的一个指针指向了所传递进来的存储空间，接着该存储空间也成了环境的一部分，所以不要传递任何自动分配的数据（本地的、非静态变量），也不要再在调用putenv后修改该字符串。

setenv更复杂：它复制传递进来的变量的名和值，并为它分配单独的存储空间。如果变量已经存在，而且第三个overwrite参数非零时，则修改它的值；否则，保持旧值不变。如果变量不存在，那么不管overwrite参数的值是什么，都将它添加到环境。

unsetenv是为setenv设置的，它的功能是从环境中移去变量及其值。如果系统中没有unsetenv，从环境中移去变量最好的方法是设置变量的值为空字符串。应用程序可能会接受这种操作，也可能不接受这种操作，这要看具体的应用而定。（一些系统，如Linux、FreeBSD和Darwin，是将unsetenv定义成void函数，因此没有错误返回。）

尽管这些函数的接口实现了标准化，但这些函数也不是必须的。FreeBSD、Linux和Solaris都有putenv，而前两个系统都具有这3个函数。SUS3需要setenv和unsetenv，但写本书时，就不再有任何SUS3系统了，因此没有一个简单的、方便的方法决定保留哪些函数。setenv和unsetenv来自BSD，它们存在于从BSD派生的任何系统中，包括FreeBSD在内；putenv来自System V系统，它们存在于从System V派生的任何系统中，包括Solaris。既然SUS推荐setenv和unsetenv，或许最好的方法是编程实现这些函数，然后将自己的实现包含到仍缺少这些函数的系统中。

如果不考虑内存泄露，那么实现setenv和unsetenv是很容易的。就unsetenv来说，问题在于它们可能必须分配内存或使内存孤立，但由于它们对先前怎样分配内存的情况毫无所知，因此不能释放内存。尽管存在那样的缺点，但仍有如下的setenv版本：

```
int setenv(const char *var, const char *val, int overwrite)
{
    int i;
    size_t varlen;
    char **e;

    if (var == NULL || val == NULL || var[0] == '\0' ||
        strchr(var, '=') != NULL) {
        errno = EINVAL;
        return -1;
    }
    varlen = strlen(var);
    for (i = 0; environ[i] != NULL; i++)
        if (strncmp(environ[i], var, varlen) == 0 &&
            environ[i][varlen] == '=')
            break;

    if (environ[i] == NULL) {
        if ((e = malloc((i + 2) * sizeof(char *))) == NULL)
            return -1;
        memcpy(e, environ, i * sizeof(char *));
        /* possible memory leaks with old pointer array */
        environ = e;
        environ[i + 1] = NULL;
        return setnew(i, var, val);
    }
    else {
        if (overwrite) {
            if (strlen(&environ[i][varlen + 1]) >= strlen(val)) {
                strcpy(&environ[i][varlen + 1], val);
                return 0;
            }
            return setnew(i, var, val);
        }
        return 0;
    }
}
```

对该函数的注释如下：

- 要做的第一件事是检查SUS所要求的参数。
- 接下来检查环境来看看是否已经定义了变量。注意必须寻找那些在名字后面跟着“=”号的字符串。
- 如果找到了变量，并且如果overwrite参数的值是false，则退出。否则有两种情况：在新值满足条件的情况下，仅需要把它复制进来，在不满足条件时，只需在调用函数setnew（见下面）时将它放进去。
- 如果没有找到变量，那么必须增长数组。像先说过的那样，因为并不知道旧的内存是如何分配的，所以不能使用realloc。因此用户自己可以使用malloc，并复制旧内容。然后用NULL指针终止新数组，并调用setnew放入新的条目。

下面是setnew，它的功能是为名字、=号和值分配空间，并把它存于数组：

```
static int setnew(int i, const char *var, const char *val)
{
```

```

char *s;

if ((s = malloc(strlen(var) + 1 + strlen(val) + 1)) == NULL)
    return -1;
strcpy(s, var);
strcat(s, "=");
strcat(s, val);
/* possible memory leak with old value of environ[i] */
environ[i] = s;
return 0;
}

```

最后，使用`unsetenv`检查参数，查找它，如果它存在，则向下移动数组的余下部分以有效地移除它。这也可能会造成内存泄露，因为不能够假定未设定变量的内存能被释放。

```

int unsetenv(const char *var)
{
    int i, found = -1;
    size_t varlen;

    if (var == NULL || var[0] == '\0' || strchr(var, '=') != NULL) {
        errno = EINVAL;
        return -1;
    }
    varlen = strlen(var);
    for (i = 0; environ[i] != NULL; i++)
        if (strncmp(environ[i], var, varlen) == 0 &&
            environ[i][varlen] == '=')
            found = i;
    if (found != -1)
        /* possible memory leak with old value of environ[found] */
        memmove(&environ[found], &environ[found + 1],
            (i - found) * sizeof(char *));
    return 0;
}

```

注意，这里使用的是`memmove`，而不是`memcpy`，因为如果源和目标内存空间重叠，那么用`memmove`可以保证前者正常工作。

你可能会奇怪为什么不在这些函数中使用“cc”错误检查宏，原因是想让这些行为尽可能地接近标准函数。

在`setenv`和`unsetenv`中纠正内存泄露问题是非常简单的；见练习5.1。

5.3 exec系统调用

在没有完全理解进程和程序的区别以前，要理解`exec`或`fork`调用是不可能的。如果对这些术语不熟悉，可以复习1.1.2节的内容。如果现在准备继续，那么这里我们用一句话来总结一下两者的不同：进程是一个执行环境，这个环境包含了指令、用户数据和系统数据段，也包括运行时需要的许多其他资源，而程序则是一个文件，该文件包含了指令和用于初始化该指令以及进程用户数据段的数据。

`exec`系统调用从指定程序重新初始化进程；虽然进程还在，但程序已经改变了。另一方面，`fork`系统调用（5.5节的主题）仅通过复制指令、用户数据和系统数据段来创建从现存进程克隆的新进程；该新进程不是从程序初始化得来的，所以旧进程和新进程执行同样的指令。

个别情况下，`fork`和`exec`的使用是有限制的，但大多数时候，可以同时使用它们。在本

书单独介绍它们时，记住这些就行了，但如果认为它们是无用的，那么也不用担心——只要试着理解它们的功能就行了。在5.4节中，当一起使用它们时，会看到它们是强有力的一对。

除了启动UNIX内核本身以外，`exec`是程序在UNIX上获得执行的唯一方法。不仅shell使用`exec`执行程序，而且shell和其祖先shell也会被`exec`调用。此外`fork`是创建新进程的唯一方式。

事实上，没有叫做“`exec`”的系统调用。所谓的“`exec`”系统调用是6个以`execAB`形式命名的调用，这里A是指`l`或`v`，这依赖于参数是直接调用（列表）中还是在数组中（向量），而B要么没有，要么为`p`，`p`表示应该使用`PATH`环境变量查找程序，要么为`e`，`e`表示将被使用的特定环境。（此外，在同一个调用中用户不能同时获得特征`p`和`e`。）因此，6个名字分别为`execl`、`execv`、`execlp`、`execvp`、`execle`和`execve`。^①首先从`execl`讲起，然后再介绍其他5个。

execl——执行带参数列表的文件

```
#include <unistd.h>

int execl(
    const char *path,          /* program pathname */
    const char *arg0,         /* first arg (file name) */
    const char *arg1,         /* second arg (if needed) */
    ...,                      /* remaining args (if needed) */
    (char *) NULL,           /* arg list terminator */
);
/* Returns -1 on error (sets errno) */
```

`path`参数必须命名为一个可由有效用户ID（比如说，模式755）执行的程序文件，而且可执行程序的内容要正确。通过重新初始化栈，来自程序的指令覆盖了进程的指令段，并且来自程序的数据也覆盖了进程的用户数据段。然后进程从顶端执行该新程序（也就是说，调用了它的`main`函数）。

因为成功的`execl`的返回位置已经没了，所以可以没有返回值。若`execl`没有成功，则必须返回-1，但没有必要测试该值，因为没有其他可能的值。`execl`不成功的最常见原因是路径不存在，或不可以执行。

跟在`path`后面的`execl`参数被收入了字符指针数组，最后一个参数的值必须是`NULL`，用来停止收入和终止数组。按常规，第一个参数都是程序文件的名字（并不是整个路径）。新程序可以通过与`main`的`argc`和`argv`相似的参数访问这些参数。也传递了由`environ`指向的环境，并且通过新程序的`environ`指针或使用`getenv`可以访问它，就像先前章节中所解释的那样。

因为进程仍旧存在，并且因为它的系统数据段几乎没有被破坏，所以几乎所有的进程属性都没有改变，这些属性包括它的进程ID、父进程ID、进程组ID、会话ID、控制终端、实际用户ID、实际组ID、当前目录、根目录、优先级和累计的执行次数，通常还包括打开文件描述符。想要列出那些确实改变了的重要属性就更容易了，原因如下：

- 如果进程已经安排捕获某个信号，而被委派捕获信号的指令已经消失，那么信号将被重置为默认动作。被忽略或默认的信号保存原有方式。（有关信号的更多内容见第9章。）
- 如果新程序文件的设置用户ID位或设置组ID位是打开的，那么有效用户ID或有效组ID将被改成文件的所有者ID或组ID。如果它们本身与实际ID不相符，那么也无法重新获

① 用2个系统调用而非6个系统调用就很容易提供相同特性，但变化稍微有点晚。见练习5.6。

得先前的有效ID了。

- 任何用`atexit`（见1.3.4节）注册的函数都可以被取消注册，因为它们已经没有代码了。
- 共享内存段（见7.12节）将被断开（取消映射），因为连接点已经失效。
- 关闭POSIX的命名信号量（见7.10节）。System V的信号量（见7.9节）不受影响。

附录A给出了完整列表。但应该知道：如果保留的某个属性或资源对整个运行的新程序而言是没有意义的，那么它会被重置为默认的或被关闭。

为了说明如何使用`execl`，下面给出一个精心设计的例子：

```
void exectest(void)
{
    printf("The quick brown fox jumped over ");
    ec_neg1( execl("/bin/echo", "echo", "the", "lazy", "dogs.", (char *) NULL) );
    return;

    EC_CLEANUP_BGN
        EC_FLUSH("exectest");
    EC_CLEANUP_END
}
```

下面是调用该函数得到的输出：

```
the lazy dogs.
```

fox怎么了？噢，原来是它不够快：标准I/O库（`printf`是该库的一部分）缓冲了它的输出，并且当进程退出时，自动刷新了所有最近的不完全满的缓冲区。但调用`execl`之前，进程不会退出，并且作为用户数据段的缓冲区在被刷新之前会被覆盖。

该问题既可以通过强制不要缓冲输出来解决，像这样：

```
setbuf(stdout, NULL);
```

也可以在调用`execl`之前刷新缓冲区，像这样：

```
fflush(stdout);
```

像前面讲过的，打开文件描述符通常借助`execl`保持打开状态。如果不希望文件描述符保持打开状态，可以首先使用`close`关闭它。但有的时候，这行不通。假定正在调用一个要求关闭所有文件描述符的程序（一个非常常见的要求），那么可以试试这样：

```
errno = 0;
ec_neg1( open_max = sysconf(_SC_OPEN_MAX) )
for ( fd = 0; fd < open_max; fd++)
    (void)close(fd); /* ignore errors */
ec_neg1( _execl(path, arg0, arg1, arg2, (char *) NULL) )
```

调用`sysconf`（见1.5.5节）能够获得一个进程可以打开的最大文件个数，而最高的文件描述符编号也比这个数目小1。最坏的打算是在关闭所有的文件描述符，不管文件描述符是否都是打开的，都要关闭它们。这是个少见的示例，它忽略了系统调用返回的错误，这样做是恰当的。

如果`execl`成功，那么一切都好，但如果它失败了，那么也不会看到错误消息，因为文件描述符2已经被关闭了！这里可以使用3.8.3节遇到的执行关闭标志，当用`fcntl`设置时，在`execl`成功的情况下会关闭文件描述符，否则不管它。我们要么仅仅为重要的文件描述符设置该标志，要么就为所有的都设置，像这样：

```

for (fd = 0; fd < open_max ; fd++) {
    ec_negl( flags = fcntl(fd, F_GETFD) )
    ec_negl( fcntl(fd, F_SETFD, flags | FD_CLOEXEC) )
}
ec_negl( execl(path, arg0, arg1, arg2, (char *) NULL) )

```

对所有文件描述符进行关闭或设置执行关闭的一个问题是：可能会有一千个甚至更多的文件描述符，这就需要进行大量的处理工作，并且大多数处理是浪费的。同时，通过exec，除了让标准文件描述符打开外，让其他任何描述符都保持打开是不对的，除非新程序所期望的多于这三个标准文件描述符，而这是不常见的。因此在大多数情况下，需要跟踪程序打开了什么样的文件描述符，并且要在调用exec之前明确地关闭它们。

因为标准文件描述符通常保持在打开状态，所以很少使用执行关闭。其有用的一个例子是将打开的文件描述符用作日志文件，在exec失败时可以通过打开的文件描述符来记录事实。

其他5个exec系统调用提供了3个execl没有的特性：

- 将参数放到数组中而不是显式地将它们列出来。当不知道编译阶段的参数个数时，这是非常必要的，就像编写一个shell程序一样。
- 使用PATH环境变量值查找程序文件，如shell所做的那样。
- 手动传递一个明确的环境指针，而不是自动使用environ。因为成功的exec将覆盖现存的环境，所以这个特性优越性并不明显。^①

下面是还没有介绍的exec变量的对照表：

execv——执行带参数向量的文件

```

#include <unistd.h>

int execv(
    const char *path,      /* program pathname */
    char *const argv[]    /* argument vector */
);
/* Returns -1 on error (sets errno) */

```

execlp——执行带参数列表和路径搜索的文件

```

#include <unistd.h>

int execlp(
    const char *file,      /* program file name */
    const char *arg0,     /* first arg (file name) */
    const char *arg1,     /* second arg (if needed) */
    ...,                 /* remaining args (if needed) */
    (char *) NULL        /* arg list terminator */
);
/* Returns -1 on error (sets errno) */

```

execvp——执行带参数向量和路径搜索的文件

```

#include <unistd.h>

int execvp(
    const char *file,      /* program file name */
    char *const argv[]    /* argument vector */
);
/* Returns -1 on error (sets errno) */

```

① 但是见练习5.2。

execl——执行带参数列表和环境变量的文件

```
#include <unistd.h>

int execl(
    const char *path,      /* program pathname */
    const char *arg0,     /* first arg (file name) */
    const char *arg1,     /* second arg (if needed) */
    ...,                  /* remaining args (if needed) */
    (char*)NULL,         /* arg list terminator */
    char *const envv[]    /* environment vector */
);
/* Returns -1 on error (sets errno) */
```

execve——执行带参数向量和环境变量的文件

```
#include <unistd.h>

int execve(
    const char *path,     /* program pathname */
    char *const argv[],  /* argument vector */
    char *const envv[]   /* environment vector */
);
/* Returns -1 on error (sets errno) */
```

注意这里使用的argv参数和main函数的argv参数的设计是相同的。不要忘记最后一个指针必须是NULL。

如果execlp或execvp的file参数没有斜线，则把PATH变量值中所列出的字符串一个地与之比较，直到定位到一个带有结果路径名的普通文件，其中结果路径名具有执行权限。如果这个文件包含一个程序（在它的第一个字处由一个代码号指示的），便执行它。如果不包含，则将其假定为一个脚本文件；通常为了运行这个脚本文件，都把路径作为shell的第一个参数来执行shell。

例如，如果file参数的值是echo，并且PATH字符串查找到了一个可执行路径是/bin/echo的文件，那么以程序方式执行该文件，因为按照正确的格式其包含二进制指令。但如果file参数是（比如）myscript，且查找到的是/home/marc/myscript（不是可执行的二进制文件），那么execlp或execvp就执行如下代码：

```
sh /home/marc/myscript arg1 arg2 ...
```

这取决于利用什么样的shell来实现，但必须得是标准确认的。

另一个普遍支持的，但并不是标准的规范是：如果脚本的第一行具有如下形式：

```
#! pathname [arg]
```

则pathname指定的解释器会代替shell执行，它使用脚本文件的路径作为第一个参数。如果在#!行上有可选项arg，那么它会成为程序的第一个参数，而脚本路径名将变成第二个参数。解释器必须略过#!行，这就是为什么UNIX脚本语言使用#开始一行注释的原因了。

例如，可以在文件mycmd中放置如下的脚本：

```
#! /usr/bin/python2.2
print "Hello World!"
```

使它成为可执行的，然后如下执行它：

```
$ mycmd
Hello World!
$
```

几乎在所有的系统中，都是用execvp来处理#!行的，而不是用shell本身来处理。

如果PATH查找的结果证明没有任何内容是可执行的，则exec失败。如果文件参数中有“/”，就不用进行查找——认为完成了路径查找。但是它仍然可能是一个脚本文件。

把execvp和execlp编程为调用execv或execl的库函数是可行的，并且这样做也可以了解很多东西。这里给出了一个和标准版本接近的execvp版本，但该版本使用了“ec”错误检查宏。

回顾可知，对于每个通过查找PATH得到的路径，execvp首先必须尝试将其作为可执行二进制文件来执行，然后再将其作为shell脚本来执行。下面是完成这个功能的函数（省略了#!特征）：

```
int exec_path(const char *path, char *const argv[], char *newargv[])
{
    int i;

    execv(path, argv);
    if (errno == ENOEXEC) {
        newargv[0] = argv[0];
        newargv[1] = (char *)path;
        i = 1;
        do {
            newargv[i + 1] = argv[i];
        } while (argv[i++] != NULL);
        return execv("/bin/sh", (char *const *)newargv);
    }
    return -1;
}
```

如果这两种方式执行路径失败，则exec_path返回-1，并设置errno。

这里的execvp版本称作execvp2，每次调用exec_path时，都必须轮流尝试PATH中的每一个路径，除非传递的文件包含“/”或PATH不存在。在这种情况下可以仅采用如下的方式来使用：

```
int execvp2(const char *file, char *const argv[])
{
    char *s, *pathseq = NULL, *path = NULL, **newargv = NULL;
    int argc;

    for (argc = 0; argv[argc] != NULL; argc++)
        ;
    /* If shell script, we'll need room for one additional arg and NULL. */
    ec_null( newargv = malloc((argc + 2) * sizeof(char *)) )
    s = getenv("PATH");
    if (strchr(file, '/') != NULL || s == NULL || s[0] == '\0')
        ec_neg1( exec_path(file, argv, newargv) )
    ec_null( pathseq = strdup(s) )
    /* Following line usually allocates too much */
    ec_null( path = malloc(strlen(file) + strlen(pathseq) + 2) )
    while ((s = strtok(pathseq, ";")) != NULL) {
        pathseq = NULL; /* tell strtok to keep going */
        if (s[0] == '\0')
            s = ".";
        strcpy(path, s);
        strcat(path, "/");
        strcat(path, file);
        exec_path(path, argv, newargv);
    }
}
```

```

    }
    errno = ENOENT;
    EC_FAIL

EC_CLEANUP_BGN
    free(pathseq);
    free(path);
    free(newargv);
    return -1;
EC_CLEANUP_END
}

```

下面是一些关于内存分配的解释：

- 无论exec成功与否，要释放分配了的内存都几乎是不必要的，也是不可能的，因为所有的用户数据都将被覆盖。
- 将PATH的长度、传入的文件名的长度以及“/”和NUL的两个字节长度都加到一起，所得到的长度对于每个要分析的路径来说都肯定够用了。
- 如果想要尝试执行脚本文件，那么需要一个比传入的参数向量要多一个槽的参数向量。因此要计算传入的参数个数（在for循环中），然后使用比其多两个槽的槽数来分配向量（一个用于附加的参数——shell要执行的路径名，另一个用于结尾的NULL指针）。
- 这里使用了strdup来分配新字符以容纳PATH的值，因为strtok将向其写入。从getenv直接得到的指针指向了该环境，不需直接修改这些字符串，以防execvp2失败。（当调用该函数时，必须把strtok的第一个参数设置为NULL——这样它可以知道怎样继续扫描初始字符。）

如上所说，通常exec和fork是配对使用的，但偶尔单独使用的情况也有。有时候大的程序会被分割成几段来执行，由exec连接执行下一个程序段。但因为所有的指令和用户数据都被覆盖了，所以程序段必须独立工作——只可以通过参数、环境或文件来传递数据。因此这种应用很少，但是应该了解。更常见的是，在调用命令前只需要做非常少的前期工作时，可以单独使用exec。例如nohup命令，在使用execvp调用用户命令前，它会强制挂起信号。再如nice命令，它改变了命令优先级（见5.15节中的例子）

5.4 实现shell（版本1）

现在对系统调用已经有了足够的了解，可以写自己的shell了，尽管不一定很好。主要的不足是：因为exec没有返回值，所以为了执行一个命令，它不得不自行销毁。如果非要坚持在错误的命令中输入，那么也可以继续运行。这样做时，会执行两个内置命令来修改和访问环境：这两个命令是赋值（例如，BOOK=/usr/marc/book）和set，set会输出环境。[⊖]

第一个任务是把命令行分解成参数。简单地说，就是在没有引用参数的情况下也可工作。另外，因为这个shell不能够控制后台进程、顺序执行或者管道，所以不必考虑特定的字符&、;和|。同样也可以忽略重定向(>、>>和<)。因此，命令行仅由一系列由空格或制表符分开的字组成。必须将它们收集到一起并把它们放入argv数组：

```

#define MAXLINE 200

static bool getargs(int *argcp, char *argv[], int max, bool *eofp)
{

```

⊖ 没有export命令，把整个环境传递给执行的命令。

```

static char cmd[MAXLINE];
char *cmdp;
int i;

*eofp = false;
if (fgets(cmd, sizeof(cmd), stdin) == NULL) {
    if (ferror(stdin))
        EC_FAIL
    *eofp = true;
    return false;
}
if (strchr(cmd, '\n') == NULL) {
    /* eat up rest of line */
    while (true) {
        switch (getchar()) {
            case '\n':
                break;
            case EOF:
                if (ferror(stdin))
                    EC_FAIL
                continue;
        }
        break;
    }
    printf("Line too long -- command ignored\n");
    return false;
}
cmdp = cmd;
for (i = 0; i < max; i++) {
    if ((argv[i] = strtok(cmdp, " \t\n")) == NULL)
        break;
    cmdp = NULL; /* tell strtok to keep going */
}
if (i >= max) {
    printf("Too many args -- command ignored\n");
    return false;
}
*argcp = i;
return true;

EC_CLEANUP_BGN
    EC_FLUSH("getargs")
return false;
EC_CLEANUP_END
}

```

注释:

- 如果getargs能正确解析参数，则返回true，否则返回false。
- 它使用了标准C函数fgets来读取输入行。即使输入行很长，函数也会很安全，因为它不会超过传递的缓冲区。但在那种情况下，并不希望将没读的字符留到以后去读，因为到那时按照本身正确的方式它们可能已成为没有意义的，而且可能还有破坏作用的shell命令。因此在那种情况下（没有换行结束符），输出错误消息前，程序读入并去掉了行中的剩余部分。
- 接着使用strtok把行分解为它的参数。（前一节中，execvp2中也使用了strtok。）
- 在getargs调用的库函数真的出现错误时，它会使用EC_FLUSH显示它本身的错误消

息，否则，例如在行很长的情况下，它会为用户输出消息。

接下来需要函数处理内置命令、赋值和set。这是很容易的，因为可以使用5.2节中的环境处理技术：

```
extern char **environ;

void set(int argc, char *argv[])
{
    int i;

    if (argc != 1)
        printf("Extra args\n");
    else
        for (i = 0; environ[i] != NULL; i++)
            printf("%s\n", environ[i]);
}

void asg(int argc, char *argv[])
{
    char *name, *val;
    if (argc != 1)
        printf("Extra args\n");
    else {
        name = strtok(argv[0], "=");
        val = strtok(NULL, ""); /* get all that's left */
        if (name == NULL || val == NULL)
            printf("Bad command\n");
        else
            ec_neg1( setenv(name, val, true) )
    }
    return;
}

EC_CLEANUP_BGN
    EC_FLUSH("asg")
EC_CLEANUP_END
}
```

接下来，使用main函数来完成该程序，它将输出提示符（这里使用@），获得参数，核对命令是否是内置的。如果不是内置的，则设法去执行它：

```
#define MAXARG 20

int main(void)
{
    char *argv[MAXARG];
    int argc;
    bool eof;

    while (true) {
        printf("@ ");
        if (getargs(&argc, argv, MAXARG, &eof) && argc > 0) {
            if (strchr(argv[0], '=') != NULL)
                asg(argc, argv);
            else if (strcmp(argv[0], "set") == 0)
                set(argc, argv);
            else
                execute(argc, argv);
        }
        if (eof)
            exit(EXIT_SUCCESS);
    }
}
```



```

    }
}

static void execute(int argc, char *argv[])
{
    execvp(argv[0], argv);
    printf("Can't execute\n");
}

```

注意，只要execvp失败，就会回送输出另一个提示符。那就是为什么不可能发现该shell是有用的了。下面是一个简单的会话（为了节省空间缩减了环境列表）：

```

$ sh0
@ set
SSH_CLIENT=192.168.0.1 4971 22
USER=marc
MAIL=/var/mail/marc
EDITOR=vi
@ LASTNAME=RochKind
@ set
SSH_CLIENT=192.168.0.1 4971 22
USER=marc
MAIL=/var/mail/marc
EDITOR=vi
LASTNAME=Rochkind
@ echo Hello World!
Hello World!
$

```

注意在执行echo之后结尾的\$符号，这时shell已经退出。实际上退出的是echo，因为它是代替shell的程序。明确地说，对shell来说，更好的办法是shell在自己本身的进程中执行命令，这正是我们的目标。

5.5 fork系统调用

在某种程度上fork和exec是相反的：它可以创建一个新进程，但它并不初始化新程序中的这个进程。相反，新进程的指令、用户数据和系统数据段几乎就是旧进程的完全复制。

fork —— 创建新进程

```

#include <unistd.h>
pid_t fork(void);
/* Returns child process-ID and 0 on success or -1 on error (sets errno) */

```

fork调用返回后，两个进程（父进程和子进程）都接受返回值。但返回值是不同的，这一点很重要，因为这样才能允许它们接下来做出不同的反应。通常，子进程执行exec，而父进程要么等待子进程终止，要么离开去做其他事情。

创建成功时，子进程接收到fork的返回值为0，父进程将接收子进程的进程ID。通常情况下，返回值为-1则表示出错，但因为fork没有任何参数，所以调用进程不会出错。引起错误的唯一原因便是资源耗尽，如交换空间不足或执行了太多的进程。父进程不会中止，相反可能会等待一会（比如用sleep），然后再试一次，这可能并不是shell的典型做法，典型做法是输出一个消息然后再次提示。

回顾可知，exec系统调用执行的程序可以保留很多的属性，因为在大多数情况下系统数

据段是单独留下来的。同样，fork创建的子进程也继承了父进程的大部分属性，因为它的系统数据段是从父进程复制而来的。正是这种继承性允许用户从shell设定某些属性，如当前目录、有效用户ID和优先级，并把这些属性应用到依次调用的每个命令中。可以认为这些属性属于“直系族”（immediate family），尽管这并不是个正式的UNIX术语。

仅有很少的属性没有得到继承：

- 显然，子进程ID与父进程ID是不同的，因为它们是不同的进程。
- 如果父进程正在运行多线程（见5.17节），那么只有执行fork的那个线程存在于子进程中。尽管如此，父进程中的所有线程都完好无损。
- 子进程得到了父进程多个打开文件描述符的复制。每个都打开到同一个文件，并且文件指针有相同的值。因此，这个打开文件描述（见2.2.3节）以及文件偏移量是共享的。如果子进程用lseek更改它，那么父进程的下一个read或write将从新位置开始。但文件描述符本身是不同的，即使子进程关闭了它，父进程的复制也不会受到影响。
- 子进程的累计执行时间会被重置为零，因为它处在生存期的开始。

（这些是主要内容——全部清单见附件A）

下面给出一个简单示例，说明fork的作用：

```
void forktest(void)
{
    int pid;

    printf("Start of test\n");
    pid = fork();
    printf("Returned %d\n", pid);
}
```

输出如下：

```
$ forktest
Start of test
Returned 98657
Returned 0
$
```

在这种情况下，父进程在子进程之前执行了它的printf，但通常并不能依据执行先后来决定，因为进程的进程表是独立的。如果这个问题很重要，那么你必须亲自同步它们，和9.2.3节一样，可以使用管道来实现，或相对难一点，使用信号。

让我们再一次运行forktest，但这次把标准输出重定向到文件：

```
$ forktest >tmp
$ cat tmp
Start of test
Returned 56807
Start of test
Returned 0
$
```

这里写了两次“Start of test”！你能发现这是为什么吗？（解释在下一段中。）

真正发生的事情是，printf缓存了它的输出（就像2.12节所解释的那样），并且子进程继承了没有被刷新的缓冲区和内容。当子进程退出时，它刷新了缓冲区，而且父进程也作了同样的处理。之前，当输出没有重定向到文件时，printf没有使用缓冲，因为它知道标准输出是个终端设备，并且它应该表现出更好的交互性。在5.7节中，将讨论怎样控制退出的副作用。

`fork`和`exec`是一个完美的搭配，因为`fork`创建子进程，但当子进程几乎是父进程的克隆时，这通常是没有用的。但被新程序覆盖是比较理想的，这恰是`exec`所做的工作。在下一节中，将介绍如何改进我们的shell。

`fork`的潜在开销是巨大的。它会遇到克隆父进程的所有麻烦，也许在重新初始化代码和数据段前，复制大量的数据段（这些指令段通常是只读的，并且是共享的）而仅仅只执行几百条指令。这似乎很浪费，并且确实如此。UNIX的虚拟内存版本很巧妙地解决了这个问题，在该版本中，复制开销是特别高的，被叫作写时复制（copy-on-write）。它工作方式如下：对于`fork`，父进程和子进程共享由页集构成的数据段。只要没有修改页面，这种快捷方式就会正常工作。然而一旦父进程或子进程试图对某一页面进行写操作，那么便复制该页的内容给它自己的每个进程。这种做法是透明的，并且当和具有动态地址转换功能的硬件一起使用时是高效的，而几乎所有的现代计算机都拥有这样的硬件。因为几乎在所有的情况下，`exec`的反应都非常快，所以只有很少的页要复制。但即使所有的页面都需要复制，情况也坏不到哪里去——事实上，情况会更好些，因为我们能够更早地运行子进程。记住写时复制的设计在内核里面；它并不改变`fork`的语义，并且除了具有更良好的性能外，用户几乎意识不到它的存在。

`vfork`是`fork`的另一个版本，其效率比写时复制高，但不像它那么透明：

vfork —— 创建新进程，共享内存（过时的）

```
#include <unistd.h>
pid_t vfork(void);
/* Returns child process-ID and 0 on success or -1 on error (sets errno) */
```

就创建子进程而言，`vfork`和`fork`的行为非常像，但这里的父进程和子进程共享相同的数据段。这里没有写时复制——它们读写相同的变量，因此除非子进程立刻退出或执行一个`exec`，否则灾难便会跟着发生。而通常，子进程的立刻退出或执行`exec`的行为是无论如何都会发生的（请看下一节的函数例子`execute2`）。^①在BSD UNIX早期的版本中，`vfork`是重要的，但是与`fork`相比，它的性能优势不再存在，并且使用它是危险的，所以建议不要使用它，这也是为什么在对照表中将其标上了“过时了”。

提高`fork`和`exec`效率的更新的方法是`posix_spawn`，它是1999年POSIX更新版的一部分。因为它的最终结果是子进程运行一个不同的程序，所以避免了`vfork`的问题。它不提供单个`fork`和`exec`所具有的灵活性，但却能控制通常的大多数情况（如，复制文件描述符），`posix_spawn`的真正目的是：当实时系统交换非常缓慢并且硬件缺乏动态地址转换功能时，为以子进程调用程序提供一个高效的方法。尽管在这类实时系统中，设计者认为`posix_spawn`必须用系统调用来实现，但也可以利用`fork`和`exec`以库函数的方式来实现（练习5.8）。

现在是讨论标准C函数`system`的好时候了：

system —— 运行命令

```
#include <stdlib.h>
int system(
    const char *command    /* command */
);
/* Returns exit status or -1 on error (sets errno) */
```

① 并发读和写相同变量的两个进程与线程相似，具有相同的潜在危险。5.17节还有更多解释。

这个函数通过传递command参数作为一个shell命令行，并跟在shell的一个exec之后执行fork。在程序中执行命令行是一种很方便的方法，但它效率不高（总是要调用shell），也没有跟在exec后单独执行fork或者posix_spawn的灵活性。

5.6 实现shell（版本2）

现在，把fork和execvp放在一起使用，以使前面章节中的shell更有用——在运行一个命令之后不再只是退出，而是对另一个命令进行提示！用函数execute2来取代execute：

```
static void execute2(int argc, char *argv[])
{
    pid_t pid;

    switch (pid = fork()) {
        case -1: /* parent (error) */
            EC_FAIL
        case 0: /* child */
            execvp(argv[0], argv);
            EC_FAIL
        default: /* parent */
            ec_negl( wait(NULL) )
    }
    return;
EC_CLEANUP_BGN
    EC_FLUSH("execute2")
    if (pid == 0)
        _exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

在5.8节将会给出wait的全部细节，而现在呢，仅仅需要知道在返回之前它需要等待子进程终止。

该函数对来自fork和wait的错误与来自execvp的错误的处理方式不同。如果fork或wait的返回值为-1，那么在父进程中只需输出错误信息（用EC_FLUSH）然后返回。调用者（在上一节中的main）会再次提示用户，而这正是我们想要的——一个shell不应该仅仅由于一个错误便终止。但如果execvp返回，我们正处在子进程中，而不是父进程中，那么最好退出。否则，子进程将继续运行并且将会出现两个shell命令提示。如果我们确实试图显示一个命令，两个进程将竞争字符，进而每个进程得到一些字符，因为一个单独的字符只能被读一次。这可能会引起严重的结果。例如，如果要输出rm t*，那么可能会一个shell读取rm *，而另一个shell读取t。

下面将解释调用_exit而不用exit的原因。

5.7 exit系统调用和进程终止

严格地说，有四种让进程终止的方法：

- 1) 调用exit。从main返回一个值和用这个值作为参数调用exit是一样的，而无返回值时和返回0一样。
- 2) 调用_exit或者它的等效调用——_Exit，稍后将会简要解释exit的这两个变体。
- 3) 接收一个终止信号。
- 4) 系统崩溃。从电源故障到操作系统漏洞的任何东西引起的系统崩溃。

本节将介绍前两个方法，在第9章中讲述终止信号，实际上没有讲述造成进程停止的系统崩溃。^①

`exit`的三个变体之间的区别在于：

- `_exit`和`_Exit`的行为完全相同，尽管技术上讲，`_exit`来自UNIX而`_Exit`来自标准C。从现在起将只涉及`_exit`，并且可以假定所说的关于它的任何内容也都能应用于`_Exit`。
- `exit`（没有下划线的）也来自于标准C。它能完成所有`_exit`所能完成的功能，并且也能完成一些更高层次的清理工作，包括由`atexit`（见1.3.4节）注册的不管怎样都要调用的函数，以及刷新标准I/O缓冲区，就像调用了`fflush`或`fclose`。（`_exit`是否刷新缓冲区由实现规定，因此不用考虑它）。

因为`exit`是`_exit`的一个超集，所以这里所说的一切关于后者的东西也适用于前者，所以大多数时候只讨论`_exit`。

通常，在一个还没有执行`exec`的子进程中可以调用`_exit`，代替`exit`，就像前一节中给出的`execute2`那样，这是因为无论对继承的子进程如何清理，通常仅做一次。但真实情况并不是总能如此，所以用户不得不检查每种情况来决定哪个是合适的。在大多数情况下，当某个`exec`覆盖了子进程时，程序会重新开始，所以当它退出时，无论做什么清理都是可以的。任何来自父进程的缓冲区或`atexit`函数都已经消失很久了。

如果读者正在使用“cc”宏指令进行错误检查，像本书这样，那么一定要记住，如果使用`_exit`终止（1.4.2节），那么不会调用由`atexit`注册的函数，所以要在注册的函数中做错误消息的自动显示和函数追溯。尽管只有子进程调用`_exit`，但在前一节的`execute2`中，通过在父进程和子进程中都使用`EC_FLUSH`宏指令解决了这个问题。

下面是`exit`函数的对照表。注意，调用了两个不同的包含文件：

`_exit`——不清理就终止进程

```
#include <unistd.h>

void _exit(
    int status      /* exit status */
);
/* Does not return */
```

`_Exit`——不清理就终止进程

```
#include <stdlib.h>

void _Exit(
    int status      /* exit status */
);
/* Does not return */
```

`exit`——清理并终止进程

```
#include <stdlib.h>

void exit(
    int status      /* exit status */
);
/* Does not return */
```

① 当电源故障时一些计算机向进程发送信号，可以把这种情况作为第三种情况。

_exit和其他两个变体利用等于status最右边（最不重要的）字节的状态代码终止调用它的进程。这样会产生许多额外的副作用；这里列出了一些最主要的，也可以在[SUS2002]中查找完整的清单。通常这些副作用确实会作用在进程终止上（崩溃除外）：

- 关闭所有打开的文件描述符。
- 就像4.3节中所解释的，如果进程是控制进程（会话头），那么这个会话会失去它的控制终端。而且，这个会话中的每个进程会得到一个挂起（SIGHUP）信号，如果进程没有得到或忽略了这个信号，那么也会引起它们终止。
- 以某种方式通知父进程的退出状态，这会在下一节中说明。
- 无论如何子进程都不会受到直接影响，除非它们的新父进程是专门的系统进程，并且如果它们执行了getppid（见5.13节）系统调用，那么它们将得到该系统进程的进程ID（通常是1）。

退出进程的父进程通过某个wait系统调用接收到它的状态代码，这些wait系统调用将会在下一节介绍。状态代码是从0到255的数字。按惯例，状态代码0意味着成功终止，而非0意味着某种非正常终止，这些都是编写退出程序时已经定义好的。定义了两个标准的宏指令，本书的很多例子中都使用了它们：EXIT_SUCCESS被定义为0，而EXIT_FAILURE被定义为某些非0值，通常是1。使用宏代替0和1这样的整数会使程序可读性更好，这便是本书为什么要这么处理的原因。

一旦进程以exit变体或信号终止，那么它会停止执行，但是不会完全消失，直到它的退出状态已经被报告给了它的父进程，除非系统已经确定（以下节介绍的方式）它的父进程对此状态不关心。一个还没有被报告状态的终止进程叫做僵尸（zombie）。^⑥

5.8 wait、waitpid和waitid系统调用

wait、waitpid或者waitid等待子进程改变状态（停止、继续或终止），并且重新得到该进程的状态。前一节中解释了如何终止进程，在4.3节中解释了怎样停止和继续进程。这里首先介绍waitpid，然后介绍其他两个变体。

到目前为止，本节和下一节将描绘的多个特征仅存在于X/Open相容系统，在FreeBSD和Linux中还没有。它们以符号“[X/Open]”来标识。（在1.2节和1.5.1节中讨论了X/Open；目前尽管Linux已宣布是X/Open系统，但当用特征测试宏对它进行测试时，它并没有本节中将要描绘的X/Open功能。）

waitpid —— 等待子进程改变状态

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

pid_t waitpid(
    pid_t pid,          /* process or process-group ID */
    int *statusp,      /* pointer to status or NULL */
    int options         /* options (see below) */
);
/* Returns process ID or 0 on success or -1 on error (sets errno) */
```

可以在以下4种情况中使用pid参数：

>0 等待具有进程ID的子进程。

⑥ 非常适合zombie的字典定义为：一个所谓行尸走肉的、具有最低智能思考的人。（Webster's New Collegiate Dictionary, 6th ed. [Springfield, Mass.: G. & C. Merriam Co., 1960]）。

-1 等待某个子进程。

0 当调用进程时，等待属于相同进程组的某个子进程。

<-1 在进程组中等待某些子进程，它们的进程组ID是-pid。

在4.3节对进程组进行过解释。典型地，在自己所属的进程组中运行管道的shell，在等待管道完成时，会将pid设置为进程组ID的负数，这样管道中的任意进程都会使waitpid返回。

当因为匹配pid参数的子进程改变了状态，而使waitpid返回时，返回的便是子进程ID。下面介绍在WNOHANG选项之下返回值是0的情况。

仅能等待那些直接的（用fork创建的）子进程。孙子进程可能就不能被等待，即使它们的父进程（直接的子进程）已经终止。像前一节中所解释的那样，孤儿进程会被专门的系统进程继承，而不是被祖辈进程继承。

通常，进程能等待其创建的每个子进程，这是很重要的——否则终止了的子进程会像僵尸一样存留在系统之中，直至父进程终止。在某些时候，继承它们的系统进程为了本身的利益会等待和清理它们。当有些进程很长一段时间都不终止时（有时数月），那么长时间保持着僵尸进程真的会妨碍系统表。如果等待太麻烦，那么进程可以使用信号删除僵尸进程，这些将在下一节中介绍。

更改状态的子进程叫作可等待（waitable）子进程，它至多能从waitpid中得到一个返回。换句话说，一个可等待子进程是曾经被等待过的，不可以被等待了。这意味着如果程序的一部分得到了状态并且发现状态不是从所期望的进程中得来的，那么就没有办法将结果填回到系统中去，所以一些其他的waitpid会较晚获得那个进程的状态。（但waitid能够做到——请看下面内容。）

如果执行了waitpid并且某个可等待子进程满足了pid规范，那么waitpid会马上返回。如果还有一个这样的子进程，但它还没有改变状态，那么waitpid会阻塞直到出现一个合适的可等待子进程。如果没有任何子进程匹配pid，则waitpid返回-1并且把errno设置成ECHILD。由于pid完全错了或子进程已经被等待，那么可能会找不到子进程。如果子进程已经被等待过了，那么是不可以不再被等待的（重复一遍，请看waitid，它可以）。

如果statusp是非NULL值，那么它所指向的整数将被设置成子进程的状态。这是_exit或exit（如果这是进程用来终止的方法）的子进程参数和指示进程怎么终止或停止的代码的组合。可以使用宏指令来解释整数：

WIFEXITED(status) 如果子进程正常终止（用_exit或exit[⊖]），则为true。

WEXITSTATUS(status) 如果WIFEXITED为真，则为指向_exit或exit参数的低8位。

WIFSIGNALED(status) 如果子进程（由于信号）非正常终止，则为true。

WTERMSIG(status) 如果WIFSIGNALED为真，则为引起终止的信号编号。

WIFSTOPPED(status) 如果子进程停止，则为true；仅在WUNTRACED选项被设定时可用。（见下面）

WSTOPSIG(status) 如果WIFSTOPPED为真，则为引起子进程停止的信号编号。

WIFCONTINUED(status) 如果进程继续则为真；仅仅在WCONTINUED选项被设定时可用[X/Open]。

WCOREDUMP(status) 如果产生了信息转储文件[在UNIX中叫做“核心转储”（core dump）]，则为真；这种转储有时候对事后分析有用（宏指令是非标准化的，但通常已经实现）。

⊖ 记住，_Exit与_exit是相同的，从main返回与调用exit也是相同的。

最后一个参数是options，它是一个或多个的标志的或操作：

WCONTINUED 除了那些已终止的子进程外，报告那些仍继续的子进程。[X/Open]

WNOHANG 如果不能立即得到状态，就不等待子进程；返回0而不是进程ID。

WUNTRACED 除了那些已终止的子进程外，报告那些已经停止的子进程。[⊖]

下面是一些有关waitpid用法的例子：

```
/* Wait for child pid to terminate and get its status. */
ec_neg1( waitpid(pid, &status, 0) )

/* Wait for any child to terminate, without getting its status. */
ec_neg1( pid = waitpid(-1, NULL, 0) )

/* Report on any child in process group pgid to terminate or stop,
   and get its status. Don't wait if no status is available immediately. */
ec_neg1( pid = waitpid(-pgid, &status, WNOHANG | WUNTRACED) )
```

下面是一个例程，它创建了3个子进程，每一个子进程的终止方式都不同。对于每个终止，都报告了状态。

```
int main(void)
{
    pid_t pid;

    /* Case 1: Explicit call to _exit */
    if (fork() == 0) /* child */
        _exit(123);
    /* parent */
    ec_false( wait_and_display() )

    /* Case 2: Termination by kernel */
    if (fork() == 0) { /* child */
        int a, b = 0;

        a = 1 / b;
        _exit(EXIT_SUCCESS);
    }
    /* parent */
    ec_false( wait_and_display() )

    /* Case 3: External signal */
    if ((pid = fork()) == 0) { /* child */
        sleep(100);
        _exit(EXIT_SUCCESS);
    }
    /* parent */
    ec_neg1( kill(pid, SIGHUP) )
    ec_false( wait_and_display() )

    exit(EXIT_SUCCESS);
EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}

static bool wait_and_display(void)
```

[⊖] 应当叫作WSTOPPED，但是由于历史的原因而称作了WUNTRACED。

```

{
    pid_t wpid;
    int status;

    ec_negl( wpid = waitpid(-1, &status, 0) )
    display_status(wpid, status);
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

void display_status(pid_t pid, int status)
{
    if (pid != 0)
        printf("Process %ld: ", (long)pid);
    if (WIFEXITED(status))
        printf("Exit value %d\n", WEXITSTATUS(status));
    else {
        char *desc;
        char *signame = get_macrostr("signal", WTERMSIG(status), &desc);
        if (desc[0] == '?')
            desc = signame;
        if (signame[0] == '?')
            printf("Signal #%d", WTERMSIG(status));
        else
            printf("%s", desc);
        if (WCOUREDUMP(status))
            printf(" - core dumped");
        if (WIFSTOPPED(status))
            printf(" (stopped)");
#ifdef _XOPEN_UNIX
        else if (WIFCONTINUED(status))
            printf(" (continued)");
#endif
        printf("\n");
    }
}

```

如上所述，因为Linux与X/Open不兼容，所以下列代码行应在Linux上进行单独测试：

```
#if defined(_XOPEN_UNIX) && !defined(LINUX)
```

函数get_macrostr提供了信号编号的可打印版本（连同其他内容一道），并且它的功能和1.4.1节中的errsymbol很相似。如果读者感兴趣，可以到网站[AUP2003]上查找相关代码。

下面是输出：

```

Process 9585: Exit value 123
Process 9586: Erroneous arithmetic operation - core dumped
Process 9587: Hangup

```

如输出所示，第一种情况下，子进程调用_exit终止自己，并传递退出值123。第二种情况下，当子进程试图被0除时，SIGFPE（浮点错误）信号终止了它。第三种情况下，父进程通过发送SIGHUP信号中止了休眠的子进程。（发送信号的kill系统调用将在9.19节介绍。）

本章后面部分，在下一个shell版本中将使用display_status函数。

第二个变量是wait系统调用，它是pid为-1并且没有options的waitpid的简单形式：

wait——等待子进程终止

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

pid_t wait(
    int *statusp,          /* pointer to status or NULL */
);
/* Returns process ID or -1 on error (sets errno) */
```

因为当waitpid的pid设置为-1时，wait和waitpid会等待任一子进程，所以当函数作为较大程序的一部分，要创建一个需要等待的子进程函数时，用它们是不合适的。或许，它可能会意外地获得一些其他子进程的返回值，但由于作为子进程只能等待一次，所以这将会剥夺程序的其他部分等待那个子进程的能力，从而引起潜在的混乱。较好的方法是使用waitpid等待某个指定的子进程或至少一个进程组的成员。正因如此，除了简单程序使用wait以外，其他程序很少使用。在创建子进程的库函数中，决不要使用它。

但假定一个进程有两个子进程，并且不能确定哪个进程会首先终止。如果所要关心的只是等待两个进程都终止，那么可以简单地先等待它们当中的任一个进程先终止，然后当那个waitpid的调用返回时，开始等待另一个进程。或者，父进程能够处理自己的工作，同时周期性地为每个带有WNOHANG选项的子进程调用waitpid。但就像上面所说的，父进程不应该执行阻塞的waitpid调用（pid参数为-1），除非父进程能够保证没有其他的子进程。通常一个大的、复杂的应用程序的各部分来自不同的工作组（例如，图像库，数据库接口），所以不能做那样的保证。如果一个子进程能把进程ID数组传递给某个wait变体，那会是很好的，但事实上没有进程能办到。

第三个变体waitid是伴随SUS1进入UNIX的，到写这本书为止，Linux和FreeBSD中不存在该变体。^①它提供了一个非常重要的新特征：当进程保持等待的时候，用户能够得到进程的状态，所以如果询问了错误进程，也不会有任何害处。另外，与waitpid或wait相比，它提供了更多的信息。如下是对照表：

waitid——等待子进程改变状态 [X/Open]

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

int waitid(
    idtype_t idtype,      /* interpretation of id */
    id_t id,             /* process or process-group ID */
    siginfo_t *infop,    /* returned info */
    int options           /* options */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

siginfo_t——waitid的结构^②

```
typedef struct {          /* only waitid-relevant members shown */
    int si_code;          /* code (see below) */
    pid_t si_pid;        /* child process ID */
    uid_t si_uid;        /* real user ID of child process */
    int si_status;       /* exit value or signal */
} siginfo_t;
```

① 在我的Linux版本中好像只有部分实现了（看起来是实现了，但仅定义了waitpid的选项，而且没有联机资料），读者必须查看自己的版本。

② 实时信号扩展（Realtime Signals Extension）对该结构进行了扩充，见9.5节。

waitid 的第一个参数idtype可为下列之一:

P_PID 等待进程ID为id的子进程 (就像pid>0时的waitpid)。

P_PGID 等待进程组id中的某个子进程 (就像pid<-1时的waitpid)。

P_ALL 等待某个子进程 (像pid= -1时的waitpid); 忽略id。

没有与pid= 0时的waitpid直接相等的idtype, 但当waitid的idtype参数为P_PID, 并且调用者的进程组ID等于id值时, 便等价于这种情况。

waitid的options参数是一个或多个标志的或操作:

WEXITED 报告那些已经退出的进程 (对于没有此标志的waitpid, 情况就一直这样)。

WSTOPPED 报告那些停止了子进程 (与带有WUNTRACED标志的waitpid相似)。

WCONTINUED 报告那些继续执行的子进程 (与waitpid中的标志相同)。

WNOHANG 如果不能立即得到状态, 就不等待子进程, 返回0 (与waitpid中的标志相同)。

WNOEXIT 保留已经报告的进程是可用的, 以便随后的waitid (或其他变体) 调用可以使用它。

通过infop参数, waitid所能提供的信息比waitpid要多。该参数指向的是siginfo_t结构; 在siginfo_t的对照表中对waitid的相关成员的使用进行了介绍。在返回的时候, 子进程ID存在于si_pid成员中。用户能够通过检查si_code来查看子进程终止的原因; 最常出现的代码如下:

CLD_EXITED 使用_exit或exit退出的。

CLD_KILLED 非正常终止 (通过信号)。

CLD_DUMPED 非正常终止并且创建了一个信息转储文件 (UNIX中叫作“核心”文件)。

CLD_STOPPED 已经停止。

CLD_CONTINUED 已经继续。

如果代码为CLD_EXITED, 那么si_status成员会保持被传递给_exit或exit的编号。另外, 如果某个信号引起了状态改变, 那么si_status的值将是这个信号的编号。

下面两个函数分别为wait_and_display和display_status, 它们来自先前的waitpid示例, 这里用waitid将其重新编写:

```
static bool wait_and_display(void)
{
    siginfo_t info;

    ec_negl( waitid(P_ALL, 0, &info, WEXITED) );
    display_status(&info);
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

void display_status(siginfo_t *infop)
{
    printf("Process %ld terminated:\n", (long)infop->si_pid);
    printf("\tcode = %s\n",
        get_macrostr("sigchld-code", infop->si_code, NULL));
    if (infop->si_code == CLD_EXITED)
        printf("\texit value = %d\n", infop->si_status);
    else
```



```

printf("\tsignal = %s\n",
    get_macrostr("signal", info->si_status, NULL));
}

```

输出如下:

```

Process 9580 terminated:
  code = CLD_EXITED
  exit value = 123
Process 9581 terminated:
  code = CLD_DUMPED
  signal = SIGFPE
Process 9582 terminated:
  code = CLD_KILLED
  signal = SIGHUP

```

当使用waitid时,出现了无意中得到错误子进程报告的问题,关于这个问题,可以利用下面的一个算法避免该问题:

- 1) 执行带有P_ALL和WNOEXIT 选项的waitid。
- 2) 如果返回的进程ID不是程序所关心的那一部分,那么返回到第一步。
- 3) 如果进程ID是有用的,那么重新为那个不具有WNOEXIT选项的进程ID激活waitid (或者仅仅使用waitpid),并清除该进程ID,以便使其不再是可等待的。
- 4) 如果正在等待某些不可能被等到的子进程,那么返回到第一步。

使用waitid的唯一问题是,在SUS1以前的系统中它都是不可用的,到写本书为止这些系统包括Linux、FreeBSD和Darwin。所以在waitpid的讨论中必须使用较多早期描述的笨拙方法。

wait、waitpid和waitid的行为进一步受到了父进程如何处理SIGCHLD信号的影响,具体内容在下一节中解释。

5.9 信号、终止和等待

关于信号大部分详细内容将在第9章中讨论,但是此时讨论SIGCHLD也是有意义的。1.1.3节中对信号的简明介绍足以理解这一节内容了;如果不能理解,可以先跳过此节,读完第9章后再来读。

就像前一节中所解释的,父进程使用wait的某个变体能够得到已经改变状态(终止、停止或者继续)的子进程的状态。如果子进程根本就没有被等待,那么它将保持可等待状态(僵尸状态)直到父进程终止。

前面没有涉及到这点,但是一个子进程改变状态时一般会向它的父进程发送SIGCHLD信号。除非父进程已经做了其他的安排,否则默认的行为是忽略该信号,这也是为什么在先前的例子中它并不重要的原因。

如果父进程需要知道子进程状态的变化,那么它可以捕获那个信号。不幸的是,父进程没有被告知是哪个子进程发出的该信号。^①如果它使用某一个wait的变体来获得这个进程ID,那么就会像前一节所解释的那样,有捕获错误进程状态的危险,因为一些其他的进程可能已经在发送信号和调用wait变体的中间改变了状态。安全的方法是使用waitid(像前一节中所解释的)或使用waitpid一个一个地检查所有可能的子进程。

① 除非使用实时信号扩展的高级特征,但是这个扩展通常不可用。

对于SIGCHLD信号，通过调用带有SA_NOCLDWAIT标志的sigaction，父进程能够防止非等待子进程变成僵尸（变成可等待的），而根本不用等待子进程。然后当子进程终止时，其状态便被简单地抛弃，因此父进程得不到它（并且假定它也不关心）。如果父进程执行了wait或waitpid，那么无论参数取什么值（甚至是WNOHANG），它都会阻塞，直到所有的子进程终止，然后将errno设置为ECHILD，并返回-1。如果父进程捕获了SIGCHLD信号，那么它仍会被告知。

如果父进程明确地设置忽略SIGCHLD信号（比如说，带有SIG_IGN标志调用sigaction），与接受默认动作相反，其结果和使用SA_NOCLDWAIT标志[X/Open]是一样的。虽然并不是所有的系统都支持它，但设置SA_NOCLDWAIT标志也是个好方法，这一点得到了广泛支持。

5.10 实现shell（版本3）

使用5.8节介绍的display_status函数的waitpid版本，可以改进5.6节中的execute2：

```
static void execute3(int argc, char *argv[])
{
    pid_t pid;
    int status;

    switch (pid = fork()) {
        case -1: /* parent (error) */
            EC_FAIL
        case 0: /* child */
            execvp(argv[0], argv);
            EC_FAIL
        default: /* parent */
            ec_neg1( waitpid(pid, &status, 0) )
            display_status(pid, status);
    }
    return;

EC_CLEANUP_BGN
    EC_FLUSH("execute3")
    if (pid == 0)
        _exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

下面是部分输出。命令fpe除了试图被0除之外不做任何事情，其仅仅是为此例子而写的：

```
$ sh3
@ date
Tue Feb 25 12:49:52 MST 2003
Process 9954: Exit value 0
@ echo The shell is getting better!
The shell is getting better!
Process 9955: Exit value 0
@ fpe
Process 9956: Erroneous arithmetic operation - core dumped
@ EOT $
```

其中字母EOT表示Ctrl-d的输入位置。

5.11 获得用户ID和组ID

有4个系统调用可以分别用来得到实际用户ID、有效用户ID、实际组ID和有效组ID（在1.1.5节中已经解释了这些概念）：

getuid——得到实际用户ID

```
#include <unistd.h>
uid_t getuid(void);
/* Returns user ID (no error return) */
```

geteuid——得到有效用户ID

```
#include <unistd.h>
uid_t geteuid(void);
/* Returns user ID (no error return) */
```

getgid——得到实际组ID

```
#include <unistd.h>
gid_t getgid(void);
/* Returns group ID (no error return) */
```

getegid——得到有效组ID

```
#include <unistd.h>
gid_t getegid(void);
/* Returns group ID (no error return) */
```

用户ID或组ID仅仅是一些数字。如果需要名称，则必须调用3.5.2节讲述的getpwuid或getgrgid。下面是示例程序，它实现了显示实际用户ID、有效用户ID、实际组ID和有效组ID的功能：

```
int main(void)
{
    uid_t uid;
    gid_t gid;
    struct passwd *pwd;
    struct group *grp;

    uid = getuid();
    ec_null( pwd = getpwuid(uid) )
    printf("Real user = %ld (%s)\n", (long)uid, pwd->pw_name);

    uid = geteuid();
    ec_null( pwd = getpwuid(uid) )
    printf("Effective user = %ld (%s)\n", (long)uid, pwd->pw_name);

    gid = getgid();
    ec_null( grp = getgrgid(gid) )
    printf("Real group = %ld (%s)\n", (long)gid, grp->gr_name);

    gid = getegid();
    ec_null( grp = getgrgid(gid) )
    printf("Effective group = %ld (%s)\n", (long)gid, grp->gr_name);

    exit(EXIT_SUCCESS);
}

EC_CLEANUP_BGN
exit(EXIT_FAILURE);
```

```
EC_CLEANUP_END
}
```

为了使输出有意义，改变了程序文件（uidgrp）的用户所有者和组所有者，而且为用户和组打开了设置ID执行（set-ID-on-execution）位：

```
$ su
Password:
# chown adm:sys uidgrp
# chmod +s uidgrp
$ uidgrp
Real user = 100 (marc)
Effective user = 4 (adm)
Real group = 14 (sysadmin)
Effective group = 3 (sys)
$
```

5.12 设置用户ID和组ID

改变实际用户ID、有效用户ID、实际组ID和有效组ID的规则是很复杂的，而且根据进程是否运行在超级用户上也有所不同。在解释这些规则以前，需要说明一下，除了进程当前的实际用户ID、有效用户ID、实际组ID和有效组ID外，内核还保存了每一个由最近的exec所设置的初始有效用户ID和组ID的记录。这些被叫作保存的设置用户ID和保存的设置组ID。

现在给出设置用户ID的规则，组ID的设置具有相似的规则：

1) 除了使用exec（它能够改变已保存的ID），普通的（非超级用户）进程绝对不能显式地改变实际用户ID或已保存的ID。

2) 普通进程能够将有效ID改变为实际ID或已保存的ID。

3) 超级用户进程能将实际用户ID和有效用户ID改变为任何用户ID的值。

4) 如果超级用户进程改变了实际用户ID，那么已保存的ID也会变为那个值。

关于超级用户的那些规则没什么可讲的（其他用户也可以做到）。关于普通用户的那两条规则实质上意味着：如果exec引起了实际用户ID和有效用户ID（或组ID）的改变，那么进程可以在它们之间来回切换运行。

进程来回往返是有用的，下面是一个场景：设想一个实用函数putfile，它能够在网络上的各个计算机之间传送文件。这个实用函数需要访问仅管理用户（称它为pfadm）能够访问的日志文件。当设置用户ID位打开时，pfadm拥有该程序文件。该程序文件开始运行并且能访问日志文件，因为其有效用户ID是pfadm。然后，为了写实际用户文件，它将有效用户ID设置成了实际用户ID，以便实际用户的权限允许控制访问。当完成了上面的工作后，它将有效用户ID重新设置为pfadm，让其再次访问日志文件。（如果没有保存初始有效用户ID，就不能够将有效用户ID重新设置回原来的值。）注意，就这个实用函数而言，似乎pfadm是专用的，但实际上对内核来说它只是一个普通用户——并不是超级用户。

现在讨论那些系统调用本身。仅对普通用户进程有用的是seteuid和setegid。

seteuid——设置有效用户ID

```
#include <unistd.h>

int seteuid(
    uid_t uid           /* effective user ID */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

setegid——设置有效组ID

```
#include <unistd.h>

int setegid(
    gid_t gid          /* effective group ID */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

根据上面列出的规则，对于普通用户进程，参数必须为实际用户（或组）ID或已保存的ID。对于超级用户，参数可以是任意的用户（或组）ID。

超级用户可以使用两个额外的调用：

setuid——设置有效、实际和已保存的用户ID

```
#include <unistd.h>

int setuid(
    uid_t uid          /* real and saved user ID */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

setgid——设置有效、实际和已保存的组ID

```
#include <unistd.h>

int setgid(
    gid_t gid          /* real and saved group ID */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

对于超级用户，参数可以是任意用户（或组）ID，它设置了实际的值和已保存的值。对于普通用户，这两个调用和刚才介绍的“c”版本中所描述的一样，是非常容易让人混淆的，所以应该尽量避免使用它们。

另外两个增加的调用setreuid和setregid，与已经介绍的4个调用功能相重叠，又增加了更多混淆，所以也应该避免使用它们。（在引入setreuid和setegid之前的旧系统中，它们是非常重要的。）

5.13 获得进程ID

使用这些调用，进程可以得到其本身的进程ID和其父进程的进程ID：

getpid——得到进程ID

```
#include <unistd.h>

pid_t getpid(void);
/* Returns process ID (no error return) */
```

getppid——得到父进程ID

```
#include <unistd.h>

pid_t getppid(void);
/* Returns parent process ID (no error return) */
```

在2.4.3节的例子中已经使用了getpid。

5.14 chroot系统调用

chroot —— 改变根目录

```
#include <unistd.h>

int chroot(
    const char *path          /* path name */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

这是一个为超级用户进程预留的系统调用，它能够改变进程的根目录，即路径为/的目录。它不是标准的调用，但实质上，在所有类UNIX的系统中都已实现了它（因为它非常地老）。

此时想起chroot的两个用途：

- 当带有内置路径名的命令（诸如/etc/passwd）不在普通文件上运行时。在任何方便的地方都能够建立新树，改变根目录，然后执行此命令。在安装新命令前，先对它们进行测试是非常有用的。
- 为了增强安全性，如当web服务器进程开始处理HTML文件时。它可以把根目录设置为基本路径，这可以保证进程无法访问指向其他文件的路径，甚至路径“..”也不可以，因为在根目录下，它只被解释成是正确返回根目录的引用。

一旦根目录改变了，便没有办法用chroot返回到根目录了。尽管如此，有些系统提供了一个姐妹函数fchroot，它以文件描述符为参数返回根目录。（它们的关系如3.6.2节中解释的fchdir和chdir的关系相似。）用户可以通过读取打开文件描述符来获得向当前根目录打开的文件描述符，接着执行chroot，然后把保存了的文件描述符作为fchroot的参数就可以返回到根目录了。

5.15 获得并设置优先级

如1.1.6节中所提到的，每个进程都有一个nice值，它是进程影响内核调度优先级的方法。较高的nice值意味着进程想运行在一个较低的优先级上。较低的nice值意味着进程想运行在较高的优先级上。为了使nice值为正数，它们取某个数字的偏移量，在不同的系统中，这个数字有所不同（奇怪的是，在UNIX文档中，它的参考值为NZERO），一个典型值是20。进程的nice值以20开始，最大可以到39，最小可以为0。

为了改变nice的值，进程需要执行nice系统调用：

nice —— 改变nice值

```
#include <unistd.h>

int nice(
    int incr          /* increment */
);
/* Returns old nice value - NZERO or -1 on error (sets errno) */
```

nice系统调用把incr添加到nice值上。nice值的结果必须在0到39之间，其中包含0和39；如果产生的值无效，则使用最近的有效值。只有超级用户可以降低nice值，获得比平均水平要好的服务。

实际上，nice系统调用返回的新nice值减去了20，因此如果NZERO是20，则返回结果范围在-20到19之间。然而，返回值的用途很少。其中原因也包括19的新nice值和错误返回码无

法区分 ($19-20 = -1$)。这个漏洞已经存在多年仍没有修正,不是没被注意到,而是因为大多数的UNIX系统程序员很少关心像nice这样较小的系统调用的错误返回值。[⊖]

大多数UNIX用户都熟悉nice命令,它能以一个较低的优先级(或如果是超级用户时,以较高的优先级)来运行程序。下面是我们的一种版本:

```
#define USAGE "usage: auptnice [-num] command\n"

int main(int argc, char *argv[])
{
    int incr, cmdarg;
    char *cmdname, *cmdpath;

    if (argc < 2) {
        fprintf(stderr, USAGE);
        exit(EXIT_FAILURE);
    }
    if (argv[1][0] == '-') {
        incr = atoi(&argv[1][1]);
        cmdarg = 2;
    }
    else {
        incr = 10;
        cmdarg = 1;
    }
    if (cmdarg >= argc) {
        fprintf(stderr, USAGE);
        exit(EXIT_FAILURE);
    }
    (void)nice(incr);
    cmdname = strchr(argv[cmdarg], '/');
    if (cmdname == NULL)
        cmdname = argv[cmdarg];
    else
        cmdname++;
    cmdpath = argv[cmdarg];
    argv[cmdarg] = cmdname;
    execvp(cmdpath, &argv[cmdarg]);
    EC_FAIL

    EC_CLEANUP_BGN
        exit(EXIT_FAILURE);
    EC_CLEANUP_END
}
```

注意调用execvp时,是怎样重新使用传入的argv参数的。变量cmdarg保存了命令路径的下标(1或2,取决于规定的增量)。以cmdarg开始的argv部分是要传递的部分。用户可能会将这个操作和5.3节中execvp2所需要的更详尽的处理操作进行比较。那里必须插入一个参数,这要求重新复制整个数组。

假定向量中的第一个字符串仅仅是命令名称,并不是整个路径,那么必须删除除了路径最后的部分外的其他所有内容,以防用带有斜线的路径名执行命令。但execvp的第一个参数必须是在auptnice命令行中所输入的内容。

此外注意,这里使用了没有fork的execvp。因为已经改变了优先级,所以没有必要保

[⊖] 该句来自于1985版本,20年已经过去了,仍没有更正。同时,由于采用了线程、较好的信号以及至少上千个其他有用的特征,所以这个缺陷多少也算得到补偿了。

持一个只是空等的父进程。

下面是Solaris上运行aupnice的结果。为了说明命令，使用ps报告其本身。注意，当nice值的增加步长为10（aupnice的默认值）时，内部优先级（不管优先级表示什么）从58变到了28。

```
$ ps -ac
  PID CLS PRI TTY      TIME CMD
 10460 TS  58 pts/2    0:00 ps
$ aupnice ps -ac
  PID CLS PRI TTY      TIME CMD
 10461 TS  28 pts/2    0:00 ps
```

有一些更新而且更奇特的调用，getpriority和setpriority，可以用它们来处理nice值，相关细节可查看[SUS2002]或系统文档。

5.16 进程限制

内核给进程资源增加了多种限制，如最大的文件大小和最大的堆栈大小。通常，当达到某个限制时，无论它是由哪个函数（如write、malloc）引起的，都会返回错误，或生成一个信号，如栈溢出时是SIGSEGV信号（第9章）。[⊖]下面列出了7个标准资源，可以设置或获得它们的限制，并且实现也可定义其他的限制。

对于每种资源，都有最大（或硬）限制和当前（或软）限制。当前限制是有效的，并且允许普通进程将当前限制设置为任何不超过最大值的合理数值。无论当前限制是多少，普通进程都能够不可逆地将最大值降低（但不能升高）到当前限制值。超级用户进程可以将两种限制之一设置成内核能支持的任意值。

下面是得到和设置这些限制的系统调用：

getrlimit —— 得到资源限制

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

int getrlimit(
    int resource,          /* resource */
    struct rlimit *rlp     /* returned limits */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

setrlimit —— 设置资源限制

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

int setrlimit(
    int resource,          /* resource */
    const struct rlimit *rlp /* limits to set */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

struct rlimit —— getrlimit和setrlimit的结构

```
struct rlimit {
    rlim_t rlim_cur;      /* current (soft) limit */
    rlim_t rlim_max;     /* maximum (hard) limit */
};
```

⊖ 如果捕获了SIGSEGV，当没有堆栈时，信号句柄如何执行？如9.3节解释的那样，可以创建其他的堆栈。如果没有那样做，信号会被设置回其默认行为，即终止该进程。

每个调用对应一个单独的资源。用户可以用`getrlimit`得到它的限制，并且根据上述规则用`setrlimit`设置它们。标准资源如下：

`RLIMIT_CORE`内核（信息转储）文件的最大字节数；0意味着根本没有写任何文件。如果超过了限制，不会报告任何错误——写文件到最大字节数后就停止，这可能会产生无用的文件。

`RLIMIT_CPU`以秒为单位的最大CPU时间。超过这个时间会产生一个`SIGXCPU`信号，默认情况下会终止该进程。如果用户捕获、忽略或阻塞该信号，标准并没有定义会发生什么，结果是进程可能会终止，因为在没有累计更多的CPU时间的情况下，进程无法继续。

`RLIMIT_DATA`数据段的最大字节数。超过此值会导致内存分配函数（如`malloc`）失败。

`RLIMIT_FSIZE`最大的文件字节数。超过此值会生成`SIGXFSZ`信号，如果该信号可以被捕获、忽略或阻塞，那么违例函数将失败。

`RLIMIT_NOFILE`进程可以使用的文件描述符的最大编号数。^①超过此值会导致违例函数失败。

`RLIMIT_STACK`栈的最大字节数。超过此值会生成`SIGSEGV`信号。

`RLIMIT_AS`内存的最大值，它包括数据段、堆栈和用`mmap`映射进来的内容（见7.14节）。超过此值会导致违例函数失败，或者如果是堆栈，后果由`RLIMIT_STACK`来描述。

进程限制是通过`exec`保留的，而且在`fork`之后由于进程继承，所以只要在登录时设置一次，它们就会影响所有由这个登录shell衍生的进程。但是，如`cd`一样，设置它们的命令必须被内置到shell中——在shell的子进程中执行它是无效的。大多数shell都有`ulimit`命令，可以对文件大小设置限制，并且其他shell可能有更通用的命令，可能叫`limit`、`limits`或`plimit`。^②

对于`rlimit`结构的`rlim_cur`和`rlim_max`成员，除了实际的数，还有一些专门的宏指令：`RLIM_SAVED_CUR`、`RLIM_SAVED_MAX`和`RLIM_INFINITY`。

当调用`getrlimit`时，如果它们配置在某个`rlim_t`中，那么将返回实际的数，其中`rlim_t`是个无符号类型，标准中没有指定无符号类型的宽度。如果某个数不合适，并当其等于最大值时，会将成员设置为`RLIM_SAVED_MAX`。否则会将它设置为`RLIM_SAVED_CUR`，这便意味着“设置限制为当前的限制。”如果成员被设置成`RLIM_INFINITY`，意味着没有限制。

当调用`setrlimit`时，如果某个成员是`RLIM_SAVED_CUR`，那么它会被设置成当前限制；如果是`RLIM_SAVED_MAX`，那么它会被设置为最大值；如果是`RLIM_INFINITY`，将不会强加任何限制。否则，限制将会被设置为曾经使用过的数。然而，只有在调用`getrlimit`返回限制值时，才可以使用`RLIM_SAVED_CUR`或`RLIM_SAVED_MAX`；否则，用户就不得使用当前的数。换句话说：因为`rlim_t`没有保持那个数，而用户又被迫使用这些宏时，才会使用它们。

对`RLIM_SAVED_CUR`和`RLIM_SAVED_MAX`来说，这些冗长无聊的废话实际上就是：即使不能得到它们的实际值，也可以一定程度地操作这些限制。

如果实现上从来不会有合适`rlim_t`的限制，那么就决不会使用`RLIM_SAVED_CUR`宏或`RLIM_SAVED_MAX`宏，所以允许这些宏有和`RLIM_INFINITY`相同的值。在下面这个例子中，将会看到这个影响，该例子显示了限制，它把文件限制设置成了不可能的小数字（没有文件会小于这个数），然后超过它，最后导致程序终止，该终止是通过`SIGXFSZ`信号来完成的。

```
int main(void)
{
    struct rlimit r;
```

① 技术上讲，其比最大文件描述符大。所有可能的文件描述符都是向着这个限制计数的，即使它们没有被打开。

② 不要把这些进程限制和用户限制混淆，如用户的磁盘限额。可以通过非标准命令（如`quota`）来查询和设置用户限制，并依据非标准系统调用（如`quotaact1`或者`ioct1`）来实现用户限制。

```

int fd;
char buf[500] = { 0 };

if (sizeof(rlim_t) > sizeof(long long))
    printf("Warning: rlim_t > long long; results may be wrong\n");
ec_false( showlimit(RLIMIT_CORE, "RLIMIT_CORE") )
ec_false( showlimit(RLIMIT_CPU, "RLIMIT_CPU") )
ec_false( showlimit(RLIMIT_DATA, "RLIMIT_DATA") )
ec_false( showlimit(RLIMIT_FSIZE, "RLIMIT_FSIZE") )
ec_false( showlimit(RLIMIT_NOFILE, "RLIMIT_NOFILE") )
ec_false( showlimit(RLIMIT_STACK, "RLIMIT_STACK") )
#ifdef FREEBSD
ec_false( showlimit(RLIMIT_AS, "RLIMIT_AS") )
#endif
ec_negl( getrlimit(RLIMIT_FSIZE, &r) )
r.rlim_cur = 500;
ec_negl( setrlimit(RLIMIT_FSIZE, &r) )
ec_negl( fd = open("tmp", O_WRONLY | O_CREAT | O_TRUNC, PERM_FILE) )
ec_negl( write(fd, buf, sizeof(buf)) )
ec_negl( write(fd, buf, sizeof(buf)) )
printf("Wrote two buffers! (?)\n");
exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
exit(EXIT_FAILURE);

EC_CLEANUP_END
}

static bool showlimit(int resource, const char *name)
{
    struct rlimit r;

    ec_negl( getrlimit(resource, &r) )
    printf("%s: ", name);
    printf("rlim_cur = ");
    showvalue(r.rlim_cur);
    printf("; rlim_max = ");
    showvalue(r.rlim_max);
    printf("\n");
    return true;
}

EC_CLEANUP_BGN
return false;
EC_CLEANUP_END
}

static void showvalue(rlim_t lim)
{
    /*
     * All macros may equal RLIM_INFINITY; that test
     * must be first; can't use switch statement.
     */
    if (lim == RLIM_INFINITY)
        printf("RLIM_INFINITY");
#ifdef BSD_DERIVED
    else if (lim == RLIM_SAVED_CUR)
        printf("RLIM_SAVED_CUR");
    else if (lim == RLIM_SAVED_MAX)
        printf("RLIM_SAVED_MAX");
#endif
}
#endif

```



```

else
    printf("%llu", (unsigned long long)lim);
}

```

FreeBSD和Darwin并没有实现全部内容，如在代码中看到的。^①

下面是在4个测试系统中所得到的，从Solaris开始：

```

RLIMIT_CORE: rlim_cur = RLIM_INFINITY; rlim_max = RLIM_INFINITY
RLIMIT_CPU: rlim_cur = RLIM_INFINITY; rlim_max = RLIM_INFINITY
RLIMIT_DATA: rlim_cur = RLIM_INFINITY; rlim_max = RLIM_INFINITY
RLIMIT_FSIZE: rlim_cur = RLIM_INFINITY; rlim_max = RLIM_INFINITY
RLIMIT_NOFILE: rlim_cur = 256; rlim_max = 1024
RLIMIT_STACK: rlim_cur = 8683520; rlim_max = 133464064
RLIMIT_AS: rlim_cur = RLIM_INFINITY; rlim_max = RLIM_INFINITY
File Size Limit Exceeded - core dumped

```

Linux:

```

RLIMIT_CORE: rlim_cur = 0; rlim_max = RLIM_INFINITY
RLIMIT_CPU: rlim_cur = RLIM_INFINITY; rlim_max = RLIM_INFINITY
RLIMIT_DATA: rlim_cur = RLIM_INFINITY; rlim_max = RLIM_INFINITY
RLIMIT_FSIZE: rlim_cur = RLIM_INFINITY; rlim_max = RLIM_INFINITY
RLIMIT_NOFILE: rlim_cur = 1024; rlim_max = 1024
RLIMIT_STACK: rlim_cur = RLIM_INFINITY; rlim_max = RLIM_INFINITY
RLIMIT_AS: rlim_cur = RLIM_INFINITY; rlim_max = RLIM_INFINITY
File size limit exceeded

```

FreeBSD:

```

RLIMIT_CORE: rlim_cur = RLIM_INFINITY; rlim_max = RLIM_INFINITY
RLIMIT_CPU: rlim_cur = RLIM_INFINITY; rlim_max = RLIM_INFINITY
RLIMIT_DATA: rlim_cur = 536870912; rlim_max = 536870912
RLIMIT_FSIZE: rlim_cur = RLIM_INFINITY; rlim_max = RLIM_INFINITY
RLIMIT_NOFILE: rlim_cur = 957; rlim_max = 957
RLIMIT_STACK: rlim_cur = 67108864; rlim_max = 67108864
Filesize limit exceeded

```

Darwin:

```

RLIMIT_CORE: rlim_cur = 0; rlim_max = RLIM_INFINITY
RLIMIT_CPU: rlim_cur = RLIM_INFINITY; rlim_max = RLIM_INFINITY
RLIMIT_DATA: rlim_cur = 6291456; rlim_max = RLIM_INFINITY
RLIMIT_FSIZE: rlim_cur = RLIM_INFINITY; rlim_max = RLIM_INFINITY
RLIMIT_NOFILE: rlim_cur = 256; rlim_max = RLIM_INFINITY
RLIMIT_STACK: rlim_cur = 524288; rlim_max = 67108864
Filesize limit exceeded

```

还有一个更早的系统调用，其功能相对较少，而且返回值有问题：

ulimit —— 得到和设置进程限制

```

#include <ulimit.h>

long ulimit(
    int cmd,                /* command */
    ...                    /* optional argument */
);
/* Returns limit or -1 with errno changed on error (sets errno) */

```

① 实际上，这些调用最初来自4.2BSD，但起源于BSD的系统并没有向系统中添加全部的、较新的POSIX扩展特征，即只添加了部分。

cmd参数的值可取下列之一:

UL_GETFSIZE得到当前的文件大小限制, 等价于带有RLIMIT_FSIZE参数的getrlimit。

UL_SETFSIZE设置文件大小限制为第二个long参数, 只有超级用户进程才能增加它。等价于带有RLIMIT_FSIZE参数的setrlimit, 其中需要调整rlim_max和rlim_cur来匹配。

设置或获得的这些数大小以512字节为单元。因为这些数可以变大, 所以返回值可能是负的, 甚至也可能是-1。因此, 检查错误的方法是在调用开始之前, 设置errno为0。如果它返回-1, 那么只有当errno也变化时, 才说明是出现了错误。

有更多关于ulimit的特性, 但这里没有深入研究它们。ulimit本质上是混乱的, 所以最好使用getrlimit和setrlimit来代替它。

getrusage系统调用可以获得有关进程、进程终止和等待着的子进程资源使用的信息:

getrusage —— 得到资源的使用

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

int getrusage(
    int who,                /* RUSAGE_SELF or RUSAGE_CHILDREN */
    struct rusage *r_usage /* returned usage information */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

struct rusage —— getrusage的结构

```
struct rusage {
    struct timeval ru_utime; /* user time used */
    struct timeval ru_stime; /* system time used */
    /* following members are nonstandard */
    long ru_maxrss;         /* maximum resident set size */
    long ru_ixrss;          /* integral shared memory size */
    long ru_idrss;          /* integral unshared data size */
    long ru_isrss;          /* integral unshared stack size */
    long ru_minflt;         /* page reclaims */
    long ru_majflt;         /* page faults */
    long ru_nswap;          /* swaps */
    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 */
};
```

[SUS2002]只规定了前两个成员。包括FreeBSD在内, 较新一点的BSD系统都支持它们。Solaris 8 (至少在Intel系列上) 只支持标准成员; Linux除了支持标准成员, 还支持ru_minflt、ru_majflt和ru_nswap。若要查看成员所代表的含义的细节, 可以查看用户系统手册getrusage的联机资料; 至于所有成员细节, 可以通过在基于BSD的系统上运行来获得。

两个返回的时间与times返回的相似(见1.7.2节), 但精确度更高, 因为timeval(见1.7.1节)是以微秒来计量的。

5.17 线程介绍

这节将对线程进行非常简洁的介绍, 但足以理解一些令人感兴趣的示例程序了。目的是解释什么是线程, 帮助判断应用程序设计是否可从线程中获益, 并且提醒在使用时可能带来

的一些危险并给出警告。

至于线程的整个原理（大约有100多个操作线程的系统调用来管理线程），可以查看有关这个主题的书籍，如[Nor1997]或[But1997]。

5.17.1 线程创建

到目前为止所有例子中，所创建的进程（用fork）内部都只有一个控制流，或线程。程序的执行以一个单一的顺序从指令到指令不断进行，同时，各种各样的指令也改变着栈、全局数据和系统资源，其中某些指令还可能也执行了系统调用。

根据UNIX的POSIX线程特性，一个进程可以有多个线程，每一个线程都有它自身的控制流（它自己的指令计数器和CPU时钟）和它自己的栈。实质上，包含诸如打开文件或当前目录的全局数据和资源在内的其他任何与进程相关的东西，都是共享的。^①下面的代码说明了这个意思。

```
static long x = 0;

static void *thread_func(void *arg)
{
    while (true) {
        printf("Thread 2 says %ld\n", ++x);
        sleep(1);
    }
}

int main(void)
{
    pthread_t tid;

    ec_rv( pthread_create(&tid, NULL, thread_func, NULL) )
    while (x < 10) {
        printf("Thread 1 says %ld\n", ++x);
        sleep(2);
    }
    return EXIT_SUCCESS;

    EC_CLEANUP_BGN
        return EXIT_FAILURE;
    EC_CLEANUP_END
}

```

线程的输出顺序是不可预知的，这是碰巧得到的输出：

```
Thread 2 says 1
Thread 1 says 2
Thread 2 says 3
Thread 1 says 4
Thread 2 says 5
Thread 2 says 6
Thread 1 says 7
Thread 2 says 8
Thread 2 says 9
Thread 1 says 10
Thread 2 says 11
Thread 2 says 12

```

因此，读者将看到，程序中的两个线程——main中的线程和thread_func中的线程，

① 线程也可安排获得某个特定线程的数据；细节可见[SUS2002]或者其他此类参考书。

都访问了同样的全局变量x。这样做存在一些问题，本书将在5.17.3节讨论这些问题。

初始化线程是一个包含main的线程，它用pthread_create系统调用删除了第二个线程。

pthread_create —— 建立线程

```
#include <pthread.h>

int pthread_create(
    pthread_t *thread_id, /* new thread's ID */
    const pthread_attr_t *attr, /* attributes (or NULL) */
    void *(*start_fcn)(void *), /* starting function */
    void *arg /* arg to starting function */
);
/* Returns 0 on success, error number on error */
```

新线程从pthread_create中规定的启动函数的调用开始，该函数必须具有如下原型：

pthread starting function

```
void *start_fcn(
    void *arg
);
/* Returns exit status */
```

无论向pthread_create函数的第4个参数传递什么值，都将被直接传给那个启动函数。通常它是一个void指针，而实际上经常是指向某个数据的指针。因为线程共享同样的地址空间，所以该指针对两个进程都是有效的。如果愿意，也可以以整型数据类型传递参数，但必须将它强制转换给某个void指针。并且为了安全，应该检查（比如说，使用assert）使用的整型类型是否和void指针相适应。如下面的示例一样，使用一行代码来检查：

```
assert(sizeof(long) <= sizeof(void *));
```

影响新线程属性的不仅仅是某些标志，而是pthread_attr_t类型的对象，而且必须调用pthread_attr_setscope和pthread_attr_setstacksize设置这个对象，在这里就不深入讨论这些调用了。（但用户或许会开始想为什么需要100多个与线程相关的调用了。）在后面例子中中将只使用默认属性，所以pthread_create的第二个参数将会是NULL。

所有“pthread”函数一律在成功时返回0，而失败时返回错误代码，但不设置errno。对于这个问题，有专门的“ec”宏——ec_rv来解决。如果有错误，它将获得错误代码，并且仿佛它真是那个errno值一样处理它。[⊖]

5.17.2 等待线程终止

一个线程可以等待另一个线程终止，并用pthread_join（与wait以及其变体类似）来获得其退出状态。

pthread_join —— 等待线程终止

```
#include <pthread.h>

int pthread_join(
    pthread_t thread_id, /* ID of thread to join */
    void **status_ptr /* returned exit status (if not NULL arg) */
);
/* Returns 0 on success, error number on error */
```

⊖ 错误检测包中的各种函数都被修改成了线程安全的，但是本书没有给出相应的代码，然而可以在网站上看到。

下面是为了实现如下功能而修改的小示例。其功能是：线程1可以传递一个限制给线程2，然后线程2向线程1报告x的值：

```
static long x = 0;

static void *thread_func(void *arg)
{
    while (x < (long)arg) {
        printf("Thread 2 says %ld\n", ++x);
        sleep(1);
    }
    return (void *)x;
}

int main(void)
{
    pthread_t tid;
    void *status;
    assert(sizeof(long) <= sizeof(void *));
    ec_rv( pthread_create(&tid, NULL, thread_func, (void *)6) )
    while (x < 10) {
        printf("Thread 1 says %ld\n", ++x);
        sleep(2);
    }
    ec_rv( pthread_join(tid, &status) )
    printf("Thread 2's exit status is %ld\n", (long)status);
    return EXIT_SUCCESS;
}

EC_CLEANUP_BGN
    return EXIT_FAILURE;
EC_CLEANUP_END
}
```

输出如下：

```
Thread 1 says 1
Thread 2 says 2
Thread 2 says 3
Thread 1 says 4
Thread 2 says 5
Thread 2 says 6
Thread 1 says 7
Thread 1 says 8
Thread 1 says 9
Thread 1 says 10
Thread 2's exit status is 7
```

5.17.3 线程同步（互斥）

在接下来的几章中将看到，在UNIX系统上，多进程同时工作（或许跨越网络）的难题是允许它们共享数据。而用线程时，则完全相反——难题是保证被分散的数据的正常共享。事实上，刚才给出的两个线程的例子是有缺陷的，因为两个线程可能会同时访问相同的数据——变量x。尽管似乎一个简单的增量操作符是一个原子操作，但并没有保证它确实是。事实上，很可能线程1更新了32位的x的一半，而同时线程2却读取了整个32位，从而导致线程2得到的不是有效的整型，[⊖]而是一个混合物。对于更复杂的共享数据结构，即更加实际的情形，这个问题

[⊖] 这只是一中可能会出错的情况。另一种情况是编译器优化可能会把整数留在寄存器中。在没有保护的情况下决不能让线程同时访问数据。

题就更糟了。我们想要访问原子态的共享数据。具有此含义的两点说明如下：

- 如果数据结构的更新操作使数据结构处在一个暂时的不连续的状态下，那么除了更新数据的线程外没有其他线程能够看到这种状态下的数据结构。
- 如果线程必须读取数据，计算结果以及将它们写回等操作，直到整个操作顺序完成之前，不允许有其他线程修改它们。否则其他线程的修改会丢失。

所提出的这些要求可以通过一些简单的系统调用来满足，这些系统调用可以实现互斥-独占对象，简称为互斥（mutex）。线程使用它们保护临界段，否则另一个线程可能会在临界段上看到不一致的数据或干扰对临界段的修改。

主要的互斥系统调用是pthread_mutex_lock和pthread_mutex_unlock：

pthread_mutex_lock —— 锁互斥

```
#include <pthread.h>

int pthread_mutex_lock(
    pthread_mutex_t *mutex /* mutex to lock */
);
/* Returns 0 on success, error number on error */
```

pthread_mutex_unlock —— 解锁互斥

```
#include <pthread.h>

int pthread_mutex_unlock(
    pthread_mutex_t *mutex /* mutex to unlock */
);
/* Returns 0 on success, error number on error */
```

互斥工作原理如下：如果某个互斥已经被锁定，那么pthread_mutex_lock将阻塞，直到互斥被解除锁定。

获得一个合适的互斥变量的最简单方法是利用一个初始化语句来声明它，像这样：

```
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
```

它可以是文件的局部变量（static），也可以是文件之间的共享变量（extern）。用户既可以在栈中设置互斥（自动变量），也可以动态分配，但之后需要调用pthread_mutex_init初始化它们，关于pthread_mutex_init，这里不再详细介绍。[⊖]

现在来修改程序以便充分保护共享数据：

```
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
static long x = 0;

static void *thread_func(void *arg)
{
    bool done;

    while (true) {
        ec_rv( pthread_mutex_lock(&mtx) )
        done = x >= (long)arg;
        ec_rv( pthread_mutex_unlock(&mtx) )
```

⊖ 即使编译器允许，也不应当使用初始程序PTHREAD_MUTEX_INITIALIZER初始自动互斥变量。因为在一些系统上，那样可能会调用一个线程不安全的函数。在C++中，如果使用PTHREAD_MUTEX_INITIALIZER初始化一个静态的内部互斥，也会出现相同的麻烦，因为直到调用了函数，C++才开始初始化。

```
        if (done)
            break;
        ec_rv( pthread_mutex_lock(&mtx) )
        printf("Thread 2 says %ld\n", ++x);
        ec_rv( pthread_mutex_unlock(&mtx) )
        sleep(1);
    }
    return (void *)x;

EC_CLEANUP_BGN
    EC_FLUSH("thread_func")
    return NULL;
EC_CLEANUP_END
}

int main(void)
{
    pthread_t tid;
    void *status;
    bool done;

    assert(sizeof(long) <= sizeof(void *));
    ec_rv( pthread_create(&tid, NULL, thread_func, (void *)6) )
    while (true) {
        ec_rv( pthread_mutex_lock(&mtx) )
        done = x >= 10;
        ec_rv( pthread_mutex_unlock(&mtx) )
        if (done)
            break;
        ec_rv( pthread_mutex_lock(&mtx) )
        printf("Thread 1 says %ld\n", ++x);
        ec_rv( pthread_mutex_unlock(&mtx) )
        sleep(2);
    }
    ec_rv( pthread_join(tid, &status) )
    printf("Thread 2's exit status is %ld\n", (long)status);
    return EXIT_SUCCESS;

EC_CLEANUP_BGN
    return EXIT_FAILURE;
EC_CLEANUP_END
}
```

注意，必须将条件放在while表达式外，以便能够围绕这些条件利用加锁和解锁调用。当然，并不需要保护整个循环，因为那样将破坏并发性。需要仔细地保护，足够安全就行，不需要过多的保护，否则会损害性能。不幸的是，即使用户可能能够用正确的测试方法检测性能问题，但测试是否已经禁止了所有的竞争条件也是很困难的。

这里还有一个缺点：假定在thread_func中调用pthread_mutex_unlock失败，那么将会导致线程退出。这可能会使互斥被锁定，将导致初始线程在对pthread_mutex_lock进行的某一个调用中永远地阻塞。一种可能的解决方法是确保互斥在thread_func的清理代码中调用另一个pthread_mutex_unlock来解除锁定，像这样：

```
EC_CLEANUP_BGN
    (void)pthread_mutex_unlock(&mtx);
    EC_FLUSH("thread_func")
    return NULL;
EC_CLEANUP_END
```

假设第一次pthread_mutex_unlock调用失败，第二次调用可能会成功，这看起来是一段延迟。因为只要互斥有一个有效值，决没有pthread_mutex_lock或pthread_mutex_unlock调用失败的情形。最简单、最安全、最明确的方法是在调试时检查来自这些函数的错误返回值，而不是在产品程序中。或者，对自己的应用程序来说，只要能够得到错误报告，你最好是不理睬它们，这样就不必浪费时间试图去查找应用程序延迟原因。这是另一个为什么要谨慎小心正确使用线程的情况。你可能会发现花费的时间中，有5%的时间用于实现线程，而95%的时间用在确保程序的正确性上。

正确的处理方法是，用函数集（或者用一个类，如果用户使用像C++那样的面向对象语言的话）封装对共享数据的访问，且在这些封闭的访问函数中使用互斥的方法。在整个程序中，不要一厢情愿地扩展加锁和解锁调用，像本书所举的例程那样。

因此再次重新编写前面的程序，这次所有对x的访问都只通过函数get_and_incr_x来完成，该函数处理了所有的加锁和解锁工作。x和互斥都被移到了函数内部。注意到这个版本的程序的可读性比先前版本的要好。较好的可读性似乎总是暗示了较好的可靠性！

```
static long get_and_incr_x(long incr)
{
    static long x = 0;
    static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
    long rtn;
    ec_rv( pthread_mutex_lock(&mtx) )
    rtn = x ++ incr;
    ec_rv( pthread_mutex_unlock(&mtx) )
    return rtn;
}

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}

static void *thread_func(void *arg)
{
    while (get_and_incr_x(0) < (long)arg) {
        printf("Thread 2 says %ld\n", get_and_incr_x(1));
        sleep(1);
    }
    return (void *)get_and_incr_x(0);
}

int main(void)
{
    pthread_t tid;
    void *status;

    assert(sizeof(long) <= sizeof(void *));
    ec_rv( pthread_create(&tid, NULL, thread_func, (void *)6) )
    while (get_and_incr_x(0) < 10) {
        printf("Thread 1 says %ld\n", get_and_incr_x(1));
        sleep(2);
    }
    ec_rv( pthread_join(tid, &status) )
    printf("Thread 2's exit status is %ld\n", (long)status);
    return EXIT_SUCCESS;
}

EC_CLEANUP_BGN
    return EXIT_FAILURE;
```



```
EC_CLEANUP_END
}
```

这次决定使加锁或解锁错误变成致命错误，这也是另一种可行的选择。其本意当然不是让你在所有情况中都那样去做。真正的意图是要说明存在着许多处理这些错误的方法，并且要根据特定应用程序的需求来判断哪个是最好的。

理解`get_and_incr_x`函数中的这几行代码的工作原理是非常重要的：

```
ec_rv( pthread_mutex_lock(&mtx) )
rtn = x += incr;
ec_rv( pthread_mutex_unlock(&mtx) )
```

这几行代码实现的是对作为全局变量的`x`的访问，`x`在需要保护的线程之间共享。变量`rtn`和`incr`位于每一个线程的特有堆栈（也就是，每个线程都有自己的复制），且不需要保护。

更多关于这个程序的讨论放到了练习5.11中。

另外还有三种线程同步对象，用户可以在[SUS2002]中或在5.17节开头提到的书中查看它们：

- 读写锁（read-write lock）类似于互斥，但为了获得额外的并发性，它们区分了对读数据加锁和对写数据加锁。想从根本上了解更多有关读写锁的信息，可以查看7.11.4节。
- 旋转锁（spin lock）也类似于互斥，但它们是为较短的持续时间而准备的，而且比互斥更快。通常在一个CPU循环中，可以通过测试锁代替阻塞线程来实现它们。
- 隔离锁（barrier）是一个或多个线程等待的同步点，其保证允许继续执行任何线程前，线程已经完成了某个任务。

5.17.4 条件变量

假设线程A正在做某项工作（例如，从网络连接中读取数据）并向某个队列中增加了一些项目，而线程B正从该队列取项目，并在项目上面做一些额外的工作（例如，更新数据库）。使用互斥M来控制对队列的访问，可以如下来组织线程：

线程A	线程B
1. Read data [B]	1. Lock M [b]
2. Lock M [b]	2. If item on queue, remove it and update database
3. Put item on queue	3. Unlock M
4. Unlock M	4. Goto step 1
5. Goto step 1	

（符号[B]表示一个不确定的阻塞，而[b]表示一个短持续时间的阻塞。）

这样组织是没问题的，但当队列为空时，线程B会浪费大量的CPU时间，因为当其不断循环时，要不断地检查。可以采用如下方式使其慢下来：

线程A	线程B
1. Read data [B]	1. Lock M
2. Lock M [b]	2. If item on queue, remove it and update database
3. Put item on queue	3. Unlock M [b]
4. Unlock M	4. Sleep for 1 sec. [b]
5. Goto step 1	5. Goto step 1

但这种改进并不明显：现在线程B的响应减少了，因为当有工作要做时，它可能正在休眠，但还是浪费了CPU时间，因为它唤醒时，队列可能是空的。所期望的改进结果是，当线程A

向队列中放置某个项目时，用信号通知线程B，并且阻塞线程B，直到它获得这个通知信号。下面尝试用另一个互斥Q来描绘“队列是非空的”这个概念：

线程A	线程B
1. Read data [B]	1. Lock <i>M</i> [b]
2. Lock <i>M</i> [b]	2. If item on queue, remove it and update database
3. Put item on queue	3. Unlock <i>M</i>
4. Unlock <i>M</i>	4. Lock <i>Q</i> [B]
5. Unlock <i>Q</i>	5. Goto step 1
6. Goto step 1	

线程B尝试锁定*Q*（第4步）是个不确定的阻塞，因为仅当线程A从B读取（第1步）返回时，*Q*才会变为非锁定的。

现在的问题是当线程B还没有使*Q*锁定时，线程A可能就试图要解除*Q*的锁定了，这将导致信号丢失——解锁的尝试并不会被下一次加锁尝试记住（例如，互斥不会为信号量计数）。所以如果发生这种情况，为了等待*Q*上的加锁将阻塞线程B，并且这个加锁永远不会被释放，直到下一次读取数据，如果要读取的话。同时线程B也没有处理队列中的这一项。

使用*Q*的一个更具体的问题是只有加锁互斥的线程能够解锁。所以A的第5步将出现错误。

可以尝试用一个计数信号量（见7.8节）来替代*Q*，但一个更好的选择是使用POSIX线程提供的另一种信号机制：条件变量。下面例子中显示了如何使用它们：

线程A	线程B
1. Read data [B]	1. Lock <i>M</i> [b]
2. Lock <i>M</i> [b]	2. while (queue is empty) {
3. Put item on queue	cond_wait(C, <i>M</i>) [B]
4. cond_signal(C)	}
5. Unlock <i>M</i>	3. Remove item; update database
6. Goto step 1	4. Unlock <i>M</i>
	5. Goto step 1

第二个互斥*Q*已经不见了，现在只有一个条件变量*C*。在线程B中，cond_wait一直等到发送了条件*C*信号，其中信号发送是在线程A中的第4步发生的。但cond_wait用互斥*M*进行特殊的交互（它的第二个参数）：当调用cond_wait时，必须加锁该互斥。等待期间不会解锁，于是当cond_wait返回时又会自动加锁。使用这种方式时，不可能出现信号混乱或死锁，而在早期尝试的两种方法中都存在这两种缺陷。并且在*M*加锁的状态下，执行了所有存在于线程B的第2步和第3步的代码，情况也应该是这样。

仅当队列非空时并且条件*C*得到（线程A）发送的信号时，为什么cond_wait在检测队列是否为空的循环中呢？这是因为cond_wait像其他所有的UNIX阻塞系统调用一样，容易被中断（例如，被到达的传统UNIX信号中断，如SIGINT）。在这种情况下，可能有队列仍然为空的返回。测试此判定（在这个例子中是空队列）的循环确保了在这样一个不真实的返回的情况下，可以正确地返回到cond_wait。因为不真实的返回是不经常的，所以浪费的CPU时间是无关紧要的。

在调用cond_wait前要检查队列的另一个原因是，在线程B调用cond_wait前可能已经调用了cond_signal，而一个没有线程等待的cond_signal将被抛弃（并不是保持未处理状态）。所以重要的是不调用cond_wait，除非必须等待或者永远等待，即使队列非空。

因此发送信号或等待一个条件，需要三个要素：条件变量、互斥和判定。前两个是专门

的数据类型，但判定仅仅是一些使程序有意义的普通代码——它是条件变量抽象描绘的内容的具体细节。

下面是实际的pthread_cond_signal和pthread_cond_wait系统调用的对照表：

pthread_cond_signal —— 信号条件

```
#include <pthread.h>

int pthread_cond_signal(
    pthread_cond_t *cond    /* condition variable */
);
/* Returns 0 on success, error number on error */
```

pthread_cond_wait —— 等待条件

```
#include <pthread.h>

int pthread_cond_wait(
    pthread_cond_t *cond,    /* condition variable */
    pthread_mutex_t *mutex   /* mutex */
);
/* Returns 0 on success, error number on error */
```

当使用互斥时，用户可以静态地声明并初始化一个条件变量，代码如下：

```
static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

栈（自动的）中的条件变量或者动态分配的条件变量需要用pthread_cond_init调用进行初始化，本书中不研究此调用。

下面是一个示例程序，它通过一个初始线程向队列中放置节点，而另一个线程将节点取出并显示它们的内容。当节点被放入队列时，该初始线程会发送一个条件，而另一个线程正等待这个条件：

```
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

struct node {
    int n_number;
    struct node *n_next;
} *head = NULL;

static void *thread_func(void *arg)
{
    struct node *p;

    while (true) {
        ec_rv( pthread_mutex_lock(&mtx) )
        while (head == NULL)
            ec_rv( pthread_cond_wait(&cond, &mtx) )
        p = head;
        head = head->n_next;
        printf("Got %d from front of queue\n", p->n_number);
        free(p);
        ec_rv( pthread_mutex_unlock(&mtx) )
    }
    return (void *)true;
}

EC_CLEANUP_BGN
(void)pthread_mutex_unlock(&mtx);
EC_FLUSH("thread_func")
```



```

    return (void *)false;
EC_CLEANUP_END
}

int main(void)
{
    pthread_t tid;
    int i;
    struct node *p;

    ec_rv( pthread_create(&tid, NULL, thread_func, NULL) )
    for (i = 0; i < 10; i++) {
        ec_null( p = malloc(sizeof(struct node)) )
        p->n_number = i;
        ec_rv( pthread_mutex_lock(&mtx) )
        p->n_next = head;
        head = p;
        ec_rv( pthread_cond_signal(&cond) )
        ec_rv( pthread_mutex_unlock(&mtx) )
        sleep(1);
    }
    ec_rv( pthread_join(tid, NULL) )
    printf("All done -- exiting\n");
    return EXIT_SUCCESS;

EC_CLEANUP_BGN
    return EXIT_FAILURE;
EC_CLEANUP_END
}

```

将新节点加入队列的初始化线程的本质是遵循了前面说明的方式——使用加锁互斥来调用 `pthread_cond_signal`：[⊖]

```

ec_rv( pthread_mutex_lock(&mtx) )
p->n_next = head;
head = p;
ec_rv( pthread_cond_signal(&cond) )
ec_rv( pthread_mutex_unlock(&mtx) )

```

移去节点的线程也遵循了那种方式，使用加锁的互斥来调用 `pthread_cond_wait`：

```

ec_rv( pthread_mutex_lock(&mtx) )
while (head == NULL)
    ec_rv( pthread_cond_wait(&cond, &mtx) )
p = head;
head = head->n_next;
printf("Got %d from front of queue\n", p->n_number);
free(p);
ec_rv( pthread_mutex_unlock(&mtx) )

```

这两个线程能够成功运行就在于，当互斥处于等待状态时，`pthread_cond_wait`对其进行了解锁，并且在返回前又重新进行了加锁。因此确保了下面的内容：

- 处理队列的两个临界段都是在互斥 `mtx` 的保护下进行的。
- 当在第二个线程中执行语句

```
p = head;
```

时，`head`是非NULL的。

[⊖] 使用不加锁的互斥调用 `pthread_cond_signal`也是可以的，实际上那样可能可以增加一点吞吐量。

下面是得到的输出结果:

```
Got 0 from front of queue
Got 1 from front of queue
Got 2 from front of queue
Got 3 from front of queue
Got 4 from front of queue
Got 5 from front of queue
Got 6 from front of queue
Got 7 from front of queue
Got 8 from front of queue
Got 9 from front of queue
```

“ALL done-exiting”消息怎么了?从没输出过它。事实上,程序在输出了“Got 9”行后就挂起了,也可以键入Ctrl-C停止它。看一下程序,便可以知道原因:第二个线程始终停留在其循环中,在寻找队列中始终无法到达的节点,并且初始线程也始终无法从pthread_join调用中返回。

一个显而易见的解决方法是向队列中放置某类“文件结束”节点来告知第二个线程退出。或者,当没有其他更多的工作时,初始线程可以取消第二个线程。这里我们选择取消线程的方法,所以下面讨论怎样去取消线程。

5.17.5 取消线程

一个线程可以使用pthread_cancel来取消另一个线程:

pthread_cancel——取消线程

```
#include <pthread.h>

int pthread_cancel(
    pthread_t thread_id      /* ID of thread to cancel */
);
/* Returns 0 on success, error number on error */
```

通常,被取消的线程并不会立刻停止,而只是处在一个取消点(cancellation point)上,当线程调用到一个能够阻塞的系统调用或标准函数(如read、waitpid或pthread_cond_wait;约有200个左右^①)时,线程才停止。如果线程调用程序中或库中其他地方定义的函数,那么将很有可能会调用到那200多个系统调用或函数。因此用户应该考虑将任何一个函数调用作为潜在的而不是肯定的取消点,除非该函数被文档以不同方式专门规定,并且信任该文档。

取消点的意义在于可以安全地执行普通代码,而不需要担心它被取消。例如,当知道代码会按次序完成时,可以修改这个链表(可能被某个互斥保护)。

可以放心,“pthread”族的互斥调用都不是取消点,free、calloc、malloc或者realloc也不是。如果互斥调用的是取消点,使用互斥将是非常麻烦的,因为每次调用pthread_mutex_lock时,都必须增加代码控制取消。

如果线程根本没有取消点,或者没有可以依赖的取消点,那么为了使取消生效,线程仍需要存在足够长的时间,此时可以在安全的地方放入一个或多个pthread_testcancel调用来显式地提供取消点。如果没有任何悬而未决的取消,那么这个调用什么都不用做。

^① SUS规定了65个左右总是取消点的系统调用或函数,另外150个左右可以是取消点的系统调用或函数。

pthread_testcancel —— 尝试撤消

```
#include <pthread.h>
void pthread_testcancel(void);
```

前面讲到，通常情况下线程是在某个取消点取消。当取消类型是PTHREAD_CANCEL_DEFERRED时，是这样的，其中PTHREAD_CANCEL_DEFERRED是默认值。如果使用pthread_setcanceltype函数（详情请参阅[SUS2002]）将类型设置成PTHREAD_CANCEL_ASYNCHRONOUS，那么线程会被立即取消，这是个使人不安的想法。大概做这种改变的任何人都明白这一点。

现在修改前面示例中的main函数，以便当队列中不再有节点时取消另一个线程：

```
for (i = 0; i < 10; i++) {
    ec_null( p = malloc(sizeof(struct node)) )
    p->n_number = i;
    ec_rv( pthread_mutex_lock(&mtx) )
    p->n_next = head;
    head = p;
    ec_rv( pthread_cond_signal(&cond) )
    ec_rv( pthread_mutex_unlock(&mtx) )
    sleep(1);
}
ec_rv( pthread_cancel(tid) )
ec_rv( pthread_join(tid, NULL) )
printf("All done -- exiting\n");
return EXIT_SUCCESS;
```

通过这个改进，程序正确地终止了：

```
Got 0 from front of queue
Got 1 from front of queue
Got 2 from front of queue
Got 3 from front of queue
Got 4 from front of queue
Got 5 from front of queue
Got 6 from front of queue
Got 7 from front of queue
Got 8 from front of queue
Got 9 from front of queue
All done -- exiting
```

但是还没有完全做完。还需要仔细地查看被取消的线程的代码，以确保取消没有使队列处在一个不连续的状态。再次修改如下：

```
static void *thread_func(void *arg)
{
    struct node *p;

    while (true) {
        ec_rv( pthread_mutex_lock(&mtx) )
        while (head == NULL)
            ec_rv( pthread_cond_wait(&cond, &mtx) )
        p = head;
        head = head->n_next;
        printf("Got %d from front of queue\n", p->n_number);
        free(p);
        ec_rv( pthread_mutex_unlock(&mtx) )
    }
}
```

```

    }
    return (void *)true;

EC_CLEANUP_BGN
    (void)pthread_mutex_unlock(&mtx);
    EC_FLUSH("thread_func");
    return (void *)false;
EC_CLEANUP_END
}

```

事实上还有两个问题没有解决:

- `pthread_cond_wait`是一个取消点。如果线程在那儿取消,那么队列不会受影响,因为对它没有其他任何操作了。但当离开`pthread_cond_wait`时,总是重新锁定互斥。我们并不希望用互斥锁定来终止线程,因为那样的话,线程将永远不会被解除锁定。
- `printf`可能是个取消点。如果线程在那儿终止,将不会释放刚刚被移除的节点,结果导致内存溢出。

取消线程之后,程序确实是退出了,但情况并不总是这样的,所以修改这些漏洞是有意义的。保护可能发生而不是经常发生的情况是非常重要的。

第二个问题的一种解决方法是在局部变量中保存将被输出的数,接着释放节点,最后调用`printf`。

但对第一个问题没有有效的解决方法。必须安装一个取消清除处理程序,它是一个只在取消前做清理工作的函数。也可以在取消清除处理程序中解决第二个问题。

取消清除处理程序经常使用这个原型(名字并不重要):

```
void cleanup_handler(void *arg);
```

线程通过`pthread_cleanup_push`安装清除处理程序,通过`pthread_cleanup_pop`卸载它:

`pthread_cleanup_push`——安装清除处理程序

```

#include <pthread.h>

void pthread_cleanup_push(
    void (*handler)(void*), /* pointer to cleanup-handler function */
    void *arg                /* data to pass to function */
);

```

`pthread_cleanup_pop`——卸载清除处理程序

```

#include <pthread.h>

void pthread_cleanup_pop(
    int execute                /* execute handler? */
);

```

当取消发生时,调用清除处理程序。如果清除处理程序不止一个,那么将以入栈顺序的逆序来调用它们。每个函数被调用之后都会弹出,这样很好,因为线程也将随之消失。

这些函数必须成对出现,同时它们必须在同一级别的C或C++块中。如果不这样做,可能会出现非常奇怪的编译时错误,因为函数通常是以嵌入一对宏来实现的(像`EC_CLEANUP_BGN`和`EC_CLEANUP_END`;见1.4.2节)。

根据用户处理问题的方式,即使线程正常退出时,调用清除处理程序可能也是有意义的。为了操作简便,可以把`pthread_cleanup_pop`中的`execute`参数设置为真。

下面是thread_func的改进版本。注意为了确保p的值对free总是有效，显式地将p初始化为NULL了。

```
static void cleanup_handler(void *arg)
{
    free(arg);
    (void)pthread_mutex_unlock(&mtx);
}

static void *thread_func(void *arg)
{
    struct node *p = NULL;

    pthread_cleanup_push(cleanup_handler, p);
    while (true) {
        ec_rv( pthread_mutex_lock(&mtx) )
        while (head == NULL)
            ec_rv( pthread_cond_wait(&cond, &mtx) )
        p = head;
        head = head->n_next;
        printf("Got %d from front of queue\n", p->n_number);
        free(p);
        ec_rv( pthread_mutex_unlock(&mtx) )
    }
    pthread_cleanup_pop(false);
    return (void *)true;

EC_CLEANUP_BGN
    (void)pthread_mutex_unlock(&mtx);
    EC_FLUSH("thread_func")
    return (void *)false;
EC_CLEANUP_END
}
```

现在事情解决了。注意，当初始线程已经停止向队列中放置节点时，如果选择其他的方式终止线程，便可以避免处理取消的所有复杂情况。这是需要在用户的应用程序中考虑的问题——当有较灵活的方式可以处理这个工作时，不要使用取消。另一方面，如果要终止的线程被阻塞了（如在read中）并且不能很容易地解除阻塞，那么线程取消可能是最好的选择了。和信号相比，它当然是个更好的选择，因为它可以同时解除系统调用的阻塞；如在第9章中所讨论的，信号有更糟糕的副作用。

5.17.6 线程和进程

用户可能会对什么时候应该使用线程和什么时候应该使用进程感到疑惑。通常需要对具有同样复杂性的数据结构进行并发处理时，要使用线程。三种常见情况如下：

- 当其他计算正在后台运行时，需要用户接口是活动的。例如字处理程序中的后台输出或后台格式化，字处理程序允许文档的查看和编辑与那些后台操作并发进行。
- 需要设计好的算法以便充分利用多处理功能计算机——一个带有多个CPU的计算机。对于这样一个系统，UNIX调度程序能够自动地给不同的CPU分配不同的线程。
- 需要处理多种类型的引起系统阻塞调用（如，read、waitpid、msgrcv）的事件。这将是下一节的主题。

对于紧耦合度不高的应用程序，将会使用进程，一些数据会在那些应用程序间传递，但它们没有必要直接操作相同的数据结构。（使用共享内存来共享数据结构是可行的，但这有一

些麻烦，第7章中会介绍这些内容。)单独的进程都有各自的有效用户ID、文件描述符、全局变量等，然而单独的线程却没有。进程能够更容易地被独立开发、测试和调试，并且很少需要锁定，因此也很少出现不可检测的紊乱情况或死锁。比较进程与线程的另外一种方法是，可以说进程是针对应用程序的大块组成部分，而线程提供的是更细致的并发性。

警告：在编写本书时，Linux和FreeBSD中安装的线程包通常并不遵循POSIX标准，这主要因为它们在实际线程时要么太弱（在用户空间内，什么都做），要么太强（每个线程使用一个进程）。前者的主要问题是某个阻塞系统调用，像`msgrcv`，会阻塞所有的线程，不仅仅是包含这个调用的线程。后者的问题是一些像`waitpid`这样的系统调用不能正确地工作，因为它们处在错误的进程中（只有父进程可能会等待子进程）。另一个麻烦是，如果系统自带的线程包不够用，那么用户可能不得不自己去寻找、编译和安装线程包。

可以寻找最新的称做Native POSIX Thread Library for Linux (NPTL)的Linux线程实现，现在正寻找将其安装到Linux系统中的办法，它解决了所有POSIX中存在的重要漏洞。

5.18 阻塞问题

本书已经介绍了一些能够引起阻塞的系统调用，如`read`、`write`、`pthread_cond_wait`以及`waitpid`，在本书中还有许多，特别是在第7章和第8章中。UNIX编程的一个最大困难是用户的应用程序很可能不只在在一个系统调用中阻塞，因为用户并不知道下一步将发生什么，本书称这种情况为阻塞问题。

对于能取得文件描述符的阻塞系统调用，像`read`和`write`，可以使用一个单独的系统调用如`select`或`poll`（见4.2节）进行阻塞，直到一个或多个文件描述符准备好才解除阻塞。但通常那并没有什么帮助，因为等待的东西太多了，如进程、信号、消息、信号量和条件变量，它们和文件描述符并没有联系。当然，可以在`select`或`poll`中通知被阻塞的线程，比方说，当消息到达时，可以通过向只为这种目的而建立的管道写入消息（下一节中将会看到），但这对`msgrcv`或`mq_receive`首先等待消息而阻塞自身的情况没有用处。没有直接将某个消息等待系统调用绑定到文件描述符的方法，这不是通知问题，而是阻塞问题。

5.18.1 进程和线程的解决办法

历史上，解决阻塞问题方法是创建阻塞的子进程。^①当阻塞消除时，无论创建的阻塞子进程正阻塞什么，它都会将其写入管道，并告知其父进程已经发生的事件。管道是一个不错的选择，因为它们容易建立（下一章将会看到），而且因为它们使用文件描述符，所以父进程可以将它们合并到`select`或`poll`中。事实上，这么做是将父进程从非文件描述符事件阻塞转变成了文件描述符上的阻塞。在父进程中使用`select`或`poll`转换任何东西都是最好的。

但是，在UNIX中进程是重量级的对象——创建和调度开销很大，而且供应有限。同时两个进程共享同样的数据结构是非常麻烦的。在阻塞发生时，如果事件恰好能够排队等候，并且父进程无论什么时候方便，都可以检查它，那么这就足够了。使用共享进程和信号量，在进程间设置事件队列是可行的，但交互进程信号量可能太缓慢了，尤其对于事件产生迅速的情况。

另一个使用独立进程所产生的问题是某些对象（如文件描述符）无法被传递给现存的进程（至少是不方便的）。它们只能通过继承来传递；因此如果进程A负责阻塞I/O，并且已经

① 该技术是我母亲在1953年左右发明的。在食品店，她让一个小孩在熟食处排队，让另一个小孩在鱼食品处排队，而她在不需要排队的地方购买商品。但是偶尔也会阻塞，因此她还有3个孩子。

在运行了，那么刚刚打开网络连接的进程B是不能允许进程A去等待以获得新文件描述符的。

对阻塞问题比较新的解决方法是使用POSIX线程。一个简单的方法是为每个要阻塞的对象创建一个新线程（例如，消息队列、文件描述符、信号量）。每个线程将只引发一个合适的阻塞系统调用（O_NONBLOCK清除）。当该系统调用返回时，线程会向共享队列中添加一个事件（为了保护使用互斥）然后返回到阻塞系统调用中。之后，主线程便只有一件要阻塞的东西：队列中的当前事件。和5.17.4节的例子非常相似，当队列为非空时，阻塞线程将使用条件变量给主线程发送信号。

5.18.2 统一的事件管理程序原型

这里介绍一个普遍的方法来解决阻塞问题，该方法称为统一的事件管理程序（Unified Event Manager）。它是一个任何应用程序都可以使用的库函数集。应用程序可以注册一个能促使该库创建线程进行阻塞的事件。在等待单个事件队列的附近重新组织应用程序。当事件出现后，应用程序会把它从队列中取走，对其进行处理，然后继续等待。

本书中没有给出所有的代码，如果需要，可以到网站上去看。之所以称其为原型，是因为对关键应用程序来说，其效率不够高，同时因为使用了本书所有例子中采用的“ec”错误检查方法，其中“ec”本身也是个原型。

下面开始列举在UNIX中可能被等待的所有事件：

```
enum UEM_TYPE {
    UEM_SVMSG,      /* System V message */
    UEM_PXMSG,      /* POSIX message */
    UEM_SVSEM,      /* System V semaphore */
    UEM_PXSEM,      /* POSIX semaphore */
    UEM_FD_READ,    /* file-descriptor set - read */
    UEM_FD_WRITE,   /* file-descriptor set - write */
    UEM_FD_ERROR,   /* file-descriptor set - error */
    UEM_SIG,        /* signal */
    UEM_PROCESS,    /* process */
    UEM_HEARTBEAT,  /* heartbeat */
    UEM_NONE        /* none */
};
```

最后两个需要解释一下：UEM_HEARTBEAT是在某些特定间隔获得一个周期性事件的方法，之所以UEM_NONE也在其中，是因为它是一个能使用某个值指示某物为空的好方法。

当注册一个事件时，需用一个结构来明确地跟踪需要阻塞系统调用得到什么。例如发起select时，需要设置一个文件描述符，所以该结构为每个UEM_TYPE保留了所有这种类型的数据。

```
struct uem_reg {
    enum UEM_TYPE ur_type;      /* type of registration */
    pthread_t ur_tid;          /* thread ID */
    union {
        int ur_mqid;           /* System V message-queue ID */
        struct {
            int s_semid;       /* System V semaphore-set ID */
            struct sembuf *s_sops; /* semaphore operations */
        } ur_svsem;
#ifdef POSIX_IPC
        mqd_t ur_mqd;          /* POSIX message-queue descriptor */
        sem_t *ur_sem;        /* POSIX semaphore */
#endif
        int ur_signum;         /* signal number */
    };
};
```

```

    pid_t ur_pid;           /* process ID */
    long ur_usecs;         /* microseconds (for heartbeat) */
    fd_set ur_fdset;      /* file-descriptor set */
} ur_resource;
void *ur_data;           /* data to be queued with event */
size_t ur_size;         /* size (used for various purposes) */
};

```

虽然宏POSIX_IPC不是一个标准宏，但在原型中使用了它。它是以一个非常复杂的方式从真实的特征测试宏中设定的，这在1.5.4节中解释过了。之所以这里需要它，是因为Linux和FreeBSD至今也不支持POSIX IPC。

当应用程序想要注册某一事件类型时，需要调用uem_register_E形式的函数，这里的E是事件类型的缩写。例如，下面注册的调用实现了等待进程终止：

```
ec_false( uem_register_process(pid, NULL) )
```

这里没有直接调用waitpid，因为那样会阻塞。当进程pid终止时，包含退出状态的事件会被放置到队列中，并且注册这个事件的应用程序也可以得到这个状态。下面简单地看一下那些细节。

这是uem_register_process的一些代码：

```

bool uem_register_process(pid_t pid, void *data)
{
    struct uem_reg *p;

    ec_null( p = new_reg() )
    p->ur_type = UEM_PROCESS;
    p->ur_resource.ur_pid = pid;
    p->ur_size = 0;
    p->ur_data = data;
    ec_rv( pthread_create(&p->ur_tid, NULL, thread_process, p) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

```

所有uem_register_E调用使用的函数new_reg，只分配了一个结构。这里把它放到一个独立的函数中，是为了防止做当前设计没有的一些常用的初始化工作。

```

static struct uem_reg *new_reg(void)
{
    struct uem_reg *p;

    ec_null( p = calloc(1, sizeof(struct uem_reg)) )
    return p;

EC_CLEANUP_BGN
    return NULL;
EC_CLEANUP_END
}

```

注册信息只能被传递给线程函数，代码如下：

```

static void *thread_process(void *arg)
{
    struct uem_event *e = NULL;

```



```

pthread_cleanup_push(cleanup_handler, e);
ec_null( e = calloc(1, sizeof(struct uem_event)) )
e->ue_reg = (struct uem_reg *)arg;
if (waitpid(e->ue_reg->ur_resource.ur_pid, &e->ue_result, 0) == -1)
    e->ue_errno = errno;
ec_false( queue_event(e) )
pthread_cleanup_pop(false);
return NULL;

EC_CLEANUP_BGN
    uem_free(e);
    EC_FLUSH("thread_process")
    return NULL;
EC_CLEANUP_END
}

```

这个函数首先把一个清除句柄（在5.17.5节中解释）压入了栈中。然后当这个事件发生时，程序会分配一个事件结构，并将其加入到事件队列中（waitpid在这种情况下返回）。下面是所有这些线程使用的事件结构：

```

struct uem_event {
    struct uem_reg *ue_reg;
    void *ue_buf;
    ssize_t ue_result;
    int ue_errno;
    struct uem_event *ue_next;
};

```

注意，它重新指向了注册，该注册包含了此类型所有事件都常用的数据。ue_buf成员用于防止数据必须返回的情况（例如消息），但在这个示例中没有使用它。我们确实有这种状态，但我们将它被放到了ue_result成员中。如果发生错误，errno将成为ue_errno的成员；获得这个事件的应用程序需要核对成员是否为零，以便查看是否被等待的函数返回了错误。ue_next成员是为了把uem_event结构链接到某个事件队列中。

如果分配了事件结构，那么除了清除句柄需要调用uem_free（这里没有给出）释放事件结构之外，清除句柄与5.17.5节中的一样：

```

static void cleanup_handler(void *arg)
{
    (void)uem_free((struct uem_event *)arg);
}

```

从队列中移除事件的应用程序也要对调用uem_free负责。

向队列中放入事件的实际工作将由queue_event来完成。当waitpid返回时，线程会调用它。注意即使waitpid报告出现了错误，事件也是要加入队列的，这样应用程序才能发现这些错误的来源。

```

static pthread_mutex_t uem_mtx = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t uem_cond_event = PTHREAD_COND_INITIALIZER;
static struct uem_event *event_head;

static bool queue_event(struct uem_event *e)
{
    struct uem_event *cur;

    ec_rv( pthread_mutex_lock(&uem_mtx) )
    if (event_head == NULL)
        event_head = e;
}

```

```

else {
    for (cur = event_head; cur->ue_next != NULL; cur = cur->ue_next)
        /* queue same error only once */
        if (e->ue_errno != 0 &&
            cur->ue_reg->ur_type == e->ue_reg->ur_type &&
            cur->ue_errno == e->ue_errno) {
                ec_rv( pthread_mutex_unlock(&uem_mtx) )
                uem_free(e);
                return true;
            }
        cur->ue_next = e;
    }
    ec_rv( pthread_cond_signal(&uem_cond_event) )
    ec_rv( pthread_mutex_unlock(&uem_mtx) )
    return true;

EC_CLEANUP_BGN
    (void)pthread_mutex_unlock(&uem_mtx);
    return false;
EC_CLEANUP_END
}

```

这个函数继承了5.17.4节的方法，即使用条件变量发送事件信号。尽管如此，如果调用 `queue_event` 的线程不断获得某一错误，那么错误事件（`ue_errno`非零）就会被不断地重生成。例如，如果传递给 `waitpid` 的进程ID是无效的，那么 `waitpid` 将保持返回状态，直到取消线程。这样浪费CPU时间已经够糟糕了，但当然也不希望同样的报告填满事件队列。所以，在将错误事件放入队列之前，必须确定队列中没有和它相似的事件。（这不是最好的方法，但这是一个原型，对吗？）

如果要把事件放入到队列中，那么首先要用代码

```
cur->ue_next = e;
```

把它排队，然后发送条件，为互斥解锁，然后返回。

除了诸如 `uem_register_process` 的注册函数外，基本上这就是整个库了。这里就不介绍那些注册函数了，因为它们做的事情几乎都是相同的，只是在阻塞系统调用上有所不同。也就是说，`uem_register_svmsg` 启动包含 `msgrcv` 调用的线程，`uem_register_pxmsg` 启动包含 `mq_receive` 调用的线程，等等。它们都使用 `queue_event`。

使用库的应用程序使用如下的代码：

```

struct uem_event *e;
...
ec_false( uem_register_process(pid, NULL) )
ec_false( uem_register_pxmsg(mqd, NULL) )
...
while (true) {
    ec_null( e = uem_wait() )
    if (e->ue_errno != 0)
        ... /* display error */
    else
        switch (e->ue_reg->ur_type) {
            case UEM_PXMSG:
                ... /* process received message */
                break;
            case UEM_PROCESS:
                ... /* process status from terminated process */
                break;
            ...
        }
}

```



```

    }
}

```

这里最重要的事情是：无论有多少种不同的事件需要处理，应用程序都只在一个地方阻塞，即`uem_wait`调用。下面是`uem_wait`的代码；注意，和`queue_event`一样，它遵照的也是5.17.4节中的条件变量：

```

struct uem_event *uem_wait(void)
{
    struct uem_event *e = NULL;
    ec_rv( pthread_mutex_lock(&uem_mtx) )
    while (event_head == NULL)
        ec_rv( pthread_cond_wait(&uem_cond_event, &uem_mtx) )
    e = event_head;
    event_head = event_head->ue_next;
    ec_rv( pthread_mutex_unlock(&uem_mtx) )
    return e;

    EC_CLEANUP_BGN
        (void)pthread_mutex_unlock(&uem_mtx);
        return NULL;
    EC_CLEANUP_END
}

```

当判定为真时（队列中现存的事件），将从队列中移除该事件，然后返回一个指向它的指针。像先前所提到的，用`uem_free`调用释放内存是调用者的责任。

除了其他一些额外的细节和注册函数外，这便是整个系统。它解决了阻塞问题！

因为它对线程是非常可靠的，所以这种方法像基础线程实现一样的好。线程必须是快的、轻量级的、丰富的，并且它们必须严格地遵循POSIX标准。否则开销太大，会消耗宝贵的系统资源，而且在程序中也会有许多的小漏洞，因为所有的多线程都是很难查找的。

如果有兴趣，可以试着使用进程取代线程重新写`uem`包。你会发现写它是非常困难的，而且想提高效率是极其困难的。但这种努力将会有助于理解线程的重要性。

练习

- 如5.2节末尾建议的那样，纠正`setenv`和`unsetenv`中存在的内存溢出问题。
- 重新编写5.2节中的环境处理函数，以便能像标准shell那样处理被输出的变量。也就是说，只有当特别宣布了在函数（如`env_export`）中输出时，才输出变量的更新值。想一想是否需要更新`environ`值，何时更新，是原样使用已有的`getenv`函数，还是需要替换它。
- 编写实现扫描形如`variable=value`赋值参数的程序，恰当地更新环境，接着用第一个未赋值参数执行程序。其他未赋值参数成为被调用程序的参数。不要使用`fork`。
- 编写一个与5.3节`execvp2`类似的函数`execlp2`。使用标准C变量参数设施（`va_arg`等）。
- 增强5.3节中`exec_path`函数的功能，以使它支持那节讨论的`#!`特征。
- 如5.3节脚注中建议的那样，设计和实现`execvx`和`execlx`函数以替代6个`exec`系统调用。在实现中可以随意使用`exec`系统调用。
- 用自己的话（也可以用图表）解释`fork`和`exec`的典型实现。（需要做一些研究，例如参阅[Bac1986]、[McK1996]、[Mau2001]或者[Bov2001]。）然后说明如何能够更加有效率地实现`posix_spawn`。不用写代码——可以用伪代码或清楚的一步一步的算法。
- 如5.5节末尾建议的那样，研究`posix_spawn`的语义[参阅SUS2002]，并用`fork`和`exec`来实现它。首

- 先，跳过属性和动作，然后实现动作，最后再实现属性。为了完成整个工作，需要实现相关的系统调用（例如，`posix_spawn_file_actions_init`），尽管你不关心实时系统，但这是一个极其有用的练习。
- 5.9 设计并运行实验来检测`fork`使用的CPU时间。（也许需要使用1.7.2节的`timestart`和`timestop`。）如果你有权使用它们，那么可以尝试不同的UNIX版本和硬件。如果系统支持，比较`fork`和`vfork`的使用时间。对`posix_spawn`也采取同样的步骤。
- 5.10 研究其他系统（如VMS、OS/390、Windows和MacOS）提供的与`exec`和`fork`对等的系统调用。比较它们的特征，并总结它们的优点和缺点。
- 5.11 在5.17.3节，在用`get_x(0)`进行读取和用`get_x(1)`增加步长之间，另一个线程可能还会增加`x`的步长，结果导致`x`变得太大。重新编写该示例，检测和增加步长`x`时都利用单个函数调用，这样可以修正这个问题。
- 5.12 写一个交互的全屏幕应用程序（见4.8节）`fileview`，实现用一条水平线把屏幕分割为两部分。“`s`”命令提示搜索字符串，搜索标准的包含文件（至少），从中找出那些在一行或者多行上包含该字符串的文件，并在上面部分显示匹配的路径名。当显示它们时或显示之后，“`v`”命令在下面部分显示被选择文件的内容，并高亮度显示匹配的字符串。选择两个键用于上下滚动上面部分，选择其他两个键用于下面部分。为了使“`v`”容易实现，为上面部分的路径名进行编号，并安排“`v`”命令用文件对应的编号给出文件提示。注意短语“当显示它们时”中的“当”，它暗示必须使用多线程，因为Curses不一定是线程安全的，所以必须使用互斥保护对它的访问。此外还要有一个退出应用的“`q`”命令。
- 5.13 不用多线程而用进程，解释如何执行`fileview`（练习5.12）。首先可能必须研究第7章。如果认为不用多线程不可以实现，那么请解释不能实现的原因。（因为尝试证明的是一个否定陈述，所以解释必须很有说服力。）
- 5.14 写一个用于尽可能多地显示附录A中列出的进程属性的程序，现在限制仅显示本书前面5章讨论的内容，以后可以拓展自己的程序包含全部内容。如果某些内容不能显示，解释不能实现的原因。为了使输出有意义，可以在打开某些文件的开始部分执行一些系统调用，设置一些信号动作等。



第6章 基本的进程间通信

6.1 概述

既然知道了如何创建进程，就需要知道如何连接它们以便它们能够相互通信。本章将使用管道来实现这项工作，其中采用的基本技术是所有UNIX版本都支持的。下一章中将使用更高效的、更鲁棒的、不太通用的、更易出错的编程技术来研究进程间通信。

作为shell设施，管道为大多数UNIX用户所熟悉。例如，要显示一个登录用户的排序列表，可以输入如下命令：

```
$ who | sort | more
```

这里有三个进程，由两个管道连接在一起。数据只从一个方向流入，从who到sort到more。使用系统调用也可以建立双向通信的管道（从进程A到进程B和从进程B回到进程A）和环形管道（从A到B到C到A）。但是，大多数shell都没有为这些更详细的任务提供提示，所以大多数UNIX用户都不知道它们。^①

这里首先给出一些与单向通信连接进程相关的简单例子，然后将会改进第5章开发的原始shell。新shell将足够称得上是“真正的”shell——它将能够处理管道、后台进程、I/O重定向以及引用的参数。但它缺少文件名生成（例如，ls t*.*?）和编程结构（如if语句）。最后说明如何连接双向通信进程，并且指出可能引起的死锁问题。

6.2 管道

本节讨论的是未命名管道，尽管这里描述的许多行为同样也适用于FIFO（命名管道）。在7.2节中将详述FIFO。

6.2.1 pipe系统调用

pipe——建立管道

```
#include <unistd.h>

int pipe(
    int pfd[2]          /* file descriptors */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

pipe系统调用可以创建一个管道，这个管道是由pfd数组返回的两个文件描述符表示的一个通信信道。向pfd[1]中写是往管道输入数据；从pfd[0]中读是从管道取出数据。

在UNIX文档中有一个叫作PIPE_BUF的参数，可以把它当作管道缓冲区的大小。如果多个进程或线程正在对同一个管道进行写操作，那么PIPE_BUF或更少字节的写操作应该保证

^① 通过FIFO可以创建这个任务，但shell是个无知的旁观者，即它会认为它们是规则的文件。

为原子操作——将没有其他写进程的数据的交叉操作。如果多个进程都在写结构化的数据，那么这个属性就显得非常重要了，因为没有它将不能保证读进程能够读到有效的数据。PIPE_BUF要始终不小于512，但是可以在运行期间通过调用fpathconf（见1.5.6节）得到某个特定管道的实际值：

```
int pfd[2];
long v;

ec_negl( pipe(pfd) )
errno = 0;
v = fpathconf(pfd[0], _PC_PIPE_BUF);

if (errno != 0)
    EC_FAIL
else if (v == -1)
    printf("No limit for PIPE_BUF\n");
else
    printf("PIPE_BUF = %ld\n", v);
```

在本系统中，在Solaris上得到的是5120，FreeBSD的是512，Linux的是4096。

最初，管道清除了O_NONBLOCK（见4.2.2节），也就是说，读和写可能会阻塞。也许你已经猜到了，可以使用fcntl（见3.8.3节）来设置标志。下面将解释这个标志是怎样影响read和write的。

6.2.2 管道（和FIFO）I/O特性

本节中的所有内容都既适用于管道，也适用于FIFO（7.2节中将进一步讨论），并且这里的术语“管道”指的是它们两个。

某些I/O系统调用对管道文件描述符的操作与对普通文件的不同，而且有一些根本什么都不做，如下面列表概括的（这些是主要的几个，[SUS2002]有更详细的内容）：

write 数据按到达顺序依次写入管道。通常（清除O_NONBLOCK），如果管道满了，write将阻塞，直到read移除了足够的旧数据；没有局部写。依据UNIX的不同实现，管道的容量有所不同，但显然总是不小于PIPE_BUF字节。如果设置了O_NONBLOCK，而且将被写入的数量是PIPE_BUF或更少，则write要么立即写入数据，要么返回-1且把errno设置为EAGAIN；没有局部写。但如果数量超过PIPE_BUF，局部写是有可能的。

read 和写入时一样，按到达顺序读取管道数据。一旦读了数据，就不能重读数据或将它放回。通常（清除O_NONBLOCK）情况下，如果管道是空的，那么read将阻塞，直到至少有一字节的数据可用，除非关闭所有的写入文件描述符，这种情况下，read返回0（通常是文件结束指示）。但read的第三个参数的字节计数不必要满足——只要和那个时刻读取的字节相同，并且返回一个合适的计数就行了。当然，永远也不会超越该字节计数；下一个读操作可以读没有被读的字节。如果设置了O_NONBLOCK，那么空管道上的读操作将返回-1，并且设置errno为EAGAIN。

close 关闭管道比关闭文件做的工作要多。不仅要释放文件描述符使之可以重用，而且当关闭了所有的写文件描述符时，对读进程而言，它还充当了文件结束的角色。如果关闭所有读文件描述符，那么在写文件描述符上的write会引起错误。通常还会产生一个致命的信号；见9.1.3节。

fstat 除了决定文件描述符向管道打开以外，对管道的用途不大。通常返回的大小是管道中的字节数，但是任意一个UNIX标准都不需要这个数。

dup 在6.3节中将解释这个系统调用和dup2。

lseek 不和管道一起使用。这意味着即使管道包含一个消息序列，浏览这些序列也无法查看到下一条将被读取的消息。就像一个牙膏管里面的牙膏一样，必须将它取出才能检查它，然而取出之后就无法将它再放回去了。这就是为什么那些在进程间传递消息的应用程序很难使用管道的原因之一。

这里没有像所期望的那样明确列出系统调用对管道的操作（如，select、poll）。例如，当select测试的一个文件描述符对某个管道是打开的，那么select会测试read或write是否会阻塞。

对于写操作而言，原子和非原子之间的关系，阻塞和非阻塞之间的关系，完全的、部分的和延迟的（返回-1并带有值为EAGAIN的errno）之间的关系，这些关系本身都有点复杂。表6-1（基于POSIX1990中的表格）对理解这些关系会有帮助。前两列包含了可能会出现O_NONBLOCK标志的状态和将要写入的数量。而后三列说明了满管道、能立即接受部分数据的管道、能接受所有数据的管道会发生什么。

表6-1 写管道

O_NONBLOCK?	写的总数	无立即可写的	部分立即可写的 (>=1和<总数)	所有立即可写的
清除	<=PIPE_BUF	阻塞；完全写；原子的	阻塞；完全写；原子的	不阻塞；完全写；原子的
清除	>PIPE_BUF	阻塞；完全写；非原子的	阻塞；完全写；非原子的	可能阻塞；完全写；非原子的
设置	<=PIPE_BUF	EAGAIN	EAGAIN	不阻塞；完全写；原子的
设置	>PIPE_BUF	EAGAIN	不阻塞；部分或EAGAIN；原子的	不阻塞；完全、部分或EAGAIN；非原子的

在该表中，符号“完全写”表示write直到所有请求的数量都被写入时才返回。“非原子的”意味着数据全部都在管道中，但不必是连续的（这种情况下，即使最开始的PIPE_BUF个字节也不保证为连续的）。“EAGAIN”代表write返回值为-1，且把errno设置为EAGAIN。“部分的”代表write返回的比要求返回的数量少。

第二行末尾带有“可能阻塞”符号的原因是：当write开始时，所有的数据都准备好，因为write是非原子的，所以在write完成之前，另一个进程或线程可能填充一部分管道而使它阻塞，因此可能阻塞是真的。

再一次对管道的写操作进行概括：

- 如果请求的数量是PIPE_BUF或更少，那么写操作将总是原子的（意味着绝对不会局部写）。
 - 如果清除了O_NONBLOCK（通常情况），即使它们是非原子的，写操作也决不会是局部的。
 - 仅当设置了O_NONBLOCK，并且请求的数量比PIPE_BUF大时，才会发生局部写。
- 表6-2是关于read的。

表6-2 读取管道

O_NONBLOCK?	无立即可读的	部分或所有立即可读的 (>=1和<总数)
清除	除非没有写操作，否则阻塞（返回0）	不阻塞；可能部分读
设置	除非没有写操作，否则为EAGAIN（返回0）	不阻塞；可能部分读

注意，read表比write表要简单得多，因为从来不用去确保原子读或完全读。read工作过程如下：

- 如果关闭了所有的写文件描述符，那么对空管道的read将总是立刻返回0，大多数程序都把它当作文件结束。
- 如果写文件描述符是打开的，那么不管是否设置了O_NONBLOCK，空管道的read都会阻塞。
- 对非空管道的read总是立刻返回，无论在管道中请求读的数量和管道中数据数量的关系如何。返回的是实际读取的数量。
- 因为没有原子操作的保证，所以决不能允许多个读进程同时进行，除非有其他的并发控制机制（例如，进程间信号量）阻止同时发生的读操作。（实践中很少这么做——可以使用一些像消息队列这样的东西代替）

此外还有一些需要记住的原则：

- 如果有一个读进程和一个写进程（例如，一个shell管道），而且读进程为局部读取做好了准备（例如，使用标准C的I/O函数），那么可以按需求随意写，尽管多个块大小是最有效率的，如2.12节中介绍的。
- 如果有多个写进程（和一个读进程），那么始终可以写PIPE_BUF或更少的字节。实践中无法使管道正确地处理更大数量的字节，除非使用另一种同步机制，如信号量（见7.8节）。
- 除了至少512字节外，不要随意假定PIPE_BUF的值。如果真的需要知道它的值，那么可以使用fpathconf。
- 即使标准没有要求原子读，但如果请求的数量是PIPE_BUF或更少，那么所有的实现本质上也都要使其成为原子的。这样用实际上在于你所冒的风险。
- 记住，为了得到文件结束符（从read返回0），所有写文件描述符（包括正在进行读操作的进程中所拥有的）都必须关闭。在后面说明如何用管道连接两个进程时，这一点会更清晰。

6.2.3 管道示例

（再一次申明，现在只讨论未命名管道；FIFO的例子在下一章介绍。）

仅考虑单个进程时管道的用途是什么呢？没用，但像这样的例子提供的信息是很多的：

```
void pipetest(void)
{
    int pfd[2];
    ssize_t nread;
    char s[100];

    ec_neg1( pipe(pfd) )
    ec_neg1( write(pfd[1], "hello", 6) )
    ec_neg1( nread = read(pfd[0], s, sizeof(s)) )
    if (nread == 0)
        printf("EOF\n");
    else
        printf("read %ld bytes: %s\n", (long)nread, s);
    return;
}

EC_CLEANUP_BGN
    EC_FLUSH("pipetest");
```



```
EC_CLEANUP_END
}
```

输出如下:

```
read 6 bytes: hello
```

可以安全地向管道中写入6个字节而不用担心会填满,因为它低于512(PIPE_BUF的最小值)。但是如果写得太多,而且确实填满了管道,那么write会阻塞,直到read将管道腾出来一些。然而,这是决不可能发生的,因为程序决不会得到read。此时可能会处于一种叫作死锁(deadlock)的状况。当使用管道时,必须小心以避免死锁,但也不必过分地关注它:当通过用于单向通信(如shell所做的那样)的单个管道连接进程时,死锁(应归于管道)是不可能发生的。只有更奇特的安排才会引起死锁。

假设现在有两个进程,怎样连接它们,才能使一个进程能从另一个进程写入的管道中读取呢?这是不可能实现的。一旦创建了进程,就不能连接它们了,因为对于创建管道的进程来说,无法向另一个进程传递文件描述符。^①当然可以传递文件描述符编号,但该编号在另一个进程中是无效的。如果在创建另一个进程之前,在一个进程中建立了一个管道,那么另一个进程将继承管道的文件描述符,而且在两个进程中它们都是有效的。因此,通过管道通信的两个进程可以是父子,或者两个都是子进程,或者祖孙关系等等。无论如何,它们必须是相关的,而且管道必须在建立时被传递。实践中,这可能是一个严格的限制,因为只要进程消失,就无法重新建立,而且也无法再重新把它连接到它的管道上——存活着的进程也必须要被删除,然后必须重建整个进程家族。

在下面的例子中,一个进程(运行函数pipewrite)生成一个管道,创建一个继承它的子进程(运行piperead),然后向管道中写入一些数据以便让子进程读取。尽管子进程已经继承了必要的读文件描述符,但仍然不知道它对应的编号,所以该编号必须作为参数传递。

```
void pipewrite(void)
{
    int pfd[2];
    char fdstr[10];

    ec_negl(pipe(pfd))
    switch (fork()) {
        case -1:
            EC_FAIL
        case 0: /* child */
            ec_negl(close(pfd[1]))
            snprintf(fdstr, sizeof(fdstr), "%d", pfd[0]);
            execlp("./piperead", "piperead", fdstr, (char *) NULL);
            EC_FAIL
        default: /* parent */
            ec_negl(close(pfd[0]))
            ec_negl(write(pfd[1], "hello", 6))
    }
    return;

    EC_CLEANUP_BGN
    EC_FLUSH("pipewrite");
    EC_CLEANUP_END
}
```

① 一些系统采用了一种不太方便的方式来达到这个目的。

下面是子进程的代码:

```
int main(int argc, char *argv[])
{
    int fd;
    ssize_t nread;
    char s[100];

    fd = atoi(argv[1]);
    printf("reading file descriptor %d\n", fd);
    ec_negl( nread = read(fd, s, sizeof(s)) )
    if (nread == 0)
        printf("EOF\n");
    else
        printf("read %ld bytes: %s\n", (long)nread, s);
    exit(EXIT_SUCCESS);

    EC_CLEANUP_BGN
        exit(EXIT_FAILURE);
    EC_CLEANUP_END
}
```

下面是输出:

```
reading file descriptor 3
read 6 bytes: hello
```

一些关于该例子的解释如下:

- 因为该子进程只是读管道，而不是写管道，所以为了保存文件描述符，立刻（在父进程的case 0语句后）关闭了写结尾（pdf[1]）。（如果该子进程需要读取更多，那么关闭写结尾将是极其重要的，否则子进程将无法获得文件的结尾。）
- 管道的读结尾对父进程没有任何用处，所以父进程关闭了pdf[0]。
- snprintf把读文件描述符从一个整型转换成一个字符串，以便能够作为程序参数使用。
- 因为知道该子程序在当前的目录中，所以将它的路径编写为./pread以避免execlp查找。这会节省时间，但更重要的是，它反而会阻止其他一些piperead的意外执行（而不必告知在用户的路径中可能有什么）。也可以使用execl完成同样的事情，execl不会查找。
- 不需要为该子进程编写wait代码，因为在这个例子中，父进程会马上退出。
- 该子进程运行piperead，把子进程的参数转换回整数，并且从文件描述符中读取。（记住，要知道那个整数不能使文件描述符有效——即文件描述符本身是有效的，因为它是继承的。）

通常情况下，下面是用管道连接两个进程进行单向通信的方法:

- 1) 创建管道。
- 2) 派生创建读子进程。
- 3) 在子进程中，关闭管道的写结尾，并做好所有需要的准备。（在接下来的例子中将会看到这些准备工作。）
- 4) 在子进程中，执行该子进程的程序。
- 5) 在父进程中，关闭管道的读结尾。
- 6) 如果第二个子进程需要写管道，那就创建它，做好必要的准备，并执行其程序。如果父进程准备写，那么直接写就可以了。

本书中所有单向管道的例子都遵循这个范例。

现在，明白了为什么fork和exec是单独的系统调用了。为了节约fork的开销，为什么不用单个系统调用来做这两项工作呢？因为它们两个分开能允许我们执行上面的第3步。我们已经发现，在fork和exec之间需要做一些工作（关闭pfd[1]），而且随着本章的深入，需要做的工作会越来越多。在fork之前不能做这个工作，因为不希望它影响父进程（父进程不一定关闭了管道的写结尾）。也不希望子进程的程序做这个工作，因为我们希望程序不知道它们是怎样被调用的以及它们的输入与输出怎样连接的（这是UNIX哲学的基础）。所以一定要在非常恰当的地方处理它：在子进程中执行从父进程克隆来的代码。另一个额外的好处是连接代码被本地化了，使得调试和修改变得更容易。

继续这个话题：既然只有少量的连接进程的典型方法，那为什么不使用单独的带各种选项的fork-exec系统调用来进行进程间连接呢？毕竟存在其他带许多选项的系统调用，那为什么这里不行呢？原因仅仅是因为UNIX的初始设计者（Thompson和Ritchie）希望最小化系统调用的数目。fork和exec是简单的，而且它们允许调用者安排各种各样的定制连接。所以何必再组装另一个系统调用呢？^①

程序piperead是专门通过传入的文件符编号来读取文件描述符数据而设计的。该程序很奇怪——标准的UNIX命令没有那样做。许多程序确实能够读取某个特定的文件描述符，而且假定它们已经打开，但那个文件描述符被固定在0（标准输入，STDIN_FILENO）。而且并不必作为参数传递。类似的，许多程序被设计成写文件描述符1（标准输出，STDOUT_FILENO）。要像shell那样连接命令，某种程度上需要强迫pipe在pfd数组中返回特定的文件描述符：读结尾0和写结尾1。可惜，pipe没有提供那样的特性。为了使它们可用，可以在调用pipe前试着关闭0和1，但这样是不安全的，因为标准并没有介绍pipe将使用什么样的文件描述符，而且，进一步说，通常并不会仅为了产生一个管道而去牺牲标准输入与输出。那么怎样解决这个问题呢？可以使用dup或dup2。

6.3 dup和dup2系统调用

dup ——复制文件描述符

```
#include <unistd.h>

int dup(
    int fd          /* file descriptor to duplicate */
);
/* Returns new file descriptor or -1 on error (sets errno) */
```

dup2 ——复制文件描述符

```
#include <unistd.h>

int dup2(
    int fd,         /* file descriptor to duplicate */
    int fd2        /* file descriptor to use */
);
/* Returns new file descriptor or -1 on error (sets errno) */
```

dup可以复制一个现有的文件描述符，返回一个新的、仍指向该文件（或管道等）的文件描述符。这两个描述符共享同一个文件描述（见2.2节），就像一个继承的文件描述符与父进程中的相应文件描述符共享文件描述一样。如果参数是错的（没有打开）或没有文件描述

^① 但是现在有了它：posix_spawn，5.5节对其已经简单介绍过了。

符可用，那么该系统调用将失败。

`dup`取编号最小的可用文件描述符作参数，所以只要知道哪个是打开的，就可以控制它的返回值。但使用`dup2`更容易，它允许使用`fd2`参数指定文件描述符返回什么。为了使`fd2`可用，如果必须的话，`dup2`关闭它。

`dup(fd)`等价于

```
fcntl(fd, F_DUPFD, 0)
```

`dup2(fd, fd2)`等价于

```
close(fd2);
fcntl(fd, F_DUPFD, fd2);
```

只是`dup2`是原子的——如果复制有问题，那么不会关闭`fd2`。本书的所有例子中使用的都是`dup2`，而不是`dup`，因为它更容易。

因为文件描述是共享的，所以使用第二个文件描述符只有一个好处：它的编号不同，而且或许更适用于调用者的目的。假定创建了一个管道，然后使用`dup2`复制`STDIN_FILENO`（文件描述符0）来生成该管道读结尾的一个副本（它的文件描述符编号可能是3、4、27或者其他什么）。之后，如果执行（`exec`）一个用于读`STDIN_FILENO`（有许多`STDIN_FILENO!`）的程序，那么它可能会被骗去读管道。可以使用一个相似的算法强迫`STDOUT_FILENO`作为管道写结尾。

如果`dup2`的两个参数是相等的，那么它将只返回那个文件描述符而不会关闭或复制任何东西。例如，在管道读结尾不知何故已经等于`STDIN_FILENO`的情况下，这是有益的。

为了阐明`dup2`，这里举一个例子：生成管道，创建子进程读取它，将子进程的`STDIN_FILENO`作为该管道的读结尾，然后调用`cat`命令读取该管道。既然可以使用`STDIN_FILENO`，就不用像上一节中的例子那样，使用类似`piperead`的专门程序了。

```
void pipewrite2(void) /* has a bug */
{
    int pfd[2];
    pid_t pid;

    ec_neg1( pipe(pfd) )
    switch (pid = fork()) {
    case -1:
        EC_FAIL
    case 0: /* child */
        ec_neg1( dup2(pfd[0], STDIN_FILENO) )
        ec_neg1( close(pfd[0]) )
        ec_neg1( close(pfd[1]) )
        execlp("cat", "cat", (char *) NULL);
        EC_FAIL
    default: /* parent */
        ec_neg1( close(pfd[0]) )
        ec_neg1( write(pfd[1], "hello", 6) )
        ec_neg1( waitpid(pid, NULL, 0) )
    }

    return;
}

EC_CLEANUP_BGN
    EC_FLUSH("pipewrite2");
EC_CLEANUP_END
}
```



得到的输出并不令人吃惊:

```
hello
```

注意，在子进程中，复制了从pipe中得到的那两个文件描述符之一后，就关闭了它们。管道的读结尾仍是打开的，但现在其文件描述符是STDIN_FILENO。父进程关闭了pfd[0]（读结尾），因为其并不需要它。调用waitpid等待cat终止，在前一节的piperead例子中省略了它。

不幸的是，在显示了“hello”之后，程序便挂起了，并且直到用Ctrl-c终止它时，它都没有显示shell提示。能明白这是为什么吗？

因为只有当cat获得文件结尾后，才会停止读数据，但（在这个例子中）因为它并没有获得文件结尾，所以并不会终止，因此程序挂起。回顾可知，只有当所有的对管道打开的写文件描述符都是关闭的，read才会返回0。但在调用waitpid时，pfd[1]仍然是打开的，因此父进程与子进程被死锁。这个问题的解决方法是在写入数据之后关闭写结尾：

```
default: /* parent */
    ec_negl( close(pfd[0]) )
    ec_negl( write(pfd[1], "hello", 6) )
    ec_negl( close(pfd[1]) )
    ec_negl( waitpid(pid, NULL, 0) )
```

我们不必局限于从父进程到子进程的管道传输。接下来的例子实现了shell命令行的等价功能：

```
$ who | wc
```

来查看有多少用户登录了。

```
void who_wc(void)
{
    int pfd[2];
    pid_t pid1, pid2;

    ec_negl( pipe(pfd) )
    switch (pid1 = fork()) {
    case -1:
        EC_FAIL
    case 0: /* first child */
        ec_negl( dup2(pfd[1], STDOUT_FILENO) )
        ec_negl( close(pfd[0]) )
        ec_negl( close(pfd[1]) )
        execlp("who", "who", (char *) NULL);
        EC_FAIL
    }
    /* parent */
    switch (pid2 = fork()) {
    case -1:
        EC_FAIL
    case 0: /* second child */
        ec_negl( dup2(pfd[0], STDIN_FILENO) )
        ec_negl( close(pfd[0]) )
        ec_negl( close(pfd[1]) )
        execlp("wc", "wc", "-l", (char *) NULL);
        EC_FAIL
    }
    /* still the parent */
    ec_negl( close(pfd[0]) )
```



```

    ec_negl( close(pfd[1]) )
    ec_negl( waitpid(pid1, NULL, 0) )
    ec_negl( waitpid(pid2, NULL, 0) )
    return;

EC_CLEANUP_BGN
    EC_FLUSH("who_wc");
EC_CLEANUP_END
}

```

下面是输出:

1

取代who和wc都是同一个父进程的子进程的做法, 可以让wc成为who的子进程, 这样做正好容易创建:

```

void who_wc2(void)
{
    int pfd[2];
    pid_t pid1, pid2;

    ec_negl( pipe(pfd) )
    switch( pid1 = fork() ) {
    case -1:
        EC_FAIL
    case 0: /* child */
        switch( pid2 = fork() ) {
        case -1:
            EC_FAIL
        case 0: /* grandchild */
            ec_negl( dup2(pfd[0], STDIN_FILENO) )
            ec_negl( close(pfd[0]) )
            ec_negl( close(pfd[1]) )
            execlp("wc", "wc", "-l", (char *) NULL);
            EC_FAIL
        }
        /* still the child */
        ec_negl( dup2(pfd[1], STDOUT_FILENO) )
        ec_negl( close(pfd[0]) )
        ec_negl( close(pfd[1]) )
        execlp("who", "who", (char *) NULL);
        EC_FAIL
    }
    /* parent */
    ec_negl( close(pfd[0]) )
    ec_negl( close(pfd[1]) )
    ec_negl( waitpid(pid1, NULL, 0) )
    return;

EC_CLEANUP_BGN
    EC_FLUSH("who_wc2");
EC_CLEANUP_END
}

```

关于这个例子的两点解释:

- 在关闭读文件描述符之前, 子进程必须创建孙进程, 以便孙进程可以继承它。
- 父进程只等待who(pid1)来终止。它不能等待wc, 因为wc不是它的子进程。运行who的子进程也不能等待wc, 因为who程序没有设计用来那样做。在exec wc调用之后放置一

一个waitpid调用是没有作用的，因为当成功执行exec时会覆盖所有代码。所要发生的（如5.8节解释的）是当who（wc的父进程）终止时，系统进程将继承wc，并且等待它。可以很容易地颠倒这个过程，即让wc成为who的父进程（见练习6.11）。当然，像这样使用shell建立命令行比编写定制程序更加容易。下一节中将编写这样的shell。

6.4 一个真正的shell

本节编写的shell是你可能使用过的典型UNIX shell的子集。它有这些特点：

- 简单命令包含命名，其后跟有用空格或tab分隔开的可选参数序列，每个参数都是一个用双引号（"）括起来的单个词或字符串。如果被引号引起来，这个参数可以包含任何其他特殊字符（|、;、&、>、<、空格、tab和新行字符）。一个被包含的引号或反斜线的前面必须带有反斜线（\"或\\）。每个命令至多可以有50个参数，每一个参数至多可以有500个字符。
- 通过在文件名前放置<，可以把一个简单命令的标准输入重定向到它来自的文件。同样，通过>可以重定向标准输出，这时输出截短了输出文件。如果输出重定向的符号是>>，那么输出会被追加到输出文件上。如果输出文件不存在，则会创建它。
- 管道由用“|”分割的一个或多个简单命令序列构成。除了最后一个简单命令外，管道中的每个命令都有自己的标准输出，并通过管道连接到右侧邻居的标准输入。
- 管道通过新行字符、分号（;）或者and号（&）而终止，对于前两种符号，shell继续之前先等待最右边的简单命令终止。它报告管道中每个简单命令的进程数，每个简单命令运行时忽略中断和退出信号。
- 内置命令有赋值、set和cd。5.4节讲述了前两个命令，cd以熟悉的方式运行。

第一步是将输入行分解为标记，这些标记是形成语法单元的符号组；例如是单字、引用字符、和像&和>>等一样的特殊符号。每个标记由如下的一个符号常量表示：

T_WORD 参数或文件名。如果被引号引起来了，在标记被识别之后，引号便被清除。

T_BAR 符号|。

T_AMP 符号&。

T_SEMI 符号;。

T_GT 符号>。

T_GTGT 符号>>。

T_LT 符号<。

T_NL 新行。

T_EOF 专门标识已经到达了文件结尾的标记。如果标准输出是一个终端，那么用户已经输入了EOT（Ctrl-d）。

T_ERROR 标识错误的特殊标记。

词汇分析器（lexical analyzer）的任务是读输入并将字符装配成标记。每一次调用它，都会返回一个标记。如果返回的标记是T_WORD，那么也将返回包含构成其实际字符的字符串。（对于其他的标记而言，实际字符是明显的。）词汇分析器应该绕过那些不相关的字符（如分隔参数的空格）不返回任何内容。[⊖]

⊖ 大多数UNIX系统都包括一个叫做lex的命令，该命令能自动地从将要识别标记的描述中产生一个词汇分析器。因为lex使用起来很复杂，而且其产生的词汇分析器有时大而且慢，通常情况下最好手动编写词汇分析器。程序写起来并不难。

这里的词汇分析器是一个有限状态机：当字符被读取时，它们要么被立刻识别为标记，要么被积聚起来（例如，一个词的字符）。对每个字符，词汇分析器都会转换到一个新状态，该状态下能够记录它正在做什么和怎样解释字符。例如，当积聚引号里字符串时，此时对待空格的方式与引号外的空格有所不同。本节的shell，需要四种状态：

NEUTRAL 每个调用词汇分析器的初始状态。忽略空格和tab。字符|、&、;、<和换行字符会被立刻识别成标记。符号>可以把状态转换为GTGT，这也要看它后面是否还有一个>符号，因为>和>>是两个不同的标记。引号可以把状态转换成聚集引号内字符串的INQUOTE。其他所有的字符都被当作非引用词的开始；字符被保存在缓冲区中，并且状态转换为INWORD。

GTGT 这个状态表示刚刚读取了>。如果下一个字符也是>，那么将返回T_GTGT标记。否则，返回T_GT。但首先是已经读取了太多次同一字符，所以用标准C函数ungetc将它放回到输入中。

INQUOTE 这个状态表示读取了一个左引号。累积字符到一个缓冲区直到读取到右引号。然后返回T_WORD标记和累积字符串，必须采取专门的操作处理退出字符\。

INWORD 这个状态表示读取了字的第一个字符，并且已经将它放到了缓冲区中。不断地累积字符，直到读取到一个非字的字符（比如，|）。不用处理的字符放回到输出，并返回T_WORD标记。

通过这个解释，应该可以理解词汇分析器——gettoken的代码了：

```
typedef enum (T_WORD, T_BAR, T_AMP, T_SEMI, T_GT, T_GTGT, T_LT,
             T_NL, T_EOF, T_ERROR) TOKEN;

static TOKEN gettoken(char *word, size_t maxword)
{
    enum {NEUTRAL, GTGT, INQUOTE, INWORD} state = NEUTRAL;
    int c;
    size_t wordn = 0;

    while ((c = getchar()) != EOF) {
        switch (state) {
            case NEUTRAL:
                switch (c) {
                    case ';':
                        return T_SEMI;
                    case '&':
                        return T_AMP;
                    case '|':
                        return T_BAR;
                    case '<':
                        return T_LT;
                    case '\n':
                        return T_NL;
                    case ' ':
                    case '\t':
                        continue;
                    case '>':
                        state = GTGT;
                        continue;
                    case '"':
                        state = INQUOTE;
                        continue;
                    default:
                        state = INWORD;
                        ec_false( store_char(word, maxword, c, &wordn) )
                }
            }
        }
    }
}
```



```

        continue;
    }
}
case GTGT:
    if (c == '>')
        return T_GTGT;
    ungetc(c, stdin);
    return T_GT;
case INQUOTE:
    switch (c) {
        case '\\':
            if ((c = getchar()) == EOF)
                c = '\\';
            ec_false( store_char(word, maxword, c, &wordn) );
            continue;
        case "'":
            ec_false( store_char(word, maxword, '\\0', &wordn) );
            return T_WORD;
        default:
            ec_false( store_char(word, maxword, c, &wordn) );
            continue;
    }
}
case INWORD:
    switch (c) {
        case ';':
        case '&':
        case '|':
        case '<':
        case '>':
        case '\n':
        case ' ':
        case '\t':
            ungetc(c, stdin);
            ec_false( store_char(word, maxword, '\\0', &wordn) );
            return T_WORD;
        default:
            ec_false( store_char(word, maxword, c, &wordn) );
            continue;
    }
}
}
ec_false( !ferror(stdin) )
return T_EOF;

EC_CLEANUP_BGN
return T_ERROR;
EC_CLEANUP_END
}

static bool store_char(char *word, size_t maxword, int c, size_t *np)
{
    errno = E2BIG;
    ec_false( *np < maxword )
    word[(*np)++] = c;
    return true;
}

EC_CLEANUP_BGN
return false;
EC_CLEANUP_END
}

```

当编写一个大程序时，分片进行调试是很方便的。这不仅能早一点儿提供反馈，而且也

容易寻找漏洞，因为只需要查找少量几行代码。下面是个gettoken的测试程序：

```
int main(void)
{
    char word[200];

    while (1)
        switch (gettoken(word, sizeof(word))) {
            case T_WORD:
                printf("T_WORD <%s>\n", word);
                break;
            case T_BAR:
                printf("T_BAR\n");
                break;
            case T_AMP:
                printf("T_AMP\n");
                break;
            case T_SEMI:
                printf("T_SEMI\n");
                break;
            case T_GT:
                printf("T_GT\n");
                break;
            case T_GTGT:
                printf("T_GTGT\n");
                break;
            case T_LT:
                printf("T_LT\n");
                break;
            case T_NL:
                printf("T_NL\n");
                break;
            case T_EOF:
                printf("T_EOF\n");
                exit(EXIT_SUCCESS);
            case T_ERROR:
                printf("T_ERROR\n");
                exit(EXIT_SUCCESS);
        }
}
```

当运行这个程序，并输入下面这行代码时（后面跟有EOT）：

```
sort <inf | pr -h "Sept. Results" >>outf&
```

可以得到如下输出：

```
T_WORD <sort>
T_LT
T_WORD <inf>
T_BAR
T_WORD <pr>
T_WORD <-h>
T_WORD <Sept. Results>
T_GTGT
T_WORD <outf>
T_AMP
T_NL
T_EOF
```

这个测试不能证明gettoken是正确的，但对继续这个shell而言是足够让人鼓舞的。



下一步是编写一个用于处理简单命令的函数，该命令是以|、&、;或换行字符来终止的。这里称这个函数为command。简单地将参数放入argv数组以便以后调用execvp使用。标准输入有三种可能：默认（STDIN_FILENO）、文件（如果<存在）或管道的读结尾（如果简单命令之前有|）。标准输出有四种可能：默认（STDOUT_FILENO）、被创建的或被截短的文件（如果>存在）、被创建的或被追加的文件（如果>>存在），或管道的写结尾（如果简单命令之后跟有|）。从gettoken接收的标记会通知我们获得的是哪种情况。

当command处理这些标记时，使用这些变量来记录简单命令的标准输入与标准输出：

srcfd 源文件描述符，初始时为STDIN_FILENO。如果用<给输入重定向，那么srcfd将被设置为-1，并且srcfile会记录该文件名。如果输入是管道，那么srcfd会被设置成文件描述符编号而不是STDIN_FILENO。

srcfile 源文件，只有当简单命令包含有<时有用。

dstfd 目的文件描述符，初始时为STDOUT_FILENO。如果用>或>>给输出重定向，那么该变量将被设置为-1且由dstfile记录该文件名。和srcfd一样，如果输出是管道，那么它将被设置为文件描述符而不是STDOUT_FILENO。

dstfile 输出文件，仅当简单命令包含>或>>时有用。

append 仅当用>>重定向输出时这个布尔变量才会被设置为true。

makepipe 这是command的一个参数。如果为true，则调用者请求command生成一个管道，使用该结尾作为标准输入，并把该文件描述符作为写结尾递归给调用者，调用者用它作为标准输出。稍后将解释这个方案。

command仔细检查标记时，也检查逻辑错误：<或>的两次出现；当用|终止简单命令时，出现>；当简单命令前面有|时，出现<，等等。有趣的是，典型的UNIX shell并不去检查这些异常，实际上可以运行如下的命令：

```
$ who >outf | wc
```

函数command会返回终止管道的标记，因为终止符的类型影响后续的处理：如果终止符是&，那么就不用等待最右边的简单命令，且管道中的所有简单命令运行时都会忽略中断信号和退出信号。如果要等待最右边的简单命令，那么也必须通过wpid参数返回其进程ID。如果终止符是一个新行字符，则是新命令的提示。

关于command最微妙的地方是，当简单命令跟有|时，它会递归调用它自己。直到碰到其他的终止符（;、&或新行字符）时，递归调用才会停止。每个递归调用实际上都负责生成管道（用pipe系统调用），因此其参数makepipe都被设置成了true。写管道文件描述符会被传递回来（通过另一个参数，pipefdp）。直到递归调用返回后，才实际调用简单命令（用函数invoke），因为直到管道的写结尾可用时，才能调用它。同样，和上面的情况一样，在调用任意一个组成简单命令的要素之前，必须知道管道终止符，以便使invoke知道如何处理信号。因此从左向右处理管道而从右向左调用简单命令。把makepipe设置为false，调用command处理最左边的简单命令，然后，把makepipe设置为true，递归调用command处理其他的简单命令。当命令调用栈返回时，就调用简单命令——此时所有需要的信息就都可用了。

下面是command的代码，该代码值得仔细研究。

```
#define MAXARG 50      /* max args in command */
#define MAXFNAME 500  /* max chars in file name */
#define MAXWORD 500   /* max chars in arg */

static TOKEN command(pid_t *wpid, bool makepipe, int *pipefdp)
{
```

```
TOKEN token, term;
int argc, srcfd, dstfd, pid, pfd[2] = {-1, -1};
char *argv[MAXARG], srcfile[MAXFNAME] = "", dstfile[MAXFNAME] = "";
char word[MAXWORD];
bool append;

argc = 0;
srcfd = STDIN_FILENO;
dstfd = STDOUT_FILENO;
while (true) {
    switch (token = gettoken(word, sizeof(word))) {
        case T_WORD:
            if (argc >= MAXARG - 1) {
                fprintf(stderr, "Too many args\n");
                continue;
            }
            if ((argv[argc] = malloc(strlen(word) + 1)) == NULL) {
                fprintf(stderr, "Out of arg memory\n");
                continue;
            }
            strcpy(argv[argc], word);
            argc++;
            continue;

        case T_LT:
            if (makepipe) {
                fprintf(stderr, "Extra <\n");
                break;
            }
            if (gettoken(srcfile, sizeof(srcfile)) != T_WORD) {
                fprintf(stderr, "Illegal <\n");
                break;
            }
            srcfd = -1;
            continue;

        case T_GT:
        case T_GTGT:
            if (dstfd != STDOUT_FILENO) {
                fprintf(stderr, "Extra > or >>\n");
                break;
            }
            if (gettoken(dstfile, sizeof(dstfile)) != T_WORD) {
                fprintf(stderr, "Illegal > or >>\n");
                break;
            }
            dstfd = -1;
            append = token == T_GTGT;
            continue;

        case T_BAR:
        case T_AMP:
        case T_SEMI:
        case T_NL:
            argv[argc] = NULL;
            if (token == T_BAR) {
                if (dstfd != STDOUT_FILENO) {
                    fprintf(stderr, "> or >> conflicts with |\n");
                    break;
                }
            }
            term = command(wpid, true, &dstfd);
            if (term == T_ERROR)
                return T_ERROR;
    }
}
```

```

    }
    else
        term = token;
    if (makepipe) {
        ec_negl( pipe(pfd) )
        *pipefdp = pfd[1];
        srcfd = pfd[0];
    }
    ec_negl( pid = invoke(argc, argv, srcfd, srcfile, dstfd,
        dstfile, append, term == T_AMP, pfd[1]) )
    if (token != T_BAR)
        *wpid = pid;

    if (argc == 0 && (token != T_NL || srcfd > 1))
        fprintf(stderr, "Missing command\n");
    while (--argc >= 0)
        free(argv[argc]);
    return term;
case T_EOF:
    exit(EXIT_SUCCESS);
case T_ERROR:
    return T_ERROR;
}
}

EC_CLEANUP_BGN
    return T_ERROR;
EC_CLEANUP_END
}

```

当command遇到T_BAR标记时，就已经意识到了需求管道，为什么要再次调用command生成管道，而不是自己生成管道并仅传递读文件描述符呢？这是为了保存文件描述符。如果在递归调用command之前而不是之后调用pipe系统调用，那么递归调用的每层（即每个简单命令）都将捆绑两个管道文件描述符，直到递归调用返回后才关闭。在某些系统上，因为文件描述符是短缺的，所以管道的数目被限制在比简单命令数目的一半还少一些。如果调用者已经要求生成一个管道，那么通过要求读进程生成管道，可以恰在调用invoke（马上就会讲到）之前调用pipe，这样就允许管道是任何长度的。

invoke创建的子进程不需要管道的写结尾（pfd[1]），因此它被作为最后一个参数传递给invoke，所以在子进程中可以关闭它；稍后会看到这点。

下面是调用第一个command的main程序。它模拟的是5.4节中独一无二的shell的main程序。

```

int main(void)
{
    pid_t pid;
    TOKEN term = T_NL;

    ignore_sig();
    while (true) {
        if (term == T_NL)
            printf("%s", PROMPT);
        term = command(&pid, false, NULL);

        if (term == T_ERROR) {
            fprintf(stderr, "Bad command\n");
            EC_FLUSH("main--bad command")
            term = T_NL;
        }
        if (term != T_AMP && pid > 0)

```

```
        wait_and_display(pid);
        fd_check();
    }
}
```

调用`ignore_sig`会使中断信号和退出信号被忽略——当碰到中断或退出键时，并不是想要删除shell。将在9.1.6节中给出`ignore_sig`的代码。由`command`返回的管道终止符合告知是否需要等待最右边的简单命令终止（稍后会看到这个版本的`wait_and_display`）和是否需要提示。

因为需要确认使用重定向和管道之后，没有忘记关闭任何文件描述符，所以每次处理命令时都要调用`fd_check`以确认只有标准文件描述符是打开的。（如果需要，也可以从这个shell的产品版本中移去`fd_check`。）只需要核查前20个文件描述符，因为这就足以揭示关闭失败的漏洞了：

```
static void fd_check(void)
{
    int fd;
    bool ok = true;

    for (fd = 3; fd < 20; fd++)
        if (fcntl(fd, F_GETFL) != -1 || errno != EBADF) {
            ok = false;
            fprintf(stderr, "**** fd %d is open ****\n", fd);
        }
    if (!ok)
        _exit(EXIT_FAILURE);
}
```

`command`调用`invoke`来调用简单命令。它传递命令参数（`argc`和`argv`）和先前描述的源与目的变量（`srcfd`、`srcfile`、`dstfd`、`dstfile`和`append`）。倒数第二个参数会告诉`invoke`简单命令是否准备在后台运行。如前所述，最后一个参数是将在子进程中关闭的文件描述符。现在应该熟悉`invoke`使用的`fork`和`exec`了：

```
static pid_t invoke(int argc, char *argv[], int srcfd, const char *srcfile,
                   int dstfd, const char *dstfile, bool append, bool bckgrnd, int closefd)
{
    pid_t pid;
    char *cmdname, *cmdpath;

    if (argc == 0 || builtin(argc, argv, srcfd, dstfd))
        return 0;
    switch (pid = fork()) {
    case -1:
        fprintf(stderr, "Can't create new process\n");
        return 0;
    case 0:
        if (closefd != -1)
            ec_neg1(close(closefd));
        if (!bckgrnd)
            ec_false(entry_sig());
        redirect(srcfd, srcfile, dstfd, dstfile, append, bckgrnd);
        cmdname = strchr(argv[0], '/');
        if (cmdname == NULL)
            cmdname = argv[0];
        else
            cmdname++;
        cmdpath = argv[0];
    }
```



```

    argv[0] = cmdname;
    execvp(cmdpath, argv);
    fprintf(stderr, "Can't execute %s\n", cmdpath);
    _exit(EXIT_FAILURE);
}
/* parent */
if (srcfd > STDOUT_FILENO)
    ec_negl( close(srcfd) )
if (dstfd > STDOUT_FILENO)
    ec_negl( close(dstfd) )
if (bckgrnd)
    printf("%ld\n", (long)pid);
return pid;

EC_CLEANUP_BGN
    if (pid == 0)
        _exit(EXIT_FAILURE);
    return -1;
EC_CLEANUP_END
}

```

因为既需要保持完整的路径（这个路径可能已经被作为execvp的第一个参数输入），但是又需要argv[0]仅指向文件名那部分，所以造成了cmdname和cmdpath之间的混乱。

可以内置被调用的命令。如果这样的话，builtin（稍后给出）会返回true。如果不是这样，可以创建一个子进程运行简单命令。回顾以前的内容，可以知道已经忽略了中断信号和退出信号。如果该命令不在后台运行，那么可以使用entry_sig调用（9.1.6节详细讨论）把信号恢复到它们传入shell的地方。

invoke调用redirect来重定向I/O，并且如果需要，也可以用来确保源和目的被复制成STDIN_FILENO和STDOUT_FILENO。下面是redirect的代码：

```

static void redirect(int srcfd, const char *srcfile, int dstfd,
                    const char *dstfile, bool append, bool bckgrnd)
{
    int flags;

    if (srcfd == STDIN_FILENO && bckgrnd) {
        srcfile = "/dev/null";
        srcfd = -1;
    }
    if (srcfile[0] != '\0')
        ec_negl( srcfd = open(srcfile, O_RDONLY, 0) )
    ec_negl( dup2(srcfd, STDIN_FILENO) )
    if (srcfd != STDIN_FILENO)
        ec_negl( close(srcfd) )
    if (dstfile[0] != '\0') {
        flags = O_WRONLY | O_CREAT;
        if (append)
            flags |= O_APPEND;
        else
            flags |= O_TRUNC;
        ec_negl( dstfd = open(dstfile, flags, PERM_FILE) )
    }
    ec_negl( dup2(dstfd, STDOUT_FILENO) )
    if (dstfd != STDOUT_FILENO)
        ec_negl( close(dstfd) )
    fd_check();
    return;
}

```



```
EC_CLEANUP_BGN
    _exit(EXIT_FAILURE); /* we are in child */
EC_CLEANUP_END
}
```

如果后台命令没有把标准输入重定向为来自的文件或管道，那么可以采取把标准输入重定向到一个特殊文件/dev/null，当进行读操作时，这个文件会给出当前的文件结尾。redirect的剩余部分应该被清除。

wait_and_display (从main调用的)是5.8节中给出的代码的扩展。它被告知该等待什么进程，但在等待期间，它可能会了解进程。如果这样，它会用display_status输出那些进程ID和终止原因的描述。当被委托的进程终止时，wait_and_display将在显示其状态后返回。下面是它的代码：

```
static bool wait_and_display(pid_t pid)
{
    pid_t wpid;
    int status;

    do {
        ec_negl( wpid = waitpid(-1, &status, 0) )
        display_status(wpid, status);
    } while (wpid != pid);
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

最后给出builtin的代码。其大部分内容来自5.4节，这里添加了处理cd命令的代码。不带参数的cd命令改变了HOME环境变量所给路径的目录。

```
static bool builtin(int argc, char *argv[], int srcfd, int dstfd)
{
    char *path;

    if (strchr(argv[0], '=') != NULL)
        asg(argc, argv);
    else if (strcmp(argv[0], "set") == 0)
        set(argc, argv);
    else if (strcmp(argv[0], "cd") == 0) {
        if (argc > 1)
            path = argv[1];
        else if ((path = getenv("HOME")) == NULL)
            path = ".";
        if (chdir(path) == -1)
            fprintf(stderr, "%s: bad directory\n", path);
    }
    else
        return false;
    if (srcfd != STDIN_FILENO || dstfd != STDOUT_FILENO)
        fprintf(stderr, "Illegal redirection or pipeline\n");
    return true;
}
```

你可能已经观察到了，这里shell发现的一些错误被强大的“ec”工具处理了，而其他一些错误只产生了输出消息，之后shell继续运行。这里把“不可能”的错误看作是严重的错误，因为如果它们发生，将意味着操作系统受到了致命伤害。这是一个折衷方法：我们当然不希望忽略这些错误，因为不可能的事情也是会发生的，但我们并不打算编写代码从某个可能从不会发生的情况中恢复。当然，有时会猜测失误——严重错误持续发生，因为毕竟它不是不可能发生的。那么就必须修改代码以处理不同的错误。

这里没有给出这个shell的任何示例。因为该shell的性能与我们平时使用的shell是相同的，所以没必要给出示例。

6.5 非重定向管道的双向通信

现在远离shell中使用的单向管道通信的讨论，转到双向通信。典型的shell没有提供在进程间建立双向通信的符号。双向管道是从C程序中建立的。

首先从一个十分简单的例子讲起。在一个程序中，希望能够通过调用sort命令来实现数据排序。当然可以采用如下的代码：

```
system("sort <datafile >outfile");
```

然后，可以读取输出文件的内容以访问排好的数据。但这里并不想用那样的方式处理，而是想要用管道把数据传递给sort，并让sort把排序后的数据通过管道传回来。因为sort可以读写其标准输入与标准输出（它是一个过滤程序），所以应该可以按我们的想法使用它。现在已经知道如何强迫任意的文件描述符成为进程的标准输入或标准输出了。

如婴儿在学习走路之前首先要学会跌倒一样，先从不正确的做法开始。这与与立刻给出正确的解决方法相比，可以学到更多的关于如何恰当处理它的方法。

因为每个管道都有一个读结尾和一个写结尾，又因为两个文件描述符都被一个子进程继承了，所以可以只使用一个管道。sort可以读取管道获得输入并向其写回，以便传回排序后的输出。父进程也拥有访问管道的文件描述符。它向管道中写入没有排序的数据，并从管道中读取排序后的数据。下面是一个从文件datafile中读取数据的程序，它调用sort对读取的数据进行排序；然后输出排序后的数据：

```
void fsort0(void) /* wrong */
{
    int pfd[2], fd;
    ssize_t nread;
    pid_t pid;
    char buf[512];

    ec_negl( pipe(pfd) )
    ec_negl( pid = fork() )
    if (pid == 0) { /* child */
        ec_negl( dup2(pfd[0], STDIN_FILENO) )
        ec_negl( close(pfd[0]) )
        ec_negl( dup2(pfd[1], STDOUT_FILENO) )
        ec_negl( close(pfd[1]) )
        execlp("sort", "sort", (char *) NULL);
        EC_FAIL
    }
    /* parent */
    ec_negl( fd = open("datafile", O_RDONLY) )
    while (true) {
        ec_negl( nread = read(fd, buf, sizeof(buf)) )
    }
}
```



```

        if (nread == 0)
            break;
        ec_negl( write(pfd[1], buf, nread) )
    }
    ec_negl( close(fd) )
    ec_negl( close(pfd[1]) )
    while (true) {
        ec_negl( nread = read(pfd[0], buf, sizeof(buf)) )
        if (nread == 0)
            break;
        ec_negl( write(STDOUT_FILENO, buf, nread) )
    }
    ec_negl( close(pfd[0]) )
    ec_negl( waitpid(pid, NULL, 0) )
    return;

EC_CLEANUP_BGN
    EC_FLUSH("fsort0");
EC_CLEANUP_END
}

```

下面是datafile中的内容:

```

peach
apple
orange
strawberry
plum
pear
cherry
banana
apricot
tomato
pineapple
mango

```

当运行这个程序时，其输出和datafile中一样——水果没有排序，然后挂起。必须键入中断键来终止它。哪里出错了呢？

仅使用一个管道会带来两个问题。第一个问题，在向管道中写入未排序数据后，父进程会假定应该能读取到sort的输出，就立刻开始读取管道。但若在sort完成之前，它就开始读，便只读取了原来自己的输出而已！所以输出的是未经排序的数据。这和6.2.3节中的第一个例子相似。

第二个问题是会引起死锁。正在运行sort的子进程开始读取其标准输入，而碰巧可能是空的，因为它的父进程已经清空了它。无论是否为空，在等待文件结尾的read系统调用中sort都可能会阻塞，这种情况仅当写结尾关闭时才会发生。可以肯定的是，父进程已经关闭了写结尾，但子进程仍然使它处于打开状态——毕竟，子进程应当在那里写它的输出。所以子进程被阻碍了。通常，读取和写入相同管道的任何过滤程序都会死锁。

有人可能会试图通过使父进程在读取管道之前等待子进程终止来解决第一个问题。首先，这听起来很不错。但并不起作用，因为如果子进程的输出填满了管道（对大量数据进行排序时这是可能的），那么子进程将阻塞write系统调用。当父进程阻塞waitpid时，也会再次死锁。

又有人可能尝试用一些信号量来同步事件，而避免死锁，但这样比较复杂。如果使用两个管道，每个仅处理控制单向流量，问题便完全解决了。一个管道处理流入sort的数据，而

另一个管道处理数据流出。下面是重写的能正确工作的fsort0的代码:

```

void fsort(void)
{
    int pfdout[2], pfdin[2], fd;
    ssize_t nread;
    pid_t pid;
    char buf[512];

    ec_negl( pipe(pfdout) )
    ec_negl( pipe(pfdin) )
    ec_negl( pid = fork() )
    if (pid == 0) { /* child */
        ec_negl( dup2(pfdout[0], STDIN_FILENO) )
        ec_negl( close(pfdout[0]) )
        ec_negl( close(pfdout[1]) )
        ec_negl( dup2(pfdin[1], STDOUT_FILENO) )
        ec_negl( close(pfdin[0]) )
        ec_negl( close(pfdin[1]) )
        execlp("sort", "sort", (char *) NULL);
        EC_FAIL
    }
    /* parent */
    ec_negl( close(pfdout[0]) )
    ec_negl( close(pfdin[1]) )
    ec_negl( fd = open("datafile", O_RDONLY) )
    while (true) {
        ec_negl( nread = read(fd, buf, sizeof(buf)) )
        if (nread == 0)
            break;
        ec_negl( write(pfdout[1], buf, nread) )
    }
    ec_negl( close(fd) )
    ec_negl( close(pfdout[1]) )
    while (true) {
        ec_negl( nread = read(pfdin[0], buf, sizeof(buf)) )
        if (nread == 0)
            break;
        ec_negl( write(STDOUT_FILENO, buf, nread) )
    }
    ec_negl( close(pfdin[0]) )
    ec_negl( waitpid(pid, NULL, 0) )
    return;

    EC_CLEANUP_BGN
    EC_FLUSH("fsort");
    EC_CLEANUP_END
}

```

结果如下:

```

apple
apricot
banana
cherry
mango
orange
peach
pear
pineapple
plum

```



```
strawberry
tomato
```

不仅排序了列表，而且该程序本身也停止了所有的运行！关于正确版本有意思的是：尽管多使用了一个管道，并没有多消耗文件描述符。在两个版本中，父进程和子进程都需要向彼此读和写，所以每个都需要两个管道文件描述符。在第二个版本中，父进程能够关闭 `pf dout` 的读结尾和 `pf din` 的写结尾。

通常，使用两个管道仍可能产生死锁，尽管这里使用 `sort` 的例子中没有出现。如果输出管道满了，而子进程没有清空它，而是写回足够的输出给父进程以至于阻塞另一管道时，父进程会阻塞写输出管道，这时就发生了死锁。必须仔细检查每一种情况，以确保绝对不会产生死锁，不要只使用少量的检测数据来检测。

还存在其他的产生死锁的方式。为了探究其中的一种，下面来看一个更复杂的进程间双向通信的例子。这里子进程是标准行编辑器——`ed`。父进程把编辑器作为服务器向它发送编辑命令行，并得到输出。可视化编辑器可能会采用这样的安排，由父进程控制键盘和屏幕，而让 `ed` 做具体的编辑工作。这里没有空间显示一个可视化的编辑器，所以下面的示例是非常简单的。它是个交互搜索程序，很像 `grep`。下面是会话示例（输入的用下划线表示）：

```
$ search
File? datafile
Search pattern? _a
apple
apricot
Search pattern? apple
apple
pineapple
Search pattern? o$
tomato
mango
Search pattern? EOT
$
```

水果数据文件来自前面的排序例子。

当使用 `sort` 时，我知道什么时候停止读取其输出：即当到达文件结尾时。这个情况是简单的，因为 `sort` 只是读取所有输入，写入所有输出，然后终止。然而，编辑器是交互的。它读取某些输入，可能会写一些不确定长度的输出，也可能不写，然后返回来读取更多的输入。可以通过将其标准输出生成管道来捕获其输出，但怎样才能知道一次要读取多少数据呢？不能等待文件结尾，因为编辑器直到终止时才会关闭其输出文件描述符。如果读取的数据太多，会产生死锁，而如果读取的不够，将失去命令和它的结果间的同步。

如果能够修改编辑器，使得每当编辑器准备读取更多输入时，都可以发送一个明确的数据行。事实上，编辑器确实有能力提示用户输入（通过 `P` 命令实现），但提示符是 `*`，很不明确。（尽管可以改变提示符；见练习 6.12。）无论如何，都需要使编辑器告知完成输出每个命令结果的时间。

有一个技巧可以使用：在每个命令之后，带有一个不存在的文件名调用 `r`（读文件）命令，并从 `ed` 查找出现错误的消息。当这个消息出现时，就能知道编辑器响应了错误的 `r` 命令，并准备接受另一个命令。不存在的文件名应该是那样的，以至于错误消息是确定的。最好使用

包含很少在文本文件中出现的控制符构成的名字，但为了清晰，这里使用“end-of-file”作为名字。为了查看消息的确切形式，运行编辑器：

```
$ ed
r_end-of-file
?end-of-file
q
$
```

所以，必须寻找“?end-of-file”。

为了使事情更容易，可以编写函数来处理 and 编辑器的交互。在处理开始部分，使用 `edinvoke` 调用编辑器并建立管道。不直接使用文件描述符，因为这样要求用 `read` 和 `write` 处理 I/O，而这里要使用 `fdopen` 创建标准 C 的 `FILE` 指针。然后，可以用 `sndfp` 向编辑器写入，用 `rcvfp` 从编辑器中读取。下面是 `edinvoke`，很像 `fsort` 的第一部分：

```
static FILE *sndfp, *rcvfp;

static bool edinvoke(void)
{
    int pfdout[2], pfdin[2];
    pid_t pid;

    ec_negl( pipe(pfdout) )
    ec_negl( pipe(pfdin) )
    switch (pid = fork()) {
    case -1:
        EC_FAIL
    case 0:
        ec_negl( dup2(pfdout[0], STDIN_FILENO) )
        ec_negl( dup2(pfdin[1], STDOUT_FILENO) )
        ec_negl( close(pfdout[0]) )
        ec_negl( close(pfdout[1]) )
        ec_negl( close(pfdin[0]) )
        ec_negl( close(pfdin[1]) )
        execlp("ed", "ed", "-.", (char *) NULL);
        EC_FAIL
    }
    ec_negl( close(pfdout[0]) )
    ec_negl( close(pfdin[1]) )
    ec_null( sndfp = fdopen(pfdout[1], "w") )
    ec_null( rcvfp = fdopen(pfdin[0], "r") )
    return true;
}

EC_CLEANUP_BGN
if (pid == 0) {
    EC_FLUSH("edinvoke");
    _exit(EXIT_FAILURE);
}
return false;
EC_CLEANUP_END
}
```

注意当遇到错误时，子进程调用的是 `_exit`，而不是 `exit`，但接着必须要调用 `EC_FLUSH` 来显示错误消息，如 5.6 节中所做的那样。

为了读写管道，这里使用 `edsnd`、`edrcv` 和 `turnaround`。当没有更多的编辑器输出可用时，`edrcv` 将返回 `false`，之所以知道这个事实是因为它已经发现了强制性的错误消息。（它把从 `fgets` 返回的、用于指示实际文件结尾的 `NULL` 看成是一个错误。）为了确认错误消息

在那里，必须调用

```
static bool edsnd(const char *s)
{
    ec_eof( fputs(s, sndfp) )
    return true;

    EC_CLEANUP_BGN
    return false;
    EC_CLEANUP_END
}

static bool edrcv(char *s, size_t smax)
{
    ec_null( fgets(s, smax, rcvfp) )
    return true;

    EC_CLEANUP_BGN
    return false;
    EC_CLEANUP_END
}

static bool turnaround(void)
{
    ec_false( edsnd("r end-of-file\n") )
    ec_eof( fflush(sndfp) )
    return true;

    EC_CLEANUP_BGN
    return false;
    EC_CLEANUP_END
}
```

这里刷新了sndfp，以便编辑器能够获得所有的输出，因为一般输出到管道的数据都会被放到缓冲区中。

因为通常需要显示编辑器所做的所有操作，所以实现它的函数应该垂手可得：

```
static bool rcvall(void)
{
    char s[200];

    ec_false( turnaround() )
    while (true) {
        ec_false( edrcv(s, sizeof(s)) )
        if (strcmp(s, "?end-of-file\n") == 0)
            break;
        printf("%s", s);
    }
    return true;

    EC_CLEANUP_BGN
    return false;
    EC_CLEANUP_END
}
```

最后，main程序管理与用户的对话与与编辑器的通信：

```
int main(void)
{
    char s[100], line[200];
    bool eof;
```



```

ec_false( prompt("File", s, sizeof(s), &eof) )
if (eof)
    exit(EXIT_SUCCESS);
ec_false( edinvoke() )
snprintf(line, sizeof(line), "e %s\n", s);
ec_false( edsnd(line) )
ec_false( rcvall() );
while (true) {
    ec_false( prompt("Search pattern", s, sizeof(s), &eof) )
    if (eof)
        break;
    snprintf(line, sizeof(line), "g/%s/p\n", s);
    ec_false( edsnd(line) )
    ec_false( rcvall() );
}
ec_false( edsnd("q\n") )
exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}

static bool prompt(const char *msg, char *result, size_t resultmax,
    bool *eofp)
{
    char *p;

    printf("\n%s? ", msg);
    if (fgets(result, resultmax, stdin) == NULL) {
        if (ferror(stdin))
            EC_FAIL
            *eofp = true;
    }
    else {
        if ((p = strrchr(result, '\n')) != NULL)
            *p = '\0';
        *eofp = false;
    }
    return true;
}

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

```

这个程序能避免死锁吗？因为已经知道管道的容量至少是512字节，又因为这里编辑器的命令都非常短，所以当向编辑器的输入管道写入时，父进程绝对不会阻塞。编辑器决不会阻塞，除了偶尔向父进程写回外，因为父进程会读取所有必须输出的东西。没有任何理由扔掉坏文件消息，除非有人确实想用那个名字生成文件，当然最好还是不要用这个名字。虽然不是一个很好的示例，但至少已经避免了明显的缺陷。

该示例显示的输出是在Solaris系统上生成的。这个程序根本不能在FreeBSD、Darwin或Linux上运行，因为它们的ed版本是不同的。如果对细节好奇，那么可以看练习6.12。

6.6 用双向管道进行双向通信

第一次介绍管道时（见6.2.1节）曾说过，第一个文件描述符（`pfid[0]`）是为读取而打开

的，第二个（`pfd[1]`）是为写入而打开的，并且写在`pfd[1]`上的东西能从`pfd[0]`中读出。然后给出过几个有用的例子。在例子中，一个进程是写进程，而另一个是读进程，包括使用两个管道（每个都是单向的）进行双向通信的例子，实际上这种方式很像两个单向公路组成一个分车道的高速公路。

所有介绍过的关于管道的内容都是真实的，但这并不意味着文件描述符仅为读或写打开。为探知其中的原因，下面给出一个能够显示管道文件描述符访问模式的小程序：

```
void pipe_access_mode(void)
{
    int pfd[2], flags, i;

    ec_negl( pipe(pfd) )
    for (i = 0; i < 2; i++) {
        ec_negl( flags = fcntl(pfd[i], F_GETFL) )
        if ((flags & O_ACCMODE) == O_RDONLY)
            printf("pfd[%d] O_RDONLY\n", i);
        if ((flags & O_ACCMODE) == O_WRONLY)
            printf("pfd[%d] O_WRONLY\n", i);
        if ((flags & O_ACCMODE) == O_RDWR)
            printf("pfd[%d] O_RDWR\n", i);
    }
    return;

    EC_CLEANUP_BGN
    EC_FLUSH("pipe_access_mode")
    EC_CLEANUP_END
}
```

在Linux系统上，得到了所期望的内容：

```
pfd[0] O_RDONLY
pfd[1] O_WRONLY
```

但看看FreeBSD和Solaris上的输出：

```
pfd[0] O_RDWR
pfd[1] O_RDWR
```

管道的两个结尾对读和写都是打开的！这证明在这两个系统上（和其他系统上）写在`pfd[0]`上的任何东西都可以在`pfd[1]`中读出，而写在`pfd[1]`上的任何东西也都可以从`pfd[0]`中读出，并且数据是绝不会混淆的。事实上，在这些系统上，单个管道就是没有冲突的、分车道的高速公路。这说明仅用一个管道便可以建立双向通信，因为管道是双向的。

为了说明这一点，下面给出来自前节中最后一个例子`edinvoke`的一个修正版本：

```
static FILE *sndfp, *rcvfp;

static bool edinvoke(void)
{
    int pfd[2];
    pid_t pid;

    ec_negl( pipe(pfd) )
    switch (fork()) {
        case -1:
            EC_FAIL
        case 0:
            ec_negl( dup2(pfd[0], STDIN_FILENO) )
            ec_negl( dup2(pfd[0], STDOUT_FILENO) )
```

```

    ec_neg1( close(pfd[0]) )
    execlp("ed", "ed", "--", (char*) NULL);
    EC_FAIL
}
ec_null( sndfp = fdopen(pfd[1], "w") )
ec_null( rcvfp = fdopen(pfd[1], "r") )
return true;

EC_CLEANUP_BGN
if (pid == 0) {
    EC_FLUSH("edinvoke");
    _exit(EXIT_FAILURE);
}
return false;
EC_CLEANUP_END
}

```

上面是改动过的所有内容——程序的其他部分是相同的。其功能和以前完全相同，但用一个管道取代了两个管道。

下面是使用双向管道来代替两个单向管道的缺点：

- 因为它们是非标准的，所以不可移植。
- 也必须读数据的写进程不能关闭管道的写结尾以模拟文件结尾，因为它只有一个结尾。这意味着，例如，前节的 `fsort` 例子不可能写成只使用一个管道。（如果不相信，可以试试。）

因此，如果需要双向通信时，最好假定管道是单向的，并使用两个管道。

练习

6.1 尽可能多地使用能得到的 shell（例如 `sh`、`ksh`、`bash`、`csk`）尝试输入错误命令行，看看会得到什么错误消息，如果出现错误，看看是怎么发生的。初学者可以尝试下面这些例子：

```

echo abc > f1 > f2
echo def | cat < f1
echo ghi > f1 | cat
cat < f1 < f2
echo jkl | | cat
echo jki | > f1

```

6.2 在 6.4 节中给出的函数 `gettoken` 调用了 `getchar`，且函数处于 `INQUOTE` 状态时，没有过多地注意文件结尾。这是个问题吗？为什么是或者为什么不是？

6.3 把参数替换物（如 `echo $PATH`）添加到 6.4 节的 shell 中。

6.4 把通配符（文件名生成）添加到 6.4 节的 shell 中。如果可以，就使用标准函数 `glob`（大部分系统都有）。如果不行，就限制匹配路径名的最后部分（即 `ls */*.c` 是非法的）。

6.5 把 `goto` 内置语句添加到 6.4 节的 shell 中。能够把它实现成像子进程一样运行的外部命令吗？（提示：复习 2.2.3 节中的打开文件描述。）

6.6 把 `if` 语句添加到 6.4 节的 shell 中。

6.7 设计并实现一个菜单 shell。不是提示命令，而是显示一个用户可以从中挑选的选择菜单，命令参数也需要菜单。这个 shell 不用嵌入各种命令的细节，应该从数据库中到命令及其参数的信息。

6.8 设计和实现一个窗口 shell，如果有访问权就使用 `Curses`。把终端屏幕分为两部分（水平划分或垂直划分，哪一个更简单？）。每一半运行一个 shell 子进程。按下功能键或控制键（如，`Ctrl-w`）可以选择活动窗口。非活动窗口运行的命令继续向屏幕输出，但有输入请求时，它会阻塞，直到用户激活了该窗口并响应。窗口满时会滚动，滚动出的行永远消失。仅支持面向行的输入和输出。设计问

题：命令的标准输出应当直接传送给CRT、特别设计的过滤程序还是返回给窗口shell呢？关于输入呢？可选的附加项（范围从难到特别难）：用户能反滚窗口查看消失的内容（取决于某种限制）。可以控制面向屏幕和面向行的输出。

- 6.9 写一个程序，实现向标准输出写它的进程ID，然后从它的标准输入读进程ID的列表。如果它本身的进程ID在列表中，就输出列表（使用文件描述符2）。否则把自己的进程ID添加到该列表，并向标准输出写整个列表。接着重复操作。把5个运行该程序的进程安排成一个环，让它们运行一会（这是让计算机自己玩的一个游戏）。
- 6.10 使用与6.5节中的交互搜索程序类似的方案，为桌面计算器dc写一个前端程序，允许用户使用中缀符号（2+3）代替后缀符号（2 3+）输入表达式。（这是bc命令的功能。）
- 6.11 颠倒6.3节中的最后一个例子的命令顺序，用wc作who的父进程。
- 6.12 如果你使用的是FreeBSD、Darwin或Linux，那么试着指出为什么6.6节中的搜索程序不能正常工作。
提示：它故意产生错误，这一直是个有问题的技术，但它们的ed版本并不像这个搜索程序。是把它当作漏洞，还是编入联机手册，还是两者都要呢？有方法修补吗？
- 6.13 拓展你在练习5.14中编写的程序，以包含在本章解释的附录A中的进程属性。



第7章 高级进程间通信

7.1 概述

到目前为止，我们已经知道如何用管道连接两个进程，但这仅针对有关的进程，而且仅针对管道在其中一个进程之前被创建的情况，这是由于使用文件描述符的唯一方式是继承。对于更现实的情况：服务器连续运行，与服务器通信的客户经常要连接、断开，这时用管道的话限制就太多了。在建立连接时，需要更大的灵活性，对信息流需要更多的控制（例如队列和优先权），并且当有许多数据通信时，需要更高的效率。

值得高兴的是，目前UNIX系统除了管道通信外还有几种进程间通信（IPC）的机制：命名管道、消息队列、信号量和文件锁、共享内存以及套接字。本章涵盖了在单一系统内传递数据的所有IPC机制。下章将讲述用于不同系统间的套接字。

大部分IPC机制都很灵活，完全可以适应大部分的应用程序体系，但是如果这里把实例限定在单服务器同多客户交换数据消息的情况，那么对于初学者来说就更容易理解了。为了简单明了地比较这些机制，这里介绍一种简单消息接口（SMI），然后用6种不同的方式实现它，这6种方式为：FIFO（命名管道）、System V消息队列、POSIX消息队列、System V共享内存和信号量、POSIX共享内存和信号量、套接字（在下一章介绍）。这会给你打下一个坚实的基础，使你能为自己的应用程序选择恰当的机制，然后当你要研究所选机制的高级特性时，可以在此基础上建立。

既然用这几种方式基本都可以实现同一件事，那么自然就想知道哪种是最好的。下面就是达到“最好”的起码的3个尺度：

- 方便使用，与应用程序的需要相匹配；
- 可移植性，可用于不同的系统，如从HP/UX、Solaris或AIX移植到Linux；
- 效率。

本书在介绍每种IPC机制和在“评论”部分进行总结时，将深入地研究前两个尺度。效率很大程度上取决于系统中IPC机制的实现情况，以及应用程序如何使用这种机制，因此，如果可行的话，可以考虑应用程序的抽象层（与SMI类似）。这就允许在程序运行后实验不同的机制。在本章末尾将提供一些定时测试的结果。当然，这肯定不是定论，可能会有不同的结果。

7.2 FIFO或命名管道

FIFO结合了普通文件和管道的特性。与普通文件相同的是，它有名字，任何有适当权限的进程都可以打开它进行读取和写入。与管道不同的是，不相关的进程可以通过FIFO进行通信，因为进程不单单依靠继承性来访问它。但是FIFO打开时，比普通文件更像管道。它遵循了6.2.2节描述的管道行为方式。

一般来说，当读进程打开FIFO时，open需要等到有写进程打开FIFO时才能执行，通常该写进程是另外一个进程。同样，一个写入进程的open操作也会阻塞，直到有另一个读进程打开FIFO为止。因此无论哪个进程（读进程或写进程）首先执行open操作，都需要等待另一个进程。这就允许进程在实际数据传输开始前使它们自身达到同步。

如果在open上设置了O_NONBLOCK标志,那么对该进程的open会立即返回(利用打开文件描述符),而不用等待写进程;而对于写,如果没有读进程打开FIFO,那么open会将errno设为ENXIO,并返回值-1。这种不对称意味着当读进程用O_NONBLOCK打开时是有用的,但当写进程这样打开时却很笨拙,这是因为必须处理错误,然后也许要必须重新尝试open。这种行为的目的是预防已经将数据放入FIFO但不能马上读取的进程,因为UNIX没有办法在FIFO中永久地存储数据。就像水管一样,两头被焊接起来时才能打开水管开关。当所有读写进程都关闭了文件描述符时,如果FIFO里面还有数据的话,那么数据会被抛弃且没有任何错误提示。这也很像水管:如果在两头断开水管,里面的水就会泄露出来。

7.2.1 创建FIFO

与普通文件不同,不能利用open创建FIFO;可以使用单独的系统调用:

```
mkfifo —— 建立FIFO

#include <sys/stat.h>

int mkfifo(
    const char *path,          /* pathname */
    mode_t perms              /* permissions */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

perms参数的使用方法与open的第三个参数相似;也就是说,它是新文件的权限。

FIFO的明显应用是替换管道。代替使用pipe系统调用,创建一个FIFO,为了得到读文件描述符和写文件描述符打开它两次。[⊖]从这时起就可以把它看作是管道。但是与管道相比,这样用FIFO没有任何优势,反而有几个缺点:创建FIFO需要额外开销,要得到文件描述符需要两个系统调用,和使用临时文件一样有命名冲突的风险。

FIFO不能添加到UNIX里以替换管道,添加FIFO是为了在服务器进程和客户端之间提供一个简单的传送数据的方法。回顾管道的一个限制是:用于读写管道的文件描述符只能通过继承传递给一个进程。但是如果服务器保持运行状态,而客户端进程经常链上断开,那么就不能这么做。由于任何拥有适当权限的进程都可以打开一个FIFO,因此服务器可以比较容易安排创建一个有固定名字的进程以便客户端能打开它。

在本章,将使用所有引入的进程间通信机制在进程间传送消息,这里消息指的是较短的结构化数据块,而不是如第6章一样采用数据流。然而,需要记住同样可以使用FIFO来传送数据流。

7.2.2 一个简单的FIFO例子

首先从两个程序开始,服务器和客户端。服务器不间断运行,等待客户端给它发送消息。本例中,消息是一组需要转换为大写字母的字符串。服务器转换完毕后,把消息送回给客户端,然后继续循环等待下一个可能来自不同客户端的消息。例中的客户端发送少量字符串给服务器,显示结果,然后退出。

服务器是这样开始的:

```
$ msg_server&
server started
```

[⊖] 在你的系统上,读进程和写进程同时(O_RDWR)打开它可能也可以正常工作,但是那样不标准。

```
[1] 8725
$
```

然后客户端运行如下：

```
$ smsg_client
client 8747 started
client 8747: applesauce --> APPLESAUCE
client 8747: tiger --> TIGER
client 8747: mountain --> MOUNTAIN
Client 8747 done
```

图7-1给出了服务器、客户端和FIFO的工作过程。服务器面向所有客户端创建一个输入FIFO，称为“fifo_server”。每个客户端（图中有两个）创建自身的FIFO用来接收服务器的应答，它们所使用的进程ID的命名是唯一的。如图，客户端8748发送一个消息（用正方形表示），其中包含要转换的数据（“tiger”）和自身的进程ID。服务器由自身的输入FIFO读取信息，进行转换（转换成“TIGER”），然后利用发送的进程ID（8748）决定把结果送回哪个FIFO。因此，它们仅有一个共同的服务器FIFO，但每个客户端分别有各自的应答FIFO。

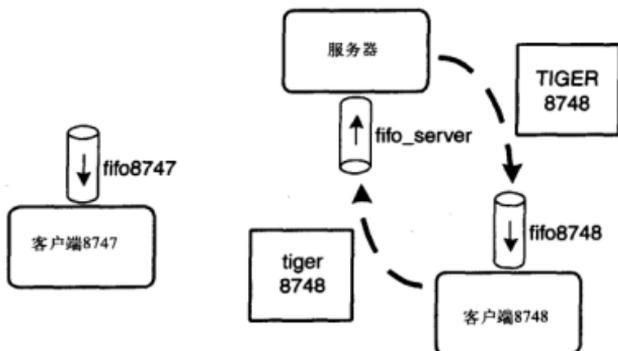


图7-1 带有两个客户端的服务器

客户端和服务端必须使用相同的算法来组成从客户端进程ID传来的FIFO名字，为了这个目的，在本例中使用它们共享的一个函数：

```
bool make_fifo_name(pid_t pid, char *name, size_t name_max)
{
    snprintf(name, name_max, "fifo%d", (long)pid);
    return true;
}
```

公共的头文件定义了固定的服务器FIFO名字及消息的结构：

```
#define SERVER_FIFO_NAME "fifo_server"

struct simple_message {
    pid_t sm_clientpid;
    char sm_data[200];
};
```

下面是服务器程序的代码：

```
int main(void)
```



```

{
    int fd_server, fd_client, i;
    ssize_t nread;
    struct simple_message msg;
    char fifo_name[100];
    printf("server started\n");
    if (mkfifo(SERVER_FIFO_NAME, PERM_FILE) == -1 && errno != EEXIST)
        EC_FAIL
    ec_negl( fd_server = open(SERVER_FIFO_NAME, O_RDONLY) )
    while (true) {
        ec_negl( nread = read(fd_server, &msg, sizeof(msg)) )
        if (nread == 0) {
            errno = ENETDOWN;
            EC_FAIL
        }
        for (i = 0; msg.sm_data[i] != '\0'; i++)
            msg.sm_data[i] = toupper(msg.sm_data[i]);
        ec_false( make_fifo_name(msg.sm_clientpid, fifo_name,
            sizeof(fifo_name)) )
        ec_negl( fd_client = open(fifo_name, O_WRONLY) )
        ec_negl( write(fd_client, &msg, sizeof(msg)) )
        ec_negl( close(fd_client) )
    }
    /* never actually get here */
    ec_negl( close(fd_server) )
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}

```

服务器的工作如下:

- 它创建自身的FIFO。除了FIFO已存在之外，这一过程只有mkfifo返回错误时才会失败，因为它可能是从服务器以前的执行中遗留下来的。
- 为读取而打开自身的FIFO。该操作会阻塞，直到有写进程打开FIFO，所以在客户端启动以前启动服务器也是可以的。
- 每次都执行这样的循环：先读取消息，再转换数据，然后打开客户端的FIFO，向该FIFO写入结果消息，最后关闭。
- 由于没有提供明确的停止服务器的方法，所以服务器只是无限期地保持在循环中，一直到kill命令终止它（本例中没有显示）。

你可能已经想象出客户端程序是如何工作的了。下面是客户端的代码:

```

int main(int argc, char *argv[])
{
    int fd_server, fd_client = -1, i;
    ssize_t nread;
    struct simple_message msg;
    char fifo_name[100];
    char *work[] = {
        "applesauce",
        "tiger",
        "mountain",
        NULL
    };
};

printf("client %ld started\n", (long)getpid());

```

```

msg.sm_clientpid = getpid();
ec_false( make_fifo_name(msg.sm_clientpid, fifo_name,
    sizeof(fifo_name)) )
if (mkfifo(fifo_name, PERM_FILE) == -1 && errno != EEXIST)
    EC_FAIL
ec_neg1( fd_server = open(SERVER_FIFO_NAME, O_WRONLY) )
for (i = 0; work[i] != NULL; i++) {
    strcpy(msg.sm_data, work[i]);
    ec_neg1( write(fd_server, &msg, sizeof(msg)) )
    if (fd_client == -1)
        ec_neg1( fd_client = open(fifo_name, O_RDONLY) )
    ec_neg1( nread = read(fd_client, &msg, sizeof(msg)) )
    if (nread == 0) {
        errno = ENETDOWN;
        EC_FAIL
    }
    printf("client %ld: %s --> %s\n", (long)getpid(),
        work[i], msg.sm_data);
}
ec_neg1( close(fd_server) )
ec_neg1( close(fd_client) )
ec_neg1( unlink(fifo_name) )
printf("Client %ld done\n", (long)getpid());
exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}

```

客户端的工作如下:

- 创建自身的FIFO。
- 为写进程打开服务器的FIFO。该操作会阻塞,直到有读进程打开FIFO,所以在服务器启动之前启动客户端是可以的。
- 对于每个需要转换的字符串,形成消息,写到服务器,如果必要的话还可以打开其自身客户端的读进程来读取结果。
- 所有的字符串转换完毕后,由于其他客户端不可能使用其FIFO,客户端将关闭已打开的文件描述符并解链FIFO。

有两条不好的消息:首先,客户端和服务器都有漏洞并具有相似的特性;其次,前面给出的客户端输出是不正确的。实际上,客户端的输出应是这样的:

```

$ smsg_client
client 8747 started
client 8747: applesauce --> APPLESAUCE
ERROR: 0: main [/aup/c7/smsg_client.c:46] 0
*** ? (100: "Network is down") ***
$

```

发生了什么事?漏洞是什么?位置和errno的值告知客户端是从自己的FIFO中获得文件结尾的。为什么客户端向服务器发送消息后没有数据?该问题是一种定时问题:服务器在收到每条消息后关闭了客户端FIFO的文件描述符,并且在客户端从write到read之前服务器没有抽出时间重新打开它。所以对写进程没有任何文件描述符打开,客户端得到的是文件结尾(如6.2.2节中解释的那样)。

修改很容易。如果客户端为了向自身的FIFO写入,保持文件描述符的打开状态(即使永

远不会写)，那么它会强制其FIFO的read进入阻塞状态，直到有（来自服务器的）数据写入。这就是所希望的行为。因此必须修改客户端以声明另一个文件描述符变量，代码如下：

```
int fd_client_w = -1;
```

然后连同读文件描述符一起打开，像这样：

```
if (fd_client == -1)
    ec_negl( fd_client = open(fifo_name, O_RDONLY) )
if (fd_client_w == -1)
    ec_negl( fd_client_w = open(fifo_name, O_WRONLY) )
```

（应当连同fd_client一起关闭。）在多数UNIX系统中，也可以仅利用一个同时为读进程和写进程打开的文件描述符来修改客户端，但是这样是不标准的。

因此，重新运行修改后的客户端，得到的输出如下：

```
client 8937 started
client 8937: applesauce --> APPLESAUCE
client 8937: tiger --> TIGER
client 8937: mountain --> MOUNTAIN
Client 8937 done
ERROR: 0: main [/aup/c7/smsg_server.c:33] 0
*** ? (100: "Network is down") ***
```

仍然不正确！现在服务器不满意的是：从它的FIFO中得到一个文件结尾。原因是当客户端没有工作时，关闭了服务器FIFO的写结尾并且在没有其他客户端运行时，服务器得到了文件结尾。这不是所希望得到的，希望的是服务器保持运行状态，理想情况是阻塞在read中，直到有消息。对服务器的修改和客户端相似，为了向自身的FIFO写入，服务器保持文件描述符的打开状态，以避免文件结尾。（这里就不给代码了，因为写起来很容易。）

修改服务器后，不仅可以顺利地运行这两个程序，而且甚至可以运行多个客户端：

```
$ smsg_client & smsg_client & smsg_client & smsg_client
client 9001 started
client 9001: applesauce --> APPLESAUCE
client 9001: tiger --> TIGER
[2] 9001
client 9002 started
client 9002: applesauce --> APPLESAUCE
client 9002: tiger --> TIGER
[3] 9002
[4] 9003
client 9004 started
client 9004: applesauce --> APPLESAUCE
client 9004: tiger --> TIGER
client 9003 started
client 9003: applesauce --> APPLESAUCE
client 9003: tiger --> TIGER
client 9002: mountain --> MOUNTAIN
client 9001: mountain --> MOUNTAIN
client 9004: mountain --> MOUNTAIN
client 9003: mountain --> MOUNTAIN
Client 9001 done
Client 9002 done
Client 9004 done
[2] Done                smsg_client
[3]- Done                smsg_client
$ Client 9003 done
[4]+ Done                smsg_client
```



7.2.3 评价FIFO

FIFO的优点是:

- 容易理解, 易于应用。因为它们实际上是管道, 所以可以使用基本的I/O系统调用 (`open`, `read`, `write`等)。
- 可用在所有版本的UNIX中。
- 效率相当高 (见本章末尾的表7-2)。
- 对数据流和离散消息的处理较好。为了加速处理, 一个读进程可以一次读取多条消息, 一个写进程可以一次写完整个消息缓冲区, 前提是只要没有超过原子写的最大值。

缺点是:

- 单个FIFO不能拥有多个读进程, 原因是当进行读操作时无法保证操作的原子性 (对该问题的更多讨论见6.2.2节)。
- 必须将数据从某个进程的用户空间复制到内核缓存区, 然后回到另一个用户空间, 这样开销会很大。(消息队列和套接字有相同的缺点。)因此, FIFO不适合最关键的应用。
- 如果消息太大 (见6.2.2节), 写进程可能会阻塞。如果不仔细控制, 应用程序可能会死锁。

7.3 抽象的简单消息接口(SMI)

把上面例子里FIFO中消息的发送和接收归纳成抽象的接口是有用的, 两个关键原因如下:

- 像保持文件描述符处于打开状态以便可写这样的重要细节可以通过抽象接口的实现来控制, 使应用程序的编写更容易、更可靠。
- 对于不同的IPC机制可以编写不同的实现程序。不用改变应用程序的源代码, 就可以用不同的实现来实验以发现其中哪一个是最好的。

本书把这种接口称为SMI, 即Simple Messaging Interface的缩写。[⊖]

7.3.1 SMI类型和函数

本节将解释SMI接口, 特定的实现将在稍后介绍。首先, 我们给出通用的消息结构:

struct smi_msg — SMI的结构

```
struct smi_msg {
    long smi_mtype; /* must be first */
    struct client_id {
        long c_id1;
        long c_id2;
    } smi_client;
    char smi_data[1];
};
```

我们将看到, 一些实现需要前两个成员 (`smi_mtype`和`smi_client`)。在7.2.2节的示例中我们已知, 客户端必须将它的进程ID传递给服务器; 在SMI消息中, 进程ID存放在`client.cl_id1`中。我们稍后将介绍`smi_mtype`和`smi_client.c_id2`的使用。

消息数据 (`smi_data`) 可以是满足如下限制的任意信息

- 服务器和客户端可能在不同的机器上, 因此, 不应在网络上传递无意义的指针或其他数据。
- 因为不同的机器有不同的字节序, 二进制数需要表示成网络标准格式。8.1.4节会介绍这

⊖ 这是为了本书设计的; 并不是标准的一部分。

一主题。

在这个简单的接口中，服务器和客户端必须就固定的消息大小达成一致，而`smi_data`数组的实际大小则取决于此。一旦掌握了本章和下一章的各项原则，你就可以在你自己的应用程序使用的SMI版本中试着放松上述的限制。

在发送或接收消息之前，服务器或客户端必须打开一个消息队列，该队列应在通信终止时关闭。概念上，要发送的消息被放入队列，要接收的消息被从队列中取出。实际的队列是与实现相关的，并且SMI实现隐藏了队列的细节。这很像是标准I/O函数隐藏了FILE类型的细节。

打开和关闭接口的操作是：

SMI types —— SMI的类型

```
typedef void *SMIQ;           /* message queue */
typedef enum {
    SMI_SERVER,             /* server */
    SMI_CLIENT              /* client */
} SMIENTITY;
#define SERVER_NAME_MAX 50  /* max size of server name */
```

smi_open —— 打开SMI消息队列

```
SMIQ *smi_open(
    const char *name,        /* server name */
    SMIENTITY entity,       /* entity being opened */
    size_t msgsize         /* fixed message size */
);
/* Returns pointer to message queue or NULL on error (sets errno) */
```

smi_close —— 关闭SMI消息队列

```
bool smi_close(
    SMIQ *smp               /* queue */
);
/* Returns true on success or false on error (sets errno) */
```

服务器和所有客户端都必须知道传递给`smi_open`的消息队列名。如果实现需要在打开它之前创建它，那么可以在`smi_open`内部完成这个工作。在这个简单接口上没有任何许可权限，这也是为什么它被称为“简单”的原因之一。消息队列在关闭之后是否需要被留下，这没有定义为接口的一部分，要取决于具体实现。

`entity`参数是`SMI_SERVER`还是`SMI_CLIENT`，要取决于哪一种类型的程序正在执行打开操作。那可能仅有一个服务器进程但有不确定数量的客户端。

对所有消息来说，`msgsize`参数是`smi_msg`结构中`smi_data`成员的大小。如前面所述，应提供相同大小的消息。

为了发送和接收，这里仅把一个缓冲区的地址传给简单的收发函数，与`write`和`read`有点相似（没有参数大小，因为它是固定值）：

```
ec_false( smi_send(smp, buffer) )
...
ec_false( smi_receive(smp, buffer) )
```

但是如果这样做，就要求所有具体实现都要复制每一条消息：从发送进程复制到内核，再从内核复制到接收进程。这里我们不那么做，而是用一个稍微更详细的接口，该接口调用两次发送进程，调用两次接收进程。每对调用的第一个调用仅得到消息的地址，第二个调用释放对那个地址的访问。这样实现可以将它保留在`SMIQ`结构中（隐含其确实已被复制），或者保留在共享内存中，或者其他任何地方。所有调用者都知道的是它们有一个可以间接引用的地

址。必须有释放地址的函数，因为该地址不可能永远可用（内存可能必须被重新分配，或者共享内存段必须被重用），又因为底层具体实现需要知道什么时候发送程序准备好了发送消息。

解释发送操作前先解释接收操作会更容易：

smi_receive_getaddr —— 得到收到的SMI消息地址

```
bool smi_receive_getaddr(
    SMIQ *smp,           /* queue */
    void **addr         /* message */
);
/* Returns true on success or false on error (sets errno) */
```

smi_receive_release —— 释放收到的SMI消息

```
bool smi_receive_release(
    SMIQ *smp           /* queue */
);
/* Returns true on success or false on error (sets errno) */
```

addr参数是一个指向void指针的指针，而不是一个指向smi_msg结构的指针，因为实际上每一个应用都将定义自己的消息结构，消息结构中的前两个成员和smi_msg中的前两个成员（smi_mtype和smi_client）匹配。

应用程序是这样使用这两个调用的：

```
struct my_msg *msg;
...
ec_false( smi_receive_getaddr(smp, (void **)&msg) )
/* process data in msg->smi_data */
ec_false( smi_receive_release(smp) )
```

从概念上讲，调用smi_receive_getaddr时实际上已经收到了消息。要注意，应用不给消息分配空间；这是由具体实现来完成的。

发送操作是非常相似的，但添加了一点用于识别客户端的技巧：

smi_send_getaddr —— 得到发送的SMI消息地址

```
bool smi_send_getaddr(
    SMIQ *smp,           /* queue */
    struct client_id *client, /* client ID (server only) */
    void **addr         /* message */
);
/* Returns true on success or false on error (sets errno) */
```

smi_send_release —— 释放并发送SMI消息

```
bool smi_send_release (
    SMIQ *smp           /* queue */
);
/* Returns true on success or false on error (sets errno) */
```

从概念上讲，当调用smi_send_release时，产生实际的发送。

服务器使用smi_send_getaddr的第二个参数识别正在向其发送消息的客户端。它是一个指向struct client_id结构的指针，该结构位于收到的消息中。这样做是因为服务器在向客户端发送消息之前，先从需要与其进行通信的客户端接收消息。发送消息的客户端不需要识别服务器（通过smi_open完成），所以第二个参数是NULL。

然后服务器接着做下面这样的工作：

```
struct my_msg *msg_in, *msg_out;
```

```

...
ec_false( smi_receive_getaddr(sqp, (void **)&msg_in) )
/* code to process data in msg_in->smi_data */
ec_false( smi_send_getaddr(sqp, &msg_in->smi_client, (void **)&msg_out) )
ec_false( smi_receive_release(sqp) )
/* code to put data into msg_out->smi_data */
ec_false( smi_send_release(sqp) )

```

可以观察到:

- 服务器使用了两个单独的my_msg地址 (msg_in和msg_out), 因为涉及两个不同的缓冲区;
- 直到调用smi_send_getaddr时才会释放msg_in, 因为直到那时才需要msg_in->smi_client结构。可以以这种方式交替插入调用; 事实上, 甚至在调用smi_receive_release之后 (发送操作和接收操作完全是独立的), 才可能会调用smi_send_release。或者说, client_id结构可能已经被复制到一个临时变量里了, 该变量的地址可能在调用smi_send_getaddr中使用。

客户端采用如下方式调用smi_send_getaddr和smi_send_release (可能有一个msg_in, 但这里没有显示):

```

struct my_msg *msg_out;
...
ec_false( smi_send_getaddr(sqp, NULL, (void **)&msg_out) )
/* code to put data into msg_out->smi_data */
ec_false( smi_send_release(sqp) )

```

下一节有更完整的解释。

7.3.2 服务器和客户端使用SMI的示例

为了说明在一个应用中如何使用SMI函数, 下面对7.2.2节中的那个服务器的例子进行改写, 以使用那个接口 (define包含在服务器和客户端包括的头文件中):

```

#define SERVER_NAME "smsg_server"
#define DATA_SIZE 200

int main(void)
{
    SMIQ *sqp;
    struct smi_msg *msg_in, *msg_out;
    int i;

    printf("server started\n");
    ec_null( sqp = smi_open(SERVER_NAME, SMI_SERVER, DATA_SIZE) )
    while (true) {
        ec_false( smi_receive_getaddr(sqp, (void **)&msg_in) )
        ec_false( smi_send_getaddr(sqp, &msg_in->smi_client,
            (void **)&msg_out) )
        for (i = 0; msg_in->smi_data[i] != '\0'; i++)
            msg_out->smi_data[i] = toupper(msg_in->smi_data[i]);
        msg_out->smi_data[i] = '\0';
        ec_false( smi_receive_release(sqp) )
        ec_false( smi_send_release(sqp) )
    }
    /* never actually get here */
    ec_false( smi_close(sqp) )
    exit(EXIT_SUCCESS);
}

```

```

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}

```

下面是客户端的代码:

```

int main(int argc, char *argv[])
{
    SMIQ *sqq;
    struct smi_msg *msg;
    int i;

    char *work[] = {
        "applesauce",
        "tiger",
        "mountain",
        NULL
    };

    printf("client %ld started\n", (long)getpid());
    ec_null( sqq = smi_open(SERVER_NAME, SMI_CLIENT, DATA_SIZE) )
    for (i = 0; work[i] != NULL; i++) {
        ec_false( smi_send_getaddr(sqq, NULL, (void **)&msg) )
        strcpy(msg->smi_data, work[i]);
        ec_false( smi_send_release(sqq) )
        ec_false( smi_receive_getaddr(sqq, (void **)&msg) )
        printf("client %ld: %s --> %s\n", (long)getpid(),
            work[i], msg->smi_data);
        ec_false( smi_receive_release(sqq) )
    }
    ec_false( smi_close(sqq) )
    printf("Client %ld done\n", (long)getpid());
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}

```

注意，如我们在7.2.2节中看到的，为了进行写操作而保持文件描述符处于打开状态，现在这些不好的鬼把戏已经不存在了，例如，像给FIFO起名字这样的细节就没有了。或许用“隐藏”比用“不存在”更准确，接下来将要讨论这一点。

7.3.3 SMI的FIFO具体实现

现在利用FIFO去除来自服务器和客户端的消息的所有复杂性必须深入到SMI函数的具体实现。另外，服务器的实现试图使客户端的FIFO文件描述符通过消息仍保持打开，以便减少每次打开和关闭它们的开销，见7.2.2节中的示例。

FIFO实现使用形如*smifcn_fifo*的函数名，其中*smifcn*是SMI名字（例如，*smi_send_getaddr_fifo*是*smi_send_getaddr*的实现）。一个小的包装程序文件就能有效地转换这些名字，如使用下面的小函数：

```

bool smi_send_getaddr(SMIQ *sqq, struct client_id *client, void **addr)
{
    return smi_send_getaddr_fifo(sqq, client, addr);
}

```

使用包装程序允许链接两类应用:

- 独立于具体实现的应用, 就像前面章节中提到的一样, 能够依据一般的SMI函数编写 (例如smi_send_getaddr), 接着用一个特别的实现和合适的包装程序链接以便把这些调用从一般转换成特殊。
- 一个想要使用几种实现方法的具体实现可以使用特别的函数名 (例如smi_send_getaddr_fifo、smi_send_getaddr_skt), 绕开没有链入的包装程序。这样做的主要目的是写一个测试程序, 例如用不同的方法运行同一个应用, 以便比较它们的性能。本书就是这么做的, 目的是为本章结尾的表7-2做准备。

本书没有给出包装程序的代码, 但可在网站上看到 (AUP2003)。

首先从利用内部消息队列的FIFO实现开始。尽管服务器和客户端处在不同的进程中, 有自己的数据, 但它们使用相同的数据结构。

```
#define MAX_CLIENTS 20

typedef struct {
    SMIENTITY sq_entity;           /* entity */
    int sq_fd_server;             /* server read and ... */
    int sq_fd_server_w;          /* ... write file descriptors */
    char sq_name[SERVER_NAME_MAX]; /* server name */
    struct {
        int cl_fd;                /* client file descriptor */
        pid_t cl_pid;             /* client process ID */
    } sq_clients[MAX_CLIENTS];
    struct client_id sq_client;    /* client ID */
    size_t sq_msgsize;           /* msg size */
    struct smi_msg *sq_msg;       /* msg buffer */
} SMIQ_FIFO;
```

每一个服务器仅有20个客户空间。使用链表可消除这个限制, 但是我不想这么麻烦。客户端和服务端都使用服务器的sq_fd_server, 除此之外服务器用sq_fd_server_w保持一个打开的文件描述符给写进程, 见7.2.2节中的解释。服务器用sq_clients数组保持客户端文件描述符, 以避免每次传入消息都需要打开和关闭它们。客户端仅需要其中之一读取自己的FIFO, 可以使用sq_clients[0]。它用sq_clients[1]来保持第二个打开的文件描述符给写进程。对客户端来说, 数组中的剩余部分和sq_fd_server_w的空间是浪费, 但是我认为它们使用相同的队列结构会更容易, 因为浪费的空间小。

服务器传递给smi_send_getaddr_fifo的客户端ID信息被保留下来, 以便为随后而来的sa_client成员中的smi_send_release_fifo所使用。传递给smi_open_fifo的大小和指向消息缓冲区的指针分别存在于sq_msgsize和sq_msg中。(注意对于FIFO, 不可避免地进行复制, 复制从用户空间到内核再回到接收进程, 这同样也适用于消息队列和套接字。)

服务器和客户端需要使用下面这个内部函数来初始化sq_clients数组:

```
static void clients_bgn(SMIQ_FIFO *p)
{
    int i;

    for (i = 0; i < MAX_CLIENTS; i++)
        p->sq_clients[i].cl_fd = -1;
}
```

并且, 最后它们要调用clients_end:

```
static void clients_end(SMIQ_FIFO *p)
```

```

{
    clients_close_all(p);
}

static void clients_close_all(SMIQ_FIFO *p)
{
    int i;

    for (i = 0; i < MAX_CLIENTS; i++)
        if (p->sq_clients[i].cl_fd != -1) {
            (void)close(p->sq_clients[i].cl_fd);
            p->sq_clients[i].cl_fd = -1;
        }
}

```

在后面的smi_open_fifo函数和smi_close_fifo函数中可以看到这些调用。

当服务器得到消息时,会使用clients_find来查看是否已经为FIFO打开了文件描述符,如果没有,就找一个可用的位置:

```

static int clients_find(SMIQ_FIFO *p, pid_t pid)
{
    int i, avail = -1;

    for (i = 0; i < MAX_CLIENTS; i++) {
        if (p->sq_clients[i].cl_pid == pid)
            return i;
        if (p->sq_clients[i].cl_fd == -1 && avail == -1)
            avail = i;
    }
    if (avail != -1)
        p->sq_clients[avail].cl_pid = pid;
    return avail;
}

```

如果没有更多客户端可以处理,函数就返回-1。

最后的几个内部函数是为FIFO命名的。服务器的命名是固定的,然而客户端的命名包含它们的进程ID,就如前面7.2.2节中较早的例子。

```

static void make_fifo_name_server(const SMIQ_FIFO *p, char *fifoname,
    size_t fifoname_max)
{
    snprintf(fifoname, fifoname_max, "/tmp/smififo-%s", p->sq_name);
}

static void make_fifo_name_client(pid_t pid, char *fifoname,
    size_t fifoname_max)
{
    snprintf(fifoname, fifoname_max, "/tmp/smififo%d", (long)pid);
}

```

现在来看看smi_open_fifo:

```

SMIQ *smi_open_fifo(const char *name, SMIENTITY entity, size_t msgsize)
{
    SMIQ_FIFO *p = NULL;
    char fifoname[SERVER_NAME_MAX + 50];

    ec_null( p = calloc(1, sizeof(SMIQ_FIFO)) )
    p->sq_msgsize = msgsize + offsetof(struct smi_msg, smi_data);
    ec_null( p->sq_msg = calloc(1, p->sq_msgsize) )
}

```

```

p->sq_entity = entity;
if (strlen(name) >= SERVER_NAME_MAX) {
    errno = ENAMETOOLONG;
    EC_FAIL
}
strcpy(p->sq_name, name);
make_fifo_name_server(p, fifoname, sizeof(fifoname));
if (p->sq_entity == SMI_SERVER) {
    clients_bgn(p);
    if (mkfifo(fifoname, PERM_FILE) == -1 && errno != EEXIST)
        EC_FAIL
    ec_negl( p->sq_fd_server = open(fifoname, O_RDONLY) )
    ec_negl( p->sq_fd_server_w = open(fifoname, O_WRONLY) )
}
else {
    ec_negl( p->sq_fd_server = open(fifoname, O_WRONLY) )
    make_fifo_name_client(getpid(), fifoname, sizeof(fifoname));
    (void)unlink(fifoname);
    ec_negl( mkfifo(fifoname, PERM_FILE) )
    ec_negl( p->sq_clients[0].cl_fd =
        open(fifoname, O_RDONLY | O_NONBLOCK) )
    ec_false( setblock(p->sq_clients[0].cl_fd, true) )
    ec_negl( p->sq_clients[1].cl_fd = open(fifoname, O_WRONLY) )
}
return (SMIQ *)p;

EC_CLEANUP_BGN
if (p != NULL) {
    free(p->sq_msg);
    free(p);
}
return NULL;
EC_CLEANUP_END
}

```

这段代码应该容易理解，特别是如果已经理解了7.2.2节中的例子。但是打开FIFO的部分值得再回顾一下。对服务器来说，打开自身FIFO的顺序是：

```

ec_negl( p->sq_fd_server = open(fifoname, O_RDONLY) )
ec_negl( p->sq_fd_server_w = open(fifoname, O_WRONLY) )

```

如果没有写进程，第一个调用会阻塞，这是可以的，因为如果没有写进程，服务器也没有事情做。但是客户端打开它的FIFO的顺序更复杂：

```

ec_negl( p->sq_clients[0].cl_fd =
    open(fifoname, O_RDONLY | O_NONBLOCK) )
ec_false( setblock(p->sq_clients[0].cl_fd, true) )
ec_negl( p->sq_clients[1].cl_fd = open(fifoname, O_WRONLY) )

```

若没有O_NONBLOCK标志，客户端将阻塞open，等待服务器为写进程打开一个FIFO。但是，服务器会等到得到消息之后才去打开，因为直到那时才知道这个特殊客户端的存在。死锁！解决方法是打开FIFO，解除阻塞，如代码中所显示的，这意味着可以返回一个有效的文件描述符而不用等待写进程。但是必须清除O_NONBLOCK标志（如设置阻塞）以便阻塞后的读进程，这正是我们想要的行为。（使用fcntl系统调用的setblock来自4.2.2节。）在7.2.2节的例子中，没有必要这么做，因为客户端直到已经向服务器发送了消息才打开自己的FIFO。这里不是那么工作的，因为SMI函数是独立的，有专门的工作要做。（有时模块化需要额外的工作。）

下面是与其配对的smi_close_fifo函数：

```

bool smi_close_fifo(SMIQ *sqp)
{
    SMIQ_FIFO *p = (SMIQ_FIFO *)sqp;

    clients_end(p);
    (void)close(p->sq_fd_server);
    if (p->sq_entity == SMI_CLIENT) {
        char fifoname[SERVER_NAME_MAX + 50];

        make_fifo_name_client(getpid(), fifoname, sizeof(fifoname));
        (void)unlink(fifoname);
    }
    else
        (void)close(p->sq_fd_server_w);
    free(p->sq_msg);
    free(p);
    return true;
}

```

注意，客户端解链了它们的FIFO，但是服务器却保留了它的FIFO，以便将来即使没有服务器运行，也可以启动客户端。

下面是smi_send_getaddr_fifo函数，除了保存客户端ID和返回缓冲区地址外不做其他事情：

```

bool smi_send_getaddr_fifo(SMIQ *sqp, struct client_id *client,
    void **addr)
{
    SMIQ_FIFO *p = (SMIQ_FIFO *)sqp;

    if (p->sq_entity == SMI_SERVER)
        p->sq_client = *client;
    *addr = p->sq_msg;
    return true;
}

```

真正的工作在smi_send_release_fifo中：

```

bool smi_send_release_fifo(SMIQ *sqp)
{
    SMIQ_FIFO *p = (SMIQ_FIFO *)sqp;
    ssize_t nwrite;

    if (p->sq_entity == SMI_SERVER) {
        int nclient = clients_find(p, p->sq_client.c_id1);
        if (nclient == -1 || p->sq_clients[nclient].cl_fd == -1) {
            errno = EADDRNOTAVAIL;
            EC_FAIL
        }
        ec_neg1( nwrite = write(p->sq_clients[nclient].cl_fd, p->sq_msg,
            p->sq_msgsize) )
    }
    else {
        p->sq_msg->smi_client.c_id1 = (long)getpid();
        ec_neg1( nwrite = write(p->sq_fd_server, p->sq_msg,
            p->sq_msgsize) )
    }
    return true;
}

```

EC_CLEANUP_BGN

```

    return false;
EC_CLEANUP_END
}

```

在7.2.2节中较早的例子中没有出现也是很正常的。smi_receive_getaddr_fifo做了接收消息的所有工作：

```

bool smi_receive_getaddr_fifo(SMIQ *sqp, void **addr)
{
    SMIQ_FIFO *p = (SMIQ_FIFO *)sqp;
    ssize_t nread;

    if (p->sq_entity == SMI_SERVER) {
        int nclient;
        char fifoname[SERVER_NAME_MAX + 50];

        while (true) {
            ec_negl( nread = read(p->sq_fd_server, p->sq_msg,
                p->sq_msgsize) )

            if (nread == 0) {
                errno = ENETDOWN;
                EC_FAIL
            }
            if (nread < offsetof(struct smi_msg, smi_data)) {
                errno = E2BIG;
                EC_FAIL
            }
            if ((nclient = clients_find(p,
                (pid_t)p->sq_msg->smi_client.c_idl)) == -1) {
                continue; /* client not notified */
            }
            if (p->sq_clients[nclient].cl_fd == -1) {
                make_fifo_name_client((pid_t)p->sq_msg->smi_client.c_idl,
                    fifoname, sizeof(fifoname));
                ec_negl( p->sq_clients[nclient].cl_fd =
                    open(fifoname, O_WRONLY) )
            }
            break;
        }
    }
    else
        ec_negl( nread = read(p->sq_clients[0].cl_fd, p->sq_msg,
            p->sq_msgsize) )
    *addr = p->sq_msg;
    return true;
EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

```

下面是棘手的部分：

```

if ((nclient = clients_find(p, (pid_t)mp->smi_client)) == -1) {
    continue; /* client not notified */
}

```

如果clients_find返回-1，这意味着服务器从客户端得到了消息，但是在sq_clients数组中却无法得到位置，所以不能打开响应FIFO。由于不能响应，所以它甚至无法发出错误

消息。示例代码只忽略了错误，但却留下了阻塞的客户端。替代方法如下：

- 发送一个信号到客户端，修改`smi_open_fifo`以便准备捕获它。
- 用一个整型变量控制一个紧急的文件描述符来打开FIFO，以便可以发送错误消息。
- 至少要修改客户端，以便时间到了可以退出，以避免永久阻塞。

还剩下一个函数，但它几乎没做什么事情：

```
bool smi_receive_release_fifo(SMIQ *sqp)
{
    return true;
}
```

这就是整个实现了。即使有已讨论过的那些限制，它仍然是一个高度可移植的、相当可靠的SMI函数的实现，并且在自己的应用中至少对原型而言可以马上使用。以后，也可以用其他某个实现，因为接口是相同的。

7.4 System V IPC

正如1.1.7节所解释的，对消息、信号量和共享存储器来说有两套系统调用。比较旧的一套叫作System V IPC（进程间通信），比较新的一套叫作POSIX IPC。本章将对这两套系统调用进行讨论，首先是System V和POSIX的消息，然后是两者的信号量，最后是共享存储器。本节解释适用于所有System V IPC机制的一般概念；之后在7.6节将解释POSIX IPC的一般概念。

7.4.1 System V IPC 对象

有三种类型的System V IPC 对象：消息队列、信号量集和共享存储器段。它们不是文件，甚至不是特殊文件，而是具有自己的命名方案、自己的生存期规则和自己的访问权限系统的对象。下面是System V IPC 对象的几个主要特征：

- 它们仅存在于单个机器内。它们不能用于跨网络通信。
- 它们的生存期与内核相同，系统重启时它们会被销毁。
- 可以使用整型标识符访问对象，在对象的生存期之内这个标识符是固定的。任何知道此标识符的进程都可直接用它访问对象——不需要事先打开对象。这不同于文件描述符，后者是进程的特性，当进程访问对象时它会消失。（System V IPC也有关键字，在下一节中对其进行讲解。）
- 因为没有信息节点或路径名，所以不能使用传统文件和目录处理系统调用，如`unlink`、`stat`、`read`或`write`。

所有这三种对象都有形如`Xget`和`Xctl`格式的系统调用，这里`X`表示的是`msg`、`sem`或`shm`。因此，6个调用分别为`msgget`、`semget`、`shmget`、`msgctl`、`semctl`和`shmctl`。当把所讨论的内容应用于`Xget`和`Xctl`的所有三个对象时，本书会继续使用术语`Xget`和`Xctl`来指“get”和“ctl”。

另外，还有5个其他调用：`msgsnd`和`msgrcv`用于发送和接收消息；`semop`用于操纵信号量集；`shmat`和`smdt`用于连接和断开共享存储器。

7.4.2 标识符、关键字和`ftok`系统调用

创建System V IPC对象时，内核会给标识符赋值，因此通常情况下系统每次重新启动时，它都会有不同的值。为了使不同的进程能够比较容易地从需要共享的对象中获得标识符，可以用值从来不会改变的永久性关键字。如果对象已经存在，那么当进程用`Xget`创建对象或仅

用Xget从关键字得到标识符时，进程可以指定关键字。然而，关键字不能标识特殊对象，比如用路径名标识文件，因为这个对象只能持续到下次系统重新启动。下一次，同样的关键字可能会用不同的标识符产生一个新对象。

关键字具有key_t类型，该类型甚至不必是算术类型，尽管在许多系统调用中它是以算术类型使用的，因此这是一种安全的假设。如果需要指定其值，可以采用如下方式简单地为程序中使用的关键字定义值：

```
#define MSGQ_KEY 1234
#define SEM_KEY 1235
#define SHM_KEY1 1236
#define SHM_KEY2 1237
```

之后任何把关键字MSGQ_KEY当作参数调用msgget的进程都会得到一个指向相同消息队列的标识符。

关键字的问题是：需要一些全面的管理方案为它们赋值，因为在应用之间会产生冲突。显然有冲突就无法工作，因此另一抽象层允许使用ftok系统调用从路径名中产生一个关键字。

ftok 产生System V IPC关键字

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

key_t ftok(
    const char *path,          /* pathname of existing file */
    int id                    /* desired key number */
);
/* Returns key on success or -1 on error (sets errno) */
```

事实上，ftok能够从同一路径中产生许多关键字；所需的关键字由id参数指定。该参数可能是字符，因为只使用它最低的8位。关键字也不能是零。例如，如果应用程序中需要4个关键字（一个消息队列、一个信号量集和两个共享存储段），那么可以用单个路径名（如tmp/myappkeys）和具有q、s、m和n值的第二个参数4次调用ftok（或其他4个所需的值）。路径名必须已经存在，因为ftok不能创建它。

回顾前面的内容，可以知道System V IPC对象不是文件而且没有信息节点。名字被传递给ftok的文件仅是为关键字产生提供一个全局名。而与文件的内容无关。尽管名称保持一致，也并不能保证是否解链文件和重新生成了相同的关键字，因此当安装应用程序或第一次运行时，必须创建文件并且让其放在一边直到卸载应用程序。

只要路径在相同的文件系统（见3.2.4节）中，就可以保证两条不同的路径能够产生两个不同的关键字。如果担心不同的应用可能会使用相同的关键字，那么可以在Xget调用中使用专门的关键字IPC_PRIVATE（将在7.5.1节进一步解释），这样在根本不使用ftok或专门关键字时可以保证有唯一的IPC对象。必须以某种方式公布这个最后的标识符以便所有需要它的进程都能得到它；实现这一点的一种方法是将其写入一个一些进程已经知道其文件名的文件。但是，在大多数情况下，也包括本书所有的例子，都假定不会出现ftok冲突。

总结如下：

- 可以通过标识符访问System V IPC对象。如何得到标识符并不重要——它可能是exec的一个参数，可以由消息传递，从文件读取，从系统调用（如msgget）返回，或其他某种方法获得。
- 如果需要（而且通常会这样），可以用一个关键字指向一个对象，关键字与标识符不同，即使在对象被销毁或重新创建时仍能保持不变。

- 由于管理关键字很难，所以可以利用ftok从路径名生成，事实上，你通常也会这么做。

关键字不唯一的可能性很小；创建IPC_PRIVATE对象是一个有可能可行的解决方法。

你可能会问自己，“既然UNIX已经有了一个完美的包含路径名和文件描述符的机制，为什么还要标识符/关键字/ftok成员呢？”稍后看一些使用System V IPC消息的示例代码，答案就更清楚了。那时会看到，服务器会把客户端的消息队列看成一个标识符，并立即使用它返回一个消息，没有寻找和打开信息节点的开销，这些是open所做的。

尽管如此，是否能设计一个更清楚的方法，该方法更有效且能使文件系统的其余部分更加紧密地结合起来。当查看POSIX IPC消息时会看到这种方法，尽管它本身有使用路径名方式的问题。

除了其复杂性和无规律性外，System V IPC方法的另一个缺点是因为标识符不是文件描述符，所以不能用select或poll（见4.2节）阻塞，例如，在消息准备好以前。我们已经在5.18节讲述了解决此问题的一种方法。

总之，现在System V IPC命令已经标准化了，不可能再对其作任何的改进。

7.4.3 System V IPC的所有权和权限

只要System V IPC对象是文件，就可以使用文件所用的权限系统，这里不必做过多的解释。但是，由于它们不是文件，所以它们有自己的系统。幸运的是，很多特性都与文件的相同。

权限通常由9位指定：所有者、组和其他用户所拥有的读、写和执行权限。然而，“执行”没有任何作用，因此没有使用那些位。当由Xget系统调用创建对象时会生成对象的权限。以后可以使用Xctl系统调用更改权限。

文件有一个属主用户ID和属主组ID。System V IPC对象也有这两个，另外还保存新建对象的用户ID和组ID。用Xctl可以改变所有者ID，不能改变新建对象ID。

当处理对象时，检查权限的算法可以使用有效用户ID和有效组ID，就像文件那样，并且仅当不能匹配任何用户ID和组ID时才使用“其他用户”权限位。然而，有效用户ID和有效组ID能匹配新建对象ID或所有者ID。

区分开的新建对象ID和所有者ID既允许一个管理员用户（如“dbmsadm”）成为消息队列的创建者，也允许他成为访问数据库文件的数据库服务进程的有效用户ID的创建者。权限稍小的用户（如“dbmsuser”）只能访问消息队列，不能访问文件。

struct ipc_perm—System V IPC权限的结构

```

struct ipc_perm {
    uid_t uid;           /* owner user-ID */
    gid_t gid;           /* owner group-ID */
    uid_t cuid;          /* creator user-ID */
    gid_t cgid;          /* creator group-ID */
    mode_t mode;         /* permission bits */
};

```

使用某个Xctl系统调用获得或设置权限时，可以使用如下结构：

7.4.4 System V IPC工具

因为System V IPC对象不是文件且没有信息节点，所以不能对它们使用像unlink或stat之类的系统调用。因此，既不能用rm删除它们，也不能使用ls命令列出它们。然而有两个专门处理System V IPC对象的命令：ipcrm用于删除对象；ipcs用于报告其状态。查阅[SUS2002]或系统文档可以看到这些命令的各种选项；这里仅概述基本的选项。

调用ipcrm时，可以使用一个或多个如下格式的参数对：

-XY

这里的X代表消息队列的q或Q、信号量集S或共享存储段M。在x代表q时，Y代表标识符；在X代表Q时，Y代表关键字。^⑥例如，命令

```
ipcrm -q 50
```

删除了标识符为50的消息队列。在删除文件时，实际上所做的是从目录中删除项；信息节点会保留，直到关闭了最后一个指向它的打开文件描述符，所以正在运行的进程不会受到过多的影响。然而，没有为System V IPC对象指定这样的行为，对象可能会立即消失，并对使用它的正在运行的进程造成严重破坏。因此，ipcrm仅用作没有负载时系统管理的一部分，或者已知那个对象没有在使用时才使用ipcrm。

在System V IPC中与ls功能相当的是ipcs。不带参数使用它时，显示如下内容：

```
IPC status from <running system> as of Wed Mar 12 15:04:06 MST 2003
T      ID      KEY      MODE      OWNER      GROUP
Message Queues:
q      50      0x1007b8c --rw-rw-rw-   marc sysadmin
Shared Memory:
m      0      0x500004b7 --rw-r--r--   root      root
m      102     0      --rw-rw-rw-   marc sysadmin
m      103     0      --rw-rw-rw-   marc sysadmin
m      104     0      --rw-rw-rw-   marc sysadmin
m      105     0      --rw-rw-rw-   marc sysadmin
Semaphores:
s      131072  0x1007b8d --ra-ra-ra-   marc sysadmin
s      131073  0      --ra-ra-ra-   marc sysadmin
```

注意有些关键字是零。这些是由带有IPC_PRIVATE参数的Xget创建的私有对象，而不是某个关键字。当标识符被直接传递给其他进程（例如消息中的数据），并且其他进程没有必要执行自己的Xget时才会这么做。在System V的SMI函数的消息队列实现中会看到这个（见7.5.3节）。（像7.4.2节中解释的那样，如果想要确保对象是唯一的，那么也可以使用IPC_PRIVATE。）

7.5 System V 消息队列

现在准备讲述System V 消息队列的系统调用的详细内容。

7.5.1 System V 消息队列的系统调用

前面已经概述了msgget的功能，下面是其细节：

```
msgget —— 得到消息队列标识符

#include <sys/msg.h>

int msgget(
    key_t key,           /* key */
    int flags           /* creation flags */
);
/* Returns identifier or -1 on error (sets errno) */
```

在7.4节中我们对System V IPC对象进行了一般性的讨论，msgget可以得到已存在的消息队列的标识符，该消息队列的关键字由第一个参数给定。如果flags参数设置成IPC_CREAT

^⑥ 无论如何那是标准。Linux（或者可能是GUN）采用不同的方式操作它，详细内容见系统文档。

标志，那么它就会在队列不存在的情况下创建它。在这种情况下，权限是从flags参数的低9位中获得的。创建者和所有者的用户ID和组ID从发起msgget的进程的有效ID中获得。

对于权限位，可以像使用open（见2.3节）一样使用相同的s_标志。但是要确认使用的是IPC_CREAT；不要误用O_CREAT！[⊖]如果队列已经存在，那么使用IPC_EXCL同样可以使msgget失败。

IPC_PRIVATE的key参数允许创建一个与特殊关键字无关的消息队列。每次用IPC_PRIVATE（不需要IPC_CREAT标志）调用msgget时都会得到不同的队列。这对那些希望用自己的消息队列作应答服务器的客户端进程来说是理想的；客户端进程把标识符传递给服务器，服务器随后利用它响应。客户端和服务端没有必要共享同一个关键字，这是很困难的，因为通常服务器事先并不知道可能会有哪些客户端。

可以用msgctl控制一个已存在的队列：

msgctl —— 控制消息队列

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

int msgctl(
    int msqid,          /* identifier */
    int cmd,           /* command */
    struct msqid_ds *data /* data for command */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

struct msqid_ds —— msgctl的结构

```
struct msqid_ds {
    struct ipc_perm msg_perm; /* permission structure */
    msgqnum_t msg_qnum;      /* number of messages currently on queue */
    msglen_t msg_qbytes;     /* maximum number of bytes allowed on queue */
    pid_t msg_lspid;        /* process ID of last msgsnd */
    pid_t msg_lrpid;        /* process ID of last msgrcv */
    time_t msg_stime;       /* time of last msgsnd */
    time_t msg_rtime;       /* time of last msgrcv */
    time_t msg_ctime;       /* time of last msgctl change */
};
```

下面是cmd参数的三个值：

IPC_RMID 移去与msqid相关的队列。有效用户ID必须是超级用户或等于队列的创建者用户ID或所有者用户ID。可以不使用data参数，它可以是NULL。

IPC_STAT 用队列的信息填充data指向的结构。

IPC_SET 通过由data指向的结构成员来设置队列的四个属性：

msg_perm.uid

msg_perm.gid

msg_perm.mode

msg_qbytes

除了只有超级用户能提高msg_qbytes的值外，所需权限与IPC_RMID要求的相同。

使用msgsnd和msgrcv可以分别实现把消息放入队列和从队列中获取消息。

⊖ 为什么设计System V IPC调用的人不和open使用相同的符号，而是建立他们自己的符号呢？因为追溯到那个时候（20世纪70年代中期），open不使用符号。因此问题应该是，为什么open不和System V IPC使用相同的符号呢？因为那些符号都以前缀IPC_开始，此外，贝尔实验室中主流的UNIX团队不喜欢IPC调用。

msgsnd——发送消息

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

int msgsnd(
    int msqid,           /* identifier */
    const void *msgp,   /* message */
    size_t msgsize,     /* size of message */
    int flags            /* flags */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

msgrcv——接收消息

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

ssize_t msgrcv(
    int msqid,           /* identifier */
    void *msgp,         /* message */
    size_t mtextsize,   /* size of mtext buffer */
    long msgtype,       /* message type requested */
    int flags            /* flags */
);
/* Returns number of bytes placed in mtext or -1 on error (sets errno) */
```

struct msg——msgsnd和msgrcv的典型结构

```
struct msg {
    long mtype;         /* message type */
    char mtext[MTEXTSIZE]; /* message text */
};
```

可以随意选择用于消息的结构形式，只要该结构的第一个成员是用于消息类型的long型。情况是这样的，当调用msgrcv时，可以把发送的消息分类，然后规定想要接收的消息的类型：

- 如果msgrcv中msgtype参数是0，那么不管类型是什么，得到的都将是队列中的第一个消息。
- 如果该参数大于0，那么得到的就是那种类型的第一个消息。
- 如果该参数小于0，那么得到的将是等于或小于msgtype绝对值的最低类型的第一个消息。这就是说，如果在队列中有类型为5、6和17的消息，而且指定的是-6，那么得到的将会是类型5的第一个消息。

如果不关心类型，那么发送时可以使用1（类型必须为非0），并且将msgrcv中的msgtype参数设为0。在使用类型时，往往需要建立一个优先级系统。多个服务器和客户端也可以使用一个队列，而且每个客户端都有自己的类型编号，但是这种组织事务的方法很笨拙。

msgsnd中的msgsize参数仅仅是该结构中mtext成员里的消息的大小，不是包含mtype成员的整个结构的大小。类似地，msgrcv返回的是mtext成员里的消息字节数，不是整个结构的大小。如果接收的消息的大小超过了mtextsize参数，那么就会产生错误，除非在标志参数中指定了MSG_NOERROR，在这种情况下会截掉超过限制的消息，而且根本毫无察觉。这通常不是理想的方法。

如果超过了队列中消息编号的限制或队列中的消息总数，msgsnd一般会阻塞。或者，可以在标志参数中指定IPC_NOWAIT（O_NONBLOCK的System V IPC版本）来返回-1并将errno设置成EAGAIN，以使其不阻塞。（在下一节有更多的限制。）

如果所需要的消息不存在，msgrcv一般会阻塞。或者，可以同msgsnd一样，通过设置IPC_NOWAIT使其不阻塞。因为消息队列不使用文件描述符，所以不能使用select或poll。

因此要尽量避免设计需要超过一个队列的等待。为了使内容清晰明了，可以使用单个队列代替不同的消息类型。如果必须等待多个队列，那么可以考虑用5.18中所讲的技术。

7.5.2 System V 消息队列的限制

下面是很重要的应用可能强烈反对的一些限制：

- 队列中所有消息大小的总数限制。通过msgqid_ds结构中的msg_qbytes成员，可以用msgctl系统调用访问此限制。
- 队列中消息个数的限制。
- 消息大小的限制。
- 队列数的限制。

超过前两个限制并不一定会出现严重的错误。正如前一节所讨论的那样，可以安排msgsnd阻塞或者返回一个已达到限制的指示。

超过其他三种限制就肯定会产生错误，并且除了重新配置内核之外，没有其他的方法可以解决。虽然通过实验可以相当容易地知道如何限制大小，但是要实际确定如何限制就不那么容易了（sysconf得不到这些限制）。例如，我在Solaris中碰到了40个消息的限制，但是那是所有队列的消息数的最大值。在FreeBSD上，它是20，但是看起来好像是对每个队列的限制。在Linux中没有限制。对于消息大小的最大值，在Solaris和FreeBSD中，可以达到2048字节，在Linux中能达到8192字节。^①

在自己的开发系统中，重新配置内核是可以的，但是如果想让一个客户在他自己的计算机上安装你的应用程序的话，这可能是不太现实的。更糟糕的是，客户可能有多个使用System V IPC消息队列的应用，因此配置指令可能会冲突。

下面是一些实际的建议：

- 要使消息尽可能简短——比如说，1024字节或低于此数。如果一定要传输更多的消息，那么可以采用共享存储器。共享存储器也避免了两次大开销的复制（从进程到内核，及从内核到其他进程）。
- 不要增加其他的限制。保持队列的数量较小，并且不要假定在队列中可以放置任何特殊数目的消息。
- 在应用程序的安装期间，运行测试软件以保证此限制是恰当的。如果必要的话，给客户准备如何重新配置系统的建议。

7.5.3 System V中SMI消息队列的实现

消息队列系统调用实际上很容易使用，下面将通过实现SMI接口来说明这一点。在进行下面的内容之前，可能需要重新复习一下7.3.1节，该节介绍了接口的内容。

与7.3.3节的FIFO消息相比，内部的消息队列是比较简单的：

```
typedef struct {
    SMIENTITY sq_entity;    /* entity */
    int sq_qid_server;     /* server identifier */
    int sq_qid_client;     /* client identifier (not used by server) */
    char sq_name[SERVER_NAME_MAX]; /* server name */
    struct client_id sq_client; /* client ID */
    size_t sq_msgsize;     /* msg size */
}
```

^① 我使用的Darwin版本是6.6的，它没有System V消息。

```

    struct smi_msg *sq_msg;          /* msg buffer */
) SMIQ_MSG;

```

需要注意的是，当准备返回消息时，为了降低开销，服务器不必保留每一个客户端的消息，因为跟随消息一起传送的标识符能够直接用来调用msgsnd。实际上，服务器甚至不需要使用sq_qid_client成员。

打开一个SMI消息队列也是非常简单的：

```

SMIQ *smi_open_msg(const char *name, SMIENTITY entity, size_t msgsize)
{
    SMIQ_MSG *p = NULL;
    char msgname[SERVER_NAME_MAX + 100];
    key_t key;

    ec_null( p = calloc(1, sizeof(SMIQ_MSG)) )
    p->sq_msgsize = msgsize + offsetof(struct smi_msg, smi_data);
    ec_null( p->sq_msg = calloc(1, p->sq_msgsize) )
    p->sq_qid_server = p->sq_qid_client = -1;
    p->sq_entity = entity;
    if (strlen(name) >= SERVER_NAME_MAX) {
        errno = ENAMETOOLONG;
        EC_FAIL
    }
    strcpy(p->sq_name, name);
    mkmsg_name_server(p, msgname, sizeof(msgname));
    (void)close(open(msgname, O_WRONLY | O_CREAT, 0));
    ec_negl( key = ftok(msgname, 1) )
    if (p->sq_entity == SMI_SERVER) {
        if ((p->sq_qid_server = msgget(key, PERM_FILE)) != -1)
            (void)msgctl(p->sq_qid_server, IPC_RMID, NULL);
        ec_negl( p->sq_qid_server = msgget(key, IPC_CREAT | PERM_FILE) )
    }
    else {
        ec_negl( p->sq_qid_server = msgget(key, 0) )
        ec_negl( p->sq_qid_client = msgget(IPC_PRIVATE, PERM_FILE) );
    }
    return (SMIQ *)p;
}

EC_CLEANUP_BGN
if (p != NULL)
    (void)smi_close_msg((SMIQ *)p);
return NULL;
EC_CLEANUP_END
}

static void mkmsg_name_server(const SMIQ_MSG *p, char *msgname,
                             size_t msgname_max)
{
    snprintf(msgname, msgname_max, "/tmp/smimg-%s", p->sq_name);
}

```

在/tmp目录中为每一个唯一的服务器名使用了一个文件；函数mkmsg_name_server用于构造路径名。如果那个文件实际上已经不存在，那么在调用ftok的前一行代码会创建它。这里我们希望服务器能够以一个新队列开始，这样如果消息队列在创建之前已经存在，服务器则可以清除它。（在你自己的应用中你也许不会那么友好——你会关心正在运行的客户端在做什么，是否真正需要一个新队列，或者想要对已存在的消息做更多的操作而不只是简单地抛弃它们。）如果服务器已经开始工作，客户端就可以访问服务器队列，为自己创建一个私有队列。

下面是smi_close_msg:

```
bool smi_close_msg(SMIQ *sqp)
{
    SMIQ_MSG *p = (SMIQ_MSG *)sqp;

    if (p->sq_entity == SMI_SERVER) {
        char msgname[FILENAME_MAX];

        (void)msgctl(p->sq_qid_server, IPC_RMID, NULL);
        mkmsg_name_server(p, msgname, sizeof(msgname));
        (void)unlink(msgname);
    }
    else
        (void)msgctl(p->sq_qid_client, IPC_RMID, NULL);
    free(p->sq_msg);
    free(p);
    return true;
}
```

接下来是smi_send_getaddr_msg和smi_send_release_msg:

```
bool smi_send_getaddr_msg(SMIQ *sqp, struct client_id *client,
void **addr)
{
    SMIQ_MSG *p = (SMIQ_MSG *)sqp;

    if (p->sq_entity == SMI_SERVER)
        p->sq_client = *client;
    *addr = p->sq_msg;
    return true;
}

bool smi_send_release_msg(SMIQ *sqp)
{
    SMIQ_MSG *p = (SMIQ_MSG *)sqp;
    int qid_receiver;

    p->sq_msg->smi_mtype = 1;
    if (p->sq_entity == SMI_SERVER)
        qid_receiver = p->sq_client.c_id1;
    else {
        qid_receiver = p->sq_qid_server;
        p->sq_msg->smi_client.c_id1 = p->sq_qid_client;
    }
    ec_neg1(msgsnd(qid_receiver, p->sq_msg,
        p->sq_msgsize - sizeof(p->sq_msg->smi_mtype), 0))
    return true;

    EC_CLEANUP_BGN
        return false;
    EC_CLEANUP_END
}
```

在smi_send_release_msg中, 如果客户端正在发送, 则它已经有了服务器的标识符, 并且已经把标识符放到了消息的私有队列中。如果服务器正在发送, 则它会利用该标识符从SMIQ_MSG结构发送, 该结构是通过前面对smi_send_getaddr_msg的调用保存的。现在就能看出为什么我们把消息类型放在smi_msg结构的开始了: 它能够允许在调用msgsnd和msgrcv中直接使用该结构。需要注意, 向msgsnd传送的大小也仅是数据部分, 而不是整个

结构。

最后，下面是smi_receive_getaddr_msg和smi_receive_release_msg，这些比发送操作的那对要简单：

```
bool smi_receive_getaddr_msg(SMIQ *sqp, void **addr)
{
    SMIQ_MSG *p = (SMIQ_MSG *)sqp;
    int qid_receiver;
    ssize_t nrcv;

    if (p->sq_entity == SMI_SERVER)
        qid_receiver = p->sq_qid_server;
    else
        qid_receiver = p->sq_qid_client;
    ec_negl( nrcv = msgrcv(qid_receiver, p->sq_msg,
        p->sq_msgsize - sizeof(p->sq_msg->smi_mtype), 0, 0) )
    *addr = p->sq_msg;
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

bool smi_receive_release_msg(SMIQ *sqp)
{
    return true;
}
```

在SMIQ-MSG结构体中，服务器和客户端手边都有它们自己的队列标识符。

7.5.4 评价System V消息队列

此SMI实现阐明了System V消息队列最好的部分：进程可以向仅有标识符指定的队列发送消息，没有额外的系统调用访问队列的开销，并且不用为保持消息是原子的而费心。这里虽然看不到，但实际上即使多个接收者从同一个队列接收消息也是没有问题的（对FIFO来说，它们是不安全的）。

一个缺点是，没有充分地规定各种限制条件（若规定了最小值会更好），并且在运行时查询也是笨拙的。然而，对所有IPC的机制而不是套接字的最坏的实情是：消息只能在单个机器上发送。对于今天的应用来说，这种限制往往会带来很大的不便。

本书的第1版曾提出过这样一个建议：

[System V消息队列是]复杂的、文档不完全的、不可移植的。应尽量避免应用它们！

现在根据2004年的实际情况重新考虑这个问题，它们看起来不再很复杂，因为有了套接字、线程和许多更复杂的其他特征。文档虽然仍不完全，但不再影响它们的普及。它们肯定是可移植的——它们遵从SUS，并且似乎在所有的主要系统中都可以实现（希望不久在Darwin上也可以），因此，只要知道了它们的局限性，就没有理由拒绝使用它们。

7.6 POSIX IPC

本节讲述POSIX IPC应用于消息、信号量和共享存储器的一些一般内容，此外在7.7节、7.10节和7.14节也涉及了这些内容。

7.6.1 POSIX IPC历史

随着POSIX 1003.1b-1993 (缩写为POSIX1993) 成为实时系统扩展的一部分, POSIX IPC被引入进来。这里描述的POSIX消息队列的系统调用构成了POSIX1993的消息传递选项, 而且还是POSIX2001的可选项。相反, System V IPC始终没能成为POSIX的组成部分, 现在仍托管在被认证的Open Group UNIX系统上。

历史上, System V IPC是非标准的, 但却得到了广泛应用; 而POSIX IPC成为标准(可适用)已经10年了, 但在很多UNIX系统上仍不可用, 如著名的Linux、FreeBSD和Darwin。在理论上, POSIX IPC比System V IPC更便于移植; 实际上却恰恰相反。

更糟的是, 即使实现了POSIX IPC, 对关键的实时应用来说, 其效率仍然是不够高的。标准对性能水平没有特别的要求, 除了相容性要求外, 一些OS经销商对资源不提出任何目标。

因此, 就像即将看到的, POSIX IPC系统调用在许多方面都比System V IPC的强, 功能更强大, 设计更明确, 但在应用中你可能不使用它们。

7.6.2 POSIX IPC命名

回顾7.4.2节可知, System V IPC使用关键字指定对象, 并增加了一个用ftok系统调用从路径名生成关键字的便捷方案。POSIX IPC使用了一个更加简单的方案, 使用名字(字符串)代替关键字。

名字必须和路径名遵循相同的规则, 标准要求如果名字以斜线开头, 那么所有参考该名字的东西所指的都是同一个对象。如果名字没有以斜线开头, 则没有那样的要求, 但这种方法有一些问题:

- 并没有要求具有特定名字的文件必须出现在文件系统中。POSIX IPC对象, 如System V IPC对象, 并不是文件。为了效率才这样的——文件系统代表许多庞大的体系, POSIX IPC的快速实现是不使用它的, 例如, 使用内存中的哈希表查找对象而不使用文件系统目录树。
- 在大多数UNIX系统中, 一般用户不能在根目录下进行写操作, 因为如果实现确实创建了符合该名字的文件, 可能会产生问题。解决该问题的一个可能的方法是管理员为每个一般应用至少首先建立一个子目录(例如/ipc), 但请看下一点。
- 斜线的解释与第一个不同, 它是依赖于实现的。有些实现可能会把它们当作一般字符, 有些可能会把它们当作目录分隔符, 还有一些(如Solaris)可能会完全禁止它们。因此, 为了便于移植, 除了开头的斜线外, 不应该再有其他的斜线了。这个规则排除了上一点的解决方法。

那么, 怎么做呢? 可以写一段小的初始化程序, 该程序仅创建应用所需要的所有对象(使用的名字中仅有开头的斜线)并设置它, 以便有效用户ID是超级用户或无论什么用户都可以写入根目录。开发时要么使用初始化程序, 要么运行不创建实际文件的系统。本书的例子是在Solaris上开发的, 该系统并不创建文件, 因此显示在根目录中的路径名并不是真的在那儿, 并且所有的工作都很正常。

一个可选择的方法是各种系统定义(`#ifdef`)自己的代码, 使用系统支持的命名方案。但这个方法通常不好, 因为无论包括系统多么小的集合, 毫无疑问总有一天都会不足, 然后必须更改代码以便包含新系统。新修改的代码必须返回来工作在基础代码上, 不可避免地会带来支持和维护上令人头疼的事。

然而另外一个想法是把POSIX IPC命名方式放在运行时读的配置文件中。然后选择合适的方案便成了安装时的一个选项, 而不再是编译时的选项了。这可能是可以的, 但仍增加了一个复杂应用程序不能正确安装的许多方式, 这是不希望的。毕竟使用关键字和ftok的System V IPC方案并不那么笨拙。

7.6.3 POSIX IPC 特征检测宏

像1.5.4节详细解释的那样，SUS和早期的POSIX标准都提供了宏，在编译期间使用该宏可以检查可选的特征项是存在还是不存在，或者是否必须编写运行时检测代码。不幸的是，如所希望的那样准确实现这些宏的系统通常有可选的特征项（这意味着可能已经跳过了该测试），而没有可选特征的系统也不能正确地实现这些宏。因此，在实际中，测试这些宏并没有带来方便。

但是，如果想尝试一下，就仔细地阅读第1章所叙述的关于它们通常怎样工作的内容，然后参考[SUS2002]或相关的POSIX标准以查看哪些特定宏需要检测。

本书的例子中通常不使用宏测试。它们在支持可选特征的系统上编译和运行，在其他系统上无法编译。

7.6.4 POSIX IPC工具

对System V而言，最方便的工具是`ipcs`和`ipcrm`（见7.4.4节）。POSIX IPC系统没有任何工具，因此本节很短。如果想检测错误应用遗留下的消息队列、信号量或共享存储器段的一些东西，就太糟了。

7.7 POSIX消息队列

本节将解释POSIX消息队列，并说明如何使用它们实现SMI。

7.7.1 POSIX消息队列的系统调用

事实证明，名字问题是使用POSIX消息队列唯一的真正困难。其余的都很容易。首先，`mq_open`可以打开消息队列。

尽管`mq_open`创建的对象不能轻易地被当作文件对待，而且它返回的是消息队列描述符，而不是文件描述符，但`mq_open`和`open`的行为还是相似的。遗憾的是，不能够对消息队列描述符使用`select`或`poll`，这和在System V消息队列中见到的缺点一样。尽管如此，当消息可用时，可以使用信号进行提示；见下面的`mq_notify`。

mq_open ——打开消息队列

```
#include <queue.h>

mqd_t mq_open(
    const char *name,      /* POSIX IPC name */
    int flags,             /* flags (excluding O_CREAT) */
);
/* Returns message-queue descriptor or -1 on error (sets errno) */

mqd_t mq_open(
    const char *name,      /* POSIX IPC name */
    int flags,             /* flags (including O_CREAT) */
    mode_t perms,         /* permissions */
    struct mq_attr *attr   /* attributes (or NULL) */
);
/* Returns message-queue descriptor or -1 on error (sets errno) */
```

struct mq_attr ——mq_open, mq_getattr和mq_setattr的结构

```
struct mq_attr {
    long mq_flags;        /* flags */
    long mq_maxmsg;      /* max number of messages */
    long mq_msgsize;     /* max message size */
    long mq_curmsgs;     /* number of messages currently queued */
};
```

flags参数使用的宏和open（例如，O_CREAT）使用的一样。并不像msgget使用的宏（如IPC_CREAT）那么特殊：

- 必须指定O_RDONLY、O_WRONLY或O_RDWR中的某一个，这取决于是想只接收消息，还是只发送消息，或者两者都要。
- 如果想要在队列不存在时，创建队列，那么可以指定O_CREAT。在这种情况下，要提供4个参数，就像对照表中的第二个表所示的那样。
- 如果想要在队列已经存在时函数失效，那么指定O_EXCL。
- 如果想消息队列描述符非阻塞，那么指定O_NONBLOCK，这意味着如果队列满或空，mq_send和mq_receive会各自返回-1，并将errno设定为EAGAIN。默认行为是阻塞。

当设定了O_CREAT，并确实创建了队列时，perms参数的解释和对文件或System V消息队列的权限相似，带有通常的S_标志（见2.3节）。读权限和写权限是指各自接收和发送消息的能力，而执行权限并不代表任何东西。和文件一样，队列也有用户ID和组ID，而且可以用创建队列的进程的有效ID设置它们。随后有效用户ID和有效组ID的所有者、组和其他用户位的相互作用和文件是一样的。

同样，如果规定了O_CREAT，创建了队列，并且attr参数是非NULL，那么mq_maxmsg和mq_msgsize这两个属性由所提供的mq_attr结构设置。这是队列能够容纳的最大消息个数和单个消息大小的最大值。标准并没有为这两个属性单独指定限制，但如果有足够的空间，mq_open将返回-1，并把errno设置为ENOSPC。大多数的实现用链表实现队列，因此不管数多大，mq_open都不会用光空间，因为新队列是空的。如果attr参数是NULL，这两个属性会被设置成具体实现定义的值。

如果在读前面几段时，你继续认为mq_open和open完全相同，那么只要假定它返回的不是文件描述符，可能就是对的。

使用mq_close关闭打开的消息队列：

mq_close —— 关闭消息队列

```
#include <mqqueue.h>

int mq_close(
    mqd_t mqd          /* message-queue descriptor */
);
/* Returns zero on success or -1 on error (sets errno) */
```

和System V消息队列一样，POSIX消息队列会持续存在直到它们被移去或内核重新启动。可以利用一个unlink那样的调用移去POSIX消息队列：

mq_unlink —— 删除消息队列

```
#include <mqqueue.h>

int mq_unlink(
    const char *name    /* POSIX IPC name */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

像unlink一样，mq_unlink会使名字马上消失，但队列的实际删除被延迟了，直到关闭了所有打开的消息队列描述符之后才能删除队列。（回想System V消息队列，使用msgctl删除队列是立刻进行的，但可能会中断。）因此，如果需要，一旦所有需要队列的进程已经打开

了它，便可以使用mq_unlink，就像有时处理临时文件那样。

好了，准备好发送消息了吗？可以调用mq_send来发送消息，你可能已经猜到了：

mq_send——发送消息

```
#include <mqqueue.h>

int mq_send(
    mqd_t mqd,          /* message-queue descriptor */
    const char *msg,    /* message */
    size_t msgsize,    /* size of message */
    unsigned priority   /* priority */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

mq_send把msgsize大小的消息放置在队列上，该消息被定位在所有优先级(priority)较低的消息的前面，但在具有相同priority的消息的后面（换句话说，就像所期望的那样）。优先级至少从0到31；可以使用sysconf（见1.5.5节）检索实际的最大值。

如上所讲，如果队列是满的，那么mq_send会阻塞，除非设置了O_NONBLOCK标志。

接收消息的调用当然是mq_receive：

mq_receive——接收消息

```
#include <mqqueue.h>

ssize_t mq_receive(
    mqd_t mqd,          /* message-queue descriptor */
    char *msg,          /* message buffer */
    size_t msgsize,    /* size of message buffer */
    unsigned *priorityp /* returned priority or NULL */
);
/* Returns size of message or -1 on error (sets errno) */
```

mq_receive获得最高优先级消息中时间最久的那个，因为定义了mq_send的方式，所以可以说成是队列中的第一个消息。

msgsize参数有点棘手：可能和你所期望的一样，它是由msg参数指向的缓冲区的大小，但它必须至少和mq_msgsize属性大小一样（见上面的mq_open），否则mq_receive将失败，即使最前面的消息可以小得足以满足要求。这确实是一个极好的设计选择：能立即捕获非常小的缓冲区的问题，不需要用恰好正确的测试数据来揭露漏洞。返回值是接收到的消息的实际大小。

如果priorityp参数是非NULL的，它会获得接收到的消息的优先级。没有办法来恰当地查询消息，只能从队列中移去接收到的消息。如果队列是空的，mq_receive会阻塞，除非设置了O_NONBLOCK。mq_send和mq_receive的变体允许被阻塞的调用超时：

mq_timedsend——用timeout发送消息

```
#include <mqqueue.h>
#include <time.h>

int mq_timedsend(
    mqd_t mqd,          /* message-queue descriptor */
    const char *msg,    /* message */
    size_t msgsize,    /* size of message */
    unsigned priority,  /* priority */
    const struct timespec *tmout /* timeout */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

mq_timedreceive ——用timeout接收消息

```
#include <mqqueue.h>
#include <time.h>

ssize_t mq_timedreceive(
    mqd_t mqd,           /* message-queue descriptor */
    char *msg,          /* message buffer */
    size_t msgsize,     /* size of message buffer */
    unsigned *priority, /* returned priority or NULL */
    const struct timespec *tmout /* timeout */
);
/* Returns size of message or -1 on error (sets errno) */
```

仅当调用发生了阻塞而且清除了O_NONBLOCK时，超时才会起作用。当超时时间到时，函数会返回-1，同时将errno设置为ETIMEDOUT。这两个函数是超时选项（_POSIX_TIMOUTS）的一部分，也是SUS3中新增的内容。

前面说过，对于一个以上的消息队列或消息队列与其他会阻塞的东西（诸如终端或网络连接）的结合来说，处理起来会很困难，因为不能够使用select或poll测试消息队列描述符。尽管如此，当消息到达时，可以安排使用信号进行通知，然后调用mq_receive，不必担心阻塞。（另一个解决方法见5.18节。）使用mq_notify设置。

mq_notify ——注册或注销消息通知

```
#include <mqqueue.h>

int mq_notify(
    mqd_t mqd,           /* message-queue descriptor */
    const struct sigevent *ep /* notification */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

对于注册的进程或线程，当消息到达mqd指派的空队列时，它会得到一个信号，除非一个或多个进程或线程已经在mq_receive中阻塞，此时其中的一个mq_receive会返回。

仅可以注册一个进程或线程；试图注册第二个进程或线程会导致错误。当信号被发出或当调用第二个参数为NULL的mq_notify时，进程或线程会成为未注册的。

非空队列中没有消息到达的通知，因此mq_notify单独使用时并不能保证每次消息到达时，进程都会得到一个信号。例如，当消息到达空队列时，它可能会得到一个信号，然而在接收第一个消息前，第二个消息可能已经到达了，此时就不会引发通知了，因为队列不是空的。因此，应用需要做如下的一些工作：

- 调用mq_notify之后，带有O_NONBLOCK设置重复调用mq_receive，直到队列为空。
- 当信号到达时，立即再次调用mq_notify，然后进行上一步骤的操作。

发送了什么样的信号以及通知的其他属性都由sigevent结构传递给mq_notify的内容决定，如9.5.6节所解释的那样。

最后，得到并设置消息队列的属性：

mq_getattr ——得到消息队列属性

```
#include <mqqueue.h>

int mq_getattr(
    mqd_t mqd,           /* message-queue descriptor */
    struct mq_attr *attr /* attributes */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

mq_setattr——设置消息队列属性

```
#include <mqqueue.h>

int mq_setattr(
    mqd_t mqd,           /* message-queue descriptor */
    const struct mq_attr *attr, /* new attributes */
    struct mq_attr *oldattr /* old attributes if not NULL */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

`mq_getattr`根据队列的当前状态填充该结构（和`mq_open`一起被显示）。最有用的属性可能是成员`mq_curmsgs`，它包含了队列中消息的当前数量。

`mq_setattr`用于设置或清除`mq_flags`成员中的`O_NONBLOCK`标志；依据标准，它对其他任何标志都没有影响。（实现中可能要设置一些其他的不可移植的标志。）因此，对`O_NONBLOCK`标志，可以在消息队列描述符上使用`mq_setattr`，就如对文件描述符使用`fcntl`。如果`oldattr`非NULL，则可用它来获得返回的旧属性。

7.7.2 POSIX消息队列的SMI实现

现在，来看SMI函数的第三个实现。首先，下面是一个内部消息队列，其几乎和FIFO中的一样，因为它们处理的问题都是相似的：

```
#define MAX_CLIENTS 20

typedef struct {
    SMIENTITY sq_entity;           /* entity */
    mqd_t sq_mqd_server;          /* server message-queue descriptor */
    char sq_name[SERVER_NAME_MAX]; /* server name */
    struct client_info {          /* Client uses only sq_clients[0] */
        mqd_t cl_mqd;            /* client message-queue descriptor */
        pid_t cl_pid;            /* client process ID */
    } sq_clients[MAX_CLIENTS];
    struct client_id sq_client;    /* client ID */
    size_t sq_msgsize;            /* msg size */
    struct smi_msg *sq_msg;        /* msg buffer */
} SMIQ_MQ;
```

和FIFO一样，为了避免每个消息都打开一个客户端消息队列的开销，服务器将跟踪每个它知道的客户端，而且对每个已经打开的队列仅查找已经保存的消息队列描述符。在许多实际的服务器中，这并不真的是一个负担，因为无论如何它都几乎总是想要跟踪客户端。客户端只用数组的第一个元素存储它自己的消息队列描述符，这里我们不考虑所浪费的空间。

有两个函数可以把SMI服务器名和客户端进程ID转换成POSIX IPC名：

```
static void make_mq_name_server(const SMIQ_MQ *p, char *mqname,
    size_t mqname_max)
{
    snprintf(mqname, mqname_max, "/smimq-s%s", p->sq_name);
}

static void make_mq_name_client(pid_t pid, char *mqname,
    size_t mqname_max)
{
    snprintf(mqname, mqname_max, "/smimq-c%d", pid);
}
```

这些名字看起来好像是在根目录下，但在Solaris中它们就不在根目录下，如7.6.2中解释的那样。

接下来介绍一个服务器从客户端进程ID获得客户消息队列描述符的函数，该进程ID是客户端在消息中发送给服务器的：

```
static mqd_t get_client_mqd(SMIQ_MQ *p, pid_t pid)
{
    int i, avail = -1;
    char mqname[SERVER_NAME_MAX + 100];

    for (i = 0; i < MAX_CLIENTS; i++) {
        if (p->sq_clients[i].cl_pid == pid)
            return p->sq_clients[i].cl_mqd;
        if (avail == -1 && p->sq_clients[i].cl_pid == 0)
            avail = i;
    }
    errno = ECONNREFUSED;
    ec_negl( avail )
    p->sq_clients[avail].cl_pid = pid;
    make_mq_name_client(pid, mqname, sizeof(mqname));
    ec_negl( p->sq_clients[avail].cl_mqd = mq_open(mqname, O_WRONLY) )
    return p->sq_clients[avail].cl_mqd;
}

EC_CLEANUP_BGN
    return (mqd_t)-1;
EC_CLEANUP_END
}
```

该函数首先在数组中搜索并查看是否已经发现了客户端，在此情况下，它已经有了消息队列描述符。如果没有，它将为写操作打开队列，并为下次操作保存该描述符。

现在准备讲述第一个SMI函数——smi_open_mq：

```
static pid_t my_pid;

SMIQ *smi_open_mq(const char *name, SMIENTITY entity, size_t msgsize)
{
    SMIQ_MQ *p = NULL;
    char mqname[SERVER_NAME_MAX + 100];
    struct mq_attr attr = {0};

    my_pid = getpid();
    ec_null( p = calloc(1, sizeof(SMIQ_MQ)) )
    p->sq_msgsize = msgsize + offsetof(struct smi_msg, smi_data);
    ec_null( p->sq_msg = calloc(1, p->sq_msgsize) )
    p->sq_entity = entity;
    p->sq_mqd_server = p->sq_clients[0].cl_mqd = (mqd_t)-1;
    if (strlen(name) >= SERVER_NAME_MAX) {
        errno = ENAMETOOLONG;
        EC_FAIL
    }
    strcpy(p->sq_name, name);
    make_mq_name_server(p, mqname, sizeof(mqname));
    attr.mq_maxmsg = 100;
    attr.mq_msgsize = p->sq_msgsize;
    if (p->sq_entity == SMI_SERVER) {
        if (mq_unlink(mqname) == -1)
            ec_cmp( errno, ENOSYS )
            ec_negl( p->sq_mqd_server = mq_open(mqname, O_RDONLY | O_CREAT,
                PERM_FILE, &attr) )
    }
```

```

    }
    else {
        ec_negl( p->sq_mqd_server = mq_open(mqname, O_WRONLY) )
        make_mq_name_client(my_pid, mqname, sizeof(mqname));
        ec_negl( p->sq_clients[0].cl_mqd = mq_open(mqname,
            O_RDONLY | O_CREAT, PERM_FILE, &attr) )
    }
    return (SMIQ *)p;

EC_CLEANUP_BGN
    if (p != NULL)
        (void)smi_close_mq((SMIQ *)p);
    return NULL;
EC_CLEANUP_END
}

```

初始化内部SMIQ队列之后，利用mq_attr结构的两个参数来初始化：一个是参数要满足队列中只能有100个消息，另一个参数是被传递的消息的大小，它是mq_attr的第三个参数。然后如果是服务器，会抛弃所有现存的服务器队列，接着创建新的服务器队列。不必为每个客户端都做这个事情，尽管也许应该这样，这是因为客户端的进程ID被嵌入到了队列名中，而不可能重新被使用。

如下的关闭函数可以完成所期望的工作：

```

bool smi_close_mq(SMIQ *sqp)
{
    SMIQ_MQ *p = (SMIQ_MQ *)sqp;
    char msgname[SERVER_NAME_MAX + 100];

    if (p->sq_entity == SMI_SERVER) {
        make_mq_name_server(p, msgname, sizeof(msgname));
        (void)mq_close(p->sq_mqd_server);
        (void)mq_unlink(msgname);
    }
    else {
        make_mq_name_client(my_pid, msgname, sizeof(msgname));
        (void)mq_close(p->sq_mqd_server);
        (void)mq_unlink(msgname);
    }
    free(p->sq_msg);
    free(p);
    return true;
}

```

接下来介绍smi_send_getaddr_mq及smi_send_release_mq:

```

bool smi_send_getaddr_mq(SMIQ *sqp, struct client_id *client,
    void **addr)
{
    SMIQ_MQ *p = (SMIQ_MQ *)sqp;

    if (p->sq_entity == SMI_SERVER)
        p->sq_client = *client;
    *addr = p->sq_msg;
    return true;
}

bool smi_send_release_mq(SMIQ *sqp)
{
    SMIQ_MQ *p = (SMIQ_MQ *)sqp;

```

```

mqd_t mqd_receiver;

if (p->sq_entity == SMI_SERVER)
    ec_negl( mqd_receiver = get_client_mqd(p, p->sq_client.c_id1) )
else {
    mqd_receiver = p->sq_mqd_server;
    p->sq_msg->smi_client.c_id1 = my_pid;
}
ec_negl( mq_send(mqd_receiver, (const char *)p->sq_msg,
    p->sq_msgsize, 0) )
return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

```

与前面的FIFO和System V消息队列一样，smi_send_getaddr_mq仅保存客户端ID（如果服务器调用它）以及返回缓冲区地址。真正的工作是在smi_send_release_mq中完成的。服务器调用查找函数get_client_mqd获得消息队列描述符。其参数通过smi_send_getaddr_mq保存在SMIQ_MQ结构中。

客户端已经拥有了服务器的描述符，而且把进程ID放到了消息中。（消息队列描述符不能在进程间传递，而System V中消息队列的标识符却可以。）

接收方的一对函数更简单，因为接收方已经拥有了指向消息队列的描述符：

```

bool smi_receive_getaddr_mq(SMIQ *sqp, void **addr)
{
    SMIQ_MQ *p = (SMIQ_MQ *)sqp;
    mqd_t mqd_receiver;
    ssize_t nrcv;

    if (p->sq_entity == SMI_SERVER)
        mqd_receiver = p->sq_mqd_server;
    else
        mqd_receiver = p->sq_clients[0].cl_mqd;
    ec_negl( nrcv = mq_receive(mqd_receiver, (char *)p->sq_msg,
        p->sq_msgsize, NULL) )
    *addr = p->sq_msg;
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

bool smi_receive_release_mq(SMIQ *sqp)
{
    return true;
}

```

7.7.3 评价POSIX消息队列

POSIX消息队列系统调用的接口比System V调用的更加清楚了，且更容易与UNIX的其他部分结合，但它们依然使用自己的描述符而不是文件描述符，因此不能使用select和poll。为了补救，有一个通知特征，但是在第9章会看到，它使用起来很困难，因为它是以很难使用的信号为基础的。

POSIX消息队列（实际上是所有的POSIX IPC）最主要的问题是不能被广泛地使用，但是，这当然不是对设计的批评。

表7-1更加详细地对POSIX消息队列和System V消息队列做了比较：

表7-1 POSIX消息队列和System V消息队列

标准	POSIX	System V
标准化了吗？	是	是
必须托管在已认证的UNIX系统上吗？	不	是
在所有主要的UNIX系统上可用吗？	不	是
消息有限制吗？	很少；可设置的	很多；只有一个可设置；难于管理
线程安全吗？（依据SUS3）	是	是
消息有优先级吗？	是	是
接收最高优先级以外的消息吗？	不	是
通知吗？	是	不
使用文件描述符吗？	不	不
效率高吗？	取决于具体实现	取决于具体实现
每个消息需要两次用户-内核复制吗？*	是	是
队列名可移植吗？	是	不

*这是指数据从用户空间复制到内核空间，或者再从内核空间复制回用户空间。

既然有优点，也有缺点，为什么在System V消息都存在了10多年的情况下，POSIX实现组成员还发明一个新的消息系统呢？而且既然这么做了，结果为什么却没有更好呢？

下面是我的答案：

- 当POSIX组成员确定对已存在的System V方法有许多吹毛求疵的意见时，需要一个新标准的主要原因是，许多已存在的、有System V实现的系统不适合实时应用，而且在相同的接口下有两个同时工作的实现是困难的，因此实现需要一个新的开始。
- 但是，在POSIX进程内没有一个实际的方法要求某种特殊级别的性能，因此在许多系统上，而不是大部分的系统上，两者（甚至有两个系统时）的效率基本相同。
- POSIX IPC名字的不可移植性是由不规范造成的，为了允许更广泛的实现集合需要它，其中包括没有任何文件系统查询的快速内存队列。
- POSIX组成员不对实现的不足负责。最有可能负责的是由20世纪90年代中期的许多应用开发者，即使不是大多数，因为应用程序的开发者把注意力集中在网络应用程序上，为此使用了套接字（见第8章），但对处理非网络消息的替代方法没有兴趣。因此，如Sun一样投资较多的设备实现POSIX可选包，而BSD和Linux这样零星资金投资设备的团体觉得有更大的压力做这件事情。

7.8 关于信号量

本节将讲述信号量的一般特性并介绍如何用文件和消息来实现它们。在7.9和7.10两节中将分别讲述System V信号量的系统调用和POSIX信号量的系统调用。

7.8.1 基本信号量的用法

在5.17.3节曾涉及过互斥。更一般地说，信号量是阻止两个或多个进程或线程同时访问共享资源的计数器。尽管这仅仅是建议性的，但是如果进程或线程在访问共享资源前不检查信号量，就可能会出现混乱。

在UNIX里，术语“信号量”通常应用于由同步进程的信号量系统调用操纵的对象，术语“互斥”通常指用于使线程同步的更轻量级的对象，该内容在5.17.3节已经讨论过。本节只讨论信号量。

二元信号量只有两个状态：锁定和非锁定。一般信号量有无穷多（至少非常多）个状态。它是一个计数器，当获得（“加锁”）时减一，而当它释放（解锁）时是加一。如果它是0，那么尝试获取它的进程必须等待另一个进程增加其值，它永远不会变为负值。（如果信号量只能具有两个值，0和1，那么它就是二元信号量。）一般信号量通常由两个操作来访问，可以抽象地称这两个操作为semwait和sempost（有时称作P和V[⊖]）。可以尝试用C语言来编写它们：

```
void semwait(int *sem)
{
    while (*sem <= 0)
        ; /* do nothing */
    (*sem)--;
}

void sempost(int *sem)
{
    (*sem)++;
}
```

必须通过调用sempost来初始化信号量；否则信号量就得不到任何初始信号量值。例如，如果用信号量计算空闲缓冲区的数量，且最初有5个缓冲区，那么就应该调用5次sempost以便实现初始化。或者说，可以将信号量变量设置为5。

在编写semwait和sempost时曾遇到过一些麻烦，这里必须说一下，在下面三种情况下，它们是不能正常工作的：

- 由sem指向的信号量变量通常不在进程间共享，这些进程有不同的数据段。（尽管在共享内存中，是可以的）。
- 函数不会自动执行——内核可以在任何时候中断进程。可能会出现如下情况：进程1完成了semwait中的while循环，并在对信号量减1之前被中断；进程2进入semwait，发现信号量为1，完成它的while循环，且减信号量到0；进程1重新开始对信号量减1，此时信号量的值就成了-1（非法值）。
- 如果sem为0，semwait将会处于叫作busy-wait的状态。这是一个使用CPU十分不明智的方法。

因此，不能仅仅在用户空间内编写semwait和sempost。必须由内核提供信号量，这样可以实现在进程间共享数据，可以执行原子操作，而且当进程阻塞时，可以将CPU让给准备好的进程。

7.8.2 用文件和消息实现信号量

在2.4.3节，已经讲述了如何使用open来产生一个原始的二元信号量。对于访问共享资源次数很少的进程，这种方法是可行的，例如对于写入邮箱文件的邮件程序。但对任务很重的用途来说，这种开销是巨大的。因此需要找到只需花费少量时间就可以检查和设置的信号量。

消息队列也可以被用作信号量：发送操作向队列中添加消息，发送操作相当于sempost；接收操作从队列中移除消息，该操作相当于semwait。当队列为空时，接收操作会阻塞，这种情况相当于信号量为0。

⊖ P和V是E. W. Dijkstra使用的荷兰惯用语proberen te verlagen（“试图减少”）和单词verhogen（“增加”）的缩写。有些书上说P代表prolanger，但那并不是一个单词，甚至连荷兰语都不是，它是整个短语的缩写。

尽管如此，任何支持消息（System V变量或POSIX 变量）的UNIX系统，按其自身正确的方式，也同样支持信号量，而且更有效。

7.9 System V信号量

System V信号量和POSIX信号量都不使用semwait和semop这样的简单的调用。它们的系统调用更复杂，尽管POSIX的信号量并不是非常复杂。在7.10.3节中，将对这两种情况一起评价。

首先，讲解System V。对我来说，这些系统调用太复杂，无法尽述。但是我确信，这里解释的内容足以用来实现semwait和semop；余下的内容，请参见[SUS2002]的系统文档。

7.4节中的知识适用于这些系统调用，因此这里不需要专门介绍关键字、标识符、所有权等内容了。

关于System V信号量设施有意思的是，系统调用不只运行在单独的信号量上，而是一次在整个数组上。借助原子操作，可以在一些信号量上调用semwait而在另一些信号量上调用semop。是否需要在实践中使用它，是值得怀疑的，但如果需要，可以这么做。尽管后来为了方便，同时对两个信号量的集合进行了操作，但是在大多数例子中，每次仅操作一个信号量。

在继续讲之前需要解释一下：System V的信号量太复杂！在任何使用信号量的程序中，都必须证明所要访问的共享资源是独占的，不会发生死锁，并且也不会发生共享资源饿死（从来不被访问）。因为那些东西是如此的依赖于时间，通常单独的测试是不够的；因此必须进行分析。对于简单的semwait和semop，这也是非常困难的。分析所有的System V系统调用大概是不可可能的。

7.9.1 System V信号量系统调用

与使用System V消息队列时一样，从Xget调用开始讲起：

semget——得到信号量集合标识符

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

int semget(
    key_t key,           /* key */
    int nsems,          /* size of set */
    int flags            /* flags */
);
/* Returns identifier or -1 on error (sets errno) */
```

就像所期待的那样，semget将关键字转换成了代表一个信号量集的ID。如果flags的IPC_CREAT位是打开的，且该信号量集合尚未存在，那么此调用将会创建它。在该集合中，有nsems信号量，它们以0开始计数。

信号量不能被立即使用——必须使用semctl调用来对其进行初始化。为什么semget不将它们初始化为0是件很奇怪的事，但SUS并不要求它这么做，^⑨并且许多实现也不这么要求。下面是semctl：

⑨ 事实上，SUS中规定，“和集合中的每个信号量相联系的数据结构都不会被初始化。”短语“不会”意味着对它初始化甚至不是一个实现选项。这是有意的还是书写马虎呢？

semctl——控制信号量集合

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

int semctl(
    int semid,           /* identifier */
    int semnum,         /* semaphore number */
    int cmd,            /* command */
    union semun arg     /* argument for command */
);
/* Returns value or 0 on success; -1 on error (sets errno) */
```

union semun——semctl的union

```
union semun {
    int val;             /* integer */
    struct semid_ds *buf; /* pointer to structure */
    unsigned short *array; /* array */
};
```

struct semid_ds——semctl的结构

```
struct semid_ds {
    struct ipc_perm sem_perm; /* permission structure */
    unsigned short sem_nsems; /* size of set */
    time_t sem_otime;        /* time of last semop */
    time_t sem_ctime;        /* time of last semctl */
};
```

很奇怪，在头文件sem.h中没有定义union semun——必须自己定义它。

除了通常的System V IPC的命令IPC_RMID、IPC_STAT和IPC_SET外，有7个专门用于信号量的命令：

- GETNCNT 得到等待信号量semnum增加而被阻塞的进程数。
- GETZCNT 得到等待信号量semnum变为0而阻塞的进程数。
- GETPID 得到最后执行semop的进程的ID。
- GETVAL 得到信号量semnum的值。
- SETVAL 设置信号量semnum的值。使用arg.val。
- GETALL 得到集合中所有信号量的值。使用arg.array。
- SETALL 设置集合中所有信号量的值。使用arg.array。

令人吃惊的是，如果有一个100个信号量的集合，且想要使用semctl将它们全部都设置为0，则必须建立一个100个元素全为0的数组作为第4个参数来使用！这对我们没什么影响，因为通常总是用SETVAL一次性地对它们进行初始化。

IPC_RMID命令的功能和使用msgctl相似（见7.5.1节）。

IPC_STAT使用union的buf成员填充由第4个参数传递的semid_ds结构。IPC_SET仅用于设置sem_perm.uid、sem_perm.gid和sem_perm.mode成员。

前面曾说过，因为semget并不初始化信号量，所以要使用semctl来完成该工作，顺序如下：

```
ec_negl(semid = semget(key, 1, PERM_FILE | IPC_CREAT) )
arg.val = 0;
ec_negl( semctl(semid, 0, SETVAL, arg) )
```

不幸的是，因为使用了两个系统调用来创建和初始化信号量，所以在semctl被执行之前，在同一个信号量上调用semget的第二个进程或线程可以开始处理一个未初始化的信号量。该问题的解决方法要求助于如下事实：当新信号量的值尚未被初始化完毕时，它的“最后的

semop时间”存储在semid_ds结构的sem_otime成员中并被semget初始化为0。基于以上事实给出^①如下方案：

- 创建信号量的进程或线程也调用semctl来初始化它，然后对它调用semop，以便sem_otime获得一个非零值。
- 所有得到信号量ID的进程或线程都要等待时间变成非零。

这个方案仅仅对新版信号量有效。如果重新启动先前运行过的有信号量的应用程序，那个信号量便已经有值了，且sem_otime有非零值，这可能会引起混乱。所以，必须确保在两次运行程序中间通过删除该信号量把信号量清理干净，具体可以使用semctl调用或使用ipcrm命令（7.4.4节）来实现。

不幸的是，到写本书为止，在FreeBSD或Darwin上等待sem_otime并不起作用，因为从来都不更新该时间。^②因此除非在第二次尝试得到信号量ID（semget）之前，能确定由于应用本身的原因初始化会发生，否则根本不能可靠地使用System V信号量。（更不幸的是，这些系统也并不完全支持POSIX信号量。）

下一节介绍操作信号量集合的系统调用semop。

7.9.2 简单信号量接口

利用简单的信号量打开调用，可以隐藏现在要实现的System V信号量和稍后要实现的POSIX信号量的复杂性。下面是SimpleSemOpen和与其配对的SimpleSemClose的对照表：

SimpleSemOpen —— 打开简单信号量

```
#include "SimpleSem.h"

struct SimpleSem *SimpleSemOpen(
    const char *name /* name (follows System V or POSIX rules) */
);
/* Returns pointer to structure or NULL on error (sets errno) */
```

SimpleSemClose —— 关闭简单信号量

```
#include "SimpleSem.h"

bool SimpleSemClose(
    struct SimpleSem *sem /* semaphore */
);
/* Returns true on success or false on error (sets errno) */
```

struct SimpleSem —— 简单信号量函数的结构

```
struct SimpleSem {
    union {
        int sm_semId; /* System V semaphore-set ID */
        void *sm_sem; /* POSIX sem_t pointer (needs a cast) */
    } sm;
};
```

SimpleSemOpen根本没有任何选项：它通过名字打开单个的信号量（在System V的实现中，它自己获得关键字），如果有必要，可以使用权限PERM_FILE（见2.3节）来创建它。它返回指向SimpleSem结构的指针，该结构包含了其他函数识别信号量集（或其中一个）所

① 我从[Ste1999]的第284页中了解了这个技术。在本书的第1版中我弄错了。

② 我听说FreeBSD5.1中修正了它。

需要的所有信息，对于System V来说，用于识别信号量的信息是一个整型信号量集的ID。（稍后，将介绍POSIX信号量的需求）。但使用简单信号量包的用户并不需要关心SimpleSem结构中有什么，因为指向它的指针只会被传递给其他一些函数，如SimpleSemClose，而该函数关闭了信号量并释放了SimpleSem结构所使用过的所有内存。

下面是System V系统下SimpleSemOpen的实现，在这儿，可以看到前面所讲的确信信号量被恰当初始化的方案：

```

struct SimpleSem *SimpleSemOpen(const char *name)
{
    struct SimpleSem *sem = NULL;
    key_t key;
    union semun {
        int val;
        struct semid_ds *buf;
        unsigned short *array;
    } arg;
    struct sembuf sop;

    (void)close(open(name, O_WRONLY | O_CREAT, 0));
    ec_negl( key = ftok(name, 1) )
    ec_null( sem = malloc(sizeof(struct SimpleSem)) )

    if ((sem->sm.sm_sem_id = semget(key, 1,
        PERM_FILE | IPC_CREAT | IPC_EXCL)) != -1) {
        arg.val = 0;
        ec_negl( semctl(sem->sm.sm_sem_id, 0, SETVAL, arg) )
        sop.sem_num = 0;
        sop.sem_op = 0;
        sop.sem_flg = 0;
        ec_negl( semop(sem->sm.sm_sem_id, &sop, 1) )
    }
    else {
        if (errno == EEXIST) {
            while (true)
                if ((sem->sm.sm_sem_id = semget(key, 1, PERM_FILE)) == -1) {
                    if (errno == ENOENT) {
                        sleep(1);
                        continue;
                    }
                    else
                        EC_FAIL
                }
            else
                break;
            while (true) {
                struct semid_ds buf;

                arg.buf = &buf;
                ec_negl( semctl(sem->sm.sm_sem_id, 0, IPC_STAT, arg) )
                if (buf.sem_otime == 0) {
                    sleep(1);
                    continue;
                }
                else
                    break;
            }
        }
        else
    }
}

```



```

        EC_FAIL
    }
    return sem;

EC_CLEANUP_BGN
    free(sem);
    return NULL;
EC_CLEANUP_END
}

```

其中如下两个语句的顺序

```

(void)close(open(name, O_WRONLY | O_CREAT, 0));
ec_neg1( key = ftok(name, 1) )

```

和7.5.3节中的一样。如果名字不存在，便创建名字，而且在使用ftok时，所有因名字而产生的问题都会被报告，而使用open或close时就不会报告所有的问题。（如果不喜欢那样做，可以独自检查open来发现除了ENOENT之外的错误，而不把它嵌入到立即调用close中。）

然后，使用IPC_CREAT和IPC_EXCL标记调用semget，以便当信号量已经存在时semget获得成功返回的进程或线程使用semctl对它进行初始化，然后调用semop，它对该值没有任何改变（稍后将作解释），但副作用是设置了sem_otime。

需要等待的进程和线程将循环直到某个semget成功，每次循环都休眠，然后再一次循环直到时间变为非零，每次循环也休眠。

对System V信号量而言，SimpleSemClose并没有做什么：

```

bool SimpleSemClose(struct SimpleSem *sem)
{
    free(sem);
    return true;
}

```

为了删除简单信号量，可以调用SimpleSemRemove：

SimpleSemRemove —— 删除简单信号量

```

#include "SimpleSem.h"

bool SimpleSemRemove(
    struct SimpleSem *sem    /* semaphore */
);
/* Returns true on success or false on error (sets errno) */

```

在一个并不存在的信号量上调用SimpleSemRemove是可以的，这并不被认为是一个错误。事实上，当应用程序需要使用新的信号量启动时，恰恰应该这么做。

下面是SimpleSemRemove的代码：

```

bool SimpleSemRemove(const char *name)
{
    key_t key;
    int semid;
    if ((key = ftok(name, 1)) == -1) {
        if (errno != ENOENT)
            EC_FAIL
    }
}

```

```

else {
    if ((semid = semget(key, 1, PERM_FILE)) == -1) {
        if (errno != ENOENT)
            EC_FAIL
    }
    else
        ec_negl( semctl(semid, 0, IPC_RMID) )
}
return true;

EC_CLEANUP_BGN
return false;
EC_CLEANUP_END
}

```

现在，回到System V。准备介绍semop系统调用，它完成了semwait和semopost所做的工作：

semop——操作信号量集合

```

#include <sys/sem.h>

int semop(
    int semid,                /* identifier */
    struct sembuf *sops,      /* operations */
    size_t nsops              /* number of operations */
);
/* Returns 0 on success or -1 on error (sets errno) */

```

struct sembuf——semop的结构

```

struct sembuf {
    unsigned short sem_num;    /* semaphore number */
    short sem_op;             /* Semaphore operation */
    short sem_flg;           /* Operation flags */
};

```

前面曾说过，semop不不仅可以操作一个信号量而且可以操作任意多个，甚至是整个信号量集合。在调用之前，需要建立一个操作数组（struct sembuf），但如果只要操作一个信号量，那么用一个单独的struct sembuf就可以了——只要传递它的地址并让nsops使用1。

每个sem_op可以是正的，可以是负的，也可以是0：

>0将sem_op的值加到信号量的值上（semopost）。

<0从信号量的值里减去sem_op的绝对值得到，除非减后该值变为负值，在这种情况下，调用阻塞直到整个值能够被减（semwait）。

0调用阻塞直到信号量的值变为0。

传递给semop的所有操作都可以自动执行，而且直到所有的操作都完成了，该函数才会返回。（除非它被线程撤消、信号或信号量集的删除所中断。）

通过为操作设置sem_flg成员的IPC_NOWAIT标志，可以防止阻塞。如果数组中有操作要阻塞且已经设置了该标志，那么semop会立刻返回。因为semop总是原子性地运行，所以即使其他操作在数组中出现得早并且这些操作可能没有阻塞，也不会运行这些操作。

另外还有一个特征：对进程增加或减少的每一个信号量，和实际值一起，不断调整。当对增加操作设置了IPC_UNDO标志时，作减少调整，对于一个减少操作反之亦然。当进程退出时，调整值被加到信号量上，然后取消所有进程对信号量所做过的操作。例如，假定一个

进程减少一个信号量（例如，semwait）去锁定缓冲区，并且当完成时，增加它（sempost）。使用IPC_UNDO标志，如果它非正常地退出，能够确保缓冲区解锁。

对于简单信号量，每次仅需对一个信号量进行操作，仅增加或减少1，并且不必使用IPC_NOWAIT或IPC_UNDO。因此SimpleSemWait和SimpleSemPost也是简单的：

SimpleSemWait——减少简单信号量

```
#include "SimpleSem.h"
bool SimpleSemWait(
    struct SimpleSem *sem    /* semaphore */
);
/* Returns true on success or false on error (sets errno) */
```

SimpleSemPost——增加简单信号量

```
#include "SimpleSem.h"
bool SimpleSemPost(
    struct SimpleSem *sem    /* semaphore */
);
/* Returns true on success or false on error (sets errno) */
```

以下是代码：

```
bool SimpleSemWait(struct SimpleSem *sem)
{
    struct sembuf sop;

    sop.sem_num = 0;
    sop.sem_op = -1;
    sop.sem_flg = 0;
    ec_negl( semop(sem->sm.sm_semid, &sop, 1) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

bool SimpleSemPost(struct SimpleSem *sem)
{
    struct sembuf sop;

    sop.sem_num = 0;
    sop.sem_op = 1;
    sop.sem_flg = 0;
    ec_negl( semop(sem->sm.sm_semid, &sop, 1) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

在继续讲述之前，先说一下System V信号量的另一个用途：在进程间传递整数。例如，假定一个客户端希望向服务器端传递它的进程ID。它访问信号量，并将信号量的值设置为进程ID。然后，服务器查看该值从而得到该数字。因为信号量集有一个信号量数组，所以可以用这种方式来传递整型数组。

7.10 POSIX信号量

这些信号量系统调用是POSIX标准的一部分（尽管是可选的），必须检查 `_POSIX_SEMAPHORES` 特性测试宏以得知它们是否存在（见1.5.4节和7.6.3节）。到目前为止，FreeBSD、Darwin或Linux中没有这些信号量。

7.10.1 命名的POSIX信号量

POSIX信号量比System V信号量使用起来更简单，更容易。事实上，该系统调用中的5个和前一节的SimpleSem接口正好一致：

sem_open——打开命名信号量

```
#include <semaphore.h>

sem_t *sem_open(
    const char *name,      /* POSIX IPC name */
    int flags              /* flags (excluding O_CREAT) */
);
/* Returns pointer to semaphore or SEM_FAILED on error (sets errno) */

sem_t *sem_open(
    const char *name,      /* POSIX IPC name */
    int flags,            /* flags (including O_CREAT) */
    mode_t perms,         /* permissions */
    unsigned value         /* initial value */
);
/* Returns pointer to semaphore or SEM_FAILED on error (sets errno) */
```

sem_close——关闭命名信号量

```
#include <semaphore.h>

int sem_close(
    sem_t *sem             /* semaphore */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

sem_unlink——删除命名信号量

```
#include <semaphore.h>

int sem_unlink(
    const char *name       /* POSIX IPC name */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

sem_wait——减少信号量

```
#include <semaphore.h>

int sem_wait(
    sem_t *sem             /* semaphore */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

sem_post——增加信号量

```
#include <semaphore.h>

int sem_post(
    sem_t *sem             /* semaphore */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

POSIX信号量是计数信号量，和System V变体一样，但步长仅是1。每个sem_t对象仅代表一个信号量，并不像System V系统那样代表它们的集合。

打开、关闭和解链的调用遵循了POSIX IPC调用的风格——该风格在7.7.1节中POSIX消息队列系统调用出现过。特别地，传递给sem_open的名字必须符合7.6.2节中所讨论的可移植性规则。像所期望的那样，sem_post将信号量值加1，而sem_wait减1，如果该值已经是0，则阻塞。

不要和sem_open一起使用O_RDONLY、O_WRONLY或O_RDWR，因为这样是没有意义的——信号量是无用的，除非sem_post和sem_wait能一起使用。

对System V所需要的所有^①的初始工作都完成了。实际上真正要做的就是要将需要的值（常是0）传递给sem_open。

当sem_open失败时，要当心它的返回值：它是SEM_FAILED，而不是NULL，NULL通常是函数失败的返回值，函数失败时会以其他方式返回一个指针。

POSIX信号量的SimpleSem调用的实现是很普通的：

```
struct SimpleSem *SimpleSemOpen(const char *name)
{
    struct SimpleSem *sem = NULL;

    ec_null( sem = malloc(sizeof(struct SimpleSem)) )
    if ((sem->sm.sm_sem = sem_open(name, O_CREAT, PERM_FILE, 0)) ==
        SEM_FAILED)
        EC_FAIL
    return sem;
EC_CLEANUP_BGN
    free(sem);
    return NULL;
EC_CLEANUP_END
}

bool SimpleSemClose(struct SimpleSem *sem)
{
    ec_neg1( sem_close(sem->sm.sm_sem) )
    free(sem);
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

bool SimpleSemRemove(const char *name)
{
    if (sem_unlink(name) == -1 && errno != ENOENT)
        EC_FAIL
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

bool SimpleSemPost(struct SimpleSem *sem)
{
    ec_neg1( sem_post(sem->sm.sm_sem) )
```

① 可能你对这个词并不熟悉，我查了一下字典，其中一个字典对这个词的定义是“复杂繁琐的一套手续”，在这里这样解释很合适。

```

    return true;
EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

bool SimpleSemWait(struct SimpleSem *sem)
{
    ec_negl( sem_wait(sem->sm.sm_sem) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

```

POSIX信号量有一些SimpleSem接口并不需要的额外特性。首先，在System V下，不必修改或等待就可以查询信号量的值：

sem_getvalue —— 得到信号量的值

```

#include <semaphore.h>

int sem_getvalue(
    sem_t *restrict sem,      /* semaphore */
    int *valuep              /* returned value */
);
/* Returns 0 on success or -1 on error (sets errno) */

```

在调用sem_getvalue时，如果信号量的值比0大，便返回该值。尽管如此，信号量的值还是会因sem_getvalue的返回值的时间的不同而略有差异，因此实际的数并不是那么有用的。如果值是0，那么返回的值将是等待信号量的进程数量的负数。如果没有任何值，将返回0。

sem_wait有两个变体：一个是sem_trywait，它是非阻塞的：

sem_trywait —— 如果可能就减少信号量

```

#include <semaphore.h>

int sem_trywait(
    sem_t *sem              /* semaphore */
);
/* Returns 0 on success or -1 on error (sets errno) */

```

如果信号量已经是0，则sem_trywait返回-1，并将errno设置为EAGAIN。（对于非阻塞，其他系统调用使用O_NONBLOCK或IPC_NOWAIT这样的标志；这里它是独立的系统调用。）

如果信号量没有变成正数，那么另一个变体sem_timedwait在一个时间段之后会暂停：

sem_timedwait —— 减少信号量

```

#include <semaphore.h>
#include <time.h>

int sem_timedwait(
    sem_t *restrict sem,      /* semaphore */
    const struct timespec *time /* absolute time */
);
/* Returns 0 on success or -1 on error (sets errno) */

```

传递给sem_timedwait的时间是绝对时间（例如，1:23:17 PM），而不是一个时间间隔（如，27秒）。它与哪个时钟相比较和使用什么样的精度取决于是否支持POSIX定时器选项。

如果支持，则使用实时时钟。如果不支持，则使用普通时钟（如time系统调用使用的那样）。（见1.7节中有关struct timespec的讨论以及UNIX时间的其他部分。）

sem_timedwait是超时选项（_POSIX_TIMEOUTS）的一部分并且是SUS3中新增加的。

7.10.2 未命名的POSIX信号量

快速回顾：sem_open、sem_close和sem_unlink是与命名信号量一起使用的，而命名信号量存在于进程外部的某个地方，需要通过POSIX IPC名来访问它们。可以通过sem_open得到一个指向sem_t对象的指针，不可以直接处理该对象。当调用sem_close时，会释放（sem_open）使用的所有内存。这些信号量固定地在线程和进程之间工作。

为了使信号量更快，也可以直接声明sem_t对象：

```
sem_t sem;
```

或动态地分配一个，像这样：

```
sem_t *semp = malloc(sizeof(sem_t));
```

但如果自己分配sem_t对象，必须调用sem_init对它初始化：

sem_init——初始化未命名信号量

```
#include <semaphore.h>

int sem_init(
    sem_t *sem,           /* semaphore */
    int pshared,         /* shared between processes? */
    unsigned value       /* initial value */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

然后调用sem_destroy来删除：

sem_destroy——删除未命名信号量

```
#include <semaphore.h>

int sem_destroy(
    sem_t *sem           /* semaphore */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

sem_destroy并不释放sem_t对象，因为它并不知道内存是怎样分配的（静态声明、malloc，等等）。必须自己释放内存（如果恰当）。当然，如果全局地或在栈上分配信号量，就什么都不必做；信号量的生存期将在适当的时间结束。

sem_init和sem_destroy用于替代sem_open和sem_close；可以使用这对，也可以使用那对，具体取决于信号量是命名的还是未命名的。另一些调用（例如，sem_post、sem_timedwait）并不在意sem_t指针是怎样获得的。

好了，未命名的信号量可能更快，但如果它们处在单进程的内存中，它们有什么优点呢？真正的优点在于：

- 当需要一个计数信号量而不仅是一个仅有两个状态（二元信号量）的互斥时，它们在线程间的同步工作完成得非常好。
- 如果它们在共享内存中，那么在进程间会工作得很好，稍后就会讲到。在这种情况下，sem_init的pshared参数必须是非零的。

对线程来说，第一个优点是非常重要的，而且一些UNIX实现也支持它，典型的有Linux和FreeBSD（但不含Darwin），尽管某些版本并不完全支持POSIX信号量。更确切地说，是支

持`sem_init` (`pshared`仅为0)和`sem_destroy`, 但不支持`sem_open`和`sem_close`。在这些有限的实现下, 也支持`sem_post`、`sem_wait`、`sem_trywait`和`sem_getvalue`, 但不支持`sem_timedwait`。

关于未命名(在内存中)信号量的另一个规则是: 只能使用传递给`sem_init`的实际内存, 而不是它的副本。所以下面的是错误的:

```
void fcn(sem_t s)
{
    sem_post(&s);
    ...
}

void fcn2(void)
{
    sem_t sem;

    sem_init(&sem);
    fcn(sem);
    ...
}
```

调用`fcn`复制该信号量, 破坏了这个规则。另外, 程序中所有处理信号量的部分都需要使用原先的初始化存储地址来运行。在7.14.2节中, 将给出一个驻留在共享内存中的未命名信号量的例子。

7.10.3 评论System V信号量和POSIX信号量

对于进程间通信, System V 信号量系统调用是非常难用的, 并且不少缺点, 但它们都是普遍可用的, 并且一旦困难的部分(主要指初始化)隐藏在像SimpleSem那样合理的接口之后, 也不会太糟糕。POSIX信号量使用起来更容易, 但并不是普遍可用的。如果对用户来说, 可移植性更重要, 就需要使用System V系统调用, 而如果不能随处使用System V系统调用, 那么用POSIX调用根本就得不到任何东西。如果移植性不重要, 并且在你的UNIX系统中, 支持POSIX调用, 那么很明显应该使用POSIX调用。[⊖]

对于进程内通信(即线程之间), System V信号量太过于笨重和缓慢, 而且未命名的、非进程共享POSIX信号量通常在POSIX线程所在的地方都是可用的, 所以如果用户需要互斥之外的东西, 应该使用POSIX信号量。

7.10.4 共享进程互斥和读写锁

在5.17.3节中, 为了使用同步线程, 介绍了互斥, 但它们是在内存中并且在一个进程中——实质上, 在内存中, 非进程共享的POSIX信号量使用的是二进制形式。用户也可以创建内存互斥并利用`PTHREAD_PROCESS_SHARED`属性设置的`pthread_mutex_init`初始化它。在这种情况下, 它是在线程间共享的, 这些线程可能在不同的进程中。但是, 即使POSIX线程(它是独立子选项)是这种情况, 也并不总是实现这些特性, 因此它可能是不可用的。

也存在许多POSIX线程读写锁, 在第5章中, 根本没有描述它们。它们很像互斥, 但为了允许更多的吞吐量, 它们在读和写上有所不同(共享和互斥)。对于互斥, 用户可以设置一个`THREAD_PROCESS_SHARED`属性。[下一节介绍, 读写锁对互斥体就像`fcntl`文件锁对`lockf`文件锁那样(见7.11.3和7.11.4节)。]

[⊖] 不要轻易假定移植性不重要。几乎现在的每一个程序都有可能将来的某一天被引入到Linux中运行。而且即使从来不移植, 当在与销售商谈判时, 潜在的移植性也会有帮助。

7.11 文件锁

这一节将解释在UNIX中，文件锁是怎样工作的，为什么它有时候并没有按用户所要求的那样去做。

7.11.1 一个糟糕的例子

我们喜欢先举一个有错误的例子——这有利于了解所采用的正确方法的动机。

为了激发关于文件锁的讨论，举一个例子：一个进程建立文件而另一个进程读该文件。文件是一个记录的链表，每条记录保存一个整型数据和下一条记录的偏移量。链表是有序的，但记录的物理位置是按照它们创建的顺序排列的。每条记录的结构是：

```
struct rec {
    int r_data;
    off_t r_next;
};
```

如果以1000、999和998这样的顺序将数据插入到记录中，文件看起来就会像图7-2那样。第一条记录只是一个头，指明了链表从哪里开始。

该例的main程序启动了一个子进程（process1）来建立链表，而父进程（process2）重复地遍历它来检查它形成得是否正确：

```
int main(void)
{
    pid_t pid;

    ec_negl( pid = fork() )
    if (pid == 0)
        process1();
    else {
        process2();
        ec_negl( waitpid(pid, NULL, 0) )
    }
    exit(EXIT_SUCCESS);
}
```

```
EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

对于数据，在写了头记录之后，process1只需要从1000向后计数：

```
#define DBNAME "termdb"
```

```
static void process1(void)
{
    int dbfd, data;
    struct rec r;

    ec_negl( dbfd = open(DBNAME, O_CREAT | O_TRUNC | O_RDWR, PERM_FILE) )
    memset(&r, 0, sizeof(r));
    ec_false( writerec(dbfd, &r, 0) )
    for (data = 100; data >= 0; data--)
        ec_false( store(dbfd, data) )
    ec_negl( close(dbfd) )
    exit(EXIT_SUCCESS);
}
```

```
EC_CLEANUP_BGN
```

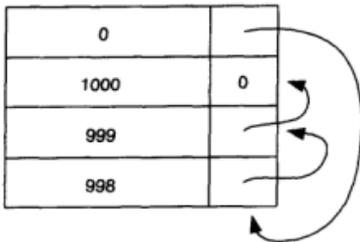


图7-2 记录的链表

```

    exit(EXIT_FAILURE);
EC_CLEANUP_END
}

```

为了使I/O操作容易，有两个函数：一个是`writerec`，它能在一个指定偏移处写一条记录；一个是`readrec`，它能从一个偏移处读取一个记录。它们都把部分读、部分写或文件结尾当作错误：

```

bool readrec(int dbfd, struct rec *r, off_t off)
{
    ssize_t nread;

    if ((nread = pread(dbfd, r, sizeof(struct rec), off)) ==
        sizeof(struct rec))
        return true;
    if (nread != -1)
        errno = EIO;
    EC_FAIL
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

bool writerec(int dbfd, struct rec *r, off_t off)
{
    ssize_t nwrote;

    if ((nwrote = pwrite(dbfd, r, sizeof(struct rec), off)) ==
        sizeof(struct rec))
        return true;
    if (nwrote != -1)
        errno = EIO;
    EC_FAIL
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

```

原本可以很容易地直接使用`pwrite`和`pread`，但该错误检查有一点混乱，因此对它们进行封装是有道理的。

函数`store`（从`process1`调用的）负责按顺序保持链表。注意，它并不会尝试最小化I/O：

```

bool store(int dbfd, int data)
{
    struct rec r, rnew;
    off_t end, prev;

    ec_negl( end = lseek(dbfd, 0, SEEK_END) )
    prev = 0;
    ec_false( readrec(dbfd, &r, prev) )
    while (r.r_next != 0) {
        ec_false( readrec(dbfd, &r, r.r_next) )
        if (r.r_data > data)
            break;
    }
}

```

```

        prev = r.r_next;
    }
    ec_false( readrec(dbfd, &r, prev) )
    rnew.r_next = r.r_next;
    r.r_next = end;
    ec_false( writerec(dbfd, &r, prev) )
    rnew.r_data = data;
    usleep(1); /* give up CPU */
    ec_false( writerec(dbfd, &rnew, end) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

```

在store上花一点时间就能清楚地了解它。注意到，它能恰当地处理新记录在开头或结尾的情况。在结尾附近对usleep的调用（休眠1微秒）只是为了放弃CPU而让其他进程运行一下。在复杂程序中，这是很自然的，但在这个简单的例子中，必须强制它执行，因为需要演示的其他进程并发运行。

下面是检查文件完整性的process2:

```

static void process2(void)
{
    int try, dbfd;
    struct rec r1, r2;

    for (try = 0; try < 10; try++)
        if ((dbfd = open(DBNAME, O_RDWR)) == -1) {
            if (errno == EWOULDBLOCK) {
                continue;
            }
            else
                EC_FAIL
        }
    ec_neg1( dbfd )
    for (try = 0; try < 100; try++) {
        ec_false( readrec(dbfd, &r1, 0) )
        while (r1.r_next != 0) {
            ec_false( readrec(dbfd, &r2, r1.r_next) )
            if (r1.r_data > r2.r_data) {
                printf("Found sorting error (try %d)\n", try);
                break;
            }
            r1 = r2;
        }
    }
    ec_neg1( close(dbfd) )
    return;

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}

```

这里尝试了几次才打开了数据库，因为要花费process1一点时间来调度并完成调用open。然后，程序重复地（100次）浏览链表检查错误，如那些丢失的记录或次序颠倒的数据。

下面的内容确实是我运行程序时所得到的：

```
ERROR: 0: process2 [/aup/c7/fl.c:107] readrec(dbfd, &r2, r1.r_next)
1: readrec [/aup/c7/fl.c:17] 0
*** EIO (5: "I/O error") ***
```

由于当process2查找新记录时，store打算向数据库结尾写人的新记录还没有被写入，所以导致了这个特殊的错误。

现在，我们的程序并不能正常工作，它没有完全完成，其不工作的特殊原因在于两个进程正在访问同一个文件，而且其中一个进程发现了不一致的数据。它们需要协调。

现在，你了解到本节所讲内容的动机了吗？

7.11.2 用信号量作文件锁

一个明显的，可以修正上一节所讲示例的方法是使用信号量阻止process2查看一个不一致的文件。使用7.9.2节中的SimpleSem接口，可以定义一个如下的全局变量：

```
static struct SimpleSem *sem;
```

每个进程独立地打开信号量，如下行这样：

```
ec_null( sem = SimpleSemOpen("sem") )
```

另外，process1首先增加信号量值以指示该数据库是一致的：

```
ec_false( SimpleSemPost(sem) )
```

store中的关键代码行是那些在恰当的位置更新记录，而在结尾存储新记录的代码行：

```
ec_false( SimpleSemWait(sem) )
ec_false( readrec(dbfd, &r, prev) )
rnew.r_next = r.r_next;
r.r_next = end;
ec_false( writerec(dbfd, &r, prev) )
rnew.r_data = data;
usleep(1); /* give up CPU */
ec_false( writerec(dbfd, &rnew, end) )
ec_false( SimpleSemPost(sem) )
```

而process2中的关键代码行是那些遍历链表的行：

```
for (try = 0; try < 100; try++) {
    ec_false( SimpleSemWait(sem) )
    ec_false( readrec(dbfd, &r1, 0) )
    while (r1.r_next != 0) {
        ec_false( readrec(dbfd, &r2, r1.r_next) )
        if (r1.r_data > r2.r_data) {
            printf("Found sorting error (try %d)\n", try);
            break;
        }
        r1 = r2;
    }
    ec_false( SimpleSemPost(sem) )
}
```

有了这些改变，应用程序就可以正确工作了。（除FreeBSD和Darwin以外，在这两个系统上System V信号量不能正确地实现，与7.9.1节所述原因相同。）

用信号量作文件锁的主要问题是，对于一个任意的文件，信号量名字应该是什么不是很清楚。对于具有几个固定文件名的应用程序而言，这并不是一个问题，但在处理特别文件名的应用中非常难用。必须设计一些把文件名映射成信号量名的设施。更糟糕的是，有时候仅

需要锁定文件的一部分，这意味着需要不同的信号量对应不同的部分。最后，管理信号量（打开和关闭它们，当它们无用时，删除它们）是一件痛苦的事。

7.11.3 lockf系统调用

争论是否使用信号量锁定文件其实并不是问题，因为UNIX有一个专门的系统调用锁定文件的某一部分：

lockf——锁定文件段

```
#include <unistd.h>

int lockf(
    int fd,           /* file descriptor */
    int op,          /* operation */
    off_t len        /* length of section */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

文件描述符必须打开用以写入（O_WRONLY或O_RDWR）。

加锁或解锁某段是从当前文件偏移量（由read、write或lseek设置）开始，当len为正则向前len字节，当len为负则向后。在向后的情况下，处在当前偏移量处的字节不是该段的一部分。在向前的情况下，不存在的字节也可以被锁定，因为文件还没有达到那么大。如果len为0，则该段可以从当前偏移量延伸到文件结尾，甚至随着文件增长而增大。因此，要锁定整个文件，仅仅需要确定使偏移量为0并使用零长度。

如果要锁定的段与已经锁定的段有了交迭，则两个段进行合并。如果锁定段的某一部分解除了锁定，锁定段缩小，而且有可能分为两个不连续的段。

当进程在文件上打开的任意一个文件描述符关闭时，该进程在文件上的所有锁都将被释放，即使要关闭的文件描述符是从传递给lockf的文件描述符独立获得（即不同的open）的。^⑨由此得出结论当进程终止时，所有的锁都会被释放，因为那样会使所有文件描述符关闭。当进程调用fork时，锁并不会被继承。

下面是关于op参数的操作：

F_LOCK 锁定段；如果它的任何一部分已经由其他进程锁定，它将阻塞。

F_TLOCK 像F_LOCK一样，但如果F_LOCK已经阻塞，它将返回-1，并且设置errno为EAGAIN（或EACCES）。

F_TEST 不锁定，但如果F_LOCK已经阻塞，像F_TLOCK一样，将返回一个错误。

F_ULOCK 解除该段的锁定。

在文件上使用lockf，不必有信号量名或打开任何特别的東西，因为它使用与访问文件的描述符相同的文件描述符。因此，可以很容易地把它放入示例中来代替上一节的信号量调用。这里仅仅给出store的关键部分：

⑨ 来自于FreeBSD联机资料：“这个接口完全遵循了System V和IEEE Std 1003.1-1998（‘POSIX.1’）的笨拙的语义。这要求，对于一个给定的进程，当某个文件的任意一个文件描述符被该进程关闭时，必须释放所有和该文件相关的锁。这个语义意味着应用程序必须注意子程序库可能访问的所有文件。例如，如果更新密码文件的程序在更新时锁定了密码文件数据库，然后调用getpwnam（3）来检索一条记录，那么因为getpwnam（3）对密码数据库要进行打开、读取和关闭操作，锁定可能会丢失。”基于BSD的系统有更好的调用——flock，但它不是标准的。

```

ec_negl( lseek(dbfd, 0, SEEK_SET) )
ec_negl( lockf(dbfd, F_LOCK, 0) )
ec_false( readrec(dbfd, &r, prev) )
rnew.r_next = r.r_next;
r.r_next = end;
ec_false( writerec(dbfd, &r, prev) )
rnew.r_data = data;
usleep(1); /* give up CPU */
ec_false( writerec(dbfd, &rnew, end) )
ec_negl( lseek(dbfd, 0, SEEK_SET) )
ec_negl( lockf(dbfd, F_ULOCK, 0) )

```

和process2的关键部分:

```

for (try = 0; try < 100; try++) {
    ec_negl( lseek(dbfd, 0, SEEK_SET) )
    ec_negl( lockf(dbfd, F_LOCK, 0) )
    ec_false( readrec(dbfd, &r1, 0) )
    while (r1.r_next != 0) {
        ec_false( readrec(dbfd, &r2, r1.r_next) )
        if (r1.r_data > r2.r_data) {
            printf("Found sorting error (try %d)\n", try);
            break;
        }
        r1 = r2;
    }
    ec_negl( lseek(dbfd, 0, SEEK_SET) )
    ec_negl( lockf(dbfd, F_ULOCK, 0) )
}

```

7.11.4 文件锁的fcntl系统调用

也可以使用fcntl系统调用锁定文件，它第一次出现是在3.8.3节中。它的锁定功能是lockf的超集。下面是fcntl对照表的扼要重述：

fcntl——控制打开的文件

```

#include <unistd.h>
#include <fcntl.h>

int fcntl(
    int fd,          /* file descriptor */
    int op,          /* operation */
    ...              /* optional argument depending on op */
);
/* Returns result depending on op or -1 on error (sets errno) */

```

有3个fcntl操作用于对结构操作的锁定，指向结构体的指针作为第三个参数传递：

struct flock——fcntl文件锁定的结构

```

struct flock {
    short l_type;      /* lock type: F_RDLCK, F_WRLCK, F_UNLCK */
    short l_whence;    /* interpretation of l_start */
    off_t l_start;     /* start of section */
    off_t l_len;       /* length of section */
    pid_t l_pid;       /* process holding lock; used with F_GETLK */
};

```

该结构的三个成员分别为l_whence、l_start和l_len，它们建立了待操作的段。前

两个参数和lseek参数相似 (l_whence可以是SEEK_SET、SEEK_CUR或SEEK_END)，而l_start是相对当前文件的偏移量或者相对文件结尾的绝对值。段长度由l_len给出。

对lockf，只有一种类型的锁，但对fcntl既可以有读锁又可有写锁，或者，通常称为共享锁和独占锁。段上的共享锁会禁止该段上的独占锁；独占锁也会禁止共享锁或独占锁。实践中，当只需要读数据时，设置共享锁，而当写数据时，设置独占锁。

在l_type成员中指定锁的类型；对照表中所示的第三个选择是解除段锁定。

现在，开始解释fcntl的加锁操作，这是非常简单的：

F_SETLK 如果可能，在结构中执行指定的操作。如果不能立即设定锁，则返回-1，并将errno设置为EAGAIN或EACCES；也就是说，不阻塞。

F_SETLKW 就像F_SETLK一样，但是如果不立刻设置锁，会阻塞。

F_GETLK 如果有任何阻塞，返回结构中引起指定锁阻塞的第一个锁的信息。利用这些结果（包括拥有该锁的进程的进程ID）重写传入结构的所有成员。如果传递来的锁不阻塞，那么除了第一个成员会被变为F_UNLCK外，结构将按原样传回。

因此，fcntl加锁功能可以完成lockf能做的所有事。另外，它还区分共享锁和独占锁，可以检索已存在的锁的信息。通常，把lockf作为fcntl之上的库函数来实现，但并不要求必须这么做。同样，[SUS2002]提到不应该假定这两个函数操纵的锁是相同的。也就是说，如果使用lockf加锁，那么不要使用fcntl解锁。事实上，甚至不希望fcntl知道该锁。

和lockf锁一样，当对文件打开的文件描述符关闭或者当进程终止时，由fcntl设置的锁将被释放，而且当进程调用fork时，它们也不被继承。

7.11.5 建议锁和强制锁

用lockf和fcntl设置的锁通常只影响那些函数调用，而不影响其他I/O操作。也就是说，如果用lockf在文件上设置一个锁，然后另一个进程在没有调用lockf的情况下写该文件，那么写操作会继续。这叫做建议锁，显然，只有所有进程通过适当地调用lockf或fcntl进行合作，它才会起作用。

强制锁是指一旦设置了锁，就真的能禁止冲突的I/O操作。POSIX和SUS标准根本不指定强制锁，但也不禁止使用它。

在那些支持强制锁的系统上，调用lockf或fcntl时没有任何不同。相反可以利用在其他方面没有意义的权限集合来标记文件：打开设置组ID执行位，关闭组执行位。下面是使用建议锁或强制锁的例子，具体使用哪种锁要取决于参数：

```
int main(int argc, char *argv[])
{
    int fd;
    mode_t perms = PERM_FILE;

    if (fork() == 0) {
        sleep(1); /* wait for parent */
        ec_negl( fd = open("tmpfile", O_WRONLY | O_NONBLOCK) );
        ec_negl( write(fd, "x", 1) );
        printf("child wrote OK\n");
    }
    else {
        (void)unlink("tmpfile");
        if (argc == 2)
            perms |= S_ISGID; /* mandatory locking */
        ec_negl( fd = open("tmpfile", O_CREAT | O_RDWR, perms) );
    }
}
```



```

        ec_negl( lockf(fd, F_LOCK, 0) )
        printf("parent has lock\n");
        ec_negl( wait(NULL) )
    }
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}

```

下面是在Solaris上运行的结果，其中有强制锁：

```

$ lockftest
parent has lock
child wrote OK
$ lockftest x
parent has lock
ERROR: 0: main [/aup/c7/lockftest.c:11] write(fd, "x", 1)
*** EAGAIN (11: "Resource temporarily unavailable") ***
$

```

没有O_NONBLOCK标志，子进程中的写操作将阻塞以等待锁被释放。

7.11.6 高性能数据库锁

由fcntl和lockf系统调用提供的文件锁，即使利用强制锁，对高性能数据库也不合适，因为在处理锁时有太多的开销，而且要完善地实现死锁检测和解除死锁的算法非常困难。尽管如此，通常也是可以实现的，因为大的数据库系统运行在自己的进程上，所以它们扮演了数据库文件的看门人角色。可以很方便地将锁保存在数据库进程的地址空间或由多个数据库进程共享的内存中。以这种方法，根本不需要使用系统调用来管理锁。

7.12 关于共享内存

回顾5.17节，进程中的线程共享所有静态数据——全局数据和函数内部的静态数据。另一方面，即使在没有exec的情况下执行fork，进程也会拥有完全独立的内存，这种情况下，子进程会得到一份父进程地址空间的副本。

使用共享内存，可以为各个独立的进程分配一些共同的内存。和线程一样，它们通常需要使用互斥或信号量协调对该内存的访问。

与消息和信号量一样，共享内存既有System V版本的也有POSIX版本的。对于这两种机制，每个进程都“打开”一个共享内存段，并且得到一个指向该内存的指针，以后用普通的C或C++算子释放它，而不需要系统调用。通常，每个进程的指针有不同的值，这个值只有在那个进程中才有意义，但所指向的内存是一样的。和前面一样，先从System V共享内存开始介绍，然后再介绍POSIX共享内存。

7.13 System V共享内存

至此，你应该对System V IPC调用怎么工作很熟悉了。7.4节中讲到，如果愿意可以使用ftok得到关键字，使用Xget(shmget)调用来得到一个标识符。使用Xctl(shmctl)调用来控制它。在共享内存的情况下，可以用shmat调用把共享内存绑定到进程上，shmat将返回一个指针，当工作完毕时，可以调用shmdt来断开共享链接。

7.13.1 System V 共享内存系统调用

下面是System V共享内存系统调用的对照表:

shmget —— 得到共享内存段

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

int shmget(
    key_t key,           /* key */
    size_t size,        /* size of segment */
    int flags            /* creation flags */
);
/* Returns shared-memory identifier or -1 on error (sets errno) */
```

shmctl —— 控制共享内存段

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

int shmctl(
    int shmid,          /* identifier */
    int cmd,            /* command */
    struct shmids *data /* data for command */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

struct shmids —— shmctl的结构

```
struct shmids {
    struct ipc_perm shm_perm; /* permission structure */
    size_t shm_segsz;        /* size of segment in bytes */
    pid_t shm_lpid;         /* process ID of last shared memory op */
    pid_t shm_cpid;        /* process ID of creator */
    shmatt_t shm_nattch;    /* number of current attaches */
    time_t shm_atime;       /* time of last shmat */
    time_t shm_dtime;       /* time of last shmdt */
    time_t shm_ctime;       /* time of last change by shmctl */
};
```

shmat —— 连接共享内存段

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

void *shmat(
    int shmid,          /* identifier */
    const void *shmaddr, /* desired address or NULL */
    int flags           /* attachment flags */
);
/* Returns pointer or -1 on error (sets errno) */
```

shmdt —— 断开共享内存段

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

int shmdt(
    const void *shmaddr /* pointer to segment */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

像所期望的那样,调用shmget可以访问共享内存段,在需要时可以使用标志IPC_CREAT和IPC_PRIVATE创建它。size参数只在该段创建后才有意义。新创建的段初始值为0。

接下来,要使用该段,还要调用shmat来给该段一个指针。奇怪的是,出现错误时shmat返回的是-1而不是NULL值。之所以这样做是因为它不会返回会被误认为是-1的指针;事实上,几乎所有的UNIX系统中返回的指针都是偶数。

当使用该段工作完毕后，调用shmdt，并把从shmat得到的指针而不是标识符传进来。该段会一直存在直到被彻底清除（用shmctl或者ipcrm命令）或者机器重启，因此可以自由地重新连接它，但可能会得到不同的指针。

通常情况下，我们并不关心shmat会给我们什么样的地址，所以可以将第二个参数设定为NULL。但如果需要，也可以试着强制让它给你分配用shmaddr指定的地址。如果不行，是因为地址已经被占用或者是无效的，这样它将会失败，并把errno设置为EINVAL。还有一个标志SHM_RND，用来将地址四舍五入为合适的整数，但这里不详细介绍它了。另一个标志是SHM_RDONLY，设置它后，可以以只读方式连接此段。

使用shmctl和其他的Xctl系统调用，使用的是相同的命令，IPC_STAT、IPC_SET和IPC_RMID。IPC_SET设置shm_perm.uid、shm_perm.gid以及shm_perm.mode的低9位。

下面是一个简单的程序，显示了如何在两个进程之间共享内存段（小的段）：

```
static int *getaddr(void)
{
    key_t key;
    int shmid, *p;

    (void)close(open("shmseg", O_WRONLY | O_CREAT, 0));
    ec_negl( key = ftok("shmseg", 1) )
    ec_negl( shmid = shmget(key, sizeof(int), IPC_CREAT | PERM_FILE) )
    ec_negl( p = shmat(shmid, NULL, 0) )
    return p;

EC_CLEANUP_BGN
    return NULL;
EC_CLEANUP_END
}

int main(void)
{
    pid_t pid;

    if ((pid = fork()) == 0) {
        int *p, prev = 0;

        ec_null( p = getaddr() )
        while (*p != 99)
            if (prev != *p) {
                printf("child saw %d\n", *p);
                prev = *p;
            }
        printf("child is done\n");
    }
    else {
        int *p;

        ec_null( p = getaddr() )
        for (*p = 1; *p < 4; (*p)++)
            sleep(1);
        *p = 99;
    }
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```



每个进程都调用`getaddr`；其中一个创建了该段，其他的仅仅是访问它。下面是得到的输出：

```
$ shmex
child saw 1
child saw 2
child saw 3
$ child saw 99
child is done
```

有点奇怪，不是吗？为什么当`while`循环中断时子进程报告它看到了99呢？实际上，下行中：

```
while (*p != 99)
```

`*p`在这一点时等于3，但是当运行到这一行程序：

```
printf("child saw %d\n", *p);
```

`*p`已经是99了。（`$`提示的出现是因为父进程没有等子进程就退出了，这在UNIX操作系统中不是缺陷而是很常见的。）

如果再次运行`shmex`，还会出现更加奇怪的事：

```
$ shmex
child is done
$
```

程序终止了！不是，实际上错误在于该段在`shmex`第一次终止时状态没变——没有断开，其值仍然是99。所以当第二次执行时，甚至父进程在到达`for`循环之前，子进程已看到了99。显然当`shmex`终止时通过删除该段可以解决该问题。但要明白一点：共享内存需慎重对待！

7.13.2 共享内存和信号量

上一节讲的示例出错还有非常细微的原因：正如使用线程时所看到的（见5.17.3节），不能假定对`*p`的引用是原子的。一般来说，如果没有以信号量的形式做一些控制，就不能在进程间共享内存。因此，本节将讨论这个问题（`getaddr`没有改变）：

```
int main(void)
{
    pid_t pid;

    ec_false( SimpleSemRemove("shmexsem") )
    if ((pid = fork()) == 0) {
        struct SimpleSem *sem;
        int *p, prev = 0, n;

        ec_null( sem = SimpleSemOpen("shmexsem") )
        ec_null( p = getaddr() )
        while (true) {
            ec_false( SimpleSemWait(sem) )
            n = *p;
            ec_false( SimpleSemPost(sem) )
            if (n == 99)
                break;
            if (prev != n) {
                printf("child saw %d\n", n);
            }
        }
    }
}
```



```

        prev = n;
    }
}
printf("child is done\n");
ec_false( SimpleSemClose(sem) )
}
else {
    struct SimpleSem *sem;
    int *p, i;

    ec_null( sem = SimpleSemOpen("shmexsem" ) )
    ec_null( p = getaddr() )
    *p = 0;
    ec_false( SimpleSemPost(sem) )
    for (i = 1; i < 4; i++) {
        ec_false( SimpleSemWait(sem) )
        *p = i;
        ec_false( SimpleSemPost(sem) )
        sleep(1);
    }
    ec_false( SimpleSemWait(sem) )
    *p = 99;
    ec_false( SimpleSemPost(sem) )
    ec_false( SimpleSemClose(sem) )
}
exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}

```

下面是现在的输出——这个结果合理多了:

```

$ shmex2
child saw 1
child saw 2
child saw 3
$ child is done
shmex2
child saw 1
child saw 2
child saw 3
$ child is done

```

看看需要加以保护的所有变化:

- 子进程把*p分配给用信号量加锁的本地内存，然后自由地使用信号量解锁的本地内存;
- 类似地，父进程在for循环中用了个本地变量，锁定了信号量仅用于访问共享内存;
- 最初，信号量是锁定的(0值)，所以父进程可以自由地把共享内存初始值赋0。然后调用SimpleSemPost继续运行。如果在那一刻，子进程访问了共享内存，一切正常。这个版本可以重复运行，因为，每次运行时都将初始化内存段。
- 在每次运行开始时删除信号量，以便信号量从0开始。

虽然已经解决了原子问题，但子进程的效率仍然不高。再看一下循环找找原因:

```

while (true) {
    ec_false( SimpleSemWait(sem) )
    n = *p;
    ec_false( SimpleSemPost(sem) )
}

```

```
if (n == 99)
    break;
if (prev != n) {
    printf("child saw %d\n", n);
    prev = n;
}
}
```

子进程反复循环工作，对信号量加锁又解锁，但是仅仅在值改变时才执行一些任务（输出）。它的多数运行是多余的，所有的加锁使信号量无效而没有任何原因。当值变化时，父进程才告诉子进程，不是更好吗？（这就是在5.17.4节中使用条件变量的动机。）

可以用两个信号量代替一个信号量来修正该程序：为了向共享内存中写入，必须锁定信号量 W ；读共享内存时使用信号量 R 。父进程在写操作前等待 W ，在写完成后设置 R 。子进程在读操作前等待 R ，然后在完成后，设置 W 。开始时， W 初始化为1， R 初始化为0。下面是改进的代码（`getaddr`仍然未变化）：

```
int main(void)
{
    pid_t pid;

    ec_false( SimpleSemRemove("shmexsem") )
    if ((pid = fork()) == 0) {
        struct SimpleSem *semR, *semW;
        int *p, n;

        ec_null( semR = SimpleSemOpen("shmexsemR") )
        ec_null( semW = SimpleSemOpen("shmexsemW") )
        ec_null( p = getaddr() )
        while (true) {
            ec_false( SimpleSemWait(semR) )
            n = *p;
            ec_false( SimpleSemPost(semW) )
            if (n == 99)
                break;
            printf("child saw %d\n", n);
        }
        printf("child is done\n");
        ec_false( SimpleSemClose(semR) )
        ec_false( SimpleSemClose(semW) )
    }
    else {
        struct SimpleSem *semR, *semW;
        int *p, i;

        ec_null( semR = SimpleSemOpen("shmexsemR") )
        ec_null( semW = SimpleSemOpen("shmexsemW") )
        ec_null( p = getaddr() )
        *p = 0;
        ec_false( SimpleSemPost(semW) )
        for (i = 1; i < 4; i++) {
            ec_false( SimpleSemWait(semW) )
            *p = i;
            ec_false( SimpleSemPost(semR) )
            sleep(1);
        }
        ec_false( SimpleSemWait(semW) )
        *p = 99;
    }
}
```



```

    ec_false( SimpleSemPost(semR) )
    ec_false( SimpleSemClose(semR) )
    ec_false( SimpleSemClose(semW) )
}
exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}

```

可以看到，子进程不再用prev来判别什么时候共享内存会变化，因为现在如果没有变化，子进程永远不会得到semR。这种方法更有效！

7.13.3 System V的SMI共享内存实现

现在扩展上一节的想法用共享内存和信号量来实现SMI函数，我们已经用FIFO(见7.3.3节)和消息(见7.5.3节和7.7.2节)实现过了。

像前面示例讲过的那样，服务器和每一个客户端使用单独的共享内存段和两个信号量来接收消息。所以，如果有两个客户端，将有3个共享的内存段和6个信号量。这种方法没有消息队列——每一个内存段一次持有一个消息，直到消息接收完成前，不能写入新消息，并且会设定写信号量(W)。这不是最好的设计，因为每一个客户端只能和服务器的服务速度相同。但是，从这种方法可以看出共享内存是如何使用的，这就是我们的目的。

图7-3显示了该服务器和两个客户端。它们共用着共享内存段mem-server；客户端1和服务器共用段mem-1；客户端2和服务器共用段mem-2。在不移动消息的情况下，三个进程中的每一个都能访问共享内存段中的消息。

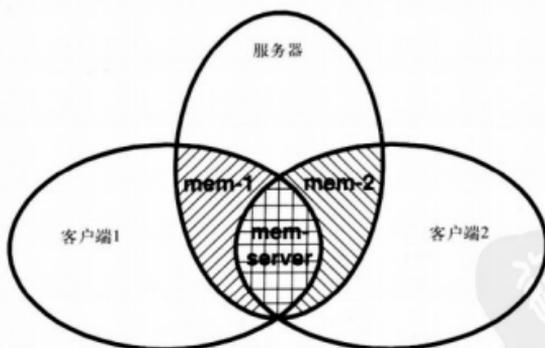


图7-3 带有两个客户端的服务器共享内存

在System V共享内存的SMI实现中，假定读者对本章的前几节讲过的FIFO的实现和消息队列的实现是熟悉的，所以对前面讲过的细节，这里不再详细解释。

回顾可知，SMI函数是专门为了允许在进程中适当位置处理消息而设计的；发送操作和接收操作被分成“getaddr”和“release”两个函数。在早期的实现中，它的好处不是很明显，但在这个实现中，将会看到它的优势。

对每一个共享内存段，我们将利用System V信号量的其中一个特征，并且使用一个包含

两个信号量的信号量集。信号量0是读信号量，1是写信号量。下面为那些数字和在其上操作的semwait与semop操作定义一些宏：

```
#define SEMI_READ 0
#define SEMI_WRITE 1
#define SEMI_POST 1
#define SEMI_WAIT -1
```

给定一个信号量，用函数op_semi对其进行操作。例如，为内存段设定写信号量，执行：

```
ec_negl( op_semi(semid_receiver, SEMI_WRITE, SEMI_POST) )
```

下面是op_semi的代码：

```
static int op_semi(int semid, int sem_num, int sem_op)
{
    struct sembuf sbuf;
    int r;

    sbuf.sem_num = sem_num;
    sbuf.sem_op = sem_op;
    sbuf.sem_flg = 0;
    ec_negl( r = semop(semid, &sbuf, 1) )
    return r;
}
EC_CLEANUP_BGN
return -1;
EC_CLEANUP_END
}
```

还有一个能方便初始化信号量的函数：

```
static int init_semi(int semid)
{
    union semun arg;
    int r;

    arg.val = 0;
    semctl(semid, SEMI_WRITE, SETVAL, arg);
    semctl(semid, SEMI_READ, SETVAL, arg);
    /* Following call will set otime, allowing clients to proceed. */
    ec_negl( r = op_semi(semid, SEMI_WRITE, SEMI_POST) )
    return r;
}
EC_CLEANUP_BGN
return -1;
EC_CLEANUP_END
}
```

下面是SMI使用的SMIQ类型的内部数据结构：

```
typedef struct {
    SMIENTITY sq_entity;          /* entity */
    int sq_semid_server;         /* server sem */
    int sq_semid_client;        /* client sem (client only) */
    int sq_shmid_server;        /* server shm ID */
    int sq_shmid_client;        /* client shm ID (client only) */
    struct smi_msg *msg_server; /* ptr to server shm */
    struct smi_msg *msg_client; /* ptr to client shm (client only) */
    char sq_name[SERVER_NAME_MAX]; /* server name */
    struct client_id sq_client; /* client identification (server only) */
} SMIQ_SHM;
```

客户端几乎要使用整个结构，但是服务器仅仅需要使用其中一些成员。客户端保存的内

容是:

- 客户端的实体 (SMI_CLIENT) 和服务器名字;
- 客户端和服务器的信号量集ID (每个信号量集有2个信号量);
- 客户端和服务器的共享内存段ID;
- 指向客户端和服务器的共享内存段的指针 (从shmat中)。

服务器保存的内容:

- 服务器的实体 (SMI_SERVER) 和名字;
- 它的信号量集ID (2个信号量);
- 它的共享内存段ID;
- 指向服务器段的指针。
- 向smi_send_getaddr传递client_id, 以便可以在随后的smi_send_release中使用。这样它能知道要向哪一个客户端发送。为此, 在消息中, client_id结构中的成员c_id1是共享内存的标识符, 而成员c_id2是信号量集的标识符。(可以在下面的smi_send_getaddr_shm代码中看到这些成员在哪里设置。)

服务器不保存任何客户端的信号量集或共享内存的信息, 因为有许多客户端。这些信息可以很轻松地通过从客户端接收的消息传递过来, 就如在System V消息队列SMI实现时, 消息队列ID在消息中传递的方式一样 (见7.5.3节)。很快就会看到这些细节。

在说明了结构如何使用后, smi_open_shm的代码应该是很容易理解的:

```
SMIQ *smi_open_shm(const char *name, SMIENTITY entity, size_t msgsize)
{
    SMIQ_SHM *p = NULL;
    char shmname[FILENAME_MAX];
    int i;
    key_t key;

    ec_null( p = calloc(1, sizeof(SMIQ_SHM)) )
    p->sq_entity = entity;
    if (strlen(name) >= SERVER_NAME_MAX) {
        errno = ENAMETOOLONG;
        EC_FAIL
    }
    strcpy(p->sq_name, name);
    mkshm_name_server(p, shmname, sizeof(shmname));
    (void)close(open(shmname, O_WRONLY | O_CREAT, 0));
    ec_negl( key = ftok(shmname, 1) )
    if (p->sq_entity == SMI_SERVER) {
        if ((p->sq_semid_server = semget(key, 2, PERM_FILE)) != -1)
            (void)shmctl(p->sq_semid_server, IPC_RMID, NULL);
        ec_negl( p->sq_semid_server = semget(key, 2,
            PERM_FILE | IPC_CREAT) )
        p->sq_semid_client = -1;
        if ((p->sq_shmid_server = shmget(key, 0, PERM_FILE)) != -1)
            (void)shmctl(p->sq_shmid_server, IPC_RMID, NULL);
        ec_negl( p->sq_shmid_server = shmget(key, msgsize,
            PERM_FILE | IPC_CREAT) )
        p->sq_shmid_client = -1;
        ec_negl( init_semi(p->sq_semid_server) )
    }
    else {
        ec_negl( p->sq_semid_server = semget(key, 2, PERM_FILE) )
        ec_negl( p->sq_semid_client = semget(IPC_PRIVATE, 2,
            PERM_FILE | IPC_CREAT) )
    }
}
```

```

ec_negl( p->sq_shmid_server = shmget(key, msgsize, PERM_FILE) )
ec_negl( p->sq_shmid_client = shmget(IPC_PRIVATE, msgsize,
    PERM_FILE | IPC_CREAT) )
ec_negl( p->msg_client = shmat(p->sq_shmid_client, NULL, 0) )
ec_negl( init_semi(p->sq_semid_client) )
for (i = 0; !smi_client_nowait && i < 10; i++) {
    union semun arg;
    struct semid_ds ds;

    arg.buf = &ds;
    ec_negl( semctl(p->sq_semid_server, SEMI_WRITE, IPC_STAT,
        arg) )
    if (ds.sem_otime > 0)
        break;
    sleep(1);
}
ec_negl( p->msg_server = shmat(p->sq_shmid_server, NULL, 0) )
return (SMIQ *)p;

EC_CLEANUP_BGN
    free(p);
    return NULL;
EC_CLEANUP_END
}

```

通过对ftok的调用，代码的第一部分和前面7.5.3节中针对System V消息队列的该函数的代码几乎一样。对服务器来说，如果有旧的信号量设置，就删除它并创建一个新的，对于共享内存段也是一样。客户端创建了私有的信号量集和共享内存段。它们都连接到了服务器的共享内存段。

注意init_semi（前面提到的），它设置了写信号量（允许继续发送），也设置了操作次数，这样，其他的进程便可以判别出信号量已经被初始化了，就像7.9.1节所讲的一样。在那节，曾提出了实现SimpleSem的一个等待算法；这里的算法更精细了：

- 因为FreeBSD和Darwin系统不设置次数，并且也不想被阻塞，所以我们最多尝试10次。
- 有一个隐藏的全局变量smi_client_nowait，它可以用于阻止任何等待。当知道服务器在所有客户端之前正常启动时，在计时测试期间才使用它。本章结尾的比较表中使用了程序中产生的数据。

smi_close_shm十分简单：

```

bool smi_close_shm(SMIQ *sqp)
{
    SMIQ_SHM *p = (SMIQ_SHM *)sqp;

    if (p->sq_entity == SMI_SERVER) {
        char shmname[FILENAME_MAX];

        (void)getaddr(-1);
        ec_negl( semctl(p->sq_semid_server, 0, IPC_RMID) )
        (void)shmdt(p->msg_server);
        (void)shmctl(p->sq_shmid_server, IPC_RMID, NULL);
        mkshm_name_server(p, shmname, sizeof(shmname));
        (void)unlink(shmname);
    }
    else {
        ec_negl( semctl(p->sq_semid_client, 0, IPC_RMID) )
        (void)shmdt(p->msg_server);
        (void)shmdt(p->msg_client);
    }
}

```



```

        (void)shmctl(p->sq_shmid_client, IPC_RMID, NULL);
    }
    free(p);
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

```

现在准备讲述smi_send_getaddr_shm:

```

bool smi_send_getaddr_shm(SMIQ *sqp, struct client_id *client,
    void **addr)
{
    SMIQ_SHM *p = (SMIQ_SHM *)sqp;
    int semid_receiver;

    if (p->sq_entity == SMI_SERVER) {
        semid_receiver = client->c_id2;
        p->sq_client = *client;
    }
    else
        semid_receiver = p->sq_sem_server;
    ec_negl( op_semi(semid_receiver, SEMI_WRITE, SEMI_WAIT) )
    if (p->sq_entity == SMI_SERVER)
        ec_null( *addr = getaddr(client->c_id1) )
    else {
        *addr = p->msg_server;
        ((struct smi_msg *)*addr)->smi_client.c_id1 = p->sq_shmid_client;
        ((struct smi_msg *)*addr)->smi_client.c_id2 = p->sq_sem_server;
    }
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

```

对于从服务器的发送，参数client必须指向客户端标识符；而对于从客户端的发送，该参数为NULL。先跟踪用于服务器的代码，然后是客户端的代码。

如先前所提到的，对于服务器来说，通过成员c_id2来设置semid_receiver，而且保存整个client_id结构以便以后由smi_send_release使用。然后等待SEMI_WRITE信号量。已经在init_semi中设置了它，所以，可以立刻开始运行。消息本身处在由client->c_id1给出的共享内存段中，可以调用getaddr得到它的地址，该地址就是通过addr参数返回的结果。稍后就可以看到getaddr；现在，可以认为它只运行shmat的情况。

对客户端来说，实际上更简单，因为客户端已经得到了访问它自身和服务器的共享内存段以及信号量所需要的所有东西。它直接从SMIQ_SHM结构中设置semid_receiver，然后等待SEMI_WRITE信号量，设置返回地址，然后将标识符保存在其共享内存段和信号量设置的消息中。这样接收者（服务器）就知道是谁在发送消息以及如何进行应答了。

一旦smi_send_getaddr_shm返回了，它的调用者就可以任意使用该消息的返回地址，而那段内存被锁定了以防其他客户端进一步的写操作。（效率有点低，前面提过了——如果设置一个接收服务器的消息池应该会好些，当然实现起来会更复杂。）

当调用者执行完成时，它会调用smi_send_release:

```

bool smi_send_release_shm(SMIQ *sqp)
{
    SMIQ_SHM *p = (SMIQ_SHM *)sqp;
    int semid_receiver;

    if (p->sq_entity == SMI_SERVER)
        semid_receiver = p->sq_client.c_id2;
    else
        semid_receiver = p->sq_semid_server;
    ec_negl( op_semi(semid_receiver, SEMI_READ, SEMI_POST) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

```

该函数需要做的内容是设置SEMI_READ信号量，但是那样做需要信号量集标识符。对服务器来说，标识符是在client_id中，而smi_send_getaddr将client_id保存在了SMIQ_SHM结构中；对于客户端，在一开始就把它放在了那个结构中。

现在，应该能够很容易地理解smi_receive_getaddr_shm了：

```

bool smi_receive_getaddr_shm(SMIQ *sqp, void **addr)
{
    SMIQ_SHM *p = (SMIQ_SHM *)sqp;
    int semid_receiver;

    if (p->sq_entity == SMI_SERVER)
        semid_receiver = p->sq_semid_server;
    else
        semid_receiver = p->sq_semid_client;
    ec_negl( op_semi(semid_receiver, SEMI_READ, SEMI_WAIT) )
    if (p->sq_entity == SMI_SERVER)
        *addr = p->msg_server;
    else
        *addr = p->msg_client;
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

```

响应函数smi_receive_release_shm必须设置SEMI_WRITE信号量：

```

bool smi_receive_release_shm(SMIQ *sqp)
{
    SMIQ_SHM *p = (SMIQ_SHM *)sqp;
    int semid_receiver;

    if (p->sq_entity == SMI_SERVER)
        semid_receiver = p->sq_semid_server;
    else
        semid_receiver = p->sq_semid_client;
    ec_negl( op_semi(semid_receiver, SEMI_WRITE, SEMI_POST) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

```

就这样了。对信号量的解释已经很多了，但却很少有关于共享内存的。如果消息非常大（如，100 000字节），就能获得巨大的成功。消息队列（不论System V还是POSIX）可能甚至不能处理那么大的消息，即使能够实现，对每一条消息，从用户空间复制到内核，然后再从内核复制到用户空间实际上是很慢的。相反，共享内存SMI实现的速度是独立于消息的大小的。

7.13.4 评论System V共享内存

System V共享内存的系统调用是相当容易理解的、有效率的，而且通常很容易实现，因此不要太讨厌它。困难的部分是同步问题，它也是使用线程的难点，而且是因为完全相同的原因：一旦共享了什么东西，程序运行就快了，但可能会出现错误，而改正错误的工作是非常难于测试的。必须要证明它是正确的，然后毫无错误地实现它。[Ⓔ]

7.14 POSIX共享内存

这一节将介绍POSIX共享内存系统调用，并介绍一种使用POSIX共享内存和POSIX信号量实现的SMI。

7.14.1 POSIX共享内存系统调用

POSIX共享内存涉及使用POSIX IPC名字来打开（或许还创建）共享内存段，这可能会带来7.6.2节中所描述的POSIX IPC名字的麻烦问题。shm_open调用返回了一个文件描述符，就像已经打开了文件一样。事实上，POSIX共享内存段更像一个内存文件。一旦打开，就可以用在2.17节中见过的ftruncate来设置大小，就像正在设置文件大小一样。然后把这个内存段用mmap映射为地址空间，mmap通常用于映射文件，而不仅是共享内存段。换句话说，只有shm_open和shm_unlink是POSIX共享内存所特有的，其他调用对于所有常规文件都可以使用。

下面是共享内存特有的调用：

shm_open —— 打开共享内存对象

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

int shm_open(
    const char *name,      /* POSIX IPC name */
    int flags,            /* flags */
    mode_t perms          /* permissions */
);
/* Returns file descriptor or -1 on error (sets errno) */
```

shm_unlink —— 删除共享内存对象

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

int shm_unlink(
    const char *name      /* POSIX IPC name */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

那些shm_open的标志是在面向文件的系统调用中已经见过的标志：O_CREAT、

Ⓔ 大约在30年前，我的一个同事曾说过如果程序不需要必须是正确的，他可以让程序的速度任意快——他将改变程序使其只输出0，然后扔掉所有其他代码。因此，如果不能确保共享内存或线程的使用是正确的，那么就不要使用这些特性——虽然慢点但只要正确就会更有效。

O_EXCL、O_TRUNC、O_RDONLY和O_RDWR。不能使用O_WRONLY。如果对象是新建的，那么第三个参数会建立它的权限。

和其他的POSIX及System V IPC对象一样，POSIX 共享内存对象的内容将持续存在，至少直到系统重新启动。

当使用close处理它时，可以关闭该文件描述符，就像其他所有文件描述符一样。

[SUS2002]没有讲可以在共享内存对象上实现I/O，例如使用read和write，对于实现，可能是允许那样做。像任何其他文件一样，这可能是一种使用内存文件的方法。使用内存文件的整个要点在于：在该文件上可以使用一般的C或C++操作，无需使用I/O系统调用，因此，即使支持这种特性，也不是特别有用。

通常，在新建一个共享内存对象以后，可以使用ftruncate设置其大小，因为其原始大小是0字节。下面是对2.17节中的ftruncate的对照表的回顾：

ftruncate —— 通过文件描述符截短或加长文件

```
#include <unistd.h>

int ftruncate(
    int fd,           /* file descriptor */
    off_t length     /* new length */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

接下来，将该对象映射到地址空间（类似于在System V 共享内存对象上使用shmat）：

mmap —— 映射内存页

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

void *mmap(
    void *addr,      /* desired address or NULL */
    size_t len,     /* length of segment */
    int prot,       /* protection (see below) */
    int flags,      /* flags */
    int fd,         /* file descriptor */
    off_t off       /* offset in file or shared-memory object */
);
/* Returns pointer to segment or MAP_FAILED on error (sets errno) */
```

第一个参数addr是该段映射地址附近的一个地址，或者更准确地说，如果在flags参数中设置了MAP_FIXED，它就是该段应该被映射的地址。否则，它就是NULL。这就意味着你将获得任意地址，这是最常见的情况。在本书的例子中，将采用这种方式来处理。

被映射的对象部分在对象内部开始于off并扩展了len个字节。没有必要一次映射整个对象（大小由ftruncate设置），尽管在本例中是那样做的，因此off将设为零，而len将和ftruncate中的参数一样。

参数prot或者是prot_none（即根本不能访问内存），或者是下面一个或多个标志的或操作：

PROT_READ 数据可读。

PROT_WRITE 数据可写。

PROT_EXEC 数据可执行（可能不支持）。

为了达到目的，这里需要使用PROT_READ|PROT_WRITE。同时，对这些标志参数来说，将要使用的唯一标志是MAP_SHARED，这意味着对段的任何改变都将立即可见。

MAP_PRIVATE是可选项，它表示改变对调用mmap进程是私有的。MAP_PRIVATE 对于共享内存来说并没有意义。

mmap是那些返回指针的函数中的另一个，它有一个针对错误返回的专门符号——MAP_FAILED。要确保对其进行测试且值不能为NULL。

因此，所有上述内容是指如果想要映射共享内存的所有对象进行读与写，那么可以如下操作：

```
ec_negl( ftruncate(fd, len) )
mem = mmap(NULL, len, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
ec_cmp(mem, MAP_FAILED)
```

当完成了段的映射时，要取消对它的映射：

munmap——取消内存页映射

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

int munmap(
    void *addr,      /* pointer to segment */
    size_t len       /* length of segment */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

没有必要取消 (unmap) 所有已映射的段，但是在本书所有例子中都会这么做。因此，addr将是mmap所返回的值，而len则等于传递给mmap的值。

7.14.2 POSIX的SMI共享内存实现

除了代替不同的信号量对象外，SMI函数的POSIX共享内存实现和System V共享内存的实现（见7.13.3节。在继续之前有必要温习一下）非常相似，我们打算利用POSIX信号量特性并使用内存中的信号量来实现。因为它们必须由服务器和客户端共享，所以将其放入共享内存段。

更准确地说，每一个共享内存段（一个用作服务器，一个用作客户端）均有如下结构：

```
struct shared_mem {
    sem_t sm_sem_w;
    sem_t sm_sem_r;
    struct smi_msg sm_msg; /* variable size -- must be last */
};
```

smi_msg的数据部分扩展到了段的尾部，其大小由smi_open_pshm使用如下宏所传递的数据来计算：

```
#define MEM_SIZE(s)\
    (sizeof(struct shared_mem) - sizeof(struct smi_msg) + (s))
```

这里s是smi_open_pshm的第三个参数。

```
#define SEMI_READ    0
```

给定一个指针指向共享内存中的struct shared_mem，下面是一个很方便的函数，它可以执行semwait、sempost或撤消操作：

```
#define SEMI_WRITE    1
#define SEMI_DESTROY  2
#define SEMI_POST     1
```

```

#define SEMI_WAIT      -1

static int op_semi(struct shared_mem *m, int sem_num, int sem_op)
{
    sem_t *sem_p = NULL;

    if (sem_num == SEMI_WRITE)
        sem_p = &m->sm_sem_w;
    else
        sem_p = &m->sm_sem_r;
    switch (sem_op) {
    case SEMI_WAIT:
        ec_negl( sem_wait(sem_p) )
        break;
    case SEMI_POST:
        ec_negl( sem_post(sem_p) )
        break;
    case SEMI_DESTROY:
        ec_negl( sem_destroy(sem_p) )
    }
    return 0;

EC_CLEANUP_BGN
    return -1;
EC_CLEANUP_END
}

```

因此，如果m是这样指针，那么我们就能够像下面这样调用：

```

ec_negl( op_semi(m, SEMI_READ, SEMI_POST) )
ec_negl( op_semi(m, SEMI_WRITE, SEMI_WAIT) )

```

POSIX调用不允许客户端随同消息一起只为共享内存段传递一个标识符给服务器。相反，必须用与7.3.3节中讲述的FIFO的类似方法：客户端传递它的进程ID，服务器使用它形成对象的POSIX名字，然后打开并映射它。由于这样做开销较大，所以服务器对每个客户端只做一次，并在表中查询一个已经映射的段。所有的这些都被存储在SMIQ_PSHM结构中，该结构以一个目前大家已经很熟悉的形式出现：

```

#define MAX_CLIENTS 50

typedef struct {
    SMENTITY sq_entity;           /* entity */
    char sq_name[SERVER_NAME_MAX]; /* server name */
    int sq_srv_fd;                /* server shm file descriptor */
    struct shared_mem *sq_srv_mem; /* server mapped shm segment */
    struct client {
        pid_t cl_pid;           /* client process ID */
        int cl_fd;              /* client shm file descriptor */
        struct shared_mem *cl_mem; /* client mapped shm segment */
    } sq_clients[MAX_CLIENTS]; /* client uses only [0] */
    struct client_id sq_client; /* client id (server only) */
    size_t sq_msgsize;         /* message size */
} SMIQ_PSHM;

```

服务器能跟踪50个客户。在实现中，客户端完全消失是客户端通知服务器它已完成的一种方法，以便服务器能重新利用它的位置。

现在，来看看SMIQ_PSHM结构是如何由smi_open_pshm配置的：

```

SMIQ *smi_open_pshm(const char *name, SMIENTITY entity, size_t msgsize)
{
    SMIQ_PSHM *p = NULL;
    char shmname[SERVER_NAME_MAX + 50];

    ec_null( p = calloc(1, sizeof(SMIQ_PSHM)) )
    p->sq_entity = entity;
    p->sq_msgsize = msgsize;
    if (strlen(name) >= SERVER_NAME_MAX) {
        errno = ENAMETOOLONG;
        EC_FAIL
    }
    strcpy(p->sq_name, name);
    mkshm_name_server(p, shmname, sizeof(shmname));
    if (p->sq_entity == SMI_SERVER) {
        if ((p->sq_srv_fd = shm_open(shmname, O_RDWR, PERM_FILE)) != -1) {
            (void)shm_unlink(shmname);
            (void)close(p->sq_srv_fd);
        }
        ec_negl( p->sq_srv_fd = shm_open(shmname, O_RDWR | O_CREAT,
            PERM_FILE) )
        ec_negl( ftruncate(p->sq_srv_fd, MEM_SIZE(msgsize)) )
        p->sq_srv_mem = mmap(NULL, MEM_SIZE(msgsize),
            PROT_READ | PROT_WRITE, MAP_SHARED, p->sq_srv_fd, 0);
        ec_cmp(p->sq_srv_mem, MAP_FAILED)
        ec_negl( sem_init(&p->sq_srv_mem->sm_sem_w, true, 1) )
        ec_negl( sem_init(&p->sq_srv_mem->sm_sem_r, true, 0) )
    }
    else {
        ec_negl( p->sq_srv_fd = shm_open(shmname, O_RDWR, PERM_FILE) )
        p->sq_srv_mem = mmap(NULL, MEM_SIZE(msgsize),
            PROT_READ | PROT_WRITE, MAP_SHARED, p->sq_srv_fd, 0);
        ec_cmp(p->sq_srv_mem, MAP_FAILED)
        mkshm_name_client(getpid(), shmname, sizeof(shmname));
        if ((p->sq_clients[0].cl_fd = shm_open(shmname, O_RDWR, PERM_FILE))
            != -1) {
            (void)shm_unlink(shmname);
            (void)close(p->sq_clients[0].cl_fd);
        }
        ec_negl( p->sq_clients[0].cl_fd = shm_open(shmname,
            O_RDWR | O_CREAT, PERM_FILE) )
        ec_negl( ftruncate(p->sq_clients[0].cl_fd, MEM_SIZE(msgsize)) )
        p->sq_clients[0].cl_mem = mmap(NULL, MEM_SIZE(msgsize),
            PROT_READ | PROT_WRITE, MAP_SHARED, p->sq_clients[0].cl_fd, 0);
        ec_cmp(p->sq_clients[0].cl_mem, MAP_FAILED)
        ec_negl( sem_init(&p->sq_clients[0].cl_mem->sm_sem_w, true,
            1) )
        ec_negl( sem_init(&p->sq_clients[0].cl_mem->sm_sem_r, true,
            0) )
    }
    return (SMIQ *)p;
}
EC_CLEANUP_BGN
if (p != NULL)
    (void)smi_close_pshm((SMIQ *)p);
return NULL;
EC_CLEANUP_END
}

static void mkshm_name_server(const SMIQ_PSHM *p, char *shmname,
    size_t shmname_max)

```

```

{
    snprintf(shmname, shmname_max, "/smipshm-%s", p->sq_name);
}

static void mkshm_name_client(pid_t pid, char *shmname,
    size_t shmname_max)
{
    snprintf(shmname, shmname_max, "/smipshm-%d", pid);
}

```

上述代码的前几行（一直到调用mkshm_name_server为止）和以前所讲的实现一样。然后服务器新建了一个全新的共享内存对象（删除了旧的），设置其大小，并映射它。一旦完成这些，那些信号量便在内存中了，而且已经初始化了。

客户端也映射到了服务器段内，但它既不设置其大小也不初始化服务器的信号量，因为服务器已经做了那些工作。尽管如此，客户端确实要新建、设置、映射它自己的段（客户端要使用数组的第一个元素来映射该段），而且也要初始化它自己的信号量。

正如所说过的，因为服务器事先并不知道客户端可能会是谁，所以直到从客户端得到了消息，服务器才会在客户端的段内映射。这里有一个函数（下面马上就会见到），服务器可以使用它得到客户端段的映射地址，而它只需要知道客户的进程ID，就像前面见过那样，该进程ID被包含在从客户端发送到服务器的每条消息中：

```

static struct client *get_client(SMIQ_PSHM *p, pid_t pid)
{
    int i, avail = -1;
    char shmname[SERVER_NAME_MAX + 50];

    for (i = 0; i < MAX_CLIENTS; i++) {
        if (p->sq_clients[i].cl_pid == pid)
            return &p->sq_clients[i];
        if (p->sq_clients[i].cl_pid == 0 && avail == -1)
            avail = i;
    }

    if (avail == -1) {
        errno = EADDRNOTAVAIL;
        EC_FAIL
    }

    p->sq_clients[avail].cl_pid = pid;
    mkshm_name_client(pid, shmname, sizeof(shmname));
    ec_neg1( p->sq_clients[avail].cl_fd = shm_open(shmname, O_RDWR,
        PERM_FILE) )
    p->sq_clients[avail].cl_mem = mmap(NULL, MEM_SIZE(p->sq_msgsize),
        PROT_READ | PROT_WRITE, MAP_SHARED, p->sq_clients[avail].cl_fd,
        0);
    ec_cmp(p->sq_clients[avail].cl_mem, MAP_FAILED)
    return &p->sq_clients[avail];

EC_CLEANUP_BGN
    return NULL;
EC_CLEANUP_END
}

```

为了解使用中的get_client，来看看下面的smi_send_getaddr_pshm：

```

bool smi_send_getaddr_pshm(SMIQ *scp, struct client_id *client,
    void **addr)
{

```

```

SMIQ_PSHM *p = (SMIQ_PSHM *)sqp;
struct client *cp;
struct shared_mem *sm;

if (p->sq_entity == SMI_SERVER) {
    p->sq_client = *client;
    ec_null( cp = get_client(p, client->c_id1) )
    sm = cp->cl_mem;
}
else
    sm = p->sq_srv_mem;
ec_negl( op_semi(sm, SEMI_WRITE, SEMI_WAIT) )
if (p->sq_entity == SMI_CLIENT)
    sm->sm_msg.smi_client.c_id1 = getpid();
*addr = &sm->sm_msg;
return true;

EC_CLEANUP_BGN
return false;
EC_CLEANUP_END
}

```

回顾可知，参数client仅在服务器发送消息时才被使用；它从其前面获得的消息中得到该结构。c_id1成员是进程ID并且那是要传递给get_client的内容。对于客户端而言，服务器的段地址正好在SMIQ_PSHM结构中。借助段地址，可以等待写信号量，并且，当它可用时，该段便由我们来使用。客户端然后将进程ID存储在消息中，并且返回该地址。

和它相应的是smi_send_release_pshm:

```

bool smi_send_release_pshm(SMIQ *sqp)
{
    SMIQ_PSHM *p = (SMIQ_PSHM *)sqp;
    struct client *cp;
    struct shared_mem *sm;

    if (p->sq_entity == SMI_SERVER) {
        ec_null( cp = get_client(p, p->sq_client.c_id1) )
        sm = cp->cl_mem;
    }
    else
        sm = p->sq_srv_mem;
    ec_negl( op_semi(sm, SEMI_READ, SEMI_POST) )
    return true;

    EC_CLEANUP_BGN
    return false;
    EC_CLEANUP_END
}

```

当服务器调用此函数时，该函数就会调用get_client通过使用smi_send_getaddr_pshm保存的进程ID(c_id1成员)来得到客户端的段地址。客户端的段已经被smi_send_getaddr_pshm映射——这里只需要查询它就可以了。作为一个客户端，就像以前一样，服务器的段正好在SMIQ_PSHM结构中。一旦拥有这个段，唯一要做的工作就是设置段内的读信号量。这将允许smi_receive_getaddr_pshm在段内执行，就像下面将要看到的这样：

```

bool smi_receive_getaddr_pshm(SMIQ *sqp, void **addr)
{

```

```

SMIQ_PSHM *p = (SMIQ_PSHM *)sqp;
struct shared_mem *sm;

if (p->sq_entity == SMI_SERVER)
    sm = p->sq_srv_mem;
else
    sm = p->sq_clients[0].cl_mem;
ec_negl( op_semi(sm, SEMI_READ, SEMI_WAIT) )
*addr = &sm->sm_msg;
return true;

EC_CLEANUP_BGN
return false;
EC_CLEANUP_END
}

```

因为服务器和客户端各自都从自己的段接收，所以地址是可得到的并能正确传递给 `op_semi`。没有必要调用 `get_client`。一旦段可读，就将返回地址。

最后，一旦接收者完成了读操作，`smi_receive_release_pshm`就能为接收段设置写信号量：

```

bool smi_receive_release_pshm(SMIQ *sqp)
{
    SMIQ_PSHM *p = (SMIQ_PSHM *)sqp;
    struct shared_mem *sm;

    if (p->sq_entity == SMI_SERVER)
        sm = p->sq_srv_mem;
    else
        sm = p->sq_clients[0].cl_mem;
    ec_negl( op_semi(sm, SEMI_WRITE, SEMI_POST) )
    return true;

    EC_CLEANUP_BGN
    return false;
    EC_CLEANUP_END
}

```

如果你还没有把这个实现和7.13.3节中的System V的共享内存实现进行比较，那么现在就应该做了。你会发现它们非常相似，但也有如下不同之处：

- 对于System V调用，客户端可以传入每一个消息共享内存段和信号量设置的标识符，这样服务器可以利用它们来快速访问对象。因此，服务器不需要用来跟踪客户端。
- 对于POSIX调用，可以使用内存中的信号量（其位于共享内存中），而不是单个的信号量对象。理论上说，这应该会使速度更快，但快与否依赖于具体的实现。

7.14.3 评论POSIX共享内存

对共享内存而言，POSIX接口和System V接口基本同样方便。至于说哪个更有效要取决于具体的实现，但是，具有两者的任何实现在内核中很可能会使用相同的内部机制。

POSIX共享内存的关键优势在于其映射调用 `mmap`，它不仅适用于 `shm_open` 所打开的内存文件，还适用于所有常规文件。因此，在任何时刻，映射段的那一部分都会有很多的控制。

POSIX共享内存的主要缺点在于它并不总是可用的。例如，在Linux、FreeBSD或Darwin的某些版本中，它是不可用的。

7.15 性能比较

为6种不同的IPC方法（包括第8章的套接字）使用SMI实现，很容易构造一个测试程序来比较发送不同大小的消息的时间。作为测试，我在4个客户端和一个服务器之间，发送了5 000条消息。然后，为4个系统（Solaris、FreeBSD、Darwin和Linux）给出了归一化的数字，该数字是通过将原数字除以在各自系统中使用100字节的消息所用的FIFO时间得到的。否则，结果可能会出现误导，因为4个系统在计算机硬件上的差异会导致性能上的差异。表7-2列出了结果。仅仅通过共享内存和POSIX消息队列，便可以发送两个大容量的消息。同时，在编写本书时，FreeBSD、Darwin和Linux还不支持POSIX消息队列或POSIX共享内存。

表7-2 不同的消息传递方法的性能

方 法	100字节的消息	2 000字节的消息	20 000字节的消息	100 000字节的消息
FIFO	S:1.00	S:1.22	太大了	太大了
	B:1.00	B:1.46		
	D:1.00	D:1.29		
	L:1.00	L:1.51		
System V消息队列	S:0.90	S:1.82	太大了	太大了
	B:0.62	B:3.76		
	L:0.31	L:0.64		
POSIX消息队列	S:2.02	S:2.40	S:7.03	S:33.39
System V共享内存	S:1.47	S:1.41	S:1.55	S:1.24
	B:0.94	B:0.91	B:0.90	B:0.90
	D:1.04	D:1.07	D:1.02	D:1.06
	L:0.55	L:0.53	L:0.47	L:0.51
POSIX共享内存	S:1.27	S:1.25	S:1.41	S:1.37
套接字	S:1.84	S:2.15	S:10.15	S:44.13
	B:0.81	B:1.00	B:7.99	B:35.83
	D:1.04	D:1.27	D:5.52	D:25.86
	L:0.75	L:0.95	L:6.06	L:31.91

S: Solaris; B: FreeBSD; D: Darwin; L: Linux。对每个系统时间都进行了归一化，因此传递100字节消息所用的FIFO时间是1.00。套接字使用了AF_UNIX域（见第8章）。

对结果的一些解释：

- 因为这仅仅是一个可能的测试，所以结果不是确定的，SMI仅仅是一个可能的接口，并且这里的实现并不是最佳的——主要是设计它们来作为教科书示例。
- 即使是归一化的，对于一个给定的方法，拥有较快速度的系统并不一定能更好地实现那个方法。它可能实现FIFO（归一化的除数）会更慢些。
- 对小消息（100字节左右），在除了Darwin6.6以外的所有的系统中，System V消息队列性能最好。Darwin6.6不支持小消息。
- 套接字几乎和两种消息队列方法的性能一样（在某些情况下甚至更好），并能处理任何大小的消息。另外，它们是机器间传递消息的唯一方法，也包括Internet中的机器，尽管时间测试只在单个机器中使用它们。
- 在Solaris上，POSIX消息队列的性能没有System V消息队列性能好，但是，它有能处理很大消息的优势。
- 正如所期望的那样，那两个共享内存方法的运行情况与消息大小没有关系。

因此，如果必须做一个总结归纳，那么我们会说，为了取得最好的性能，可以将System V消息队列用于小消息，而将共享内存用于大消息。没有尝试的想法是，把两者结合：利用System V消息来告诉服务器它什么时候有工作可做，而利用共享内存段传递数据。

如果最佳性能不是很关键，而且希望简化，则可以对所有东西都使用套接字。它们相当有效，能控制任何大小的消息，普遍被支持，并且能在机器之间传递。

这些建议只是对面向消息的IPC而言的。对其他目地的用处，一种IPC方法可能比另一个更好。

练习

前6个练习要求用POSIX IPC函数或相关的方法来实现System V IPC函数。在大多数情况下，并不能够实现每个特性和行为，因此该练习的一部分是为了仔细准确地评论你的实现的不足。

- 7.1 用POSIX IPC函数实现msgget、msgctl、msgsnd以及msgrcv。
- 7.2 用System V IPC函数实现mq_open、mq_close、mq_unlink、mq_send以及mq_receive。
- 7.3 使用POSIX IPC函数实现semget、semctl以及semop，且每个设置只允许一个信号量。你能处理大于1的增加与减少吗？
- 7.4 与上一个练习一样，但每个设置中允许信号量的个数大于1。
- 7.5 使用System V IPC函数实现sem_open、sem_close、sem_unlink、sem_post和sem_wait。
- 7.6 使用POSIX IPC函数实现shmget、shmctl、shmat和shmdt。
- 7.7 并没有练习要求用System V IPC函数实现shm_open、shm_unlink、ftruncate、mmap和munmap。这是为什么呢？你可以提出一个恰当的练习吗？你能完成它吗？
- 7.8 设计并做实验：比较在普通文件上利用read和write来进行I/O与借助内存映射文件访问的效率。包括顺序I/O和随机I/O，而且可能每个都有几个变体。
- 7.9 设计并做实验来比较只用写锁（使用lockf）的效率和和使用读写锁结合（使用fcntl）的效率。尝试创建一种读写方式来显示它们在效率上的最大不同。
- 7.10 扩展在练习5.14中所写的程序，使它包括在本章解释的附录A中的进程属性。



第8章 网络和套接字

第6章和第7章讨论的IPC机制的确有用途，但许多现代的程序不仅仅要在同一台机器的进程之间传送数据，还需要在不同的机器之间传送数据。“在不同机器”不仅仅指在跨越一个房间或同一栋楼中的机器，而是指世界上的任何使用Internet的机器。

这种在不同机器之间运行的基本机制即联网——称为“套接字”。共有8个基本的套接字系统调用，其中5个都是套接字所独有的：`socket`、`bind`、`listen`、`accept`、`connect`、`read`、`write`和`close`。然而，因为套接字（特别是底层通信协议）会变得很复杂，所以总共包含有60个左右的系统调用，本章将对所有这些进行讨论。

本书带有大量的示例，包括Web浏览器和服务器，这些足以让读者开始起步学习了，但要进一步研究UNIX网络通信，可能需要更高级的资料。目前最为完整的书是[Ste2003]，但还需要熟悉系统文档，特别是在所使用的协议与TCP/IP协议相比有很大差异时，就更需要熟悉这些协议的细节。

本章是这样展开的：首先讲解与套接字相关的基本系统调用，给出一些简单的客户/服务器例子。然后讲解套接字地址和套接字可选项。还要介绍一个隐藏了大量复杂细节的简单接口，并用它实现第7章深入讨论的简单消息接口（SMI）的套接字版本。接着是更深入的内容：无连接套接字、带外数据、网络数据库函数以及其他各种各样的函数。最后，将讨论一些关于在构建能同时处理数千台客户端的服务器时所遇到的关键问题。

8.1 套接字基础

本节介绍套接字的概念，并说明基本的套接字系统调用。

8.1.1 套接字的工作机制

套接字是非常复杂的，所以让我们从已经熟悉的内容开始。回顾7.2节的讨论可知，进程可以采用如下方式打开FIFO：

```
fd = open("MyFifo", O_RDONLY);
```

现在考虑这个系统调用内部实现了什么功能呢？

- 1) 创建一个I/O端点并为其分配文件描述符。
- 2) 将文件描述符绑定到外部名字“MyFifo”。
- 3) 等待，直到出现写进程。
- 4) 返回文件描述符，该文件描述符可以用于read系统调用。

套接字的工作流程与此类似，不同之处在于每一步都分解为一个独立的系统调用：

`socket`创建端点并分配文件描述符。

`bind`将该套接字与外部名字关联起来，让其他进程可以引用。

`listen`将该套接字标识为可以接收来自其他套接字的连接。

`accept`阻塞等待连接。

`connect`连接到在`accept`中阻塞的套接字。

FIFO在客户端与服务器端是对称的，两者都要执行完全相同的系统调用`open`，但带有不

同的标志（例如O_RDONLY与O_WRONLY）。有连接套接字通常是非对称的，因为客户端和服务器使用不同序列的系统调用，如图8-1所示。

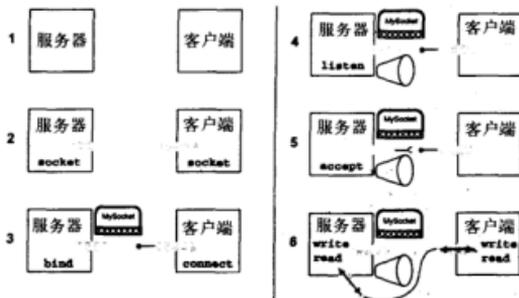


图8-1 建立有连接套接字

服务器端:

- 调用`socket`创建端点和文件描述符（步骤2）。
- 调用`bind`将套接字绑定到名字上（步骤3）。
- 调用`listen`将其标识为接收连接（步骤4）。
- 调用`accept`阻塞直至建立一条连接。然后`accept`使用新的文件描述符创建第二个套接字（步骤5）。
- 使用新的文件描述符在第二个套接字上进行读和写（步骤6）。

客户端:

- 调用`socket`创建端点和文件描述符（步骤2）。
- 使用服务器被绑定的名字作为参数调用`connect`，阻塞直至服务器接受连接（步骤3，尽管它不需要与服务器步骤3同步）。
- 使用套接字的文件描述符进行读写数据（步骤6）。

在正式引入套接字系统调用之前，先看一个示例程序。该程序创建一个子进程作为客户端，父进程作为服务器端:

```
#define SOCKETNAME "MySocket"

int main(void)
{
    struct sockaddr_un sa;

    (void)unlink(SOCKETNAME);
    strcpy(sa.sun_path, SOCKETNAME);
    sa.sun_family = AF_UNIX;
    if (fork() == 0) { /* child -- client */
        int fd_skt;
        char buf[100];

        ec_negl( fd_skt = socket(AF_UNIX, SOCK_STREAM, 0) )
        while (connect(fd_skt, (struct sockaddr *)&sa, sizeof(sa)) == -1)
            if (errno == ENOENT) {
                sleep(1);
                continue;
            }
        }
    else
        ;
}
```

```

        EC_FAIL
        ec_negl( write(fd_skt, "Hello!", 7 ) )
        ec_negl( read(fd_skt, buf, sizeof(buf)) )
        printf("Client got \"%s\"\n", buf);
        ec_negl( close(fd_skt) )
        exit(EXIT_SUCCESS);
    }
    else { /* parent -- server */
        int fd_skt, fd_client;
        char buf[100];

        ec_negl( fd_skt = socket(AF_UNIX, SOCK_STREAM, 0) )
        ec_negl( bind(fd_skt, (struct sockaddr *)&sa, sizeof(sa)) )
        ec_negl( listen(fd_skt, SOMAXCONN) )
        ec_negl( fd_client = accept(fd_skt, NULL, 0) )
        ec_negl( read(fd_client, buf, sizeof(buf)) )
        printf("Server got \"%s\"\n", buf);
        ec_negl( write(fd_client, "Goodbye!", 9) )
        ec_negl( close(fd_skt) )
        ec_negl( close(fd_client) )
        exit(EXIT_SUCCESS);
    }
}

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}

```

我们首先看到的是传递给bind和connect的名字不是简单的字符串，而是sockaddr的结构类型。sockaddr结构包含了sun_path成员中的字符串和sun_family成员中的地址族信息。套接字地址在其中是一个较大的主题，后面会详细阐述，这里暂且假定AF_UNIX的意思是“单机内部的本地通信”，这正是这个示例所需要的。

套接字文件名没有链接。因为对AF_UNIX来说，bind不能重用已经存在的名字，所以需要确保从新名字开始。

服务器端（父进程）遵循图8-1中的6个步骤序列。（暂时忽略传递给listen和accept的参数。）客户端（子进程）也遵循图8-1的步骤，但有一点儿小小的不同：如果客户端在服务器bind之前进行connect操作，connect会因为缺少套接字文件而失败。这种情况下，可以先睡眠然后再重试。

当服务器从accept返回时，它继续读写从accept得到的文件描述符（不是原始的套接字文件描述符）。当客户端从connect得到返回值时，它会读写它的套接字文件描述符（它唯一拥有的）。记住，accept和connect是两个阻塞调用——其他函数仅完成自己的工作并立即返回。

只要服务器和客户端需要，连接就一直保持着，其功能就像一根双向的管道（见6.6节）。套接字文件实际上是那样的文件类型，如ls命令所示：

```

$ ls -l MySocket
srwxr-xr-x 1 marc sysadmin 0 Apr 4 10:03 MySocket

```

下面是运行该示例程序的输出：

```

Server got "Hello!"
Client got "Goodbye!"

```

所以，套接字实际上非常简单，但需要注意以下细节：

- 同一服务器同时处理多台客户端。
- 网络通信的地址族，包括有意义的AF_INET族。
- 非流型通信，包括数据报而不包括数据流。
- 无连接通信，在这种通信中，进程向已经命名的接收者发送数据报，而不是像本例那样建立一条半永久性连接。
- 有许多用于调整套接字性能的选项，特别是针对AF_INET和AF_INET6的那些选项。
- 在机器间以不同的存储值方式来传送二进制数据。
- 访问名字数据库（比如www.basepath.com/aup）。

实际上的细节远不止这些。在建立网络通信程序中，它们是必须了解的内容，也是本章后续的内容。

8.1.2 有连接套接字的基本系统调用

套接字有两种：一种是有连接套接字，需要用accept和connect建立一条通道；另一种是无连接套接字，这种套接字中数据报发送给已命名接收者（不需要使用listen和accept，使用connect采用的也是不同的方式）。这一节所讨论的只限于有连接套接字，无连接套接字推后到8.6节讨论。

本节只探讨前面非正式提到的5个基本的专用套接字系统调用。首先看看socket：

socket——建立通信终点

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

int socket(
    int domain,           /* domain (AF_UNIX, AF_INET, etc.) */
    int type,            /* SOCK_STREAM, SOCK_DGRAM, etc. */
    int protocol         /* specific to type; usually zero */
);
/* Returns file descriptor or -1 on error (sets errno) */
```

socket函数所返回的文件描述符可以有两种不同的用途：

- 在服务器端，作为端点以accept来接收连接（实际的I/O是在文件描述符上完成的，而文件描述符是从accept函数返回的）。
- 在客户端，只要connect成功地连接上套接字，用于直接的I/O。

domain[⊖]和type都用于套接字地址。在调用bind和connect时会用到套接字地址。如果type提供了选择，那么protocol参数也会进行相应的选择。但在通常情况下采用默认值就可以了，所以通常为0。

下一步，服务器必须使用bind来命名套接字：

bind——把名字与套接字绑定

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

int bind(
    int socket_fd,       /* socket file descriptor */
    const struct sockaddr *sa, /* socket address */
    socklen_t sa_len    /* address length */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

⊖ 以AF_开始的宏有时写作PF_，但按标准应该采用AF_，本书也推荐采用AF_。

套接字地址需要用一整节(8.2节)讲解。我们已经见过了关于AF_UNIX的示例,它仅仅是一个单机内部的通信,类似于第7章所讲的IPC机制。后面将要说明如何设置其他域(包括AF_INET在内)的地址。

记住,在多数系统中,对于AF_UNIX,bind会创建一个新的套接字文件;它不重用已经存在的文件。

服务器也必须调用listen来建立用于接收连接的套接字:

listen——标识接收套接字并设置队列限制

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

int listen(
    int socket_fd,          /* socket file descriptor */
    int backlog            /* maximum connection queue length */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

上一节的示例中没有说明这些,服务器可以接受来自多个客户机的连接。另外,因为它只能以一定的速率发出accept,所以连接请求就必须在队列中等待。listen的第二个参数就是用来限制队列的长度的。通常,如果队列已满,客户端的连接(connect)请求会返回-1,并将errno设置为ECONNREFUSED。前面的示例中使用的是常量SOMAXCONN,这个常量是系统定义的最大值。

接着,服务器端接收连接请求:

accept——在套接字上接收新的连接并产生新的套接字

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

int accept(
    int socket_fd,          /* socket file descriptor */
    struct sockaddr *sa,    /* socket address or NULL */
    socklen_t *sa_len      /* address length */
);
/* Returns file descriptor or -1 on error (sets errno) */
```

一般情况下,accept会阻塞直至连接请求到达(来自其他进程的connect调用),然后在那条连接上创建一个用于I/O的新套接字,并返回新的文件描述符。该文件描述符可以用于常规的read和write系统调用,尽管对于更多的I/O控制可以使用专用套接字调用——send、sendto、sendmsg、recv、recvfrom、recvmsg,这些将会在8.6.2节、8.6.3节和8.9.1节中讲解。

如果为套接字文件描述符设置了O_NONBLOCK标志(用fcntl;见3.8.3节),并且队列中没有请求,那么accept就不等待连接,而是立即返回-1,并把errno设置成EAGAIN或EWOULDBLOCK。

套接字文件描述符可以用于select或poll,通常也是这样使用的,8.1.3节将会讲到这一点。典型的情况是,服务器使用文件描述符和由accept返回的每个文件描述符为select(或者对应的是poll)设置fd_set。如果select或poll表明套接字文件描述符已经准备好了,这就说明服务器应该accept(不会阻塞)。如果这些返回的文件描述符中的其中一个已经准备好了,这就意味着数据已经从客户端到达并可以读取了。

如果sa参数不是NULL,那么该参数就是用来返回所连接的套接字地址的。在输入时必须设置sa_len参数为sa所指向的存储空间的大小,返回时这个参数设置为实际的地址空间大小。

服务器的内容就讲到这里。客户端，在创建其套接字后，只调用connect，所使用的套接字地址与服务器中bind所使用的套接字地址相同：

```
connect —— 连接套接字

#include <sys/socket.h>

int connect(
    int socket_fd,           /* socket file descriptor */
    const struct sockaddr *sa, /* socket address */
    socklen_t sa_len        /* address length */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

像accept一样，connect通常阻塞直至接受连接请求，不同的是connect不返回文件描述符——客户端在套接字文件描述符上进行I/O。

如果设置了O_NONBLOCK标志，那么connect不会阻塞连接等待，但会返回-1，同时设置errno为EINPROGRESS。不抛弃连接请求，而是保留在队列中等待最终处理。在此期间如果继续调用connect也会返回-1，但此时将errno设置成了EALREADY。当连接成功时，套接字文件描述符就可以使用了。如果需要，可以使用select或poll等待它的可写（非读）状态。一个典型的用途就是应用程序需要初始化工作。它发布一个非阻塞的connect进行初始化，然后发布select或poll进行阻塞。

8.1.3 处理多个客户端

现在扩展8.1.1节中的示例，以使服务器能够处理多个客户端。下面的示例是对服务器端代码的修改：

```
static bool run_server(struct sockaddr_un *sap)
{
    int fd_skt, fd_client, fd_hwm = 0, fd;
    char buf[100];
    fd_set set, read_set;
    ssize_t nread;

    ec_negl( fd_skt = socket(AF_UNIX, SOCK_STREAM, 0) )
    ec_negl( bind(fd_skt, (struct sockaddr *)sap, sizeof(*sap)) )
    ec_negl( listen(fd_skt, SOMAXCONN) )
    if (fd_skt > fd_hwm)
        fd_hwm = fd_skt;
    FD_ZERO(&set);
    FD_SET(fd_skt, &set);
    while (true) {
        read_set = set;
        ec_negl( select(fd_hwm + 1, &read_set, NULL, NULL, NULL) )
        for (fd = 0; fd <= fd_hwm; fd++)
            if (FD_ISSET(fd, &read_set)) {
                if (fd == fd_skt) {
                    ec_negl( fd_client = accept(fd_skt, NULL, 0) )
                    FD_SET(fd_client, &set);
                    if (fd_client > fd_hwm)
                        fd_hwm = fd_client;
                }
                else {
                    ec_negl( nread = read(fd, buf, sizeof(buf)) )
                    if (nread == 0) {
```

```

        FD_CLR(fd, &set);
        if (fd == fd_hwm)
            fd_hwm--;
        ec_negl( close(fd) )
    }
    else {
        printf("Server got '%s'\n", buf);
        ec_negl( write(fd, "Goodbye!", 9) )
    }
}
}
}
ec_negl( close(fd_skt) )
return true;

EC_CLEANUP_BGN
return false;
EC_CLEANUP_END
}

```

新加的代码多数是为创建和使用select而准备的。这在4.2.3节已经讨论过，这里简要重述：

select —— 等待I/O准备好

```

#include <sys/select.h>

int select(
    int nfds,                /* highest fd + 1 */
    fd_set *readset,        /* read set or NULL */
    fd_set *writeset,       /* write set or NULL */
    fd_set *errorset,       /* error set or NULL */
    struct timeval *timeout /* time-out (microseconds) or NULL */
);
/* Returns number of bits set or -1 on error (sets errno) */

```

在run_server函数中有两个fd_set：一个包含了所有相关的文件描述符：套接字文件描述符和每个客户端accept返回的文件描述符。我们需要select集合中文件描述符的数目，以便使变量fd_hwm（“高水位标志”）保持到从accept得到新的文件描述符的时间。当我们关闭最大的文件描述符时，从set中删除它并将高水位标志减1。从set中删除它是极其重要的，否则，select会一直报告它处于准备好的状态，这意味着不是数据准备好可以读了，而是说读不会阻塞。

第二个fd_set是read_set，是每次调用select时从set中复制的，然后用select去修改来表明哪个文件描述符处于准备好状态。[⊖]

然后此函数就一直循环下去，直至给出一个销毁信号。每当它从select得到返回值时，它都会遍历整个返回的集合（修改后的read_set变量）以寻找准备好的文件描述符。如果fd_skt准备好了，它就调用accept；如果其他的也准备好了，这说明客户端已经发送了一些数据。可以读取、显示并返回一个短的消息。

客户端的代码同先前的示例类似，不同之处在于返回给服务器的消息中包含有进程ID：

```

static bool run_client(struct sockaddr_un *sap)
{

```

[⊖] 一个常见的错误是只让一个得到的fd_set传递给select，因为它为所有未准备好的文件描述符清除了位，所以它们决不会等待。

```

if (fork() == 0) {
    int fd_skt;
    char buf[100];

    ec_negl( fd_skt = socket(AF_UNIX, SOCK_STREAM, 0) )
    while (connect(fd_skt, (struct sockaddr *)sap, sizeof(*sap)) == -1)
        if (errno == EWOULDBLOCK) {
            sleep(1);
            continue;
        }
        else
            EC_FAIL
    snprintf(buf, sizeof(buf), "Hello from %ld!",
        (long)getpid());
    ec_negl( write(fd_skt, buf, strlen(buf) + 1) )
    ec_negl( read(fd_skt, buf, sizeof(buf)) )
    printf("Client got \"%s\"\n", buf);
    ec_negl( close(fd_skt) )
    exit(EXIT_SUCCESS);
}
return true;

EC_CLEANUP_BGN
/* only child gets here */
exit(EXIT_FAILURE);
EC_CLEANUP_END
}

```

最后，有个小的main函数建立套接字地址，生成4个客户端子进程，同时父进程作为服务器进程：

```

#define SOCKETNAME "MySocket"

int main(void)
{
    struct sockaddr_un sa;
    int nclient;

    (void)unlink(SOCKETNAME);
    strcpy(sa.sun_path, SOCKETNAME);
    sa.sun_family = AF_UNIX;
    for (nclient = 1; nclient <= 4; nclient++)
        ec_false( run_client(&sa) )
    ec_false( run_server(&sa) )
    exit(EXIT_SUCCESS);

    EC_CLEANUP_BGN
        exit(EXIT_FAILURE);
    EC_CLEANUP_END
}

```

其输出如下：

```

Server got "Hello from 31786!"
Client got "Goodbye!"
Server got "Hello from 31785!"
Client got "Goodbye!"
Server got "Hello from 31784!"
Client got "Goodbye!"
Server got "Hello from 31787!"

```



```
Client got "Goodbye!"
```

到此为止，我们已经有了一定的能力，完全可以实现一个相当复杂的客户/服务器系统。我们还需要进一步知道如何设置套接字地址（AF_UNIX不是那么让人感兴趣）和如何设置套接字选项。在讲解完字节顺序之后，就会介绍这些内容。

8.1.4 字节顺序

在同一网络中的机器之间传送诸如“Hello”和“Goodbye”之类的字符串绝对不会有什问题，因为所有通信都会保持字节顺序。但是，如果传送二进制数就会出现问，因为数字中的字节顺序的排列在不同机器之间是不同的。

为说明这个问题，请看图8-2，该图显示了在地址为106204的内存空间中存储两个字节的整数所采用的两种不同的方式，其十六进制值为0xD04C。

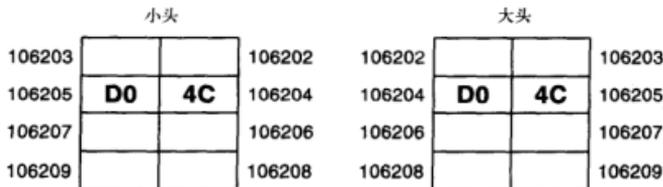


图8-2 小头与大头

在称为“小头”（little-endian）的机器中，数字的地址是其最低字节的地址；在“大头”（big-endian）机器中，数字的地址是其最高字节^①的地址。它们都是表示地址数字的有效方法。问题在于，如果一台机器向另一台机器发送二进制数字，且两台机器采用的字节顺序不同，那么数字就会出现混乱。换句话说，如果小头机器以字节（发送数据的唯一方式）发送数字，那么这些字节发送的顺序是4C紧跟在D0之后；如果接收机器是大头的，那么它会将这个数字解释为4CD0，因为它将第一个接收到的字节作为高位字节。

解决方案是采用达成一致的网络字节顺序来发送二进制数字。这样，所有的发送方必须将本机（本地）字节数据转换为网络数据，所有的接收方必须将网络数据转换为本机字节顺序。但是需要这样做的前提是数据不是采用标准的方式组织的。例如，如果有一张采用JPEG编码的照片，就可以直接发送它——无需将JPEG转换为网络字节顺序。

对于大多数情况，无需知道本机字节顺序和网络字节顺序，因为已经有一组标准的转换函数专门用来进行16位和32位整数的转换工作：

htons——把16位值从主机字节序转换成网络字节序

```
#include <arpa/inet.h>

uint16_t htons(
    uint16_t hostnum          /* 16-bit number in host byte order */
);
/* Returns number in network byte order (no error return) */
```

① 这个术语是自解释的，但术语“大头”实际上源自Jonathan Swift的《格列佛游记》，其中描述了一个关于鸡蛋是在大头还是小头打破的争论。实际上，“一万一千个[大头]数次都是完全破碎，而小头只是打破鸡蛋。”

htonl——把32位值从主机字节序转换成网络字节序

```
#include <arpa/inet.h>

uint32_t htonl(
    uint32_t hostnum      /* 32-bit number in host byte order */
);
/* Returns number in network byte order (no error return) */
```

ntohs——把16位值从网络字节序转换成主机字节序

```
#include <arpa/inet.h>

uint16_t ntohs(
    uint16_t netnum      /* 16-bit number in network byte order */
);
/* Returns number in host byte order (no error return) */
```

ntohl——把32位值从网络字节序转换成主机字节序

```
#include <arpa/inet.h>

uint32_t ntohl(
    uint32_t netnum      /* 32-bit number in network byte order */
);
/* Returns number in host byte order (no error return) */
```

不幸的是，16位的函数后面都带有表示short的后缀“s”，尽管short可能不止16位；同样的情况是，32位函数后面都带有表示long的后缀“l”。而实际上，这些函数并不是进行short类型和long类型的运算；它们是uint16_t（16位无符号整数）和uint32_t（32位无符号整数）的运算。

为了说明这两个函数是如何使用的，下面给出一个示例程序。该程序将0xD04C转换为网络字节顺序，并将这些字节显示出来，然后重新转换回去：

```
int main(void)
{
    uint16_t nhost = 0xD04C, nnetwork;
    unsigned char *p;

    p = (unsigned char *)&nhost;
    printf("%x %x\n", *p, *(p + 1));
    nnetwork = htons(nhost);
    p = (unsigned char *)&nnetwork;
    printf("%x %x\n", *p, *(p + 1));
    exit(EXIT_SUCCESS);
}
```

输出表明，在Intel Pentium CPU上运行时采用的字节顺序与网络字节顺序是不同的：

```
4c d0
d0 4c
```

我们由此可以进一步推断出，Pentium是小头的，而网络字节顺序是大头的。[⊖]

总的来讲，当所使用的协议需要发送二进制数时，就需要使用标准的转换函数，下一节我们将讲到这部分内容。对于我们自身来说，设计程序时应尽量发送字符，避免发送二进制

⊖ 之所以是大头的，是因为在多年前的BSD系统中，当时运行在大头的VAX 11/780型计算机上，而套接字系统调用是从BSD起源的。因此网络字节顺序从根本上，是“谁老大谁说了算”。

数据，除非要发送的是标准格式（JPEG、MP3等等）的对象。实际上HTTP（Web协议）就是这样处理的（见8.4.2节）。

8.2 套接字地址

本节讲述的内容都是在调用bind和connect时如何使用套接字地址（struct sockaddr）的问题。前面已经讲过AF_UNIX域是如何处理的，这种情况下地址通常只是某个路径名字：

```
struct sockaddr_un sa;

strcpy(sa.sun_path, SOCKETNAME);
sa.sun_family = AF_UNIX;
```

但这只是很简单的情况。其他的域要复杂一些。

8.2.1 套接字地址结构

每种地址族都有其自己的结构类型和头文件。例如，sockaddr_un是针对AF_UNIX的结构类型，sockaddr_in是针对AF_INET的结构类型，sockaddr_x25是针对AF_X25的结构类型，等等，尽管在[SUS2002]中仅标准化了前两个域。一种方法（这种方法已经用在AF_UNIX上）就是声明一个所需要的专用的结构类型变量，对该变量的成员赋值，然后在调用bind或connect时将该变量的地址强制转换为指向struct sockaddr（通用类型）的指针。

简而言之，结构sockaddr只是定义了一种抽象类型；不能使用它来声明要使用的结构，因为它可能没有足够的空间。相反，sockaddr_storage是一个标准的结构类型，该结构类型有足够的空间，但它的成员不是为特定的域专门定制的。

这些内容好像让人感到迷惑不解，但实际上没有必要这样担心：

- 如果知道所使用的到底是什么域，那么可以直接声明对应类型（比如sockaddr_un）的结构变量，或者为这种结构动态分配空间（比如使用malloc）。
- 如果任何的域结构类型都需要足够的空间，就使用sockaddr_storage类型，但在使用它之前需要将指针强制转换为相应的类型。
- 在调用bind和connect时强制转换为struct sockaddr——根据它们的原型，别无它选。

下面的示例同时使用了上面的后两条规则：

```
struct sockaddr_storage sas;
struct sockaddr_un *sa = (struct sockaddr_un *)&sas;
sa->sun_family = AF_UNIX;
...
ec_negl( bind(fd, (struct sockaddr *)sa, sizeof(*sa)) )
```

不用过于担心最后一条规则。如果出错，编译器会显示。也不用太担心直接使用sockaddr_storage，它没有任何你所需要的成员。但是，由于使用了强制类型转换，所以在编译时根本就不检测是否使用了正确的域结构类型（所使用的域专有的结构类型）。所以就必须保证第一次就是正确的。

8.2.2 AF_UNIX套接字地址

struct sockaddr_un —— AF_UNIX套接字地址的结构

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

struct sockaddr_un {
    sa_family_t sun_family; /* AF_UNIX */
    char sun_path[X];      /* socket pathname */
};
```

对照表中的X分配的路径名所需要的实际空间在各个系统之间有所不同。不要假定它超过90个字节左右的空间。前面已经解释了如何使用该结构。

8.2.3 AF_INET套接字地址

如前面所提及的，AF_INET域用于在因特网上通过套接字进行通信。下面是sockaddr_in结构：

struct sockaddr_in —— AF_INET套接字地址的结构

```
#include <netinet/in.h>

struct sockaddr_in {
    sa_family_t sin_family; /* AF_INET */
    in_port_t sin_port;     /* port number (uint16_t) */
    struct in_addr sin_addr; /* IPv4 address */
};

struct in_addr {
    in_addr_t s_addr;      /* IPv4 address (uint32_t) */
};
```

IPv4[Ⓔ]地址（通常简称为“IP地址”）是分配给机器的网络接口的一个32位二进制数。这些数字不采用通常的方式（比如1, 182, 625, 240）引用，而是使用点分法。在点分法中，每个字节表示为一个单独的十进制数字，如216.109.125.70。即使如此，用户有可能仍然觉得不够方便，所以就有了采用描述性名字的命名系统，这样我们可以不需要使用216.109.125.70（见8.2.5节），而是使用www.yahoo.com。不过这里暂时还是采用点分表示法。

每个IP可以与许多不同的服务相关联，每个服务都被分配了一个16位的端口号。为方便起见，在这些端口号中，其中不少端口号已经分配给了常见的服务，例如，HTTP（Web）在80端口提供服务，FTP在21端口提供服务，Telnet在23端口提供服务，等等。要查看机器的端口分配情况，可以查看/etc/services文件。例如，下面是从我的FreeBSD系统上的这个文件（总共超过2000行）中抽取的一部分：

```
...
ftp          21/tcp      #File Transfer [Control]
ftp          21/udp      #File Transfer [Control]
ssh          22/tcp      #Secure Shell Login
ssh          22/udp      #Secure Shell Login
telnet       23/tcp
telnet       23/udp
...
```

[Ⓔ] 就是网际协议版本4（Version 4 of the Internet Protocol）。

```

finger      79/tcp
finger      79/udp
http        80/tcp      www www-http      #World Wide Web HTTP
http        80/udp      www www-http      #World Wide Web HTTP
...

```

端口和IP地址的类型 `in_port_t` 和 `in_addr_t` 分别被定义为 `uint16_t` 和 `uint32_t`，而且必须采用网络字节顺序，这意味着可以使用 `htons` 函数和 `htonl` 函数，例如：

```

struct sockaddr_in sa;

sa.sin_family = AF_INET;
sa.sin_port = htons(80);
sa.sin_addr.s_addr = htonl((216UL << 24) + (109UL << 16) +
(125UL << 8) + 70UL); /* 216.109.125.70 */

```

但是对于点分表示法的IP地址，有个更好的函数能按照网络字节顺序直接将字符串转换为32位IP地址：

inet_addr——把IPv4点串地址转换成整数

```

#include <arpa/inet.h>

in_addr_t inet_addr(
    const char *cp          /* dotted IP address */
);
/* Returns IP address or (in_addr_t)-1 on error (errno not defined) */

```

返回值-1强制转换为32位无符号整数，和255.255.255.255转换后的结果相同。但这不会有问题，因为这是一个非法的IP地址。

反向转换函数有时候用起来也挺方便：

inet_ntoa——把IPv4整数地址转换成点串

```

#include <arpa/inet.h>

char *inet_ntoa(
    struct in_addr in      /* integer address */
);
/* Returns string (no error return) */

```

除了 `inet_addr` 和 `inet_ntoa` 之外，还可以使用更通用的函数 `inet_ntop` 和 `inet_pton`（见8.9.5节），它们也适用于IPv6地址。

回到刚才的示例，这次使用 `inet_addr` 代替 `htonl` 进行IP地址转换，同时加入在端口80连接HTTP服务器的代码，向Web页面发送请求并显示部分返回信息：

```

#define REQUEST "GET / HTTP/1.0\r\n\r\n"

int main(void)
{
    struct sockaddr_in sa;
    int fd_skt;
    char buf[1000];
    ssize_t nread;

    sa.sin_family = AF_INET;
    sa.sin_port = htons(80);
    sa.sin_addr.s_addr = inet_addr("216.109.125.70");
    ec_negl( fd_skt = socket(AF_INET, SOCK_STREAM, 0) )
    ec_negl( connect(fd_skt, (struct sockaddr *)&sa, sizeof(sa)) )
    ec_negl( write(fd_skt, REQUEST, strlen(REQUEST)) )
}

```

```

ec_negl( nread = read(fd_skt, buf, sizeof(buf)) )
(void)write(STDOUT_FILENO, buf, nread);
ec_negl( close(fd_skt) )
exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}

```

请求是GET命令，这个命令在定义Web通信（见8.4.2节）的HTTP协议中进行了详细说明。下面是部分返回信息，可以看出这是yahoo主页的HTML源码。它以状态行开始，紧跟着的是HTML代码（所示的示例已经进行了很大的简化）：

```

HTTP/1.1 200 OK
Date: Sat, 19 Jul 2003 18:51:56 GMT
P3P: policyref="http://p3p.yahoo.com/w3c/p3p.xml", CP="CAO DSP COR CUR ADM ..."
Cache-Control: private
Connection: close
Content-Type: text/html

<html><head>
<title>Yahoo!</title>
...

```

8.2.4 AF_INET6套接字地址

如果IPv4地址的所有32位都能得到使用，那么总共有43亿个网络地址，也许暂时是够用了。但是因为IPv4地址的分配方式的原因，大型组织所用的B类地址（从192开始到223结束）只有53 400万个，在1992年左右的时候，人们就预计到1995年年中就会用完这些地址。另外一个问题是，因特网骨干节点上的路由表太大，以至于超出了当时路由器的有效内存。

提出的解决方案就是网际协议版本6（Version 6 of the Internet Protocol），简称IPv6，IPv6对IPv4进行了诸多方面的改进，其中一项就是将IP地址从4字节扩展到16字节，可容纳 2^{128} 个地址。[⊖]

IPv6地址写法的首选方案是每组2字节共分8组，例如FEDC:BA98:7654: 3210:FEDC:BA98:7654:3210。

设置AF_INET6套接字地址需要使用sockaddr_in6结构；可以预料，与用于AF_INET的结构类型相比，成员更多：

struct sockaddr_in6 —— AF_INET6套接字地址的结构

```

#include <netinet/in.h>

struct sockaddr_in6 {
    sa_family_t sin6_family;    /* AF_INET6 */
    in_port_t sin6_port;       /* port number (uint16_t) */
    uint32_t sin6_flowinfo;    /* traffic class and flow information */
    struct in6_addr sin6_addr;  /* IPv4 address */
    uint32_t sin6_scope_id;    /* set of interfaces for a scope */
};

struct in6_addr {
    uint8_t s6_addr[16];       /* IPv6 address */
};

```

⊖ 这个数字超过了宇宙中所有粒子的数目，即地球表面每平方方米有超过6 550万亿亿个地址！

这里的两个新成员在`sockaddr_in`结构类型中没有对应项，一个是`sin6_flowinfo`，包含一个“流标签”和优先权值，但它的用途目前还没有指定；另一个是`sin6_scope_id`，它指定了一组为某些地址使用的接口，并且它是依赖于实现的。如果设置自己的`sockaddr_in6`结构，就将它们设置为0。事实上，建议在设置所关心的成员值之前，先将该结构整个设置为0，因为有些实现除了以上对照表中的这些成员之外，还有其他成员。

实际应用中，不需要直接初始化`sockaddr_in6`结构。你可以通过`getaddrinfo`得到该结构，这部分内容将在8.2.6节中介绍。

`inet_addr`和`inet_ntoa`结构不能用于IPv6地址。如果是IPv6地址，需要使用`inet_ntop`和`inet_pton`，这两个结构的相关介绍在8.9.5节中。

8.2.5 域名系统

在使用Internet浏览器或者FTP客户端时，我们很少使用数字方式表示的IP地址——实际输入通常是容易记忆的名字，像`www.yahoo.com`或`www.basepath.com`。这些主机名字是通过一个称为DNS的世界范围的分布式数据库转换为IP地址的。UNIX应用程序访问DNS有几个标准函数，其中最新最有效的一个函数是`getaddrinfo`，在后面的例子中会用到这个函数。8.8.1节简要地描述了几个较早的函数（如`gethostbyname`）。

在像`http://www.basepath.com/aup`（本书站点的URL）这样的URL中，主机名仅是中间部分。确切地说，总的语法是：[⊖]

```
scheme://hostname/path
```

其中，`scheme`指定了访问资源的方法（比如HTTP、FTP），`hostname`是主机名字，该名字记录在DNS数据库中，`path`是主机文件系统内的一条路径。

在后面的例子中，使用套接字系统调用来连接主机。一旦有了连接，就根据`scheme`与主机交互。例如，在8.2.3节的示例中，`scheme`是HTTP，所以发送给服务器的请求是：

```
GET / HTTP/1.0
```

这个请求解释为请求一个返回的Web页面。路径的处理取决于`scheme`；对于HTTP而言，路径是GET命令的一个参数（上面的例子是单个的斜线）。所有的套接字系统都不关心路径和访问方法。

除了需要了解那些在简单示例中所需的一些最基本的知识外，在本书中没有必要深入了解HTTP、HTML、FTP或其他与访问方法相关的内容。了解不同系统最好的方法是，读请求注解（Request for Comment, RFC）文档[RFC]。

8.2.6 getaddrinfo

较容易的方法是调用创建套接字的`getaddrinfo`，而不是从零开始构造`sockaddr_in`或`sockaddr_in6`地址：

getaddrinfo——得到套接字地址信息

```
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo(
    const char *nodename,      /* node name */
    const char *servname,     /* service name */
    const struct addrinfo *hint, /* hint */
    struct addrinfo **info)   /* returned info as linked list */
);
/* Returns 0 on success or error number on error (errno not set) */
```

⊖ 这是一种简化的语法，但就我们的目的而言已足够。URL的完整语法更复杂。

struct addrinfo —— getaddrinfo的结构

```

struct addrinfo {
    int ai_flags;                /* input flags */
    int ai_family;              /* address family */
    int ai_socktype;            /* socket type */
    int ai_protocol;           /* protocol */
    socklen_t ai_addrlen;      /* length of socket address */
    struct sockaddr *ai_addr;   /* socket address */
    char *ai_canonname;        /* canonical name of service location */
    struct addrinfo *ai_next;   /* pointer to next structure in list */
};

```

典型地，调用getaddrinfo时，可以用nodename作为参数来设置需要的主机名，用servname作为参数来设置端口号（以字符串表示），用hint作为参数来设置需要的地址族和套接字类型。通过infop参数可以得到返回值，该参数是addrinfo结构类型的链表，其中包含了套接字信息和其他匹配提示的信息。选择一个适当的套接字地址，直接在connect和bind调用中使用，将ai_addr成员强制转换为struct sockaddr*。

“主机名”可能是一个在DNS服务器上能够查找的名字，或者是一个在本地机器的/etc/hosts文件中定义的名字，或者是用点分标记法表示的IPv4地址（字符串型），也可能是一个冒号标记法表示的IPv6地址（字符串型）。

一个关键的标志是AI_PASSIVE，意思是返回的套接字地址是为了accept使用；也就是说，是服务器进行的getaddrinfo调用。否则，如果清除标识，就说明该调用来自于客户端，那么套接字地址将被connect使用。如果是无连接协议（比如SOCK_DGRAM），它还可以被sendto或sendmsg使用。

getaddrinfo返回的是其特有类型的错误代码，不能将其当作errno值，这就说明在出错时，不能使用标准的错误处理函数（比如perror或strerror）。相反，有一个专门用于处理由getaddrinfo和另一个函数getnameinfo返回的错误代码的函数（见8.8.1节）。

gai_strerror —— 得到错误代码描述

```

#include <netdb.h>

const char *gai_strerror(
    int code                /* error code */
);
/* Returns string (no error return) */

```

下面的例子中使用了错误检查宏ec_ai，该宏知道如何处理从getaddrinfo和getnameinfo返回的值，所以不直接调用gai_strerror。

下面是一个使用getaddrinfo的简单示例：

```

int main(void)
{
    struct addrinfo *infop = NULL, hint;

    memset(&hint, 0, sizeof(hint));
    hint.ai_family = AF_INET;
    hint.ai_socktype = SOCK_STREAM;
    ec_ai( getaddrinfo("www.yahoo.com", "80", &hint, &infop) )
    for ( ; infop != NULL; infop = infop->ai_next ) {
        struct sockaddr_in *sa = (struct sockaddr_in *)infop->ai_addr;

        printf("%s port: %d protocol: %d\n", inet_ntoa(sa->sin_addr),

```

```

        ntohs(sa->sin_port), infop->ai_protocol);
    }
    exit(EXIT_SUCCESS);
EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}

```

输出如下:

```

66.218.70.48 port: 80 protocol: 0
66.218.70.49 port: 80 protocol: 0
66.218.71.88 port: 80 protocol: 0
66.218.71.86 port: 80 protocol: 0
66.218.70.50 port: 80 protocol: 0
66.218.71.91 port: 80 protocol: 0
66.218.71.80 port: 80 protocol: 0
66.218.71.84 port: 80 protocol: 0
66.218.71.90 port: 80 protocol: 0
66.218.71.93 port: 80 protocol: 0
66.218.71.94 port: 80 protocol: 0
66.218.71.92 port: 80 protocol: 0
66.218.71.89 port: 80 protocol: 0

```

所以看起来, www.yahoo.com与多个不同的IP地址相关联。根据提示可以知道, 它们都是SOCK_STREAM类型的AF_INET地址, 所以都适用于Web页面访问。为说明这一点, 我们将通过getaddrinfo来对套接字地址检索与连接进行匹配, 并给出8.2.3节的示例代码:

```

#define REQUEST "GET / HTTP/1.0\r\n\r\n"
int main(void)
{
    struct addrinfo *infop = NULL, hint;
    int fd_skt;
    char buf[1000];
    ssize_t nread;

    memset(&hint, 0, sizeof(hint));
    hint.ai_family = AF_INET;
    hint.ai_socktype = SOCK_STREAM;
    ec_ai( getaddrinfo("www.yahoo.com", "80", &hint, &infop) )
    ec_negl( fd_skt = socket(infop->ai_family, infop->ai_socktype,
        infop->ai_protocol) )
    ec_negl( connect(fd_skt, (struct sockaddr *)infop->ai_addr,
        infop->ai_addrlen) )
    ec_negl( write(fd_skt, REQUEST, strlen(REQUEST)) )
    ec_negl( nread = read(fd_skt, buf, sizeof(buf)) )
    (void)write(STDOUT_FILENO, buf, nread);
    ec_negl( close(fd_skt) )
    exit(EXIT_SUCCESS);
EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}

```

运行上面这段程序会得到和8.2.3节中一样的结果。

如果所使用的系统、DNS服务器和所访问的主机支持, 那么使用getaddrinfo在80端口访问的地址就不止是SOCK_STREAM AF_INET了。可以使用NULL来指定nodename或servname参数(但不能同时使用NULL), 也可以(或者)使用提示域的AF_UNSPEC地址

“族”来找出可利用的信息，然后从所返回的多个地址中找出想使用的地址。例如，如果能找到的话，可能想使用AF_INET6 (IPv6)地址。关于getaddrinfo高级应用的更详细的信息，可以参见[SUS2002]、所使用的系统文档或者[Ste2003]的第11章。

在本节结束之前，需要提醒的是：当不再需要从getaddrinfo返回的链表时，应当用freeaddrinf调用来释放它：

freeaddrinfo —— 释放套接字地址信息

```
#include <sys/socket.h>
#include <netdb.h>

void freeaddrinfo(
    struct addrinfo *info /* list to free */
);
```

在我们的示例中没有使用freeaddrinfo，因为这些示例都是main函数，它们总是会退出的。

8.2.7 gethostname

有时，比如在8.4.4节中的Web服务器示例程序中，程序需要知道它自己的主机名，通常使用通用的“localhost”就可以了。但如果是其他机器需要连接到该主机名，需要使用的就是系统管理员分配的公开的主机名。这正是gethostname的功能：

gethostname —— 得到主机名

```
#include <unistd.h>

int gethostname(
    char *name,          /* returned name */
    size_t namelen      /* size of name buffer */
);
/* Returns 0 on success or -1 on error (errno not defined) */
```

8.4.4节中有使用gethostname的示例。

8.3 套接字选项

虽然套接字很复杂，而socket系统调用却很简单，其中的一个原因就是所有的选项设置都是通过一个单独的系统调用处理的：

setsockopt —— 设置套接字选项

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

int setsockopt(
    int socket_fd,      /* socket file descriptor */
    int level,         /* level to be accessed */
    int option,        /* option to set */
    const void *value, /* value to set */
    socklen_t value_len /* length of value */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

以上5个参数中，有4个是很容易解释的：socket_fd是套接字文件描述符，option是选项名字，value指向要设置值的指针，value_len是value所指的值的长度。值通常为整

型，但有时对于特定的选项可能是结构类型，后面会看到这种情况。

第二个参数level指明了选项属于哪种协议级别。SOL_SOCKET级别适用于套接字级别本身，它可能是使用最多的级别。[SUS2002]在<netinet/in.h>中定义了6种其他的协议级别：

```
IPPROTO_IP      网际协议 (Internet Protocol)
IPPROTO_IPV6    网际协议版本6 (Internet Protocol Version 6)
IPPROTO_ICMP    控制消息协议 (Control message protocol)
IPPROTO_RAW     原始IP包协议 (Raw IP Packets Protocol)
IPPROTO_TCP     传输控制协议 (Transmission control protocol)
IPPROTO_UDP     用户数据报协议 (User datagram protocol)
```

[SUS2002]为这6种协议定义了一些选项，但实现时通常又会定义其他的选项。不幸的是，很难得到所有这些选项的列表，所以必须通读所使用的协议的文档。例如，在Solaris上，键入man ip，就可以得到描述IPPROTO_IP选项的用户手册。

这里我们只描述标准的SOL_SOCKET选项；其他级别的选项，请查阅所使用的系统文档或者[Ste2003]。

有一个与得到选项相匹配的调用，参数几乎完全相同：

```
getsockopt——得到套接字选项

#include <sys/socket.h>

int getsockopt(
    int socket_fd,          /* socket file descriptor */
    int level,              /* level to be accessed */
    int option,             /* option to get */
    void *value,           /* returned value */
    socklen_t *value_len  /* length of value */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

setsockopt和getsockopt之间的一个区别是：getsockopt的value_len参数是指向长度的指针，在调用之前，必须将它指定为value参数所指向的缓冲区的大小。返回时，它被设置成返回值的长度。

下面是设置套接字SO_REUSEADDR选项的简单示例（后面会解释其实际含义）：

```
int socket_option_value = 1;

ec_negl( setsockopt(fd, SOL_SOCKET, SO_REUSEADDR,
    &socket_option_value, sizeof(socket_option_value)) )
```

这里简要介绍一下SOL_SOCKET所有的可选项。大多数选项值是整型的，其中value必须指向int，value_len为sizeof(int)。有些整数是布尔值，1为真0为假。不要使用C语言常量true来代替1，因为true的值可能为-1，这种情况下-1对于C可能是正确的，但对setsockopt来说可能是不可接受的。在下面的清单中，标记(B)指的是布尔值，(I)指的是整型值，(X)表示一个需要进一步进行解释的类型。字母后面跟的“s”和“g”是指选项可以用于setsockopt、getsockopt或者可以同时用于两者。

注意，一些选项是否真的做了什么（比如SO_KEEPAIVE）取决于实际的协议和该协议的实现。

SO_ACCEPTCONN 套接字正在接受连接；也就是说，已经调用了listen。(Bg)

SO_BROADCAST 如果协议支持广播报文，则允许发送。(Bsg)

SO_DEBUG 调试信息由底层协议实现记录。(Bsg)

SO_DONTROUTE 通过标准的路由设施来发送报文，根据目的地址直接到达网络接口。(Bsg)

SO_ERROR 套接字错误状态，在取得错误值后清除该状态。(Ig)

SO_KEEPAALIVE 通过周期性地发送报文使连接处于活动状态。如果没有任何应答，就断开套接字。(对断开的套接字执行写操作，就如同写一个没有读者的管道，会产生SIGPIPE信号。)(Bsg)

SO_LINGER 设置了该选项时，如果有未发送完的报文，那么会导致close调用阻塞进程，直到所剩数据发送完毕或超时。(Xsg) 值是一个linger结构：

```
struct linger {
    int l_onoff; /* 开(1)或关(0) */
    int l_linger; /* 以秒计算的延迟时间 */
};
```

SO_OOINLINE 接收到的带外数据为内嵌式(见8.7节)。(Bsg)

SO_RCVBUF 接收缓冲区的大小。(Isg)

SO_RCVLOWAT 接收低潮标志。阻塞接收操作(例如read)处于阻塞状态，除非接收的数量少于该数和所要求的数。默认为1。(Isg)

SO_RCVTIMEO 使用时间val结构(见1.7.1节)表示等待阻塞接收操作完成的最大时间。0时间(默认)表示无限。如果时间用完，该操作返回一个部分的计数，或者在返回-1的同时将errno设置为EAGAIN或EWOULDBLOCK。(Xsg)

SO_REUSEADDR bind允许复用本地地址。否则，如果前一个bind在系统规定时间(比如几分钟)内已经发生过，则bind返回-1的同时会将errno设置为EADDRINUSE。调试和测试非常方便。(Bsg)

SO_SNDBUF 发送缓冲区的大小。该选项取整数值。(Isg)

SO_SNDLOWAT 发送低潮标志。非阻塞性发送操作(比如write)不发送任何数据，除非能够立即发送的量少于该数和所要求的数。(Isg)

SO_SNDTIMEO 使用时间val结构(见1.7.1节)表示等待阻塞发送操作完毕的最大时间。0时间(默认)表示无限。如果时间用完，该操作返回一个部分计数，或者返回-1的同时设置errno为EAGAIN或EWOULDBLOCK。(Xsg)

SO_TYPE 套接字类型(比如SOCK_STREAM)。(Isg)

为了说明这些选项是如何与getsockopt一起使用的，下面这个程序显示了几种不同类型的套接字的值：

```
typedef enum {OT_INT, OT_LINGER, OT_TIMEVAL} OPT_TYPE;

static void show(int skt, int level, int option, const char *name,
    OPT_TYPE type)
{
    socklen_t len;
    int n;
    struct linger lng;
    struct timeval tv;

    switch (type) {
        case OT_INT:
            len = sizeof(n);
```

```

    if (getsockopt(skt, level, option, &n, &len) == -1)
        printf("%s FAILED (%s)\n", name, strerror(errno));
    else
        printf("%s = %d\n", name, n);
    break;
case OT_LINGER:
    len = sizeof(lng);
    if (getsockopt(skt, level, option, &lng, &len) == -1)
        printf("%s FAILED (%s)\n", name, strerror(errno));
    else
        printf("%s = l_onoff: %d; l_linger: %d secs.\n", name,
            lng.l_onoff, lng.l_linger);
    break;
case OT_TIMEVAL:
    len = sizeof(tv);
    if (getsockopt(skt, level, option, &tv, &len) == -1)
        printf("%s FAILED (%s)\n", name, strerror(errno));
    else
        printf("%s = %ld secs.; %ld usecs.\n", name,
            (long)tv.tv_sec, (long)tv.tv_usec);
}
}

static void showall(int skt, const char *caption)
{
    printf("\n%s\n", caption);
    show(skt, SOL_SOCKET, SO_ACCEPTCONN, "SO_ACCEPTCONN", OT_INT);
    show(skt, SOL_SOCKET, SO_BROADCAST, "SO_BROADCAST", OT_INT);
    show(skt, SOL_SOCKET, SO_DEBUG, "SO_DEBUG", OT_INT);
    show(skt, SOL_SOCKET, SO_DONTROUTE, "SO_DONTROUTE", OT_INT);
    show(skt, SOL_SOCKET, SO_ERROR, "SO_ERROR", OT_INT);
    show(skt, SOL_SOCKET, SO_KEEPAIVE, "SO_KEEPAIVE", OT_INT);
    show(skt, SOL_SOCKET, SO_LINGER, "SO_LINGER", OT_LINGER);
    show(skt, SOL_SOCKET, SO_OOBINLINE, "SO_OOBINLINE", OT_INT);
    show(skt, SOL_SOCKET, SO_RCVBUF, "SO_RCVBUF", OT_INT);
    show(skt, SOL_SOCKET, SO_RCVLOWAT, "SO_RCVLOWAT", OT_INT);
    show(skt, SOL_SOCKET, SO_RCVTIMEO, "SO_RCVTIMEO", OT_TIMEVAL);
    show(skt, SOL_SOCKET, SO_REUSEADDR, "SO_REUSEADDR", OT_INT);
    show(skt, SOL_SOCKET, SO_SNDBUF, "SO_SNDBUF", OT_INT);
    show(skt, SOL_SOCKET, SO_SNDLOWAT, "SO_SNDLOWAT", OT_INT);
    show(skt, SOL_SOCKET, SO_SNDTIMEO, "SO_SNDTIMEO", OT_TIMEVAL);
    show(skt, SOL_SOCKET, SO_TYPE, "SO_TYPE", OT_INT);
}

int main(void)
{
    int skt;

    ec_negl( skt = socket(AF_INET, SOCK_STREAM, 0) )
    showall(skt, "AF_INET SOCK_STREAM");
    ec_negl( close(skt) )
    ec_negl( skt = socket(AF_INET, SOCK_DGRAM, 0) )
    showall(skt, "AF_INET SOCK_DGRAM");
    ec_negl( close(skt) )
    exit(EXIT_SUCCESS);
}

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}

```



在Solaris上的输出为:

```
AF_INET SOCK_STREAM
SO_ACCEPTCONN = 0
SO_BROADCAST = 0
SO_DEBUG = 0
SO_DONTROUTE = 0
SO_ERROR = 0
SO_KEEPAIVE = 0
SO_LINGER = l_onoff: 0; l_linger: 0 secs.
SO_OOINLINE = 0
SO_RCVBUF = 65536
SO_RCVLOWAT FAILED (Option not supported by protocol)
SO_RCVTIMEO FAILED (Option not supported by protocol)
SO_REUSEADDR = 0
SO_SNDBUF = 65536
SO_SNDLOWAT FAILED (Option not supported by protocol)
SO_SNDTIMEO FAILED (Option not supported by protocol)
SO_TYPE = 2
```

```
AF_INET SOCK_DGRAM
SO_ACCEPTCONN = 0
SO_BROADCAST = 0
SO_DEBUG = 0
SO_DONTROUTE = 0
SO_ERROR = 0
SO_KEEPAIVE = 0
SO_LINGER = l_onoff: 0; l_linger: 0 secs.
SO_OOINLINE = 0
SO_RCVBUF = 65536
SO_RCVLOWAT FAILED (Option not supported by protocol)
SO_RCVTIMEO FAILED (Option not supported by protocol)
SO_REUSEADDR = 0
SO_SNDBUF = 65536
SO_SNDLOWAT FAILED (Option not supported by protocol)
SO_SNDTIMEO FAILED (Option not supported by protocol)
SO_TYPE = 1
```

8.4 简单套接字接口

当需要使用套接字时,一般不是调用getaddrinfo、socket、bind、connect或其他相关的函数,而是编写更高层的函数来隐藏其中一些冗长的细节。这里称这些函数接口为SSI,即简单套接字接口(Simple Socket Interface, SSI)。

8.4.1 SSI函数调用

SSI函数将打开连接的状态保存在SSI结构类型中,后面马上就会讲到这些。当调用ssi_open时,就能得到指向SSI的指针;完成任务后,使用ssi_close关闭SSI。

ssi_open——打开SSI连接

```
SSI *ssi_open(
    const char *name, /* server name */
    bool server /* called from server? */
);
/* Returns pointer to SSI or NULL on error (sets errno) */
```

ssi_close——关闭SSI连接

```
bool ssi_close(
    SSI *ssip           /* pointer to SSI */
);
/* Returns true on success or false on error (sets errno) */
```

可以看出，`ssi_open`封装了套接字地址的构建（比如通过调用`getaddrinfo`）及对`socket`、`bind`、`listen`和`connect`的调用。

如果传递给`ssi_open`的名字以两条正斜杠开始（它们不是名字的一部分），那么这个名字就作为`AF_INET`主机名字，名字后面必须紧跟冒号和端口号。如果它不以两个正斜杠开始，那么这个名字就是一个如8.1.1节所讲的本地`AF_UNIX`名字一样。比如：

```
//www.basepath.com:80  连接到www.basepath.com主机的80端口的AF_INET连接
//firecracker:31000  连接到firecracker主机31000端口的AF_INET连接
//216.109.125.43:21  216.109.125.43主机21端口的AF_INET连接
MyServer AF_UNIX连接
```

服务器调用`ssi_wait_server`等待客户端文件描述符准备好；它封装了对`select`和`accept`的调用，我们在8.1.3节中讲到过：

ssi_wait_server——等待文件描述符准备好

```
int ssi_wait_server(
    SSI *ssip           /* pointer to SSI */
);
/* Returns file descriptor or -1 on error (sets errno) */
```

客户端调用`ssi_get_server_fd`得到与服务器的连接的文件描述符：

ssi_get_server_fd——得到服务器的文件描述符

```
int ssi_get_server_fd(
    SSI *ssip           /* pointer to SSI */
);
/* Returns file descriptor or -1 on error (sets errno) */
```

最后，当服务器从客户端文件描述符中得到EOF标记或者知道不再需要它时，服务器就调用`ssi_close_fd`：

ssi_close_fd——关闭客户文件描述符

```
bool ssi_close_fd(
    SSI *ssip,         /* pointer to SSI */
    int fd             /* file descriptor */
);
/* Returns true on success or false on error (sets errno) */
```

以上这些就是我们通过`AF_UNIX`或`AF_INET`采用有连接（`SOCK_STREAM`）客户端来实现简单服务器和客户端时所需要的全部函数。后面将会给出简单的Web浏览器和简单的Web服务器，届时将说明如何使用这些函数。然后讲解SSI的实现。

8.4.2 HTTP简介

超文本传输协议（HTTP）是由因特网工程任务组（Internet Engineering Task Force，

IETF) 在一份称为RFC 2616的文档中定义的。在它们的Web站点[RFC]上, 可以读到全部的RFC以及其他文档。下面是对HTTP非常简单的介绍——仅足够理解本节中的例子。

对于HTTP, 客户端通常是Web浏览器, 服务器是Web服务器。一旦建立了连接, 客户端通过向服务器发送以下形式的字符串来开始交互:

```
GET path HTTP/version \r\n\r\n
```

其中, *path*是要求的文档 (比如/index.html), *version*是所使用的HTTP版本 (比如1.0)。

服务器判断该文档是否存在, 客户端是否有权访问该文档。如果不允许访问, 它将用如下状态行应答:

```
HTTP/1.1 404 Not Found\r\n
```

后面还跟有一个HTML文档对错误进行解释。(后面马上会解释文档是如何发送的。)

如果该文件可以发送, 那么状态行类似于:

```
HTTP/1.1 200 OK\r\n
```

每一份文档, 不管是HTML文本、JPEG或其他任何东西, 前面都有一个头对其进行描述。这个头描述有统一的形式 (至少在本书的例子中是这样的):

```
Server: servername\r\nContent-Length: length\r\nContent-Type: type\r\n\r\n
```

*servername*可以是任何名字; 这里使用“AUP-ws”作为服务器。*length*是以字节表示的文档的长度, 这样客户端就知道要读的数据有多少。因为客户端不需要以EOF告知停止读数据, 所以连接就可以保持打开状态。*type*是所谓的多用途因特网邮件扩展 (Multipurpose Internet Mail Extension, MIME) 类型, 该类型中我们关注的有两种: “text/html”和“image/jpeg” (还有几十种)。头之后就是文档, 和在服务器文件系统中完全一致。

8.4.3 SSI Web浏览器

这里有一个简单的Web浏览器, 称为minibr。如前面的例子所示, 它可以检索HTML; 但是, 它不知道如何解释这些标签以达到很好的屏幕显示效果, 所以就简单地将所有东西转储到标准输出上:

```
int main(void)
{
    char url[100], s[500], *path = "", *p;
    SSI *ssip;
    int fd;
    ssize_t nread;

    while (true) {
        printf("URL: ");
        if (fgets(url, sizeof(url), stdin) == NULL)
            break;
        if ((p = strrchr(url, '\n')) != NULL)
            *p = '\0';
        if ((p = strchr(url, '/')) != NULL) {
            path = p + 1;
            *p = '\0';
        }
        snprintf(s, sizeof(s), "://%s:80", url);
        ec_null( ssip = ssi_open(s, false) )
        ec_negl( fd = ssi_get_server_fd(ssip) )
    }
}
```



```

    snprintf(s, sizeof(s), "GET /%s HTTP/1.0\r\n\r\n", path);
    ec_negl( writeall(fd, s, strlen(s)) )
    while (true) {
        switch (nread = read(fd, s, sizeof(s))) {
            case 0:
                printf("EOF\n");
                break;
            case -1:
                EC_FAIL
            default:
                ec_negl( writeall(STDOUT_FILENO, s, nread) )
                continue;
        }
        break;
    }
    ec_false( ssi_close(ssip) )
}
ec_false( !ferror(stdin) )
exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}

```

紧随调用fgets后的代码把输入的URL分成了两部分：主机名和路径（如8.2.5节所讲），在主机名前面加上斜杠，后面加上端口80，之后传递给ssi_open。当建立连接后，路径被格式化成发送给Web服务器的HTTP GET请求。

下面是minibr会话的简单示例，其中大大地简化了每一个检索到的页：

```

URL: www.basepath.com
HTTP/1.1 200 OK
Date: Sat, 19 Jul 2003 19:01:41 GMT
Server: Apache/1.3.27 (Unix) FrontPage/5.0.2.2510 mod_jk/1.1.0
Last-Modified: Thu, 15 May 2003 19:56:49 GMT
ETag: "61744-191-3ec3f101"
Accept-Ranges: bytes
Content-Length: 401
Connection: close
Content-Type: text/html

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html>

<head>
...
URL: 216.109.125.70
HTTP/1.1 200 OK
Date: Sat, 19 Jul 2003 19:02:49 GMT
P3P: policyref="http://p3p.yahoo.com/w3c/p3p.xml", CP="CAO DSP COR CUR ADM
...
Cache-Control: private
Connection: close
Content-Type: text/html

</html><head>

<title>Yahoo!</title>
...

```

8.4.4 SSI Web服务器

到目前为止，已经讲解了客户端的情况——现在要讲解的就是服务器了。简单Web服务器查找来自客户端的GET请求，抽取出路经，然后写出响应，并紧跟要发给客户端的文件描述符，如8.4.2节所示。

服务器端程序以一些宏定义开始，这些宏表明了它所能提供的HTTP响应和“not found”错误的HTML代码：

```
#define HEADER\
  "HTTP/1.0 %s\r\n"\
  "Server: AUP-ws\r\n"\
  "Content-Length: %ld\r\n"

#define CONTENT_TEXT\
  "Content-Type: text/html\r\n\r\n"

#define CONTENT_JPEG\
  "Content-Type: image/jpeg\r\n\r\n"

#define HTML_NOTFOUND\
  "<!DOCTYPE html PUBLIC "-//IETF//DTD HTML 2.0//EN">\n"\
  "<html><head><title>Error 404</title>\n"\
  "</head><body>\n"\
  "<h2>AUP-ws server can't find document</h2>\n"\
  "</body></html>\r\n"
```

注意：当函数send_header正使用HEADER时，HEADER包含的格式代码会被状态代码和内容长度所取代：

```
static void send_header(SSl *ssip, const char *msg, off_t len,
  const char *path, int fd)
{
  char buf[1000], *dot;

  snprintf(buf, sizeof(buf), HEADER, msg, (long)len);
  ec_negl( writeall(fd, buf, strlen(buf)) )
  dot = strrchr(path, '.');
  if (dot != NULL && (strcasemp(dot, ".jpg") == 0 ||
    strcasemp(dot, ".jpeg") == 0))
    ec_negl( writeall(fd, CONTENT_JPEG, strlen(CONTENT_JPEG)) )
  else
    ec_negl( writeall(fd, CONTENT_TEXT, strlen(CONTENT_TEXT)) )
  return;

  EC_CLEANUP_BGN
  EC_FLUSH("Error sending response (nonfatal)")
  EC_CLEANUP_END
}
```

该函数调用了带有“404 Not Found”或者“200 OK”的消息参数msg，正如在8.4.2节所解释的那样。len参数是文档的长度，path是其文件名，仅仅用来判断文件是JPEG还是HTML。send_header的调用者负责发送该文件。

send_header被函数handle_request调用，handle_request负责处理GET请求：

```
#define DEFAULT_DOC "index.html"
#define WEB_ROOT "/aup/webroot/"

static bool handle_request(SSl *ssip, char *s, int fd)
```

```

{
    char *tkn, buf[1000], path[500];
    int ntkn;
    FILE *in;
    struct stat statbuf;
    ssize_t nread;

    for (ntkn = 1; (tkn = strtok(s, " ")) != NULL; ntkn++) {
        s = NULL;
        switch (ntkn) {
            case 1:
                if (strcasemp(tkn, "get") != 0) {
                    printf("Unknown request\n");

                    return false;
                }
                continue;
            case 2:
                break;
        }
        break;
    }
    snprintf(path, sizeof(path) - 1 - strlen(DEFAULT_DOC),
             "%s%s", WEB_ROOT, tkn);
    if (stat(path, &statbuf) == 0 && S_ISDIR(statbuf.st_mode)) {
        if (path[strlen(path) - 1] != '/')
            strcat(path, "/");
        strcat(path, DEFAULT_DOC);
    }
    if (stat(path, &statbuf) == -1 ||
        (in = fopen(path, "rb")) == NULL) {
        send_header(ssip, "404 Not Found", strlen(HTML_NOTFOUND), "",
                  fd);
        ec_negl( writeall(fd, HTML_NOTFOUND, strlen(HTML_NOTFOUND)) )
        return false;
    }
    send_header(ssip, "200 OK", statbuf.st_size, path, fd);
    while ((nread = fread(buf, 1, sizeof(buf), in)) > 0)
        ec_negl( writeall(fd, buf, nread) )
    ec_eof( fclose(in) )
    return true;

EC_CLEANUP_BGN
    EC_FLUSH("Error sending response (nonfatal)")
    return false;
EC_CLEANUP_END
}

```

for 循环只是用来检查请求是否确实是某种 GET 请求并抽出路径名（这个路径名作为 WEB_ROOT 的子目录），以确保不是将本地系统上的所有路径都提出来。如果该路径是目录，那么会在该路径后加上默认的 HTML 文件 index.html；否则，该路径被假定以 HTML 或 JPEG 文件命名。

如果路径不存在，就送回 404 头，后面会跟一些 HTML。否则，送回 200 头，后面跟的是对应的文件。注意：该文件可能是文本或二进制 JPEG。

最后，看一看主程序。该主程序包含了处理与客户端连接的实际工作中所调用的 SSI 函数：

```

#define PORT ":8000"

int main(void)
{
    SSI *ssip = NULL;
    char msg[1600];
    ssize_t nrcv;
    int fd;
    char host[100] = "/*";

    ec_negl( gethostname(&host[2], sizeof(host) - 2 - strlen(PORT)) )
    strcat(host, PORT);
    printf("Connecting to host \"%s\"\n", host);
    ec_null( ssip = ssi_open(host, true) )
    printf("\t...connected\n");
    while (true) {
        ec_negl( fd = ssi_wait_server(ssip) )
        switch (nrcv = read(fd, msg, sizeof(msg) - 1)) {
            case -1:
                printf("Read error (nonfatal)\n");
                /* fall through */
            case 0:
                ec_false( ssi_close_fd(ssip, fd) )
                continue;
            default:
                msg[nrcv] = '\0';
                (void)handle_request(ssip, msg, fd);
        }
    }
    ec_false( ssi_close(ssip) )
    printf("Done.\n");
    exit(EXIT_SUCCESS);

    EC_CLEANUP_BGN
        return EXIT_FAILURE;
    EC_CLEANUP_END
}

```

ssi_open所用的服务器名字形式上类似于:

```
//host:8000
```

之所以使用端口8000, 是因为该系统中端口80已经被一个实际的Web服务器 (Apache) 使用, 而且1024以下的端口号只能由超级用户使用。

该函数进行连接, 然后进入一个循环。在该循环中先等待准备好的文件描述符, 再读一条命令, 然后将该命令送给handle_request函数。如果客户端中断, 或者简单关闭连接 (有些客户有时会做该操作), 服务器会得到EOF标记。这种情况下, 就像8.1.3节所讲的那样, 程序就必须调用ssi_close_fd来告知ssi_wait_server不要再考虑那个文件描述符。

这个Web服务器确实管用, 你可以使用任何浏览器对它进行访问, 不限于前一节中所讲的那个简单浏览器。例如, 图8-3展示了采用URL <http://suse2:8000>访问服务器的结果:

服务器 (之所以被称为suse2是因为它上面运行的是SuSE Linux) 上aup/webroot/index.html的HTML代码是:

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<html>
<head>
</head>
<body>

```

```
<h1>Marc's Bike</h1>
<p>
</body>
</html>
```



图8-3 在Macintosh机器上采用Safari浏览器访问SSI Web服务器

8.4.5 SSI实现

已经讨论了在实现SSI函数中所涉及的所有问题，还给出了一两个例子的完整代码。可以复习8.1.3节、8.2.2节和8.2.6节。

下面是SSI类型的结构：

```
#define SSI_NAME_SIZE 200

typedef struct {
    bool ssi_server;           /* server? (vs. client) */
    int ssi_domain;           /* AF_INET or AF_UNIX */
    int ssi_fd;                /* socket fd */
    char ssi_name_server[SSI_NAME_SIZE]; /* server name */
    fd_set ssi_fd_set;        /* set for server's select */
    int ssi_fd_hwm;           /* high-water-mark for fds seen */
} SSI;
```

在后面的代码分析中将会看到这些成员变量是如何使用的。

先看一个经常使用的函数make_sockaddr。该函数用来建立AF_UNIX或AF_INET的套接字地址。AF_INET的名字参数是主机：端口格式的，而AF_UNIX的名字参数只是一般的名。will_bind参数表明套接字地址是否会用于bind或connect，如8.2.6节所述。

```
bool make_sockaddr(struct sockaddr *sa, socklen_t *len, const char *name,
    int domain, bool will_bind)
{
    struct addrinfo *infop = NULL;

    if (domain == AF_UNIX) {
        struct sockaddr_un *sunp = (struct sockaddr_un *)sa;
```

```

    if (strlen(name) >= sizeof(sunp->sun_path)) {
        errno = ENAMETOOLONG;
        EC_FAIL
    }
    strcpy(sunp->sun_path, name);
    sunp->sun_family = AF_UNIX;
    *len = sizeof(*sunp);
}
else {
    struct addrinfo hint;
    char nodename[SSI_NAME_SIZE], *servicename;

    memset(&hint, 0, sizeof(hint));
    hint.ai_family = domain;
    hint.ai_socktype = SOCK_STREAM;
    if (will_bind)
        hint.ai_flags = AI_PASSIVE;
    strcpy(nodename, name);
    servicename = strchr(nodename, ':');
    if (servicename == NULL) {
        errno = EINVAL;
        EC_FAIL
    }
    *servicename = '\0';
    servicename++;
    ec_ai( getaddrinfo(nodename, servicename, &hint, &infp) )
    memcpy(sa, infop->ai_addr, infop->ai_addrlen);
    *len = infop->ai_addrlen;
    freeaddrinfo(infp);
}
return true;

EC_CLEANUP_BGN
    if (infp != NULL)
        freeaddrinfo(infp);
    return false;
EC_CLEANUP_END
}

```

这里故意将make_sockaddr从SSI结构中摘了出来，以便在你需要时可以将其抽出用于你自己的程序。

接下来来看ssi_open和ssi_close中使用的两个函数，这两个函数用来维护文件描述符的高水位标志，如8.1.3节所示：

```

static void set_fd_hwm(SSI *ssip, int fd)
{
    if (fd > ssip->ssi_fd_hwm)
        ssip->ssi_fd_hwm = fd;
}

static void reset_fd_hwm(SSI *ssip, int fd)
{
    if (fd == ssip->ssi_fd_hwm)
        ssip->ssi_fd_hwm--;
}

```

现在来看是ssi_open，这个函数能完成已经讲过得很多功能：

```

SSI *ssi_open(const char *name_server, bool server)
{

```



```

SSI *ssip = NULL;
struct sockaddr_storage sa;
socklen_t sa_len;

ec_null( ssip = calloc(1, sizeof(SSI)) )
ssip->ssi_server = server;
if (strncmp(name_server, "///<", 2) == 0) {
    ssip->ssi_domain = AF_INET;
    name_server += 2;
}
else {
    ssip->ssi_domain = AF_UNIX;
    if (ssip->ssi_server)
        (void)unlink(name_server);
}
if (strlen(name_server) >= sizeof(ssip->ssi_name_server)) {
    errno = ENAMETOOLONG;
    EC_FAIL
}
strcpy(ssip->ssi_name_server, name_server);
ec_false( make_sockaddr((struct sockaddr *)&sa, &sa_len,
    ssip->ssi_name_server, ssip->ssi_domain, ssip->ssi_server) )
ec_negl( ssip->ssi_fd = socket(ssip->ssi_domain, SOCK_STREAM, 0) )
set_fd_hwm(ssip, ssip->ssi_fd);
if (ssip->ssi_domain == AF_INET) {
    int socket_option_value = 1;

    ec_negl( setsockopt(ssip->ssi_fd, SOL_SOCKET, SO_REUSEADDR,
        &socket_option_value, sizeof(socket_option_value)) )
}
if (ssip->ssi_server) {
    FD_ZERO(&ssip->ssi_fd_set);
    ec_negl( bind(ssip->ssi_fd, (struct sockaddr *)&sa, sa_len) )
    ec_negl( listen(ssip->ssi_fd, SOMAXCONN) )
    FD_SET(ssip->ssi_fd, &ssip->ssi_fd_set);
}
else
    ec_negl( connect(ssip->ssi_fd, (struct sockaddr *)&sa, sa_len) )
return ssip;

EC_CLEANUP_BGN
free(ssip);
return NULL;
EC_CLEANUP_END
)

```

在8.3节中解释了套接字选项SO_REUSEADDR。

关闭SSI是很简单的:

```

bool ssi_close(SSI *ssip)
{
    if (ssip->ssi_server) {
        int fd;

        for (fd = 0; fd <= ssip->ssi_fd_hwm; fd++)
            if (FD_ISSET(fd, &ssip->ssi_fd_set))
                (void)close(fd);
        if (ssip->ssi_domain == AF_UNIX)
            (void)unlink(ssip->ssi_name_server);
    }
}

```



```

else
    (void)close(ssip->ssi_fd);
free(ssip);
return true;
}

```

下面来看`ssi_wait_server`，该函数只是重写了8.1.3节中的代码，只不过它返回的是准备好的文件描述符（即，不同于服务器的套接字）而不是自己使用这个参数：

```

int ssi_wait_server(SSi *ssip)
{
    if (ssip->ssi_server) {
        fd_set fd_set_read;
        int fd, clientfd;
        struct sockaddr_un from;
        socklen_t from_len = sizeof(from);

        while (true) {
            fd_set_read = ssip->ssi_fd_set;
            ec_negl( select(ssip->ssi_fd_hwm + 1, &fd_set_read, NULL, NULL,
                NULL) );
            for (fd = 0; fd <= ssip->ssi_fd_hwm; fd++)
                if (FD_ISSET(fd, &fd_set_read)) {
                    if (fd == ssip->ssi_fd) {
                        ec_negl( clientfd = accept(ssip->ssi_fd,
                            (struct sockaddr *)&from, &from_len) );
                        FD_SET(clientfd, &ssip->ssi_fd_set);
                        set_fd_hwm(ssip, clientfd);
                        continue;
                    }
                    else
                        return fd;
                }
        }
    }
    else {
        errno = ENOTSUP;
        EC_FAIL
    }
}

EC_CLEANUP_BGN
return -1;
EC_CLEANUP_END
}

```

还有两个不重要的函数：

```

int ssi_get_server_fd(SSi *ssip)
{
    return ssip->ssi_fd;
}

bool ssi_close_fd(SSi *ssip, int fd)
{
    ec_negl( close(fd) );
    FD_CLR(fd, &ssip->ssi_fd_set);
    reset_fd_hwm(ssip, fd);
    return true;
}

EC_CLEANUP_BGN

```



```

    return false;
EC_CLEANUP_END
}

```

所有这些都是全部的实现了。对于有连接套接字 (SOCK_STREAM), 在以后的示例中不必再使用任何的原始系统调用了。

8.5 SMI套接字实现

用套接字来实现第7章的简单消息接口 (SMI) 是很有趣的, 这样可以将它和第7章的其他5种实现进行对比。因为我们可以使用前一节的SSI函数, 不用直接调用各种与套接字相关的系统调用, 所以这项工作很简单。

本节假定你很熟悉第7章以及那一章的SMI实现; 如果你不是很熟悉, 那么可以暂时跳过这一节, 直到你有时间阅读第7章——否则, 阅读本章毫无意义。

对于套接字, 内部的SMIQ_SKT结构很简单, 很大原因在于跟踪客户端的工作 (这对其他实现来说是很麻烦的) 在这种情况下是由accept系统调用完成的, 而且使用SSI来处理了大量的细节:

```

typedef struct {
    SMIENTITY sq_entity;           /* entity */
    SSI *sq_ssip;                 /* structure for SSI */
    struct client_id sq_client;   /* client ID */
    size_t sq_msgsize;           /* msg size */
    struct smi_msg *sq_msg;       /* msg buffer */
} SMIQ_SKT;

```

因为ssi_open完成了大多数工作, 所以打开SMIQ_SKT没有多少工作要做:

```

SMIQ *smi_open_skt(const char *name, SMIENTITY entity, size_t msgsize)
{
    SMIQ_SKT *p = NULL;

    ec_null( p = calloc(1, sizeof(SMIQ_SKT)) )
    p->sq_msgsize = msgsize + offsetof(struct smi_msg, smi_data);
    ec_null( p->sq_msg = calloc(1, p->sq_msgsize) )
    p->sq_entity = entity;
    ec_null( p->sq_ssip = ssi_open(name, entity == SMI_SERVER) )
    return (SMIQ *)p;

EC_CLEANUP_BGN
    (void)smi_close_skt((SMIQ *)p);
    return NULL;
EC_CLEANUP_END
}

```

同样关闭如下:

```

bool smi_close_skt(SMIQ *sqp)
{
    SMIQ_SKT *p = (SMIQ_SKT *)sqp;
    SSI *ssip;

    if (p != NULL) {
        ssip = p->sq_ssip;
        free(p->sq_msg);
        free(p);
    }
}

```



```

        if (ssip != NULL)
            ec_false( ssi_close(ssip) )
    }
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

```

与第7章中的一些其他实现一样，如果`smi_send_getaddr_skt`是由服务器调用的，并且返回消息地址，那么所有的`smi_send_getaddr_skt`必须要作的是保存`client_id`：

```

bool smi_send_getaddr_skt(SMIQ *sqp, struct client_id *client,
    void **addr)
{
    SMIQ_SKT *p = (SMIQ_SKT *)sqp;

    if (p->sq_entity == SMI_SERVER)
        p->sq_client = *client;
    *addr = p->sq_msg;
    return true;
}

```

`smi_send_release_skt`写消息（用2.9节的`writeall`）：如果消息来自服务器，就写到客户端的文件描述符，或者如果消息来自客户端，就写到服务器的套接字文件描述符：

```

bool smi_send_release_skt(SMIQ *sqp)
{
    SMIQ_SKT *p = (SMIQ_SKT *)sqp;
    int fd;

    if (p->sq_entity == SMI_SERVER)
        ec_negl( fd = p->sq_client.c_id1 )
    else
        ec_negl( fd = ssi_get_server_fd(p->sq_ssip) )
    ec_negl( writeall(fd, p->sq_msg, p->sq_msgsize) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

```

对于服务器，通过服务器接收消息意味着必须等待一个准备好的文件描述符，这由SSI通过`ssi_wait_server`来处理。服务器还必须保存已经以消息的`client_id`部分准备好的文件描述符，这样调用者（服务器）知道如何响应。客户端必须要作的就是调用`ssi_get_server_fd`来获得服务器的套接字文件描述符：

```

bool smi_receive_getaddr_skt(SMIQ *sqp, void **addr)
{
    SMIQ_SKT *p = (SMIQ_SKT *)sqp;
    ssize_t /*nremain,*/ nread;
    int fd;

    *addr = p->sq_msg;
    while (true) {
        if (p->sq_entity == SMI_SERVER)

```

```

        ec_negl( fd = ssi_wait_server(p->sq_ssi) )
    else
        ec_negl( fd = ssi_get_server_fd(p->sq_ssi) )
    ec_negl( nread = readall(fd, p->sq_msg, p->sq_msgsize) )
    if (nread == 0) {
        if (p->sq_entity == SMI_SERVER) {
            ec_false( ssi_close_fd(p->sq_ssi, fd) )
            continue;
        }
        else {
            errno = ENOMSG;
            EC_FAIL
        }
    }
    else
        break;
}
if (p->sq_entity == SMI_SERVER)
    p->sq_msg->smi_client.c_idl = fd;
return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

```

最后一个函数实际上没有任何工作可做:

```

bool smi_receive_release_skt(SMIQ *sqp)
{
    return true;
}

```

通过以上SMI实现，第7章的所有发送消息的例子就都可以工作在套接字上了。在7.15节中已经对比了它与其他IPC方法之间的性能。

注意：在设计用于一台机器中的IPC的SMI的例子中，使用的都是一般的服务器名字。这意味着当名字传递给`ssi_open`时，这个名字被看作在`AF_UNIX`域内。尽管名字必须以两条斜杠开始，这样SMI的套接字实现才会工作，但是你仍然可以同样简单地在机器之间的`AF_INET`消息发送中使用SMI接口。

8.6 无连接套接字

目前为止，所有的例子以及SSI实现使用的都是`SOCK_STREAM`类型的有连接套接字。一台服务器调用`listen`和`accept`，一台或一台以上的客户机连接到服务器。连接一直处于打开状态，直到有一方关闭了这条连接。

相反，无连接套接字不需要使用连接，所以就不调用`listen`和`accept`。发送者指明它要发送的地址。接收者指明它要接收的地址，或者可以从任何地址接收并被告知发送者的地址。接收者总是必须绑定到一个外部名字，这是发送者可以到达它们的唯一方式。这样，无连接套接字比有连接套接字更加对称；它们之间可能有客户机/服务器的关系，但有时认为它们之间是对等关系更有用。

8.6.1 关于数据报

无连接通信使用数据报 (datagram)，数据报是包含目标地址的独立的数据块。数据报不

保持一定的顺序，也不保证它们同时到达。相反，SOCK_STREAM必须保证顺序和到达时间。可以把数据报想象为邮件或者电子邮件，把SOCK_STREAM想象为打电话。

在许多应用中，顺序和到达时间并不重要。例如，假设应用是为了预定图书馆图书。在屏幕上有一个表单，当提交表单时，向图书馆的服务器发送一个数据报，服务器也给出了确认的响应。如果一个预定请求比另一个提前了一两秒钟到达，那么数据报的重新排序就有可能带来不公平。但是，由于订阅人之间意识不到其他人的行为，所以不会出现抱怨的情况。在极少数情况下，数据报可能根本没有到达，订阅人得不到确认，只能重新发送表单。所以，在这种应用中，与只是为了订阅图书而建立一条连接然后又立即断开连接相比，数据报要高效得多。另一方面，如果订阅人想交互式地浏览一会儿书目，那么建立连接恐怕是最合适的选择。

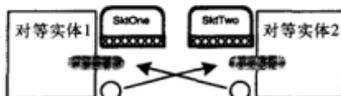


图8-4 对等实体发送数据报

图8-4显示了两个对等实体之间互相发送数据报到套接字的情况。与图8-1（显示的是有连接套接字）

对比一下。用于侦听和接收的符号不见了，但是两边的套接字都绑定了名字，而不只是对服务器套接字进行了绑定。

8.6.2 sendto和recvfrom系统调用

在所有前面的套接字例子中，使用的都是write（或writeall）来向套接字发送数据，因为拥有文件描述符来代表进行写操作连接端点，所以这样做是有效的。但是，如果套接字是无连接的，对自己的套接来说，就只有一个文件描述符；需要将目标指定为套接字地址。所以，write就不再奏效了。我们需要带有套接字地址的sendto系统调用：

sendto——向套接字发送消息

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

ssize_t sendto(
    int socket_fd,          /* socket file descriptor */
    const void *message,   /* message to send */
    size_t length,        /* length of message */
    int flags,             /* flags */
    const struct sockaddr *sa, /* target address */
    socklen_t sa_len      /* length of target address */
);
/* Returns number of bytes sent or -1 on error (sets errno) */
```

socket_fd参数是发送者的套接字；接收者是用sa参数和sa_len参数来指定的。sendto的成功返回并不意味着报文已经成功到达，只是说明本地没有检测到错误。

你的实现中可能定义了一些标志，但唯一重要的常用标志是MSG_OOB，它是用来发送带外（紧急）数据的；见8.7节。

对于无连接套接字，length个字节长度的消息被认为是一个不能分割的数据报。如果套接字通道已满，那么sendto通常会阻塞，直到所有的消息适合发送为止；如果设置了O_NONBLOCK标志并且有一个消息不适合发送，则都不发送，返回-1，并将errno设置为EAGAIN或EWOULDBLOCK。

你也可以在有连接套接字中使用sendto，但在那种情况下，目标地址就被忽略了，输出发送给socket_fd套接字，和write用法差不多。与write相比的优点在于你可以指定标志。但是，如果这就是你想要的，那么还有更直接的方式，就是使用send（见8.9.1节）。

现在看一个例子代码，它完成了图8-4所示的功能，不过增加了双方读取进入数据的功能。对等实体2取得报文，进行一点点修改并将它发送给对等实体1。对等实体1将4份报文放在一起发送给对等实体2，并显示响应。注意，与有连接套接字例子不同的是，对等双方都作绑定，都不调用listen、accept或connect。

```
#define SOCKETNAME1 "SktOne"
#define SOCKETNAME2 "SktTwo"

#define MSG_SIZE 100

int main(void)
{
    struct sockaddr_un sa1, sa2;

    strcpy(sa1.sun_path, SOCKETNAME1);
    sa1.sun_family = AF_UNIX;
    strcpy(sa2.sun_path, SOCKETNAME2);
    sa2.sun_family = AF_UNIX;
    (void)unlink(SOCKETNAME1);
    (void)unlink(SOCKETNAME2);
    if (fork() == 0) { /* Peer 1 */
        int fd_skt;
        ssize_t nread;
        char msg[MSG_SIZE];
        int i;

        sleep(1); /* let peer 2 startup first */
        ec_negl( fd_skt = socket(AF_UNIX, SOCK_DGRAM, 0) )
        ec_negl( bind(fd_skt, (struct sockaddr *)&sa1, sizeof(sa1)) )
        for (i = 1; i <= 4; i++) {
            snprintf(msg, sizeof(msg), "Message #%d", i);
            ec_negl( sendto(fd_skt, msg, sizeof(msg), 0,
                (struct sockaddr *)&sa2, sizeof(sa2)) )
            ec_negl( nread = read(fd_skt, msg, sizeof(msg)) )
            if (nread != sizeof(msg)) {
                printf("Peer 1 got short message\n");
                break;
            }
            printf("Got \"%s\" back\n", msg);
        }
        ec_negl( close(fd_skt) )
        exit(EXIT_SUCCESS);
    }
    else { /* Peer 2 */
        int fd_skt;
        ssize_t nread;
        char msg[MSG_SIZE];

        ec_negl( fd_skt = socket(AF_UNIX, SOCK_DGRAM, 0) )
        ec_negl( bind(fd_skt, (struct sockaddr *)&sa2, sizeof(sa2)) )
        ec_negl( nread = read(fd_skt, msg, sizeof(msg)) )
        while (true) {
            if (nread != sizeof(msg)) {
                printf("Peer 2 got short message\n");
                break;
            }
            msg[0] = 'm';
            ec_negl( sendto(fd_skt, msg, sizeof(msg), 0,
                (struct sockaddr *)&sa1, sizeof(sa1)) )
        }
    }
}
```



```

    }
    ec_negl( close(fd_skt) )
    exit(EXIT_SUCCESS);
}

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}

```

将两个进程放置在同一个程序中不太标准；这里只是用它来使示例简化。然而，我们必须将子进程（对等实体1）的开始部分调用sleep来让其停留一会儿好让对等实体2先行。这种次序并非最好，但对现在这个程序来讲是可行的。在实际的应用中，让服务器在客户机之前运行起来通常不是问题。如果有问题，那么正确的方法是让两者同步起来，见9.2节。

下面是输出：

```

Got "message #1" back
Got "message #2" back
Got "message #3" back
Got "message #4" back

```

对数据报而言，read实际上不是好的选择，因为read没有指出是谁发送了数据。这对有连接套接字不是什么问题，因为从accept返回后都提供了一个独立的文件描述符（每个客户端一个），使用该文件描述符既可以读也可以写。在刚才给出的例子中，对等实体2只是假定它应该sendto实体1，但是，总的来讲，应用远比这要复杂。

毫无疑问，有一个与sendto相对的函数，称为recvfrom：

recvfrom —— 从套接字接收消息

```

#include <sys/socket.h>

ssize_t recvfrom(
    int socket_fd,           /* socket file descriptor */
    void *buffer,          /* buffer for received message */
    size_t length,         /* length of buffer */
    int flags,             /* flags */
    struct sockaddr *sa,    /* address of sender */
    socklen_t *sa_len      /* address length */
);
/* Returns number of bytes received, 0, or -1 on error (sets errno) */

```

recvfrom的前三个参数类似于read的参数。另外，它还有一个flags参数和两个关于接收者套接字地址的参数。在调用之前，必须设置sa使其具有足够大的空间能容纳发送者套接字地址，然后设置sa_len为该空间的大小。在返回上，设置sa_len为实际的地址大小。然后，需要的话，可以将sa和sa_len直接传递给sendto来接收响应。

像sendto一样，recvfrom总是返回无连接套接字的完整消息。如果O_NONBLOCK标识被消除了，那么它就等待一条消息，如果设置了，就返回-1并设置errno为EAGAIN或者EWOULDBLOCK。

如果想知道源地址，你可以使用recvfrom得到有连接套接字。但这个地址没有太多的作用，正如前面解释的那样，因为这个地址会在sendto调用中被忽略。如果你拥有套接字地址的文件描述符的话，那么得到其套接字地址的另一个方法就是调用getsockname（见8.9.2节）。

在使用recvfrom时，有三个常用的标志可供使用：

MSG_OOB 如果有带外数据的话, 则接收带外数据; 见8.7节。

MSG_PEEK 返回报文, 但不读该报文, 而是留给下一次read、recvfrom或其他输入操作。

MSG_WAITALL 只对有连接套接字有效, 让recvfrom阻塞直至整个请求的长度可用, 除非也设置了MSG_PEEK或者该调用被信号、连接终止或错误所中止。因为消息总是不可分割的, 所以对无连接套接字没有作用。

下面是重写的示例代码, 利用了recvfrom函数, 同时有多个对等实体发送给相同的套接字, 所以它是客户机/服务器布置:

```
#define SOCKETNAME_SERVER "SktOne"
#define SOCKETNAME_CLIENT "SktTwo"

static struct sockaddr_un sa_server;

#define MSG_SIZE 100

static void run_client(int nclient)
{
    struct sockaddr_un sa_client;
    int fd_skt;
    ssize_t nrecv;
    char msg[MSG_SIZE];
    int i;

    if (fork() == 0) { /* client */
        sleep(1); /* let server startup first */
        ec_negl( fd_skt = socket(AF_UNIX, SOCK_DGRAM, 0) )
        snprintf(sa_client.sun_path, sizeof(sa_client.sun_path),
            "%s-%d", SOCKETNAME_CLIENT, nclient);
        (void)unlink(sa_client.sun_path);
        sa_client.sun_family = AF_UNIX;
        ec_negl( bind(fd_skt, (struct sockaddr *)&sa_client,
            sizeof(sa_client)) )
        for (i = 1; i <= 4; i++) {
            snprintf(msg, sizeof(msg), "Message #%d", i);
            ec_negl( sendto(fd_skt, msg, sizeof(msg), 0,
                (struct sockaddr *)&sa_server, sizeof(sa_server)) )
            ec_negl( nrecv = read(fd_skt, msg, sizeof(msg)) )
            if (nrecv != sizeof(msg)) {
                printf("client got short message\n");
                break;
            }
            printf("Got \"%s\" back\n", msg);
        }
        ec_negl( close(fd_skt) )
        exit(EXIT_SUCCESS);
    }
    return;
}

EC_CLEANUP_BGN
exit(EXIT_FAILURE);
EC_CLEANUP_END
}

static void run_server(void)
{
    int fd_skt;
    ssize_t nrecv;
    char msg[MSG_SIZE];
    struct sockaddr_storage sa;
```



```

socklen_t sa_len;

ec_negl( fd_skt = socket(AF_UNIX, SOCK_DGRAM, 0) )
ec_negl( bind(fd_skt, (struct sockaddr *)&sa_server, sizeof(sa_server)) )
while (true) {
    sa_len = sizeof(sa);
    ec_negl( nrecv = recvfrom(fd_skt, msg, sizeof(msg), 0,
        (struct sockaddr *)&sa, &sa_len) )
    if (nrecv != sizeof(msg)) {
        printf("server got short message\n");
        break;
    }
    msg[0] = 'm';
    ec_negl( sendto(fd_skt, msg, sizeof(msg), 0,
        (struct sockaddr *)&sa, sa_len) )
}
ec_negl( close(fd_skt) )
exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
exit(EXIT_FAILURE);
EC_CLEANUP_END
}

int main(void)
{
    int nclient;

    strcpy(sa_server.sun_path, SOCKETNAME_SERVER);
    sa_server.sun_family = AF_UNIX;
    (void)unlink(SOCKETNAME_SERVER);
    for (nclient = 1; nclient <= 3; nclient++)
        run_client(nclient);
    run_server();
    exit(EXIT_SUCCESS);
}

```

在这个新的例子中，只有服务器的套接字地址是对服务器和客户机公开的。客户机套接字地址只有客户机知道；服务器通过调用recvfrom得到客户机套接字地址。这里将read调用全都放在了客户端中，但这些调用同样也可以使用recvfrom。因为示例中现在有三个客户端，所以可以有更多的输出：

```

Got "message #1" back
Got "message #2" back
Got "message #3" back
Got "message #4" back
Got "message #1" back
Got "message #2" back
Got "message #3" back
Got "message #4" back
Got "message #1" back
Got "message #2" back
Got "message #3" back
Got "message #4" back

```

8.6.3 sendmsg和recvmsg

sendto和recvfrom有几个使用相同标志和相同返回值的变体，但这些变体使用的都是

一个单独的msg_hdr结构类型参数，该结构参数包含了套接字地址和消息本身。它们可以进行分散读和集中写，就像readv和writev一样，这两个调用在前面的2.15节中讲到过。

sendmsg —— 使用msg_hdr结构向套接字发送消息

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

ssize_t sendmsg(
    int socket_fd,           /* socket file descriptor */
    const struct msg_hdr *message, /* message */
    int flags                /* flags */
);
/* Returns number of bytes sent or -1 on error (sets errno) */
```

recvmsg —— 使用msg_hdr结构从套接字接收消息

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

ssize_t recvmsg(
    int socket_fd,           /* socket file descriptor */
    struct msg_hdr *message, /* message */
    int flags                /* flags */
);
/* Returns number of bytes received, 0, or -1 on error (sets errno) */
```

struct msg_hdr —— sendmsg和recvmsg的结构

```
struct msg_hdr {
    void *msg_name;           /* optional address */
    socklen_t msg_namelen;    /* size of address */
    struct iovec *msg_iov;     /* scatter/gather array */
    int msg_iovlen;           /* number of elements in msg_iov */
    void *msg_control;        /* ancillary data */
    socklen_t msg_controllen; /* ancillary data buffer len */
    int msg_flags;           /* flags on received message */
};
```

对于sendmsg，套接字地址是由错误命名的msg_name成员变量指出的，套接字地址长度是由msg_namelen指出的。要发送的数据是由msg_iov成员变量指出的，该成员变量指向一个iovec结构，用法和在writev中一样。msg_iovlen成员变量指的是msg_iov数组中成员的个数。在2.15节中讲过这些成员的使用。msg_flags成员变量在此用不上，被忽略。

对于recvmsg，可以将msg_name设置为NULL或者是用于接收发送者套接字地址的缓冲区，缓冲区的长度为msg_namelen，这些和recvfrom有些类似。当recvmsg返回时，msg_namelen会被转换为实际的套接字地址长度。成员变量msg_iov和msg_iovlen用于readv。在返回的msg_flags成员变量中，有一个可以设置的常用标志：MSG_TRUNC，意思是报文太长，超出了所提供的缓冲区长度。

套接字成员变量msg_name和msg_namelen用于无连接套接字；有连接套接字用不上它们，忽略不用。

还有两个成员变量，即msg_control和msg_controllen，还没有解释。它们用于与访问权限有关的辅助数据。这里不对其进行解释，但是可以阅读[SUS2002]或者所使用的系统文档。

下面是用recvmsg和sendmsg对前一节中run_server函数的重写：

```
static void run_server(void)
{
    int fd_skt;
    ssize_t nrecv;
    char msg[MSG_SIZE];
```

```
struct sockaddr_storage sa;
struct msghdr m;
struct iovec v;

ec_negl( fd_skt = socket(AF_UNIX, SOCK_DGRAM, 0) )
ec_negl( bind(fd_skt, (struct sockaddr *)&sa_server, sizeof(sa_server)) );
while (true) {
    memset(&m, 0, sizeof(m));
    m.msg_name = &sa;
    m.msg_namelen = sizeof(sa);
    v.iov_base = msg;
    v.iov_len = sizeof(msg);
    m.msg_iov = &v;
    m.msg_iovlen = 1;
    ec_negl( nrcv = recvmsg(fd_skt, &m, 0) )
    if (nrcv != sizeof(msg)) {
        printf("server got short message\n");
        break;
    }
    ((char *)m.msg_iov->iov_base)[0] = 'm';
    ec_negl( sendmsg(fd_skt, &m, 0) )
}
ec_negl( close(fd_skt) )
exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

额外的工作就是要设置msghdr结构，但是recvmsg和sendmsg调用非常简单。想象一下应用程序要向不同的地址发送大量报文，就可以看出将地址放在同样的结构中作为报文数据的一部分好处了。

8.6.4 采用无连接套接字的连接

通常客户端使用connect（见8.1.2节）来连接服务器套接字，但是利用无连接套接字为不带套接字地址参数的send和recv建立默认地址，也是可行的。这两个系统调用的详细讲解在8.9.1节中。

使用NULL套接字地址参数来调用connect就能删除默认地址。

8.7 带外数据

偶而要在套接字连接上要发送一个紧急的消息。如果它是内嵌式发送，那么只有等到先于它的消息被处理后才能被读取，因而可以通过发送带外消息的方法来解决。接收者也必须能接收带外消息。

为了发送带外消息，要用sendto、sendmsg或send来设置MSG_OOB标志。

接收者通过称为文件描述符的“error”设置作为第4个参数传递给select（8.1.3节）来发现带外消息。然后通过从recvfrom、recvmsg或recv的一个调用中设置MSG_OOB标志来获得消息。或者通过在套接字文件描述符上设置O_NONBLOCK标志和为接收调用设置MSG_OOB标志，从而调用一个接收函数，以在要接收一般数据时，每次都检查带外消息。

如果设置了SO_OOBINLINE选项(8.3节),带外数据可以内置于其他数据中。协议可以放在带有带外标志的带外数据之前,当读数据时,忽略带外标志,但可以通过一个调用检测到该标志的存在,这个调用只有这个功能:

socketmark —— 测试带外标志是否存在

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

int socketmark(
    int socket_fd      /* socket file descriptor */
);
/* Returns 1 if at mark, 0 if not, or -1 on error (sets errno) */
```

socketmark的问题是,如果因为套接字为空从而使其返回值是0,而带外数据可能会在对socketmark的调用之后但在下一次接收操作之前到达。当接收数据时,由于忽略了带外标志,所以会丢失带外标志。避免这种情况的方法就是在调用socketmark之前,先准备好接收数据,例如用select调用。这样socketmark的返回值0就明确意味着有数据,但带外标志不会先于接收的数据。

当带外数据到达时,也可以安排获得SIGURG信号。为了达到这个目的,调用带有F_SETOWN操作的fcntl来设置可以获得信号的进程或进程组。更多细节,参见[SUS2002]或系统文档。

8.8 网络数据库函数

这些网络数据库函数提供了有关主机、网络、协议和服务的信息。本书已在8.2.6节介绍了此类函数中最重要的函数getaddrinfo。另外一个常用函数是8.2.7节中介绍的gethostname。现在按这些函数所处理的信息类别对其进行分类讲解。

8.8.1 主机函数

扫描主机数据库的函数有三个:

sethostent —— 开始主机数据库扫描

```
#include <netdb.h>

void sethostent(
    int stayopen      /* leave connection open? */
);
```

gethostent —— 得到下一个主机数据库入口

```
#include <netdb.h>

struct hostent *gethostent(void);
/* Returns next entry or NULL on end of database (errno not defined) */
```

endhostent —— 结束主机数据库扫描

```
#include <netdb.h>

void endhostent(void);
```

struct hostent——主机数据库函数的结构

```

struct hostent {
    char *h_name;           /* official host name */
    char **h_aliases;      /* array of alternative host names */
    int h_addrtype;        /* address family (not type) */
    int h_length;          /* length of each address */
    char **h_addr_list;    /* array of pointers to network addresses */
};

```

通过循环调用`gethostent`可以扫描到所有已知的主机名。用`sethostent`开始扫描，并设置参数来指示所连的主机是否仍保持着打开状态，或者指示在每次调用它时`gethostent`是否都应该打开和关闭它。最后调用`endhostent`。这看起来发生了许多事件，实际上所有这些函数可能只是扫描本机的`/etc/hosts`文件。

在`hostent`结构中有两个数组，都是以`NULL`指针结束的。一个是主机别名的数组`h_aliases`，另一个是网络地址指针数组`h_addr_list`。对于`AF_INET`来说，这些地址是以32位二进制网络字节序存储的IP地址。因此可以使用`ntohl`（8.1.4节）将它们转换成本地字节序，或用`inet_ntoa`（8.2.3节）或`inet_ntop`（8.9.5节）将它们转换成点串。

示例：

```

static void hostdb(void)
{
    struct hostent *h;

    sethostent(true);
    while ((h = gethostent()) != NULL)
        display_hostent(h);
    endhostent();
}

static void display_hostent(struct hostent *h)
{
    int i;

    printf("name: %s; type: %d; len: %d\n", h->h_name, h->h_addrtype,
           h->h_length);
    for (i = 0; h->h_aliases[i] != NULL; i++)
        printf("\t%s\n", h->h_aliases[i]);
    if (h->h_addrtype == AF_INET) {
        for (i = 0; h->h_addr_list[i] != NULL; i++)
            printf("\t%s\n",
                   inet_ntoa(*(struct in_addr *)h->h_addr_list[i]));
    }
}

```

一个称为`hostdb`的主程序（这里没有给出）的输出为：

```

name: localhost; type: 2; len: 4
    127.0.0.1
name: sol; type: 2; len: 4
    loghost
    192.168.0.10
name: bsd; type: 2; len: 4
    192.168.0.15
name: suse2; type: 2; len: 4
    suse2.MSHOME
    192.168.0.19

```

作为比较，在同一个系统中的文件`/etc/hosts`的内容如下：

```
127.0.0.1      localhost
192.168.0.10  sol      loghost
192.168.0.15  bsd
192.168.0.19  suse2    suse2.MSHOME
```

为了通过名字查询主机，可以调用`gethostbyname`。不仅可以通过`/etc/hosts`文件得到主机名，在有访问权限的情况下也可以用DNS来得到主机名。实际上在`getaddrinfo`出现以前，这是得到网络地址的基本方法，它比`getaddrinfo`用得更多，因为它是一个老方法，几乎每个人都会用它做事。但与`getaddrinfo`不同，用`gethostbyname`所得到的是IP地址，同时还必须自己建立`sockataddr_in`结构，就像8.2.3节所做的那样。另外`gethostbyname`不能处理IPv6地址。下面是对照表：

gethostbyname —— 通过名字查询主机

```
#include <netdb.h>

struct hostent *gethostbyname(
    const char *nodename, /* node name */
);
/* Returns pointer to hostent or NULL on error (sets h_errno) */
```

对照表的末行注释并没有印错，这个函数设置的确实是`h_errno`，不是`errno`，它用的代码也不是`error`代码。（因为本书没有使用这个函数，所以就没了`h_errno`编写“`ec`”宏。）

示例（`display_hostent`来自前一个示例）：

```
static void gethostbyname_ex(void)
{
    struct hostent *h;

    if ((h = gethostbyname("www.yahoo.com")) == NULL) {
        if (h_errno == HOST_NOT_FOUND)
            printf("host not found\n");
        else
            printf("h_errno = %d\n", h_errno);
    }
    else
        display_hostent(h);
}
```

下面是调用该函数所得到的输出：

```
name: www.yahoo.akadns.net; type: 2; len: 4
www.yahoo.com
66.218.71.95
66.218.70.49
66.218.71.88
66.218.71.81
66.218.71.86
66.218.71.92
66.218.71.94
66.218.71.89
66.218.70.48
66.218.71.93
66.218.70.50
66.218.71.87
66.218.71.84
```

你可能想要与8.2.6节所给的输出作一比较，那里利用`getaddrinfo`做了同样的事。



与gethostbyname相对的是gethostbyaddr，它利用主机数据库（或许是DNS）把IP地址转换成了名字：

gethostbyaddr——通过地址查询主机

```
#include <netdb.h>

struct hostent *gethostbyaddr(
    const void *addr,      /* IP address */
    socklen_t len,        /* length of address */
    int family            /* family (called "type" in SUS) */
);
/* Returns pointer to hostent or NULL on error (sets h_errno) */
```

示例（inet见8.2.3节）：

```
static void gethostbyaddr_ex(void)
{
    struct hostent *h;
    in_addr_t a;

    ec_negl( a = inet_addr("66.218.71.94") )
    if ((h = gethostbyaddr(&a, sizeof(a), AF_INET)) == NULL) {
        if (h_errno == HOST_NOT_FOUND)
            printf("address not found\n");
        else
            printf("h_errno = %d\n", h_errno);
    }
    else
        display_hostent(h);
    return;
}
EC_CLEANUP_BGN
    EC_FLUSH("gethostbyaddr_ex")
EC_CLEANUP_END
}
```

输出：

```
name: w15.www.scd.yahoo.com; type: 2; len: 4
66.218.71.94
```

当得到www.yahoo.com的IP地址时，列表中显示的是66.218.71.94，但当请求IP的名字时，得到的是w15.www.scd.yahoo.com。这是因为DNS起了作用，显然Yahoo有许多服务器，当然它也必须那么多。

gethostbyaddr和gethostbyname一样过时了。相应于getaddrinfo（8.2.6节）的新函数是getnameinfo：

getnameinfo——得到名字信息

```
#include <sys/socket.h>
#include <netdb.h>

int getnameinfo(
    const struct sockaddr *sa,      /* socket address */
    socklen_t sa_len,             /* socket-address length */
    char *nodename,               /* node name */
    socklen_t nodelen,           /* node-name buffer length */
    char *servname,              /* service name */
    socklen_t servlen,           /* service-name buffer length */
    unsigned flags                /* flags */
);
/* Returns 0 on success or error number on error */
```

`getnameinfo`取整个套接字地址作参数，不仅只有IP地址，因为它还可能会与许多不同的族一起工作。最重要的是能与IPv6地址一起工作，而`gethostbyname`就不行了。答案在于两个缓冲区：一个是存节点（或主机）名的`nodename`，长度为`nodelen`；一个是存服务器名的`servname`，长度为`servlen`。如果缓冲区指针为NULL或它的长度为0，那么就得不到返回信息。

和`getaddrinfo`一样，`getnameinfo`返回的错误代码不是`errno`代码，而是特别的“EAI”码，如果需要，可以在其上使用函数`gai_strerror`（8.2.6节），或利用只为这两个函数提供的`ec_ai`宏。

可以用各种标志在一起进行或操作来控制`getnameinfo`的返回：

`NI_NOFQDN` 返回的只是本地机器的节点名，而不是完全意义的域名。

`NI_NUMERICHOST` 返回的是地址的数字形式（点分法表示IPv4，冒号记法表示IPv6）而不是名字。

`NI_NAMEREQD` 如果没有找到主机名就返回错误。在这种情况下通常返回数值形式。

`NI_NUMERICSERV` 返回端口号（字符串型）而不是服务名。

`NI_DGRAM` 查找SOCK_DGRAM (UDP)服务。通常查找SOCK_STREAM (TCP) 服务。

示例：

```
static void getnameinfo_ex(void)
{
    struct sockaddr_in sa;
    char nodename[200], servname[200];

    sa.sin_family = AF_INET;
    sa.sin_port = htons(80);
    sa.sin_addr.s_addr = inet_addr("216.109.125.70");
    ec_ai( getnameinfo((struct sockaddr *)&sa, sizeof(sa), nodename,
        sizeof(nodename), servname, sizeof(servname), 0) )
    printf("node: %s; service: %s\n", nodename, servname);
    return;
}

EC_CLEANUP_BGN
    EC_FLUSH("getnameinfo_ex")
EC_CLEANUP_END
}
```

输出：

```
node: w17.www.dcn.yahoo.com; service: http
```

另一个面向主机的函数是：

gethostid —— 得到本地主机标识符

```
#include <unistd.h>

long gethostid(void);
/* Returns identifier (no error return) */
```

`gethostid`看起来可以返回本地机器IP，但它没有必要那样做。要求它做的就是返回一个唯一的标识号，是否起作用依赖于在系统启动时是否设置了这个标识号。这里提及以免偶而会与其他有用的函数混淆。

还有一个提供识别系统信息的函数，与网络没有直接的联系：

uname——得到关于当前系统的信息

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

int uname(
    struct utsname *info    /* returned info */
);
/* Returns non-negative value on success or -1 on error (sets errno) */
```

struct utsname——uname的结构

```
struct utsname {
    char sysname[];          /* OS name */
    char nodename[];        /* node name within network */
    char release[];         /* release number (as string) */
    char version[];         /* version number (as string) */
    char machine[];         /* hardware type or computer model */
};
```

不幸的是，utsname结构^①的成员没有一个被标准化，因此不能用它们来控制应用程序的运行。也就是说用测试机器序列号来看看是否运行在Intel CPU上可能不错；但运行在CPU上的每个系统都把字符串格式化了它需要的形式，因此“Intel”或“x86”字符串可能并不在那里。你所能做的就是使用字符串进行显示，大概也可以标注性能测试输出。自然，uname系统调用是uname命令的本质。示例（没有选项）：

```
int main(void)
{
    struct utsname info;

    ec_negl( uname(&info) )
    printf("sysname = %s\n", info.sysname);
    printf("nodename = %s\n", info.nodename);
    printf("release = %s\n", info.release);
    printf("version = %s\n", info.version);
    printf("machine = %s\n", info.machine);
    exit(EXIT_SUCCESS);

    EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
    EC_CLEANUP_END
}
```

下面是我们的4种测试系统上运行的结果；^②也许你能认出其中的一种形式：

```
sysname = SunOS
nodename = sol
release = 5.8
version = Generic_108529-13
machine = i86pc

sysname = Darwin
nodename = Marc-Rochkinds-Computer.local
release = 6.6
version = Darwin Kernel Version 6.6: Thu May 1 21:48:54 PDT 2003;
root:xnu/xnu-344.34.obj-1/RELEASE_PPC
machine = Power Macintosh
```

① 对照表中所示括号不是合法的C语言，只是一种思想。在实现时要分配所需空间。

② 输出换行以适合页边界。

```

sysname = FreeBSD
nodename = bsd.MSHOME
release = 4.6-RELEASE
version = FreeBSD 4.6-RELEASE #0: Tue Jun
machine = i386

```

```

sysname = Linux
nodename = suse2
release = 2.4.18-4GB
version = #1 Wed Mar 27 13:57:05 UTC 2002
machine = i686

```

8.8.2 网络函数

本类函数能得到可能连接到本地机器的网络信息。先讲三个像扫描网络主机那样的函数(例如: `gethostent`):

setnetent —— 开始网络数据库扫描

```

#include <netdb.h>

void setnetent(
    int stayopen           /* leave connection open? */
);

```

getnetent —— 得到网络数据库入口

```

#include <netdb.h>

struct netent *getnetent(void);
/* Returns pointer to netent or NULL on end (errno not defined) */

```

endnetent —— 结束网络数据库扫描

```

#include <netdb.h>

void endnetent(void);

```

struct netent —— 网络数据库函数的结构

```

struct netent {
    char *n_name;           /* official network name */
    char **n_aliases;      /* array of alternative network names */
    int n_addrtype;        /* address family (not type) */
    uint32_t n_net;        /* network number */
};

```

示例:

```

static void netdb(void)
{
    struct netent *n;

    setnetent(true);
    while ((n = getnetent()) != NULL)
        display_netent(n);
    endnetent();
}

static void display_netent(struct netent *n)
{
    int i;

```

```

printf("name: %s; type: %d; number: %lu\n", n->n_name, n->n_addrtype,
(unsigned long)n->n_net);
for (i = 0; n->n_aliases[i] != NULL; i++)
    printf("\t%s\n", n->n_aliases[i]);
}

```

输出为:

```

name: loopback; type: 2; number: 127
name: arpanet; type: 2; number: 10
    arpa

```

大多数（但并非所有的）机器都有用于测试的loopback网络。运行上述示例的这台机器（运行在Solaris系统上）也有一个与Internet的连接，在历史上Internet称为“arpanet”网。记住，所显示的“type”实际上是族，它的输出说明宏AF_INET被定义成了2。

与主机数据库相似，有些函数能通过网络号或名字找到入口指针：

getnetbyname —— 通过名字查询网络

```

#include <netdb.h>

struct netent *getnetbyname(
    const char *name /* network name (to match n_name member) */
);
/* Returns pointer to netent or NULL if not found (errno not defined) */

```

getnetbyaddr —— 通过网络号查询网络

```

#include <netdb.h>

struct netent *getnetbyaddr(
    uint32_t net, /* network number (to match n_net member) */
    int type /* family (to match n_addrtype member) */
);
/* Returns pointer to netent or NULL if not found (errno not defined) */

```

8.8.3 协议函数

在形式上，扫描协议数据库的函数如下：

setprotoent —— 开始协议数据库扫描

```

#include <netdb.h>

void setprotoent(
    int stayopen /* leave connection open? */
);

```

getprotoent —— 得到协议数据库入口

```

#include <netdb.h>

struct protoent *getprotoent(void);
/* Returns pointer to protoent or NULL on end (errno not defined) */

```

endprotoent —— 结束协议数据库扫描

```

#include <netdb.h>

void endprotoent(void);

```

struct protoent —— 协议数据库函数的结构

```

struct protoent {
    char *p_name;           /* official protocol name */
    char **p_aliases;      /* array of alternative protocol names */
    int p_proto;           /* protocol number */
};

```

在一些系统上，当调用 `setsockopt` 或 `getsockopt` 时，可以用协议号表示协议层，但这并不标准。

以下程序很明显：

```

static void protodb(void)
{
    struct protoent *p;

    setprotoent(true);
    while ((p = getprotoent()) != NULL)
        display_protoent(p);
    endprotoent();
}

static void display_protoent(struct protoent *p)
{
    int i;

    printf("name: %s; number: %d\n", p->p_name, p->p_proto);
    for (i = 0; p->p_aliases[i] != NULL; i++)
        printf("\t%s\n", p->p_aliases[i]);
}

```

但输出很有趣，我在 SuSE Linux 上得到了 135 个协议，以下是部分输出：

```

...
name: mobile; number: 55
    MOBILE
name: tlsp; number: 56
    TLSP
name: skip; number: 57
    SKIP
name: ipv6-icmp; number: 58
    IPv6-ICMP
    ICMPV6
    icmpv6
    icmp6
name: ipv6-nonxt; number: 59
    IPv6-NoNxt
name: ipv6-opts; number: 60
    IPv6-Opts
name: cftp; number: 62
    CFTP
name: sat-expak; number: 64
    SAT-EXPAK
name: kryptolan; number: 65
    KRYPTOLAN
...

```



有两个更有预测性的函数来显示协议集:

getprotobyname——通过名字查询协议

```
#include <netdb.h>

struct protoent *getprotobyname(
    const char *name        /* protocol name */
);
/* Returns pointer to protoent or NULL if not found (errno not defined) */
```

getprotobynumber——通过协议号查询协议

```
#include <netdb.h>

struct protoent *getprotobynumber(
    int proto              /* protocol number */
);
/* Returns pointer to protoent or NULL if not found (errno not defined) */
```

8.8.4 服务函数

有多组通过名字和数字来搜索服务的函数，它们扫描本地系统中的/etc/services文件。

setservent——开始服务数据库扫描

```
#include <netdb.h>

void setservent(
    int stayopen          /* leave connection open? */
);
```

getservent——得到服务数据库入口

```
#include <netdb.h>

struct servent *getservent(void);
/* Returns pointer to servent or NULL on end (errno not defined) */
```

endservent——结束服务数据库扫描

```
#include <netdb.h>

void endservent(void);
```

struct servent——服务数据库函数的结构

```
struct servent {
    char *s_name;          /* official service name */
    char **s_aliases;     /* array of alternative service names */
    int s_port;           /* port number */
    char *s_proto;        /* name of protocol for this service */
};
```

getservbyname——通过名字查询服务

```
#include <netdb.h>

struct servent *getservbyname(
    const char *name,     /* service name */
    const char *proto     /* protocol name */
);
/* Returns pointer to protoent or NULL if not found (errno not defined) */
```

getservbyport——通过端口查询服务

```
#include <netdb.h>

struct servent *getservbyport(
    int port,          /* port */
    const char *proto /* protocol name */
);
/* Returns pointer to protoent or NULL if not found (errno not defined) */
```

最后一个此类示例是:

```
static void servdb(void)
{
    struct servent *s;

    setservent(true);
    while ((s = getservent()) != NULL)
        display_servent(s);
    endservent();
}

static void display_servent(struct servent *s)
{
    int i;

    printf("name: %s; port: %d; protocol: %s\n", s->s_name, ntohs(s->s_port),
        s->s_proto);
    for (i = 0; s->s_aliases[i] != NULL; i++)
        printf("\t%s\n", s->s_aliases[i]);
}
}
```

由于/etc/services文件很大，所以输出也很大，以下是一部分:

```
...
name: ftp-data; port: 20; protocol: tcp
name: ftp-data; port: 20; protocol: udp
name: ftp; port: 21 ; protocol: tcp
name: ftp; port: 21 ; protocol: udp
name: ssh; port: 22 ; protocol: tcp
name: ssh; port: 22 ; protocol: udp
name: telnet; port: 23 ; protocol: tcp
name: telnet; port: 23 ; protocol: udp
name: smtp; port: 25 ; protocol: tcp
    mail
name: smtp; port: 25 ; protocol: udp
    mail
name: nsw-fe; port: 27 ; protocol: tcp
name: nsw-fe; port: 27 ; protocol: udp
name: msg-icp; port: 29 ; protocol: tcp
name: msg-icp; port: 29 ; protocol: udp
name: msg-auth; port: 31 ; protocol: tcp
name: msg-auth; port: 31 ; protocol: udp
...
```

8.8.5 网络接口函数

有一组函数可以检索网络接口名字和它们的索引号。首先有一个函数把它们取到 `if_nameindex` 结构数组中，并有相应的函数释放数组:

if_nameindex ——得到所有的网络接口名字和索引

```
#include <net/if.h>

struct if_nameindex *if_nameindex(void);
/* Returns array or NULL on error (sets errno) */
```

if_freenameindex ——通过if_nameindex释放分配的数组

```
#include <net/if.h>

void if_freenameindex(
    struct if_nameindex *ptr /* pointer to array */
);
```

struct if_nameindex ——网络接口函数的结构

```
struct if_nameindex {
    unsigned if_index; /* interface index */
    char *if_name; /* interface name */
};
```

下面是一个显示索引和接口的函数:

```
static void ifdb(void)
{
    struct if_nameindex *ni;
    int i;

    ec_null( ni = if_nameindex() )
    for ( i = 0; ni[i].if_index != 0 || ni[i].if_name != NULL; i++)
        printf("index: %d; name: %s\n", ni[i].if_index, ni[i].if_name);
    if_freenameindex(ni);
    return;

    EC_CLEANUP_BGN
        EC_FLUSH("ifdb")
    EC_CLEANUP_END
}
```

在Solaris系统上的输出为:

```
index: 1; name: lo0
index: 2; name: iprb0
```

在SuSE Linux上为:

```
index: 1; name: lo
index: 2; name: eth0
```

通过8.8.2节的函数来显示匹配的网络名有两种: 一个是loopback接口, 一个是使用以太网的Internet接口。

另有一个把名字映射为索引的函数:

if_nametoindex ——把网络接口名字映射成索引

```
#include <net/if.h>

unsigned if_nametoindex(
    const char *ifname /* interface name */
);
/* Returns index or 0 on error (errno not defined) */
```

注意：如果名字没找到，那么返回的是0而不是-1。

把索引映射成名字，可以调用：

```
if_indextoname —— 把网络接口索引映射成名字

#include <net/if.h>

char *if_indextoname(
    unsigned ifindex,      /* interface index */
    char *ifname          /* interface name */
);
/* Returns name or NULL on error (sets errno) */
```

ifname参数必须是至少IF_NAMESIZE字节的缓冲，其中要包括终止符NUL的字节空间。并返回指向缓冲的指针。

8.9 其他系统调用

本节描述几个不适合归入前几节的网络系统调用。

8.9.1 send和recv

这两个函数除了允许定义标志外，其他行为很像write和read。因为它们没有套接字地址参数，所以它们通常使用有连接的套接字。至于无连接的套接字，sendto、sendmsg、recvfrom和recvmsg更合适。

send —— 向套接字发送数据

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

ssize_t send(
    int socket_fd,        /* socket file descriptor */
    const void *data,     /* data to send */
    size_t length,       /* length of data */
    int flags             /* flags */
);
/* Returns number of bytes sent or -1 on error (sets errno) */
```

recv —— 从套接字接收数据

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

ssize_t recv(
    int socket_fd,        /* socket file descriptor */
    void *buffer,        /* buffer to receive data */
    size_t length,       /* length of buffer */
    int flags             /* flags */
);
/* Returns number of bytes received, 0, or -1 on error (sets errno) */
```

send和recv都能使用MSG_OOB标志；这已在8.7节解释过了。另外，对于recv可以限定MSG_PEEK和/或MSG_WAITALL，8.6.2节中解释过这两个标志。能用带MSG_WAITALL的recv来替代8.5节SMI实现中的调用readall。

8.9.2 getsockname和getpeername

getsockname能够得到与前一个调用绑定的套接字地址：

getsockname——得到套接字地址

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

int getsockname(
    int socket_fd,          /* socket file descriptor */
    struct sockaddr *sa,    /* socket address */
    socklen_t *sa_len      /* address length */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

与通常的套接字地址返回函数一样，在输入上，sa应该指向一个足够大的缓冲区以容纳套接字地址，sa_len为缓冲区大小。在输出上，sa_len为实际的地址大小。

能获得相连套接字的套接字地址的一个相似的函数为：

getpeername——得到已连接套接字的套接字地址

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

int getpeername(
    int socket_fd,          /* socket file descriptor */
    struct sockaddr *sa,    /* socket address */
    socklen_t *sa_len      /* address length */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

通常，如果有连接的两个套接字未被绑定，那么getpeername的返回就是未定义的，在这种情况下不会得到错误返回。

8.9.3 socketpair

AF_UNIX套接字的行为有点像双向的FIFO，当用FIFO时，每个要通信的进程都会建立自己的套接字。相反，pipe系统调用会返回子进程能继承的两个文件描述符。socketpair系统调用融合了这两种方法，一次调用能获得两个套接字文件描述符：

socketpair——建立套接字对

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

int socketpair(
    int domain,             /* domain (AF_UNIX, AF_INET, etc.) */
    int type,               /* SOCK_STREAM, SOCK_DGRAM, etc. */
    int protocol            /* specific to type; usually zero */
    int socket_vector[2]   /* returned socket file descriptors */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

前三个参数很像socket的三个参数。处理什么样的domain和type归因于实现，它所支持的主要是AF_UNIX和SOCK_STREAM。

8.9.4 shutdown

当关闭(close)仍有数据传输的套接字时，系统会尝试着接收或发送数据直到放弃尝试并扔掉数据。在关闭套接字之前，可以通过调用shutdown来指明不要尝试收发数据：

shutdown——关闭套接字发送和/或接收操作

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

int shutdown(
    int socket_fd,          /* socket file descriptor */
    int how                 /* SHUT_RD, SHUT_WR, or SHUT_RDWR */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

SHUT_RD指示不要再接收，SHUT_WR指示不要发送，SHUT_RDWR指示两者都不要。

8.9.5 inet_ntop和inet_pton

在8.2.3节曾讨论过inet_ntoa和inet_addr函数在二进制IPv4地址与点串之间的转换。下面要讲的inet_ntop和inet_pton更巧妙：它们既能转换IPv4地址，也能转换IPv6地址。

inet_ntop——把IPv4或IPv6二进制地址转换成字符串

```
#include <arpa/inet.h>

const char *inet_ntop(
    int domain,           /* AF_INET or AF_INET6 */
    const void *src,     /* pointer to binary address (input) */
    char *dst,           /* string (output) */
    socklen_t dst_len    /* length of dst buffer */
);
/* Returns string or NULL on error (sets errno) */
```

inet_pton——把IPv4或IPv6字符串地址转换成二进制

```
#include <arpa/inet.h>

int inet_pton(
    int domain,           /* AF_INET or AF_INET6 */
    const char *src,     /* string (input) */
    void *dst            /* buffer for binary address (output) */
);
/* Returns 1 on success, 0 on bad string, or -1 on error (sets errno) */
```

inet_ntop中的src指向的是一个二进制地址：对IPv4来说是32位数字，对IPv6来说是16字节的数组。输出结果放在了由dst指向的缓冲区。dst的大小由两个宏来指定：IPv4使用INET_ADDRSTRLEN宏，IPv6使用INET6_ADDRSTRLEN宏。

对于inet_pton而言，输入是由点分表示法或冒号表示法表示的字符串，二进制的输出结果放入到由dst指向的缓冲区，输出缓冲区最好足够大，因为它没有长度参数。IPv4地址需要32位，IPv6需要128位（16字节）。

示例：

```
static void cvt(void)
{
    char ipv6[16], ipv6str[INET6_ADDRSTRLEN], ipv4str[INET_ADDRSTRLEN];
    uint32_t ipv4;
    int r;
    ec_neg1( r = inet_pton(AF_INET, "66.218.71.94", &ipv4) )
    if (r == 0)
        printf("Can't convert\n");
    else {
        ec_null( inet_ntop(AF_INET, &ipv4, ipv4str, sizeof(ipv4str)) )
        printf("%s\n", ipv4str);
    }
}
```

```

ec_neg1( r = inet_pton(AF_INET6,
    "FEDC:BA98:7654:3210:FEDC:BA98:7654:3210", &ipv6) )
if (r == 0)
    printf("Can't convert\n");
else {
    ec_null( inet_ntop(AF_INET6, &ipv6, ipv6str, sizeof(ipv6str)) )
    printf("%s\n", ipv6str);
}
return;

EC_CLEANUP_BGN
    EC_FLUSH("cvt")
EC_CLEANUP_END
}

```

调用该函数得到的输出为:

```

66.218.71.94
fedc:ba98:7654:3210:fedc:ba98:7654:3210

```

8.10 高性能方面的考虑

在8.4.4节中给出的Web服务器的例子，利用了8.1.3节中的多客户端的方法。现在假设我们的服务器一次要处理1000个客户端或10 000个客户端，它还能正常工作吗？

通过一些简单的例子可以列出所谓的“C10K”（“10 000个客户端”）问题：

- `select`必须查看10 001个文件描述符。这可能比`fd_set`的容量还大。即使它能容纳这个数，`select`也要花很长时间来处理这些文件描述符。
- 一个进程能打开的文件描述符数量是有限的，通常比10 001小得多。
- 为了提供不错的响应，需要使多进程或多线程能处理所有的工作，但这些资源的处理能力也是有限的。
- 将数据从文件复制到套接字（即进程内存的输入和输出）非常耗时，它是10 000个客户端的瓶颈。
- 服务器可能会使用许多内部表格和其他资源，从而使空间匮乏或使运行速度变慢，或者两者兼而有之。

可以看出问题相当严重，这就是为什么实时Web服务器（或者其他大容量的服务器）是一种非常复杂的软件的一个原因。

如果你要处理10 000或500个客户端，那么有一篇不错的名为《The C10K Problem》的文章可参阅，它讲述了这个问题，并探讨了可能的解决方案[Keg2003]。

练习

- 8.1 修改8.1.3节的多客户端程序，使它能在两台计算机上工作（`AF_INET`），让服务器程序和客户端程序运行在不同的机器上。
- 8.2 利用[RFC1288]中的信息，实现一个简单的`finger`命令，除`user@host`参数以外没有其他选项。并利用有连接的套接字（`SOCK_STREAM`）。在下一个练习中可能需要一个设置端口号的选项。
- 8.3 同样利用[RFC1288]，实现一个简单的`finger`服务器，但要使用没有用过的高端口号（例如3079），不要用标准端口79。利用有连接的套接字。用前一个练习写的`finger`命令测试这个服务器。如果你有一台可用于测试的机器，那么你可以把服务器安装在端口79上，并在另一台机器上用标准`finger`命令测试它。

- 8.4 与练习8.2相同，但要利用无连接的套接字来实现。
- 8.5 与练习8.3相同，但要利用无连接的套接字来实现。
- 8.6 利用RFC 2549来设计和实现一个简单的通信程序。要保证程序的协调和服务质量水平。
- 8.7 利用Qt这样的GUI工具包实现一个图形界面的Web浏览器。注意：这的确很难！可能要花一学期的时间才能完成。提示：用处理嵌套表格的方法来设计程序。如果能完成这个练习，其他事就相对容易了。
- 8.8 利用无连接套接字实现SMI，不要用8.5节的有连接套接字。运行一些7.5节那样的计时测试来测试一下程序。
- 8.9 实现一个Web网页爬行程序，它从一些URL地址开始，在其上找到其他的URL地址（例如通过在其上查找以“href”开始的字符串），把所找到的URL地址添加到列表中，再通过扫描这些URL找到更多的地址。当找到特定数量的URL地址或当它被中断时停止。（显然，当遇到错误时需要能够继续。）要有一种方法显示结果，至少显示所有找到的URL地址和它们的扫描是否成功（成功时HTTP状态为200），如果不成功，要显示出原因是什么。因为不能爬行已查找过的网页，所以必须实现Robot不相容协议（Robot Exclusion Protocol）（详情见www.robotstxt.org）。要提供爬行每台主机（第一个斜线之前的URL部分）仅一次的“独一无二的主机”选项，可以使用爬行那台主机时第一个尝试的URL（例如，www.basepath.com/index.htm）。通过这种选项，尝试找到代表成功扫描到的唯一主机的最开始的URL（我认为从www.yahoo.com开始是一个好主意，因为所有的链接都和www.yahoo.com有关系。）当运行爬行程序时，确保不要独占网络资源，否则影响他人。
- 8.10 扩展练习5.14所写的程序，使它包括在本章解释过的附录A中进程属性。



第9章 信号和定时器

9.1 信号的基本概念

信号是指当事件发生时所发出的通知，诸如用户键入中断命令（通常使用Ctrl-c），产生浮点异常或者警报。通常，信号会被异步地传输给进程或线程，而不管进程或线程正在做什么都会被打断。信号可能会立刻结束进程，或按预先安排，运行一个指定的函数来捕捉信号。

9.1.1 信号简介

为说明程序如何处理信号，这里给出一个简单的例子。例如，某程序每3秒显示一个数字，但当中断信号出现时，它会显示一个消息并终止：

```
static void fcn(int signum),
{
    (void)write(STDOUT_FILENO, "Got signal\n", 11);
    _exit(EXIT_FAILURE);
}

int main(void)
{
    int i;
    struct sigaction act;

    memset(&act, 0, sizeof(act));
    act.sa_handler = fcn;
    ec_negl( sigaction(SIGINT, &act, NULL) )

    for (i = 1; ; i++) {
        sleep(3);
        printf("%d\n", i);
    }

    exit(EXIT_SUCCESS);

    EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
    EC_CLEANUP_END
}
```

调用sigaction时，设置了SIGINT信号的信号处理函数。当运行此程序时，只需在“2”出现后键入Ctrl-c键。无论什么程序正在运行什么（也许是在休眠，或者是在执行printf函数，也可能正处在循环中），都会被中断，并立刻调用fcn函数，该函数会显示一个消息，并调用_exit终止程序。（不使用exit的原因见9.1.7节。）

其屏幕输出为：

```
1
2
Got signal
```

如果没有安装该信号处理程序，按下Ctrl-c将会立即终止该进程。从技术上说，原因是

SIGINT信号的默认动作是终止进程。

也可以调用sigaction来忽略SIGINT信号:

```
int main(void)
{
    int i;
    struct sigaction act;

    memset(&act, 0, sizeof(act));
    act.sa_handler = SIG_IGN;
    ec_negl( sigaction(SIGINT, &act, NULL) )

    for (i = 1; ; i++) {
        sleep(3);
        printf("%d\n", i);
    }
    exit(EXIT_SUCCESS);

    EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
    EC_CLEANUP_END
}
```

这一次它不停地显示数字，按下Ctrl-c没有任何作用。我们最终通过按下Ctrl-\键终止了该进程，因为这会产生一个退出信号（SIGQUIT），其未经修改的默认动作是终止进程。

好了，基础知识就介绍到这里。实际上，信号是非常复杂的，因为：

- 存在大量不同的信号，有时产生这些信号的环境和它们的含义是复杂的。
- 为信号设定合适的动作可能会很复杂。
- 处理信号可能会很复杂。

下面请跟随我逐步深入该问题。读完本章后，就能明白所有这些问题。

9.1.2 信号的生命周期

信号生命周期的起点是，与其相关联的某个事件发生并产生出该信号，其终点是信号完成传递时，这意味着已经执行了某种专为其设置的动作。可能的动作有以下三种：

- 1) 默认动作（SIG_DFL）：终止、停止或继续执行进程，或忽略信号。
- 2) 忽略信号（SIG_IGN）。
- 3) 在完成信号传递后，通过运行信号处理程序来捕获信号。

无论和信号相关联的是什么用户或者什么系统事件，大多数信号都可以自然地产生。例如，被零除自然产生一个SIGFPE事件，终止一个子进程自然产生一个SIGCHLD事件。作为另一种选择，任何信号都可以通过以下5种系统调用人工产生：kill、killpg、pthread_kill、raise或sigqueue。在下一节中将列出所有这些信号，如果某信号能够自然产生，就标示出其自然起因。SIGKILL、SIGSTOP、SIGTERM、SIGUSR1和SIGUSR2 5种信号不存在自然起因，只能通过人工产生，通常由kill系统调用产生。

信号从产生后到传递前所处的状态叫做挂起。当某个信号挂起时，如果另一个同类信号到达，那么是否传递多个同类信号就要取决于具体实现了，在可移植编程中不能事先假定。有关本主题的更多内容见9.5.3节。

线程（见5.17节）可以通过对信号进行阻塞，使得挂起的信号继续保持挂起。[⊖]全部当前

⊖ 如果进程中只有一个线程，可能是因为它没有执行任何多线程的工作，那么本书关于线程的内容总体上也能用于这种进程。

被阻塞的信号的集合叫做信号屏蔽。本书9.1.5节中讲述的许多系统调用，可以用来创建和处理屏蔽，并使得某个特定屏蔽成为线程的有效信号屏蔽。

生成的信号既可以用于某个特定的线程，也可以用于整个进程。对于后者，如果有多个拥有该信号的线程都未阻塞，则信号传递到哪个线程是不确定的。在9.1.3节中，对在何种情况下哪些事件被发送到线程与进程，给出了更详细的信息。^①

虽然上文说过，信号屏蔽是针对每个线程的，但针对信号的动作是全进程范围的，尽管可能存在多个线程。

一般来说，当运行信号处理程序时，其处理的信号会被临时加入到那个线程的信号屏蔽中去，因此直到处理程序返回之前，将不会传递第二个同类信号。这样，就不必担心发生处理程序对同一信号的递归调用。然而，如果对几种不同类型的信号使用同一函数，还是可能会发生递归调用。

9.1.3 信号的类型

在[SUS2002]中定义了28种信号，其中大多数的实现定义较多并可以自行查找，在此不做描述。同样，有一些附加信号是实时信号扩展（9.5节）中的一部分。将SUS信号分类是很有用的。在下表中，圆括号中的字母标明了信号的默认动作，稍后对其进行解释。每个信号的自然起因已进行了标示；那5种只能人工产生的信号（上节中已作解释）已明确标出。请记住，所有具有自然起因的信号都可以人工产生。

• 错误检测:

SIGBUS——对未定义的存储器对象部分进行访问 (A)

SIGFPE——错误的算法操作 (A)

SIGILL——非法指令 (A)

SIGPIPE——对不存在读者的管道写入 (T)

SIGSEGV——无效的存储器引用 (A)

SIGSYS——失败的系统调用 (A)

SIGXCPU——超出CPU时间限制 (A)

SIGXFSZ——超出文件大小限制 (A)

• 用户/应用程序产生的:

SIGABRT——调用abort (A)

SIGHUP——挂断 (T)

SIGINT——中断 (来自键盘) (T)

SIGKILL——终止；只能人工产生 (T)

SIGQUIT——退出 (来自键盘) (A)

SIGTERM——终止；只能人工产生 (T)

SIGUSR1——用户信号1；只能人工产生 (T)

SIGUSR2——用户信号2；只能人工产生 (T)

• 任务控制:

SIGCHLD——子进程的终止或停止 (I)

^① 本章中关于线程的内容仅适用于POSIX线程的实现。一些在Linux和FreeBSD系统中广泛使用的“Linux线程”并不是POSIX线程的忠实实现，信号不能以标准的方式与其一起工作。然而，最新发布版本NPTL工作得很好。

SIGCONT—继续运行（来自键盘）(C)
 SIGSTOP—停止运行；只能人工产生(S)
 SIGTSTP—终止停止信号（来自键盘）(S)
 SIGTTIN—后台进程尝试读取(S)
 SIGTTOU—后台进程尝试写入(S)

• 定时器:

SIGALRM—闹钟超时(T)
 SIGVTALRM—虚拟定时器超时(T)
 SIGPROF—剖面定时器超时(T)

• 其他事件:

SIGPOLL—可查询事件(T)
 SIGTRAP—跟踪/断点陷门(A)
 SIGURG—socket可用的带外数据(I)

以下是圆括号中的字母（默认行为）的含义:

I 信号被忽略。

T 终止。

A 和T相同，但包含更多的必须实现的动作，如写入核心转储文件。

S 停止。

C 停止后继续。

自然产生的错误检测信号是程序错误的结果。对于SIGBUS、SIGFPE、SIGILL和SIGSEGV来说，错误产生的确切原因不是标准化的，通常是由硬件检测出来的错误。同样，当这4个信号自然产生时，它们服从一些特定的规则:

- 如果其设置被sigaction忽略，其行为是不定的。
- 信号捕获函数返回的结果是不定的。
- 在阻塞时其发生结果是不定的。

换一种说法，如果硬件检测出的错误是真实的，程序不能等闲视之。忽略它，在信号处理程序返回之后继续运行，或阻塞它以使动作延期，都是不安全的。应马上进行处理，并必须使用具有退出（或长跳转，见9.6节）功能的信号处理程序，而不是返回，或者执行默认动作立刻终止进程。

有两个由用户/应用程序产生的信号：SIGINT和SIGQUIT，它们通常和在4.5.7节中解释过的键盘控制序列相关联。SIGHUP通常是挂断终端设备产生的结果。SIGABRT是由abort系统调用（见9.1.9节）产生的，SIGTERM是kill命令的默认信号——这是终止任意进程的主要方法，例如当系统管理员需要关闭系统时。SIGUSR1和SIGUSR2不被任何系统调用所使用，它们仅能被应用程序所使用。

任务控制信号已经在4.3节中讨论过了。

SIGALRM将在9.7.1节中讨论。另外两种定时器信号在9.7.4节中讨论。

在其他事件信号中，SIGPOLL可以和STREAMS（见4.9节）一起使用；通过调用ioctl能使其有效。该信号通常并不会产生，因此不必担心它。SIGURG已在8.7节中解释过了。SIGTRAP是供调试程序使用的。

信号被传递后，就无从知晓它是自然产生的还是人工产生的了。如果某个错误检测信号是自然产生的，那么它会被送往违例线程；而其他信号被送往进程。人工产生的信号可以被送往一个线程或进程，这取决于使用哪个系统调用（见9.1.9节）来产生它。

那些在终止之前需要执行清理的程序，应当设置成具有捕获信号SIGHUP、SIGINT和SIGTERM的功能。在程序成形之前，应该一直保留着SIGQUIT，以便从键盘可以终止该程序（并进行核心转储）。对其他信号并不需要经常进行设置，通常它们被留作终止进程之用。但是，精心构造的程序会需要捕获所有能捕获的东西，为了执行清理，可能还需要记录错误，并输出详尽的错误信息。从心理学角度上来讲，形如“内部错误53：请与客户支持进行联络”的信息，要比shell提供的“已核心转储的总线错误”之类的信息更为人们所接受。在后面的9.1.8节中会给出该主题的更多信息及示例程序。

9.1.4 中断系统调用

对不能被忽视的信号进行传递可能会引起系统调用的中断。如果动作的结果是终止进程，那么原因可能有两个，或者因为这是信号的默认动作，或者因为信号处理程序终止了进程。正如上文例子显示的那样，中断的系统调用绝对不会再继续了。如果动作是停止进程，那么当进程继续时，运行也会恢复。

然而，如果动作是捕获信号并且信号处理程序也返回了，那么中断的系统通常不会被重新启动。相反，它通常会返回-1并将errno置为EINTR。在某些情况下，这正是你所需要的；例如，你可能精心地设置了某个定时器，以产生一个SIGALRM信号，用来在一段时间（如10秒钟）后中断某个等待的read。但在另外一些情况下，因为算法不允许调用被中断，这时中断系统调用会出问题。

要遵循的最简单的规则是：绝对不要从信号处理程序返回，除非已经对信号发生的上下文进行了仔细的控制。如果那样做不实际的话，那么就可以在为信号调用sigaction时设置SA_RESTART标志（见9.1.6节），这样系统调用不会被中断——相反，在信号处理程序返回时，它们会从中断处继续。

只有阻塞的系统调用可以被中断。在该上下文中，“阻塞”意味着调用正在等待某些不可预知来临时间的事件，如来自终端或socket的输入、进程的中止、消息的到达、信号量的发送等。那些仅仅需要花费时间的系统调用不会被阻塞，如读取文件或创建进程；虽然存在少量的延迟，但这些时间用在了处理或等待处理程序，而不是等待某些不可预知的事件。

并非所有阻塞的系统调用都可以被中断。例如pthread_mutex_lock，它甚至在信号到达及其处理程序返回后仍继续等待。确定某系统调用是否可以被中断的唯一途径，是阅读它的文档，最好是SUS中的文档。

9.1.5 管理信号屏蔽

正如select（见4.2.3节）使用fd_set一样，信号屏蔽拥有一组函数来处理不同的位：①

sigemptyset——初始化空信号集

```
#include <signal.h>

int sigemptyset(
    sigset_t *set          /* signal set */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

① 替代方法是只使用一个unsigned long，但在引入这些函数的时候，long在几乎所有的机器上都是32位的（long long还没被引入），现在认为32位太小了。

sigfillset —— 初始化整个信号集

```
#include <signal.h>

int sigfillset(
    sigset_t *set          /* signal set */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

sigaddset —— 把信号加入信号集

```
#include <signal.h>

int sigaddset(
    sigset_t *set,        /* signal set */
    int signum           /* signal */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

sigdelset —— 从信号集中删除信号

```
#include <signal.h>

int sigdelset(
    sigset_t *set,        /* signal set */
    int signum           /* signal */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

sigismember —— 在信号集中测试信号

```
#include <signal.h>

int sigismember(
    const sigset_t *set,  /* signal set */
    int signum           /* signal */
);
/* Returns 1 if member, 0 if not, or -1 on error (sets errno) */
```

设定sigset_t后，从sigemptyset或sigfillset开始，接着调用sigaddset或sigdelset来添加或删除成员。可以调用sigismember来测试某个信号是否是成员。

在同一时间一个线程只能有一个信号屏蔽，可以通过pthread_sigmask对其进行设置：

pthread_sigmask —— 改变线程的信号掩码

```
int pthread_sigmask(
    int how,              /* how signal mask is to be changed */
    const sigset_t *set,  /* input set */
    sigset_t *oset        /* previous signal mask */
);
/* Returns 0 on success, error number on failure (errno not set) */
```

输入参数set如何改变信号屏蔽，由how参数决定：

SIG_BLOCK 新信号屏蔽成为当前信号屏蔽和set的并集。

SIG_SETMASK 新信号屏蔽成为set，完全取代了当前的信号屏蔽。

SIG_UNBLOCK 新信号屏蔽成为当前信号屏蔽和set的补集的并集。

换句话说，SIG_BLOCK将set参数中的信号添加到了信号屏蔽中，SIG_UNBLOCK从信号屏蔽中删除了set参数中的信号，而SIG_SETMASK只是将信号屏蔽置成了set。

如果oset非NULL，则oset返回的是指向上一个信号屏蔽的指针。同样，set可以为NULL，这时信号屏蔽不被改变（无论how是什么），但会通过oset（如果oset非NULL）返回该信号屏蔽的指针；这是仅获取信号屏蔽而不改变它的一种方法。

不能阻塞SIGKILL或SIGSTOP信号，因为它们总是被发送给进程的，并且总是用于终止或停止进程的。（同样，它们不能被捕获或忽略。）

如果进程中只有一个线程，那么可以选择调用一个更老（在POSIX中引入线程之前就已经存在）的函数，除了使用errno代替返回错误代码外，其执行过程和pthread_sigmask完全一致：

sigprocmask —— 改变线程的信号掩码（仅单个线程）

```
#include <signal.h>

int sigprocmask(
    int how, /* how signal mask is to be changed */
    const sigset_t *set, /* input set */
    sigset_t *oset /* previous signal mask */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

通常，因为完全未使用线程的程序并没有链接到“pthread”库，因此pthread_sigmask是不可用的，对此没有其他办法，只能使用sigprocmask。[Ⓒ]

不要阻塞具有信号屏蔽的信号，因为需要进程来忽略它——这就是SIG_IGN动作存在的原因。相反，阻塞是一种临时的状态，用于保护部分代码不受信号到达的影响。上文已提到了这样一个例子：当信号处理程序运行时，引起处理程序被调用的信号会被自动临时性地加入到信号屏蔽中，并在（如果）函数返回时被删除。[Ⓒ]

另一个例子是，在应用程序启动完毕之前，有一个机会来设置它所关心的所有信号的动作。所需要进行的工作就是，调用sigfillset和pthread_sigmask（或sigprocmask），以获得临时的改变。

本章中还有其他一些说明信号屏蔽重要性的示例。

9.1.6 sigaction系统调用

当信号传递后，可通过sigaction系统调用来决定采取的动作。对每个要进行动作设置的信号，都要调用它：

sigaction —— 设置信号动作

```
#include <signal.h>

int sigaction(
    int signum, /* signal */
    const struct sigaction *act, /* new action */
    struct sigaction *oact /* old action */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

struct sigaction —— sigaction的结构

```
struct sigaction {
    void (*sa_handler)(int); /* SIG_DFL, SIG_IGN, or function pointer */
    sigset_t sa_mask; /* additional signals to be blocked */
    int sa_flags; /* flags */
    void (*sa_sigaction)(int, siginfo_t *, void *); /* Realtime Signal handler */
};
```

Ⓒ 在我使用的Solaris版本中没有pthread库，但也定义了pthread_sigmask，然而它不起任何作用。我不将其改为sigprocmask。

Ⓒ 如果使用longjmp从信号处理程序中跳出，信号屏蔽如何改变是不定的。所以应该使用siglongjmp（见9.6节）。

参数act指向为整个进程指定新动作的一个结构——它并不为单独的线程保留信号动作。如果oact不是NULL，其指针将指向返回的原动作。如果就是想得到原动作，可以将act置为NULL，这样就不会改变动作。本书的大多数示例都将oact置为了NULL，但在9.7.2节的一个例子中，它被用来保存原动作以作恢复之用。

不能改变SIGKILL的动作，它总是用来终止进程（而不仅仅是某个线程）；也不能改变SIGSTOP的动作，它总是用来停止进程（也不仅仅是线程）。

在sigaction结构中，sa_handler通过以下值中的一个来指定动作：

SIG_DFL 信号采用其默认动作，具体依信号而定。如9.1.3节中描述的那样，动作通常是忽略、终止、停止或继续。同样，它们总是应用于整个进程，从不应用到单个的线程。

SIG_IGN 忽略信号，因此其传递不起作用。当SIGCHLD信号置为SIG_IGN时也存在副作用，其中SIG_IGN的作用和SA_NOCLDWAIT标志相同；请见下文。令人感到不解的是，将SIGCHLD置为SIG_DFL并不会产生这种副作用，尽管该信号的默认动作是忽略它。

函数指向信号处理函数的指针；信号被其捕获。

信号处理程序和9.1.1节中的一个例子很相似：

```
static void fcn(int signum)
{
    (void)write(STDOUT_FILENO, "Got signal\n", 11);
    _exit(EXIT_FAILURE);
}
```

函数只需要有正确的原型，可以是静态（static）的，也可以不是。信号传递后，调用函数，其参数设置为信号的编号（例如，SIGINT、SIGUSR1）。9.1.3节中描述了不同的信号及其宏名称。既可以为每个信号编号设定一个单独的函数，也可以为全部信号只设定一个函数，或者介于上述二者之间。关于在信号处理程序中可以进行哪些工作，将在9.1.7节中详述。

如果支持实时信号选项，就可以设置SA_SIGINFO标志（见下文），然后在信号处理程序中使用sa_sigaction成员代替sa_handler成员，这会为信号处理程序提供更多的信息。这一特性将在9.5节中讨论。在实现中，sa_sigaction和sa_handler可能会使用同一个存储器，因此要确保只设置二者中的一个。设置一个然后将另一个置为零是错误的。

如上文所述，当处理程序运行时，引起处理程序被调用的信号会被阻塞，但可以通过使用9.1.8节中的信号屏蔽处理函数，在sa_mask参数中进行指定，以阻塞附加的信号。

当线程接收到被捕获的信号后，将在该线程中运行信号处理程序。同时，信号处理程序的运行中被临时修改的内容就是该线程的信号屏蔽。回顾可知，将捕获的信号传递给线程的方式有两种：一种是先传递给进程，然后用随机选择的方式传递给一个未被其阻塞的线程，另一种是传递给一个指定的线程。

下面是一个sa_flags成员的简单标志列表。注意，前两种只能用于SIGCHLD信号。

SA_NOCLDSTOP 当其子进程停止或继续时不产生SIGCHLD信号。

SA_NOCLDWAIT 不将已终止的子进程转换为僵尸进程，见5.9节中的解释。将SIGCHLD信号明确设置为SIG_IGN，具有同样的效果。

SA_NODEFER 除非信号明确包含在sa_mask成员中，否则不要将信号加入到信号处理程序入口的信号屏蔽中。该标志仅用于已废弃的signal函数（见9.4节）的实现。

SA_ONSTACK 如果另一个可选的信号堆栈已经通过sigaltstack进行声明，则将信号传递到该堆栈。见9.3节。

SA_RESETHAND 重设信号动作为SIG_DFL，并在进入信号处理程序时清除SA_SIGINFO标志。对SIGILL和SIGTRAP信号无效。同时，和SA_NODEFER标志的行为类

似，它仅用于允许signal函数的实现。

SA_RESTART 不允许信号中断系统调用；见9.1.4节。在9.7.4节中有一个相关示例。

SA_SIGINFO 使用sa_sigaction成员代替sa_handler成员；见9.5节。

对上述标志进行一下总结：

- SA_NOCLDSTOP和SA_NOCLDWAIT只能用于SIGCHLD信号。
- SA_NODEFER与SA_RESETHAND可以和一种已废弃的、不可靠的、应决不使用（除非做9.4节中的练习）的信号机制相兼容。
- SA_ONSTACK只用于非常特殊的情况。
- SA_SIGINFO用于实时信号选项。
- SA_RESTART有时是很有用的。

总结一下动作及其对线程的影响：

- 信号的动作始终是基于整个进程范围的，可以是捕获、忽略、终止、停止或继续。
- 信号屏蔽是基于每个线程的。
- 可以明确设置sigaction来捕获和忽略信号；如果动作是SIG_DFL，就可以捕获、忽略、终止、停止或继续。但不能具体选择是上述4个中的哪一个（见9.1.3节）。
- 忽略、终止、停止和继续总是应用到整个进程。
- 被捕获的信号只能在一个线程中运行一个信号处理程序。如果信号被传递给进程（而不是指定的某个线程），同时如果有多个拥有该信号的线程都未阻塞，则信号传递到哪个线程完全是随机的。

下面给出一个忽略SIGINT和SIGQUIT信号的函数作为示例。它被6.4节中的shell调用：

```
static struct sigaction entry_int, entry_quit;

static bool ignore_sig(void)
{
    static bool first = true;
    struct sigaction act_ignore;

    memset(&act_ignore, 0, sizeof(act_ignore));
    act_ignore.sa_handler = SIG_IGN;
    if (first) {
        first = false;
        ec_negl( sigaction(SIGINT, &act_ignore, &entry_int) )
        ec_negl( sigaction(SIGQUIT, &act_ignore, &entry_quit) )
    }
    else {
        ec_negl( sigaction(SIGINT, &act_ignore, NULL) )
        ec_negl( sigaction(SIGQUIT, &act_ignore, NULL) )
    }
    return true;

    EC_CLEANUP_BGN
        return false;
    EC_CLEANUP_END
}

```

注意静态的bool变量first，它用来保证对每个信号只在第一次调用sigaction的时候捕获原始的动作。

以下是一个伴随函数，用来将动作恢复到调用ignore_sig之前的状态：

```
static bool entry_sig(void)
{

```

```

    ec_negl( sigaction(SIGINT, &entry_int, NULL) )
    ec_negl( sigaction(SIGQUIT, &entry_quit, NULL) )
    return true;
EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

```

9.1.7 信号处理程序

在调用 `sigaction` 为某个信号安装了信号处理程序之后，当捕获的信号传递后，该信号处理程序就会被调用。除非已设置了 `SA_NODEFER`（这很不常见），否则当信号处理程序运行时，捕获的信号将被阻塞。另外，在 `sigaction` 结构的 `sa_mask` 成员中设定的任何信号也会被阻塞。当信号处理程序返回时，即使已经在信号处理程序运行时明确修改了的临时屏蔽（通过调用 `pthread_sigmask` 或 `sigprocmask`），也会恢复原先的屏蔽。

那么，在捕获了一个信号之后该如何处理呢？这取决于它是什么类型的信号以及它产生的原因：

- 在内核检测到错误时，可能会产生信号。例如 `SIGFPE`（算法错误）或 `SIGPIPE`（对不存在读者的管道写入）。你可能需要显示或记录错误信息并终止线程或进程。由于某些信号的计算状态是不确定的，所以从处理程序返回可能不是一个好主意。同时，对于硬件产生的错误，如 `SIGFPE`，如果返回，进程可能会被终止，如 9.1.3 节中所述。
- 用户的某些操作可能会产生信号，例如按下 `Ctrl-c` 通常会产生一个 `SIGINT` 信号。可能需要在执行清理后终止程序，或者需要停止运算（如数据库查询）并返回到用户提示。这只是些例子——实际处理内容是和应用高度相关的。
- 信号可能是应用程序设计的一部分。例如给某进程发送一个 `SIGUSR1` 信号来表示数据文件已准备就绪。
- 定时器可能会超时。

无论做什么，都必须考虑以下两点：

- 1) 改变应用程序状态的信号处理程序中需要做哪些工作，以便使应用程序知道该信号发生了。
- 2) 从信号处理程序出发要到何处去。备选方案包括从处理程序返回、终止程序、全局跳转到程序的其他部分或产生另一个信号。

在信号处理程序中，所使用的系统调用或标准函数必须严格限制在那些能调用的范围内，因为信号可能发生在某个不能安全地重新进入的地方。事实上，SUS（第3版）仅定义了116个所谓的异步信号安全（`async-signal-safe`）函数，如表9-1所示。

一般而言，调用较高层的函数是不安全的，例如库中的函数甚至自己应用程序中的函数。因为一般不能明确了解它会完成怎样的工作，尤其是在它经过长时间发展演变之后。甚至不能调用 `printf`，这就是在本章开头的例子中不使用它而使用 `write` 的原因。

还有更糟的限制：也不能安全地引用全局变量，除非它是 `volatile sig_atomic_t` 类型的。

从技术角度上说，如果知道不会中断不安全（非异步信号安全）的函数，那么在处理程序中调用不安全的函数或引用不安全的存储器是可行的。但应该知道这只用于不常见的环境中，这种编程方式是不明智的。最好限制自己使用上表中列出的那些函数，并将全局变量的类型改为 `volatile sig_atomic_t`。

表9-1 异步信号安全函数

accept	getppid	sigdelset
access	getsocname	sigemptyset
aio_error	getsockopt	sigfillset
aio_return	getuid	sigismember
aio_suspend	kill	signal
alarm	link	sigpause
bind	listen	sigpending
cfgetispeed	lseek	sigprocmask
cfgetospeed	lstat	sigqueue
cfsetispeed	mkdir	sigset
cfsetospeed	mkfifo	sigsuspend
chdir	open	sleep
chmod	pathconf	socket
chown	pause	socketpair
clock_gettime	pipe	stat
close	poll	symlink
connect	posix_trace_event	sysconf
creat	pselect	tcdrain
dup	raise	tcflow
dup2	read	tcflush
execle	readlink	tcgetattr
execve	recv	tcgetpgrp
_exit/_Exit	recvfrom	tcsendbreak
fchmod	recvmsg	tcsetattr
fchown	rename	tcsetpgrp
fcntl	rmdir	time
fdatasync	select	timer_getoverrun
fork	sem_post	timer_gettime
fpathconf	send	timer_settime
fstat	sendmsg	times
fsync	sendto	umask
ftruncate	setgid	uname
getegid	setpgid	unlink
geteuid	setsid	utime
getgid	setsockopt	wait
getgroups	setuid	waitpid
getpeername	shutdown	write
getpgrp	sigaction	
getpid	sigaddset	

看来整个状况都充满了风险，事实也确实是这样的。下面给出一些建议，可以使你保持头脑清楚，使程序可靠：

- 先显示某个错误（使用write或其他安全函数），然后执行_exit退出的信号处理程序是正确的。
- 如果已经为信号设置了SA_RESTART标志，那么将标志类型设为volatile sig_atomic_t并返回是正确的。

• 避免使用信号处理程序是最佳选择。取而代之，可以使用线程和sigwait（参见9.2节）。事实上，读到这里，你可能已经决定采用最后一条建议了！但是，即使不用信号处理程序，信号还是很有用的，因为有两种不用信号处理程序就能完全安全地处理信号的方法：其一是sigsuspend（见9.2.3节），其二是sigwait（见9.2.2节），后者是更好的选择。

在本节结束之前，再举一个例子。它展示的是一个完全合法的信号处理程序，作用是对到达的信号进行记录然后返回。即使指定了SA_RESTART，sleep也还是会被中断，因为它不受SA_RESTART标志的影响。

```
static volatile sig_atomic_t gotsig = -1;

static void handler(int signum)
{
    gotsig = signum;
}

int main(void)
{
    struct sigaction act;
    time_t start, stop;

    memset(&act, 0, sizeof(act));
    act.sa_handler = handler;
    act.sa_flags = SA_RESTART;
    ec_negl( sigaction(SIGINT, &act, NULL) )
    printf("Type Ctrl-c in the next 10 secs.\n");
    ec_negl( start = time(NULL) )
    sleep(20);
    ec_negl( stop = time(NULL) )
    printf("Slept for %ld secs\n", (long)(stop - start));
    if (gotsig > 0)
        printf("Got signal number %ld\n", (long)gotsig);
    else
        printf("Did not get signal\n");
    exit(EXIT_SUCCESS);

    EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
    EC_CLEANUP_END
}
```

我在看到第一行后就按下了Ctrl-c，输出结果如下：

```
Type Ctrl-c in the next 10 secs.
Slept for 4 secs
Got signal number 2
```

9.1.8 信号处理的底线

即使决定了不使用信号处理程序，但为了防止应用程序被用户意外终止，通常也还是需要进行最小限度的信号处理。同样，如果由于存在诸如引用无效存储器[“segmentation violation”（段冲突）]之类的bug而使得程序不正常终止，是很不专业的。更好的做法是捕获信号、记录问题和通知用户（采用比“程序因段冲突中断”更详细的信息，或者由shell来决定显示的内容）。

因此，对大多数应用程序而言，至少都需要进行以下工作：

- 程序一旦开始，就立即阻塞所有信号，像这样：

```
sigset_t set;

ec_negl( sigfillset(&set) )
ec_negl( sigprocmask(SIG_SETMASK, &set, NULL) )
```

(如果有多个线程，则可以使用pthread_sigmask。)

- 将所有不想捕获的键盘产生的信号设置成忽略，如SIGINT。
- 捕获SIGTERM信号，当其到达时执行清理并终止，就像系统管理员关闭进程时的标准做法一样。
- 捕获所有错误产生的信号，当其中某个信号到达时进行显示并/或记录错误。
- 忽略SIGPIPE信号，使得在写入空管道时write返回一个错误。这比收到一个信号更方便。
- 调用sigemptyset和sigprocmask (或pthread_sigmask) 对所有信号解除阻塞。

在应用程序的开头可以调用以下函数，来进行最小化的信号处理：

```
static bool handle_signals(void)
{
    sigset_t set;
    struct sigaction act;

    ec_negl( sigfillset(&set) )
    ec_negl( sigprocmask(SIG_SETMASK, &set, NULL) )
    memset(&act, 0, sizeof(act));
    ec_negl( sigfillset(&act.sa_mask) )
    act.sa_handler = SIG_IGN;
    ec_negl( sigaction(SIGHUP, &act, NULL) )
    ec_negl( sigaction(SIGINT, &act, NULL) )
    ec_negl( sigaction(SIGQUIT, &act, NULL) )
    ec_negl( sigaction(SIGPIPE, &act, NULL) )
    act.sa_handler = handler;
    ec_negl( sigaction(SIGTERM, &act, NULL) )
    ec_negl( sigaction(SIGBUS, &act, NULL) )
    ec_negl( sigaction(SIGFPE, &act, NULL) )
    ec_negl( sigaction(SIGILL, &act, NULL) )
    ec_negl( sigaction(SIGSEGV, &act, NULL) )
    ec_negl( sigaction(SIGSYS, &act, NULL) )
    ec_negl( sigaction(SIGXCPU, &act, NULL) )
    ec_negl( sigaction(SIGXFSZ, &act, NULL) )
    ec_negl( sigemptyset(&set) )
    ec_negl( sigprocmask(SIG_SETMASK, &set, NULL) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

以下是实际的处理程序及其调用的两个支持函数：

```
static void handler(int signum)
{
    int i;
    struct {
        int signum;
        char *msg;
    } sigmsg[] = {
```



```

    { SIGTERM, "Termination signal" },
    { SIGBUS, "Access to undefined portion of a memory object" },
    { SIGFPE, "Erroneous arithmetic operation" },
    { SIGILL, "Illegal instruction" },
    { SIGSEGV, "Invalid memory reference" },
    { SIGSYS, "Bad system call" },
    { SIGXCPU, "CPU-time limit exceeded" },
    { SIGXFSZ, "File-size limit exceeded" },
    { 0, NULL }
};

clean_up();
for (i = 0; sigmsg[i].signum > 0; i++)
    if (sigmsg[i].signum == signum) {
        (void)write(STDERR_FILENO, sigmsg[i].msg,
                    strlen_safe(sigmsg[i].msg));
        (void)write(STDERR_FILENO, "\n", 1);
        break;
    }
_exit(EXIT_FAILURE);
}

static void clean_up(void)
{
    /*
     * Clean-up code goes here --
     * must be async-signal-safe.
     */
}

static size_t strlen_safe(const char *s)
{
    size_t n = 0;
    while (*s++ != '\0')
        n++;
    return n;
}

```

这个办法就是将clean_up的内容替换成应用程序需要的代码。听上去真可笑，标准的strlen并不在异步信号安全函数的列表中，因此我编写了自己的strlen版本。同样的原因，在处理程序中使用write代替fprintf。还要注意的，使用的是_exit而不是exit——如5.7节中所述，加下划线版中跳过了对atexit的调用并刷新了标准C的I/O缓冲区，这是使异步信号安全正常退出的唯一办法。

9.1.9 人工产生信号

如9.1.3节所述，每个可以自然产生的信号，也可以由调用kill、killpg、pthread_kill、abort、raise或sigqueue（见9.5.4节）显式地产生：

kill —— 为进程产生信号

```

#include <signal.h>

int kill(
    pid_t pid,           /* process ID or other specification */
    int signum          /* signal */
);
/* Returns 0 on success or -1 on error (sets errno) */

```

killpg——为进程组产生信号

```
#include <signal.h>

int killpg(
    pid_t pgrp,          /* process-group ID */
    int signum          /* signal */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

pthread_kill——为线程产生信号

```
#include <signal.h>

int pthread_kill(
    pthread_t thread_id, /* thread ID */
    int signum          /* signal */
);
/* Returns 0 on success, error number on failure (errno not set) */
```

abort——产生SIGABRT

```
#include <stdlib.h>

void abort(void);
/* Does not return */
```

raise——为线程产生信号

```
#include <signal.h>

int raise(
    int signum          /* signal */
);
/* Returns 0 on success or non-zero on error (sets errno) */
```

错误命名的kill系统调用可以向一个或多个具有向其发送信号权限的进程发送任何信号，而不仅仅是SIGKILL。如果进程由超级用户运行，或者发送进程的实际用户ID或有效用户ID和接收进程的实际用户ID或保存的set-user-ID相符，那么就会拥有该权限。信号将发送给哪个进程，取决于pid参数：

- >0 进程ID为pid的进程。
- 0 进程组ID和发送进程相同的进程组。
- <-1 进程组ID和pid绝对值相同的进程组。
- 1 发送者拥有权限的所有进程，由实现定义的系统进程集合除外。

如果signum为0，则kill仅测试其pid参数的有效性。这是一种检测进程或进程组是否存活的方法。如果发送进程没有发送信号的权限，那么调用会失败，但errno值会表明进程或进程组是否存活：如果pid不存在，则其值为ESRCH；如果它存活但发送者没有权限，则其值为EPERM。

killpg是一个完全没必要存在的系统调用，仅对进程组ID是pgrp的进程组发送信号，因此它相当于：

```
kill(-pgrp, signum);
```

pthread_kill和kill类似，但它只向thread_id线程发送信号，该线程必须和发送线程处于同一进程。它不像kill对进程那样拥有广播能力。请小心使用pthread_kill——记住终止、停止和继续信号总是会影响整个进程。因此，pthread_kill只在完成捕获信号工作时和你的期望相一致。如果你运行

```
pthread_kill(tid, SIGKILL);
```

则进程会被终止，而不仅仅是tid线程。（应该使用pthread_cancel来终止单个线程。）

abort和带有一个SIGABRT参数的kill几乎相同，但除非捕获信号，并且信号处理程序不会返回（例如调用siglongjmp或_exit），否则会终止进程，就像执行了SIGABRT的默认行为；abort从来不会返回。例如：如果使用sigaction将SIGABRT置为SIG_IGN，则abort会终止进程，而

```
kill(getpid(), SIGABRT)
```

则不起作用。

raise实际上是一个标准的C函数，它向运行它的线程发送信号。就是说，它相当于：

```
pthread_kill(pthread_self(), signal);
```

然而，即使在POSIX线程选项不被支持的情况下，raise也总是可用的。这种情况下，它相当于：

```
kill(getpid(), signal);
```

9.1.10 fork、pthread_create和exec对信号的影响

三个用于设置新进程、线程或程序的属性的线程是fork、pthread_create或exec：

1) 信号动作：在执行fork之后，子进程将继承全部的信号动作。在执行exec之后，被设置成SIG_DFL的信号将保持原样；被设置成SIG_IGN的信号也将保持原样，除了SIGCHLD（它将根据实现而定）被设置为SIG_IGN或SIG_DFL；已捕获的信号被设置为SIG_DFL。由于所有的动作都是整个进程范围的，所以pthread_create不起作用。

2) 信号屏蔽：在执行fork之后，从父线程继承；在执行exec之后，保持父线程的原样；在执行pthread_create之后，从父线程复制到新的线程。

3) 挂起信号：在执行fork之后清除挂起信号；在执行exec之后，和父线程相同；在执行pthread_create之后清除挂起信号。

记住上述9条规则（3条属性乘以3个系统调用）的最简单的方法是：区别就是如果是复制或继承，属性就未经更改。因此只需要记住三条例外，其中前两条很有意义：

- 如果信号处理程序消失，就像对exec那样，捕获的信号必须改成SIG_DFL。
- 挂起的信号是针对每个进程或每个线程的，因此在出现新进程或新线程时，要将它们清除。
- 和加强移植能力相比而言，标准制定者更关心的是不要打破现存的实现方式，因此他们对SIGCHLD表示不满。

9.2 等待信号

本节描述允许进程等待信号传递的系统调用。

9.2.1 pause系统调用

至此我们已经讲述了许多系统调用，它们在完成某些动作之前都会阻塞对某事件的等待。例如，当读取终端时，read通常等待输入一个整行。pause是纯粹的等待：它不做任何事情，也不等待任何特殊的事情。

pause——等待信号

```
#include <unistd.h>

int pause(void);
/* Returns -1 on error (sets errno) */
```

由于已传递的信号会中断大多数被阻塞的系统调用，因此也可以说**pause**等待的是已捕获的信号。如果信号捕获函数返回，则**pause**会返回并将**errno**置为**EINTR**，但因为这可能是**pause**返回的唯一方式，所以很难对它进行测试。

通常，可使用更完善的调用如**sigwait**代替**pause**。

9.2.2 sigwait系统调用

和**pause**不同，**sigwait**能让你选择想要等待什么。你不需要信号处理程序，因为它返回时，你就能知道到达的是什么信号：

sigwait——等待信号

```
#include <signal.h>

int sigwait(
    const sigset_t *set,      /* signals to wait for */
    int *signum              /* signal that was accepted */
);
/* Returns 0 on success or error number on error */
```

参数**set**是一组**sigwait**所要等待的信号。当其中某个挂起时，其编号将通过**signum**参数和**sigwait**的返回而返回。如果在调用**sigwait**时一个或更多信号已经挂起，则该调用会以不定的方式选择其一并立即返回。**sigwait**返回信号的术语是“已接受”；而不是“已传递”。

在使用**sigwait**时，需要的是让信号保持在挂起状态直到**sigwait**返回它；而不是想让信号被传递。因此，需要阻塞**sigwait**正等待的信号并将其保持在阻塞状态。（信号的生命周期为：产生—挂起—已传递，这已在9.1.2节中描述过了。）

如果不止一个线程在使用**sigwait**等待发往某进程的同一个信号，那么只有一个线程可以获得它，其选择方法是不定的。如果信号被发往某个指定的线程，那么只有那个线程的**sigwait**（如果有的话）可以返回它。

典型情况下，**sigwait**用于以下两个目的之一：

- 直到和某信号关联的一些事件发生后，线程才能继续进行的情况。例如，当某消息到达时，一个线程可能向另一个线程发送**SIGUSR1**。但是，通常情况下，达到这一目的的更好做法是使用一个条件变量（见5.17.4节）。本节的后面有一个示例。
- 在某线程被设计用来处理信号时。就是说，使用等待线程代替信号处理函数。一个线程运行**sigwait**，同时其他线程都在等待该信号被阻塞。但这只在信号发送给进程时有效——如果信号发送给线程，那么只有该线程的**sigwait**可以返回它。

对发往进程的信号而言，使用**sigwait**比使用信号处理函数更好，因为不存在信号处理程序要遵循的各种限制。当**sigwait**返回时，可以自由地调用任何系统调用或函数，或进行任何能在线程中完成的其他工作。

9.1.8节中给出过一个**handle_signals**函数，它捕获了所有检测出的错误信号（如**SIGFPE**、**SIGSEGV**），因此在其退出前可以调用清理函数并显示详细的信息。下面让我们重

写该函数的代码，在线程中使用sigwait代替信号处理程序：

```

static bool handle_signals(void) /* do not use -- see below */
{
    sigset_t *set;
    struct sigaction act;
    pthread_t tid;

    ec_null( set = malloc(sizeof(*set)) )
    ec_negl( sigfillset(set) )
    ec_rv( pthread_sigmask(SIG_SETMASK, set, NULL) )
    memset(&act, 0, sizeof(act));
    act.sa_handler = SIG_IGN;
    ec_negl( sigaction(SIGHUP, &act, NULL) )
    ec_negl( sigaction(SIGINT, &act, NULL) )
    ec_negl( sigaction(SIGQUIT, &act, NULL) )
    ec_negl( sigaction(SIGPIPE, &act, NULL) )
    ec_negl( sigemptyset(set) )
    ec_negl( sigaddset(set, SIGTERM) )
    ec_negl( sigaddset(set, SIGBUS) )
    ec_negl( sigaddset(set, SIGFPE) )
    ec_negl( sigaddset(set, SIGILL) )
    ec_negl( sigaddset(set, SIGSEGV) )
    ec_negl( sigaddset(set, SIGSYS) )
    ec_negl( sigaddset(set, SIGXCPU) )
    ec_negl( sigaddset(set, SIGXFSZ) )
    ec_rv( pthread_sigmask(SIG_SETMASK, set, NULL) )
    ec_rv( pthread_create(&tid, NULL, sig_thread, set) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

static void *sig_thread(void *arg)
{
    int signum;
    int i;
    struct {
        int signum;
        char *msg;
    } sigmsg[] = {
        { SIGTERM, "Termination signal" },
        { SIGBUS, "Access to undefined portion of a memory object" },
        { SIGFPE, "Erroneous arithmetic operation" },
        { SIGILL, "Illegal instruction" },
        { SIGSEGV, "Invalid memory reference" },
        { SIGSYS, "Bad system call" },
        { SIGXCPU, "CPU-time limit exceeded" },
        { SIGXFSZ, "File-size limit exceeded" },
        { 0, NULL }
    };

    while (true) {
        ec_rv( sigwait((sigset_t *)arg, &signum) )
        clean_up();
        for (i = 0; sigmsg[i].signum > 0; i++)
            if (sigmsg[i].signum == signum) {
                fprintf(stderr, "%s\n", sigmsg[i].msg);
                break;
            }
    }
}

```

```

    }
    _exit(EXIT_FAILURE);
}
return (void *)true; /* never get here */

EC_CLEANUP_BGN
    EC_FLUSH(*sig_thread*)
    return (void *)false;
EC_CLEANUP_END
}

static void clean_up(void)
{
    /*
     * Clean-up code goes here --
     * need not be async-signal-safe.
     */
}

```

这个版本的代码和9.1.8节中的代码相比，具有如下优点：当信号被sigwait返回时，因为位于线程而非信号处理程序中，所以可自由地调用任何系统调用或喜欢的函数——而不必局限在异步信号安全函数列表中。注意，clean_up函数中的注释也作了相应的改变。

该版本代码的缺点就是它根本就不能工作！其原因有两条，都很严重：

- 例如，如果某个其他的线程获得了SIGSYS，则信号将被送往该线程而不是其进程，sig_thread函数中的sigwait无法返回它。事实上，因为该信号在所有线程中都被阻塞，所以它将会被永远挂起。因此，使用信号处理程序的最初版本，就是错误检测信号应该采用的版本。
- 如果硬件检测信号SIGBUS、SIGFPE、SIGILL和SIGSEGV中的一个在阻塞时自然发生，则其结果是不定的（见9.1.3节）。最大的可能是进程被立刻终止，因此sigwait永远没有返回的机会。由于在初始化设置后这4种信号都是未阻塞的，所以这在信号处理程序版本中不会成为问题。

以上并不是说sigwait是没有用的。大多数信号，包括除9.1.3节中错误检测信号组以外的全部信号，当它们自然（即不是由pthread_kill）产生时都会被送往进程，因此线程在sigwait中等待工作得非常好，并且是一种比信号处理程序要好得多的选择。

9.2.3 sigsuspend系统调用

sigsuspend是一种比较老的、不支持多线程的系统调用，也用于等待信号。在详细讲述之前，先浏览一下它所能解决的问题。在上文第8章中，一直在使用这样一个示例：它通过派生来产生和父进程的socket相连接的进程。对父进程而言，在子进程连接之前先进行socket绑定是很重要的。为此在示例中使用了拙劣的技巧：使子进程休眠几秒，来为父进程提供绑定的机会。休眠是不可靠的，不仅因为无从知晓这几秒是否够用，而且还不知道它是不是太长而降低了效率。简便起见，下面给出一个具有同样问题的简化示例：

```

void try1(void)
{
    if (fork() == 0) {
        printf("child\n");
        exit(EXIT_SUCCESS);
    }
    printf("parent\n");
    return;
}

```

该函数显示

```
child
parent
```

但需要父进程先运行，然后告知子进程何时开始进行。这里首先尝试将它们同步，父进程向子进程发送SIGUSR1信号，子进程捕获它并设置一个变量：

```
static volatile sig_atomic_t got_sig;
static void handler(int signum)
{
    if (signum == SIGUSR1)
        got_sig = 1;
}

void try2(void)
{
    pid_t pid;

    got_sig = 0;
    ec_negl( pid = fork() )
    if (pid == 0) {
        struct sigaction act;

        memset(&act, 0, sizeof(act));
        act.sa_handler = handler;
        ec_negl( sigaction(SIGUSR1, &act, NULL) )
        while (got_sig == 0)
            if (pause() == -1 && errno != EINTR)
                EC_FAIL
        printf("child\n");
        exit(EXIT_SUCCESS);
    }
    printf("parent\n");
    ec_negl( kill(pid, SIGUSR1) )
    return;
}

EC_CLEANUP_BGN
    EC_FLUSH(*try2*)
EC_CLEANUP_END
}
```

处理程序是安全的——它只是设置了一个认可类型的变量。子进程在循环中测试该变量，挂起并等待信号的到达。（`pause`挂起见9.2.1节，它会阻塞直至信号到达。）这时，输出的次序和我们期望得到的就一致了：

```
parent
child
```

但还存在两个问题：

- 如果在子进程有机会安装处理程序之前就将SIGUSR1传递给它，那么SIGUSR1会终止该子进程。可能的解决方案是在`fork`之前就安装处理程序，这样子进程会继承它，但这只在父进程与子进程之间有效。我们想要一个对任意需要同步的进程都有效的解决方案。
- 如果在`while`语句中的测试和`pause`调用两个动作之间传递SIGUSR1，则`pause`将会永远等待下去，因为用来唤醒它的信号在它进入休眠之前就早早到达了。

下面尝试通过在做好准备之前阻塞SIGUSR1信号的方法来解决上述问题：

```

void try3(void)
{
    sigset_t set;
    pid_t pid;

    got_sig = 0;
    ec_negl( sigemptyset(&set) )
    ec_negl( sigaddset(&set, SIGUSR1) )
    ec_negl( sigprocmask(SIG_SETMASK, &set, NULL) )
    ec_negl( pid = fork() )
    if (pid == 0) {
        struct sigaction act;
        sigset_t suspendset;

        memset(&act, 0, sizeof(act));
        act.sa_handler = handler;
        ec_negl( sigaction(SIGUSR1, &act, NULL) )
        ec_negl( sigfillset(&suspendset) )
        ec_negl( sigdelset(&suspendset, SIGUSR1) )
        ec_negl( sigprocmask(SIG_SETMASK, &suspendset, NULL) )
        while (got_sig == 0)
            if (pause() == -1 && errno != EINTR)
                EC_FAIL
        printf("child\n");
        exit(EXIT_SUCCESS);
    }
    printf("parent\n");
    ec_negl( kill(pid, SIGUSR1) )
    return;
}

EC_CLEANUP_BGN
    EC_FLUSH("try3")
EC_CLEANUP_END
}

```

这种方法能够彻底解决第一个问题。由于阻塞了SIGUSR1，所以直到处理程序安装完毕后才可能到达。但第二个问题仍然存在，无法解决。虽然这个时间间隔很小，但信号仍有可能在测试和pause之间到达。

我们需要找到一种保持信号阻塞直到pause开始的方法。或者，换句话说，需要取消阻塞并使pause变成自动的。而这正好就是sigsuspend所能完成的：

sigsuspend——改变信号掩码并等待信号

```

#include <signal.h>

int sigsuspend(
    const sigset_t *sigmask      /* temporary signal mask */
);
/* Returns -1 on error, always (sets errno) */

```

sigsuspend用sigmask临时代替了线程的信号屏蔽，然后等待直到某个动作为终止或被捕获的未阻塞信号完成传递。如果其动作为终止，则进程（记住，不仅仅是线程）会被终止，sigsuspend并不返回。如果是被捕获，并且信号处理程序返回，则会恢复原先的信号屏蔽，同时sigsuspend会返回一个错误。通常，这个错误只是EINTR；就是说，当errno或EINTR的返回值为-1时，对中断的系统调用来说是正常的（和pause类似，如上例所示）。

从本质上说,在所有情况下,传递给sigsuspend的屏蔽对在调用之前就已经阻塞的一个或更多的信号具有取消阻塞的作用,尽管实际上这不是必要条件。这只是一个细节,对此不必太关注。

很好,这种方法非常完美。现在开始解决上面的同步问题,只需要简单地将pause替换成sigsuspend就可以了。不再需要while循环,因为现在取消信号阻塞和挂起都已经是自动的了。

```
void try4(void)
{
    sigset_t set;
    pid_t pid;

    ec_negl( sigemptyset(&set) )
    ec_negl( sigaddset(&set, SIGUSR1) )
    ec_negl( sigprocmask(SIG_SETMASK, &set, NULL) )
    ec_negl( pid = fork() )
    if (pid == 0) {
        struct sigaction act;
        sigset_t suspendset;

        memset(&act, 0, sizeof(act));
        act.sa_handler = handler;
        ec_negl( sigaction(SIGUSR1, &act, NULL) )
        ec_negl( sigfillset(&suspendset) )
        ec_negl( sigdelset(&suspendset, SIGUSR1) )
        if (sigsuspend(&suspendset) == -1 && errno != EINTR)
            EC_FAIL
        printf("child\n");
        exit(EXIT_SUCCESS);
    }
    printf("parent\n");
    ec_negl( kill(pid, SIGUSR1) )
    return;

    EC_CLEANUP_BGN
    EC_FLUSH("try4")
    EC_CLEANUP_END
}
```

仍然可以使用前面那个设置了got_sig变量的处理函数;但是,现在已经不再需要那个变量了,只要使用空的处理程序就行了:

```
static void handler(int signum)
{
}
```

同时,也不需要 sigsuspend 返回的代码是否为-1进行测试了,因为其返回值总是-1。但是去掉它会显得有些古怪,还会使得那些恰好不知道或不记得 sigsuspend 全部细节的读者感到迷惑。

注意,由于在执行fork之前SIGUSR1一直是被阻塞的,所以子进程继承了其信号屏蔽。在需要同步的两个进程没有关系时,调用sigsuspend的进程只会简单地执行自己的阻塞。实际上,这只是那个在9.1.8节中提到的启动每个应用程序时应屏蔽所有信号的一般性建议的一个特例。

另外请注意,和sigwait不同, sigsuspend几乎总可以和信号处理程序一起使用。但

通常处理程序什么也不干，只是使被传递的信号可以中断sigsuspend。

sigsuspend和sigwait之间的另一个较大的区别是：使用sigwait时，所等待的信号将保持被阻塞的状态。事实上，它从来不会被传递——而是sigwait接受它：将它从挂起的信号集合中删除并返回它。因此，在使用sigwait时，并不存在那个sigsuspend消除的，取消信号阻塞和等待信号被传递之间的竞争条件，因为它一直阻塞信号。所以，上述同步代码还可以进一步简化：

```
void try5(void)
{
    sigset_t set;
    pid_t pid;

    ec_negl( sigemptyset(&set) )
    ec_negl( sigaddset(&set, SIGUSR1) )
    ec_negl( sigprocmask(SIG_SETMASK, &set, NULL) )
    ec_negl( pid = fork() )
    if (pid == 0) {
        int signum;

        ec_rv( sigwait(&set, &signum) )
        printf("child\n");
        exit(EXIT_SUCCESS);
    }
    printf("parent\n");
    ec_negl( kill(pid, SIGUSR1) )
    return;
}

EC_CLEANUP_BGN
    EC_FLUSH("try5")
EC_CLEANUP_END
}
```

应当说明的是，使用管道也可以完成父进程和子进程之间的同步，并完全绕过了使用信号的复杂性：

```
void try6(void)
{
    int pfd[2];
    pid_t pid;

    ec_negl( pipe(pfd) )
    ec_negl( pid = fork() )
    if (pid == 0) {
        char c;

        ec_negl( close(pfd[1]) )
        ec_negl( read(pfd[0], &c, 1) )
        ec_negl( close(pfd[0]) )
        printf("child\n");
        exit(EXIT_SUCCESS);
    }

    printf("parent\n");
    ec_negl( close(pfd[0]) )
    ec_negl( close(pfd[1]) )
    return;
}

EC_CLEANUP_BGN
```



```

    EC_FLUSH("try6")
    EC_CLEANUP_END
}

```

在该例中，在父进程关闭管道的写入端，使得子进程得到返回值0之前，子进程一直阻塞在read中。

9.3 其他信号系统调用

可以使用sigpending来找出挂起的信号，它会返回一组处于挂起状态的信号：

sigpending —— 检查未决信号

```

#include <signal.h>

int sigpending(
    sigset_t *set          /* returned set of pending signals */
);
/* Returns 0 on success or -1 on error (sets errno) */

```

可以使用sigismember（见9.1.5节）对返回集合中的挂起信号进行测试。当然，除非阻塞了信号，否则在测试时信号可能已经不属于未决状态了。

回顾9.1.6节可知，SA_ONSTACK标志使得信号处理程序可以在另一个可选的堆栈运行。可以使用sigaltstack来管理该堆栈：

sigaltstack —— 设置或得到备用栈上下文

```

#include <signal.h>

int sigaltstack(
    const stack_t *stack, /* new stack */
    stack_t *ostack      /* old stack */
);
/* Returns 0 on success or -1 on error (sets errno) */

```

请参阅系统文档或SUS，了解该系统调用的详细信息。

同样回想可知，SA_RESTART标志阻止函数被信号中断。使用siginterrupt时，无需调用sigaction，就可将该标志开启或关闭：

siginterrupt —— 设置或清除SA_RESTART标志

```

#include <signal.h>

int siginterrupt(
    int signum,          /* signal */
    int flag             /* non-zero to clear, zero to set */
);
/* Returns 0 on success or -1 on error (sets errno) */

```

9.4 不赞成使用的信号系统调用

本节中的系统调用是标准化过的，但它们并没有为现存的系统调用增加任何的功能性，所以不值得花费你太多的时间，除非为了完成练习9.4和练习9.5。

为信号设置动作的典型方法是使用signal系统调用：

signal —— 设置信号动作

```
#include <signal.h>

void (*signal(
    int signum,          /* signal */
    void (*act)(int)    /* new action */
))(int);
/* Returns old action or SIG_ERR on error (sets errno) */
```

其古怪的声明意味着signal返回一个动作，它可以是一个指向带有一个整数参数（信号编号）的void函数的指针。

在使用signal捕获信号时存在两个问题：

- 在传递时，动作被设为其默认值。如果还需要捕获它，就必须再次调用signal。
- 已传递的信号不会被阻塞，因此，第二个信号的到达会终止进程。

绕过这些问题的方法是使用sigaction（见9.1.6节），忘了signal吧。

有5个系统调用可以提供简化的信号处理，但应当避免使用它们，因为它们也完成不了那些基本函数（如sigaction）做不好的事情。下面对其进行简单描述。

在不使用结构的情况下，可以使用sigset为信号设置sigaction风格的动作：

sigset —— 设置信号动作

```
#include <signal.h>

void (*sigset(
    int signum,          /* signal */
    void (*act)(int)    /* new action */
))(int);
/* Returns old action or SIG_ERR on error (sets errno) */
```

sigset和signal调用一样简单，但它具有sigaction的行为，即在处理函数运行时屏蔽了已传递的信号，同时动作不发生改变。另外，它具有一个新的动作——SIG_HOLD，该动作只是将信号加入到信号屏蔽中去。和sigaction不同，如果调用没有SIG_HOLD动作的sigset，那么伴随它将发生这样的情况：它也会解除信号阻塞（将其从信号屏蔽中去除）。

避免使用sigset及伴随它的其他相关函数的主要原因是：掌握sigaction的功能就已经很困难了，再试图学会其他几种不同的功能组合岂不是难上加难。少才是好！避免使用它们的另一个原因是：它们没有在多线程环境中定义。想象一下，如果已经在单线程程序中使用了它们，那么在日后增加多线程的时候，会遇到什么样的问题！

可以调用sigrelse来直接取消信号阻塞（将其从信号屏蔽中去除）：

sigrelse —— 解除信号阻塞

```
#include <signal.h>

int sigrelse(
    int signum,          /* signal */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

sigpause是sigsuspend的简化版本，它可以从信号屏蔽中去除一个信号，并挂起直到某信号到达（任何未阻塞的信号），然后恢复原屏蔽：

sigpause——改变信号掩码并等待信号

```
#include <signal.h>

int sigpause(
    int signum,          /* signal */
);
/* Returns -1 on error, always (sets errno) */
```

最后，还有两个多余的系统调用，因为它们的功能和带有SIG_HOLD与SIG_IGN动作的sigset完全相同：

sighold——阻塞信号

```
#include <signal.h>

int sighold(
    int signum,          /* signal */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

sigignore——忽略信号

```
#include <signal.h>

int sigignore(
    int signum,          /* signal */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

9.5 实时信号扩展

在所谓的POSIX.4（实时）标准中包含了一种新的信号机制——实时信号扩展（RTS），它对本章前面描述的信号相关的系统调用的进行了如下改进：

- 增加了应用程序信号的数量，而不仅限于SIGUSR1和SIGUSR2。
- 允许信号排队，因此在同类信号挂起时不会丢失产生的信号。
- 指定了信号传递的次序。

• 在信号中包含了附加的信息，例如发送方是谁，为何发送以及可能的一些应用程序数据。

新功能附加在了原有的功能之上，因此仍然可以使用28种典型的信号（见9.1.3节），包括使用同样的处理程序和其他动作，使用同样的同步调用（比如kill）等。虽然在许多系统中可以对原有的信号使用新的RTS功能（如排队、传递值），但在标准中对此没做说明，所以需要移植的话应避免这样做。

有几个功能测试宏（见1.5.4节）可以检测出这些功能是否可用。其中重要的是_POSIX_REALTIME_SIGNALS。

本节中的各个小节将详细讨论大多数RTS功能。在很多情况下，查阅使用RTS信号机制来发送信号的函数组是很方便的。在下一节中将列出那些与代码SI_QUEUE、SI_TIMER、SI_ASYNCIO和SI_MESGQ相关的函数。本书将该组函数称为RTS产生函数（这是本书作者为其命的名，其他地方并未使用该名称）。

更多的RTS功能描述见9.7.5节和9.7.6节。

9.5.1 RTS信号处理程序

在继续阅读之前，可能需要回顾一下9.1.6节中关于sigaction的讨论。

如果在传递给sigaction的结构中设定了SA_SIGINFO标志，那么可以使用sa_sigaction成员代替sa_handler成员来保存指向信号处理函数的指针。^①它并不是一个单纯的信号编号参数，而有一个更加精心设计的原型：

```
void rts_handler(int signum, siginfo_t *info, void *context);
```

info参数指向的是siginfo_t结构中有关信号的信息，我们在5.8节使用waitid时曾首次使用过该结构：

siginfo_t —— sigaction的结构

```
typedef struct {
    int si_signo;           /* signal number */
    int si_errno;         /* errno value associated with signal */
    int si_code;          /* signal code (see below) */
    pid_t si_pid;         /* sending process ID */
    uid_t si_uid;         /* real user ID of sending process */
    void *si_addr;        /* address of faulting instruction */
    int si_status;        /* exit value or signal */
    long si_band;         /* band event for SIGPOLL */
    union sigval si_value; /* signal value */
} siginfo_t;
```

union sigval —— 信号值的union^②

```
union sigval {
    int sival_int;        /* integer signal value */
    void *sival_ptr;     /* pointer signal value */
};
```

成员si_signo只是对处理程序中signum参数的重复。

si_errno的用法取决于实现；它可能包含一个表明信号起因的错误代码。

成员si_code表明了信号产生的原因。通常，如果它是0或负数，则说明信号来自于某个进程；其进程ID在si_pid中，实际用户ID在su_uid中。si_code成员可以取下列值中的一个：

SI_USER 由kill发送，或由实现随意决定，由一个用于合成信号的系统调用（如raise）发送。

SI_QUEUE 由sigqueue发送（见9.5.4节）。

SI_TIMER 定时器的超时时间，由timer_settime确定（见9.7.6节）。

SI_ASYNCIO 完成异步I/O（见3.9节）。

SI_MESGQ 消息的到达（见7.7节）。

后4种RTS产生函数还有一个可随信号一起发送的值，能通过si_value成员进行访问。

如果si_code是正确的，则说明信号来自于内核，而代码取决于信号。例如，如果信号是SIGFPE，则如果整数被零除，代码就是FPE_INTDIV；如果整数溢出，代码就是FPE_INTOVF；如果浮点数被零除，代码就是FPE_FLTDIV。有关SIGFPE的代码和其他信号的代码的完整列表，请参见[SUS2002]或所使用的系统文档。此外，5.8节中列出了SIGCHLD的代码。

① 对于sa_sigaction成员中是否能使用SIG_IGN或SIG_DFL，SUS并没有明确说明；为安全起见，仅在sa_handler中使用它们。

② FreeBSD（可能还有其他系统）将成员定义成了sival_int和sival_ptr，但没有声明它们支持RTS，可能会支持吧。

怎样使用其他的成员，是由信号决定的。其中有关SIGCHLD的内容在5.8节中。对于SIGILL和SIGSEGV，成员si_addr给出了导致问题的实际机器地址。对于SIGPOLL（和STREAMS一起使用），成员si_band指明了波段事件。

以下是一个程序，当设置SA_SIGINFO标志时，它可以显示在信号处理程序中可用的一些附加信息：

```
int main(void)
{
    struct sigaction act;
    union sigval val;

    memset(&act, 0, sizeof(act));
    act.sa_flags = SA_SIGINFO;
    act.sa_sigaction = handler;
    ec_negl( sigaction(SIGUSR1, &act, NULL) )
    ec_negl( sigaction(SIGRTMIN, &act, NULL) )
    ec_negl( kill(getpid(), SIGUSR1) )
    val.sival_int = 1234;
    ec_negl( sigqueue(getpid(), SIGRTMIN, val) )
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}

static void handler(int signum, siginfo_t *info, void *context)
{
    printf("signal number: %d\n", info->si_signo);
    printf("sending process ID: %ld\n", (long)info->si_pid);
    printf("real user ID of sending process: %ld\n", (long)info->si_uid);
    switch (info->si_code) {
    case SI_USER:
        printf("Signal from user\n");
        break;
    case SI_QUEUE:
        printf("Signal from sigqueue; value = %d\n",
            info->si_value.sival_int);
        break;
    case SI_TIMER:
        printf("Signal from timer expiration; value = %d\n",
            info->si_value.sival_int);
        break;
    case SI_ASYNCIO:
        printf("Signal from asynchronous I/O completion; value = %d\n",
            info->si_value.sival_int);
        break;
    case SI_MESGQ:
        printf("Signal from message arrival; value = %d\n",
            info->si_value.sival_int);
        break;
    default:
        printf("Other signal\n");
    }
}
```

将以上代码和本章的第一个例子相比，可以看到为调用sigaction所进行的设置中存在一些不同：设置了SA_SIGINFO标志，同时处理程序的成员是sa_sigaction而不是

sa_handler。在下一节中将引入SIGRTMIN信号；但现在，为方便理解代码，请把它看作是SIGUSR2。

尽管printf不是异步信号安全的，但在处理程序中调用printf是合法的，因为信号的产生者是异步信号安全的系统调用：kill或sigqueue（将在9.5.4节中讲到）。

以下是我得到的输出结果：

```
signal number: 10
sending process ID: 29501
real user ID of sending process: 500
Signal from user
signal number: 32
sending process ID: 29501
real user ID of sending process: 500
Signal from sigqueue; value = 1234
```

信号处理程序的第三个参数context是一个ucontext_t类型对象的指针，指向描述接收进程中断时的上下文环境。该指针指向的不是ucontext_t类型，而是void类型，因为在该原型引入时，新类型还没有标准化。该指针通常并未实现而且很少使用，详细信息请参见[SUS2002]或所使用的系统文档。

9.5.2 RTS信号

RTS引入了一组新的信号，其编号范围从SIGRTMIN到SIGRTMAX，其中至少RTSIG_MAX信号是可用的。可以使用sysconf（见1.5.5节）在运行时获得实际可用的数目，但一般来说至少有8个。（这是作者的Solaris版本中的实际数目；在Linux上有32个。）

对于RTS产生函数（在9.5节开始处进行了定义）而言，开始应确定所需要的信号，然后选出一个RTS信号；至于能否使用一个典型的信号，是由实现决定的。

RTS信号的符号只有两个——第一个和最后一个，因此必须使用诸如SIGRTMIN、SIGRTMIN+1、…、SIGRTMAX的形式来引用它们。当然，也可以自己定义宏，形式如下：

```
#define DB_IO_DONE SIGRTMIN + 4
```

唉，POSIX.4小组并没有解决两个库都使用相同信号的问题，因此请务必小心。

SIGRTMIN和SIGRTMAX并不必须是常整数表达式，因此，不能将它们轻易地用作case标签或用于预处理程序表达式（如#if）中。

如果不止有一个RTS信号（SIGRTMIN到SIGRTMAX）处于挂起状态，那么编号最小的信号将被最先传递。因此可以认为它们具有优先次序，在把它们分派给不同应用目的的程序时，可能需要将其考虑在内。所有RTS信号的默认动作都是终止。

9.5.3 信号队列

一般来说，如果在同类信号挂起时产生一个信号，其后果是无法预料的。例如，如果实现没有对信号排队的能力，而仅为每个信号保存一个标志，那么结果只会是丢弃第二个信号。决不要将信号用于计数目的，例如：用来记住SIGUSR1信号发送了多少次。

但在使用RTS时，对于一个由RTS产生函数（在9.5节开始处进行了定义）发送的RTS信号（SIGRTMIN到SIGRTMAX），如果使用sigaction对其设置SA_SIGINFO标志，那么信号将会被排入队列。排队是否也对典型信号有效，是随实现而定的，因此不要对它抱什么希望。许多系统甚至在没有设置SA_SIGINFO标志时也对RTS信号进行排队，虽然这也是合法的，但如果要保证可移植性，无论怎样都应设置这一标志。

如果需要排队，但你不又想使用信号处理程序，因为你想使用如sigwait（见9.2.2节）这样的函数，那么也许可以把sigaction结构中的sa_sigaction成员设为SIG_DFL，但SUS对此并没有明确说明。安全起见，可编写一个空的处理程序并将动作指向它。

一个进程的最大排队长度至少是32；可以使用sysconf（见1.1.5节）来找出实际的限制值。

9.5.4 sigqueue

sigqueue——为进程产生信号

```
#include <signal.h>

int sigqueue(
    pid_t pid,           /* process ID */
    int signum,         /* signal */
    const union sigval value /* value */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

在9.5.1节的例子中，已经见到了对sigqueue的使用。前两个参数和kill的相似，只是pid只能是进程ID——它不具有kill的广播能力（如发送给所有的进程）。另外，如果设置了SA_SIGINFO标志，同时它使用了9.5.1节中描述的三参数形式，那么你就可以给信号处理程序传递一个能接受的值。先决定是使用int还是使用该union的指针成员；处理程序需要知道使用了哪种，因为在传递的相关信息里没有包含此类信息。

如果要将信号发送到另一个进程，也许不能使用指针，因为它对另一个进程来说没有任何意义。唯一有效的方法是，如果两个进程共享位于同一位置的存储器，那么指针可指向共享存储器。

如上节所述，如果设置了SA_SIGINFO标志，并且接收进程已经有一个signum信号处于挂起状态，那么新的信号将被排入队列。

你只能对RTS信号（见9.5.2节）进行排队和传递值。

如上节所说，只能对一定数量的信号进行排队。新信号如果不能排入队列，sigqueue将返回-1并将errno设为EAGAIN。下节中还有一个使用sigqueue的示例。

9.5.5 sigwaitinfo和sigtimedwait

9.2.2节中的sigwait不属于RTS，但它也能很好地处理RTS信号，甚至能对它们进行排队操作。它接受某个挂起的信号并返回它；如果还有挂起的同类信号，它们将留在队列中。和信号处理程序类似，如果挂起的RTS信号（SIGRTMIN到SIGRTMAX）多于一个，那么拥有最高优先级（编号最小）的信号将会最先被接受。

但使用sigwait的问题在于不能获得siginfo_t信息，包括值，后者可能是最重要的RTS功能。因此，还有一个稍微加强了的系统调用叫作sigwaitinfo：

sigwaitinfo——等待信号

```
#include <signal.h>

int sigwaitinfo(
    const sigset_t *set, /* signals to wait for */
    siginfo_t *info     /* returned info */
);
/* Returns signal number or -1 on error (sets errno) */
```

如果info是NULL, 那么就得不到返回信息。这时, 除了信号编号是作为函数值返回而不是通过参数返回之外, sigwaitinfo和sigwait完全相同。如果info是非NULL, 将收到一个和9.5.1节中描述的SA_SIGINFO风格的信号处理程序中的结构类似的结构。对于sigwait, 必须保证在set中的信号已被阻塞; 否则, 当其中的某个被传递后, 将发生不可预期的事情。

这里有一个使用排队信号的例子。首先是两个将要使用的RTS信号的定义:

```
#define MYSIG_COUNT SIGRTMIN
#define MYSIG_STOP SIGRTMIN + 1
```

下面是线程用来等待信号的函数, 如果是MYSIG_COUNT, 那么就将其值作为字符串显示, 如果是MYSIG_STOP, 就返回(终止该线程):

```
static void *sig_thread(void *arg)
{
    int signum;
    siginfo_t info;

    do {
        signum = sigwaitinfo((sigset_t *)arg, &info);
        if (signum == MYSIG_COUNT)
            printf("Got MYSIG_COUNT; value: %s\n",
                (char *)info.si_value.sival_ptr);
        else if (signum == MYSIG_STOP) {
            printf("Got MYSIG_STOP; terminating thread\n");
            return (void *)true;
        }
        else
            printf("Got %d\n", signum);
    } while (signum != -1 || errno == EINTR);
    EC_FAIL

EC_CLEANUP_BGN
    EC_FLUSH("sig_thread")
    return (void *)false;
EC_CLEANUP_END
}
```

注意不能使用switch语句, 正如9.5.2节中提到的那样, MYSIG_COUNT和MYSIG_STOP可能是非固定的整数表达式。

以下是main函数:

```
int main(void)
{
    sigset_t set;
    struct sigaction act;
    union sigval value;
    pthread_t tid;

    ec_negl( sigemptyset(&set) )
    ec_negl( sigaddset(&set, MYSIG_COUNT) )
    ec_negl( sigaddset(&set, MYSIG_STOP) )
    ec_rv( pthread_sigmask(SIG_SETMASK, &set, NULL) )
    memset(&act, 0, sizeof(act));
    act.sa_flags = SA_SIGINFO;
    act.sa_sigaction = dummy_handler;
    ec_negl( sigaction(MYSIG_COUNT, &act, NULL) )
    ec_negl( sigaction(MYSIG_STOP, &act, NULL) )
```



```

value.sival_ptr = "One";
ec_negl( sigqueue(getpid(), MYSIG_COUNT, value) )
value.sival_ptr = "Two";
ec_negl( sigqueue(getpid(), MYSIG_COUNT, value) )
value.sival_ptr = "Three";
ec_negl( sigqueue(getpid(), MYSIG_COUNT, value) )
value.sival_ptr = NULL;
ec_negl( sigqueue(getpid(), MYSIG_STOP, value) )
ec_rv( pthread_create(&tid, NULL, sig_thread, &set) )
ec_rv( pthread_join(tid, NULL) )
exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}

static void dummy_handler(int signum, siginfo_t *info, void *context)
{
}

```

请注意，为了演示排队情况，在线程创建之前就已经产生了4个信号。可以保证MYSIG_COUNT信号会在MYSIG_STOP信号之前被接受，因为它们的编号更小。同时，请注意信号被阻塞并且保持该阻塞状态。由于必须设置SA_SIGINFO标志来允许排队，所以需要加入虚拟的处理程序。虽然在Solaris中对RTS信号进行排队时，加入和不加入虚拟处理程序都是一样的，但在可移植编程时不要心存侥幸。

以下是输出，证明信号已经排队：

```

Got MYSIG_COUNT; value: One
Got MYSIG_COUNT; value: Two
Got MYSIG_COUNT; value: Three
Got MYSIG_STOP; terminating thread

```

如果只需要等待有限的时间，可以使用sigtimedwait，它为sigwaitinfo添加了一个超时参数：

sigtimedwait —— 等待信号

```

#include <signal.h>

int sigtimedwait(
    const sigset_t *set,          /* signals to wait for */
    siginfo_t *info,            /* returned info */
    const struct timespec *ts    /* max. time to wait */
);
/* Returns signal number or -1 on error (sets errno) */

```

sigtimedwait在ts指定的最大纳秒时间内等待set中的信号变为挂起状态并返回。如果超时，它将返回-1同时将errno置为EAGAIN。不会得到值为NULL的timeout。如果timespec成员为零，同时set中没有挂起的信号，那么sigtimedwait将立刻返回，但实际上这并不是一个特例——因为它同样超时了。

9.5.6 sigevent结构和SIGEV_THREAD

除sigqueue之外的RTS产生函数在别处都已经描述过了，如9.5.1节列出的，但它们都使用sigevent结构来指定要发送什么信号和什么值。下面专门对sigevent结构进行解释。

struct sigevent——RTS产生函数的结构

```

struct sigevent {
    int sigev_notify;           /* notification type (see below) */
    int sigev_signo;          /* signal number */
    union sigval sigev_value; /* signal value */
    void (*sigev_notify_function)(union sigval); /* thread function */
    pthread_attr_t *sigev_notify_attributes; /* thread attributes */
};

```

要被发送的信号编号(如SIGRTMIN)会被存入sigev_signo中, 伴随的值会被存入sigev_value中。如同其他RTS功能一样, 它仅保证对RTS信号有效。sigev_notify成员指定信号是如何发送的:

SIGEV_SIGNAL 产生一个信号, 就好像已调用了sigqueue
SIGEV_THREAD 开始一个线程, 就好像已调用了pthread_create
SIGEV_NONE 不提供任何通知, 因为通知通常没有多大意义

其中, SIGEV_THREAD是最有趣的。当事件发生时, 它都会启动一个新线程。每次启动新线程时, sigev_notify_function指向的函数都会成为线程的启动函数。通常, 启动函数都有一个指向void的指针参数; 而在本情况下它是一个指向union-sigval的指针参数。这或多或少是一样的, 因为union成员之一就是void指针。可以使用sigev_notify_attributes成员设置线程的属性, 但不能使线程成为可连接的。如果该成员为NULL, 则线程默认会被断开, 这正和pthread_create的作用相反。

创建线程的开销很大, 因此很明显, 最好不要指定SIGEV_THREAD, 除非发生相当罕见的事件和/或事件的产生意味着要进行大量的工作, 并和启动新线程的成本相当。例如, 在每次输入到达时都启动一个新线程可能就有些过分了。

为防止发生混淆, 当以别的方式产生信号时创建线程, 和使线程在sigwait中等待是完全不同的。二者的所有共同点就是它们都使用线程。

9.6 全局跳转

通常, 函数仅在运行return语句时, 或者, 如果是void函数则在运行超出其外部块底部时(和前者同一意思), 才返回到它的调用者。但是, 使用标准C函数setjmp和longjmp, 也可能跳转到程序中任意的(不过是预先计划的)一点:

setjmp——设置跳转点

```

#include <setjmp.h>

int setjmp(
    jmp_buf loc_info /* saved location information */
);
/* Returns 0 if called directly, non-zero if from longjmp (no error
return) */

```

longjmp——跳到跳转点

```

#include <setjmp.h>

void longjmp(
    jmp_buf loc_info, /* saved location information */
    int val           /* value for setjmp to return */
);

```

通过调用`setjmp`，可以标记要跳转到的位置，也可以称为标签，`setjmp`在参数中保存了所有它所需要的位置信息（如当前堆栈指针、机器地址）。虽然其参数看上去不像一个可以接收信息的指针，但它确实是。然后，在任意一个函数中（不管嵌套的有多深），都可以使用相同的参数运行`longjmp`，效果就像是`setjmp`返回了不为零的`val`；如果`val`为零，则`setjmp`返回1。

当调用含有`longjmp`的函数时，可能已经在堆栈中压入了数据，但`longjmp`会自动将它们全部弹出栈。而含有`setjmp`的函数不能被“调用”，并且在返回时不存在递归，即使可能已经去调用`longjmp`了。

下面是一个例子。如果在此之前从未接触过`setjmp`和`longjmp`，可能会认为它非常古怪：

```
static jmp_buf loc_info;

static void fcn2(void)
{
    printf("In fcn2\n");
    longjmp(loc_info, 1234);
    printf("Leaving fcn2\n");
}

static void fcn1(void)
{
    printf("In fcn1\n");
    fcn2();
    printf("Leaving fcn1\n");
}

int main(void)
{
    int rtn;

    rtn = setjmp(loc_info);
    printf("setjmp returned %d\n", rtn);
    if (rtn == 0)
        fcn1();
    printf("Exiting\n");
    exit(EXIT_SUCCESS);
}
```

输出如下：

```
setjmp returned 0
In fcn1
In fcn2
setjmp returned 1234
Exiting
```

可以看到，`setjmp`被调用了一次，却返回了两次，而且从`fcn2`的中部直接跳到了`main`的中部；两个“Leaving”行都未被输出。

如果没有完全理解也不必担心，因为我将要说的是：绝对不要使用`longjmp`。原因是：

- 当堆栈清空后，就没剩下什么了（如分配的内存、打开的文件）。
- 对何处可使用`setjmp`和`longjmp`有一些限制，如果忘记了这些限制，就会带来麻烦。
- 一些人（包括我）不喜欢函数中存在转向（`goto`）语句。他们更不喜欢`longjmp`。它会使得程序非常难于理解和维护。
- `longjmp`主要用于从信号处理程序中跳转，但它不是异步信号安全的，因此除非可以

保证信号处理程序是被异步信号安全的函数调用的，例如在同一个进程内运行kill（见9.1.7节），否则就不要使用它。

如果你认为需要使用longjmp，那么可能的原因是：函数设计得不够好，不能使函数（也许还包括一条错误说明或其他异常情况）返回到调用者。再加把劲，你就会有所发现。

longjmp可能会也可能不会恢复信号屏蔽。如果要强制其恢复，可使用setjmp/longjmp的变体，除了可以保存和恢复信号屏蔽外，它们和原型是相同的：

sigsetjmp —— 设置跳转点

```
#include <setjmp.h>

int sigsetjmp(
    sigjmp_buf loc_info, /* saved location information */
    int savemask         /* non-zero to save signal mask */
);
/* Returns 0 if called directly, non-zero if from siglongjmp (no error
return) */
```

siglongjmp —— 跳到跳转点，如果保存的话恢复信号掩码

```
#include <setjmp.h>

void siglongjmp(
    sigjmp_buf loc_info, /* saved location information */
    int val             /* value for sigsetjmp to return */
);
```

得到恢复的信号屏蔽，是在sigsetjmp被调用时生效的那个屏蔽。如果siglongjmp在信号处理程序中被调用，也即其设计原意，那么信号处理程序要恢复的那个屏蔽和允许正常返回的那个屏蔽可能是不同的。

由于siglongjmp和longjmp一样都不是异步信号安全的，通常不能用于信号处理程序，因此它的恢复信号屏蔽的附加能力几乎毫无用武之地。

如果不需要保存信号屏蔽，同时由于某些原因，你不想只是将sigsetjmp的savemask参数置为零，那么还可以使用另外一对函数：

_setjmp —— 设置跳转点

```
#include <setjmp.h>

int _setjmp(
    jmp_buf loc_info /* saved location information */
);
/* Returns 0 if called directly, non-zero if from longjmp (no error
return) */
```

_longjmp —— 不用恢复信号掩码就跳至跳转点

```
#include <setjmp.h>

void _longjmp(
    jmp_buf loc_info, /* saved location information */
    int val          /* value for setjmp to return */
);
```

当然，_longjmp也不是异步信号安全的。实际上，这两个加下划线的函数只是一个早期标准的遗物。

9.7 时钟和定时器

本节描述了不同的系统时钟及使用这些时钟在预定间隔后产生信号的定时器。

9.7.1 alarm系统调用

alarm —— 调度一个alarm信号

```
#include <unistd.h>

unsigned alarm(
    unsigned secs      /* seconds until signal */
);
/* Returns seconds left on previous alarm or zero if none (no error
return) */
```

每个进程都有一个为alarm系统调用预留的报警时钟。当开始报警时，会发送一个SIGALRM。子进程继承其父进程的报警时钟值，但实际的时钟并不共享。执行exec后，报警时钟仍然保持其设置。

alarm按照secs指定的秒数来设置时钟。返回前一设置；如果时钟上没有以前保留的时间，或者以前没有报过警，则会返回0。前一设置用来将时钟恢复到调用alarm之前的状态。

如果secs为0，则关闭报警时钟。记住这样做是很重要的。举一个例子，假设有一个系统调用正在被堵塞，比如read，只需要阻塞它有限的一段时间。先捕获SIGALRM（使用一个空处理程序），调用alarm（比如5秒），然后发出read（被阻塞的）。当报警开始时，SIGALRM中断被阻塞的系统调用，read返回-1同时将errno置为EINTR。但如果read不到5秒就返回了，而你又忘了关闭报警时钟的话，那么它将会在不久后报警（不，这太晚了，因为从计算机的观点看，5秒是非常漫长的时间），可能会不可思议地中断其他一些什么！如果严格地检查返回的错误，可能会发觉有些事情出了差错，就像本书中所做的那样。但是，唉，不是所有人都这么做。

以下是一个示例：

```
int main(void)
{
    struct sigaction act;
    char buf[100];
    ssize_t rtn;

    memset(&act, 0, sizeof(act));
    act.sa_handler = handler;
    ec_negl( sigaction(SIGALRM, &act, NULL) )
    alarm(5);
    if ((rtn = read(STDIN_FILENO, buf, sizeof(buf) - 1)) == -1) {
        if (errno == EINTR)
            printf("Timed out... type faster next time!\n");
        else
            EC_FAIL
    }
    alarm(0);
    if (rtn == 0)
        printf("Got EOF\n");
    else if (rtn > 0) {
        buf[rtn] = '\0';
        printf("Got %s", buf);
    }
}
```

```

    }
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
static void handler(int signum)
{
}

```

在下面的样例交互中，在初次运行`alarm_test`后，我没有输入任何东西；在其第二次运行时，我输入了“faster”：

```

$ alarm_test
Timed out... type faster next time!
$ alarm_test
faster
Got faster
$

```

在简单的情况下，使用`alarm`中断阻塞的系统调用是不错的，就像上文展示的那样。但在更复杂的状况下，可能要在`select`或`poll`中阻塞，而不是直接在`read`中阻塞。如要超时，应使用`pselect`（如果可用的话）代替设定报警时钟。

`alarm`的一个局限性是每个进程只能有一个这样的报警时钟。在9.7.4节和9.7.6节中描述的`timer`系统调用拥有更高的灵活性。

9.7.2 sleep系统调用

在本书中，`sleep`无处不在，我们对它已经很熟悉了。它可以在特定的秒数内阻塞一个线程：

sleep——挂起执行几秒钟或等待信号

```

#include <unistd.h>

unsigned sleep(
    unsigned secs          /* seconds to sleep */
);
/* Returns unslept amount */

```

在SUS中，对允许`SIGALRM`与`sleep`交互的规则着墨很多，明确说明可以使用`alarm`和`pause`，或更好地，使用`alarm`和`sigsuspend`来把`sleep`实现成函数。以下是实现`sleep`的一种简单方法：

```

unsigned aup_sleep(unsigned secs)
{
    struct sigaction act;
    unsigned unslept;

    memset(&act, 0, sizeof(act));
    act.sa_handler = slp_handler;
    ec_negl( sigaction(SIGALRM, &act, NULL) )
    alarm(secs);
    pause();
    unslept = alarm(0);
    return unslept;
}

```

```

EC_CLEANUP_BGN
    EC_FLUSH("aup_sleep")
    return 0;
EC_CLEANUP_END
}

static void slp_handler(int signum)
{
}

```

虽然这个版本可以运行，但还存在一些问题：

- 如果SIGALRM没有被阻塞，那么若它在调用sigaction之前到达可能会终止进程。
- 如果它们都被阻塞，那么函数可能就无法工作了。
- 如果在调用alarm和pause之间报警开始，那么等待将永远继续下去。
- 没有恢复SIGALRM的原有动作。
- alarm和sleep是假设兼容的。

最后一点意味着在一个如下的序列中

```

alarm(10);
...
sleep(20);

```

报警开始时，应当中断休眠，并运行SIGALRM的处理程序。在如下的序列中

```

alarm(20);
...
sleep(10);

```

报警时间应设为在sleep返回后的10秒之后。

处理alarm和sleep的交互，阻止某个错误的SIGALRM终止进程，阻止无穷尽的等待以及恢复原先的动作和屏蔽，这些都要求编写大量的技巧性代码来处理以下三种可能的情况：

- 当sleep被调用时，没有设置alarm。
- alarm的剩余时间小于等于所要求的sleep时间。
- alarm的剩余时间所要求的sleep时间要长。

下面的这个版本就解决了上述问题：^①

```

unsigned aup_sleep(unsigned secs)
{
    sigset_t set, oset;
    struct sigaction act, oact;
    unsigned prev_alarm, slept, unslept, effective_secs;

    ec_negl( sigemptyset(&set) )
    ec_negl( sigaddset(&set, SIGALRM) )
    ec_negl( sigprocmask(SIG_BLOCK, &set, &oset) )
    prev_alarm = alarm(0);
    if (prev_alarm != 0 && prev_alarm <= secs)
        effective_secs = prev_alarm;
    else {
        memset(&act, 0, sizeof(act));
        act.sa_handler = slp_handler;
        ec_negl( sigaction(SIGALRM, &act, &oact) )
        effective_secs = secs;
    }
}

```

① 在此要感谢Geoff Clare，他在我最初尝试的版本中找出了bug，从而有了现在这个版本。

```

    alarm(effective_secs);
    set = oset;
    ec_negl( sigdelset(&set, SIGALRM) )
    if (sigsuspend(&set) == -1 && errno != EINTR)
        EC_FAIL
    unslept = alarm(0);
    slept = effective_secs - unslept;
    ec_negl( sigaction(SIGALRM, &oact, NULL) )
    if (prev_alarm > slept)
        alarm(prev_alarm - slept);
    ec_negl( sigprocmask(SIG_SETMASK, &oset, NULL) )
    return unslept;

EC_CLEANUP_BGN
    EC_FLUSH("aup_sleep")
    return 0;
EC_CLEANUP_END
}

static void slp_handler(int signum)
{
}

```

以下是程序执行的步骤:

1) 我们阻塞了SIGALRM, 因此操作时无需担心会有信号被传递。将原先的信号屏蔽存储在oset中。

2) 如果设置了报警时钟, 则将其关闭, 并保存剩余时间。

3) 计算了休眠时间的长短 (effective_secs), 如果剩余的报警时间少, 则将要求的数量减少, 并将报警时钟设置到那个时间。

4) 如果事先没有设置报警, 或休眠时间比剩余的报警时间要少, 则安装一个空处理程序并将原有的动作保存在oact里。

5) 调用sigsuspend代替pause, 以便可以自动取消对SIGALRM的阻塞。对屏蔽中的其他位, 保持其进入aup_sleep时的原有状态。

6) 如果sigsuspend返回, 那么可能是因为报警开始了, 或者是因为有某个其他信号中断了它。我们不必特别注意是哪个原因。但是, 我们确实需要报警时钟剩余的时间数量, 尽管在关闭它的时候就可以获得该值。

7) 计算实际休眠时间。

8) 复位SIGALRM的原有动作。

9) 如果休眠时间比进入时剩余的报警时间要少, 那么按新的剩余时间来重新设置报警时钟 (一些时间已经休眠过去了)。

10) 复位信号屏蔽。

11) 返回未休眠的时间。

如果你能理解这个函数, 那么你就已经掌握了本章中90%的内容, 可以给自己的学习成绩打一个A。如果你发现了书中的错误并将它发送到aup@basepath.com, 那么就可以给自己一个A+了。

9.7.3 更高精度的休眠

sleep会休眠数秒, 但对许多用途来说, 这个时间太长了。另外还有两种调用能以更精

确的时间间隔进行休眠:

```
usleep——挂起执行几微秒或等待信号

#include <unistd.h>

int usleep(
    useconds_t usecs          /* microseconds to sleep */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

除了用它休眠的时间达到或超出1秒时会出现错误以外，`usleep`和`sleep`几乎完全一样。也就是说，`usecs`必须小于100万。如果知道已经使用了定时器选项（`_POSIX_TIMERS`），则可以使用更加精确而且没有上述限制的`nanosleep`：

```
nanosleep——挂起执行几纳秒或等待信号

#include <time.h>

int nanosleep(
    const struct timespec *nsecs, /* nanoseconds to sleep */
    struct timespec *remain      /* remaining time or NULL */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

`nanosleep`有一个已在1.7.2节中定义过的`timespec`结构。回想可知，它有一个对应于秒的`tv_sec`成员和一个对应于纳秒的`tv_nsec`成员。

和平常一样，如果它被中断，它将返回-1同时将`errno`置为`EINTR`，不过它还会将`remain`指向的内容设为剩余的时间。如果它返回零或如果`remain`是`NULL`，那么它不会通过参数返回任何信息。

在9.7.5节中我们还会介绍一种叫作`clock_nanosleep`的休眠函数。

9.7.4 基本间隔定时器系统调用

在所有UNIX版本中都存在的`alarm`系统调用（见9.7.1节）使用的是一个整个进程范围的间隔定时器。所有与SUS兼容的系统，以及一些其他系统，还具有其他三个可以独立使用的间隔定时器：

`ITIMER_REAL` 实际时间减1；超时的时候产生一个`SIGALRM`。

`ITIMER_VIRTUAL` 进程虚拟时间减1；超时的时候产生一个`SIGVTALRM`。

`ITIMER_PROF` 进程虚拟时间减1并且代表该进程运行的系统时间减1；超时的时候产生一个`SIGPROF`。它有意这样设计以便配置者使用来调整程序，以指明这些程序在何处花费了时间。

在以下两个系统调用中，`which`参数必须是以上列出的三个宏之一：

```
getitimer——得到间隔定时器的值

#include <sys/time.h>

int getitimer(
    int which,          /* timer to get */
    struct itimerval *val /* returned value */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

setitimer ——设置间隔定时器的值

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

int setitimer(
    int which,          /* timer to set */
    const struct itimerval *val, /* value to set */
    struct itimerval *oval /* returned old value */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

struct itimerval ——getitimer和setitimer的结构

```
struct itimerval {
    struct timeval it_interval; /* reset value */
    struct timeval it_value;    /* current value */
};
```

在itimerval结构中，it_interval是当定时器超时后应重设成的那个时间，it_value是当前时间间隔的值。也就是说，和只能报警一次的alarm不同，这些定时器可以自动地重新设置自己何时报警。如果在it_value成员置为零时调用setitimer，会立即停止定时器。如果在it_interval成员置为零时调用setitimer，将会在当前间隔超时后停止定时器。1.7.1节中定义的timeval结构有两个成员：对应于秒的tv_sec成员和对应于微秒的tv_usec成员。

如果setitimer的oval参数是非NULL的，那么它将获取原来的值。下面是一些例子：

```
/* 3.5 sec. timer, one time only */
itv.it_interval.tv_sec = 0;
itv.it_interval.tv_usec = 0;
itv.it_value.tv_sec = 3;
itv.it_value.tv_usec = 500000;
ec_negl( setitimer(ITIMER_REAL, &itv, NULL) )

/* 3.5 sec. timer, then repeating 2.25 sec. timers */
itv.it_interval.tv_sec = 2;
itv.it_interval.tv_usec = 250000;
itv.it_value.tv_sec = 3;
itv.it_value.tv_usec = 500000;
ec_negl( setitimer(ITIMER_REAL, &itv, NULL) )

/* stop timer immediately; interval doesn't matter */
itv.it_value.tv_sec = 0;
itv.it_value.tv_usec = 0;
ec_negl( setitimer(ITIMER_REAL, &itv, NULL) )
```

例子中，定时器将在3.5秒时执行，然后在5.7秒时再次执行，再然后是在8秒时执行，依次类推，每隔2.25秒执行一次直到被停止。

下面的例子每隔2秒真实（“挂钟”）时间显示一个X。注意一旦设置了定时器，程序的其他部分就可以自由地进行自己的工作，在本例中只是从终端读取并回显读入的内容：

```
void timer_try1(void)
{
    struct sigaction act;
    struct itimerval itv;
    char buf[100];
    ssize_t nread;

    memset(&act, 0, sizeof(act));
    act.sa_handler = handler;
```

```

ec_negl( sigaction(SIGALRM, &act, NULL) )
memset(&itv, 0, sizeof(itv));
itv.it_interval.tv_sec = 2;
itv.it_value.tv_sec = 2;
ec_negl( setitimer(ITIMER_REAL, &itv, NULL) )
while (true) {
    switch( nread = read(STDIN_FILENO, buf, sizeof(buf) - 1) ) {
        case -1:
            EC_FAIL
        case 0:
            printf("EOF\n");
            break;
        default:
            if (nread > 0)
                buf[nread] = '\0';
            ec_negl( write(STDOUT_FILENO, buf, strlen(buf)) )
            continue;
    }
    break;
}
return;

EC_CLEANUP_BGN
EC_FLUSH("timer_try1")
EC_CLEANUP_END
}

void handler(int signum)
{
    write(STDOUT_FILENO, "\nX\n", 3);
}

```

其输出如下:

```

X
ERROR: 0: tml [/aup/c9/tmr.c:26] 0
*** EINTR (4: "Interrupted system call") ***

```

发生的情况是: 2秒过后, 第一个X可以正常显示, 但是SIGALRM信号中断了已经被阻塞的read。解决办法是设置SA_RESTART标志, 这样系统调用就不会被中断了:

```

memset(&act, 0, sizeof(act));
act.sa_handler = handler;
act.sa_flags = SA_RESTART;
ec_negl( sigaction(SIGALRM, &act, NULL) )

```

现在输出如下, 可以看到输入了一个“hello”然后回显了出来, 同时每隔2秒显示一个X:

```

X

X
hello
X

hello

X
...

```

还有一个被称为ualarm的间隔定时器系统调用, 但现在已经不使用了。

9.7.5 实时时钟

所有的UNIX系统都有一个基本时钟，可以使用time和gettimeofday系统调用（见1.7.1节）来读取它的值。但提供nanosleep（见9.7.3节）的那个定时器选项（_POSIX_TIMERS）还包括了一个或更多的附加时钟，其数目取决于软件和硬件所支持的数量。要访问它们中的一个，可以通过其时钟ID来引用它。

所有带定时器选项的系统都支持一个ID——CLOCK_REALTIME，它保存了当天的时间记录。是否还有其他的时钟，它们的宏是什么，以及它们是单个系统范围的还是单个进程范围的，这些都和具体实现相关。例如，Solaris还支持一个单个系统范围的CLOCK_HIGHRES时钟，它从过去某个时间点开始计数。读取它并不会告诉你实时时间，因为你不知道它是怎样设置的，但它可以用于比较两个不同的时间。

调用clock_gettime可以在返回的timespec结构（见1.7.2节和9.7.3节）中获得时间的纳秒形式：

clock_gettime——从时钟中获得时间

```
#include <time.h>

int clock_gettime(
    clockid_t clock_id,      /* clock ID (CLOCK_REALTIME, etc.) */
    struct timespec *tp     /* time */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

使用clock_getres可以获得一个纳秒精度的时钟：

clock_getres——得到时钟精度

```
#include <time.h>

int clock_getres(
    clockid_t clock_id,     /* clock ID */
    struct timespec *res    /* resolution */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

同时，如果拥有适当的特权（对CLOCK_REALTIME来说，需要超级用户特权），还可以设置一个时钟：

clock_settime——设置时钟

```
#include <time.h>

int clock_settime(
    clockid_t clock_id,     /* clock ID */
    const struct timespec *tp /* time */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

下面是一个函数，它从某个时钟获取时间和精度，然后与使用time系统调用（见1.7.1节）得到的秒形式的时间一起显示出来：

```
void clocks(void)
{
    struct timespec ts;
    time_t tm;
    ec_negl( time(&tm) )
```

```
printf("time() Time: %ld secs.\n", (long)tm);
printf("CLOCK_REALTIME:\n");
ec_neg1( clock_gettime(CLOCK_REALTIME, &ts) )
printf("Time: %ld.%09ld secs.\n", (long)ts.tv_sec, (long)ts.tv_nsec);
ec_neg1( clock_getres(CLOCK_REALTIME, &ts) )
printf("Res.: %ld.%09ld secs.\n", (long)ts.tv_sec, (long)ts.tv_nsec);
return;
```

```
EC_CLEANUP_BGN
    EC_FLUSH("clocks")
EC_CLEANUP_END
}
```

在FreeBSD系统上运行得到的输出如下:

```
time() Time: 1051646878 secs.
CLOCK_REALTIME:
Time: 1051646878.568628061 secs.
Res.: 0.000000838 secs.
```

这表明它的精度刚好是微秒级的。换用不同的硬件，在Solaris系统的输出如下:

```
time() Time: 1051646409 secs.
CLOCK_REALTIME:
Time: 1051646409.686869683 secs.
Res.: 0.010000000 secs.
```

这里的精度仅仅是百分之一秒级的。把时间表示为小数点右边有9位数字的形式是一种误导——将显示形式限制为其精度能达到的形式才是一种更好的做法。然而，这些时钟并不是用来显示时间的，而是用来计时的。

以下是用于特定时钟的另一个nanosleep（见9.7.3节）版本:

clock_nanosleep ——挂起几纳秒，或等待某个信号

```
#include <time.h>

int clock_nanosleep(
    clockid_t clock_id,          /* clock ID */
    int flags,                  /* TIMER_ABSTIME or zero */
    const struct timespec *nsecs, /* nanoseconds to sleep */
    struct timespec *remain      /* remaining time or NULL */
);
/* Returns 0 on success or error number on error */
```

第一个参数是时钟ID；最后两个参数和nanosleep的相同。如果flag为零，将会按指定数量的纳秒时间间隔进行休眠，就像使用nanosleep一样。但如果flag为TIMER_ABSTIME，则在nsec指定的绝对时间来临之前会一直休眠。最后一个参数是用来返回剩余时间的，它仅用于相对休眠；也就是说，在flag为零的时候。

最后，可以通过进程CPU时间时钟选项（_POSIX_CPUTIME）获得某个进程CPU时间时钟的时钟ID:

clock_getcpuclockid ——得到进程CPU时钟

```
#include <time.h>

int clock_getcpuclockid(
    pid_t pid,                  /* process ID */
    clockid_t *clock_id        /* returned clock ID */
);
/* Returns 0 on success or error number on error */
```

9.7.6 高级间隔定时器系统调用

就像实时时钟胜过基本时钟那样，高级间隔定时器也胜过在9.7.4节中讨论过的间隔定时器。它们并非仅仅是几个整个系统范围的定时器，调用它们可以为每个进程设置几个都基于同一时钟的间隔定时器；回想上一节可知，唯一能保证的时钟只有CLOCK_REALTIME。

开始先创建一个定时器：

timer_create —— 建立每个进程定时器

```
#include <signal.h>
#include <time.h>

int timer_create(
    clockid_t clockid,          /* clock ID */
    struct sigevent *sig,      /* NULL or signal to generate */
    timer_t *timer_id         /* returned timer ID */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

如果成功了，timer_create将通过timer_id参数返回一个新的定时器ID。如果sig是非NULL的，则在定时器超时后，sig会产生的一个信号、一个值并指定信号的传递方式，就像在9.5.6节中解释的那样。如果sig是NULL，则会产生一个SIGALRM，同时把值设为定时器ID。

使用timer_delete来删除一个定时器：

timer_delete —— 删除每个进程定时器

```
#include <time.h>

int timer_delete(
    timer_t timer_id          /* timer ID */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

下面介绍的两个与getitimer及setitimer相似的调用，只是它们存储在itimerspec结构中的精度是纳秒级而不是毫秒级的：

timer_gettime —— 得到每个进程定时器的值

```
#include <time.h>

int timer_gettime(
    timer_t timer_id,          /* timer ID */
    struct itimerspec *val     /* returned value */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

timer_settime —— 设置每个进程定时器的值

```
#include <time.h>

int timer_settime(
    timer_t timer_id,          /* timer ID */
    int flags,                 /* flags */
    const struct itimerspec *val, /* value to set */
    struct itimerspec *oval     /* returned old value or NULL */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

struct itimerspec——定时器函数的结构

```

struct itimerspec {
    struct timespec it_interval; /* reset value */
    struct timespec it_value; /* current value */
};

```

如果timer_settime的flag参数为零，则函数的行为与setitimer就十分相似了。但如果设置了TIMER_ABSTIME标志，对it_value时间的解释将类似于clock_nanosleep中对nsecs的解释：作为定时器超时的绝对时间。（也就是说，像你床头旁的闹钟一样。）在定时器超时后，还可以使用it_interval成员对其进行重新设置。

定时器超时后只能有一个信号被加入到队列中，同时如果it_interval成员足够小，那么在产生信号到接受信号的时间段内，定时器有可能会发生几次超时。在这种情况下，可以通过以下系统调用获得额外超时（超限运行）的数目：

timer_getoverrun——得到每个进程定时器消耗的时间

```

#include <time.h>

int timer_getoverrun(
    timer_t timer_id /* timer ID */
);
/* Returns overrun count or -1 on error (sets errno) */

```

练习

- 研究一下在你的系统中，除了9.1.3节中列出的28种标准信号外，还实现了哪些信号。
- 编写一个在调用write时产生SIGPIPE信号（提示：使用管道）的程序。然后对其进行修改，使之忽略SIGPIPE，改而验证write是否返回了一个写入错误。这时errno的值是多少？代表什么意思？
- 使用sigwait和其他需要的系统调用来实现pause。
- 尝试使用sigaction编写signal（见9.4节）。对付处理程序是需要技巧的——看看你能否找出一种方法。
- 使用sigaction编写sigset（见9.4节）。
- 编写一个当信号到达时执行全局跳转的信号处理函数（见9.6节），并演示在使用setjmp或sigsetjmp之后其运行可以正常继续。为什么longjmp和siglongjmp不在异步信号安全函数的列表（见9.1.7节）中？
- longjmp和siglongjmp只能清理堆栈；它们不能处理由malloc和realloc动态分配的内存。请对其衍生的问题进行讨论。有无可能的自动解决方案？这样做值得吗？请提出一种可行的能处理该问题的设计建议，可以使用setjmp、sigsetjmp、longjmp、siglongjmp、malloc、realloc以及free。
- 使用usleep编写sleep，请遵从usleep休眠必须少于一秒钟的限制。也就是说，你必须找出一种方法来休眠更长时间，比如说3秒。
- 从本章和先前章节中所介绍的系统调用里选出你需要的系统调用（例如间隔定时器调用），编写一个名为alarmclock的命令，以一个时间和一条消息作为参数，然后在那个时间到达时，在标准输出中显示那条消息并响铃。该命令应当使用内核设施来等待那个时间，而不用设置任何东西，它应该在

时间到达时才运行。将它设计为在后台运行，即使用户忘记了以&号结尾。

- 9.10 解释一下（无需编写代码）你会怎样加强上一个练习中的alarmclock命令，使之甚至可以在系统崩溃的时候也能记住其设置。你还需要找出如何使后台进程重新启动的办法。是否已经存在一个UNIX命令，可以完成类似的工作？
- 9.11 扩充你在练习5.14中编写的程序，使之包括那些在本章提到的附录A中的进程属性。



附录A 进程属性

本附录中的表列出了由fork和exec影响到的进程属性，并指出了在本书中何处讨论过。注意，下述几个不是进程属性，正如第7章所讲它们独立于任何进程。但是与文件不同，它们的生存时间与活动的系统一样，重启就会丢失。

POSIX消息队列（见7.7节）。

POSIX命名信号量（见7.10节）。

POSIX共享内存段（见7.14节）。

System V消息队列ID（见7.5节）。

System V消息队列（见7.5节）。

System V信号量设置ID（见7.9节）。

System V信号量设置（见7.9节）。

System V共享内存段ID（见7.13节）。

System V共享内存段（见7.13节）。

表A-1 进程属性

属性	fork	exec	所在章节
异步I/O操作	不被复制	没有通知即取消或完成	3.9
注册的atexit函数	复制	未注册	1.3.4
控制终端	复制	保留	4.3
目录流定位	共享	不可访问	3.6.1
目录流	复制	关闭	3.6.1
当前目录	复制	保留	3.6.2
根目录	复制	保留	5.14
退出状态 [⊖]	-	-	5.7
文件描述	共享	无改变	2.2
文件描述符	复制	除FD_CLOEXEC设置外保留	2.2
文件锁	不被复制	除关闭fd外，保留	7.11
文件方式创建屏蔽（字）	复制	保留	2.4.2
浮点环境 [⊖]	复制	默认	-
有效组ID	复制	取决于set-group-ID位	5.12
有效用户ID	复制	取决于set-user-ID位	5.12
父进程ID	变化	保留	5.13
进程ID	新增	保留	5.13
进程组ID	复制	保留	4.3
实际组ID	复制	保留	5.12
实际用户ID	复制	保留	5.12
保存的set-group-ID	复制	取决于set-group-ID位	5.12

(续)

属性	fork	exec	所在章节
保存的set-user-ID	复制	取决于set-user-ID位	5.12
线程ID	复制	丢失	5.17.1
备用组ID	复制	保留	1.1.8
资源限制(包括文件大小)	复制	保留	5.16
内存, 数据段	复制	新	1.1.5
内存, 数据段, 参数	复制	新	5.3
内存, 数据段, 环境	复制	新	5.2
内存, 指令段	复制	新	1.1.5
内存, 锁 [Ⓞ]	不被复制	删除	-
内存, 映射 (POSIX和System V)	复制	无映射	7.12
内存, 栈(线程)	复制	新	5.17
消息队列描述 (POSIX)	共享	无影响	7.5
消息队列描述符 (POSIX)	复制	关闭	7.5
管道 (FIFO) [Ⓞ]	复制	保留	7.2
管道(未命名)	复制	保留	6.2
调度, nice值	复制	保留	5.15
调度, SCHED_FIFO和SCHED_RR策略 [Ⓞ]	复制	保留	-
信号量, 指向命名的指针 (POSIX)	复制	关闭	7.10
信号量, 内存 (POSIX)	复制	关闭	7.10
信号量, semadj值(SystemV)	清零	保留	7.9
会话成员资格	复制	保留	4.3
信号动作	复制	保留, 除非改变为默认 (SIGCHLD例外)	9.1.6
信号备用栈	复制	丢失; 清除所有的SA_ONSTACK标志	9.3
信号屏蔽(字)(线程)	复制	保留	9.1.5
信号挂起	不被复制	保留	9.1.2
套接字	复制	保留	8.1
线程属性, 优先级等	复制	丢失	5.17
线程栅栏(线程)	复制	丢失	5.17.3
线程条件变量(线程)	复制	丢失	5.17.4
线程互斥(线程)	复制	丢失	5.17.3
线程读写锁(线程)	复制	丢失	5.17.3
线程旋转锁(线程)	复制	丢失	5.17.3
线程(线程)	一个复制	终止	5.17
专用于线程的数据(线程)	复制	丢失	5.17.1
CPU时间	清零	保留	1.7.2
时间, CPU时钟	清零	保留	9.7.5
时间, 线程的CPU时间时钟 [Ⓞ]	清零	清零	5.17.1
定时器, 报警	取消或清零	保留	9.7.1
定时器, 间隔	重新设置	保留	9.7.4
定时器, 每进程	没有复制	删除	9.7.6
跟踪属性	依赖 [Ⓞ]	保留(大部分)	1.1.10

- ⊙ 进程退出时才设置。
- ⊙ 浮点环境指浮点状态标志和控制方式。
- ⊙ 用mlock和munlock加锁和解锁；内存范围内的锁（Range Memory Locking）选项部分。
- ⊙ FIFO本身和其他文件一样有持续性，但是如果没有进程打开它读取数据的话，FIFO中的数据就会消失。
- ⊙ 用sched_setscheduler设置；见进程调度选项部分。
- ⊙ 用pthread_getcpuclockid设置；见CPU时间时钟线程选项部分。
- ⊙ 依赖跟踪流的继承策略。



附录B Ux: 一个对标准UNIX函数进行包装的程序

我对整本书一直不满意的是标准UNIX API的错误处理不一致、令人混淆的命名和代码冗余。正如在1.4.3节所述, C++异常处理提供了一种比较好的处理错误的方法。只要这么做, 就可以把附录D中所列的大部分函数重新命名并把它们组织成类。

我尝试做的这个C++包装程序叫作Ux。它的设计要达到如下目标:

- 1) 对所有函数都采用100%一致的错误处理机制。
- 2) 被包装的函数100%地功能化。
- 3) 组织成反映UNIX体系的对象。
- 4) 删除冗余、过时和有缺陷的函数(如`readdir`、`signal`、`mktemp`), 用其他函数替代。
- 5) 尽可能接近原C接口的速度。

为达到上述目标, 在设计Ux时遵循了如下规则:

1) 处在标准UNIX规范v3层次上, 即没有更高层次的语义, 但语义统一(除了错误处理外)。也就是说与I/O流或Boost线程[⊖]不同, 更没有ACE[⊖]那么详尽的软件包。但包括标准没有的附加函数(如`setblock`、`find_and_open_master`)。

2) 100%地实现SUS中1108个接口中的290个左右的接口。没有实现的那些接口是标准的C函数、POSIX线程、如`bsearch`这种一般的用户函数, 账户管理, 进程复制, 跟踪, 过时的函数和其他一些函数。

3) 所有抛出的错误都没有返回。被抛出的对象是一种包含`errno`和其他错误代码的类型(例如来自`getaddrinfo`)。

4) 只有在“`alloc`”字出现在成员函数名中时, 内存才被动态分配, 但这样的实例很少。

5) 除了调用成员函数的开销以外, 空间效率和时间效率都十分接近于原始的C SUS。

6) 对象反映了UNIX和SUS对象(如文件、目录、目录流、进程)的组织。即Ux实现的是UNIX的抽象而不是引入不同的抽象。

7) 在定义重入函数和非重入函数的地方(如`readdir`和`readdir_r`)通常使用重入版本, 并且缓冲区在对象中。如果有必要, 在对象中可以使用“`alloc`”成员函数分配空间(见第4条)。

8) 对SUS函数提供的数据没有额外的错误检查。例如, 如果`NULL`指针被传递给`File::open`, 那么它会被直接传递给底层的`open`。原因见第5条。

9) 自动选择, 例如, 如果对象是打开的, 那么自动选择`fchown`, 而不是`chown`。但对于它们只需调用成员函数`File::chown`。

10) 在一些情况下能自动检测是否发生了错误。如, 对于`pathconf`, 如果改变了`errno`, 那么返回值-1就表明发生了错误。

⊖ 一个C++线程库 (www.boost.org)。

⊖ ADAPTIVE通信环境 (www.cs.wustl.edu/~schmidt/ACE.html)。

11) 一些函数结合使用了可选参数。例如, read/pread、write/pwrite、fsync/fdatasync。

12) 除非默认, 否则不复制构造函数和赋值运算符。部分遵循第4条。

以下是Ux类的层次结构:

```
Ux::Base
    Ux::Aio
    Ux::Clock
    Ux::DirStream

    Ux::ExitStatus
    Ux::File
        Ux::Dir
        Ux::PosixShm
        Ux::Socket
        Ux::Terminal
        Ux::Pty

    Ux::Netdb
    Ux::PosixMsg
    Ux::PosixSem
    Ux::Process
    Ux::Sigset
    Ux::SockAddr
        Ux::SockAddrIn
        Ux::SockAddrUn
    Ux::SockIPv4
    Ux::SockIPv6
    Ux::System
    Ux::SysVIPC
        Ux::SysVMsg
        Ux::SysVSem
        Ux::SysVShm
    Ux::Termios
    Ux::TimeMsec
    Ux::TimeNsec
    Ux::TimeParts
    Ux::Timer
        Ux::IntervalTimer
        Ux::RealtimeTimer
    Ux::TimeSec
    Ux::TimeStr
    Ux::TimeString
    Ux::Timet
    Ux::Timetm
    Ux::Timeval

Ux::Error
mq_attr
Ux::semun
```

详细内容请到网站www.basepath.com/aup上查询, 那里有完整的文档和源文件。尽管Ux看上去很有前途, 但只把它用在了一些细小的例子上, 因此本书的许多读者仍未对其进行评论。如你有什么想法, 可电邮到aup@basepath.com。

附录C Jtux: 标准UNIX函数的 Java/Jython接口

30年前, UNIX刚开始, 几乎所有的UNIX应用程序都是用C写的, 目前因为主要的UNIX标准用的仍然是C, 所以C仍是“UNIX语言”。因为UNIX内核是用C写的, 所以几乎所有的UNIX编程书(如本书)都用C写示例。现在有许多(如果不占大多数的话)UNIX程序已不用C写了, 而是用其他语言写的, 如: C++、Java、Perl、Python、Jython、[⊖]PHP等等。

C++是一个特殊案例, 因为你能直接使用C接口或使用像附录B中Ux那样的小的C++类库。其他语言主要提供了POSIX 1990描述的经典UNIX系统调用的接口, 没有提供如getaddrinfo或msgsnd这样的新调用。因此想学UNIX的那些严谨的学生们被迫去学C或C++, 即使他们所学的其他课程都用Java, 而且开发者们也不准备给他们提供Java所需要的系统调用。

为了解决Java的这些问题, 我写了Java-to-UNIX (Jtux)。Jtux的目标是提供:

- 用Java或Jython学习UNIX编程的一个教学工具。
- 对目前Java包不支持的UNIX特性进行访问。
- 一致的错误处理。来自系统调用的一个错误, 不管它被表示成什么(返回-1、返回NULL、返回错误代码等)都抛出一个异常——UErrorException。

Jtux不能做到:

- 提供一个面向对象的UNIX API观点。
- 修正UNIX系统调用命名或参数方面的一致性, 或像附录B中的Ux所做的那样清理大部分的冗余代码。实际上, 几乎每一个Jtux方法都有和POSIX/SUS调用相应的相同命名和相同参数, 否则Jtux就不能帮助学生编写UNIX。(Jtux修正了错误处理的不一致。)
- 提供不同于POSIX/SUS系统中已提供的可移植性。Jtux无法在没安装POSIX/SUS系统的Java/Jython环境中运行, 例如Windows。
- 在applet上运行, 因为用C API连接Java的Java本地接口(Java Native Interface, JNI)不能工作。

针对UNIX, 若想使用面向对象的方法(因其更符合Java的精神), 最好使用其他的Java包。例如使用java.net.Socket而不是jtux.UNetwork。但是在广意上, 对Java或Jython来说Jtux都是最完整的UNIX包。

Jtux包括以下186个系统调用:

abort	getppid	poll	setpgid
accept	getrlimit	pread	setrlimit

[⊖] Jython是一种Python的实现, 是用Java写的, 运行在Java环境中, 因此许多Java类能被直接访问。例如在Jython中你可以用AWT或Swing写你的用户接口, 而不用Tkinter, 那就是为什么Jtux与Java和Jython都兼容的原因。更多关于Jython的内容见www.jython.org。

access	getrusage	pselect	setuid
alarm	getsid	pthread_sigmask	setsockopt
bind	getsockopt	putenv	setuid
chdir	getuid	pwrite	shm_open
chmod	htonl	read	shm_unlink
chown	htons	readdir	shmat
chroot	inet_ntop	readlink	shmctl
clock	inet_pton	readv	shmdt
close	kill	recv	shmget
closedir	lchown	recvfrom	sigaction
connect	link	recvmmsg	sigaddset
creat	listen	rename	sigaltstack
dup	lockf	rewinddir	sigdelset
dup2	lseek	rmdir	sigemptyset
execvp	lstat	S_ISBLK	sigfillset
_exit	mkdir	S_ISCHR	siginterrupt
exit	mkfifo	S_ISDIR	sigismember
fchdir	mknod	S_ISFIFO	sigpending
fchmod	mkstemp	S_ISLNK	sigprocmask
fchown	mmap	S_ISREG	sigqueue
fcntl	mq_close	S_ISSOCK	sigsuspend
FD_CLR	mq_getattr	seekdir	sigtimedwait
FD_ISSET	mq_notify	select	sigwait
FD_SET	mq_open	sem_close	sigwaitinfo
FD_ZERO	mq_open	sem_destroy	sleep
fdatasync	mq_receive	sem_getvalue	socketmark
fork	mq_send	sem_init	socket
freeaddrinfo	mq_setattr	sem_open	stat

fstat	mq_timedreceive	sem_open	statvfs
fstatvfs	mq_timedsend	sem_post	symlink
fsync	mq_unlink	sem_timedwait	sync
ftok	msgctl	sem_trywait	system
ftruncate	msgget	sem_unlink	telldir
gai_strerror	msgrcv	sem_wait	times
getaddrinfo	msgsnd	semctl	truncate
getcwd	munmap	semget	umask
getegid	nanosleep	semop	unlink
getenv	nice	send	unsetenv
geteuid	ntohl	sendmsg	usleep
getgid	ntohs	sendto	utime
gethostid	open	setegid	wait
gethostname	open	setenv	waitpid
getnameinfo	opendir	seteuid	write
getpgid	pause	setgid	writev
getpid	pipe		

这里所列的就是POSIX/SUS用来处理如下内容的所有调用：文件、文件系统、目录、套接字、管道、进程、System V消息/信号量/共享内存、POSIX消息/信号量/共享内存、信号、环境等。完全地包含这些调用很费力。例如，不仅有write，也有pwrite和writev；不仅有send，也用sendmsg和sendto。此外，技术上不可能做到支持所列函数的每个参数和每个结构中的每个域变量。对于终端I/O、伪终端、内部定时器（除了alarm）和实时时钟函数已经做了，但未列出，你可以把它们加入。

Jython是一个非常好的实验UNIX系统调用的系统。为了说明Jython和Jtux是什么样的系统，下面给出一个程序，它有两个使用数据报通信（见8.6节）的进程：

```
# Jython example showing use of AF_INET sockets with SOCK_DGRAMs.
# System calls used: bind, close, _exit, fork, getaddrinfo,
# recvfrom, sendto, setsockopt, sleep, socket, waitpid
import jtux.UClock as UClock
import jtux.UConstant as UConstant
import jtux.UErrorException as UErrorException
import jtux.UFile as UFile
import jtux.UNetwork as UNetwork
import jtux.UProcess as UProcess
import jtux.UUtil as UUtil
import java.lang
```

```

import jarray
def b_to_s(ba): # convert byte array to string
    s = ""
    for b in ba:
        if b == 0:
            break
        s += chr(b)
    return s
nodename = "localhost"
servname1 = "5431" # unused port for peer 1
servname2 = "5432" # unused port for peer 2
msgsize = 300
hint = UNetwork.s_addrinfo()
hint.ai_family = UConstant.AF_INET
hint.ai_socktype = UConstant.SOCK_DGRAM
infop = UNetwork.AddrInfoListHead()
UNetwork.getaddrinfo(nodename, servname2, hint, infop)
sa_peer2 = infop.ai_next.ai_addr; # both peers need this socket addr
int_opt = UNetwork.SockOptValue_int() # Jtux's way of handling setsockopt values
int_opt.value = 1
pid = UProcess.fork()
if pid == 0: # Peer 1
    msg = jarray.zeros(msgsize, 'b') # jarray is a standard Jython module
    UNetwork.getaddrinfo(nodename, servname1, hint, infop)
    sa_peer1 = infop.ai_next.ai_addr
    UClock.sleep(1); # let peer 2 startup first -- not the best approach
    fd_skt = UNetwork.socket(UConstant.AF_INET, UConstant.SOCK_DGRAM, 0)
    UNetwork.setsockopt(fd_skt, UConstant.SOL_SOCKET, UConstant.SO_REUSEADDR,
        int_opt, 0)
    UNetwork.bind(fd_skt, sa_peer1, 0)
    sa_sender = UNetwork.s_sockaddr_in()
    sa_len = UUtil.IntHolder(); # Jtux class for passing ints by reference
    maxmsgs = 4
    for i in xrange(maxmsgs + 1):
        if i == maxmsgs:
            m = "Stop"
        else:
            m = "Message #" + str(i)
        UNetwork.sendto(fd_skt, m, len(m), 0, sa_peer2, 0)
        if i == maxmsgs:
            break
        nrcv = UNetwork.recvfrom(fd_skt, msg, len(msg), 0, sa_sender, sa_len)
        print "Peer 1 got \'" + b_to_s(msg) + '\'' from ' + str(sa_sender)
    UFile.close(fd_skt)
    print "Peer 1 exiting"
    UProcess._exit(UConstant.EXIT_SUCCESS)
else: # Peer 2
    msg = jarray.zeros(msgsize, 'b') # jarray is a standard Jython module
    fd_skt = UNetwork.socket(UConstant.AF_INET, UConstant.SOCK_DGRAM, 0)
    UNetwork.setsockopt(fd_skt, UConstant.SOL_SOCKET, UConstant.SO_REUSEADDR,
        int_opt, 0)
    UNetwork.bind(fd_skt, sa_peer2, 0)
    sa_sender = UNetwork.s_sockaddr_in()
    sa_len = UUtil.IntHolder()
    while 1:
        nrcv = UNetwork.recvfrom(fd_skt, msg, len(msg), 0, sa_sender, sa_len)
        if b_to_s(msg[:4]) == "Stop":
            break
        print "Peer 2 got \'" + b_to_s(msg[:nrcv]) + '\'' from ' + str(sa_sender)

```

```

    msg[0] = ord('m')
    UNetwork.sendto(fd_skt, msg, len(msg), 0, sa_sender, sa_len.value)
    UFile.close(fd_skt)
    UProcess.waitpid(pid, None, 0)
    print "Peer 2 exiting"

```

下面是Java示例:

```

// Java example that recursively descends the directory tree.
import jtux.*;
class TreeList {
    static void list(String entry, int level) {
        int dir_fd = -1;
        try {
            String tabs = "";
            for (int i = 0; i < level; i++)
                tabs += "\t";
            long dir = -1;
            try {
                dir = UDir.opendir(entry);
            }
            catch (UErrorException e) {
                System.out.println(tabs + entry + " - " + e);
                return;
            }
            dir_fd = UFile.open(".", UConstant.O_RDONLY);
            UProcess.chdir(entry);
            UDir.s_dirent dirent = new UDir.s_dirent();
            UFile.s_stat sbuf = new UFile.s_stat();

            while ((dirent = UDir.readdir(dir)) != null) {
                UFile.lstat(dirent.d_name, sbuf);
                if (UFile.S_ISDIR(sbuf.st_mode) && !dirent.d_name.equals(".") &&
                    !dirent.d_name.equals("..")) {
                    System.out.println(tabs + dirent.d_name + ".*");
                    if (UFile.S_ISLNK(sbuf.st_mode)) {
                        System.out.println(tabs + "[symbolic link -- skipping]");
                    }
                    else
                        list(dirent.d_name, level + 1);
                }
                else
                    System.out.println(tabs + dirent.d_name);
            }
            UDir.closedir(dir);
            UProcess.fchdir(dir_fd);
            UFile.close(dir_fd);
            dir_fd = -1;
        }
        catch (UErrorException e) {
            try {
                if (dir_fd != -1)
                    UProcess.fchdir(dir_fd);
            }
            catch (Exception edummy) {
            }
            System.out.println(e);
        }
    }
    public static void main(String args[]) {
        list("/", 0);
    }
}

```

```
}  
}
```

UNIX C API和Java并不能完全融合。Java没有指针这样的工具。实现定义的常量在Java程序中不可见，如ENOSYS或O_CREAT。有关如何克服这些困难和其他一些难题，以及如何建立和安装Jtux的详情见www.basepath.com/aup，当然也可以下载源代码。



附录D 函数字母速查表及其分类表

本附录分别按字母顺序和函数所属分类列出了本书涵盖的307个函数，并附有函数解释所对应的章节。

函数字母速查表

- FD_CLR——清除fd_set位 (4.2.3)
- FD_ISSET——检测fd_set位 (4.2.3)
- FD_SET——设置fd_set位 (4.2.3)
- FD_ZERO——清除整个fd_set (4.2.3)
- _Exit——不清理就终止进程 (5.7)
- _exit——不清理就终止进程 (5.7)
- _longjmp——不用恢复信号掩码就跳至跳转点 (9.6)
- _setjmp——设置跳转点 (9.6)
- abort——产生SIGABRT (9.1.9)
- accept——在套接字上接收新的连接并产生新的套接字 (8.1.2)
- access——确认文件的访问 (3.8.1)
- aio_cancel——取消异步I/O请求 (3.9.5)
- aio_error——为异步I/O操作检索错误状态 (3.9.4)
- aio_fsync——为某个文件初始化缓存刷新 (3.9.6)
- aio_read——从文件中异步读 (3.9.3)
- aio_return——检索异步I/O操作的返回状态 (3.9.4)
- aio_suspend——等待异步I/O操作请求 (3.9.7)
- aio_write——异步写入文件 (3.9.3)
- alarm——调度一个alarm信号 (9.7.1)
- asctime——把分散时间转换成本地时间字符串 (1.7.1)
- atexit——当进程退出时注册已调用的函数 (1.3.4)
- bind——把名字与套接字绑定 (8.1.2)
- cfgetispeed——从termios结构得到输入速率 (4.5.3)
- cfgetospeed——从termios结构得到输出速率 (4.5.3)
- cfsetispeed——在termios结构中设置输入速率 (4.5.3)
- cfsetospeed——在termios结构中设置输出速率 (4.5.3)
- chdir——根据路径改变当前目录 (3.6.2)
- chmod——根据路径改变文件模式 (3.7.1)
- chown——根据路径改变文件的所有者和文件的组 (3.7.2)



- chroot——改变根目录 (5.14)
- clock——得到执行时间 (1.7.2)
- clock_getcpuclockid——得到进程CPU时钟 (9.7.5)
- clock_getres——得到时钟精度 (9.7.5)
- clock_gettime——从时钟中获得时间 (9.7.5)
- clock_nanosleep——挂起几纳秒, 或等待某个信号 (9.7.5)
- clock_settime——设置时钟 (9.7.5)
- close——关闭文件描述符 (2.11)
- closedir——关闭目录 (3.6.1)
- confstr——得到配置字符串 (1.5.7)
- connect——连接套接字 (8.1.2)
- creat——创建或清空文件以便写入 (2.4.2)
- ctermid——为控制终端得到路径名 (4.7)
- ctime——把time_t转换成本地时间字符串 (1.7.1)
- difftime——计算两个time_t值的差 (1.7.1)
- dup——复制文件描述符 (6.3)
- dup2——复制文件描述符 (6.3)
- endhostent——结束主机数据库扫描 (8.8.1)
- endnetent——结束网络数据库扫描 (8.8.2)
- endprotoent——结束协议数据库扫描 (8.8.3)
- endservent——结束服务数据库扫描 (8.8.4)
- execl——执行带参数列表的文件 (5.3)
- execle——执行带参数列表和环境变量的文件 (5.3)
- execlp——执行带参数列表和路径搜索的文件 (5.3)
- execv——执行带参数数组的文件 (5.3)
- execve——执行带参数数组和环境变量的文件 (5.3)
- execvp——执行带参数数组和路径搜索的文件 (5.3)
- exit——清理并终止进程 (5.7)
- fchdir——根据文件描述符改变当前目录 (3.6.2)
- fchmod——根据文件描述符改变文件模式 (3.7.1)
- fchown——通过文件描述符来改变所有者和组 (3.7.2)
- fcntl——控制打开的文件 (3.8.3)
- fdatasync——对某个文件数据执行强制刷新缓存操作 (2.16.2)
- fork——建立新进程 (5.5)
- fpathconf——通过文件描述符得到系统选项或限制 (1.5.6)
- freeaddrinfo——释放套接字地址信息 (8.2.6)
- fstat——通过文件描述符得到文件信息 (3.5.1)
- fstatvfs——通过文件描述符得到文件系统信息 (3.2.3)
- fsync——对某个文件执行调度或强制刷新缓存操作 (2.16.2)



- ftok——产生System V IPC关键字 (7.4.2)
- ftruncate——通过文件描述符截短或加长文件 (2.17)
- gai_strerror——得到错误代码描述 (8.2.6)
- getaddrinfo——得到套接字地址信息 (8.2.6)
- getcwd——得到当前目录路径名 (3.4.2)
- getdate——用某些规则把字符串转换成分散时间 (1.7.1)
- getegid——得到有效组ID (5.11)
- getenv——得到环境变量值 (5.2)
- geteuid——得到有效用户ID (5.11)
- getgid——得到实际组ID (5.11)
- getgrgid——得到组文件入口 (3.5.2)
- gethostbyaddr——通过地址查询主机 (8.8.1)
- gethostbyname——通过名字查询主机 (8.8.1)
- gethostent——得到下一个主机数据库入口 (8.8.1)
- gethostid——得到本地主机标识符 (8.8.1)
- gethostname——得到主机名 (8.2.7)
- getitimer——得到间隔定时器的值 (9.7.4)
- getlogin——得到登录名 (3.5.2)
- getnameinfo——得到名字信息 (8.8.1)
- getnetbyaddr——通过网络号查询网络 (8.8.2)
- getnetbyname——通过名字查询网络 (8.8.2)
- getnetent——得到网络数据库入口 (8.8.2)
- getpeername——得到已连接套接字的套接字地址 (8.9.2)
- getpgid——得到进程组ID (4.3.3)
- getpid——得到进程ID (5.13)
- getppid——得到父进程ID (5.13)
- getprotobyname——通过名字查询协议 (8.8.3)
- getprotobynumber——通过协议号查询协议 (8.8.3)
- getprotoent——得到协议数据库入口 (8.8.3)
- getpwuid——得到口令文件入口 (3.5.2)
- getrlimit——得到资源限制 (5.16)
- getrusage——得到资源的使用 (5.16)
- getservbyname——通过名字查询服务 (8.8.4)
- getservbyport——通过端口查询服务 (8.8.4)
- getservent——得到服务数据库的入口 (8.8.4)
- getsid——得到会话ID (4.3.2)
- getsockname——得到套接字地址 (8.9.2)
- getsockopt——得到套接字选项 (8.3)
- gettimeofday——得到当前日期和时间作为timeval (1.7.1)



- getuid——得到实际用户ID (5.11)
- gmtime——把time_t转换成UTC的分散时间 (1.7.1)
- grantpt——取得pty从面的访问权限 (4.10.1)
- htonl——把32位值从主机字节序转换成网络字节序 (8.1.4)
- htons——把16位值从主机字节序转换成网络字节序 (8.1.4)
- if_freenameindex——通过if_nameindex释放分配的数组 (8.8.5)
- if_indextoname——把网络接口索引映射成名字 (8.8.5)
- if_nameindex——得到所有的网络接口名字和索引 (8.8.5)
- if_nametoindex——把网络接口名字映射成索引 (8.8.5)
- inet_addr——把IPv4点串地址转换成整数 (8.2.3)
- inet_ntoa——把IPv4整数地址转换成点串 (8.2.3)
- inet_ntop——把IPv4或IPv6二进制地址转换成字符串 (8.9.5)
- inet_pton——把IPv4或IPv6字符串地址转换成二进制 (8.9.5)
- ioctl——控制字符设备 (4.4)
- isatty——测试终端 (4.7)
- kill——为进程产生信号 (9.1.9)
- killpg——为进程组产生信号 (9.1.9)
- lchown——通过路径改变符号链接的所有者和组 (3.7.2)
- link——建立一个硬连接 (3.3.1)
- lio_listio——列出定向的I/O (3.9.9)
- listen——标识接收套接字并设置队列限制 (8.1.2)
- localtime——把time_t转换成本地分散时间 (1.7.1)
- lockf——锁定文件段 (7.11.3)
- longjmp——跳到跳转点 (9.6)
- lseek——设置和得到文件偏移量 (2.13)
- lstat——不遵循符号链接，通过路径得到文件信息 (3.5.1)
- mkdir——建立目录 (3.6.3)
- mkfifo——建立FIFO (7.2.1)
- mknod——建立文件 (3.8.2)
- mkstemp——用唯一的名字建立和打开文件 (2.7)
- mktime——把本地分散时间转换成time_t (1.7.1)
- mmap——映射内存页 (7.14.1)
- mount——安装文件系统 (非标准的) (3.2.4)
- mq_close——关闭消息队列 (7.7.1)
- mq_getattr——得到消息队列属性 (7.7.1)
- mq_notify——注册或注销消息通知 (7.7.1)
- mq_open——打开消息队列 (7.7.1)
- mq_receive——接收消息 (7.7.1)
- mq_send——发送消息 (7.7.1)

- mq_setattr—设置消息队列属性 (7.7.1)
- mq_timedreceive—以timeout接收消息 (7.7.1)
- mq_timedsend—以timeout发送消息 (7.7.1)
- mq_unlink—删除消息队列 (7.7.1)
- msgctl—控制消息队列 (7.5.1)
- msgget—得到消息队列标识符 (7.5.1)
- msgrcv—接收消息 (7.5.1)
- msgsnd—发送消息 (7.5.1)
- munmap—取消内存页映射 (7.14.1)
- nanosleep—执行挂起几纳秒或等待信号 (9.7.3)
- nice—改变nice值 (5.15)
- ntohl—把32位值从网络字节序转换成主机字节序 (8.1.4)
- ntohs—把16位值从网络字节序转换成主机字节序 (8.1.4)
- open—打开或创建文件 (2.4)
- opendir—打开目录 (3.6.1)
- pathconf—通过路径得到系统选项或限制 (1.5.6)
- pause—等待信号 (9.2.1)
- pipe—建立管道 (6.2.1)
- poll—等I/O准备好 (4.2.4)
- posix_openpt—打开pty (4.10.1)
- pread—在偏移处从文件描述符读 (2.14)
- pselect—等待I/O准备好 (4.2.3)
- pthread_cancel—取消线程 (5.17.5)
- pthread_cleanup_pop—卸载清除处理程序 (5.17.5)
- pthread_cleanup_push—安装清除处理程序 (5.17.5)
- pthread_cond_signal—信号条件 (5.17.4)
- pthread_cond_wait—等待条件 (5.17.4)
- pthread_create—建立线程 (5.17.1)
- pthread_join—等待线程终止 (5.17.2)
- pthread_kill—为线程产生信号 (9.1.9)
- pthread_mutex_lock—锁互斥 (5.17.3)
- pthread_mutex_unlock—解锁互斥 (5.17.3)
- pthread_sigmask—改变线程的信号掩码 (9.1.5)
- pthread_testcancel—尝试撤消 (5.17.5)
- ptsname—得到pty从面的名字 (4.10.1)
- putenv—改变或加入环境变量 (5.2)
- pwrite—在偏移处向文件描述符写 (2.14)
- raise—为线程产生信号 (9.1.9)
- read—从文件描述符中读入 (2.10)



- readdir——读目录 (3.6.1)
- readdir_r——读目录 (3.6.1)
- readlink——读符号连接 (3.3.3)
- readv——非连续的读 (2.15)
- recv——从套接字接收数据 (8.9.1)
- recvfrom——从套接字接收消息 (8.6.2)
- recvmsg——从套接字接收消息 (8.6.3)
- rename——重命名文件 (3.3.2)
- rewinddir——返绕目录 (3.6.1)
- rmdir——删除目录 (3.6.3)
- seekdir——搜索目录 (3.6.1)
- select——等待I/O准备好 (4.2.3)
- sem_close——关闭命名信号量 (7.10.1)
- sem_destroy——删除未命名信号量 (7.10.2)
- sem_getvalue——得到信号量的值 (7.10.1)
- sem_init——初始化未命名信号量 (7.10.2)
- sem_open——打开命名信号量 (7.10.1)
- sem_post——增加信号量 (7.10.1)
- sem_timedwait——减少信号量 (7.10.1)
- sem_trywait——如果可能就减少信号量 (7.10.1)
- sem_unlink——删除命名信号量 (7.10.1)
- sem_wait——减少信号量 (7.10.1)
- semctl——控制信号量集合 (7.9.1)
- semget——得到信号量集合标识符 (7.9.1)
- semop——操作信号量集合 (7.9.2)
- send——向套接字发送数据 (8.9.1)
- sendmsg——以消息结构向套接字发送消息 (8.6.3)
- sendto——向套接字发送消息 (8.6.2)
- setegid——设置有效的组ID (5.12)
- setenv——改变或加入环境变量 (5.2)
- seteuid——设置有效用户ID (5.12)
- setgid——设置有效组ID、实际组ID和已保存的组ID (5.12)
- sethostent——开始主机数据库扫描 (8.8.1)
- setitimer——设置间隔定时器的值 (9.7.4)
- setjmp——设置跳转点 (9.6)
- setnetent——开始网络数据库扫描 (8.8.2)
- setpgid——设置或建立进程组 (4.3.3)
- setprotoent——开始协议数据库扫描 (8.8.3)
- setrlimit——设置资源限制 (5.16)



- setservent——开始服务数据库扫描 (8.8.4)
- setsid——建立会话和进程组 (4.3.2)
- setsockopt——设置套接字选项 (8.3)
- setuid——设置有效用户ID、实际用户ID和已保存的用户ID (5.12)
- shm_open——打开共享内存对象 (7.14.1)
- shm_unlink——删除共享内存对象 (7.14.1)
- shmat——连接共享内存段 (7.13.1)
- shmctl——控制共享内存段 (7.13.1)
- shmdt——断开共享内存段 (7.13.1)
- shmget——得到共享内存段 (7.13.1)
- shutdown——关闭套接字发送和接收操作 (8.9.4)
- sigaction——设置信号动作 (9.1.6)
- sigaddset——把信号加入信号集 (9.1.5)
- sigaltstack——设置或得到备用栈上下文 (9.3)
- sigdelset——从信号集中删除信号 (9.1.5)
- sigemptyset——初始化空信号集 (9.1.5)
- sigfillset——初始化整个信号集 (9.1.5)
- sighold——阻塞信号 (9.4)
- sigignore——忽略信号 (9.4)
- siginterrupt——设置或清除SA_RESTART标志 (9.3)
- sigismember——在信号集中测试信号 (9.1.5)
- siglongjmp——跳到跳转点, 如果保存的话恢复信号掩码 (9.6)
- signal——设置信号动作 (9.4)
- sigpause——改变信号掩码并等待信号 (9.4)
- sigpending——检查未决信号 (9.3)
- sigprocmask——改变线程的信号掩码 (仅单个线程) (9.1.5)
- sigqueue——为进程产生信号 (9.5.4)
- sigrelse——解除信号阻塞 (9.4)
- sigset——信号管理 (9.4)
- sigsetjmp——设置跳转点 (9.6)
- sigsuspend——改变信号掩码并等待信号 (9.2.3)
- sigtimedwait——等待信号 (9.5.5)
- sigwait——等待信号 (9.2.2)
- sigwaitinfo——等待信号 (9.5.5)
- sleep——挂起执行几秒钟或等待信号 (9.7.2)
- socketatmark——测试带外标志是否存在 (8.7)
- socket——建立通信终点 (8.1.2)
- socketpair——建立套接字对 (8.9.3)
- stat——通过路径得到文件信息 (3.5.1)



- statvfs——通过路径得到文件系统信息 (3.2.3)
- strftime——把分散时间转换成带格式的字符串 (1.7.1)
- strptime——把字符串转换成带格式的分散时间 (1.7.1)
- symlink——建立符号链接 (3.3.3)
- sync——调度缓存刷新 (2.16.2)
- sysconf——得到系统选项或限制值 (1.5.5)
- system——运行命令 (5.5)
- tcdrain——耗尽 (等待) 终端输出 (4.6)
- tcflow——挂起或重启终端输入流或输出流 (4.6)
- tcflush——刷新 (丢弃) 终端输入、输出或者两者 (4.6)
- tcgetattr——得到终端属性 (4.5.1)
- tcgetpgrp——得到前台进程组ID (4.3.4)
- tcgetsid——得到会话ID (4.3.4)
- tcsendbreak——发送中断到终端 (4.6)
- tcsetattr——设置终端属性 (4.5.1)
- tcsetpgrp——设置前台进程组ID (4.3.4)
- telldir——得到目录位置 (3.6.1)
- time——得到当前日期和时间作为time_t (1.7.1)
- timer_create——建立每个进程定时器 (9.7.6)
- timer_delete——删除每个进程定时器 (9.7.6)
- timer_getoverrun——得到每个进程定时器消耗的时间 (9.7.6)
- timer_gettime——得到每个进程定时器的值 (9.7.6)
- timer_settime——设置每个进程定时器的值 (9.7.6)
- times——得到进程和子进程的执行时间 (1.7.2)
- truncate——通过路径截短或加长文件 (2.17)
- ttyname——查找终端的路径名 (4.7)
- ttyname_r——查找终端的路径名 (4.7)
- tzset——设置时区信息 (1.7.1)
- ulimit——得到和设置进程限制 (5.16)
- umask——设置和得到文件模式的创建掩码 (2.5)
- umount——卸载文件系统 (非标准的) (3.2.4)
- uname——得到关于当前系统的信息 (8.8.1)
- unlink——删除目录项 (2.6)
- unlockpt——解锁pty (4.10.1)
- unsetenv——删除环境变量 (5.2)
- usleep——挂起执行几个微秒或等待信号 (9.7.3)
- utime——设置文件访问时间和修改时间 (3.7.3)
- vfork——建立新进程; 共享内存 (过时的) (5.5)
- wait——等待子进程结束 (5.8)



- waitid——等待子进程改变状态[X/Open] (5.8)
- waitpid——等待子进程改变状态 (5.8)
- wcsftime——把分散时间转换成带格式的宽字符串 (1.7.1)
- write——向文件描述符写 (2.9)
- writew——非连续的写入 (2.15)

函数分类检索

配置

- confstr——得到配置字符串 (1.5.7)
- fpathconf——通过文件描述符得到系统选项和限制 (1.5.6)
- gethostid——得到本地主机标识符 (8.8.1)
- gethostname——得到主机名 (8.2.7)
- pathconf——通过路径得到系统选项或限制值 (1.5.6)
- sysconf——得到系统选项或限制 (1.5.5)
- uname——得到关于当前系统的信息 (8.8.1)

目录I/O

- closedir——关闭目录 (3.6.1)
- opendir——打开目录 (3.6.1)
- readdir——读目录 (3.6.1)
- readdir_r——读目录 (3.6.1)
- rewinddir——返绕目录 (3.6.1)
- seekdir——搜索目录 (3.6.1)
- telldir——得到目录位置 (3.6.1)

目录管理

- link——建立一个硬连接 (3.3.1)
- mkdir——建立目录 (3.6.3)
- readlink——读符号链接 (3.3.3)
- rename——重命名文件 (3.3.2)
- rmdir——删除目录 (3.6.3)
- symlink——建立符号链接 (3.3.3)
- unlink——删除目录项 (2.6)

文件属性

- access——确认文件的访问 (3.8.1)
- chmod——根据路径改变文件模式 (3.7.1)
- chown——根据路径改变文件的所有者和组 (3.7.2)



- fchmod——根据文件描述符改变文件模式 (3.7.1)
- fchown——通过文件描述来改变所有者和组 (3.7.2)
- fstat——通过文件描述符得到文件信息 (3.5.1)
- lchown——根据路径改变符号链接的所有者和组 (3.7.2)
- lstat——不遵循符号链接, 通过路径得到文件信息 (3.5.1)
- stat——通过路径得到文件信息 (3.5.1)
- utime——设置文件访问时间和修改时间 (3.7.3)

文件I/O

- close——关闭文件描述符 (2.11)
- creat——建立或清空文件以便写入 (2.4.2)
- dup——复制文件描述符 (6.3)
- dup2——复制文件描述符 (6.3)
- fcntl——控制打开的文件 (3.8.3)
- fdatasync——对某个文件数据执行强制刷新缓存操作 (2.16.2)
- fsync——对某个文件执行调度或强制刷新缓存操作 (2.16.2)
- ftruncate——通过文件描述符截短或加长文件 (2.17)
- lseek——设置和得到文件偏移量 (2.13)
- mkstemp——用唯一的名字建立和打开文件 (2.7)
- open——打开或创建文件 (2.4)
- pipe——建立管道 (6.2.1)
- poll——等I/O准备好 (4.2.4)
- pread——在偏移处从文件描述符读 (2.14)
- pselect——等待I/O准备好 (4.2.3)
- pwrite——在偏移处向文件描述符写 (2.14)
- read——从文件描述符中读入 (2.10)
- readv——非连续的读 (2.15)
- select——等待I/O准备好 (4.2.3)
- sync——调度缓存刷新 (2.16.2)
- truncate——通过路径截短或加长文件 (2.17)
- write——向文件描述符写 (2.9)
- writv——非连续的写入 (2.15)

文件I/O(异步)

- aio_cancel——取消异步I/O请求 (3.9.5)
- aio_error——为异步I/O操作检索错误状态 (3.9.4)
- aio_fsync——为某个文件初始化缓存刷新 (3.9.6)
- aio_read——从文件中异步读 (3.9.3)
- aio_return——检索异步I/O操作的返回状态 (3.9.4)



aio_suspend—等待异步I/O操作请求 (3.9.7)

aio_write—异步写入文件 (3.9.3)

lio_listio—列出定向的I/O (3.9.9)

文件管理

lockf—锁定文件段 (7.11.3)

mkfifo—建立FIFO (7.2.1)

mknod—建立文件 (3.8.2)

文件系统

fstatvfs—通过文件描述符得到文件系统信息 (3.2.3)

mount—安装文件系统 (非标准的) (3.2.4)

statvfs—通过路径得到文件系统信息 (3.2.2)

umount—卸载文件系统 (非标准的) (3.2.4)

文件描述符集

FD_CLR—清除fd_set位 (4.2.3)

FD_ISSET—检测fd_set位 (4.2.3)

FD_SET—设置fd_set位 (4.2.3)

FD_ZERO—清除整个fd_set (4.2.3)

IPC—POSIX消息队列

mq_close—关闭消息队列 (7.7.1)

mq_getattr—得到消息队列属性 (7.7.1)

mq_notify—注册或注销消息通知 (7.7.1)

mq_open—打开消息队列 (7.7.1)

mq_receive—接收消息 (7.7.1)

mq_send—发送消息 (7.7.1)

mq_setattr—设置消息队列属性 (7.7.1)

mq_timedreceive—用timeout接收消息 (7.7.1)

mq_timedsend—用timeout发送消息 (7.7.1)

mq_unlink—移除消息队列 (7.7.1)

IPC—POSIX信号量

sem_close—关闭命名信号量 (7.10.1)

sem_destroy—释放未命名信号量 (7.10.2)

sem_getvalue—得到信号量的值 (7.10.1)

sem_init—初始化未命名信号量 (7.10.2)

sem_open—打开命名信号量 (7.10.1)



- sem_post——增加信号量 (7.10.1)
- sem_timedwait——减少信号量 (7.10.1)
- sem_trywait——如果可能就减少信号量 (7.10.1)
- sem_unlink——删除命名信号量 (7.10.1)
- sem_wait——减少信号量 (7.10.1)

IPC——POSIX共享内存

- mmap——映射内存页 (7.14.1)
- munmap——取消内存页映射 (7.14.1)
- shm_open——打开共享内存对象 (7.14.1)
- shm_unlink——删除共享内存对象 (7.14.1)

IPC——System V消息队列

- ftok——产生System V IPC关键字 (7.4.2)
- msgctl——控制消息队列 (7.5.1)
- msgget——得到消息队列标识符 (7.5.1)
- msgrcv——接收消息 (7.5.1)
- msgsnd——发送消息 (7.5.1)

IPC——System V信号量

- semctl——控制信号量设置 (7.9.1)
- semget——得到信号量设置标识符 (7.9.1)
- semop——操作信号量设置 (7.9.2)

IPC——System V共享内存

- shmat——连接共享内存段 (7.13.1)
- shmctl——控制共享内存段 (7.13.1)
- shmdt——断开共享内存段 (7.13.1)
- shmget——得到共享的内存段 (7.13.1)

网络数据库

- endhostent——结束主机数据库扫描 (8.8.1)
- endnetent——结束网络数据库扫描 (8.8.2)
- endprotoent——结束协议数据库扫描 (8.8.3)
- endservent——结束服务数据库扫描 (8.8.4)
- gethostent——得到下一个主机数据库入口 (8.8.1)
- getnetbyaddr——通过数字查询网络 (8.8.2)
- getnetbyname——通过名字查询网络 (8.8.2)
- getnetent——得到网络数据库入口 (8.8.2)



- getprotobyname——通过名字查询协议 (8.8.3)
- getprotobynumber——通过协议号查询协议 (8.8.3)
- getprotoent——得到协议数据库入口 (8.8.3)
- getservbyname——通过名字查询服务 (8.8.4)
- getservbyport——通过端口查询服务 (8.8.4)
- getservent——得到服务数据库的入口 (8.8.4)
- htonl——把32位值从主机字节序转换成网络字节序 (8.1.4)
- htons——把16位值从主机字节序转换成网络字节序 (8.1.4)
- if_freenameindex——通过if_nameindex释放分配的数组 (8.8.5)
- if_indextoname——把网络接口索引映射成名字 (8.8.5)
- if_nameindex——得到所有的网络接口名字和索引 (8.8.5)
- if_nametoindex——把网络接口名字映射成索引 (8.8.5)
- ntohl——把32位值从网络字节序转换成主机字节序 (8.1.4)
- ntohs——把16位值从网络字节序转换成主机字节序 (8.1.4)
- sethostent——开始主机数据库扫描 (8.8.1)
- setnetent——开始网络数据库扫描 (8.8.2)
- setprotoent——开始协议数据库扫描 (8.8.3)
- setservent——开始服务数据库扫描 (8.8.4)

进程属性

- chdir——根据路径改变当前目录 (3.6.2)
- chroot——改变根目录 (5.14)
- fchdir——根据文件描述符改变当前目录 (3.6.2)
- getcwd——得到当前目录路径名 (3.4.2)
- getrusage——得到资源的使用 (5.16)
- nice——改变nice值 (5.15)

进程控制流

- _longjmp——不用恢复信号掩码就跳至中断点 (9.6)
- _setjmp——设置跳转点 (9.6)
- longjmp——跳到跳转点 (9.6)
- pause——等待信号 (9.2.1)
- setjmp——设置跳转点 (9.6)
- siglongjmp——跳到跳转点, 如果保存的话恢复信号掩码 (9.6)
- sigsetjmp——设置跳转点 (9.6)

进程环境

- getenv——得到环境变量值 (5.2)
- putenv——改变或加入环境变量 (5.2)
- setenv——改变或增加环境变量 (5.2)



unsetenv——删除环境变量 (5.2)

进程限制

getrlimit——得到资源限制 (5.16)

setrlimit——设置资源限制 (5.16)

ulimit——得到和设置进程限制 (5.16)

进程管理

_Exit——不清理就终止进程 (5.7)

_exit——不清理就终止进程 (5.7)

abort——产生SIGABRT (9.1.9)

atexit——当进程退出时注册已调用的函数 (1.3.4)

execl——执行带参数列表的文件 (5.3)

execle——执行带参数列表和环境变量的文件 (5.3)

execlp——执行带参数列表和路径搜索的文件 (5.3)

execv——执行带参数数组的文件 (5.3)

execve——执行带参数数组和环境变量的文件 (5.3)

execvp——执行带参数数组和路径搜索的文件 (5.3)

exit——清理并终止进程 (5.7)

fork——建立新进程 (5.5)

system——运行命令 (5.5)

vfork——建立新进程; 共享内存 (过时的) (5.5)

wait——等待子进程结束 (5.8)

waitid——等待子进程改变状态[X/Open] (5.8)

waitpid——等待子进程改变状态 (5.8)

进程权限

getegid——得到有效组ID (5.11)

geteuid——得到有效用户ID (5.11)

getgid——得到实际组ID (5.11)

getuid——得到实际用户ID (5.11)

setegid——设置有效组ID (5.12)

seteuid——设置有效用户ID (5.12)

setgid——设置有效组ID、实际组ID和已保存的组ID (5.12)

setuid——设置有效用户ID、实际用户ID和已保存的用户ID (5.12)

umask——设置和得到文件模式的创建掩码 (2.5)

进程资源

getpgid——得到进程组ID (4.3.3)



- getpid——得到进程ID (5.13)
- getppid——得到父进程ID (5.13)
- getsid——得到会话ID (4.3.2)
- setpgid——设置或建立进程组 (4.3.3)
- setsid——建立会话和进程组 (4.3.2)

信号

- kill——为进程产生信号 (9.1.9)
- killpg——为进程组产生信号 (9.1.9)
- raise——为线程产生信号 (9.1.9)
- sigaction——设置信号动作 (9.1.6)
- sigaltstack——设置或得到备用栈上下文 (9.3)
- sighold——阻塞信号 (9.4)
- sigignore——忽略信号 (9.4)
- siginterrupt——设置或清除SA_RESTART标志 (9.3)
- sigismember——在信号集中测试信号 (9.1.5)
- signal——设置信号动作 (9.4)
- sigpause——改变信号掩码并等待信号 (9.4)
- sigpending——检查未决信号 (9.3)
- sigqueue——为进程产生信号 (9.5.4)
- sigrelse——解除信号阻塞 (9.4)
- sigset——信号管理 (9.4)
- sigsuspend——改变信号掩码并等待信号 (9.2.3)
- sigtimedwait——等待信号 (9.5.5)
- sigwait——等待信号 (9.2.2)
- sigwaitinfo——等待信号 (9.5.5)

信号标记

- sigaddset——把信号加入信号集 (9.1.5)
- sigdelset——从信号集中删除信号 (9.1.5)
- sigemptyset——初始化空信号集 (9.1.5)
- sigfillset——初始化整个信号集 (9.1.5)
- sigprocmask——改变线程的信号掩码 (仅单个信号) (9.1.5)

套接字

- accept——在套接字上接收新的连接并产生新的套接字 (8.1.2)
- bind——把名字与套接字绑定 (8.1.2)
- connect——连接套接字 (8.1.2)
- getpeername——得到已连接套接字的套接字地址 (8.9.2)



- getsockname——得到套接字地址 (8.9.2)
- getsockopt——得到套接字选项 (8.3)
- listen——标识接收套接字并设置队列限制 (8.1.2)
- recv——从套接字接收数据 (8.9.1)
- recvfrom——从套接字接收消息 (8.6.2)
- recvmsg——从套接字接收消息 (8.6.3)
- send——向套接字发送数据 (8.9.1)
- sendmsg——以消息结构向套接字发送消息 (8.6.3)
- sendto——向套接字发送消息 (8.6.2)
- setsockopt——设置套接字选项 (8.3)
- shutdown——关闭套接字发送和接收操作 (8.9.4)
- socketatmark——测试带外标志是否存在 (8.7)
- socket——建立通信的终点 (8.1.2)
- socketpair——建立套接字对 (8.9.3)

套接字地址

- freeaddrinfo——释放套接字地址信息 (8.2.6)
- gai_strerror——得到错误代码描述 (8.2.6)
- getaddrinfo——得到套接字地址信息 (8.2.6)
- gethostbyaddr——通过地址查询主机 (8.8.1)
- gethostbyname——通过名字查询主机 (8.8.1)
- getnameinfo——得到名字信息 (8.8.1)
- inet_addr——把IPv4点串地址转换成整数 (8.2.3)
- inet_ntoa——把IPv4整数地址转换成点串 (8.2.3)
- inet_ntop——把IPv4或IPv6二进制地址转换成字符串 (8.9.5)
- inet_pton——把IPv4或IPv6字符串地址转换成二进制 (8.9.5)

终端

- cfgetispeed——从termios结构得到输入速率 (4.5.3)
- cfgetospeed——从termios结构得到输出速率 (4.5.3)
- cfsetispeed——在termios结构中设置输入速率 (4.5.3)
- cfsetospeed——在termios结构中设置输出速率 (4.5.3)
- ctermid——为控制终端得到路径名 (4.7)
- ioctl——控制字符设备 (4.4)
- isatty——测试终端 (4.7)
- tcdrain——耗尽 (等待) 终端输出 (4.6)
- tcflow——挂起或重启终端输入流或输出流 (4.6)
- tcflush——刷新 (丢弃) 终端输入、输出或者两者 (4.6)
- tcgetattr——得到终端属性 (4.5.1)



- tcgetpgrp——得到前台进程组ID (4.3.4)
- tcgetsid——得到会话ID (4.3.4)
- tcsendbreak——发送中断到终端 (4.6)
- tcsetattr——设置终端属性 (4.5.1)
- tcsetpgrp——设置前台进程组ID (4.3.4)
- ttyname——查找终端的路径名 (4.7)
- ttyname_r——查找终端的路径名 (4.7)

终端(Pty)

- grantpt——取得pty从面的访问权限 (4.10.1)
- posix_openpt——打开pty (4.10.1)
- ptsname——得到pty从面的名字 (4.10.1)
- unlockpt——解锁pty (4.10.1)

线程

- pthread_cancel——取消线程 (5.17.5)
- pthread_cleanup_pop——卸载清除句柄 (5.17.5)
- pthread_cleanup_push——安装清除句柄 (5.17.5)
- pthread_cond_signal——信号条件 (5.17.4)
- pthread_cond_wait——等待条件 (5.17.4)
- pthread_create——建立线程 (5.17.1)
- pthread_join——等待线程终止 (5.17.2)
- pthread_kill——为线程产生信号 (9.1.9)
- pthread_mutex_lock——锁互斥 (5.17.3)
- pthread_mutex_unlock——解锁互斥 (5.17.3)
- pthread_sigmask——改变线程的信号掩码 (9.1.5)
- pthread_testcancel——尝试撤消 (5.17.5)

时间

- asctime——把分散时间转换成本地时间字符串 (1.7.1)
- clock——得到执行时间 (1.7.2)
- clock_getcpuclockid——得到进程CPU时钟 (9.7.5)
- clock_getres——等到时钟精度 (9.7.5)
- clock_gettime——从时钟中获得时间 (9.7.5)
- clock_nanosleep——挂起几纳秒, 或等待某个信号 (9.7.5)
- clock_settime——设置时钟 (9.7.5)
- ctime——把time_t转换成本地时间字符串 (1.7.1)
- difftime——计算两个time_t值的差 (1.7.1)
- getdate——用某些规则把字符串转换成分散时间 (1.7.1)



- gettimeofday——得到当前日期和时间作为timeval (1.7.1)
- gmtime——把time_t转换成UTC的分散时间 (1.7.1)
- localtime——把time_t转换成本地分散时间 (1.7.1)
- mktime——把本地分散时间转换成time_t (1.7.1)
- nanosleep——执行挂起几纳秒或等待信号 (9.7.3)
- sleep——挂起执行几秒钟或等待信号 (9.7.2)
- strftime——把分散时间转换成带格式的字符串 (1.7.1)
- strptime——把字符串转换成带格式的分散时间 (1.7.1)
- time——得到当前日期和时间如time_t (1.7.1)
- times——得到进程和子进程的运行时间 (1.7.2)
- tzset——设置时区信息 (1.7.1)
- usleep——挂起执行几个微秒或等待信号 (9.7.3)
- wcsftime——把分散时间转换成带格式的宽字符串 (1.7.1)

定时器

- alarm——调度一个alarm信号 (9.7.1)
- getitimer——得到间隔定时器的值 (9.7.4)
- setitimer——设置间隔定时器的值 (9.7.4)
- timer_create——建立每个进程定时器 (9.7.6)
- timer_delete——删除每个进程定时器 (9.7.6)
- timer_getoverrun——得到每个进程定时器消耗时间 (9.7.6)
- timer_gettime——得到每个进程定时器的值 (9.7.6)
- timer_settime——设置每个进程定时器的值 (9.7.6)

用户信息

- getgrgid——得到组文件入口 (3.5.2)
- getlogin——得到登录名 (3.5.2)
- getpwuid——得到口令文件入口 (3.5.2)



参考文献

- [Abr1996] Abrahams, Paul W. and Bruce R. Larson, *UNIX for the Impatient, 2nd Ed.*, Addison-Wesley Longman, 1996.
- [AUP2003] Rochkind, Marc, Advanced UNIX Programming Web Site, www.basepath.com/aup
- [Bac1986] Bach, Maurice J., *The Design of the UNIX Operating System*, Prentice Hall, 1986.
- [Bov2001] Bovet, Daniel P. and Marco Cesati, *Understanding the Linux Kernel*, O'Reilly & Associates, 2001.
- [But1997] Butenhof, David R., *Programming with POSIX Threads*, Addison-Wesley Longman, 1997.
- [Dre2003] Drepper, Ulrich, "POSIX Option Groups," <http://people.redhat.com/~drepper/posix-option-groups.html>
- [Har2002] Harbison III, Samuel P. and Guy L. Steele Jr., *C: A Reference Manual, 5th Ed.*, Prentice Hall, 2002.
- [Keg2003] Kegel, Dan, "The C10K Problem," www.kegel.com/c10k.html
- [Ker1984] Kernighan, Brian W. and Rob Pike, *The UNIX Programming Environment*, Prentice Hall Computer Books, 1984.
- [Mau2001] Mauro, Jim and Richard McDougall, *Solaris Internals*, Prentice Hall PTR, 2001.
- [McK1996] McKusick, Marshall Kirk, Keith Bostic, Michael J. Karels, and John S. Quarterman, *The Design and Implementation of the 4.4BSD Operating System*, Addison-Wesley, 1996.
- [Nor1997] Norton, Scott J. and Mark D. DiPasquale, *Thread Time: The Multithreaded Programming Guide*, Prentice-Hall PTR, 1997.
- [RFC] IETF RFC Page, www.ietf.org/rfc
- [RFC854] Postel, J. and J. Reynolds, "Telnet Protocol Specification," www.ietf.org/rfc/rfc0854.txt?number=854
- [RFC1288] Zimmerman, D., "The Finger User Information Protocol," www.ietf.org/rfc/rfc1288.txt?number=1288
- [Ste1999] Stevens, Richard P., *UNIX Network Programming, Vol. 2, 2nd Ed.*, Prentice Hall PTR, 1999.
- [Ste2003] Stevens, Richard P., Bill Fenner, and Andrew M. Rudoff, *UNIX Network Programming, Vol. 1, 3rd Ed.*, Addison-Wesley, 2004.
- [Str2000] Stroustrup, Bjarne, *The C++ Programming Language (Special 3rd Edition)*, Addison-Wesley, 2000.
- [SUS2002] The Open Group, "The Single UNIX Specification, Version 3," www.unix.org/version3

- [Ste2003] Stevens, Richard P., Bill Fenner, and Andrew M. Rudoff, *UNIX Network Programming, Vol. 1, 3rd Ed.*, Addison-Wesley, 2004.
- [Str2000] Stroustrup, Bjarne, *The C++ Programming Language (Special 3rd Edition)*, Addison-Wesley, 2000.
- [SUS2002] The Open Group, "The Single UNIX Specification, Version 3," www.unix.org/version3

其他推荐读物

- Bell Labs, "The Creation of the UNIX* Operating System," www.bell-labs.com/history/unix/. An interesting site sponsored by Bell Labs, where UNIX originated.
- Gallmeister, Bill O., *POSIX.4: Programming for the Real World*, O'Reilly & Associates, 1995. The only book on the POSIX realtime extensions, written by the vice-chair of the group that wrote the standard.
- Garfinkel, Simson, Daniel Weise, and Steven Strassmann, *The UNIX-Haters Handbook*, IDG Books Worldwide, 1994. UNIX-lovers hate this book, but its criticisms are right on. The book is hilarious and deserves to be recommended for that reason alone. Out of print, but you can read it for free at <http://web.mit.edu/~simsong/www/ugh.pdf>. (If the link doesn't work, google the book title.)
- Google Groups, www.google.com/grphp. Where you'll find newsgroups such as comp.unix.programmer.
- Libes, Don and Sandy Ressler, *Life With UNIX*, Prentice Hall, 1989. UNIX history and lots of other interesting information, both trivial and important. Most of its technical and market information is dated, but there's plenty left that isn't.
- Nemeth, Evi, Garth Snyder, Scott Seebass, and Trent R. Hein, *UNIX System Administration Handbook, 3rd Ed.*, Prentice Hall PTR, 2001. The best UNIX administration book. Covers Solaris, HP-UX, Red Hat Linux, and FreeBSD specifically, but most of the book applies to any UNIX system. From its title you might not think this book is a pleasure to read, but it is.
- Salus, Peter H., *A Quarter Century of UNIX*, Addison-Wesley, 1994. Exceptionally complete history of UNIX.
- Taylor, Christopher C., "Unix Is a Four Letter Word," <http://unix.t-a-y-l-o-r.com>. Online introduction to UNIX and vi.
- Unix Heritage Society, www.tuhs.org. Outstanding collection of links to historical UNIX materials.

