Addison Wesley

TURING

图灵计算机科学丛书

UNIX 网络编程

卷2：进程间通信

第2版

UNIX Network Programming

Volume 2: Interprocess Communications, Second Edition

[美] W. Richard Stevens 著



人民邮电出版社
POSTS & TELECOM PRESS

UNIX Network Programming

Volume 2: Interprocess Communications, Second Edition

“我做专业程序员已经二十余年了，每当要写新程序的时候，我都要参考这本书。Stevens的书是我进入VoIP和音频CODEC网络编程领域的敲门砖。”

——C. T. Vanderdecken

“在我做过的项目中，已经记不清楚有多少次靠这套书挽狂澜于既倒……每位真正的UNIX系统程序员都应该买一套。”

——David Sharpe

“对每位UNIX平台的程序员而言，这本书都是必备的。它深入阐释了UNIX平台上的各种IPC API。既涵盖System V也涉及POSIX标准，对于IPC的论述无可匹敌。书末的附录对各种管道、FIFO、Posix消息队列、System V消息队列、门、Sun RPC的性能进行了比较，我从来没有看到任何书能如此广泛而深入地介绍这一主题。更何况，它出自权威人物之手。”

——asingh

“Stevens是UNIX网络编程的‘神’，至少是我的‘神’。”

——Matthew MacGibbon

两卷本的《UNIX网络编程》是已故著名技术作家W. Richard Stevens的传世之作。卷2着重讨论如何让应用程序与在其他机器上的应用程序进行对话。

良好的进程间通信（IPC）机制是提高UNIX程序性能的关键。本书全面深入地讲解了各种进程间通信形式，包括消息传递、同步、共享内存及远程调用（RPC）。书中包含了大量经过优化的源代码，帮助读者加深理解。这些源代码可以从图灵网站本书网页免费注册下载。

本书是网络研究和开发人员公认的权威参考书，深入理解本书内容，方能设计出良好的UNIX软件。

Addison Wesley

www.informit.com

图灵网站 http://www.turingbook.com 热线：(010) 51095186
反馈/投稿/推荐信箱：contact@turingbook.com
有奖勘误：debug@turingbook.com

分类建议：计算机/操作系统/UNIX
人民邮电出版社网址 www.ptpress.com.cn

人民邮电出版社 POSTS & TELECOM PRESS

PEARSON Education

ISBN 978-7-115-23028-7

9 787115 230287 >

ISBN 978-7-115-23028-7

定价：89.00元

UNIX 网络编程

卷2：进程间通信

第2版

UNIX Network Programming

[美] W. Richard Stevens 著

人民邮电出版社
北京

图灵计算机科学丛书

图书在版编目（CIP）数据

UNIX网络编程 : 第2版. 第2卷, 进程间通信 / (美)史蒂文斯 (Stevens, W. R.) 著. — 北京 : 人民邮电出版社, 2010.7
(图灵计算机科学丛书)
书名原文: UNIX Network Programming, Volume 2: Interprocess Communications
ISBN 978-7-115-23028-7

Ⅰ. ①U… Ⅱ. ①史… Ⅲ. ①UNIX操作系统—程序设计 Ⅳ. ①TP316.81

中国版本图书馆CIP数据核字(2010)第086883号

内 容 提 要

本书是一部UNIX网络编程的经典之作！进程间通信（IPC）几乎是所有Unix程序性能的关键，理解IPC也是理解如何开发不同主机间网络应用程序的必要条件。本书从对Posix IPC和System V IPC的内部结构开始讨论，全面深入地介绍了4种IPC形式：消息传递（管道、FIFO、消息队列）、同步（互斥锁、条件变量、读写锁、文件与记录锁、信号量）、共享内存（匿名共享内存、具名共享内存）及远程过程调用（Solaris门、Sun RPC）。附录中给出了测量各种IPC形式性能的方法。

本书内容详尽且具权威性，几乎每章都提供精选的习题，并提供了部分习题的答案，是网络研究和开发人员理想的参考书。

图灵计算机科学丛书

UNIX网络编程 卷2：进程间通信（第2版）

◆ 著 [美] W. Richard Stevens
责任编辑 杨海玲
执行编辑 傅尔也

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子函件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
三河市海波印务有限公司印刷

◆ 开本：787×1092 1/16
印张：29.5
字数：830千字 2010年7月第1版
印数：1－4 000册 2010年7月河北第1次印刷

著作权合同登记号 图字：01-2009-5716号

ISBN 978-7-115-23028-7

定价：89.00元

读者服务热线：(010)51095186 印装质量热线：(010)67129223
反盗版热线：(010)67171154

版 权 声 明

Authorized translation from the English language edition, entitled *UNIX Network Programming, Volume 2: Interprocess Communications, Second Edition*, 9780130810816 by W. Richard Stevens, published by Pearson Education, Inc., publishing as Prentice Hall PTR, Copyright © 1999 by Prentice Hall PTR.

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

CHINESE SIMPLIFIED language edition published by PEARSON EDUCATION ASIA LTD. And POSTS & TELECOM PRESS Copyright © 2010.

本书中文简体字版由 Pearson Education Asia Ltd.授权人民邮电出版社独家出版。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。

本书封面贴有 Pearson Education（培生教育出版集团）激光防伪标签，无标签者不得销售。

版权所有，侵权必究。

前　言

概述

大多数重要的程序都涉及进程间通信（Interprocess Communication, IPC）。这是受下述设计原则影响的自然结果：把应用程序设计为一组相互通信的小片断比将其设计为单个庞大的程序更好。从历史角度看，应用程序有如下几种构建方法。

(1) 用一个庞大的程序完成全部工作。程序的各部分可以实现为函数，函数之间通过参数、返回值和全局变量来交换信息。

(2) 使用多个程序，程序之间用某种形式的IPC进行通信。许多标准的Unix工具都是按这种风格设计的，它们使用shell管道（IPC的一种形式）在程序之间传递信息。

(3) 使用一个包含多个线程的程序，线程之间使用某种IPC。这里仍然使用术语IPC，尽管通信是在线程之间而不是在进程之间进行的。

还可以把后两种设计形式结合起来：用多个进程来实现，其中每个进程包含几个线程。在这种情况下，进程内部的线程之间可以通信，不同的进程之间也可以通信。

上面讲述了可以把完成给定任务所需的工作分到多个进程中，或许还可以进一步分到进程内的多个线程中。在包含多个处理器（CPU）的系统中，多个进程也许可以（在不同的CPU上）同时运行，或许给定进程内的多个线程也能同时运行。因此，把任务分到多个进程或线程中有望减少完成指定任务的时间。

本书详细描述了以下4种不同的IPC形式：

(1) 消息传递（管道、FIFO和消息队列）；

(2) 同步（互斥量、条件变量、读写锁、文件和记录锁、信号量）；

(3) 共享内存（匿名的和具名的）；

(4) 远程过程调用（Solaris门和Sun RPC）。

本书不讨论如何编写通过计算机网络通信的程序。这种通信通常涉及使用TCP/IP协议族的套接字API，相关主题在第1卷[Stevens 1998]中有详细讨论。

有人可能会提出质疑：不应该使用单主机或非网络IPC（本卷的主题），所有程序都应该在网络上的多台主机上同时运行。但在日常实践中，单主机IPC往往比网络通信快得多，而且有时还简单些。共享内存、同步等方法通常也只能用于单主机，跨网络时可能无法使用。经验和历史表明，非网络IPC（本卷）与跨网络IPC（第1卷）都是需要的。

本卷建立在第1卷和我写的另外4本书的基础上，这5本书在本书中简记如下：

- UNPv1：*UNIX Network Programming, Volume 1* [Stevens 1998]；
- APUE：*Advanced Programming in the UNIX Environment* [Stevens 1992]；
- TCPv1：*TCP/IP Illustrated, Volume 1* [Stevens 1994]；
- TCPv2：*TCP/IP Illustrated, Volume 2* [Wright and Stevens 1995]；

- TCPv3：*TCP/IP Illustrated, Volume 3* [Stevens 1996]。

在一本以“网络编程”为书名一部分的书中讨论IPC看似有点奇怪，但事实上IPC经常用于网络应用程序。我在《UNIX网络编程》1990年版的前言里就指出：“想知道如何为网络开发软件，必须先理解进程间通信（IPC）。”

与第1版的区别

本书完全重写并扩充了1990年版《UNIX网络编程》的第3章和第18章。字数统计表明，现在的内容是第1版的5倍。新版的主要改动归纳如下。

- 不仅讨论了“System V IPC”的三种形式（消息队列、信号量以及共享内存），还对实现了这些IPC的新的Posix函数进行了介绍。（1.7节将详细介绍Posix标准族。）我认为使用Posix IPC函数是大势所趋，因为它们比System V中的相应部分更具优势。
- 讨论了用于同步的Posix函数：互斥锁、条件变量以及读写锁。它们可用于线程或进程的同步，而且往往在访问共享内存时使用。
- 本卷假定使用Posix线程环境（称为“Pthreads”），许多示例都是用多线程而不是多进程构建的。
- 对管道、FIFO和记录锁的讨论侧重于从它们的Posix定义出发。
- 本卷不仅描述了IPC机制及其使用方法，还实现了Posix消息队列、读写锁与Posix信号量（都可以实现为用户库）。这些实现可以把多种不同的特性捆绑起来（例如，Posix信号量的一种实现用到了互斥量、条件变量和内存映射I/O），还强调了我们在应用程序中经常要处理的一些问题（如竞争状态、错误处理、内存泄漏和变长参数列表）。理解某种特性的实现通常有助于了解如何使用该特性。
- 对RPC的讨论侧重于Sun的RPC包。在此之前讲述了新的Solaris门API，它类似于RPC但用于单主机。这么一来我们就介绍了许多在调用其他进程中的过程时需要考虑的特性，而不用关心网络方面的细节。

读者对象

本书既可以用作IPC的教程，也可以用作有经验的程序员的参考书。全书划分为以下4个主要部分：

- 消息传递；
- 同步；
- 共享内存；
- 远程过程调用。

但许多读者可能只对特定的部分感兴趣。第2章总结了所有Posix IPC函数共有的特性，第3章归纳了所有System V IPC函数共有的特性，第12章介绍了Posix和System V的共享内存，但书中多数章节都可以独立于其他章节阅读。所有读者都应该阅读第1章，尤其是1.6节，该节介绍了一些贯穿全书的包装函数。讨论Posix IPC的各章与讨论System V IPC的各章彼此独立，有关管道、FIFO和记录锁的几章不属于上述两个阵营，关于RPC的两章也独立于其他IPC方法。

为了方便读者把本书作为参考书，本书提供了完整的全文索引，并在最后几页总结了每个函数和结构的详细描述在正文中的哪里可以找到。为了给不按顺序阅读本书的读者提供方便，

我们在书中为各个主题提供了大量的交叉引用。

源代码与勘误

书中所有示例的源代码可以从作者主页（列在前言的最后）获得[①]。学习本书讲述的IPC技术的最好方法就是下载这些程序，对其进行修改和改进。只有这样实际编写代码才能深入理解有关概念和方法。每章末尾提供了大量的习题，大部分在附录D中给出答案。

本书的最新勘误表也可以从作者主页获取。

致谢

尽管封面上只出现了作者一个人的名字，但一本高质量的书的出版需要许多人的共同努力。首先要感谢我的家人，他们在我写书的那段时间里承担了一切。再次感谢你们：Sally、Bill、Ellen和David。

感谢技术审稿人给出的宝贵的反馈意见（打印出来有135页）。他们发现了许多错误，指出了需要更多解释的地方，并对表达、用词和代码提出了许多修改建议，他们是Gavin Bowe、Allen Briggs、Dave Butenhof、Wan-Teh Chang、Chris Cleeland、Bob Friesenhahn、Andrew Gierth、Scott Johnson、Marty Leisner、Larry McVoy、Craig Metz、Bob Nelson、Steve Rago、Jim Reid、Swamy K. Sitarama、Jon C. Snader、Ian Lance Taylor、Rich Teer和Andy Tucker。

下列诸位通过电子邮件回答过我的问题，有人甚至回答过很多问题。澄清这些问题提高了本书的准确性并改进了语言表达，他们是David Bausum、Dave Butenhof、Bill Gallmeister、Mukesh Kacker、Brian Kernighan、Larry McVoy、Steve Rago、Keith Skowran、Bart Smaalders、Andy Tucker和John Wait。

特别感谢GSquared的Larry Rafsky提供了很多帮助。像以往一样，感谢国家光学天文台（NOAO）、Sidney Wolff、Richard Wolff和Steve Grandi，他们为我提供了网络与主机的访问权限。DEC公司的Jim Bound、Matt Thomas、Mary Clouter和Barb Glover提供了用于本书多数示例的Alpha系统。书中的一部分代码是在其他Unix系统上测试的：感谢Red Hat软件公司的Michael Johnson提供了最新版本的Red Hat Linux，感谢IBM奥斯汀实验室的Dave Marquardt和Jessie Haug提供了RS/6000系统以及最新版本的AIX的访问权限。

最后还要感谢Prentice Hall的优秀员工（本书的编辑Mary Franz，还有Noreen Regina、Sophie Papanikolaou和Patti Guerrieri）给予的帮助，尤其是在很紧的时间内完成一切所付出的努力。

版权说明

我制作了本书的最终电子版（PostScript格式），最后排版成现在的书。我用James Clark编写的优秀的`groff`包为本书排版，该软件包安装在一台运行Solaris 2.6的SparcStation工作站上。（认为troff已经过时的报导显然太夸张了。）我使用`vi`编辑器键入了所有的138 897个单词，用`gpic`程序绘制了72幅插图（其中用到了许多由Gary Wright编写的宏），用`gtbl`程序生成了35张表格，为全书添加了索引（用到了Jon Bentley与Brian Kernighan编写的一组`awk`脚本），并设计了

① 书中所有示例的源代码也可以从图灵网站（www.turingbook.com）本书网页免费注册下载。——编者注

最终的版式。我录入书中的8 046行C语言源代码，使用的是Dave Hanson的loom程序、GNU的indent程序和Gary Wright写的一些脚本。

欢迎读者以电子邮件的方式反馈意见、提出建议或订正错误。

W. Richard Stevens
1998年7月于亚利桑那州图森市
http://www.kohala.com/~rstevens

目　录

第一部分　简介

第 1 章　简介 …… 2
1.1　概述 …… 2
1.2　进程、线程与信息共享 …… 3
1.3　IPC对象的持续性 …… 4
1.4　名字空间 …… 5
1.5　fork、exec和exit对IPC对象的影响 …… 7
1.6　出错处理：包裹函数 …… 8
1.7　Unix标准 …… 9
1.8　书中IPC例子索引表 …… 11
1.9　小结 …… 13
习题 …… 13
第 2 章　Posix IPC …… 14
2.1　概述 …… 14
2.2　IPC名字 …… 14
2.3　创建与打开IPC通道 …… 16
2.4　IPC权限 …… 18
2.5　小结 …… 19
习题 …… 19
第 3 章　System V IPC …… 20
3.1　概述 …… 20
3.2　key_t键和ftok函数 …… 20
3.3　ipc_perm结构 …… 22
3.4　创建与打开IPC通道 …… 22
3.5　IPC权限 …… 24
3.6　标识符重用 …… 25
3.7　ipcs和ipcrm程序 …… 27
3.8　内核限制 …… 27
3.9　小结 …… 28
习题 …… 29

第二部分　消息传递

第 4 章　管道和 FIFO …… 32
4.1　概述 …… 32
4.2　一个简单的客户-服务器例子 …… 32
4.3　管道 …… 32
4.4　全双工管道 …… 37
4.5　popen和pclose函数 …… 39
4.6　FIFO …… 40
4.7　管道和FIFO的额外属性 …… 44
4.8　单个服务器，多个客户 …… 46
4.9　对比迭代服务器与并发服务器 …… 50
4.10　字节流与消息 …… 51
4.11　管道和FIFO限制 …… 55
4.12　小结 …… 56
习题 …… 57
第 5 章　Posix 消息队列 …… 58
5.1　概述 …… 58
5.2　mq_open、mq_close和mq_unlink函数 …… 59
5.3　mq_getattr和mq_setattr函数 …… 61
5.4　mq_send和mq_receive函数 …… 64
5.5　消息队列限制 …… 67
5.6　mq_notify函数 …… 68
5.7　Posix实时信号 …… 78
5.8　使用内存映射I/O实现Posix消息队列 …… 85
5.9　小结 …… 101
习题 …… 101
第 6 章　System V 消息队列 …… 103
6.1　概述 …… 103
6.2　msgget函数 …… 104
6.3　msgsnd函数 …… 104

6.4 msgrcv函数……105
6.5 msgctl函数……106
6.6 简单的程序……107
6.7 客户-服务器例子……112
6.8 复用消息……113
6.9 消息队列上使用select和poll……121
6.10 消息队列限制……122
6.11 小结……124
习题……124

第三部分 同步

第 7 章 互斥锁和条件变量……126
7.1 概述……126
7.2 互斥锁：上锁与解锁……126
7.3 生产者-消费者问题……127
7.4 对比上锁与等待……131
7.5 条件变量：等待与信号发送……132
7.6 条件变量：定时等待和广播……136
7.7 互斥锁和条件变量的属性……136
7.8 小结……139
习题……139
第 8 章 读写锁……140
8.1 概述……140
8.2 获取与释放读写锁……140
8.3 读写锁属性……141
8.4 使用互斥锁和条件变量实现读写锁……142
8.5 线程取消……148
8.6 小结……153
习题……153
第 9 章 记录上锁……154
9.1 概述……154
9.2 对比记录上锁与文件上锁……157
9.3 Posix fcntl记录上锁……158
9.4 劝告性上锁……162
9.5 强制性上锁……164
9.6 读出者和写入者的优先级……166
9.7 启动一个守护进程的唯一副本……170
9.8 文件作锁用……171
9.9 NFS上锁……173
9.10 小结……173
习题……174
第 10 章 Posix 信号量……175
10.1 概述……175
10.2 sem_open、sem_close和sem_unlink函数……179
10.3 sem_wait和sem_trywait函数……180
10.4 sem_post和sem_getvalue函数……180
10.5 简单的程序……181
10.6 生产者-消费者问题……186
10.7 文件上锁……190
10.8 sem_init和sem_destroy函数……191
10.9 多个生产者，单个消费者……193
10.10 多个生产者，多个消费者……196
10.11 多个缓冲区……199
10.12 进程间共享信号量……205
10.13 信号量限制……206
10.14 使用FIFO实现信号量……206
10.15 使用内存映射I/O实现信号量……210
10.16 使用System V信号量实现Posix信号量……218
10.17 小结……224
习题……225
第 11 章 System V 信号量……226
11.1 概述……226
11.2 semget函数……227
11.3 semop函数……229
11.4 semctl函数……231
11.5 简单的程序……232
11.6 文件上锁……237
11.7 信号量限制……238
11.8 小结……242
习题……242

第四部分 共享内存区

第 12 章 共享内存区介绍……244
12.1 概述……244
12.2 mmap、munmap和msync函数……247
12.3 在内存映射文件中给计数器持续加1……250
12.4 4.4BSD匿名内存映射……254
12.5 SVR4 /dev/zero内存映射……255

12.6　访问内存映射的对象……255
12.7　小结……259
习题……260
第 13 章　Posix 共享内存区……261
13.1　概述……261
13.2　shm_open和shm_unlink函数……261
13.3　ftruncate和fstat函数……262
13.4　简单的程序……263
13.5　给一个共享的计数器持续加1……267
13.6　向一个服务器发送消息……270
13.7　小结……275
习题……275
第 14 章　System V 共享内存区……276
14.1　概述……276
14.2　shmget函数……276
14.3　shmat函数……277
14.4　shmdt函数……277
14.5　shmctl函数……277
14.6　简单的程序……278
14.7　共享内存区限制……281
14.8　小结……282
习题……283
第五部分　远程过程调用
第 15 章　门……286
15.1　概述……286
15.2　door_call函数……291
15.3　door_create函数……292
15.4　door_return函数……293
15.5　door_cred函数……294
15.6　door_info函数……294
15.7　例子……295
15.8　描述符传递……306
15.9　door_sever_create函数……310
15.10　door_bind、door_unbind和door_revoke函数……315
15.11　客户或服务器的过早终止……315
15.12　小结……321
习题……322
第 16 章　Sun RPC……323
16.1　概述……323
16.2　多线程化……330
16.3　服务器捆绑……333
16.4　认证……336
16.5　超时和重传……338
16.6　调用语义……342
16.7　客户或服务器的过早终止……343
16.8　XDR：外部数据表示……345
16.9　RPC分组格式……361
16.10　小结……365
习题……366
后记……368
附录 A　性能测量……371
附录 B　线程入门……406
附录 C　杂凑的源代码……409
附录 D　精选习题解答……417
参考文献……433
索引……435

第一部分

简　介



第1章 简 介

1.1 概述

IPC是进程间通信（interprocess communication）的简称。传统上该术语描述的是运行在某个操作系统之上的不同进程间各种消息传递（message passing）的方式。本书还讲述多种形式的同步（synchronization），因为像共享内存区这样的较新式的通信需要某种形式的同步参与运作。

在Unix操作系统过去30年的演变史中，消息传递历经了如下几个发展阶段。

- 管道（pipe，第4章）是第一个广泛使用的IPC形式，既可在程序中使用，也可从shell中使用。管道的问题在于它们只能在具有共同祖先（指父子进程关系）的进程间使用，不过该问题已随有名管道（named pipe）即FIFO（第4章）的引入而解决了。
- System V消息队列（System V message queue，第6章）是在20世纪80年代早期加到System V内核中的。它们可用在同一主机上有亲缘关系或无亲缘关系的进程之间。尽管称呼它们时仍冠以“System V”前缀，当今多数版本的Unix却不论自己是否源自System V都支持它们。

> 在谈论Unix进程时，有亲缘关系（related）的说法意味着所论及的进程具有某个共同的祖先。说得更明白点，这些有亲缘关系的进程是从该祖先进程经过一次或多次fork派生来的。一个常见的例子是在某个进程调用fork两次，派生出两个子进程。我们说这两个子进程是有亲缘关系的。同样，每个子进程与其父进程也是有亲缘关系的。考虑到IPC，父进程可以在调用fork前建立某种形式的IPC（例如管道或消息队列），因为它知道随后派生的两个子进程将穿越fork继承该IPC对象。我们随图1-6详细讨论各种IPC对象的继承性。我们还得注意，从理论上说，所有Unix进程与init进程都有亲缘关系，它是在系统自举时启动所有初始化进程的祖先进程。然而从实践上说，进程亲缘关系开始于一个登录shell（称为一个会话）以及由该shell派生的所有进程。APUE的第9章详细讨论会话和进程亲缘关系。
>
> 本书将全文使用缩进的插入式注解（如此处所示）来说明实现上的细节、历史上的观点以及其他琐事。

- Posix消息队列（Posix消息队列，第5章）是由Posix实时标准（1003.1b-1993，将在1.7节详细讨论）加入的。它们可用在同一主机上有亲缘关系和无亲缘关系的进程之间。
- 远程过程调用（Remote Procedure Call，简称RPC，第5部分）出现在20世纪80年代中期，它是从一个系统（客户主机）上某个程序调用另一个系统（服务器主机）上某个函数的一种方法，是作为显式网络编程的一种替换方法开发的。既然客户和服务器之间通常传递一些信息（被调用函数的参数与返回值），而且RPC可用在同一主机上的客户和服务器之间，因此可认为RPC是另一种形式的消息传递。

看一看由Unix提供的各种同步形式的演变同样颇有教益。

- 需要某种同步形式（往往是为了防止多个进程同时修改同一文件）的早期程序使用了文件系统的诡秘特性，我们将在9.8节讨论其中的一些。
- 记录上锁（record locking，第9章）是在20世纪80年代早期加到Unix内核中的，然后在1988年由Posix.1标准化的。
- System V信号量（System V semaphore，第11章）是在System V消息队列加入System V内核的同时（20世纪80年代早期）伴随System V共享内存区（System V shared memory）加入的。当今多数版本的Unix都支持它们。
- Posix信号量（Posix semaphore，第10章）和Posix共享内存区（Posix shared memory，第13章）也由Posix实时标准（1003.1b-1993）加入。
- 互斥锁（mutex）和条件变量（condition variable，见第7章）是由Posix线程标准（1003.1c-1995）定义的两种同步形式。尽管往往用于线程间的同步，它们也能提供不同进程间的同步。
- 读写锁（read-write lock，第8章）是另一种形式的同步。它们还没有被Posix标准化，不过也许不久后会被标准化。

1.2 进程、线程与信息共享

按照传统的Unix编程模型，我们在一个系统上运行多个进程，每个进程都有各自的地址空间。Unix进程间的信息共享可以有多种方式。图1-1对此作了总结。

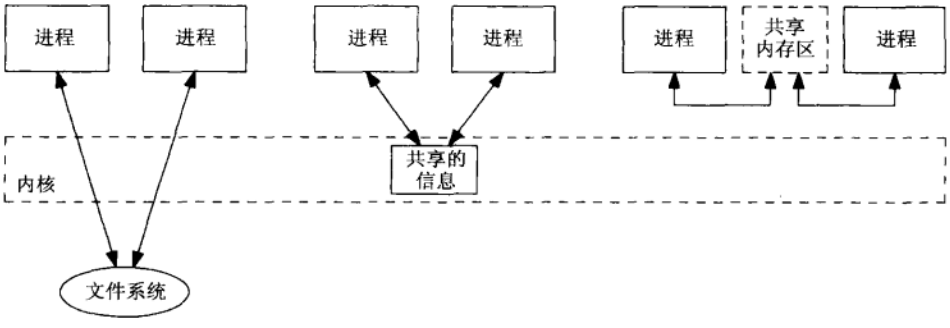


图1-1 Unix进程间共享信息的三种方式

(1) 左边的两个进程共享存留于文件系统中某个文件上的某些信息。为访问这些信息，每个进程都得穿越内核（例如read、write、lseek等）。当一个文件有待更新时，某种形式的同步是必要的，这样既可保护多个写入者，防止相互串扰，也可保护一个或多个读出者，防止写入者的干扰。

(2) 中间的两个进程共享驻留于内核中的某些信息。管道是这种共享类型的一个例子，System V消息队列和System V信号量也是。现在访问共享信息的每次操作涉及对内核的一次系统调用。

(3) 右边的两个进程有一个双方都能访问的共享内存区。每个进程一旦设置好该共享内存区，就能根本不涉及内核而访问其中的数据。共享该内存区的进程需要某种形式的同步。

注意没有任何东西限制任何IPC技术只能使用两个进程。我们讲述的技术适用于任意数目的进程。在图1-1中只展示两个进程是为了简单起见。

线程

5 虽然Unix系统中进程的概念已使用了很久，一个给定进程内多个线程（thread）的概念却相对较新。Posix.1线程标准（称为“Pthreads”）是于1995年通过的。从IPC角度看来，一个给定进程内的所有线程共享同样的全局变量（也就是说共享内存区的概念对这种模型来说是内在的）。然而我们必须关注的是各个线程间对全局数据的同步访问。同步尽管不是一种明确的IPC形式，但它确实伴随许多形式的IPC使用，以控制对某些共享数据的访问。

本书中我们讲述进程间的IPC和线程间的IPC。我们假设有一个线程环境，并作类似如下形式的陈述：“如果管道为空，调用线程就阻塞在它的read调用上，直到某个线程往该管道写入数据。”要是你的系统不支持线程，那你可以将该句子中的“线程”替换成“进程”，从而提供“阻塞在对空管道的read调用上”的经典Unix定义。然而在支持线程的系统上，只有对空管道调用read的那个线程阻塞，同一进程中的其余线程才可以继续执行。向该空管道写数据的工作既可以由同一进程中的另一个线程去做，也可以由另一个进程中的某个线程去做。

附录B汇总了线程的某些特征以及全书都用到的5个基本的Pthread函数。

1.3　IPC对象的持续性

我们可以把任意类型的IPC的持续性（persistence）定义成该类型的一个对象一直存在多长时间。图1-2展示了三种类型的持续性。

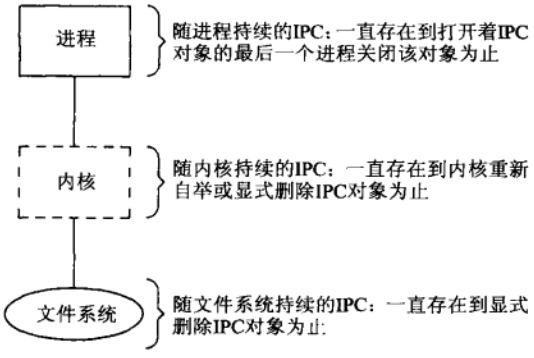


图1-2　IPC对象的持续性

(1) 随进程持续的（process-persistent）IPC对象一直存在到打开着该对象的最后一个进程关闭该对象为止。例如管道和FIFO就是这种对象。

(2) 随内核持续的（kernel-persistent）IPC对象一直存在到内核重新自举或显式删除该对象为止。例如System V的消息队列、信号量和共享内存区就是此类对象。Posix的消息队列、信号
6 量和共享内存区必须至少是随内核持续的，但也可以是随文件系统持续的，具体取决于实现。

(3) 随文件系统持续的（filesystem-persistent）IPC对象一直存在到显式删除该对象为止。即使内核重新自举了，该对象还是保持其值。Posix消息队列、信号量和共享内存区如果是使用映射文件实现的（不是必需条件），那么它们就是随文件系统持续的。

在定义一个IPC对象的持续性时我们必须小心，因为它并不总是像看起来的那样。例如管道内的数据是在内核中维护的，但管道具备的是随进程的持续性而不是随内核的持续性：最后

一个将某个管道打开着用于读的进程关闭该管道后，内核将丢弃所有的数据并删除该管道。类似地，尽管FIFO在文件系统中有名字，它们也只是具备随进程的持续性，因为最后一个将某个FIFO打开着的进程关闭该FIFO后，FIFO中的数据都被丢弃。

图1-3汇总了将在本书中讲述的各种类型IPC对象的持续性。

IPC类型	持续性
管道	随进程
FIFO	随进程
Posix互斥锁	随进程
Posix条件变量	随进程
Posix读写锁	随进程
fcntl记录上锁	随进程
Posix消息队列	随内核
Posix有名信号量	随内核
Posix基于内存的信号量	随进程
Posix共享内存区	随内核
System V消息队列	随内核
System V信号量	随内核
System V共享内存区	随内核
TCP套接字	随进程
UDP套接字	随进程
Unix域套接字	随进程

图1-3 各种类型IPC对象的持续性

注意该列表中没有任何类型的IPC具备随文件系统的持续性，但是我们说过有三种类型的Posix IPC可能会具备该持续性，这取决于它们的实现。显然，向一个文件写入数据提供了随文件系统的持续性，但这通常不作为一种IPC形式使用。多数形式的IPC并没有在系统重新自举后继续存在的打算，因为进程不可能跨越重新自举继续存活。对于一种给定形式的IPC，要求它具备随文件系统的持续性可能会使其性能降级，而IPC的一个基本的设计目标是高性能。

1.4 名字空间

当两个或多个无亲缘关系的进程使用某种类型的IPC对象来彼此交换信息时，该IPC对象必须有一个某种形式的名字（name）或标识符（identifier），这样其中一个进程（往往是服务器）可以创建该IPC对象，其余进程则可以指定同一个IPC对象。

管道没有名字（因此不能用于无亲缘关系的进程间），但是FIFO有一个在文件系统中的Unix路径名作为其标识符（因此可用于无亲缘关系的进程间）。在以后各章具体讲述其他形式的IPC时，我们将使用另外的命名约定。对于一种给定的IPC类型，其可能的名字的集合称为它的名字空间（name space）。名字空间非常重要，因为对于除普通管道以外的所有形式的IPC来说，名字是客户与服务器彼此连接以交换消息的手段。

图1-4汇总了不同形式的IPC所用的命名约定。

我们还指出哪些形式的IPC是由1996年版的Posix.1和Unix 98标准化的，这两个标准本身则在1.7节详细讨论。为了比较的目的，我们还包含了3种类型的套接字，它们在UNPv1中具体讲述。注意套接字API（应用程序编程接口）是由Posix.1g工作组标准化的，最终应该成为某个未来的Posix.1标准的一部分。

IPC类型	用于打开或创建IPC的名字空间	IPC打开后的标识	Posix.1 1996	Unix 98
管道	（没有名字）	描述符	•	•
FIFO	路径名	描述符	•	•
Posix互斥锁	（没有名字）	pthread_mutex_t指针	•	•
Posix条件变量	（没有名字）	pthread_cond_t指针	•	•
Posix读写锁	（没有名字）	pthread_rwlock_t指针		•
fcntl记录上锁	路径名	描述符	•	•
Posix消息队列	Posix IPC名字	mqd_t值	•	•
Posix有名信号量	Posix IPC名字	sem_t指针	•	•
Posix基于内存的信号量	（没有名字）	sem_t指针	•	•
Posix共享内存区	Posix IPC名字	描述符	•	•
System V消息队列	key_t键	System V IPC标识符		•
System V信号量	key_t键	System V IPC标识符		•
System V共享内存区	key_t键	System V IPC标识符		•
门	路径名	描述符		
Sun RPC	程序/版本	RPC句柄		
TCP套接字	IP地址与TCP端口	描述符	.1g	•
UDP套接字	IP地址与UDP端口	描述符	.1g	•
Unix域套接字	路径名	描述符	.1g	•

图1-4　各种形式IPC的名字空间

尽管Posix.1标准化了信号量，它们仍然是可选的特性。图1-5汇总了Posix.1和Unix 98对各种IPC特性的说明。每种特性有强制、未定义和可选三种选择。对于可选的特性，我们指出了其中每种特性受支持时（通常在<unistd.h>头文件中）定义的常值的名字，例如_POSIX_THREADS。

8

注意，Unix 98是Posix.1的超集。

IPC类型	Posix.1 1996	Unix 98
管道	强制	强制
FIFO	强制	强制
Posix互斥锁	_POSIX_THREADS	强制
Posix条件变量	_POSIX_THREADS	强制
进程间共享的互斥锁/条件变量	_POSIX_THREADS_PROCESS_SHARED	强制
Posix读写锁	（未定义）	强制
fcntl记录上锁	强制	强制
Posix消息队列	_POSIX_MESSAGE_PASSING	_XOPEN_REALTIME
Posix信号量	_POSIX_SEMAPHORES	_XOPEN_REALTIME
Posix共享内存区	_POSIX_SHARED_MEMORY_OBJECTS	_XOPEN_REALTIME
System V消息队列	（未定义）	强制
System V信号量	（未定义）	强制
System V共享内存区	（未定义）	强制
门	（未定义）	（未定义）
Sun RPC	（未定义）	（未定义）
mmap	_POSIX_MAPPED_FILE或_POSIX_SHARED_MEMORY_OBJECTS	强制
实时信号	_POSIX_REALTIME_SIGNALS	_XOPEN_REALTIME

图1-5　各种形式IPC的可用性

1.5 fork、exec和exit对IPC对象的影响

我们需要理解fork、exec和_exit函数对于所讨论的各种形式的IPC的影响（_exit是由exit调用的一个函数）。图1-6对此作了总结。

IPC类型	fork	exec	_exit
管道和FIFO	子进程取得父进程的所有打开着的描述符的副本	所有打开着的描述符继续打开着，除非已设置描述符的FD_CLOEXEC位	关闭所有打开着的描述符，最后一个关闭时删除管道或FIFO中残留的所有数据
Posix消息队列	子进程取得父进程的所有打开着的消息队列描述符的副本	关闭所有打开着的消息队列描述符	关闭所有打开着的消息队列描述符
System V消息队列	没有效果	没有效果	没有效果
Posix互斥锁和条件变量	若驻留在共享内存区中而且具有进程间共享属性，则共享	除非在继续打开着的共享内存区中而且具有进程间共享属性，否则消失	除非在继续打开着的共享内存区中而且具有进程间共享属性，否则消失
Posix读写锁	若驻留在共享内存区中而且具有进程间共享属性，则共享	除非在继续打开着的共享内存区中而且具有进程间共享属性，否则消失	除非在继续打开着的共享内存区中而且具有进程间共享属性，否则消失
Posix基于内存的信号量	若驻留在共享内存区中而且具有进程间共享属性，则共享	除非在继续打开着的共享内存区中而且具有进程间共享属性，否则消失	除非在继续打开着的共享内存区中而且具有进程间共享属性，否则消失
Posix有名信号量	父进程中所有打开着的有名信号量在子进程中继续打开着	关闭所有打开着的有名信号量	关闭所有打开着的有名信号量
System V信号量	子进程中所有semadj值都置为0	所有semadj值都携入新程序中	所有semadj值都加到相应的信号量值上
fcntl记录上锁	子进程不继承由父进程持有的锁	只要描述符继续打开着，锁就不变	解开由进程持有的所有未处理的锁
mmap内存映射	父进程中的内存映射存留到子进程中	去除内存映射	去除内存映射
Posix共享内存区	父进程中的内存映射存留到子进程中	去除内存映射	去除内存映射
System V共享内存区	附接着的共享内存区在子进程中继续附接着	断开所有附接着的共享内存区	断开所有附接着的共享内存区
门	子进程取得父进程的所有打开着的描述符，但是客户在门描述符上激活其过程时，只有父进程是服务器	所有门描述符都应关闭，因为它们创建时设置了FD_CLOEXEC位	关闭所有打开着的描述符

图1-6 调用fork、exec和_exit对于IPC的影响

表中多数特性将在以后的章节中讲述，不过我们需要强调几点。首先，考虑到无名同步变量（互斥锁、条件变量、读写锁、基于内存的信号量），从一个具有多个线程的进程中调用fork将变得混乱不堪。[Butenhof 1997]的6.1节提供了其中的细节。我们在表中只是简单地注明：如果这些变量驻留在共享内存区中，而且创建时设置了进程间共享属性，那么对于能访问该共享内存区的任意进程来说，其任意线程能继续访问这些变量。其次，System V IPC的三种形式没有打开或关闭的说法。我们将从图6-8和习题11.1和习题14.1中看出，访问这三种形式的IPC所需知道的只是一个标识符，因此知道该标识符的任何进程都能访问它们，尽管信号量和共享内存区可附带提出某种特殊处理要求。

1.6　出错处理：包裹函数

在现实程序中，我们必须检查每个函数调用是否返回错误。由于碰到错误时终止程序执行是个惯例，因此我们可以通过定义包裹函数（wrapper function）来缩短程序的长度。包裹函数执行实际的函数调用，测试其返回值，并在碰到错误时终止进程。我们使用的命名约定是将函数名第一个字母改为大写字母，例如：

```
Sem_post(ptr);
```

图1-7定义了这个包裹函数。

lib/wrapunix.c

```
387 void
388 Sem_post(sem_t *sem)
389 {
390     if (sem_post(sem) == -1)
391         err_sys("sem_post error");
392 }
```

lib/wrapunix.c

图1-7　sem_post函数的包裹函数

每当你遇到一个以大写字母打头的函数名时，它就是我们所说的包裹函数。它调用一个名字相同但以相应小写字母开头的实际函数。当碰到错误时，包裹函数总是在输出一个出错消息后终止。

我们在讲解书中提供的源代码时，所指代的总是被调用的最低层函数（例如sem_post），而不是包裹函数（例如Sem_post）。类似地，书后的索引也总是指代被调用的最低层函数，而不是指代包裹函数。

> 刚刚展示的源代码格式全书都在使用。每一非空行都被编号。代码的正文说明部分的左边标有起始与结束的行号。有的段落开始处含有一个醒目的简短标题，概述本段代码的内容。
>
> 源代码片段起始与结束处的水平划线标出了该片段所在源代码文件名，本例中就是lib目录下的wrapunix.c文件。既然本书所有例子的源代码都可免费获得（见前言），你就可以凭这个文件名找到相应的文件。阅读本书的过程中，编译、运行并修改这些程序是学习进程间通信概念的好方法。

尽管包裹函数不见得如何节省代码量，当在第7章中讨论线程时，我们会发现线程函数出错时并不设置标准的Unix errno变量；相反，本该设置errno的值改由线程函数作为其返回值返回调用者。这意味着我们每次调用任意一个线程函数时，都得分配一个变量来保存函数返回值，然后在调用我们的err_sys函数（图C-4）前，把errno设置成所保存的值。为避免源代码中到处出现花括弧，我们可以使用C语言的逗号运算符，把给errno赋值与调用err_sys组合成单个语句，如下所示：

11

```
int    n;

if ( (n = pthread_mutex_lock(&ndone_mutex)) != 0)
    errno = n, err_sys("pthread_mutex_lock error");
```

另一种办法是定义一个新的出错处理函数，它需要的另一个参数是系统的错误号[①]。但是我们可以将这段代码简化得更容易些：

```
Pthread_mutex_lock(&ndone_mutex);
```

① 意思是用新的出错处理函数代替原来的err_sys，这样对线程函数的调用可直接作为新的出错处理函数中增设的参数用，即err_sys_new("pthread_mutex_lock error", pthread_mutex_lock(&ndone_mutex))。——译者注

其前提是定义自己的包裹函数，如图1-8所示。

lib/wrappthread.c

```
125 void
126 Pthread_mutex_lock(pthread_mutex_t *mptr)
127 {
128     int     n;

129     if ( (n = pthread_mutex_lock(mptr)) == 0)
130         return;
131     errno = n;
132     err_sys("pthread_mutex_lock error");
133 }
```

lib/wrappthread.c

图1-8 给pthread_mutex_lock定义的包裹函数

仔细推敲编码，我们可改用宏代替函数，从而稍稍提高运行效率，不过即使有过的话，包裹函数也很少是程序性能的瓶颈所在。

选择将函数名的第一个字母大写是一种较折中的方法。还有许多其他方法：例如用e作为函数名的前缀（如［Kernighan and Pike 1984］第184页所示），或者用_e作为函数名的后缀等。同样提供确实在调用某个其他函数的可视化指示，我们的方法看来是最少分散人们的注意力的。

这种技巧还有助于检查那些其错误返回值通常被忽略的函数，例如close和pthread_mutex_lock。

本书后面的例子中我们将普遍使用包裹函数，除非必须检查某个确定的错误并处理它（而不是终止进程）。我们并不给出所有包裹函数的源代码，但它们是免费可得的（见前言）。

Unix errno 值

每当在一个Unix函数中发生错误时，全局变量errno将被设置成一个指示错误类型的正数，函数本身则通常返回-1。我们的err_sys函数检查errno的值并输出相应的出错消息，例如，errno的值等于EAGAIN时的出错消息为“Resource temporarily unavailable”（资源暂时不可用）。

errno的值只在某个函数发生错误时设置。如果该函数不返回错误，errno的值就无定义。所有正的错误值都是常值，具有以E打头的全部为大写字母的名字，通常定义在头文件<sys/errno.h>中。没有值为0的错误。 12

在多线程环境中，每个线程必须有自己的errno变量。提供一个局限于线程的errno变量的隐式请求是自动处理的，不过通常需要告诉编译器所编译的程序是可重入的。给编译器指定类似-D_REENTRANT或-D_POSIX_C_SOURCE=199506L这样的命令行选项是较典型的方法。<errno.h>头文件往往把errno定义成一个宏，当常值_REENTRANT有定义时，该宏就扩展成一个函数，由它访问errno变量的某个局限于线程的副本。

全书使用类似“mq_send函数返回EMSGSIZE错误”的用语来简略地表示这样的意思：该函数返回一个错误（典型情况是返回值为-1），并且在errno中设置了指定的常值。

1.7 Unix 标准

有关Unix标准化的大多数活动是由Posix和Open Group做的。

1.7.1 Posix

Posix是“可移植操作系统接口”（Portable Operating System Interface）的首字母缩写。它并

不是一个单一标准，而是一个由电气与电子工程师学会即*IEEE*开发的一系列标准。Posix标准还是由ISO（国际标准化组织）和IEC（国际电工委员会）采纳的国际标准，这两个组织合称为ISO/IEC。Posix标准经历了以下若干代。

- IEEE Std 1003.1-1988（共317页）是第一个Posix标准。它说明进入类Unix内核的C语言接口，涉及下列领域：进程原语（fork、exec、信号、定时器）、进程环境（用户ID、进程组）、文件与目录（所有I/O函数）、终端I/O、系统数据库（口令文件和用户组文件）、tar与cpio归档格式。

  第一个Posix标准是出现于1986年称为“IEEEIX”的试用版本。Posix这个名字是由Richard Stallman建议使用的。

- IEEE Std 1003.1-1990（共356页）是下一个Posix标准，它也是国际标准ISO/IEC 9945-1:1990。从1988年版本到1990年版本只做了少量的修改。新添的副标题为“Part 1: System Application Program Interface (API) [C Language]”，指示本标准为C语言API。
- IEEE Std 1003.2-1992出版成两卷本，共约1300页，其副标题为“Part2: Shell and Utilities”。这一部分定义了shell（基于System V的Bourne shell）和大约100个实用程序（即通常从shell启动执行的程序，包括awk、basename、vi和yacc等）。本书称这个标准为*Posix.2*。
- IEEE Std 1003.1b-1993（共590页）先前称为IEEE P1003.4。这是对1003.1-1990标准的更新，添加了由P1003.4工作组开发的实时扩展：文件同步、异步I/O、信号量、内存管理（mmap和共享内存区）、执行调度、时钟与定时器、消息队列。
- IEEE Std 1003.1，1996年版［IEEE 1996］包括1003.1-1990（基本API）、1003.lb-1993（实时扩展）、1003.1c-1995 （Pthreads）和1003.1i-1995（对1003.1b的技术性修正）。这个标准也称为ISO/IEC 9945-1: 1996。其中增加了三章线程内容以及有关线程同步（互斥锁和条件变量）、线程调度和同步调度的额外各节。本书称这个标准为*Posix.1*。

  743页中有四分之一强的篇幅是标题为“Rationale and Notes”（原理与注解）的附录。这些原理含有历史性信息以及某些特性必须加入或删除的理由，它们通常跟正式标准一样有教益。

  遗憾的是IEEE标准在因特网上不是免费可得的。其订购信息在［IEEE 1996］的参考文献说明中给出。

  注意信号量在实时标准中定义，它与在Pthreads标准中定义的互斥锁和条件变量相分离，这足以解释它们的API中存在的某些差异。

  最后注意读写锁（尚）不属于任何Posix标准。我们将在第8章中详细讨论。

将来某个时候印制的新版本的IEEE Std 1003.1应包括P1003.lg标准，它是我们在UNPvl中讲述的网络编程API（套接字和XTI）。

1996年版的Posix.1标准的前言中声称ISO/IEC 9945由下面三个部分构成。

- Part 1: System application program interface (API) [C Language]（第一部分：系统应用程序接口（API）[C语言]）。
- Part 2: Shell and utilities（第二部分：Shell和实用程序）。
- Part 3: System administration（第三部分：系统管理）（正在开发中）。

第一部分和第二部分就是我们所称的Posix.1和Posix.2。

Posix标准化工作仍将继续，任何论述到它的书籍都在跟踪这项工作。从http://www.pasc.org/standing/sdll.html可获得各种Posix标准的最新状态。

13

1.7.2 Open Group

Open Group是由X/Open公司（1984年成立）和开放软件基金会（OSF，1988年成立）于1996年合并而成的组织。它是由厂家、业界最终用户、政府部门和学术机构组成的国际组织。它们的标准经历了以下若干代。

- X/Open公司于1989年出版了“X/Open Portability Guide”（《X/Open移植性指南》）第3期（XPG3）。
- 第4期于1992年出版，这一期的第2版于1994年出版。这个最终版本也称为“Spec 1170”，其中魔数1170是系统接口数（926个）、头文件数（70个）和命令数（174个）的总和。这组规范的最终名字是“X/Open Single Unix Specification”（X/Open单一Unix规范），也称为“Unix 95”。
- 1997年3月单一Unix规范的第2版发表。符合这个规范的产品可称为“Unix 98”，这也是本书提到这个规范所用的名称。Unix 98所需的接口数从1170个增加到1434个，然而，适用于工作站的接口数却猛增到3030个，因为它包含CDE（公共桌面环境，Common Desktop Environment），而CDE又反过来要求有X Windows系统和Motif用户接口。其详情见[Josey 1997]和http://www.UNIX-systems.org/version2。

单一Unix规范的许多文档可在因特网上从这个站点免费取得。

1.7.3 Unix 版本和移植性

当今大多数Unix系统符合Posix.1和Posix.2的某个版本。我们使用限定词“某个”是因为Posix每次更新（例如1993年增加实时扩展，1996年增加Pthreads内容），厂家都得花一两年（甚至更长的时间）去实现并加入最近的更新内容。

从历史上看，多数Unix系统或者源自Berkeley，或者源自System V，不过这些差别在慢慢消失，因为大多数厂家已开始采用Posix标准。仍然存在的主要差别在于系统管理，这是一个目前还没有任何Posix标准可循的领域。

运行本书中大多数例子的平台是Solaris 2.6和Digital Unix 4.0B。其原因在于写到此处时（1997年末到1998年初），只有这两种Unix系统支持System V IPC、Posix IPC及Posix线程。

1.8 书中 IPC 例子索引表

为分析各种特性，全书主要使用了三种交互模式。

(1) 文件服务器：客户-服务器应用程序，客户向服务器发送一个路径名，服务器把该文件的内容返回给客户。

(2) 生产者-消费者：一个或多个线程或进程（生产者）把数据放到一个共享缓冲区中，另有一个或多个线程或进程（消费者）对该共享缓冲区中的数据进行操作。

(3) 序列号持续增1：一个或多个线程或进程给一个共享的序列号持续增1。该序列号有时在一个共享文件中，有时在共享内存区中。

第一个例子分析各种形式的消息传递，另外两个例子则分析各种类型的同步和共享内存区。

为了提供本书所涵盖的不同主题的索引，图1-9、图1-10和图1-11汇总了我们开发的程序及它们的源代码所在的起始图号和页码。

图　号	说　明
4-8	使用两个管道，父子进程间
4-15	使用popen和cat
4-16	使用两个FIFO，父子进程间
4-18	使用两个FIFO，独立的服务器，与服务器无亲缘关系的客户
4-23	使用多个FIFO，独立的迭代服务器，多个客户
4-25	使用管道或FIFO：在字节流上构筑记录
6-9	使用两个System V消息队列
6-15	使用一个System V消息队列，多个客户
6-20	每个客户使用一个System V消息队列，多个客户
15-18	使用穿越门的描述符传递

图1-9　不同版本的文件服务器客户-服务器例子

图　号	说　明
7-2	只用互斥锁，多个生产者，单个消费者
7-6	互斥锁和条件变量，多个生产者，单个消费者
10-17	Posix有名信号量，单个生产者，单个消费者
10-20	Posix基于内存的信号量，单个生产者，单个消费者
10-21	Posix基于内存的信号量，多个生产者，单个消费者
10-24	Posix基于内存的信号量，多个生产者，多个消费者
10-33	Posix基于内存的信号量，单个生产者，单个消费者，多个缓冲区

图1-10　不同版本的生产者-消费者例子

图　号	说　明
9-1	序列号在文件中，不上锁
9-3	序列号在文件中，fcntl上锁
9-12	序列号在文件中，使用open进行文件系统上锁
10-19	序列号在文件中，Posix有名信号量上锁
12-10	序列号在mmap共享内存区，Posix有名信号量上锁
12-12	序列号在mmap共享内存区，Posix基于内存的信号量上锁
12-14	序列号在4.4BSD匿名共享内存区，Posix有名信号量上锁
12-15	序列号在SVR4/dev/zero共享内存区，Posix有名信号量上锁
13-7	序列号在Posix共享内存区，Posix基于内存的信号量上锁
A-34	性能测量：线程间互斥锁上锁
A-36	性能测量：线程间读写锁上锁
A-39	性能测量：线程间Posix基于内存的信号量上锁
A-41	性能测量：线程间Posix有名信号量上锁
A-42	性能测量：线程间System V信号量上锁
A-45	性能测量：线程间fcntl记录上锁
A-48	性能测量：线程间互斥锁上锁

图1-11　不同版本的序列号持续增1例子

1.9 小结

IPC传统上是Unix中一个杂乱不堪的领域。虽然有了各种各样的解决办法，但没有一个是完美的。我们的讨论分成4个主要领域：

(1) 消息传递（管道、FIFO、消息队列）；

(2) 同步（互斥锁、条件变量、读写锁、信号量）；

(3) 共享内存区（匿名共享内存区、有名共享内存区）；

(4) 过程调用（Solaris门、Sun RPC）。

我们考虑单个进程中多个线程间的IPC以及多个进程间的IPC。

各种类型IPC的持续性可以是随进程持续的、随内核持续的或随文件系统持续的，这取决于IPC对象存在时间的长短。在为给定的应用选择所用的IPC类型时，我们必须清楚相应IPC对象的持续性。

各种类型IPC的另一个特性是名字空间，也就是使用IPC对象的进程和线程标识各个IPC对象的方式。各种类型的IPC有些没有名字（管道、互斥锁、条件变量、读写锁），有些具有在文件系统中的名字（FIFO），有些具有将在第2章中讲述的Posix IPC名字，有些则具有其他类型的名字（将在第3章中讲述的System V IPC键或标识符）。典型做法是：服务器以某个名字创建一个IPC对象，客户则使用该名字访问同一个IPC对象。

本书中所有源代码使用1.6节中讲述的包裹函数来缩短篇幅，同时达到检查每个函数调用是否返回错误的目的。我们的包裹函数都以一个大写字母开头。

IEEE Posix标准一直是多数厂家努力遵循的标准，其中Posix.1定义了访问Unix的基本C接口，Posix.2定义了标准命令。然而商业标准也在迅速地吸纳并扩展Posix标准，著名的有Open Group的Unix标准，例如Unix 98。 16 ~ 17

习题

1.1 图1-1中我们展示了两个进程访问单个文件的情形。如果这两个进程都只是往该文件的末尾添加新的数据（譬如说这是一个日志文件），那么需要什么类型的同步？

1.2 查看一下你的系统中的<error.h>头文件是如何定义errno变量的。

1.3 在图1-5上标注出你使用的Unix系统所支持的特性。 18

第2章 Posix IPC

2.1 概述

以下三种类型的IPC合称为“Posix IPC”：

- Posix消息队列（第5章）；
- Posix信号量（第10章）；
- Posix共享内存区（第13章）。

Posix IPC在访问它们的函数和描述它们的信息上有一些类似点。本章讲述所有这些共同属性：用于标识的路径名、打开或创建时指定的标志以及访问权限。

图2-1汇总了所有Posix IPC函数。

	消息队列	信号量	共享内存区
头文件	<mqueue.h>	<semaphore.h>	<sys/mman.h>
创建、打开或删除IPC的函数	mq_open mq_close mq_unlink	sem_open sem_close sem_unlink	shm_open shm_unlink
		sem_init sem_destroy	
控制IPC操作的函数	mq_getattr mq_setattr		ftruncate fstat
IPC操作函数	mq_send mq_receive mq_notify	sem_wait sem_trywait sem_post sem_getvalue	mmap munmap

图2-1 Posix IPC函数汇总

2.2 IPC名字

在图1-4中我们指出，三种类型的Posix IPC都使用“Posix IPC名字”进行标识。mq_open、sem_open和shm_open这三个函数的第一个参数就是这样的一个名字，它可能是某个文件系统中的一个真正的路径名，也可能不是。Posix.1是这么描述Posix IPC名字的。

- 它必须符合已有的路径名规则（必须最多由PATH_MAX个字节构成，包括结尾的空字节）。
- 如果它以斜杠符开头，那么对这些函数的不同调用将访问同一个队列。如果它不以斜杠符开头，那么效果取决于实现。
- 名字中额外的斜杠符的解释由实现定义。

因此，为便于移植起见，Posix IPC名字必须以一个斜杠符打头，并且不能再含有任何其他斜杠符。遗憾的是这些规则还不够，仍会出现移植性问题。

Solaris 2.6要求有打头的斜杠符，但是不允许有另外的斜杠符。假设要创建的是一个消息队列，创建函数将在/tmp中创建三个以.MQ开头的文件。例如，如果给mq_open的参数为/queue.

1234，那么这三个文件分别为/tmp/.MQDqueue.1234、/tmp/.MQLqueue.1234和/tmp/.MQPqueue.1234。Digital Unix 4.0B则在文件系统中创建所指定的路径名。

当我们指定一个只有单个斜杠符（作为首字符）的名字时，移植性问题就发生了：我们必须在根目录中具有写权限。例如，/tmp.1234符合Posix规则，在Solaris下也可行，但是Digital Unix却会试图创建这个文件，这时除非我们有在根目录中的写权限，否则这样的尝试将失败。如果我们指定一个/tmp/test.1234这样的名字，那么在以该名字创建一个真正文件的所有系统上都将成功（前提是/tmp目录存在，而且我们在该目录中有写权限，对于多数Unix系统来说，这是正常情况），在Solaris下则失败。

为避免这些移植性问题，我们应该把Posix IPC名字的#define行放在一个便于修改的头文件中，这样应用程序转移到另一个系统上时，只需修改这个头文件。

> 这是一个标准试图变得相当通用（本例子中，实时标准试图允许消息队列、信号量和共享内存区都在现有的Unix内核中实现，而且在独立的无盘系统上也能工作），结果标准的具体实现却变得不可移植的个例之一。在Posix中，这种现象称为“造成不标准的标准方式”（a standard way of being nonstandard）。

Posix.1定义了三个宏：

```
S_TYPEISMQ(buf)
S_TYPEISSEM(buf)
S_TYPEISSHM(buf)
```

它们的单个参数是指向某个stat结构的指针，其内容由fstat、lstat或stat这三个函数填入。如果所指定的IPC对象（消息队列、信号量或共享内存区对象）是作为一种独特的文件类型实现的，而且参数所指向的stat结构访问这样的文件类型，那么这三个宏计算出一个非零值。否则，计算出的值为0。

> 不幸的是，这三个宏没有多大用处，因为无法保证这三种类型的IPC使用一种独特的文件类型实现。举例来说，在Solaris 2.6下，这三个宏的计算结果总是0。
>
> 测试某个文件是否为给定文件类型的所有其他宏的名字都以S_IS开头，而且它们的单个参数是某个stat结构的st_mode成员。由于上面三个新宏的参数不同于其他宏，因此它们的名字改为以S_TYPEIS开头。

px_ipc_name函数

解决上述移植性问题的另一种办法是自己定义一个名为px_ipc_name的函数，它为定位Posix IPC名字而添加上正确的前缀目录。

```
#include "unpipc.h"
char *px_ipc_name(const char *name);
```

均返回：若成功则为非空指针，若出错则为NULL

> 本书中我们给自己定义的非标准系统函数都使用这样的版式：围绕函数原型和返回值的方框是虚框。开头包含的头文件通常是我们的unpipc.h（图C-1）。

*name*参数中不能有任何斜杠符。例如，调用

```
px_ipc_name("test1")
```

在Solaris 2.6下返回一个指向字符串/test1的指针，在Digital Unix 4.0B下返回一个指向字符串/tmp/test1的指针。存放结果字符串的内存空间是动态分配的，并可通过调用free释放。另外，

环境变量PX_IPC_NAME能够覆盖默认目录。

图2-2给出了该函数的实现。

lib/px_ipc_name.c

```
 1 #include    "unpipc.h"

 2 char *
 3 px_ipc_name(const char *name)
 4 {
 5     char    *dir, *dst, *slash;

 6     if ( (dst = malloc(PATH_MAX)) == NULL)
 7         return(NULL);

 8         /* can override default directory with environment variable */
 9     if ( (dir = getenv("PX_IPC_NAME")) == NULL) {
10 #ifdef  POSIX_IPC_PREFIX
11         dir = POSIX_IPC_PREFIX;    /* from "config.h" */
12 #else
13         dir = "/tmp/";             /* default */
14 #endif
15     }
16         /* dir must end in a slash */
17     slash = (dir[strlen(dir) - 1] == '/') ? "" : "/";
18     snprintf(dst, PATH_MAX, "%s%s%s", dir, slash, name);

19     return(dst);                   /* caller can free() this pointer */
20 }
```

lib/px_ipc_name.c

图2-2　我们的px_ipc_name函数

这也许是你第一次碰到snprintf函数。许多现有代码调用的是sprintf，但是sprintf不检查目标缓冲区是否溢出，不过snprintf要求其第二个参数是目标缓冲区的大小，因此可确保缓冲区不溢出。提供能有意溢出一个程序的sprintf缓冲区的输入数据是黑客们已使用很多年的一种攻破系统的方法。

snprintf不是标准ANSI C的一部分，但这个标准的修订版C9X正在考虑。[①]不过，许多厂家提供的标准C函数库含有这个函数。我们在本书中使用snprintf，如果你的系统不提供这个函数，那就使用我们自己的通过调用sprintf实现的版本。

2.3　创建与打开 IPC 通道

mq_open、sem_open和shm_open这三个创建或打开一个IPC对象的函数，它们的名为*oflag*的第二个参数指定怎样打开所请求的对象。这与标准open函数的第二个参数类似。图2-3给出了可组合构成该参数的各种常值。

前3行指定怎样打开对象：只读、只写或读写。消息队列能以其中任何一种模式打开，信号量的打开不指定任何模式（任意信号量操作，都需要读写访问权），共享内存区对象则不能以只写模式打开。

① snprinf现在已经是C99标准的函数。——编者注

说　明	mq_open	sem_open	shm_open
只读	O_RDONLY		O_RDONLY
只写	O_WRONLY		
读写	O_RDWR		O_RDWR
若不存在则创建	O_CREAT	O_CREAT	O_CREAT
排他性创建	O_EXCL	O_EXCL	O_EXCL
非阻塞模式	O_NONBLOCK		
若已存在则截短			O_TRUNC

图2-3　打开或创建Posix IPC对象所用的各种*oflag*常值

图2-3中余下4行标志是可选的。

O_CREAT　若不存在则创建由函数第一个参数所指定名字的消息队列、信号量或共享内存区对象（同时检查O_EXCL标志，我们不久将要说明）。

创建一个新的消息队列、信号量或共享内存区对象时，至少需要另外一个称为*mode*的参数。该参数指定权限位，它是由图2-4中所示常值按位或形成的。

常　值	说　明
S_IRUSR	用户（属主）读
S_IWUSR	用户（属主）写
S_IRGRP	（属）组成员读
S_IWGRP	（属）组成员写
S_IROTH	其他用户读
S_IWOTH	其他用户写

图2-4　创建新的IPC对象所用的*mode*常值

这些常值定义在<sys/stat.h>头文件中。所指定的权限位受当前进程的文件模式创建掩码（file mode creation mask）修正，而该掩码可通过调用umask函数（APUE第83~85页[①]）或使用shell的umask命令来设置。

跟新创建的文件一样，当创建一个新的消息队列、信号量或共享内存区对象时，其用户ID被置为当前进程的有效用户ID。信号量或共享内存区对象的组ID被置为当前进程的有效组ID或某个系统默认组ID。新消息队列对象的组ID则被置为当前进程的有效组ID（APUE第77~78页[②]详细讨论了用户ID和组ID。）

这三种Posix IPC类型在设置组ID上存在的差异多少有点奇怪。由open新创建的文件的组ID或者是当前进程的有效组ID，或者是该文件所在目录的组ID，但是IPC函数不能假定系统为IPC对象创建了一个在文件系统中的路径名。

O_EXCL　如果该标志和O_CREAT一起指定，那么IPC函数只在所指定名字的消息队列、信号量或共享内存区对象不存在时才创建新的对象。如果该对象已经存在，而且指定了O_CREAT|O_EXCL，那么返回一个EEXIST错误。

21 ~ 23

① 此处为APUE第1版英文原版书页码，第2版为第97~100页，第2版中文版为第80~82页。——编者注

② 同样为第1版英文原版书中页码，第2版为第16~17页，第2版中文版为第12~13页。——编者注

考虑到其他进程的存在，检查所指定名字的消息队列、信号量或共享内存区对象的存在与否和创建它（如果它不存在）这两步必须是原子的（atomic）。我们将在3.4节看到适用于System V IPC的两个类似标志。

O_NONBLOCK　该标志使得一个消息队列在队列为空时的读或队列填满时的写不被阻塞。我们将在5.4节随mq_receive和mq_send这两个函数详细讨论该标志。

O_TRUNC　如果以读写模式打开了一个已存在的共享内存区对象，那么该标志将使得该对象的长度被截成0。

图2-5展示了打开一个IPC对象的真正逻辑流程。我们将在2.4节通过访问权限的测试说明该图。图2-6是展示图2-5中逻辑的另一种形式。

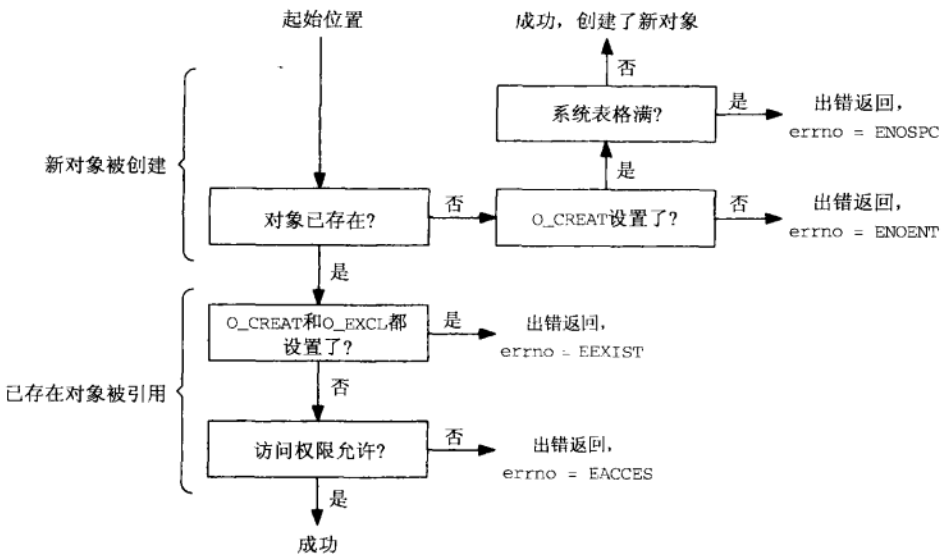


图2-5　打开或创建一个IPC对象的逻辑

*oflag*标志	对象不存在	对象已存在
无特殊标志	出错，errno = ENOENT	成功，引用已存在对象
O_CREAT	成功，创建新对象	成功，引用已存在对象
O_CREAT | O_EXCL	成功，创建新对象	出错，errno = EEXIST

图2-6　创建或打开一个IPC对象的逻辑

注意图2-6指定了O_CREAT标志但没有指定O_EXCL标志的中间那行，我们无法得到一个指示以判别是创建了一个新对象，还是在引用一个已存在的对象。

2.4　IPC权限

新的消息队列、有名信号量或共享内存区对象是由其*oflag*参数中含有O_CREAT标志的mq_open、sem_open或shm_open函数创建的。如图2-4所注，权限位与这些IPC类型的每个对象相关联，就像它们与每个Unix文件相关联一样。

当同样由这三个函数打开一个已存在的消息队列、信号量或共享内存区对象时（或者未指定O_CREAT，或者指定了O_CREAT但没有指定O_EXCL，同时对象已经存在），将基于如下信息执行权限测试：

(1) 创建时赋予该IPC对象的权限位；

(2) 所请求的访问类型（O_RDONLY、O_WRONLY或O_RDWR）；

(3) 调用进程的有效用户ID、有效组ID以及各个辅助组ID（若支持的话）。

大多数Unix内核按如下步骤执行权限测试。

(1) 如果当前进程的有效用户ID为0（超级用户），那就允许访问。

(2) 在当前进程的有效用户ID等于该IPC对象的属主ID的前提下，如果相应的用户访问权限位已设置，那就允许访问，否则拒绝访问。

这里相应的访问权限位的意思是：如果当前进程为读访问而打开该IPC对象，那么用户读权限位必须设置；如果当前进程为写访问而打开该IPC对象，那么用户写权限位必须设置。

(3) 在当前进程的有效组ID或它的某个辅助组ID等于该IPC对象的组ID的前提下，如果相应的组访问权限位已设置，那就允许访问，否则拒绝访问。

(4) 如果相应的其他用户访问权限位已设置，那就允许访问，否则拒绝访问。

这4个步骤是按所列的顺序尝试的。因此，如果当前进程拥有该IPC对象（第2步），那么访问权的授予与拒绝只依赖于用户访问权限——组访问权限绝不会考虑。类似地，如果当前进程不拥有该IPC对象，但它属于某个合适的组，那么访问权的授予与拒绝只依赖于组访问权限——其他用户访问权限绝不会考虑。 25

> 我们从图2-3中指出，sem_open不使用O_RDONLY、O_WRONLY或O_RDWR标志。然而在10.2节我们将指出，某些Unix实现采用O_RDWR，因为只要使用一个信号量，都涉及读写该信号量的值。

2.5 小结

三种类型的Posix IPC——消息队列、信号量、共享内存区——都是用路径名标识的。但是这些路径名既可以是文件系统中的实际路径名，也可以不是，而这点不一致性会导致一个移植性问题。全书采用的解决办法是使用我们自己的px_ipc_name函数。

当创建或打开一个IPC对象时，我们指定一组类似于open函数所用的标志。创建一个新的IPC对象时，我们必须给这个新对象指定访问权限，所用的是同样由open函数使用的S_*xxx*常值（见图2-4）。当打开一个已存在的IPC对象时，所执行的权限测试与打开一个已存在的文件时一样。

习题

2.1 使用Posix IPC的程序，其SUID与SGID位（APUE的4.4节）是如何影响2.4节中所述的权限测试的？

2.2 当一个程序打开一个Posix IPC对象时，它怎样才能判定是创建了一个新对象还是在引用一个已有的对象？ 26

第3章 System V IPC

3.1 概述

以下三种类型的IPC合称为System V IPC:

- System V消息队列（第6章）;
- System V信号量（第11章）;
- System V共享内存区（第14章）。

这个称谓作为这三种IPC机制的通称是因为它们源自System V Unix。System V IPC在访问它们的函数和内核为它们维护的信息上享有许多类似点。本章讲述所有这些共同属性。

图3-1汇总了所有System V IPC函数。

	消息队列	信号量	共享内存区
头文件	<sys/msg.h>	<sys/sem.h>	<sys/shm.h>
创建或打开IPC的函数	msgget	semget	shmget
控制IPC操作的函数	msgctl	semctl	shmctl
IPC操作函数	msgsnd msgrcv	semop	shmat shmdt

图3-1 System V IPC函数汇总

System V IPC函数的设计与开发信息难以找到。[Rochkind 1985] 提供了下述信息：System V消息队列、信号量和共享内存区是20世纪70年代后期在俄亥俄州哥伦布市的一个贝尔实验室分支机构开发的，他们开发了一个内部Unix版本，（顺理成章地）称为“Columbus Unix”或简称“CB Unix”。CB Unix用于“操作支持系统”（Operation Support System），即自动完成电话公司的管理和记录保存工作的事务处理系统。System V IPC大约于1983年随System V加入到商用Unix系统中。

3.2 key_t 键和 ftok 函数

图1-4中注明，三种类型的System V IPC使用key_t值作为它们的名字。头文件<sys/types.h>把key_t这个数据类型定义为一个整数，它通常是一个至少32位的整数。这些整数值通常是由ftok函数赋予的。

函数ftok把一个已存在的路径名和一个整数标识符转换成一个key_t值，称为IPC键。

```
#include <sys/ipc.h>
key_t ftok(const char *pathname, int id);
```

返回：若成功则为IPC键，若出错则为-1

该函数把从*pathname*导出的信息与*id*的低序8位组合成一个整数IPC键。

该函数假定对于使用System V IPC的某个给定应用来说，客户和服务器同意使用对该应用有一定意义的*pathname*。它可以是服务器守护程序的路径名、服务器使用的某个公共数据文件的路径名或者系统上的某个其他路径名。如果客户和服务器之间只需单个IPC通道，那么可以使用譬如说值为1的*id*。如果需要多个IPC通道，譬如说从客户到服务器一个通道，从服务器到客户又一个通道，那么作为一个例子，一个通道可使用值为1的*id*，另一个通道可使用值为2的*id*。客户和服务器一旦在*pathname*和*id*上达成一致，双方就都能调用ftok函数把*pathname*和*id*转换成同一个IPC键。

ftok的典型实现调用stat函数，然后组合以下三个值。

(1) *pathname*所在的文件系统的信息（stat结构的st_dev成员）。

(2) 该文件在本文件系统内的索引节点号（stat结构的st_ino成员）。

(3) *id*的低序8位（不能为0）。[①]

这三个值的组合通常会产生一个32位键。不能保证两个不同的路径名与同一个*id*的组合产生不同的键，因为上面所列三个条目（文件系统标识符、索引节点、*id*）中的信息位数可能大于一个整数的信息位数（见习题3.5）。 28

索引节点绝不会是0，因此大多数实现把IPC_PRIVATE（将在3.4节讲述）定义为0。

如果*pathname*不存在，或者对于调用进程不可访问，ftok就返回−1。注意，路径名用于产生键的文件不能是在服务器存活期间由服务器反复创建并删除的文件，因为该文件每次创建时由系统赋予的索引节点号很可能不一样，于是对下一个调用者来说，由ftok返回的键也可能不同。

例子

图3-2中的程序取一个作为命令行参数的路径名，调用stat，调用ftok，然后输出stat结构的st_dev和st_ino成员以及得出的IPC键。这三个值是以十六进制输出的，这样我们可从这两个值以及*id*值0x57很容易地看出IPC键是如何构造的。

svipc/ftok.c

```
 1 #include    "unpipc.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     struct stat stat;
 6     if (argc != 2)
 7         err_quit("usage: ftok <pathname>");
 8     Stat(argv[1], &stat);
 9     printf("st_dev: %lx, st_ino: %lx, key: %x\n",
10            (u_long) stat.st_dev, (u_long) stat.st_ino,
11            Ftok(argv[1], 0x57));
12     exit(0);
13 }
```

svipc/ftok.c

图3-2 获取并输出文件系统信息和IPC键

① Unix 98现在宣称：当ftok的*id*参数的低序8位为0时，该函数的行为是未指定的。查看一番后作者发现Solaris和Digital Unix中关于ftok的手册页面也作了同样的声明。作者不知道这是什么时候加上的，而且作者于1991年编写的“System V Interface Definition”中也没有这样的声明。AIX甚至走得更远，若*id*为0则返回一个错误。实际上ftok的三种不同实现——System V Release 2、GNU libc和BSD/OS——没有一个要求*id*为非零：它们只是在id的低序8位中作逻辑或，而不管它的值。

在Solaris 2.6下执行该程序的结果如下：

```
solaris % ftok /etc/system
st_dev: 800018, st_ino: 4a1b, key: 57018a1b
solaris % ftok /usr/tmp
st_dev: 800015, st_ino: 10b78, key: 57015b78
solaris % ftok /home/rstevens/Mail.out
st_dev: 80001f, st_ino: 3b03, key: 5701fb03
```

很明显，*id*在IPC键的高序8位，st_dev的低序12位IPC在键的接下来12位，st_ino的低序12位则在IPC键的低序12位。

我们展示本例子的目的不是让大家依据这种信息组合方式构造出IPC键，而是让大家看看一个实现是如何组合*pathname*和*id*的。其他实现可能以不同的方式组合。

29

FreeBSD使用*id*的低8位、st_dev的低8位以及st_ino的低16位。

注意由ftok完成的映射是单向的，因为st_dev和st_ino中某些位未被使用。这就是说，我们不能从一个给定的键确定创建它时所用的路径名。

3.3　ipc_perm结构

内核给每个IPC对象维护一个信息结构，其内容跟内核给文件维护的信息类似。

```
struct ipc_perm {
  uid_t    uid;    /* owner's user id */
  gid_t    gid;    /* owner's group id */
  uid_t    cuid;   /* creator's user id */
  gid_t    cgid;   /* creator's group id */
  mode_t   mode;   /* read-write permissions */
  ulong_t  seq;    /* slot usage sequence number */
  key_t    key;    /* IPC key */
};
```

该结构以及System V IPC函数使用的较为明显的常值定义在<sys/ipc.h>头文件中。我们将在本章讨论该结构的所有成员。

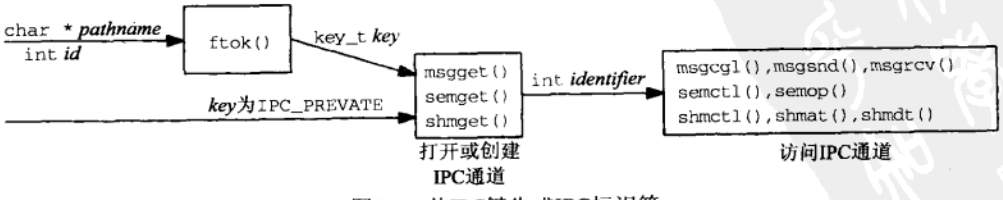
3.4　创建与打开IPC通道

创建或打开一个IPC对象的三个get*XXX*函数（见图3-1）的第一个参数*key*是类型为key_t的IPC键，返回值*identifier*是一个整数标识符。该标识符不同于ftok函数的*id*参数，我们不久就会看到。对于*key*值，应用程序有两种选择。

(1) 调用ftok，给它传递*pathname*和*id*。

(2) 指定*key*为IPC_PRIVATE，这将保证会创建一个新的、唯一的IPC对象。

图3-3展示有关步骤的顺序。



30

图3-3　从IPC键生成IPC标识符

所有三个get*XXX*函数（见图3-1）都有一个名为*oflag*的参数，它指定IPC对象的读写权限位（ipc_perm结构的mode成员），并选择是创建一个新的IPC对象还是访问一个已存在的IPC对象。这种选择的规则如下。

- 指定*key*为IPC_PRIVATE能保证创建一个唯一的IPC对象。没有一对id和pathname的组合会导致ftok产生IPC_PRIVATE这个键值。
- 设置*oflag*参数的IPC_CREAT位但不设置它的IPC_EXCL位时，如果所指定键的IPC对象不存在，那就创建一个新的对象，否则返回该对象。
- 同时设置*oflag*的IPC_CREAT和IPC_EXCL位时，如果所指定键的IPC对象不存在，那就创建一个新的对象，否则返回一个EEXIST错误，因为该对象已存在。

对IPC对象来说，IPC_CREAT和IPC_EXCL的组合跟open函数的O_CREAT和O_EXCL的组合类似。

设置IPC_EXCL位但不设置IPC_CREAT位是没有意义的。

图3-4展示了打开一个IPC对象的逻辑流程。图3-5是展示图3-4所示逻辑的另一种形式。

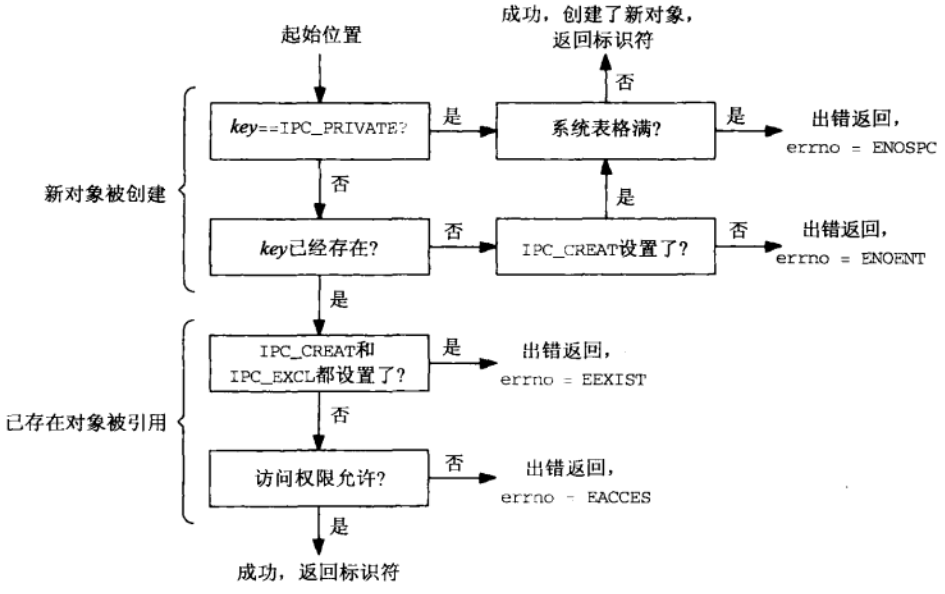


图3-4 创建或打开一个IPC对象的逻辑

*oflag*参数	*key*不存在	*key*已存在
无特殊标志	出错，errno = ENOENT	成功，引用已存在对象
IPC_CREAT	成功，创建新对象	成功，引用已存在对象
IPC_CREAT | IPC_EXCL	成功，创建新对象	出错，errno = EEXIST

图3-5 创建或打开一个IPC通道的逻辑

注意图3-5中间只有IPC_CREAT而没有IPC_EXCL标志的那一行，我们得不到一个指示以判别是创建了一个新对象，还是在引用一个已存在的对象。大多数应用程序中，由服务器创建IPC对象并指定IPC_CREAT标志（如果它不关心该对象是否存在）或IPC_CREAT | IPC_EXCL标志（如果它需要检查该对象是否已经存在）。客户则不指定其中任何一个标志（它们假定服务器已

经创建了该对象）。

System V IPC定义了自己的IPC_xxx常值，而不像标准open函数以及Posix IPC函数那样使用O_CREAT和O_EXCL常值（参见图2-3）。

还要注意的是，System V IPC函数把它们的IPC_xxx常值跟权限位（将在下一节讲述）组合到单个*oflag*参数中。open函数以及Posix IPC函数有一个名为*oflag*的参数，用以指定各种O_xxx标志，另有一个名为*mode*的参数，用以指定权限位。

3.5　IPC权限

每当使用某个get*XXX*函数（指定IPC_CREAT标志）创建一个新的IPC对象时，以下信息就保存到该对象的ipc_perm结构中（3.3节）。

(1) *oflag*参数中某些位初始化ipc_perm结构的mode成员。图3-6展示了System V三种不同IPC机制的权限位（记号>>3的意思是将值右移3位）。

31 ~ 32

数字值（八进制）	符号值			说　明
	消息队列	信号量	共享内存区	
0400	MSG_R	SEM_R	SHM_R	由用户（属主）读
0200	MSG_W	SEM_A	SHM_W	由用户（属主）写
0040	MSG_R >> 3	SEM_R >> 3	SHM_R >> 3	由（属）组成员读
0020	MSG_W >> 3	SEM_A >> 3	SHM_W >> 3	由（属）组成员写
0004	MSG_R >> 6	SEM_R >> 6	SHM_R >> 6	由其他用户读
0002	MSG_W >> 6	SEM_A>> 6	SHM_W >> 6	由其他用户写

图3-6　IPC读写权限的*mode*值

(2) cuid和cgid成员分别设置为调用进程的有效用户ID和有效组ID。这两个成员合称为创建者ID（creator ID）。

(3) ipc_perm结构的uid和gid成员也分别设置为调用进程的有效用户ID和有效组ID。这两个成员合称为属主ID（owner ID）。

尽管一个进程可通过调用相应IPC机制ctl*XXX*函数（所用命令为IPC_SET）修改属主ID，创建者ID却从不改变。这三个ctl*XXX*函数还允许一个进程修改某个IPC对象的mode成员。

多数实现在<sys/msg.h>、<sys/sem.h>和<sys/shm.h>这三个头文件中定义图3-6中所示的6个常值：MSG_R、MSG_W、SEM_R、SEM_A、SHM_R、SHM_W。不过Unix 98没有这样的要求。SEM_A中的后缀A代表“alter”（改变）。

这三个ctl*XXX*函数不使用通常的文件模式创建掩码。消息队列、信号量或共享内存区对象的权限准确地设置成由这些函数所指定的值。

Posix IPC并不允许一个IPC对象的创建者改变该对象的属主。Posix IPC中没有类似于IPC_SET命令的操作。然而如果Posix IPC名字存储在文件系统中，那么超级用户可使用chown命令改变其属主。

每当有一个进程访问某个IPC对象时，IPC就执行两级检查，该IPC对象被打开时（get*XXX*函数）执行一次，以后每次使用该对象时执行一次。

(1) 每当有一个进程以某个get*XXX*函数建立访问某个已存在IPC对象的通道时，IPC就执行一次初始检查，验证调用者的*oflag*参数没有指定不在该对象ipc_perm结构mode成员中的任何访问位。这就是图3-4中底部的方框。举例来说，一个服务器进程可以把它的输入消息队列的mode

成员设置成关掉组成员读和其他用户读这两个权限位。任何进程调用针对该消息队列的msgget函数时，如果所指定的*oflag*参数包含这两位，那么该函数都将返回一个错误。然而由get*XXX*函数完成的这种测试并没有多大用处。它隐含假定调用者知道自己属于哪个权限范畴——用户、组成员或其他用户。如果创建者特意关掉了某些权限位，而调用者却指定了这些位，那么get*XXX*函数将检测出这个错误。然而任何进程都能够完全绕过这种检查，其办法是在得知该IPC对象已存在后，简单地指定一个值为0的*oflag*参数即可。

(2) 每次IPC操作都对使用该操作的进程执行一次权限测试。举例来说，每当有一个进程试图使用msgsnd函数往某个消息队列放置一个消息时，msgsnd函数将以下面所列的顺序执行（多个）测试。一旦某个测试赋予了访问权，其后的测试就不再执行。

a) 超级用户总是赋予访问权。

b) 如果当前进程的有效用户ID等于该IPC对象的uid值或cuid值，而且相应的访问位在该IPC对象的mode成员中是打开的，那么赋予访问权。这儿“相应的访问位”的意思是，如果调用者想要在该IPC对象上执行一个读操作（例如从某个消息队列接收一个消息），那么读位必须设置，如果想要执行一个写操作，那么写位必须设置。

c) 如果当前进程的有效组ID等于该IPC对象的gid值或cgid值，而且相应的访问位在该IPC对象的mode成员中是打开的，那么赋予访问权。

d) 如果上面的测试没有一个为真，那么相应的“其他用户”访问位在该IPC对象的mode成员中必须是打开的才能赋予访问权。

3.6 标识符重用

ipc_perm结构（3.3节）还含有一个名为seq的变量，它是一个槽位使用情况序列号。该变量是一个由内核为系统中每个潜在的IPC对象维护的计数器。每当删除一个IPC对象时，内核就递增相应的槽位号，若溢出则循环回0。

我们在本节讲述的是普通SVR4实现。Unix 98没有强制使用该实现技巧。

该计数器的存在有两个原因。首先，考虑由内核维护的用于打开文件的文件描述符。它们是些小整数，只在单个进程内有意义，也就是它们是进程特定的值。如果我们试图从譬如说文件描述符4读，那么这种尝试只有该进程已在该描述符上打开了一个文件后才会凑效。它对于可能在另外一个与本进程无亲缘关系的进程中打开在文件描述符4上的文件来说，根本没有意义。然而，System V IPC标识符却是系统范围的，而不是特定于进程的。

我们从某个get函数（msgget、semget和shmget）获得一个IPC标识符（类似于文件描述符）。这些标识符也是整数，不过它们的意义适用于所有进程。举例来说，如果有两个无亲缘关系的进程（一个是客户，一个是服务器）使用单个消息队列，那么由msgget函数返回的该消息队列的标识符在双方进程中必须是同一个整数值，这样双方才能访问同一个消息队列。这种特性意味着某个行为不端的进程可能尝试从另外某个应用的消息队列中读消息，办法是尝试不同的小整数标识符，以期待找出一个当前在使用的允许大家读访问的消息队列。要是这些标识符的取值是小整数（像文件描述符那样），那么找到一个有效标识符的可能性约为1:50（假设每个进程最多有约50个描述符）。

为避免这样的问题，这些IPC机制的设计者们把标识符值的可能范围扩大到包含所有整数，而不是仅仅包含小整数。这种扩大是这么实现的：每次重用一个IPC表项时，把返回给调用进程

的标识符值增加一个IPC表项数。举例来说，如果系统配置成最多50个消息队列，那么内核中的第一个消息队列表项首次被使用时，返回给进程的标识符值为0。该消息队列被删除，从而第一个表项得以重用后，所返回的标识符为50。再下一次重用时，该标识符变为100，如此等等。既然seq变量通常作为一个无符号长整数实现（见3.3节所示的ipc_perm结构），那么该表项只有在被重用85 899 346（$2^{32}/50$，假设长整数为32位）次后才循环回0。

递增槽位使用情况序列号的另一个原因是为了避免短时间内重用System V IPC标识符。这有助于确保过早终止的服务器重新启动后不会重用标识符。

作为这种特性的一个例子，图3-7中的程序输出由msgget返回的前10个标识符值。

svmsg/slot.c

```
1 #include    "unpipc.h"
2 int
3 main(int argc, char **argv)
4 {
5     int     i, msqid;
6     for (i = 0; i < 10; i++) {
7         msqid = Msgget(IPC_PRIVATE, SVMSG_MODE | IPC_CREAT);
8         printf("msqid = %d\n", msqid);
9         Msgctl(msqid, IPC_RMID, NULL);
10    }
11    exit(0);
12 }
```

svmsg/slot.c

图3-7　连续输出由内核赋予的消息队列标识符10次

每次循环由msgget创建一个消息队列，然后由使用IPC_RMID命令的msgctl删除该队列。常值SVMSG_MODE定义在我们的unpipc.h头文件中（图C-1），它给我们的System V消息队列指定默认权限位。该程序的输出如下：

```
solaris % slot
msqid = 0
msqid = 50
msqid = 100
msqid = 150
msqid = 200
msqid = 250
msqid = 300
msqid = 350
msqid = 400
msqid = 450
```

35

如果再次运行该程序，我们就能看出槽位使用情况序列号是一个跨进程保持的内核变量。

```
solaris % slot
msqid = 500
msqid = 550
msqid = 600
msqid = 650
msqid = 700
msqid = 750
msqid = 800
msqid = 850
msqid = 900
msqid = 950
```

3.7 ipcs 和 ipcrm 程序

由于System V IPC的三种类型不是以文件系统中的路径名标识的，因此使用标准的ls和rm程序无法看到它们，也无法删除它们。不过实现了这些类型IPC的任何系统都提供两个特殊的程序：ipcs和ipcrm。ipcs输出有关System V IPC特性的各种信息，ipcrm则删除一个System V消息队列、信号量集或共享内存区。前者支持约十来个命令行选项，它们决定报告哪种类型的IPC以及输出哪些信息，后者支持6个命令行选项。所有这些选项的详细信息可查阅它们的手册页面。

System V IPC不是Posix中的内容，因此这两个命令也未被Posix.2标准化。不过它们是Unix 98的内容。

3.8 内核限制

System V IPC的多数实现有内在的内核限制，例如消息队列的最大数目、每个信号量集的最大信号量数，等等。我们将在图6-25、图11-8和图14-5中给出这些限制的某些典型值。这些限制通常起源于最初的System V实现。

[Bach 1986] 的11.2节和 [Goodheart and Cox 1994] 的第8章都讲述了消息队列、信号量和共享内存区的实现。某些限制就在那儿说明。

36

不幸的是，这些对象的大小被内核限制得往往太小，这是因为其中许多限制起源于在某个小地址空间系统（16位PDP-11）上完成的最初实现。然而万幸的是，多数系统允许管理员部分或完全修改这些默认限制，但是不同风格的Unix所需的步骤也不一样。多数系统要求在修改完值后重新自举运行中的内核。尽管如此，某些实现仍然给其中一些限制使用16位整数，这在无形之中提供了一个难以突破的硬限制。

举例来说，Solaris 2.6有20个这些限制。它们的当前值可使用sysdef命令输出，不过如果相应的内核模块尚未加载（也就是说尚未使用IPC机制），那么所输出的值为0。它们的值可通过在/etc/system文件中加入如下语句来修改，而/etc/system是自举内核时读入的。

```
set msgsys:msginfo_msgseg = value
set msgsys:msginfo_msgssz = value
set msgsys:msginfo_msgtql = value
set msgsys:msginfo_msgmap = value
set msgsys:msginfo_msgmax = value
set msgsys:msginfo_msgmnb = value
set msgsys:msginfo_msgmni = value

set semsys:seminfo_semopm = value
set semsys:seminfo_semume = value
set semsys:seminfo_semaem = value
set semsys:seminfo_semmap = value
set semsys:seminfo_semvmx = value
set semsys:seminfo_semmsl = value
set semsys:seminfo_semmni = value
set semsys:seminfo_semmns = value
set semsys:seminfo_semmnu = value

set shmsys:shminfo_shmmin = value
set shmsys:shminfo_shmseg = value
set shmsys:shminfo_shmmax = value
set shmsys:shminfo_shmmni = value
```

等号左边名字中最后6个字符就是列在图6-25、图11-8和图14-5中的变量名。

至于Digital Unix 4.0B，sysconfig程序可用于查询或修改许多内核参数和限制。下面是使用-q选项时该程序的输出，它就ipc子系统查询内核以输出当前限制值。我们已省略掉了与System V IPC机制无关的一些行。

```
alpha % /sbin/sysconfig -q ipc
ipc:
msg-max = 8192
msg-mnb = 16384
msg-mni = 64
msg-tql = 40

shm-max = 4194304
shm-min = 1
shm-mni = 128
shm-seg = 32

sem-mni = 16
sem-msl = 25
sem-opm = 10
sem-ume = 10
sem-vmx = 32767
sem-aem = 16384
num-of-sems = 60
```


这些参数的默认值可通过在/etc/sysconfigtab文件中指定不同的值来修改，不过该文件应使用sysconfigdb程序维护。该文件是在系统自举时读入的。

3.9　小结

msgget、semget和shmget这三个函数的第一个参数是一个System V IPC键。这些键通常是使用系统的ftok函数从某个路径名创建出的。键还可以是IPC_PRIVATE这个特殊值。这三个函数创建一个新的IPC对象或打开一个已存在的IPC对象，并返回一个System V IPC标识符：接下去用于给其余IPC函数标识该对象的一个整数。这些整数不是特定于进程的标识符（像描述符那样），而是系统范围的标识符。这些标识符还由内核在一段时间后重用。

与每个System V IPC对象相关联的是一个ipc_perm结构，它含有诸如属主的用户ID、组ID、读写权限等信息。Posix IPC和System V IPC的差别之一是，这些信息对于System V IPC对象总是可用的（通过以IPC_STAT命令参数调用三个ctl*XXX*函数中的某一个），但是对于Posix IPC对象来说，能否访问这些信息要看具体实现。如果Posix IPC对象存放在文件系统中，而且我们知道它们在文件系统中的名字，那么使用现有的文件系统工具就能访问到与ipc_perm结构的内容相同的信息。

在创建一个新的System V IPC对象或打开一个已存在的对象时，可给get*XXX*函数指定两个标志（IPC_CREAT和IPC_EXCL），外加9个权限位。

毫无疑问，使用System V IPC的最大问题在于多数实现在这些对象的大小上施加了人为的内核限制，这些限制可追溯到它们历史上的最初实现。这就是说，较多使用System V IPC的多数应用需要系统管理员修改这些内核限制，然而不同风格的Unix完成这些修改工作的步骤也不一样。

习题

3.1 粗读一下6.5节的msgctl函数，把图3-7中的程序修改成除输出所赋予的标识符外，还输出ipc_perm结构的seq成员。

3.2 运行图3-7中的程序两次后立即运行一个创建两个消息队列的程序。假设内核从自举以来没有任何其他应用程序使用过其他消息队列，那么由内核返回的作为消息队列标识符的两个值是什么？

3.3 我们在3.5节中指出System V IPC get*XXX*函数不使用文件模式创建掩码。编写一个测试程序，由它创建一个FIFO（使用4.6节中所述的mkfifo函数）和一个System V消息队列，给它们指定的权限都是666（八进制）。比较创建成的FIFO和消息队列二者的权限。在运行该程序前，确保你的shell umask为非零值。

3.4 如果一个服务器想要为其客户创建一个唯一的消息队列，那么采用哪种方法更恰当——使用某个常值路径名（譬如说该服务器的可执行文件）作为ftok的一个参数呢，还是使用IPC_PRIVATE？

3.5 修改图3-2使其只输出IPC键和路径名。运行find程序输出你的系统上的所有路径名，并将这些路径名逐个作为刚修改过的程序的命令行参数运行之。有多少路径名映射到同一键值上？

3.6 如果你的系统支持sar程序（“系统行为报告程序”），那么运行如下命令：

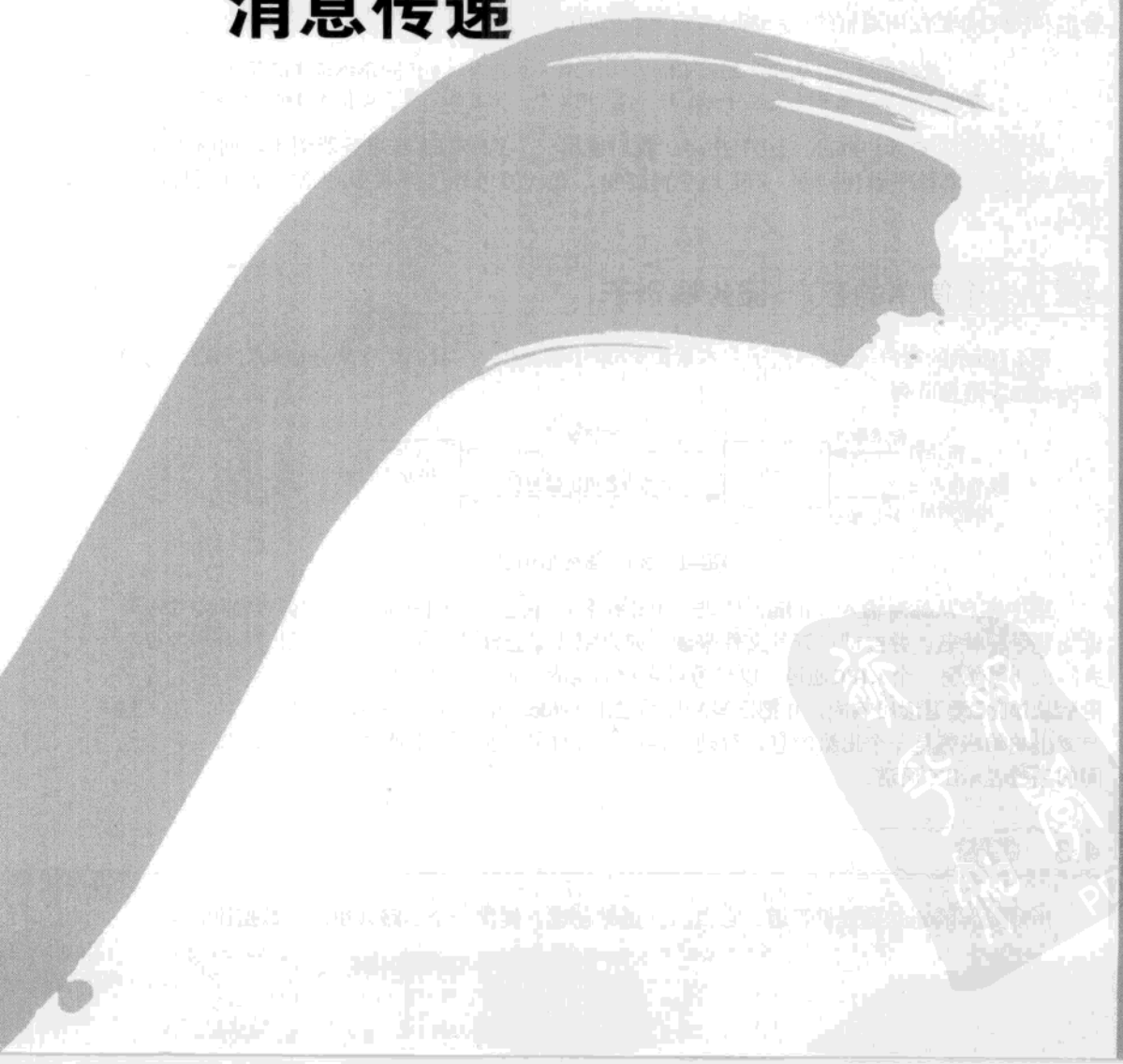
```
sar -m 5 6
```

该命令输出每秒钟的消息队列操作数和信号量操作数，采样频率为5秒钟一次，共6次。

38
39

第二部分

消息传递



第4章 管道和FIFO

4.1 概述

管道是最初的Unix IPC形式，可追溯到1973年的Unix第3版［Salus 1994］。尽管对于许多操作来说很有用，但它们的根本局限在于没有名字，从而只能由有亲缘关系的进程使用。这一点随FIFO的加入在System III Unix（1982年）中得以改正。FIFO有时称为有名管道（named pipe）。管道和FIFO都是使用通常的read和write函数访问的。

> 技术上讲，自从可以在进程间传递描述符（在本书15.8节和UNPv1的14.7节中讲述）后，管道也能用于无亲缘关系的进程间。然而现实中，管道通常用于具有共同祖先的进程间。

本章讲述管道和FIFO的创建与使用。我们使用一个简单的文件服务器例子，同时查看一些客户-服务器程序设计问题：IPC通道需要量、迭代服务器与并发服务器、字节流与消息接口。

4.2 一个简单的客户-服务器例子

图4-1所示的客户-服务器例子在本章和第6章中都要用到，我们用它来分析说明管道、FIFO和System V消息队列。

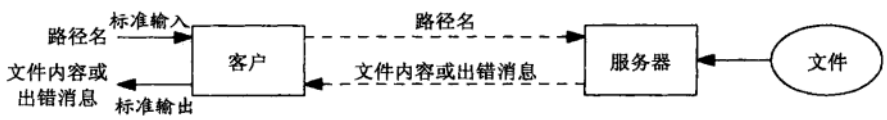


图4-1 客户-服务器例子

图中客户从标准输入（stdin）读进一个路径名，并把它写入IPC通道。服务器从该IPC通道读出这个路径名，并尝试打开其文件来读。如果服务器能打开该文件，它就读出其中的内容，并写入（可能另一个）IPC通道，以作为对客户的响应；否则，它就响应以一个出错消息。客户随后从该IPC通道读出响应，并把它写到标准输出（stdout）。如果服务器无法读该文件，那么客户读出的响应将是一个出错消息。否则，客户读出的响应就是该文件的内容。客户和服务器之间的虚线表示IPC通道。

4.3 管道

所有式样的Unix都提供管道。它由pipe函数创建，提供一个单路（单向）数据流。

```
#include <unistd.h>
int pipe(int fd[2]);
                                        返回：若成功则为0，若出错则为-1
```

该函数返回两个文件描述符：*fd[0]*和*fd[1]*。前者打开来读，后者打开来写。

> 有些版本的Unix（例如SVR4）提供全双工管道，也就是说这些管道的两端都是既可用于读，也可用于写。创建一个全双工IPC管道的另一种方法是使用UNPv1的14.3节中讲述的socketpair函数，它在大多数现行Unix系统上都能工作。然而管道的最常见用途是用在各种shell中，这种情况下半双工管道足够了。
>
> Posix.1和Unix 98只要求半双工管道，本章中我们也这么假设。

宏S_ISFIFO可用于确定一个描述符或文件是管道还是FIFO。它的唯一参数是stat结构的st_mode成员，计算结果或者为真（非零值），或者为假（0）。对于管道来说，这个stat结构是由fstat函数填写的。对于FIFO来说，这个结构是由fstat、lstat或stat函数填写的。

图4-2展示了单个进程中管道的模样。

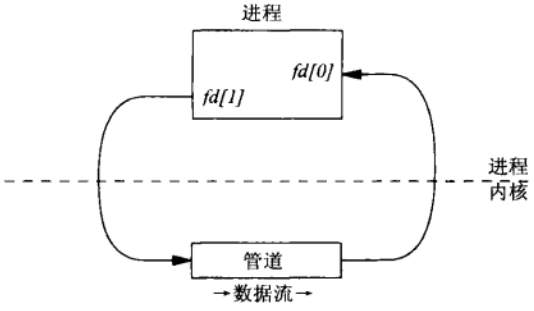


图4-2 单个进程中的管道

尽管管道是由单个进程创建的，它却很少在单个进程内使用（我们将在图5-14中给出在单个进程内使用管道的一个例子）。管道的典型用途是以下述方式为两个不同进程（一个是父进程，一个是子进程）提供进程间的通信手段。首先，由一个进程（它将成为父进程）创建一个管道后调用fork派生一个自身的副本，如图4-3所示。

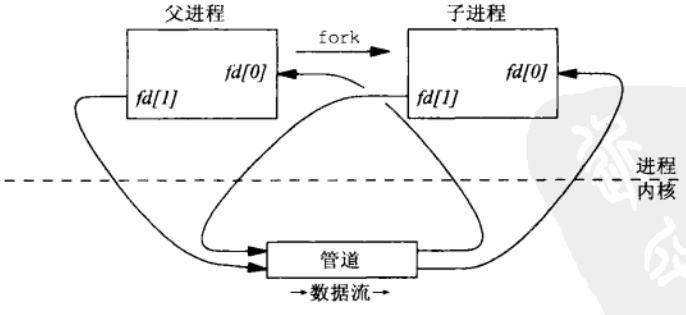


图4-3 单个进程内的管道，刚刚fork后

接着，父进程关闭这个管道的读出端，子进程关闭同一管道的写入端。这就在父子进程间提供了一个单向数据流，如图4-4所示。

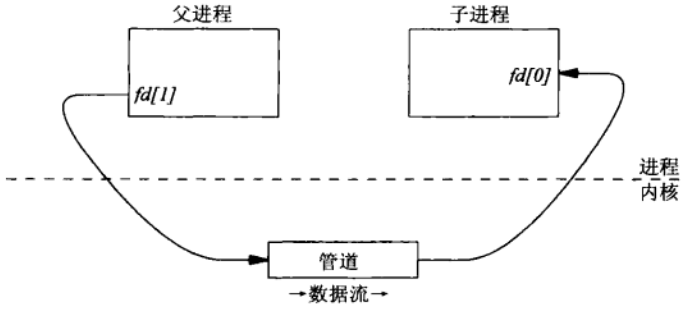


图4-4　两个进程间的管道

我们在某个Unix shell中输入一个像下面这样的命令时：

```
who | sort | lp
```

45 该shell将执行上述步骤创建三个进程和其间的两个管道。它还把每个管道的读出端复制到相应进程的标准输入，把每个管道的写入端复制到相应进程的标准输出。图4-5展示了这样的管道线。

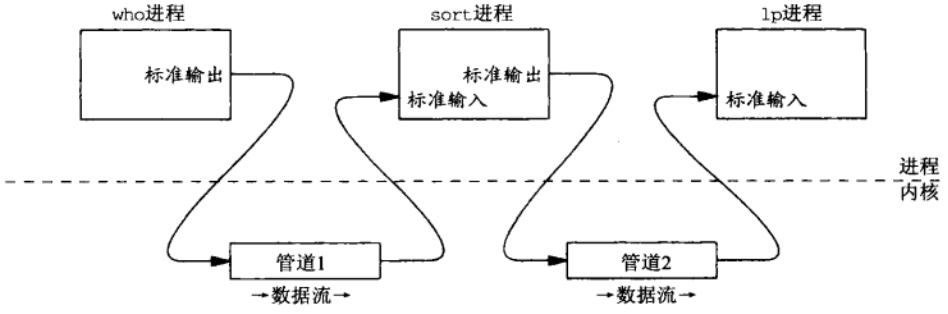


图4-5　某个shell管道线中三个进程间的管道

到此为止所示的所有管道都是半双工的即单向的，只提供一个方向的数据流。当需要一个双向数据流时，我们必须创建两个管道，每个方向一个。实际步骤如下：

(1) 创建管道1（*fd1[0]*和*fd1[1]*）和管道2（*fd2[0]*和*fd2[1]*）；

(2) fork；

(3) 父进程关闭管道1的读出端（*fd1[0]*）；

(4) 父进程关闭管道2的写入端（*fd2[1]*）；

(5) 子进程关闭管道1的写入端（*fd1[1]*）；

(6) 子进程关闭管道2的读出端（*fd2[0]*）。

46 图4-8给出了执行这些步骤的代码。它产生如图4-6所示的管道布局。

例子

现在使用管道实现4.2节中描述的客户-服务器例子。main函数创建两个管道并用fork生成一个子进程。客户然后作为父进程运行，服务器则作为子进程运行。第一个管道用于从客户向服务器发送路径名，第二个管道用于从服务器向客户发送该文件的内容（或者一个出错消息）。这样设置后的布局如图4-7所示。

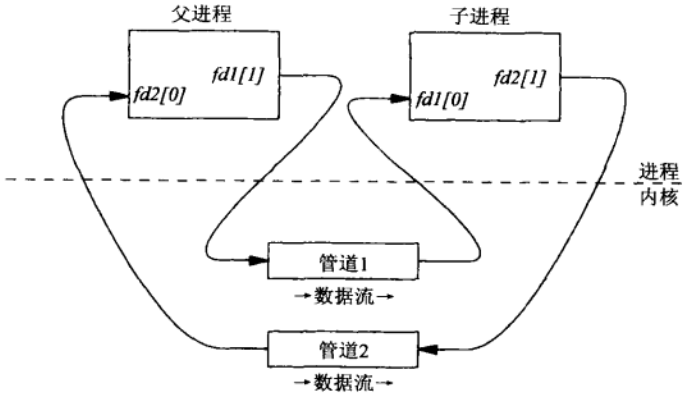


图4-6 提供一个双向数据流的两个管道

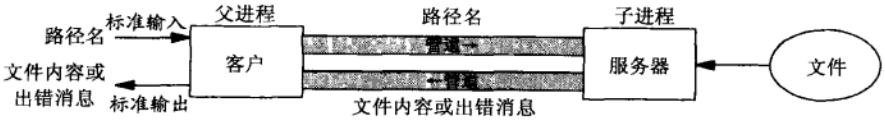


图4-7 使用两个管道实现图4-1

注意本图所示的两个管道直接连接着两个进程，然而实际上它们都是通过内核运作的，如图4-6所示。因此，从客户到服务器以及从服务器到客户的所有数据都穿越了用户-内核接口两次：一次是在写入管道时，一次是在从管道读出时。

图4-8给出了这个例子的main函数。

pipe/mainpipe.c

```
 1 #include    "unpipc.h"

 2 void    client(int, int), server(int, int);

 3 int
 4 main(int argc, char **argv)
 5 {
 6     int     pipe1[2], pipe2[2];
 7     pid_t   childpid;

 8     Pipe(pipe1);    /* create two pipes */
 9     Pipe(pipe2);

10     if ( (childpid = Fork()) == 0) {        /* child */
11         Close(pipe1[1]);
12         Close(pipe2[0]);

13         server(pipe1[0], pipe2[1]);
14         exit(0);
15     }
16         /* parent */
17     Close(pipe1[0]);
18     Close(pipe2[1]);

19     client(pipe2[0], pipe1[1]);

20     Waitpid(childpid, NULL, 0);        /* wait for child to terminate */
21     exit(0);
22 }
```

pipe/mainpipe.c

图4-8 使用两个管道的客户-服务器程序main函数

创建管道，fork

8~19　创建两个管道，然后执行随图4-6列出的6个步骤。父进程调用client函数（图4-9），子进程调用server函数（图4-10）。

为子进程waitpid

20　服务器（子进程）在往管道写入最终数据后调用exit首先终止。它随后变成了一个僵尸进程（zombie）：自身已终止、但其父进程仍在运行且尚未等待该子进程的进程。当子进程终止时，内核还给其父进程产生一个SIGCHLD信号，不过父进程没有捕获这个信号，而该信号的默认行为就是忽略。此后不久，父进程的client函数在从管道读入最终数据后返回。父进程随后调用waitpid取得已终止子进程（僵尸进程）的终止状态。要是父进程没有调用waitpid，而是直接终止，那么子进程将成为托孤给init进程的孤儿进程，内核将为此向init进程发送另外一个SIGCHLD信号，init进程随后将取得该僵尸进程的终止状态。

client函数如图4-9所示。

pipe/client.c

```
 1 #include    "unpipc.h"

 2 void
 3 client(int readfd, int writefd)
 4 {
 5     size_t  len;
 6     ssize_t n;
 7     char    buff[MAXLINE];

 8         /* read pathname */
 9     Fgets(buff, MAXLINE, stdin);
10     len = strlen(buff);         /* fgets() guarantees null byte at end */
11     if (buff[len-1] == '\n')
12         len--;                  /* delete newline from fgets() */

13         /* write pathname to IPC channel */
14     Write(writefd, buff, len);

15         /* read from IPC, write to standard output */
16     while ( (n = Read(readfd, buff, MAXLINE)) > 0)
17         Write(STDOUT_FILENO, buff, n);
18 }
```

pipe/client.c

图4-9　使用两个管道的客户-服务器程序client函数

从标准输入读进路径名

8~14　从标准输入读进路径名后，删除其中由fgets存入的换行符，再写入管道。

从管道复制到标准输出

48

15~17　客户随后读出由服务器写入管道的全部内容，并写到标准输出。正常情况下它是所请求文件的内容，但是如果服务器打不开所指定的路径名，那它将返回一个出错消息。

server函数如图4-10所示。

从管道读出路径名

8~11　从管道读出由客户写入的路径名，并以空字节作为其结尾。注意，对一个管道的read只要该管道中存在一些数据就会马上返回，它不必等待达到所请求的字节数（本例中为MAXLINE）。

pipe/server.c

```
1 #include    "unpipc.h"
2 void
3 server(int readfd, int writefd)
4 {
5     int     fd;
6     ssize_t n;
7     char    buff[MAXLINE+1];
8         /* read pathname from IPC channel */
9     if ( (n = Read(readfd, buff, MAXLINE)) == 0)
10        err_quit("end-of-file while reading pathname");
11    buff[n] = '\0';             /* null terminate pathname */
12    if ( (fd = open(buff, O_RDONLY)) < 0) {
13            /* error: must tell client */
14        snprintf(buff + n, sizeof(buff) - n, ": can't open, %s\n",
15                 strerror(errno));
16        n = strlen(buff);
17        Write(writefd, buff, n);
18    } else {
19            /* open succeeded: copy file to IPC channel */
20        while ( (n = Read(fd, buff, MAXLINE)) > 0)
21            Write(writefd, buff, n);
22        Close(fd);
23    }
24 }
```

pipe/server.c

图4-10 使用两个管道的客户-服务器程序server函数

打开文件，处理错误

12~17 打开所请求的文件来读，若出错则通过管道返回给客户一个出错消息串。我们调用strerror函数以返回对应于errno的出错消息串。（UNPv1第690~691页详细讨论了strerror函数[①]。）

把文件复制到管道

18~23 如果open成功，就将该文件的内容复制到管道中。

下面的例子给出了路径名正确及发生错误时该程序的输出。

```
solaris % mainpipe
/etc/inet/ntp.conf                    一个由两行文本构成的文件
multicastclient 224.0.1.1
driftfile /etc/inet/ntp.drift
solaris % mainpipe
/etc/shadow                           一个我们不能读的文件
/etc/shadow: can't open, Permission denied
solaris % mainpipe
/no/such/file                         一个不存在的文件
/no/such/file: can't open, No such file or directory
```

4.4 全双工管道

上一节中我们提到，某些系统提供全双工管道：SVR4的pipe函数以及许多内核都提供的

① 此处为UNPv1第2版英文原版书页码，第3版英文原版书为第774~775页。——编者注

socketpair函数。那么全双工管道到底提供什么呢？首先，我们可以如图4-11所示考虑一个半双工管道，它是对图4-2的修改，省略了其中的进程。



图4-11　半双工管道

全双工管道可能实现成如图4-12所示。它隐含的意思是：整个管道只存在一个缓冲区，（在任意一个描述符上）写入管道的任何数据都添加到该缓冲区末尾，（在任意一个描述符上）从管道读出的都是取自该缓冲区开头的数据。

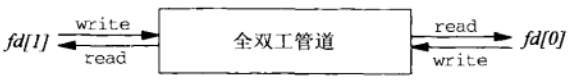


图4-12　全双工管道的一个可能实现（不正确）

这种实现所存在的问题在像图A-29这样的程序中变得很明显。我们需要双向通信，但所需的是两个独立的数据流，每个方向一个。若不是这样，当一个进程往该全双工管道写入数据，过后再对该管道调用read时，有可能读回刚写入的数据。

图4-13展示了全双工管道的真正实现。

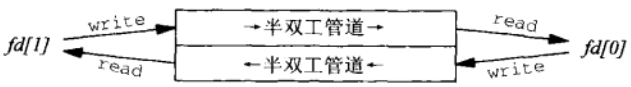


图4-13　全双工管道的真正实现

50　这儿的全双工管道是由两个半双工管道构成的。写入*fd[1]*的数据只能从*fd[0]*读出，写入*fd[0]*的数据只能从*fd[1]*读出。

图4-14中的程序表明可使用单个全双工管道完成双向通信。

pipe/fduplex.c

```
 1 #include    "unpipc.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     fd[2], n;
 6     char    c;
 7     pid_t   childpid;

 8     Pipe(fd);                   /* assumes a full-duplex pipe (e.g., SVR4) */
 9     if ( (childpid = Fork()) == 0) {        /* child */
10         sleep(3);
11         if ( (n = Read(fd[0], &c, 1)) != 1)
12             err_quit("child: read returned %d", n);
13         printf("child read %c\n", c);
14         Write(fd[0], "c", 1);
15         exit(0);
16     }
17         /* parent */
18     Write(fd[1], "p", 1);
19     if ( (n = Read(fd[1], &c, 1)) != 1)
20         err_quit("parent: read returned %d", n);
21     printf("parent read %c\n", c);
22     exit(0);
23 }
```

pipe/fduplex.c

图4-14　测试全双工管道的双向通信能力

我们创建一个全双工管道后调用fork。父进程往该管道写入字符p，然后从中读出一个字符。子进程先睡眠3秒，从该管道读出一个字符后，往它写入字符c。子进程中的睡眠是为了让父进程的read调用先于子进程的read调用执行，从而查看父进程是否读回自己刚写的字符。

在提供全双工管道的Solaris 2.6下运行该程序，我们观察到了所期望的行为。

```
solaris % fduplex
child read p
parent read c
```

字符p穿越图4-13中所示顶部的半双工管道，字符c则穿越底部的半双工管道。父进程并没有读回自己写入管道的数据（字符p）。

在默认提供半双工管道的Digital Unix 4.0B下运行该程序，我们看到了半双工管道的预期行为。编译时如果指定不同的选项，那它也能像SVR4那样提供全双工管道。 51

```
alpha % fduplex
read error: Bad file number
alpha % child read p
write error: Bad file number
```

父进程写入字符p，它由子进程读出，但此后父进程在试图从*fd[1]* read时中止，子进程则在试图往*fd[0]* write时中止（回想图4-11）。由read返回的错误是EBADF，意思是描述符*fd[1]*不是打开来读的。类似地，write返回同样的错误，表示描述符*fd[1]*不是打开来写的。

4.5 popen 和 pclose 函数

作为另一个关于管道的例子，标准I/O函数库提供了popen函数，它创建一个管道并启动另外一个进程，该进程要么从该管道读出标准输入，要么往该管道写入标准输出。

```
#include <stdio.h>

FILE *popen(const char *command, const char *type);
                                        返回：若成功则为文件指针，若出错则为NULL

int pclose(FILE *stream);
                                        返回：若成功则为shell的终止状态，若出错则为-1
```

其中*command*是一个shell命令行。它是由sh程序（通常为Bourne shell）处理的，因此PATH环境变量可用于定位*command*。popen在调用进程和所指定的命令之间创建一个管道。由popen返回的值是一个标准I/O FILE指针，该指针或者用于输入，或者用于输出，具体取决于字符串*type*。

- 如果*type*为r，那么调用进程读进*command*的标准输出。
- 如果*type*为w，那么调用进程写到*command*的标准输入。

pclose函数关闭由popen创建的标准I/O流（*stream*），等待其中的命令终止，然后返回shell的终止状态。

APUE的14.3节[①]提供了popen和pclose的一个实现。

例子

图4-15给出了我们的客户-服务器例子的另一个实现，它使用popen函数和Unix的cat程序。 52

① 此处为APUE第1版英文原版书节号，第2版为15.3节。——编者注

pipe/mainpopen.c

```
 1 #include    "unpipc.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     size_t  n;
 6     char    buff[MAXLINE], command[MAXLINE];
 7     FILE    *fp;
 8         /* read pathname */
 9     Fgets(buff, MAXLINE, stdin);
10     n = strlen(buff);         /* fgets() guarantees null byte at end */
11     if (buff[n-1] == '\n')
12         n--;                  /* delete newline from fgets() */
13     snprintf(command, sizeof(command), "cat %s", buff);
14     fp = Popen(command, "r");
15         /* copy from pipe to standard output */
16     while (Fgets(buff, MAXLINE, fp) != NULL)
17         Fputs(buff, stdout);
18     Pclose(fp);
19     exit(0);
20 }
```

pipe/mainpopen.c

图4-15　使用popen的客户-服务器程序

8~17　跟图4-9一样，路径名从标准输入读出。随后构建一个命令并把它传递给popen。来自shell或cat程序的输出被复制到标准输出。

这个实现与图4-8中的实现的差别之一是：现在我们依赖于由系统的cat程序产生的出错消息，而这些消息往往不足以说明具体错误。例如在Solaris 2.6下，当试图读一个我们没有读权限的文件时，将得到如下的错误：

```
solaris % cat /etc/shadow
cat: cannot open /etc/shadow
```

但是在BSD/OS 3.1下，当试图读一下类似的文件时，将得到一个更清晰的错误消息：

```
bsdi % cat /etc/master.passwd
cat: /etc/master.passwd: cannot open [Permission denied]
```

还要认识到在上面的例子中，popen调用是成功的，但是其后的fgets只是在首次被调用时返回一个文件结束符。cat程序将它的出错消息写到标准错误输出，但popen不对标准错误输出作任何特殊的处理——只有标准输出才被重定向到由它创建的管道。

53

4.6　FIFO

管道没有名字，因此它们的最大劣势是只能用于有一个共同祖先进程的各个进程之间。我们无法在无亲缘关系的两个进程间创建一个管道并将它用作IPC通道（不考虑描述符传递）。

FIFO指代先进先出（first in，first out），Unix中的FIFO类似于管道。它是一个单向（半双工）数据流。不同于管道的是，每个FIFO有一个路径名与之关联，从而允许无亲缘关系的进程访问同一个FIFO。FIFO也称为有名管道（named pipe）。

FIFO由mkfifo函数创建。

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

int mkfifo(const char *pathname, mode_t mode);
```

返回：若成功则为0，若出错则为-1

其中*pathname*是一个普通的Unix路径名，它是该FIFO的名字。

*mode*参数指定文件权限位，类似于open的第二个参数。图2-4给出了定义在<sys/stat.h>头文件中的6个常值，用于给一个FIFO指定权限位。

mkfifo函数已隐含指定O_CREAT | O_EXCL。也就是说，它要么创建一个新的FIFO，要么返回一个EEXIST错误（如果所指定名字的FIFO已经存在）。如果不希望创建一个新的FIFO，那就改为调用open而不是mkfifo。要打开一个已存在的FIFO或创建一个新的FIFO，应先调用mkfifo，再检查它是否返回EEXIST错误，若返回该错误则改为调用open。

mkfifo命令也能创建FIFO。可以从shell脚本或命令行中使用它。

在创建出一个FIFO后，它必须或者打开来读，或者打开来写，所用的可以是open函数，也可以是某个标准I/O打开函数，例如fopen。FIFO不能打开来既读又写，因为它是半双工的。

对管道或FIFO的write总是往末尾添加数据，对它们的read则总是从开头返回数据。如果对管道或FIFO调用lseek，那就返回ESPIPE错误。

4.6.1 例子

现在重新编写图4-8中的客户-服务器程序，这次改用两个FIFO代替两个管道。client和server函数保持不变，所有变动都在main函数上，它如图4-16所示。

创建两个FIFO

10~16 在/tmp文件系统中创建两个FIFO。这两个FIFO事先存在与否无关紧要。常值FILE_MODE在我们的unpipc.h头文件（图C-1）中定义如下。

```
#define  FILE_MODE  (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
                    /* default permissions for new files */
```

这允许用户读、用户写、组成员读和其他用户读。这些权限位会被当前进程的文件模式创建掩码修正。

fork

17~27 调用fork后，子进程调用我们的server函数（图4-10），父进程调用我们的client函数（图4-9）。在执行这些调用前，父进程打开第一个FIFO来写，打开第二个FIFO来读，子进程打开第一个FIFO来读，打开第二个FIFO来写。这与我们的管道例子类似，图4-17展示了这个布局。

这个FIFO例子比起之前的管道例子变动如下。

- 创建并打开一个管道只需调用pipe。创建并打开一个FIFO则需在调用mkfifo后再调用open。
- 管道在所有进程最终都关闭它之后自动消失。FIFO的名字则只有通过调用unlink才从文件系统中删除。

FIFO需要额外调用的好处是：FIFO在文件系统中有一个名字，该名字允许某个进程创建一个FIFO，与它无亲缘关系的另一个进程来打开这个FIFO。对于管道来说，这是不可能的。

pipe/mainfifo.c

```
 1 #include    "unpipc.h"
 2 #define FIFO1   "/tmp/fifo.1"
 3 #define FIFO2   "/tmp/fifo.2"
 4 void    client(int, int), server(int, int);
 5 int
 6 main(int argc, char **argv)
 7 {
 8     int     readfd, writefd;
 9     pid_t   childpid;
10         /* create two FIFOs; OK if they already exist */
11     if ((mkfifo(FIFO1, FILE_MODE) < 0) && (errno != EEXIST))
12         err_sys("can't create %s", FIFO1);
13     if ((mkfifo(FIFO2, FILE_MODE) < 0) && (errno != EEXIST)) {
14         unlink(FIFO1);
15         err_sys("can't create %s", FIFO2);
16     }
17     if ( (childpid = Fork()) == 0) {        /* child */
18         readfd = Open(FIFO1, O_RDONLY, 0);
19         writefd = Open(FIFO2, O_WRONLY, 0);
20         server(readfd, writefd);
21         exit(0);
22     }
23         /* parent */
24     writefd = Open(FIFO1, O_WRONLY, 0);
25     readfd = Open(FIFO2, O_RDONLY, 0);
26     client(readfd, writefd);
27     Waitpid(childpid, NULL, 0);    /* wait for child to terminate */
28     Close(readfd);
29     Close(writefd);
30     Unlink(FIFO1);
31     Unlink(FIFO2);
32     exit(0);
33 }
```

pipe/mainfifo.c

图4-16　使用两个FIFO的客户-服务器程序main函数

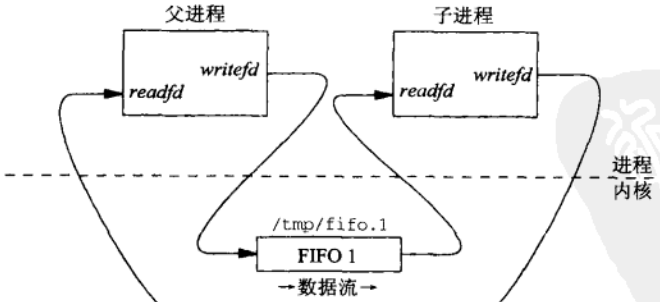


图4-17　使用两个FIFO的客户-服务器例子

没有正确使用FIFO的程序会发生微妙的问题。考虑图4-16：如果我们对换父进程中两个open调用的顺序，该程序就不工作。其原因在于，如果当前尚没有任何进程打开某个FIFO来写，那么打开该FIFO来读的进程将阻塞。对换父进程中两个open调用的顺序后，父子进程将都打开同一个FIFO来读，然而当时并没有任何进程已打开该文件来写，于是父子进程都阻塞。这种现象称为死锁（deadlock）。我们将在下一节讨论这种情形。

4.6.2 例子：无亲缘关系的客户与服务器

图4-16中客户和服务器仍然是有亲缘关系的进程。不过我们可以把这个例子重新编写成客户与服务器无亲缘关系的状态。图4-18给出了服务器程序，它与图4-16的服务器部分差不多一样。

pipe/server_main.c

```
1 #include    "fifo.h"
2 void    server(int, int);
3 int
4 main(int argc, char **argv)
5 {
6     int     readfd, writefd;
7         /* create two FIFOs; OK if they already exist */
8     if ((mkfifo(FIFO1, FILE_MODE) < 0) && (errno != EEXIST))
9         err_sys("can't create %s", FIFO1);
10    if ((mkfifo(FIFO2, FILE_MODE) < 0) && (errno != EEXIST)) {
11        unlink(FIFO1);
12        err_sys("can't create %s", FIFO2);
13    }
14    readfd = Open(FIFO1, O_RDONLY, 0);
15    writefd = Open(FIFO2, O_WRONLY, 0);
16    server(readfd, writefd);
17    exit(0);
18 }
```

pipe/server_main.c

图4-18 独立服务器程序main函数

头文件fifo.h如图4-19所示，它提供了程序中两个FIFO名字的定义，客户和服务器都得知道它们。

pipe/fifo.h

```
1 #include    "unpipc.h"
2 #define   FIFO1   "/tmp/fifo.1"
3 #define   FIFO2   "/tmp/fifo.2"
```

pipe/fifo.h

图4-19 客户程序和服务器程序都包含的fifo.h头文件

图4-20给出了客户程序，它与图4-16的客户部分差不多一样。注意最后删除所用FIFO的是客户而不是服务器，因为对这些FIFO执行最终操作的是客户。

> 内核为管道和FIFO维护一个访问计数器，它的值是访问同一个管道或FIFO的打开着的描述符的个数。有了访问计数器后，客户或服务器就能成功地调用unlink。尽管该函数从文件系统中删除了所指定的路径名，先前已经打开该路径名、目前仍打开着的描述符却不受影响。然而对于其他形式的IPC来说（例如System V消息队列），这样的计数器并不存在，因此要是服务器在向某个消息队列写入自己的最终消息后删除了该队列，那么当客户尝试读出这个最终消息时，该队列可能已消失了。

pipe/client_main.c

```
1 #include    "fifo.h"
2 void    client(int, int);
3 int
4 main(int argc, char **argv)
5 {
6     int     readfd, writefd;
7     writefd = Open(FIFO1, O_WRONLY, 0);
8     readfd = Open(FIFO2, O_RDONLY, 0);
9     client(readfd, writefd);
10    Close(readfd);
11    Close(writefd);
12    Unlink(FIFO1);
13    Unlink(FIFO2);
14    exit(0);
15 }
```

pipe/client_main.c

图4-20　独立客户程序main函数

为运行这对客户和服务器，先在后台启动服务器：

```
% server_file &
```

再启动客户。另一种办法是只启动客户，服务器则由客户通过调用fork和exec来激活。客户还可以把所用的两个FIFO的名字作为命令行参数通过exec函数传递给服务器，从而不必将它们编写到一个头文件中。不过这种情形会使得服务器成为客户的一个子进程，这么一来管道就够用了。

4.7　管道和FIFO的额外属性

我们需要就管道和FIFO的打开、读出和写入更为详细地描述它们的某些属性。首先，一个描述符能以两种方式设置成非阻塞。

(1) 调用open时可指定O_NONBLOCK标志。例如图4-20中第一个open调用可以是：

```
writefd = Open(FIFO1, O_WRONLY | O_NONBLOCK, 0);
```

(2) 如果一个描述符已经打开，那么可以调用fcntl以启用O_NONBLOCK标志。对于管道来说，必须使用这种技术，因为管道没有open调用，在pipe调用中也无法指定O_NONBLOCK标志。使用fcntl时，我们先使用F_GETFL命令取得当前文件状态标志，将它与O_NONBLOCK标志按位或后，再使用F_SETFL命令存储这些文件状态标志：

```
int    flags;
if ( (flags = fcntl(fd, F_GETFL, 0)) < 0)
    err_sys("F_GETFL error");
flags |= O_NONBLOCK;
if (fcntl(fd, F_SETFL, flags) < 0)
    err_sys("F_SETFL error");
```

留心你可能会碰到的简单地设置所需标志的代码，因为这样的代码在设置所需标志的同时清除了所有其他可能存在的文件状态标志：

```
    /* wrong way to set nonblocking */
```

```
if (fcntl(fd, F_SETFL, O_NONBLOCK) < 0)
    err_sys("F_SETFL error");
```

54 ~ 58

图4-21给出了非阻塞标志对打开一个FIFO的影响、对从一个空管道或空FIFO读出数据的影响以及对往一个管道或FIFO写入数据的影响。

当前操作	管道或FIFO的现有打开操作	返回	
		阻塞（默认设置）	O_NONBLOCK设置
open FIFO只读	FIFO打开来写	成功返回	成功返回
	FIFO不是打开来写	阻塞到FIFO打开来写为止	成功返回
open FIFO只写	FIFO打开来读	成功返回	成功返回
	FIFO不是打开来读	阻塞到FIFO打开来读为止	返回ENXIO错误
从空管道或空FIFO read	管道或FIFO打开来写	阻塞到管道或FIFO中有数据或者管道或FIFO不再为写打开着为止	返回EAGAIN错误
	管道或FIFO不是打开来写	read返回0（文件结束符）	read返回0（文件结束符）
往管道或FIFO write	管道或FIFO打开来读	（见正文）	（见正文）
	管道或FIFO不是打开来读	给线程产生SIGPIPE	给线程产生SIGPIPE

图4-21 O_NONBLOCK标志对管道和FIFO的影响

下面是关于管道或FIFO的读出与写入的若干额外规则。

- 如果请求读出的数据量多于管道或FIFO中当前可用数据量，那么只返回这些可用的数据。我们必须准备好处理来自read的小于所请求数目的返回值。
- 如果请求写入的数据的字节数小于或等于PIPE_BUF（一个Posix限制值，将在4.11节详细讨论），那么write操作保证是原子的。这意味着，如果有两个进程差不多同时往同一个管道或FIFO写，那么或者先写入来自第一个进程的所有数据，再写入来自第二个进程的所有数据，或者颠倒过来。系统不会相互混杂来自这两个进程的数据。然而，如果请求写入的数据的字节数大于PIPE_BUF，那么write操作不能保证是原子的。

  Posix.1要求PIPE_BUF至少为512字节。常见的值处于从BSD/0S 3.1的1024到Solaris 2.6的5120之间。4.11节有一个程序可输出这个值。

- O_NONBLOCK标志的设置对write操作的原子性没有影响——原子性完全是由所请求字节数是否小于等于PIPE_BUF决定的。然而当一个管道或FIFO设置成非阻塞时，来自write的返回值取决于待写的字节数以及该管道或FIFO中当前可用空间的大小。如果待写的字节数小于等于PIPE_BUF：

  a. 如果该管道或FIFO中有足以存放所请求字节数的空间，那么所有数据字节都写入。

  b. 如果该管道或FIFO中没有足以存放所请求字节数的空间，那么立即返回一个EAGAIN错误。既然设置了O_NONBLOCK标志，调用进程就不希望自己被投入睡眠中。但是内核无法在接受部分数据的同时仍保证write操作的原子性，于是它必须返回一个错误，告诉调用进程以后再试。

  如果待写的字节数大于PIPE_BUF：

  a. 如果该管道或FIFO中至少有1字节空间，那么内核写入该管道或FIFO能容纳数目的数据字节，该数目同时作为来自write的返回值。

  b. 如果该管道或FIFO已满，那么立即返回一个EAGAIN错误。

- 如果向一个没有为读打开着的管道或FIFO写入，那么内核将产生一个SIGPIPE信号：

59

a. 如果调用进程既没有捕获也没有忽略该SIGPIPE信号，所采取的默认行为就是终止该进程。
b. 如果调用进程忽略了该SIGPIPE信号，或者捕获了该信号并从其信号处理程序中返回，那么write返回一个EPIPE错误。

SIGPIPE被认为是一个同步信号，也就是说，这是一个由特定的线程（调用write的线程）引起的信号。然而处理这个信号最容易的办法是忽略它（把它的处理办法设置成SIG_IGN），让write返回一个EPIPE错误。应用程序应该无遗漏地检测由write返回的错误，而检测某个进程被SIGPIPE终止却困难得多。如果该信号未被捕获，我们就得从shell中查看被终止进程的终止状态，以确定该进程是否被某个信号所杀死以及具体是被哪个信号杀死的。UNPv1的5.13节详细讨论SIGPIPE。

4.8　单个服务器，多个客户

FIFO的真正优势表现在服务器可以是一个长期运行的进程（例如守护进程，如UNPv1第12章所述），而且与其客户可以无亲缘关系。作为服务器的守护进程以某个众所周知的路径名创建一个FIFO，并打开该FIFO来读。此后某个时刻启动的客户打开该FIFO来写，并将其命令或给守护进程的其他任何东西通过该FIFO发送出去。使用FIFO很容易实现这种形式的单向通信（从客户到服务器），但是如果守护进程需要向客户发送回一些东西，事情就困难了。图4-22是我们随例子使用的技巧。

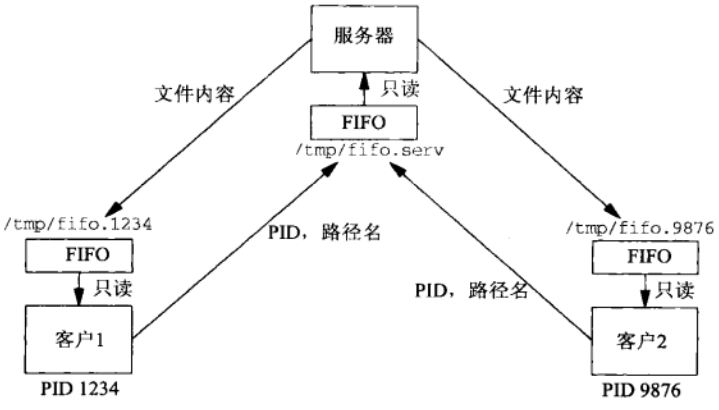


图4-22　单个服务器，多个客户

服务器以一个众所周知的路径名（本例中为/tmp/fifo.serv）创建一个FIFO，它将从这个FIFO读入客户的请求。每个客户在启动时创建自己的FIFO，所用的路径名含有自己的进程ID。每个客户把自己的请求写入服务器的众所周知FIFO中，该请求含有客户的进程ID以及一个路径名，具有该路径名的文件就是客户希望服务器打开并发回的文件。

图4-23给出了服务器程序。

创建众所周知FIFO，打开来读，打开来写

10~15　创建服务器的众所周知FIFO，就算它已经存在也没问题。接着打开该管道两次，一次只读，一次只写。readfifo描述符用于读出到达该FIFO的每个客户请求，dummyfd描述符则从来不用。打开该FIFO来写的原因可从图4-21中看出。要是我们不这么做，那么每当有一个客户终止时，该FIFO就变空，于是服务器的read返回0，表示是一个文

件结束符。我们将不得不close该FIFO，并以O_RDONLY标志再次调用open，不过该调用会一直阻塞到下一个客户请求到达为止。然而如果我们总是有一个该FIFO的描述符打开着用于写，那么当不再有客户存在时，服务器的read一定不会返回0以指示读到一个文件结束符。相反，服务器只是阻塞在read调用中，等待下一个客户请求。于是这个技巧简化了我们的服务器代码，减少了为其众所周知FIFO调用open的次数。

fifocliserv/mainserver.c

```
1 #include    "fifo.h"
2 void    server(int, int);
3 int
4 main(int argc, char **argv)
5 {
6     int      readfifo, writefifo, dummyfd, fd;
7     char    *ptr, buff[MAXLINE+1], fifoname[MAXLINE];
8     pid_t    pid;
9     ssize_t n;
10        /* create server's well-known FIFO; OK if already exists */
11    if ((mkfifo(SERV_FIFO, FILE_MODE) < 0) && (errno != EEXIST))
12        err_sys("can't create %s", SERV_FIFO);
13        /* open server's well-known FIFO for reading and writing */
14    readfifo = Open(SERV_FIFO, O_RDONLY, 0);
15    dummyfd = Open(SERV_FIFO, O_WRONLY, 0);     /* never used */
16    while ( (n = Readline(readfifo, buff, MAXLINE)) > 0) {
17        if (buff[n-1] == '\n')
18            n--;                /* delete newline from readline() */
19        buff[n] = '\0';         /* null terminate pathname */
20        if ( (ptr = strchr(buff, ' ')) == NULL) {
21            err_msg("bogus request: %s", buff);
22            continue;
23        }
24        *ptr++ = 0;                 /* null terminate PID, ptr = pathname */
25        pid = atol(buff);
26        snprintf(fifoname, sizeof(fifoname), "/tmp/fifo.%ld", (long) pid);
27        if ( (writefifo = open(fifoname, O_WRONLY, 0)) < 0) {
28            err_msg("cannot open: %s", fifoname);
29            continue;
30        }
31        if ( (fd = open(ptr, O_RDONLY)) < 0) {
32                /* error: must tell client */
33            snprintf(buff + n, sizeof(buff) - n, ": can't open, %s\n",
34                     strerror(errno));
35            n = strlen(ptr);
36            Write(writefifo, ptr, n);
37            Close(writefifo);
38        } else {
39                /* open succeeded: copy file to FIFO */
40            while ( (n = Read(fd, buff, MAXLINE)) > 0)
41                Write(writefifo, buff, n);
42            Close(fd);
43            Close(writefifo);
44        }
45    }
46    exit(0);
47 }
```

fifocliserv/mainserver.c

图4-23 处理多个客户请求的FIFO服务器程序

服务器启动时，它的第一个open（使用O_RDONLY标志）将阻塞到第一个客户只写打开服务器的FIFO为止（回想图4-21）。它的第二个open（使用O_WRONLY标志）则立即返回，因为该FIFO已经打开着用于读了。

读出客户请求

60～62

16　每个客户请求是由进程ID、一个空格再加路径名构成的单行。我们使用自己的readline函数（见UNPv1第79页）。

分析客户请求

17～26　删除通常由readline返回的换行符。该换行符只有在这样两种情况下才会拉掉：遇到它之前缓冲区已填满，或者输入的最后一行不是以换行符终止。由strchr函数返回的赋给ptr的指针指向客户请求行中的空格，ptr增1后即指向后跟的路径名的首字符。客户的FIFO的路径名是根据其进程ID构造的，服务器以只写的方式打开该FIFO。

打开客户请求的文件，将它发送到客户的FIFO

27～44　服务器的剩余部分类似于图4-10中的server函数。它打开客户请求的文件，若失败则通过客户的FIFO向客户返回一个出错消息。若open成功则把文件的内容复制到客户的FIFO中。完成后close客户的FIFO的服务器端，以使得客户的read返回0（文件结束符）。服务器不删除客户的FIFO，这个工作由客户在读出来自服务器的文件结束符后完成。

客户程序在图4-24中给出。

fifocliserv/mainclient.c

```
 1 #include    "fifo.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     readfifo, writefifo;
 6     size_t  len;
 7     ssize_t n;
 8     char    *ptr, fifoname[MAXLINE], buff[MAXLINE];
 9     pid_t   pid;
10         /* create FIFO with our PID as part of name */
11     pid = getpid();
12     snprintf(fifoname, sizeof(fifoname), "/tmp/fifo.%ld", (long) pid);
13     if ((mkfifo(fifoname, FILE_MODE) < 0) && (errno != EEXIST))
14         err_sys("can't create %s", fifoname);
15         /* start buffer with pid and a blank */
16     snprintf(buff, sizeof(buff), "%ld ", (long) pid);
17     len = strlen(buff);
18     ptr = buff + len;
19         /* read pathname */
20     Fgets(ptr, MAXLINE - len, stdin);
21     len = strlen(buff);             /* fgets() guarantees null byte at end */
22         /* open FIFO to server and write PID and pathname to FIFO */
23     writefifo = Open(SERV_FIFO, O_WRONLY, 0);
24     Write(writefifo, buff, len);
25         /* now open our FIFO; blocks until server opens for writing */
26     readfifo = Open(fifoname, O_RDONLY, 0);
27         /* read from IPC, write to standard output */
28     while ( (n = Read(readfifo, buff, MAXLINE)) > 0)
29         Write(STDOUT_FILENO, buff, n);
30     Close(readfifo);
31     Unlink(fifoname);
32     exit(0);
33 }
```

fifocliserv/mainclient.c

图4-24　与图4-23中服务器程序协同工作的客户程序

创建FIFO

10~14 客户的FIFO是使用以进程ID作为其最后一部分的路径名创建的。

构建客户请求行

15~21 客户的请求由其进程ID、一个空格、一个路径名和一个换行符构成，其中的路径名指代请求服务器发送给本客户的文件。这个请求行在字符数组buff中构建，路径名则从标准输入读入。

打开服务器的FIFO，写入请求

22~24 打开服务器的FIFO后往其中写入请求。如果本客户是自服务器启动以来第一个打开该FIFO的客户，那么这儿的open将把服务器从它的open调用（使用O_RDONLY标志）中解阻塞出来。

读出来自服务器的文件内容或出错消息

25~31 从本客户的FIFO中读出服务器的应答，并写到标准输出。随后关闭并删除该FIFO。

我们可以在一个窗口中启动服务器，在另一个窗口中运行客户，它们将如期地工作。下面给出的只是客户的交互例子。

```
solaris % mainclient
/etc/shadow                                    一个我们不能读的文件
/etc/shadow: can't open, Permission denied
solaris % mainclient
/etc/inet/ntp.conf                             一个由2行文本构成的文件
multicastclient 224.0.1.1
driftfile /etc/inet/ntp.drift
```

我们还可以从shell中与服务器交互，因为FIFO在文件系统中有名字。

```
solaris % Pid=$$                               本shell的进程ID
solaris % mkfifo /tmp/fifo.$Pid                创建客户的FIFO
solaris % echo "$Pid /etc/inet/ntp.conf" > /tmp/fifo.serv
solaris % cat < /tmp/fifo.$Pid                 读出服务器的应答
multicastclient 224.0.1.1
driftfile /etc/inet/ntp.drift
solaris % rm /tmp/fifo.$Pid
```

我们用一个shell命令（echo）把客户（本shell）的进程ID和所请求的路径名发送给服务器，用另一个命令（cat）读出服务器的应答。这两个命令之间可相隔任意长度的时间。这么一来，表面上看先由服务器往客户的FIFO中写文件，再由客户执行cat命令从该FIFO中读出数据，这样的表象可能会使我们认为即使没有进程打开着客户的FIFO，数据也会以某种方式存留于该FIFO中。但事情并不是这样，真正的规则是：当对一个管道或FIFO的最终close发生时，该管道或FIFO中的任何残余数据都被丢弃。在我们的shell例子中，服务器读出客户的请求行后，会阻塞在对客户的FIFO的open调用中，因为客户（即我们的shell）还没有打开该FIFO来读（回想图4-21）。服务器对该FIFO的open调用一直阻塞到我们在以后某个时候执行cat命令为止，该命令打开这个FIFO来读，服务器的open调用随之返回。这种时间顺序关系还会导致拒绝服务（denial-of-service）型攻击，我们将在下一节讨论。

63 ~ 64

使用shell还允许简单测试服务器的出错处理。我们可以简单地向服务器发送一行不带进程ID的请求，也可以向它发送一行所带进程ID在/tmp目录中没有对应的FIFO的请求。举例来说，如果在启动服务器后输入如下行：

```
solaris % cat > /tmp/fifo.serv
/no/process/id
999999 /invalid/process/id
```

那么服务器的输出（在另一个窗口中）如下：

```
solaris % server
bogus request: /no/process/id
cannot open: /tmp/fifo.999999
```

4.8.1　FIFO write 的原子性

本节介绍的简单客户-服务器程序还能让我们体会到管道和FIFO的write操作的原子性相当重要。假设有两个客户差不多同时向服务器发送请求。第一个客户的请求行如下：

```
1234 /etc/inet/ntp.conf
```

第二个客户的请求行如下：

```
9876 /etc/passwd
```

假设每个客户给自己的请求行执行单个write函数调用，而且每个请求行都小于或等于PIPE_BUF（这样假设是合理的，因为PIPE_BUF通常在1024和5120之间，而路径名往往限制成最多1024字节），那么系统保证该FIFO中的数据或者是

```
1234 /etc/inet/ntp.conf
9876 /etc/passwd
```

或者是

```
9876 /etc/passwd
1234 /etc/inet/ntp.conf
```

该FIFO中的数据不会像是下面的模样：

65

```
1234 /etc/inet9876 /etc/passwd
/ntp.conf
```

4.8.2　FIFO 与 NFS 的关系

FIFO是一种只能在单台主机上使用的IPC形式。尽管在文件系统中有名字，它们也只能用在本地文件系统上，而不能用在通过NFS安装的文件系统上。

```
solaris % mkfifo /nfs/bsdi/usr/rstevens/fifo.temp
mkfifo: I/O error
```

上面的例子中，文件系统/nfs/bsdi/usr是主机bsdi上的/usr文件系统。

有些系统（例如BSD/OS）确实允许在通过NFS安装的文件系统上创建FIFO，但是数据无法在这样的两个系统间通过这些FIFO传递。这种情形下，FIFO只是用作同一主机上客户和服务器之间位于文件系统中的集结点。即使在不同主机上的某两个进程都能通过NFS打开同一个FIFO，它们也不能通过该FIFO从一个进程向另一个进程发送数据。

4.9　对比迭代服务器与并发服务器

上一节的简单客户-服务器程序例子中，服务器是一个*迭代服务器*（iterative server）。它逐一处理客户请求，而且是在完全处理每个客户的请求后再接待下一个客户。举例来说，如果有两个客户几乎同时向服务器发送一个请求——第一个客户请求一个10M字节的文件，服务器把它发送给该客户需花（譬如说）10秒，第二个客户请求一个10字节的文件——那么第二个客户必须等待至少10秒，以让第一个客户的请求被处理完。

另一种设计是*并发服务器*（concurrent server）。Unix下最常见的并发服务器类型称为每个客

户一个子进程（one-child-per-client）服务器，每当有一个客户请求到达时，这种服务器就让主进程调用fork派生出一个新的子进程。该新子进程处理相应的客户请求，直到完成为止，而Unix的多程序运行特性提供了所有不同进程间的并发性。UNPv1第27章还详细讨论了其他并发服务器设计技巧。

- 创建一个子进程池，让池中某个空闲子进程为一个新的客户服务。
- 为每个客户创建一个线程。
- 创建一个线程池，让池中某个空闲线程为一个新的客户服务。

尽管UNPv1中的讨论是针对网络服务器的，同样的技巧也适用于IPC服务器，差别只是IPC服务器的客户总是跟服务器运行在同一主机上。

拒绝服务型攻击

我们已提及迭代服务器的一个问题——某些客户必须等待比预期要长的时间，因为它们被排在请求处理时间较长的其他客户之后——然而还存在另一个问题。回想上一节中从shell完成与服务器交互的例子，我们讨论了当客户还没有打开自己的FIFO时（客户的打开操作要到我们执行cat命令时才发生），服务器是怎样阻塞在对该FIFO的open调用上的。这意味着某个恶意的客户可以让服务器处于停顿状态，办法是给它发送一个请求行，但从来不打开自己的FIFO来读。这称为拒绝服务（DoS）型攻击。为避免这种攻击，在编写任何服务器程序的迭代部分时必须小心，要留意服务器可能在哪儿阻塞以及可能阻塞多久。处理这种问题的方法之一是在特定操作上设置一个超时时钟，但是把服务器程序编写成并发服务器而不是迭代服务器通常更为简单，这么一来，上述类型的拒绝服务型攻击只影响一个子进程，而不会影响主服务器。即使采用并发服务器，拒绝服务型攻击仍可能发生：一个恶意的客户可能发送大量的独立请求，导致服务器达到它的子进程数限制，从而使得后续的fork失败。

66

4.10 字节流与消息

到此为止所给出的使用管道和FIFO的例子都使用字节流I/O模型，这是Unix的原生I/O模型。这种模型不存在记录边界，也就是说读写操作根本不检查数据。举例来说，从某个FIFO中读出100个字节的进程无法判定往该FIFO中写入这100个字节的进程执行了单个100字节的写操作、5个20字节的写操作、2个50字节的写操作还是另外某种总共为100字节的写操作的组合。一个进程往该FIFO中写入55个字节后，另一个进程再写入45字节，这样的情况同样是可能的。这样的数据是一个字节流（byte stream），系统不对它作解释。如果需要某种解释，写进程和读进程就得先验[①]地同意这种解释，并亲自去做。

有时候应用希望对所传送的数据加上某种结构。当数据由长度可变消息构成，并且读出者必须知道这些消息的边界以判定何时已读出单个消息时，这种需求可能发生。下面三种技巧经常用于这个目的。

(1) 带内特殊终止序列：许多Unix应用程序使用换行符来分隔每个消息。写进程会给每个消息添加一个换行符，读进程则每次读出一行。图4-23和图4-24中的客户程序和服务器程序就用这种方法分隔各个客户请求。这种技巧一般要求数据中任何出现分隔符处都作转义处理（也就是说以某种方式把它们标志成数据，而不是作为分隔符）。

① 由原因推出结果。——编者注

许多因特网应用程序（FTP、SMTP、HTTP、NNTP）使用由一个回车符后跟一个换行符构成的双字符序列（CR/LF）来分隔文本记录。

(2) 显式长度：每个记录前冠以它的长度。我们将马上使用这种技巧。当用在TCP上时，Sun RPC也使用这种技巧。这种技巧的优势之一是不再需要通过转义出现在数据中的分隔符，因为接收者不必扫描整个数据以寻找每个记录的结束位置。

67 (3) 每次连接一个记录：应用通过关闭与其对端的连接（网络应用时为TCP连接，IPC应用时为IPC连接）来指示一个记录的结束。这要求为每个记录创建一个新连接，HTTP 1.0就使用了这一技术。

标准I/O函数库也能用于读或写一个管道或FIFO。既然打开一个管道的唯一方法是使用pipe函数（它返回的是文件描述符），为创建一个新的标准I/O流（stream），必须使用标准I/O函数fdopen以将该标准I/O流与由pipe返回的某个已打开描述符相关联。

也可以构建更为结构化的消息，这种能力是由Posix消息队列和System V消息队列提供的。我们将看到每个消息有一个长度和一个优先级（System V称后者为“类型”）。长度和优先级是由发送者指定的，消息被读出后，这两者都返回给读出者。每个消息是一个记录（record），类似于UDP数据报（UNPvl）。

我们也能给一个管道或FIFO增加些结构。在图4-25所示的mesg.h头文件中，我们定义了一个消息。

pipemesg/mesg.h

```
 1 #include     "unpipc.h"

 2 /* Our own "messages" to use with pipes, FIFOs, and message queues. */

 3         /* want sizeof(struct mymesg) <= PIPE_BUF */
 4 #define MAXMESGDATA (PIPE_BUF - 2*sizeof(long))

 5         /* length of mesg_len and mesg_type */
 6 #define MESGHDRSIZE (sizeof(struct mymesg) - MAXMESGDATA)

 7 struct mymesg {
 8     long    mesg_len;              /* #bytes in mesg_data, can be 0 */
 9     long    mesg_type;             /* message type, must be > 0 */
10     char    mesg_data[MAXMESGDATA];
11 };

12 ssize_t  mesg_send(int, struct mymesg *);
13 void     Mesg_send(int, struct mymesg *);
14 ssize_t  mesg_recv(int, struct mymesg *);
15 ssize_t  Mesg_recv(int, struct mymesg *);
```

pipemesg/mesg.h

图4-25　我们的mymesg结构及相关定义

每个消息有一个*mesg_type*，我们把它定义成一个值必须大于0的整数。我们现在暂时忽略这个类型成员，到第6章中讲述System V消息队列时再讨论它。每个消息还有一个长度，我们允许它为0。我们使用mymesg结构在每个消息前冠以它的长度，而没有使用换行符来分隔消息。早些时候我们提到过这种设计方法的两个好处：接收者不必扫描所收到的每个字节以找出消息的结束位置，即使分隔符（换行符）出现在消息中也不必将它转义。

68 图4-26展示了mymesg结构的图示以及我们如何随管道、FIFO和System V消息队列使用它。

我们定义两个函数分别发送和接收消息。图4-27给出了我们的mesg_send函数，图4-28给出了我们的mesg_recv函数。

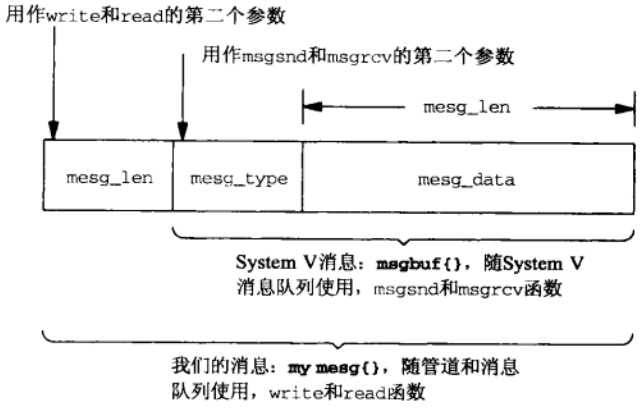


图4-26 我们的mymesg结构

pipemesg/mesg_send.c

```
1 #include     "mesg.h"

2 ssize_t
3 mesg_send(int fd, struct mymesg *mptr)
4 {
5       return(write(fd, mptr, MESGHDRSIZE + mptr->mesg_len));
6 }
```

pipemesg/mesg_send.c

图4-27 mesg_send函数

pipemesg/mesg_recv.c

```
 1 #include     "mesg.h"

 2 ssize_t
 3 mesg_recv(int fd, struct mymesg *mptr)
 4 {
 5     size_t  len;
 6     ssize_t n;

 7         /* read message header first, to get len of data that follows */
 8     if ( (n = Read(fd, mptr, MESGHDRSIZE)) == 0)
 9         return(0);              /* end of file */
10     else if (n != MESGHDRSIZE)
11         err_quit("message header: expected %d, got %d", MESGHDRSIZE, n);

12     if ( (len = mptr->mesg_len) > 0)
13         if ( (n = Read(fd, mptr->mesg_data, len)) != len)
14             err_quit("message data: expected %d, got %d", len, n);
15     return(len);
16 }
```

pipemesg/mesg_recv.c

图4-28 mesg_recv函数

现在读出每个消息需调用两个read，一个读出消息长度，另一个读出真正的消息（如果长度大于0的话）。

细心的读者可能注意到mesg_recv会检查所有可能的错误是否发生，一旦发现则马上终止。然而为一致性起见，我们还是定义了一个名为Mesg_recv的包裹函数，并在我们的程序中调用它。

现在把我们的客户函数和服务器函数改为使用mesg_send和mesg_recv函数。图4-29给出了新的客户函数。

pipemesg/client.c

```
 1 #include    "mesg.h"

 2 void
 3 client(int readfd, int writefd)
 4 {
 5     size_t  len;
 6     ssize_t n;
 7     struct mymesg  mesg;

 8         /* read pathname */
 9     Fgets(mesg.mesg_data, MAXMESGDATA, stdin);
10     len = strlen(mesg.mesg_data);
11     if (mesg.mesg_data[len-1] == '\n')
12         len--;            /* delete newline from fgets() */
13     mesg.mesg_len = len;
14     mesg.mesg_type = 1;

15         /* write pathname to IPC channel */
16     Mesg_send(writefd, &mesg);

17         /* read from IPC, write to standard output */
18     while ( (n = Mesg_recv(readfd, &mesg)) > 0)
19         Write(STDOUT_FILENO, mesg.mesg_data, n);
20 }
```

pipemesg/client.c

图4-29　我们的使用消息的client函数

读出路径名，发送给服务器

8~16　从标准输入读出路径名，然后使用mesg_send将其发送给服务器。

读出来自服务器的文件内容或出错消息

17~19　客户在一个循环中调用mesg_recv，读出服务器发送回的所有东西。按照约定，mesg_recv返回一个值为0的长度表示已到达来自服务器的数据的结尾。我们将看到服务器将在发送给客户的每个消息中都包含换行符，因此空行也会有一个值为1的消息长度。

69 ~ 70

图4-30给出了新的服务器函数。

从IPC通道读出路径名，打开文件

8~18　读出来自客户的路径名。尽管这儿给mesg_type赋值为1看来无用（它将被图4-28中的mesg_recv覆写），在使用System V消息队列时（图6-10），我们仍然会调用本函数，那时是需要这样的赋值的（例如图6-13）。标准I/O函数fopen打开该路径名的文件，这与图4-10中调用Unix I/O函数open获得访问该文件的一个描述符不一样。这儿我们调用标准I/O函数库的原因是为了调用fgets逐行地读出该文件，然后把每一行作为一个消息发送给客户。

将文件复制给客户

19~26　如果fopen调用成功，就使用fgets读出该文件并发送给客户，每个消息一行。一个长度为0的消息表示已到达文件尾。

在使用管道或FIFO时，也可以通过关闭IPC通道来通知对端已到达输入文件的结尾。不过我们通过发送回一个长度为0的消息来达到同样目的，因为之后还会遇到没有文件结束符概念的其他类型的IPC。

pipemesg/server.c

```
1 #include    "mesg.h"
2 void
3 server(int readfd, int writefd)
4 {
5     FILE    *fp;
6     ssize_t n;
7     struct mymesg  mesg;
8         /* read pathname from IPC channel */
9     mesg.mesg_type = 1;
10    if ( (n = Mesg_recv(readfd, &mesg)) == 0)
11        err_quit("pathname missing");
12    mesg.mesg_data[n] = '\0';      /* null terminate pathname */
13    if ( (fp = fopen(mesg.mesg_data, "r")) == NULL) {
14            /* error: must tell client */
15        snprintf(mesg.mesg_data + n, sizeof(mesg.mesg_data) - n,
16                 ": can't open, %s\n", strerror(errno));
17        mesg.mesg_len = strlen(mesg.mesg_data);
18        Mesg_send(writefd, &mesg);
19    } else {
20            /* fopen succeeded: copy file to IPC channel */
21        while (Fgets(mesg.mesg_data, MAXMESGDATA, fp) != NULL) {
22            mesg.mesg_len = strlen(mesg.mesg_data);
23            Mesg_send(writefd, &mesg);
24        }
25        Fclose(fp);
26    }
27        /* send a 0-length message to signify the end */
28    mesg.mesg_len = 0;
29    Mesg_send(writefd, &mesg);
30 }
```

pipemesg/server.c

图4-30 我们的使用消息的server函数

调用我们的client和server函数的main函数根本不变。我们既可使用管道版本（图4-8），也可使用FIFO版本（图4-16）。

4.11 管道和FIFO限制

系统加于管道和FIFO的唯一限制为：

OPEN_MAX　一个进程在任意时刻打开的最大描述符数（Posix要求至少为16）；

PIPE_BUF　可原子地写往一个管道或FIFO的最大数据量（我们在4.7节讲述过，Posix要求至少为512）。

我们马上会看到OPEN_MAX的值可通过调用sysconf函数查询。它通常可通过执行ulimit命令（Bourne shell或KornShell，我们马上会看到）或limit命令（C shell）从shell中修改。它也可通过调用setrlimit函数（在APUE的7.11节中详细讲述）从一个进程中修改。

PIPE_BUF的值通常定义在<limits.h>头文件中，但是Posix认为它是一个路径名变量（pathname variable）。这意味着它的值可以随所指定的路径名而变化（只对FIFO而言，因为管道没有名字），因为不同的路径名可以落在不同的文件系统上，而这些文件系统可能有不同的特征。

于是PIPE_BUF的值可在运行时通过调用pathconf或fpathconf取得。图4-31给出了输出这两个限制值的一个例子程序。

pipe/pipeconf.c

```
1 #include    "unpipc.h"
2 int
3 main(int argc, char **argv)
4 {
5     if (argc != 2)
6         err_quit("usage: pipeconf <pathname>");
7     printf("PIPE_BUF = %ld, OPEN_MAX = %ld\n",
8            Pathconf(argv[1], _PC_PIPE_BUF), Sysconf(_SC_OPEN_MAX));
9     exit(0);
10 }
```

pipe/pipeconf.c

71 ~ 72

图4-31　在运行时确定PIPE_BUF和OPEN_MAX的值

下面是一些例子，指定了不同的文件系统。

```
solaris % pipeconf /                    Solaris 2.6默认值
PIPE_BUF = 5120, OPEN_MAX = 64
solaris % pipeconf /home
PIPE_BUF = 5120, OPEN_MAX = 64
solaris % pipeconf /tmp
PIPE_BUF = 5120, OPEN_MAX = 64

alpha % pipeconf /                      Digital Unix 4.0B默认值
PIPE_BUF = 4096, OPEN_MAX = 4096
alpha % pipeconf /usr
PIPE_BUF = 4096, OPEN_MAX = 4096
```

下面给出在Solaris下如何使用KornShell修改OPEN_MAX的值。

```
solaris % ulimit -nS                    输出最大描述符数，软限制
64
solaris % ulimit -nH                    输出最大描述符数，硬限制
1024
solaris % ulimit -ns 512                设置软限制为512
solaris % pipeconf /                    验证该变动已发生
PIPE_BUF = 5120, OPEN_MAX = 512
```

尽管能够修改FIFO的PIPE_BUF的值，这取决于路径名所存放的底层文件系统，但实际应该很少这么做。

APUE的第2章描述了fpathconf、pathconf和sysconf函数，这些函数提供了有关特定内核限制的运行时信息。Posix.1定义了以_PC_开头的12个常值和以_SC_开头的52个常值。Digital Unix 4.0B和Solaris 2.6都对后者作了扩充，定义了约100个可使用sysconf查询的运行时常值。

Posix.2定义了getconf命令，它可以输出这些实现限制值中的大多数。例如：

```
alpha % getconf OPEN_MAX
4096
alpha % getconf PIPE_BUF /
4096
```

4.12　小结

管道和FIFO是许多应用程序的基本构建模块。管道普遍用于shell中，不过也可以从程序中

使用，往往是用于从子进程向父进程回传信息。使用管道时涉及的某些代码（pipe、fork、close、exec和waitpid）可通过使用popen和pclose来避免，由它们处理具体细节并激活一个shell。 73

FIFO与管道类似，但它们是用mkfifo创建的，之后需用open打开。打开管道时必须小心，因为有许多规则（图4-21）制约着open的阻塞与否。

在使用管道和FIFO的前提下，我们查看了一些客户-服务器程序设计：一个服务器服务多个客户，服务器可以是迭代的，也可以是并发的。迭代服务器以串行方式每次处理一个客户请求，它们易遭受拒绝服务型攻击。并发服务器则让另外一个进程或线程处理每个客户请求。

管道和FIFO的特征之一是它们的数据是一个字节流，类似于TCP连接。把这种字节流分隔成各个记录的任何方法都得由应用程序来实现。我们将在接下去的两章中看到消息队列会提供记录边界，类似于UDP数据报。

习题

4.1 在从图4-3到图4-4的转换中，如果子进程没有执行close(fd[1])，那么会发生什么情况？

4.2 在4.6节描述mkfifo时，我们说过要打开一个已有的FIFO或创建一个新的FIFO，应该调用mkfifo，检查是否返回EEXIST错误，若是则调用open。如果把这个逻辑关系变换一下，先调用open，当不存在所期望的FIFO时再调用mkfifo，那么会发生什么情况？

4.3 在图4-15中调用popen时，如果其中执行命令的shell碰到错误，那么会发生什么情况？

4.4 把图4-23中针对服务器的FIFO的open去掉，验证一下这将导致当不再有客户存在时，服务器即终止。

4.5 在图4-23中我们指出，当服务器启动后，它阻塞在自己的第一个open调用中，直到客户的第一个open打开同一个FIFO用于写为止。我们怎样才能绕过这样的阻塞，使得两个open都立即返回，转而阻塞在首次调用readline上？

4.6 如果图4-24中的客户程序对换其两个open调用的顺序，那么会发生什么情况？

4.7 为什么在读进程关闭管道或FIFO之后给写进程产生一个信号，而不会在写进程关闭管道或FIFO之后给读进程产生一个信号？

4.8 编写一个小测试程序，确定fstat是否以stat结构的st_size成员的形式返回当前在某个FIFO中的数据字节数。

4.9 写一个小测试程序，以确定在为一个读出端已关闭的管道描述符选择可写条件时select返回的内容。 74

第5章 Posix 消息队列

5.1 概述

消息队列可认为是一个消息链表。有足够写权限的线程可往队列中放置消息，有足够读权限的线程可从队列中取走消息。每个消息都是一个记录（回想我们在4.10节就字节流和消息的讨论），它由发送者赋予一个优先级。在某个进程往一个队列写入消息之前，并不需要另外某个进程在该队列上等待消息的到达。这跟管道和FIFO是相反的，对后两者来说，除非读出者已存在，否则先有写入者是没有意义的。

一个进程可以往某个队列写入一些消息，然后终止，再让另外一个进程在以后某个时刻该出这些消息。我们说过消息队列具有随内核的持续性（1.3节），这跟管道和FIFO不一样。我们在第4章中说过，当一个管道或FIFO的最后一次关闭发生时，仍在该管道或FIFO上的数据将被丢弃。

本章讲述Posix消息队列，第6章讲述System V消息队列。这两组函数间存在许多相似性，下面是主要的差别。

- 对Posix消息队列的读总是返回最高优先级的最早消息，对System V消息队列的读则可以返回任意指定优先级的消息。
- 当往一个空队列放置一个消息时，Posix消息队列允许产生一个信号或启动一个线程，System V消息队列则不提供类似机制。

队列中的每个消息具有如下属性：

- 一个无符号整数优先级（Posix）或一个长整数类型（System V）；
- 消息的数据部分长度（可以为0）；
- 数据本身（如果长度大于0）。

注意这些特征不同于管道和FIFO。后两者是字节流模型，没有消息边界，也没有与每个消息关联的类型。我们在4.10节就此讨论过，并给管道和FIFO增设了自己的消息接口。

图5-1展示了一个消息队列的可能布局。

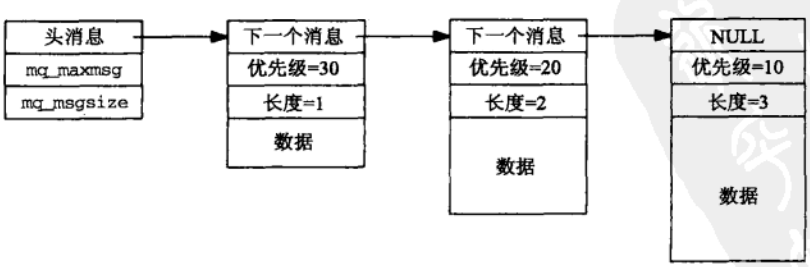


图5-1 含有三个消息的某个Posix消息队列的可能布局

我们所设想的是一个链表，该链表的头中含有当前队列的两个属性：队列中允许的最大消息数以及每个消息的最大大小。我将在5.3节中详细讨论这两个属性。

从本章开始我们使用一种新的程序设计技巧，它在以后讨论消息队列、信号量和共享内存区的各章中也会用到。既然所有这些IPC对象至少具有随内核的持续性（回想1.3节），于是我们可以编写若干小程序来使用这些IPC机制，以便深入地认识它们，并深刻地了解它们的操作。例如，我们可编写一个程序来创建一个Posix消息队列，编写另一个程序来往某个Posix消息队列中加入一个消息，再编写另一个程序来从某个Posix消息队列中读出一个消息。通过以不同的优先级构造各个消息，我们可以看出mq_receive函数是怎样返回这些消息的。

5.2 mq_open、mq_close 和 mq_unlink 函数

mq_open函数创建一个新的消息队列或打开一个已存在的消息队列。

```
#include <mqueue.h>

mqd_t mq_open(const char *name, int oflag, ...
              /* mode_t mode, struct mq_attr *attr */ );
```

返回：若成功则为消息队列描述符，若出错则为-1

76

我们已在2.2节描述过有关*name*参数的规则。

*oflag*参数是O_RDONLY、O_WRONLY或O_RDWR之一，可能按位或上O_CREAT、O_EXCL或O_NONBLOCK。我们已在2.3节讲述过所有这些标志。

当实际操作是创建一个新队列时（已指定O_CREAT标志，且所请求的消息队列尚未存在），*mode*和*attr*参数是需要的。我们在图2-4中给出了*mode*值。*attr*参数用于给新队列指定某些属性。如果它为空指针，那就使用默认属性。我们将在5.3节讨论这些属性。

mq_open的返回值称为消息队列描述符（mesage queue descriptor），但它不必是（而且很可能不是）像文件描述符或套接字描述符这样的短整数。这个值用作其余7个消息队列函数的第一个参数。

> Solaris 2.6把mqd_t定义为void*，而Digital Unix 4.0B把它定义为int。在5.8节我们的Posix消息队列实现例子中，消息队列描述符是一个结构指针。称这些数据类型为描述符是一个不幸的错误。

已打开的消息队列是由mq_close关闭的。

```
#include <mqueue.h>

int mq_close(mqd_t mqdes);
```

返回：若成功则为0，若出错则为-1

其功能与关闭一个已打开文件的close函数类似：调用进程可以不再使用该描述符，但其消息队列并不从系统中删除。一个进程终止时，它的所有打开着的消息队列都关闭，就像调用了mq_close一样。

要从系统中删除用作mq_open第一个参数的某个*name*，必须调用mq_unlink。

```
#include <mqueue.h>

int mq_unlink(const char *name);
```

返回：若成功则为0，若出错则为-1

每个消息队列有一个保存其当前打开着描述符数的引用计数器（就像文件一样）[①]，因而本函数能够实现类似于unlink函数删除一个文件的机制：当一个消息队列的引用计数仍大于0时，其*name*就能删除，但是该队列的析构（这与从系统中删除其名字不同）要到最后一个mq_close发生时才进行[②]。

Posix消息队列至少具备随内核的持续性（回想1.3节）。这就是说，即使当前没有进程打开着某个消息队列，该队列及其上的各个消息也将一直存在，直到调用mq_unlink并让它的引用计数达到0以删除该队列为止。

77

我们将看到，如果消息队列是使用内存映射文件（12.2节）实现的，那么它们具有随文件系统的持续性，但这不是必需的，因而不能指望。

5.2.1　例子：mqcreate1程序

既然Posix消息队列至少具有随内核的持续性，我们于是可以编写一组小程序来操纵这些队列，以提供认知它们的简易办法。图5-2中的程序创建一个消息队列，其名字是作为命令行参数指定的。

pxmsg/mqcreate1.c

```
 1 #include    "unpipc.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     c, flags;
 6     mqd_t   mqd;

 7     flags = O_RDWR | O_CREAT;
 8     while ( (c = Getopt(argc, argv, "e")) != -1) {
 9         switch (c) {
10         case 'e':
11             flags |= O_EXCL;
12             break;
13         }
14     }
15     if (optind != argc - 1)
16         err_quit("usage: mqcreate [ -e ] <name>");

17     mqd = Mq_open(argv[optind], flags, FILE_MODE, NULL);

18     Mq_close(mqd);
19     exit(0);
20 }
```

pxmsg/mqcreate1.c

图5-2　指定排他性创建标志，创建一个消息队列

8~16　我们允许有一个指定排他性创建的-e选项。（关于getopt函数和它的Getopt包裹函数，我们将随图5-5详细讨论。）返回时，getopt在optind中存放下一个待处理参数的下标。

17　直接以来自命令行的IPC名字调用mq_open，而不去调用px_ipc_name函数（2.2节）。这样我们就能准确地看出实现是如何处理这些Posix IPC名字的。（本书中的所有简单测试程序都这么处理。）

① 一个消息队列的名字在系统中的存在本身也占用其引用计数器的一个引用数。mq_unlink从系统中删除该名字意味着同时将其引用计数减1，若变为0则真正拆除该队列。——译者注

② 跟mq_unlink一样，mq_close也将当前消息队列的引用计数减1，若变为0则附带拆除该队列。——译者注

下面是在Solaris 2.6下的输出。

```
solaris % mqcreate1 /temp.1234                第一次创建成功
solaris % ls -l /tmp/.*1234
-rw-rw-rw-   1 rstevens other1  132632 Oct 23 17:08 /tmp/.MQDtemp.1234
-rw-rw-rw-   1 rstevens other1       0 Oct 23 17:08 /tmp/.MQLtemp.1234
-rw-r--r--   1 rstevens other1       0 Oct 23 17:08 /tmp/.MQPtemp.1234
solaris % mqcreate1 -e /temp.1234             指定-e选项的第二次创建失败
mq_open error for /temp.1234: File exists
```

78

（我们称这个程序为mqcreate1，因为它将在我们说明属性后，在图5-5中得以改进。）第三个文件（/temp/.MQPtemp.1234）具有我们用FILE_MODE常值（用户可读可写，组成员和其他用户只读）指定的权限，另外两个文件的权限则不一样。我们猜测文件名中含有D的文件含有数据，含有L的文件是某种类型的锁，含有P的文件指定权限。

在Digital Unix 4.0B下，我们可看到所创建的是真正的路径名。

```
alpha % mqcreate1 /tmp/myq.1234
alpha % ls -l /tmp/myq.1234
-rw-r--r--   1 rstevens system   11976 Oct 23 17:04 /tmp/myq.1234
alpha % mqcreate1 -e /tmp/myq.1234
mq_open error for /tmp/myq.1234: File exists
```

5.2.2 例子：mqunlink 程序

图5-3中的mqunlink程序从系统中删除一个消息队列。

pxmsg/mqunlink.c

```
1 #include     "unpipc.h"
2 int
3 main(int argc, char **argv)
4 {
5       if (argc != 2)
6           err_quit("usage: mqunlink <name>");
7       Mq_unlink(argv[1]);
8       exit(0);
9 }
```

pxmsg/mqunlink.c

图5-3 mq_unlink一个消息队列

我们可使用该程序删除由mqcreate程序创建的消息队列。

```
solaris % mqunlink /temp.1234
```

我们早先给出的/tmp目录下的所有三个文件都被删除了。

5.3 mq_getattr 和 mq_setattr 函数

每个消息队列有四个属性，mq_getattr返回所有这些属性，mq_setattr则设置其中某个属性。

```
#include <mqueue.h>
int mq_getattr(mqd_t mqdes, struct mq_attr *attr);
int mq_setattr(mqd_t mqdes, const struct mq_attr *attr, struct mq_attr *oattr);
                                                    均返回：若成功则为0，若出错则为-1
```

79

mq_attr结构含有以下属性。

```
struct mq_attr {
  long mq_flags;   /* message queue flag: 0, O_NONBLOCK */
  long mq_maxmsg;  /* max number of messages allowed on queue */
  long mq_msgsize; /* max size of a message (in bytes) */
  long mq_curmsgs; /* number of messages currently on queue */
};
```

指向某个mq_attr结构的指针可作为mq_open的第四个参数传递，从而允许我们在该函数的实际操作是创建一个新队列时，给它指定mq_maxmsg和mq_msgsize属性。mq_open忽略该结构的另外两个成员。

mq_getattr把所指定队列的当前属性填入由*attr*指向的结构。

mq_setattr给所指定队列设置属性，但是只使用由*attr*指向的mq_attr结构的mq_flags成员，以设置或清除非阻塞标志。该结构的另外三个成员被忽略：每个队列的最大消息数和每个消息的最大字节数只能在创建队列时设置，队列中的当前消息数则只能获取而不能设置。

另外，如果*oattr*指针非空，那么所指定队列的先前属性（mq_flags、mq_maxmsg和mq_msgsize）和当前状态（mq_curmsgs）将返回到由该指针指向的结构中。

5.3.1　例子：mqgetattr程序

图5-4中的程序打开一个指定的消息队列，并输出其属性。

pxmsg/mqgetattr.c

```
 1 #include    "unpipc.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     mqd_t   mqd;
 6     struct mq_attr  attr;
 7     if (argc != 2)
 8         err_quit("usage: mqgetattr <name>");
 9     mqd = Mq_open(argv[1], O_RDONLY);
10     Mq_getattr(mqd, &attr);
11     printf("max #msgs = %ld, max #bytes/msg = %ld, "
12            "#currently on queue = %ld\n",
13            attr.mq_maxmsg, attr.mq_msgsize, attr.mq_curmsgs);
14     Mq_close(mqd);
15     exit(0);
16 }
```

pxmsg/mqgetattr.c

图5-4　取得并输出某个消息队列的属性

80　我们可以创建一个消息队列并输出其默认属性。

```
solaris % mqcreate1 /hello.world
solaris % mqgetattr /hello.world
max #msgs = 128, max #bytes/msg = 1024, #currently on queue = 0
```

现在可以看出，在图5-2之后以默认属性创建一个队列的例子中，由ls列出的数据文件（/tmp/.MQDtemp.1234）的大小为128×1024+1560=132 632。1560个额外字节也许是开销信息：每个消息8字节，外加另外536个字节。

5.3.2 例子：mqcreate 程序

我们可以对图5-2中的程序作修改，以允许指定所创建队列的最大消息数和每个消息的最大大小。我们不能只指定其中一个而不指定另一个，即两者都得指定（不过见习题5.1）。图5-5是修改后的程序。

pxmsg/mqcreate.c

```
1 #include        "unpipc.h"
2 struct mq_attr  attr;              /* mq_maxmsg and mq_msgsize both init to 0 */
3 int
4 main(int argc, char **argv)
5 {
6     int     c, flags;
7     mqd_t   mqd;
8     flags = O_RDWR | O_CREAT;
9     while ( (c = Getopt(argc, argv, "em:z:")) != -1) {
10        switch (c) {
11        case 'e':
12            flags |= O_EXCL;
13            break;
14        case 'm':
15            attr.mq_maxmsg = atol(optarg);
16            break;
17        case 'z':
18            attr.mq_msgsize = atol(optarg);
19            break;
20        }
21    }
22    if (optind != argc - 1)
23        err_quit("usage: mqcreate [ -e ] [ -m maxmsg -z msgsize ] <name>");
24    if ((attr.mq_maxmsg != 0 && attr.mq_msgsize == 0) ||
25        (attr.mq_maxmsg == 0 && attr.mq_msgsize != 0))
26        err_quit("must specify both -m maxmsg and -z msgsize");
27    mqd = Mq_open(argv[optind], flags, FILE_MODE,
28                  (attr.mq_maxmsg != 0) ? &attr : NULL);
29    Mq_close(mqd);
30    exit(0);
31 }
```

pxmsg/mqcreate.c

图5-5 改进后的图5-2，允许指定属性

81

指定某个命令行选项需要一个参数，我们在getopt调用中给这些选项字节（m和z）指定了一个后跟的冒号。在处理这样的选项字符时，optarg指向其参数。

我们的Getopt包裹函数调用标准函数库中的getopt函数，并在getopt检测到错误时终止当前进程，这些错误包括：遇到一个没有包含在getopt第三个参数中的选项字母，或者遇到一个没有所需参数的选项字母（通过在getopt的第三个参数中的该选项字母之后跟一个冒号指示）。不论遇到哪种错误，getopt都将一个出错消息写到标准错误输出，然后返回一个错误，这个错误导致我们的Getopt包裹函数终止。例如，如下两个错误由getopt检测出。

```
solaris % mqcreate -z
mqcreate: option requires an argument -- z
```

```
solaris % mqcreate -q
mqcreate: illegal option -- q
```

下面这个错误（没有指定所需的名字参数）则由我们的程序检测出。

```
solaris % mqcreate
usage: mqcreate [ -e ] [ -m maxmsg -z msgsize ] <name>
```

如果这两个新的命令行选项都没有指定，我们就给mq_open传递一个空指针作为最后一个参数，否则传递一个根据所指定命令行选项的参数构造的*attr*结构。

现在运行这个新版本的程序，指定一个最多有1024个消息的队列，每个消息最多有8192个字节。

```
solaris % mqcreate -e -m 1024 -z 8192 /foobar
solaris % ls -al /tmp/.*foobar
-rw-rw-rw-   1 rstevens other1 8397336 Oct 25 11:29 /tmp/.MQDfoobar
-rw-rw-rw-   1 rstevens other1       0 Oct 25 11:29 /tmp/.MQLfoobar
-rw-r--r--   1 rstevens other1       0 Oct 25 11:29 /tmp/.MQPfoobar
```

含有该队列之数据的文件（/tmp/.MQDfoobar）其大小能容纳最大数目的最长消息（1024×8192= 8 388 608），剩余的8728字节开销允许每个消息占用8个字节（8×1024=8192），外加536个字节。

在Digital Unix 4.0B下执行同一程序的结果如下：

```
alpha % mqcreate -m 256 -z 2048 /tmp/bigq
alpha % ls -l /tmp/bigq
-rw-r--r--   1 rstevens system  537288 Oct 25 15:38 /tmp/bigq
```

该系统上的实现看来能容纳最大数目的最长消息（256×2048=524 288），剩余的13 000字节开销允许每个消息占用48个字节（48×256 = 12 288），外加712个字节。

5.4　mq_send和mq_receive函数

这两个函数分别用于往一个队列中放置一个消息和从一个队列中取走一个消息。每个消息有一个优先级，它是一个小于MQ_PRIO_MAX的无符号整数。Posix要求这个上限至少为32。

Solaris 2.6的上限为32，Digital Unix 4.0B的上限为256。我们将随图5-8给出取得该上限值的方法。

82

mq_receive总是返回所指定队列中最高优先级的最早消息，而且该优先级能随该消息的内容及其长度一同返回。

mq_receive的操作不同于Systme V的msgrcv（6.4节）。System V消息有一个类似于优先级的类型字段，但使用msgrcv时，我们可以就返回哪一个消息指定三种不同的情形：所指定队列中最早的消息、具有某个特定类型的最早消息、其类型小于或等于某个值的最早消息。

```
#include <mqueue.h>

int mq_send(mqd_t mqdes, const char *ptr, size_t len, unsigned int prio);
                                                    返回：若成功则为0，若出错则为-1

ssize_t mq_receive(mqd_t mqdes, char *ptr, size_t len, unsigned int *priop);
                                                    返回：若成功则为消息中字节数，若出错则为-1
```

这两个函数的前三个参数分别与write和read的前三个参数类似。

把指向缓冲区的指针参数定义为char*看来是个错误。使用void*将与其他Posix.1函数更为一致。

mq_receive的*len*参数的值不能小于能加到所指定队列中的消息的最大大小（该队列mq_attr结构的mq_msgsize成员）。要是*len*小于该值，mq_receive就立即返回EMSGSIZE错误。

这意味着使用Posix消息队列的大多数应用程序必须在打开某个队列后调用mq_getattr确定最大消息大小，然后分配一个或多个那样大小的读缓冲区。通过要求每个缓冲区总是足以存放队列中的任意消息，mq_receive就不必返回消息是否大于缓冲区的通知。作为比较的例子，System V消息队列（6.4节）可能使用MSG_NOERROR标志，返回E2BIG错误，接收UDP数据报的recvmsg函数（UNPv1的13.5节）可能使用MSG_TRUNC标志。

mq_send的*prio*参数是待发送消息的优先级，其值必须小于MQ_PRIO_MAX。如果mq_receive的*priop*参数是一个非空指针，所返回消息的优先级就通过该指针存放。如果应用不必使用优先级不同的消息，那就给mq_send指定值为0的优先级，给mq_receive指定一个空指针作为其最后一个参数。

0字节长度的消息是允许的。这里重要的不是标准（即Posix.1）中说的内容，而是它的言外之意：没有地方可以禁止使用0字节长度的消息。mq_receive的返回值是所接收消息中的字节数（如果成功）或-1（如果出错），因此返回值为0表示返回长度为0的消息。

Posix消息队列和System V消息队列都不具备的一个特性是：向接收者准确地标识每条消息的发送者。这个信息在许多应用中可能有用。不幸的是，大多数IPC消息机制并不标识发送者。在15.5节中，我们将讲述门如何提供这个标识。UNPv1的14.8节讲述了使用Unix域套接字时，BSD/OS如何提供这个标识。APUE的15.3.1节[①]讲述了通过管道传递描述符时，SVR4如何通过同一管道传递发送者的标识。BSD/OS技术没有得到广泛实现，而SVR4技术尽管是Unix 98的一部分，却要求通过管道传递描述符，这通常比通过管道直接传递数据开销更大。我们不能让发送者随消息包含自己的标识（例如有效用户ID），因为难以认定发送者不说假话。尽管消息队列的访问权限决定了是否允许发送者往队列中放置消息，这仍然没有标识发送者。另外，尽管存在为每个发送者创建一个队列的可能性（对此我们将在6.8节就System V消息队列展开讨论），但对于大的应用来说，这种方法的可扩展性并不好。最后，要是消息队列函数完全是作为用户函数实现的（5.8节中的实现就是这样），根本没在内核中，那么我们不能信任伴随消息的任何发送者标识，因为它很容易伪造。

83

5.4.1 例子：mqsend程序

图5-6给出了往某个队列中增加一个消息的mqsend程序。

pxmsg/mqsend.c

```
1 #include     "unpipc.h"

2 int
3 main(int argc, char **argv)
4 {
5       mqd_t    mqd;
6       void    *ptr;
7       size_t   len;
8       uint_t   prio;
```

图5-6 mqsend程序

① 此处为APUE第1版英文原版书节号，第2版的17.4.1节讲述通过管道传递文件描述符。——编者注

```
 9      if (argc != 4)
10          err_quit("usage: mqsend <name> <#bytes> <priority>");
11      len = atoi(argv[2]);
12      prio = atoi(argv[3]);

13      mqd = Mq_open(argv[1], O_WRONLY);

14      ptr = Calloc(len, sizeof(char));
15      Mq_send(mqd, ptr, len, prio);

16      exit(0);
17  }
```

pxmsg/mqsend.c

图5-6（续）

待发送消息的大小和优先级必须作为命令行参数指定。所用缓冲区使用calloc分配，该函数会把该缓冲区初始化为0。

5.4.2　例子：mqreceive程序

84

图5-7中的程序从某个队列中读入下一个消息。

pxmsg/mqreceive.c

```
 1 #include    "unpipc.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5      int     c, flags;
 6      mqd_t   mqd;
 7      ssize_t n;
 8      uint_t  prio;
 9      void    *buff;
10      struct mq_attr  attr;

11      flags = O_RDONLY;
12      while ( (c = Getopt(argc, argv, "n")) != -1) {
13          switch (c) {
14          case 'n':
15              flags |= O_NONBLOCK;
16              break;
17          }
18      }
19      if (optind != argc - 1)
20          err_quit("usage: mqreceive [ -n ] <name>");

21      mqd = Mq_open(argv[optind], flags);
22      Mq_getattr(mqd, &attr);

23      buff = Malloc(attr.mq_msgsize);

24      n = Mq_receive(mqd, buff, attr.mq_msgsize, &prio);
25      printf("read %ld bytes, priority = %u\n", (long) n, prio);

26      exit(0);
27  }
```

pxmsg/mqreceive.c

图5-7　mqreceive程序

允许-n选项以指定非阻塞属性

14~17　命令行选项-n指定非阻塞属性，这样如果所指定队列中没有消息，mqreceive程序就

返回一个错误。

打开队列并取得属性

21~25 调用mq_getattr打开队列并取得属性。我们需要确定最大消息大小，因为必须为调用mq_receive分配一个这样大小的缓冲区。最后输出所读出消息的大小及其属性。

> 既然n是一个size_t数据类型，而我们又不知道size_t是int还是long，那么要使用%ld格式化串将n类型强制转换成一个长整数。在64位系统上，int是32位整数，long和size_t则都是64位整数。

我们可使用这两个程序来查看优先级字段是如何使用的。

```
solaris % mqcreate /test1                    创建并取得属性
solaris % mqgetattr /test1
max #msgs = 128, max #bytes/msg = 1024, #currently on queue = 0

solaris % mqsend /test1 100 99999            以无效的优先级发送
mq_send error: Invalid argument

solaris % mqsend /test1 100 6                100字节，优先级为6
solaris % mqsend /test1 50 18                50字节，优先级为18
solaris % mqsend /test1 33 18                50字节，优先级为18

solaris % mqreceive /test1
read 50 bytes, priority = 18                 返回优先级最高的最早消息
solaris % mqreceive /test1
read 33 bytes, priority = 18
solaris % mqreceive /test1
read 100 bytes, priority = 6
solaris % mqreceive-n /test1                 指定非阻塞属性，队列为空
mq_receive error: Resource temporarily unavailable
```

可以看出，mq_receive返回优先级最高的最早消息。

5.5 消息队列限制

我们已遇到任意给定队列的两个限制，它们都是在创建该队列时建立的：

mq_maxmsg　队列中的最大消息数；

mq_msgsize　给定消息的最大字节数。

这两个值都没有内在的限制，尽管对于我们已查看过的两种实现（Solaris 2.6和Digital Unix 4.0B）来说，大小为这两个数之积再加少量开销的某个文件在文件系统中必须有容纳空间。基于队列大小的虚拟内存要求也可能存在（见习题5.5）。

消息队列的实现定义了另外两个限制：

MQ_OPEN_MAX　一个进程能够同时拥有的打开着消息队列的最大数目（Posix要求它至少为8）；

MQ_PRIO_MAX　任意消息的最大优先级值加1（Posix要求它至少为32）。

这两个常值往往定义在<unistd.h>头文件中，也可以在运行时通过调用sysconf函数获取，如接下来的例子所示。

例子：mqsysconf 程序

图5-8中的程序调用sysconf，输出消息队列的两个由实现定义的限制。

pxmsg/mqsysconf.c

```
1 #include     "unpipc.h"
2 int
3 main(int argc, char **argv)
4 {
5       printf("MQ_OPEN_MAX = %ld, MQ_PRIO_MAX = %ld\n",
6              Sysconf(_SC_MQ_OPEN_MAX), Sysconf(_SC_MQ_PRIO_MAX));
7       exit(0);
8 }
```

pxmsg/mqsysconf.c

图5-8　调用sysconf获取消息队列限制

在我们的两个系统上执行该程序的结果如下：

```
solaris % mqsysconf
MQ_OPEN_MAX = 32, MQ_PRIO_MAX = 32

alpha % mqsysconf
MQ_OPEN_MAX = 64, MQ_PRIO_MAX = 256
```

5.6　mq_notify函数

第6章中讨论的System V消息队列的问题之一是无法通知一个进程何时在某个队列中放置了一个消息。我们可以阻塞在msgrcv调用中，但那将阻止我们在等待期间做其他任何事。如果给msgrcv指定非阻塞标志（IPC_NOWAIT），那么尽管不阻塞了，但必须持续调用该函数以确定何时有一个消息到达。我们说过这称为轮询（polling），是对CPU时间的一种浪费。我们需要一种方法，让系统告诉我们何时有一个消息放置到了先前为空的某个队列中。

本节和本章的剩余各节含有高级主题，你第一次阅读时可暂时跳过去。

Posix消息队列允许异步事件通知（asynchronous event notification），以告知何时有一个消息放置到了某个空消息队列中。这种通知有两种方式可供选择：

- 产生一个信号；
- 创建一个线程来执行一个指定的函数。

这种通知通过调用mq_notify建立。

```
#include <mqueue.h>
int mq_notify(mqd_t mqdes, const struct sigevent *notification);
```

返回：若成功则为0，若出错则为-1

该函数为指定队列建立或删除异步事件通知。sigevent结构是随Posix.1实时信号新加的，后者将在下一节详细讨论。该结构以及本章中引入的所有新的信号相关常值都定义在<signal.h>头文件中。

87

```
union sigval {
  int       sival_int;              /* integer value */
  void     *sival_ptr;              /* pointer value */
};

struct sigevent {
  int              sigev_notify;    /* SIGEV_{NONE,SIGNAL,THREAD} */
  int              sigev_signo;     /* signal number if SIGEV_SIGNAL */
```

```
    union sigval     sigev_value;       /* passed to signal handler or thread */
                                        /* following two if SIGEV_THREAD */
    void             (*sigev_notify_function)(union sigval);
    pthread_attr_t   *sigev_notify_attributes;
};
```

我们马上给出以不同方法使用异步事件通知的几个例子，但在此前先给出一些普遍适用于该函数的若干规则。

(1) 如果*notification*参数非空，那么当前进程希望在有一个消息到达所指定的先前为空的队列时得到通知。我们说“该进程被注册为接收该队列的通知”。

(2) 如果*notification*参数为空指针，而且当前进程目前被注册为接收所指定队列的通知，那么已存在的注册将被撤销。

(3) 任意时刻只有一个进程可以被注册为接收某个给定队列的通知。

(4) 当有一个消息到达某个先前为空的队列，而且已有一个进程被注册为接收该队列的通知时，只有在没有任何线程阻塞在该队列的mq_receive调用中的前提下，通知才会发出。这就是说，在mq_reveive调用中的阻塞比任何通知的注册都优先。

(5) 当该通知被发送给它的注册进程时，其注册即被撤销。该进程必须再次调用mq_notify以重新注册（如果想要的话）。

> Unix信号最初的问题之一是：每当一个信号产生后，其行为就被复位成默认行为（APUE的10.4节）。信号处理程序调用的第一个函数通常是signal，用于重新建立处理程序。这么一来提供了一个短的时间窗口，它处于该信号的产生与当前进程重建其信号处理程序之间，这段时间内再次产生的同一信号可能终止当前进程。初看起来，mq_notify似乎有类似的问题，因为当前进程必须在每次通知发生后重新注册。然而消息队列不同于信号，因为在队列变空前通知不会再次发生。因此我们必须小心，保证在从队列中读出消息之前（而不是之后）重新注册。

5.6.1 例子：简单的信号通知

在深入探讨Posix实时信号和线程之前，我们可以编写一个简单的程序，当有一个消息放置到某个空队列中，该程序产生一个SIGUSR1信号。图5-9给出了这个程序，注意它含有一个我们不久后将详细讨论的错误。 88

声明全局变量

2~6 声明main函数和信号处理程序（sig_usr1）都使用的一些全局变量。

打开队列，取得属性，分配读缓冲区

12~15 打开通过命令行参数指定的消息队列，获取其属性，然后分配一个读缓冲区。

建立信号处理程序，启用通知

16~20 首先给SIGUSR1建立信号处理程序。在sigevent结构的sigev_notify成员中填入SIGEV_SIGNAL常值，其意思是当所指定队列由空变为非空时，我们希望有一个信号会产生。将sigev_signo成员设置成希望产生的信号后，调用mq_notify。

无限循环

21~22 main函数接下来是个无限睡眠在pause函数中的循环，该函数每次捕获一个信号时都会返回-1。

捕获信号，读出消息

25~33 我们的信号处理程序调用mq_notify以便为下一个事件重新注册，然后读出消息并输出其长度。本程序中我们忽略了接收到的消息的优先级。

pxmsg/mqnotifysig1.c

```
 1 #include        "unpipc.h"

 2 mqd_t   mqd;
 3 void   *buff;
 4 struct mq_attr  attr;
 5 struct sigevent sigev;

 6 static void sig_usr1(int);

 7 int
 8 main(int argc, char **argv)
 9 {
10     if (argc != 2)
11         err_quit("usage: mqnotifysig1 <name>");

12         /* open queue, get attributes, allocate read buffer */
13     mqd = Mq_open(argv[1], O_RDONLY);
14     Mq_getattr(mqd, &attr);
15     buff = Malloc(attr.mq_msgsize);

16         /* establish signal handler, enable notification */
17     Signal(SIGUSR1, sig_usr1);
18     sigev.sigev_notify = SIGEV_SIGNAL;
19     sigev.sigev_signo = SIGUSR1;
20     Mq_notify(mqd, &sigev);

21     for ( ; ; )
22         pause();                    /* signal handler does everything */
23     exit(0);
24 }

25 static void
26 sig_usr1(int signo)
27 {
28     ssize_t n;

29     Mq_notify(mqd, &sigev);         /* reregister first */
30     n = Mq_receive(mqd, buff, attr.mq_msgsize, NULL);
31     printf("SIGUSR1 received, read %ld bytes\n", (long) n);
32     return;
33 }
```

pxmsg/mqnotifysig1.c

图5-9　当有消息放置到某个空队列中时产生SIGUSR1（不正确版本）

sig_usr1结束处的return语句不是必要的，因为并没有返回值需返回，而且掉出该函数的末尾也是一个向调用者的隐式返回。然而作者总会在信号处理程序的末尾写上一个显式的return语句，目的是为了强调从这种函数的返回是特殊的。它可能会导致处理该信号的线程中某个函数调用过早返回（返回一个EINTR错误）。

我们现在从一个窗口运行这个程序：

```
solaris % mqcreate /test1                    创建队列
solaris % mqnotifysig1 /test1                启动图5-9中的程序
```

从另外一个窗口中执行如下命令：

```
solaris % mqsend /test1 50 16                发送50字节优先级为16的消息
```

正如所料，程序mynotifysig1输出："SIGUSR1 reveived, read 50 bytes"。

我们可以验证每次只有一个进程可被注册为接收通知，方法是从另一个窗口中启动该程序的另一个副本：

```
solaris % mqnotifysig1 /test1
mq_notify error: Device busy
```

这个出错消息对应EBUSY错误。

5.6.2 Posix 信号：异步信号安全函数

图5-9中程序的问题是它从信号处理程序中调用mq_notify、mq_receive和printf。这些函数实际上都不可以从信号处理程序中调用。

Posix使用异步信号安全（async-signal-safe）这一术语描述可以从信号处理程序中调用的函数。图5-10列出了这些Posix函数以及由Unix 98加上的其他几个函数。

access	fpathconf	rename	sysconf
aio_return	fstat	rmdir	tcdrain
aio_suspend	fsync	sem_post	tcflow
alarm	getegid	setgid	tcflush
cfgetispeed	geteuid	setpgid	tcgetattr
cfgetospeed	getgid	setsid	tcgetpgrp
cfsetispeed	getgroups	setuid	tcsendbreak
cfsetospeed	getpgrp	sigaction	tcsetattr
chdir	getpid	sigaddset	tcsetpgrp
chmod	getppid	sigdelset	time
chown	getuid	sigemptyset	timer_getoverrun
clock_gettime	kill	sigfillset	timer_gettime
close	link	sigismember	timer_settime
creat	lseek	signal	times
dup	mkdir	sigpause	umask
dup2	mkfifo	sigpending	uname
execle	open	sigprocmask	unlink
execve	pathconf	sigqueue	utime
_exit	pause	sigset	wait
fcntl	pipe	sigsuspend	waitpid
fdatasync	raise	sleep	write
fork	read	stat	

图5-10 异步信号安全的函数

没有列在该表中的函数不可以从信号处理程序中调用。注意所有标准I/O函数和pthread_*XXX*函数都没有列在其中。本书所涵盖的所有IPC函数中，只有sem_post、read和write列在其中（我们假定read和write可用于管道和FIFO）。

> ANSI C列出了可以从信号处理程序中调用的四个函数：abort、exit、longjmp和signal。Unix 98没有把前三个函数列为异步信号安全函数。

5.6.3 例子：信号通知

避免从信号处理程序中调用任何函数的方法之一是：让处理程序仅仅设置一个全局标志，由某个线程检查该标志以确定何时接收到一个消息。图5-11展示了这种技巧，不过它含有另外一个错误，我们不久会讲到。

全局变量

2 既然信号处理程序执行的唯一操作是把mqflag置为非零，于是图5-9中的全局变量不必仍然是全局变量。降低全局变量的数目肯定是一种好技巧，当使用线程时尤为如此。

pxmsg/mqnotifysig2.c

```
1 #include    "unpipc.h"
2 volatile sig_atomic_t   mqflag;    /* set nonzero by signal handler */
3 static void sig_usr1(int);
4 int
5 main(int argc, char **argv)
6 {
7     mqd_t   mqd;
8     void   *buff;
9     ssize_t n;
10    sigset_t    zeromask, newmask, oldmask;
11    struct mq_attr  attr;
12    struct sigevent sigev;
13    if (argc != 2)
14        err_quit("usage: mqnotifysig2 <name>");
15        /* open queue, get attributes, allocate read buffer */
16    mqd = Mq_open(argv[1], O_RDONLY);
17    Mq_getattr(mqd, &attr);
18    buff = Malloc(attr.mq_msgsize);
19    Sigemptyset(&zeromask);      /* no signals blocked */
20    Sigemptyset(&newmask);
21    Sigemptyset(&oldmask);
22    Sigaddset(&newmask, SIGUSR1);
23        /* establish signal handler, enable notification */
24    Signal(SIGUSR1, sig_usr1);
25    sigev.sigev_notify = SIGEV_SIGNAL;
26    sigev.sigev_signo = SIGUSR1;
27    Mq_notify(mqd, &sigev);
28    for ( ; ; ) {
29        Sigprocmask(SIG_BLOCK, &newmask, &oldmask);       /* block SIGUSR1 */
30        while (mqflag == 0)
31            sigsuspend(&zeromask);
32        mqflag = 0;                 /* reset flag */
33        Mq_notify(mqd, &sigev);    /* reregister first */
34        n = Mq_receive(mqd, buff, attr.mq_msgsize, NULL);
35        printf("read %ld bytes\n", (long) n);
36        Sigprocmask(SIG_UNBLOCK, &newmask, NULL);     /* unblock SIGUSR1 */
37    }
38    exit(0);
39 }
40 static void
41 sig_usr1(int signo)
42 {
43    mqflag = 1;
44    return;
45 }
```

pxmsg/mqnotifysig2.c

图5-11　信号处理程序只是给主线程设置一个标志（不正确版本）

打开消息队列

15~18　打开通过命令行参数指定的消息队列，获取其属性，然后分配一个读缓冲区。

初始化信号集

19~22　初始化三个信号集，并在newmask集中打开对应SIGUSR1的位。

建立信号处理程序，启用通知

23~27 给SIGUSR1建立一个信号处理程序，填写sigevent结构，调用mq_notify。

等待信号处理程序设置标志

28~32 调用sigprocmask阻塞SIGUSR1，并把当前信号掩码保存到oldmask中。随后在一个循环中测试全局变量mqflag，以等待信号处理程序将它设置成非零。只要它为0，我们就调用sigsuspend，它原子性地将调用线程投入睡眠，并把它的信号掩码复位成zeromask（没有一个信号被阻塞）。APUE的10.16节详细讨论sigsuspend以及为什么只能在SIGUSR1被阻塞时测试mqflag变量。每次sigsuspend返回时，SIGUSR1被重新阻塞。

重新注册并读出消息

33~36 当mqflag为非零时，重新注册并从指定队列中读出消息。随后给SIGUSR1解阻塞并返回for循环顶部。

我们提到过这种办法仍存在一个问题。考虑一下第一个消息被读出之前有两个消息到达的情形。我们可通过在mq_notify调用前增加一个sleep语句来模拟这种情形。这里的基本问题是：通知只是在有一个消息被放置到某个空队列上时才发出。如果在能够读出第一个消息前有两个消息到达，那么只有一个通知被发出：我们于是读出第一个消息，并调用sigsuspend等待另一个消息，而对应它的通知可能永远不会发出。在此期间，另一个消息已放置于该队列中等待读出，而我们却一直在忽略它。

5.6.4 例子：使用非阻塞 mq_receive 的信号通知

上述问题的解决办法是：当使用mq_notify产生信号时，总是以非阻塞模式读消息队列。图5-12给出了图5-11的一个修改版本，它以非阻塞模式读消息队列。

pxmsg/mqnotifysig3.c

```
 1 #include    "unpipc.h"
 2 volatile sig_atomic_t   mqflag;   /* set nonzero by signal handler */
 3 static void sig_usr1(int);
 4 int
 5 main(int argc, char **argv)
 6 {
 7     mqd_t   mqd;
 8     void   *buff;
 9     ssize_t n;
10     sigset_t zeromask, newmask, oldmask;
11     struct mq_attr  attr;
12     struct sigevent sigev;
13     if (argc != 2)
14         err_quit("usage: mqnotifysig3 <name>");
15         /* open queue, get attributes, allocate read buffer */
16     mqd = Mq_open(argv[1], O_RDONLY | O_NONBLOCK);
17     Mq_getattr(mqd, &attr);
18     buff = Malloc(attr.mq_msgsize);
19     Sigemptyset(&zeromask);     /* no signals blocked */
20     Sigemptyset(&newmask);
21     Sigemptyset(&oldmask);
22     Sigaddset(&newmask, SIGUSR1);
23         /* establish signal handler, enable notification */
24     Signal(SIGUSR1, sig_usr1);
```

图5-12 使用信号通知读Posix消息队列

```
25      sigev.sigev_notify = SIGEV_SIGNAL;
26      sigev.sigev_signo = SIGUSR1;
27      Mq_notify(mqd, &sigev);

28      for ( ; ; ) {
29          Sigprocmask(SIG_BLOCK, &newmask, &oldmask);    /* block SIGUSR1 */
30          while (mqflag == 0)
31              sigsuspend(&zeromask);
32          mqflag = 0;                /* reset flag */

33          Mq_notify(mqd, &sigev);    /* reregister first */
34          while ( (n = mq_receive(mqd, buff, attr.mq_msgsize, NULL)) >= 0) {
35              printf("read %ld bytes\n", (long) n);
36          }
37          if (errno != EAGAIN)
38              err_sys("mq_receive error");
39          Sigprocmask(SIG_UNBLOCK, &newmask, NULL);    /* unblock SIGUSR1 */
40      }
41      exit(0);
42  }

43  static void
44  sig_usr1(int signo)
45  {
46      mqflag = 1;
47      return;
48  }
```

pxmsg/mqnotifysig3.c

图5-12（续）

打开消息队列以非阻塞模式

15~18　第一个变动是在打开消息队列时指定O_NONBLOCK标志。

从队列中读出所有消息

34~38　另一个变动是在一个循环中调用mq_receive，处理队列中的每个消息。返回一个EAGAIN错误不表示有问题，它只是意味着暂时没有消息可读。

5.6.5　例子：使用 sigwait 代替信号处理程序的信号通知

上一个例子尽管正确，但效率还可以更高些。我们的程序通过调用sigsuspend阻塞，以等待某个消息的到达。当有一个消息被放置到某个空队列中时，该信号产生，主线程被阻止，信号处理程序执行并设置mqflag变量，主线程再次执行，发现mq_flag为非零，于是读出该消息。更为简易（并且可能更为高效）的办法之一是阻塞在某个函数中，仅仅等待该信号的递交，而不是让内核执行一个只为设置一个标志的信号处理程序。sigwait提供了这种能力。

93
~
94

```
#include <signal.h>

int sigwait(const sigset_t *set, int *sig);
```

返回：若成功则为0，若出错则为正的E*xxx*值

调用sigwait前，我们阻塞某个信号集。我们将这个信号集指定为*set*参数。sigwait然后一直阻塞到这些信号中有一个或多个待处理，这时它返回其中一个信号。该信号值通过指针sig存放，函数的返回值则为0。这个过程称为“同步地等待一个异步事件”：我们是在使用信号，但没有涉及异步信号处理程序。

图5-13给出了用到sigwait时mq_notify的使用。

pxmsg/mqnotifysig4.c

```
 1 #include    "unpipc.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     signo;
 6     mqd_t   mqd;
 7     void   *buff;
 8     ssize_t n;
 9     sigset_t    newmask;
10     struct mq_attr  attr;
11     struct sigevent sigev;
12     if (argc != 2)
13         err_quit("usage: mqnotifysig4 <name>");
14         /* open queue, get attributes, allocate read buffer */
15     mqd = Mq_open(argv[1], O_RDONLY | O_NONBLOCK);
16     Mq_getattr(mqd, &attr);
17     buff = Malloc(attr.mq_msgsize);
18     Sigemptyset(&newmask);
19     Sigaddset(&newmask, SIGUSR1);
20     Sigprocmask(SIG_BLOCK, &newmask, NULL);      /* block SIGUSR1 */
21         /* establish signal handler, enable notification */
22     sigev.sigev_notify = SIGEV_SIGNAL;
23     sigev.sigev_signo = SIGUSR1;
24     Mq_notify(mqd, &sigev);
25     for ( ; ; ) {
26         Sigwait(&newmask, &signo);
27         if (signo == SIGUSR1) {
28             Mq_notify(mqd, &sigev);    /* reregister first */
29             while ( (n = mq_receive(mqd, buff, attr.mq_msgsize, NULL)) >= 0){
30                 printf("read %ld bytes\n", (long) n);
31             }
32             if (errno != EAGAIN)
33                 err_sys("mq_receive error");
34         }
35     }
36     exit(0);
37 }
```

pxmsg/mqnotifysig4.c

图5-13 伴随sigwait使用mq_notify

初始化信号集并阻塞SIGUSR1

18~20 把某个信号集初始化成只含有SIGUSR1，然后用sigprocmask阻塞该信号。

等待信号

26~34 在sigwait调用中阻塞并等待该信号。SIGUSR1被递交后，重新注册通知并读出所有可用消息。

> sigwait往往在多线程化的进程中使用。实际上，看一看它的函数原型，我们会发现其返回值或为0，或为某个Exxx错误，这与大多数Pthread函数一样。但是在多线程化的进程中不能使用sigprocmask，而必须调用pthread_sigmask，它只是改变调用线程的信号掩码。pthread_sigmask的参数与sigprocmask的相同。

sigwait存在两个变种：sigwaitinfo和sigtimedwait。sigwaitinfo还返回一个siginfo_t结构（将在下一节中定义），目的是用于可靠信号中。sigtimedwait也返回一个siginfo_t结构，并允许调用者指定一个时间限制。

大多数讨论线程的书（例如[Butenhof 1997]）推荐在多线程化的进程中使用sigwait来处理所有信号，而绝不要使用异步信号处理程序。

5.6.6　例子：使用 select 的 Posix 消息队列

消息队列描述符（mqd_t变量）不是“普通”描述符，它不能用在select或poll（UNPv1第6章）中。然而我们可以伴随一个管道和mq_notify函数使用它们。（我们将在6.9节中随System V消息队列展示类似的技巧，那时涉及的是一个子进程和一个管道。）首先，从图5-10注意到write函数是异步信号安全的，因此可以从信号处理程序中调用它。图5-14给出了我们的程序。

pxmsg/mqnotifysig5.c

```
1 #include    "unpipc.h"

2 int     pipefd[2];
3 static void sig_usr1(int);

4 int
5 main(int argc, char **argv)
6 {
7     int     nfds;
8     char    c;
9     fd_set  rset;
10    mqd_t   mqd;
11    void   *buff;
12    ssize_t n;
13    struct mq_attr  attr;
14    struct sigevent sigev;

15    if (argc != 2)
16        err_quit("usage: mqnotifysig5 <name>");

17        /* open queue, get attributes, allocate read buffer */
18    mqd = Mq_open(argv[1], O_RDONLY | O_NONBLOCK);
19    Mq_getattr(mqd, &attr);
20    buff = Malloc(attr.mq_msgsize);

21    Pipe(pipefd);

22        /* establish signal handler, enable notification */
23    Signal(SIGUSR1, sig_usr1);
24    sigev.sigev_notify = SIGEV_SIGNAL;
25    sigev.sigev_signo = SIGUSR1;
26    Mq_notify(mqd, &sigev);

27    FD_ZERO(&rset);
28    for ( ; ; ) {
29        FD_SET(pipefd[0], &rset);
30        nfds = Select(pipefd[0] + 1, &rset, NULL, NULL, NULL);

31        if (FD_ISSET(pipefd[0], &rset)) {
32            Read(pipefd[0], &c, 1);
```

图5-14　伴随管道使用信号通知

```
33              Mq_notify(mqd, &sigev);       /* reregister first */
34              while ( (n = mq_receive(mqd, buff, attr.mq_msgsize, NULL)) >= 0){
35                  printf("read %ld bytes\n", (long) n);
36              }
37              if (errno != EAGAIN)
38                  err_sys("mq_receive error");
39          }
40      }
41      exit(0);
42  }

43  static void
44  sig_usr1(int signo)
45  {
46      Write(pipefd[1], "", 1);      /* one byte of 0 */
47      return;
48  }
```

pxmsg/mqnotifysig5.c

95 ~ 97

图5-14（续）

创建一个管道

21 创建一个管道，当接收到消息队列的异步事件通知时，信号处理程序就会往该管道中写入数据。这是一个管道用于信号处理程序中的例子。

调用select

27~40 初始化描述符集rset，每次循环时打开对应于pipefd[0]（管道的读出端）的那一位。然后调用select只等待该描述符，不过在典型的应用中，这儿是多个描述符上的输入或输出复用的地方。当管道的读出端可读时，重新注册消息队列的通知，并读出所有可得的消息。

信号处理程序

43~48 我们的信号处理程序只是往管道write 1个字节。我们已提及这是一个异步信号安全的操作。

5.6.7 例子：启动线程

异步事件通知的另一种方式是把sigev_notify设置成SIGEV_THREAD，这会创建一个新的线程。该线程调用由sigev_notify_function指定的函数，所用的参数由sigev_value指定。新线程的线程属性由sigev_notify_attributes指定，要是默认属性合适的话，它可以是一个空指针。图5-15给出了使用这种技术的一个例子。

我们把给新线程的参数（sigev_value）指定成一个空指针，因此不会有任何东西传递给该线程的起始函数。我们能以参数的形式传递一个指向所处理消息队列描述符的指针，而不是把它声明为一个全局变量，不过新线程仍然需要消息队列属性和sigev结构（以便重新注册）。我们把给新线程的属性指定成一个空指针，因此使用的是系统默认属性。这样的新线程是作为脱离的线程创建的。

> 遗憾的是，本书例子所用的两个系统（Solaris 2.6和Digital Unix 4.0B）没有一个支持SIGEV_THREAD。这两个系统都要求sigev_notify或者为SIGEV_NONE，或者为SIGEV_SIGNAL。

pxmsg/mqnotifythread1.c

```
1 #include    "unpipc.h"

2 mqd_t  mqd;
3 struct mq_attr  attr;
4 struct sigevent sigev;

5 static void notify_thread(union sigval);    /* our thread function */

6 int
7 main(int argc, char **argv)
8 {
9     if (argc != 2)
10        err_quit("usage: mqnotifythread1 <name>");

11    mqd = Mq_open(argv[1], O_RDONLY | O_NONBLOCK);
12    Mq_getattr(mqd, &attr);

13    sigev.sigev_notify = SIGEV_THREAD;
14    sigev.sigev_value.sival_ptr = NULL;
15    sigev.sigev_notify_function = notify_thread;
16    sigev.sigev_notify_attributes = NULL;
17    Mq_notify(mqd, &sigev);

18    for ( ; ; )
19        pause();              /* each new thread does everything */

20    exit(0);
21 }

22 static void
23 notify_thread(union sigval arg)
24 {
25    ssize_t n;
26    void    *buff;

27    printf("notify_thread started\n");
28    buff = Malloc(attr.mq_msgsize);
29    Mq_notify(mqd, &sigev);    /* reregister */

30    while ( (n = mq_receive(mqd, buff, attr.mq_msgsize, NULL)) >= 0) {
31        printf("read %ld bytes\n", (long) n);
32    }
33    if (errno != EAGAIN)
34        err_sys("mq_receive error");

35    free(buff);
36    pthread_exit(NULL);
37 }
```

pxmsg/mqnotifythread1.c

图5-15　启动一个新线程的mq_notify

5.7　Posix实时信号

在过去几十年中，Unix信号经历了多次重大的演变。

(1) 由Version 7 Unix（1978年）提供的信号模型是不可靠的。信号可能丢失，而且进程难以在执行临界代码段时关掉选中的若干信号。

(2) 4.3BSD（1986年）增加了可靠的信号。

(3) System V Release 3.0（1986年）也增加了可靠的信号，不过不同于BSD模型。

(4) Posix.1（1990年）标准化了BSD可靠信号模型，APUE的第10章详细讲述了该模型。

(5) Posix.1（1996年）给Posix模型增加了实时信号。该工作起源于Posix.1b实时扩展（以前称为Posix.4）。

当今几乎每种Unix系统都提供Posix可靠信号，更新的系统正在逐步提供Posix实时信号。（在描述信号时，注意区分可靠和实时。）我们有必要较详细地讨论实时信号，因为已在上一节中使用过由这个扩展定义的一些结构（sigval结构和sigevent结构）。

信号可划分为两个大组。

(1)其值在SIGRTMIN和SIGRTMAX之间（包括两者在内）的实时信号。Posix要求至少提供RTSIG_MAX种实时信号，而该常值的最小值为8。

(2)所有其他信号：SIGALRM、SIGINT、SIGKILL，等等。

> Solaris 2.6上普通Unix信号的值为1~37，8种实时信号的值为38~45。Digital Unix 4.0B上普通Unix信号的值为1~32，16种实时信号的值为33~48。这两种实现都把SIGRTMIN和SIGRTMAX定义为调用sysconf的宏，以允许在将来修改它们的值。

接下去我们关注接收某个信号的进程的sigaction调用中是否指定了新的SA_SIGINFO标志。这些差异带来了图5-16中所示的四种可能情形。

信　号	sigaction调用	
	SA_SIGINFO已指定	SA_SIGINFO未指定
SIGRTMIN到SIGRTMAX	实时行为有保证	实时行为未指定
所有其他信号	实时行为未指定	实时行为未指定

图5-16 Posix信号实时行为，取决于SA_SIGINFO

其中有三个框标以“实时行为未指定”，其含义是有些实现可能提供实时行为，有些实现可能不提供。如果需要实时行为，我们必须使用SIGRTMIN和SIGRTMAX之间的新的实时信号，而且在安装信号处理程序时必须给sigaction指定SA_SIGINFO标志。

术语实时行为（realtime behavior）隐含着如下特征。

- 信号是排队的。这就是说，如果同一信号产生了三次，它就递交三次。另外，一种给定信号的多次发生以先进先出（FIFO）顺序排队。我们不久将给出一个信号排队的例子。对于不排队的信号来说，产生了三次的某种信号可能只递交一次。
- 当有多个SIGRTMIN到SIGRTMAX范围内的解阻塞信号排队时，值较小的信号先于值较大的信号递交。这就是说，SIGRTMIN比值为SIGRTMIN+1的信号“更为优先”，值为SIGRTMIN+1的信号比值为SIGRTMIN+2的信号“更为优先”，依此类推。
- 当某个非实时信号递交时，传递给它的信号处理程序的唯一参数是该信号的值。实时信号比其他信号携带更多的信息。通过设置SA_SIGINFO标志安装的任意实时信号的信号处理程序声明如下：

```
void func(int signo, siginfo_t *info, void *context);
```

其中*signo*是该信号的值，siginfo_t结构则定义如下：

```
typedef struct {
  int           si_signo;     /* same value as signo argument */
  int           si_code;      /* SI_{USER,QUEUE,TIMER,ASYNCIO,MEGEQ} */
  union sigval  si_value;     /* integer or pointer value from sender */
} siginfo_t;
```

*context*参数所指向的内容依赖于实现。

技术上讲，非实时Posix信号的处理程序只用一个参数调用。许多系统有一个较早的使用三个参数的约定，适用于先于Posix实时标准的信号处理程序。

siginfo_t是使用typedef定义的具有以_t结尾的名字的唯一一个Posix结构。图5-17中我们把指向这些结构的指针声明为siginfo_t*，而不出现struct一词[①]。

- 一些新函数定义成使用实时信号工作。例如，sigqueue函数用于代替kill函数向某个进程发送一个信号，该新函数允许发送者随所发送信号传递一个sigval联合。

实时信号由下列Posix.1特性产生，它们由包含在传递给信号处理程序的siginfo_t结构中的si_code值来标识。

SI_ASYNCIO	信号由某个异步I/O请求的完成产生，这些异步I/O请求就是Posix的aio_*XXX*函数，我们不讲述。
SI_MESGQ	信号在有一个消息被放置到某个空消息队列中时产生，如5.6节中所述。
SI_QUEUE	信号由sigqueue函数发出。稍后我们将给出一个这样的例子。
SI_TIMER	信号由使用timer_settime函数设置的某个定时器的到时产生，我们不讲述。
SI_USER	信号由kill函数发出。

如果信号是由某个其他事件产生的，si_code就会被设置成不同于这里所列的某个值。siginfo_t结构的si_value成员的内容只在si_code为SI_ASYNCIO、SI_MESGQ、SI_QUEUE或
101 SI_TIMER时才有效。

5.7.1 例子

图5-17是一个演示实时信号的简单程序。该程序调用fork，子进程阻塞三种实时信号，父进程随后发送9个信号（三种实时信号中每种3个），子进程接着解阻塞信号，我们于是看到每种信号各有多少个递交以及它们的先后递交顺序。

输出实时信号值

10　输出最小和最大实时信号值，以查看系统实现支持多少种实时信号。我们把这两个常值类型强制转换成一个整数，因为有些实现把这两个常值定义为调用sysconf的宏，例如：

```
#define    SIGRTMAX    (sysconf(_SC_RTSIG_MAX))
```

而sysconf返回一个长整数（见习题5.4）。

fork：子进程阻塞三种实时信号

11~17　派生一个子进程，由子进程调用sigprocmask阻塞我们将使用的三种实时信号：*SIGRTMAX*、*SIGRTMAX-1*、*SIGRTMAX-2*。

① 在由Posix.1定义的所有结构中（aiocb、group、itimerspec、lock、mq_attr、passwd、sched_param、sigaction、sigevent、siginfo_t、sival、stat、termio、timespec、utimbuf），只有siginfo_t是使用typedef定义的。Posix.1给它的所有简单系统数据类型（pid_t、pthread_t，等等）的名字添上_t后缀，但它们没有一个必须是结构，Posix.1也没有给这些数据类型的任何一个定义结构成员名。因此把一个必须是结构的数据类型定义为加_t后缀的siginfo_t导致作者认为是一个错误。这个古怪现象的原因也许是因为siginfo_t是由SVR4定义的，后来Posix一直忽略它，直到发展到P1003.4实时扩展为止。事实上1003.1b-1993第354页谈到该名字破坏了约定，所列的原因是为了遵循已有的实践，从而提高可移植性。

rtsignals/test1.c

```
1 #include    "unpipc.h"
2 static void sig_rt(int, siginfo_t *, void *);
3 int
4 main(int argc, char **argv)
5 {
6     int     i, j;
7     pid_t   pid;
8     sigset_t newset;
9     union sigval val;
10    printf("SIGRTMIN = %d, SIGRTMAX = %d\n", (int) SIGRTMIN, (int) SIGRTMAX);
11    if ( (pid = Fork()) == 0) {
12            /* child: block three realtime signals */
13        Sigemptyset(&newset);
14        Sigaddset(&newset, SIGRTMAX);
15        Sigaddset(&newset, SIGRTMAX - 1);
16        Sigaddset(&newset, SIGRTMAX - 2);
17        Sigprocmask(SIG_BLOCK, &newset, NULL);
18            /* establish signal handler with SA_SIGINFO set */
19        Signal_rt(SIGRTMAX, sig_rt, &newset);
20        Signal_rt(SIGRTMAX - 1, sig_rt, &newset);
21        Signal_rt(SIGRTMAX - 2, sig_rt, &newset);
22        sleep(6);                   /* let parent send all the signals */
23        Sigprocmask(SIG_UNBLOCK, &newset, NULL); /* unblock */
24        sleep(3);                   /* let all queued signals be delivered */
25        exit(0);
26    }
27        /* parent sends nine signals to child */
28    sleep(3);                       /* let child block all signals */
29    for (i = SIGRTMAX; i >= SIGRTMAX - 2; i--) {
30        for (j = 0; j <= 2; j++) {
31            val.sival_int = j;
32            Sigqueue(pid, i, val);
33            printf("sent signal %d, val = %d\n", i, j);
34        }
35    }
36    exit(0);
37 }
38 static void
39 sig_rt(int signo, siginfo_t *info, void *context)
40 {
41    printf("received signal #%d, code = %d, ival = %d\n",
42           signo, info->si_code, info->si_value.sival_int);
43 }
```

rtsignals/test1.c

图5-17 演示实时信号的简单测试程序

建立信号处理程序

18~21 调用signal_rt函数（将在图5-18中给出），建立我们的sig_rt函数来作为这三种实时信号的处理程序。该函数设置SA_SIGINFO标志，再加上这三种信号都是实时信号，于是我们预期它们具备实时行为。该函数还设置执行信号处理程序期间需阻塞的信

号掩码[1]。

等待父进程产生信号，然后解阻塞信号

22~25　等待6秒钟以允许父进程产生预定的9个信号。然后调用sigprocmask解阻塞那三种实时信号。该操作应允许所有排队的信号都被递交。子进程停顿另外3秒，以便信号处理程序调用printf 9次，然后终止。

父进程发送9个信号

27~36　父进程停顿3秒，以便子进程阻塞所有信号。父进程随后给那三种实时信号的第一种产生3个信号：i取这三种实时信号的值，对于每个i值，j取值为0、1和2。我们特意从最高的信号值开始产生信号，因为期待它们从最低的信号值开始递交。我们还伴随每个信号发送一个不同的整数值（sival_int），以验证一种给定信号的3次发生是按FIFO的顺序产生的。

信号处理程序

38~43　我们的信号处理程序只是输出所递交信号的有关信息。

102
↓
103

我们从图5-10注意到printf不是异步信号安全的，因而不能从信号处理程序中调用。在这儿的小测试程序中，我们将它作为一个简单的诊断工具来调用。[2]

我们首先在Solaris 2.6下运行该程序，但是发现它的输出与我们预期的不一致。

```
solaris % test1
SIGRTMIN = 38, SIGRTMAX = 45          提供8种实时信号

                                      这儿停顿3秒
sent signal 45, val = 0               父进程现在发送9个信号
sent signal 45, val = 1
sent signal 45, val = 2
sent signal 44, val = 0
```

① 无论何时处理不止一种实时信号，我们都必须给每种实时信号的信号处理函数指定一个sa_mask值，该掩码应该阻塞所有剩余的值较大的（即优先级较低的）实时信号。Posix规则保证当有多种实时信号待处理时，值最小的信号最先递交，然而保证较低优先级的实时信号不中断当前信号处理函数却是我们的责任。我们通过给信号处理函数指定一个sa_mask值来做到这一点。

② 既然本例子中所有三种实时信号都从信号处理函数中相互阻塞了，也就是说在执行某个信号的处理函数期间，其他信号不会递交，因而不会中断当前信号处理函数的执行，这种情况下把printf作为一个简单的诊断工具来调用也许可行。然而输出各个信号递交顺序的更好技巧之一是按下述修改图5-17。首先分配两个全局变量：

```
static volatile siginfo_t arrival[10];
static volatile int       nsig;
```

然后让信号处理程序在该数组中保存其info参数：

```
static void
sig_rt(int signo, siginfo_t *info, void *context)
{
    arrival[nsig++] = *info; /* save info for child to print */
}
```

最后由子进程在终止前输出该数组中的信息：

```
sleep(3);     /* let all queued signals be delivered */
for (i = 0; i < nsig; i++) {
    printf("received signal #%d, code = %d, ival = %d\n",
           arrival[i].si_signo, arrival[i].si_code,
           arrival[i].si_value.sival_int);
}
exit(0);
```

```
sent signal 44, val = 1
sent signal 44, val = 2
sent signal 43, val = 0
sent signal 43, val = 1
sent signal 43, val = 2
solaris %                                          父进程终止，shell提示符输出
                                                   在子进程解阻塞信号前停顿3秒
received signal #45, code = -2, ival = 2           子进程捕获信号
received signal #45, code = -2, ival = 1
received signal #45, code = -2, ival = 0
received signal #44, code = -2, ival = 2
received signal #44, code = -2, ival = 1
received signal #44, code = -2, ival = 0
received signal #43, code = -2, ival = 2
received signal #43, code = -2, ival = 1
received signal #43, code = -2, ival = 0
```

父进程发送的9个信号排了队，但是那三种信号是从值最大的信号开始产生的（而我们却期待值最小的信号最先产生）。对于一种给定信号，它的3个排了队的信号看来是以LIFO顺序而不是FIFO顺序递交的。值为–2的si_code对应SI_QUEUE。

接着改在Digital Unix 4.0B下运行该程序，我们看到了预期的结果。

```
alpha % test1
SIGRTMIN = 33, SIGRTMAX = 48                       提供16种实时信号
                                                   这儿停顿3秒
sent signal 48, val = 0                            父进程现在发送9个信号
sent signal 48, val = 1
sent signal 48, val = 2
sent signal 47, val = 0
sent signal 47, val = 1
sent signal 47, val = 2
sent signal 46, val = 0
sent signal 46, val = 1
sent signal 46, val = 2
alpha %                                            父进程终止，shell提示符输出
                                                   在子进程解阻塞信号前停顿3秒
received signal #46, code = -1, ival = 0           子进程捕获信号
received signal #46, code = -1, ival = 1
received signal #46, code = -1, ival = 2
received signal #47, code = -1, ival = 0
received signal #47, code = -1, ival = 1
received signal #47, code = -1, ival = 2
received signal #48, code = -1, ival = 0
received signal #48, code = -1, ival = 1
received signal #48, code = -1, ival = 2
```

父进程发送的9个信号排队后按我们期待的顺序递交：值最小的信号最先递交，对于一种给定信号，它的3次发生按FIFO顺序递交。 104

Solaris 2.6的实现看来存在缺陷。

5.7.2 signal_rt 函数

我们在UNPv1第120页给出了自己的signal函数，它调用Posix sigaction函数建立一个提供可靠Posix语义的信号处理程序。现在我们把该函数修改成提供实时行为。我们称这个新函数为signal_rt，如图5-18所示。

lib/signal_rt.c

```
 1 #include    "unpipc.h"
 2 Sigfunc_rt *
 3 signal_rt(int signo, Sigfunc_rt *func, sigset_t *mask)
 4 {
 5     struct sigaction act, oact;
 6     act.sa_sigaction = func;       /* must store function addr here */
 7     act.sa_mask = *mask;           /* signals to block */
 8     act.sa_flags = SA_SIGINFO;     /* must specify this for realtime */
 9     if (signo == SIGALRM) {
10 #ifdef  SA_INTERRUPT
11         act.sa_flags |= SA_INTERRUPT;       /* SunOS 4.x */
12 #endif
13     } else {
14 #ifdef  SA_RESTART
15         act.sa_flags |= SA_RESTART;         /* SVR4, 44BSD */
16 #endif
17     }
18     if (sigaction(signo, &act, &oact) < 0)
19         return((Sigfunc_rt *) SIG_ERR);
20     return(oact.sa_sigaction);
21 }
```

lib/signal_rt.c

图5-18　提供实时行为的signal_rt函数

使用typedef简化函数原型

1~3　在我们的unpipc.h头文件（图C-1）中，Sigfunc_rt定义如下：

```
typedef  void  Sigfunc_rt(int, siginfo_t *, void *);
```

我们在本节早先说过，这是在设置SA_SIGINFO标志的前提下安装的信号处理程序的函数原型。

指定处理程序函数

105

5~7　加入实时信号支持后，sigaction发生变化，即增加了新的sa_sigaction成员。

```
struct sigaction {
  void      (*sa_handler)();/* SIG_DFL,SIG_IGN,or add of signal handler */
  sigset_t  sa_mask;        /* additional signals to block */
  int       sa_flags;       /* signal options: SA_xxx */
  void      (*sa_sigaction)(int, siginfo_t, void *);
                            /* addr of signal handler if SA_SIGINFO set */
};
```

规则如下。

- 如果在sa_flags成员中设置了SA_SIGINFO标志，那么sa_sigaction成员会指定信号处理函数的地址。
- 如果在sa_flags成员中没有设置SA_SIGINFO标志，那么sa_handler成员会指定信号处理函数的地址。
- 为给某个信号指定默认行为或忽略该信号，应把sa_handler设置为SIG_DFL或SIG_IGN，并且不设置SA_SIGINFO标志。

设置SA_SIGINFO

8~17　我们总是设置SA_SIGINFO标志，如果信号不是SIGALRM，那就再指定SA_RESTART标志。

5.8 使用内存映射 I/O 实现 Posix 消息队列

我们现在提供一个使用内存映射I/O以及Posix互斥锁和条件变量完成的Posix消息队列的实现。

> 我们在第7章中讨论互斥锁和条件变量，在第12章和第13章中讨论内存映射I/O。你可能希望跳过本节，阅读过所列各章后再返回来。

图5-19展示了我们用于实现Posix消息队列的各种数据结构的布局。该图中我们假设创建出的消息队列最多容纳4个消息，每个消息7个字节。

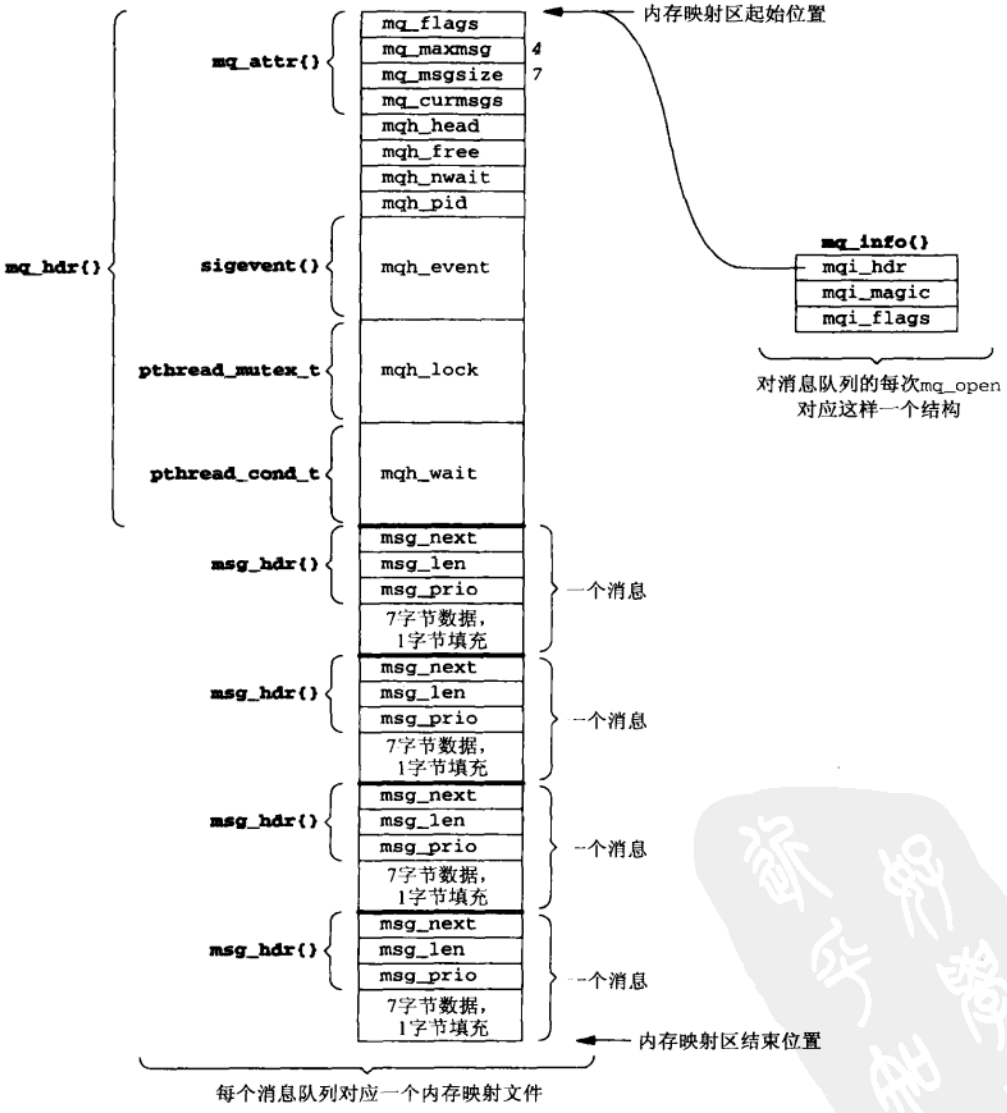


图5-19 使用内存映射文件实现Posix消息队列的各种数据结构的布局

图5-20给出了我们的mqueue.h头文件，它定义了本实现的基本结构。

my_pxmsg_mmap/mqueue.h

```
1 typedef struct mq_info *mqd_t;   /* opaque datatype */

2 struct mq_attr {
3     long    mq_flags;           /* message queue flag: O_NONBLOCK */
4     long    mq_maxmsg;          /* max number of messages allowed on queue */
5     long    mq_msgsize;         /* max size of a message (in bytes) */
6     long    mq_curmsgs;         /* number of messages currently on queue */
7 };

8         /* one mq_hdr{} per queue, at beginning of mapped file */
9 struct mq_hdr {
10    struct mq_attr  mqh_attr;   /* the queue's attributes */
11    long    mqh_head;           /* index of first message */
12    long    mqh_free;           /* index of first free message */
13    long    mqh_nwait;          /* #threads blocked in mq_receive() */
14    pid_t   mqh_pid;            /* nonzero PID if mqh_event set */
15    struct sigevent mqh_event;  /* for mq_notify() */
16    pthread_mutex_t mqh_lock;   /* mutex lock */
17    pthread_cond_t mqh_wait;    /* and condition variable */
18 };

19        /* one msg_hdr{} at the front of each message in the mapped file */
20 struct msg_hdr {
21    long    msg_next;           /* index of next on linked list */
22        /* msg_next must be first member in struct */
23    ssize_t msg_len;            /* actual length */
24    unsigned int    msg_prio;   /* priority */
25 };

26        /* one mq_info{} malloc'ed per process per mq_open() */
27 struct mq_info {
28    struct mq_hdr *mqi_hdr;     /* start of mmap'ed region */
29    long    mqi_magic;          /* magic number if open */
30    int     mqi_flags;          /* flags for this process */
31 };
32 #define MQI_MAGIC   0x98765432

33        /* size of message in file is rounded up for alignment */
34 #define MSGSIZE(i)  ((((i) + sizeof(long)-1) / sizeof(long)) * sizeof(long))
```

my_pxmsg_mmap/mqueue.h

图5-20　mqueue.h头文件

mqd_t数据类型

1　我们的消息队列描述符只是一个指向某个mq_info结构的指针。每次调用mq_open都会分配一个这种结构，其指针就返回给调用者。这一点再次强调了消息队列描述符不必像文件描述符那样是一个小整数——唯一的Posix要求是这种数据类型不能是一个数组类型。

mq_hdr结构

8~18　该结构出现在映射文件的开头，含有针对每个队列的所有信息。mq_attr结构的mq_flags成员没有用上，因为标志（唯一定义了的是非阻塞标志）必须以每次打开为基而不是以每个队列为基来维护。也就是说，标志在mq_info结构中维护。该结构的其余成员随它们在各种函数中的使用而说明。

现在开始注意，我们称之为索引（index）的任何东西（本结构的mqh_head和mqh_free成员，下一个结构的msg_next成员）都含有从映射文件头开始的字节索引。举例来说，

106~108

Solaris 2.6下mq_hdr结构的大小为96字节，因此该首部之后第一个消息的索引为96。图5-19中的每个消息占据20字节（12字节的msg_hdr结构和8字节的消息数据），因此其余三个消息的索引分别为116、136和156，该映射文件的大小为176字节。这些索引用于维护映射文件中的两个链表：一个链表（mqh_head）含有当前在队列中的所有消息，另一个链表（mqh_free）含有队列中的所有空闲消息。我们不能给这些链表指针使用真正的内存指针（地址），因为同一映射文件在映射它的各个进程中可以在不同的内存地址开始（如图13-6中所示）。

msg_hdr结构

19~25 该结构出现在映射文件中每个消息的开头。所有消息要么在消息链表中，要么在空闲链表中，msg_next成员含有本链表中下一个消息的索引（如果本消息为所在链表最后一个消息，那么下一个消息的索引为0）。msg_len是消息数据的真正长度，对于图5-19中的例子来说，它可以在0~7字节之间（包括0和7）。msg_prio是由mq_send的调用者赋予消息的优先级。

mq_info结构

26~32 每次打开一个队列时，mq_open会动态分配一个这种结构，它由mq_close释放。mqi_hdr指向映射文件（由mmap返回的起始地址）。我们的实现中的基本数据类型mqd_t就是指向该结构的指针，该指针是mq_open的返回值。

一旦mq_info结构初始化，其mqi_magic成员便含有MQI_MAGIC，被传递以一个mqd_t指针的每个函数都检查该成员，以确信该指针确实指向一个mq_info结构。mqi_flags成员含有当前队列的本次打开实例的非阻塞标志。

MSGSIZE宏

33~34 为了对齐，我们希望映射文件中的每个消息从一个长整数边界开始。因此，如果每个消息的最大大小不是这样对齐的，我们就得给每个消息的数据部分增加1~3个填充字节，如图5-19中所示。这里假设长整数的大小为4字节（对于Solaris 2.6来说是正确的），但是如果长整数的大小为8字节（Digit Unix 4.0上就是这样），那么填充字节数在1~7之间。

5.8.1 mq_open函数

图5-21给出了mq_open函数的第一部分，它创建一个新消息队列或打开一个已存在的消息队列。

处理可变长度参数表

29~32 本函数能够以两个或四个参数调用，这取决于是否指定了O_CREAT标志。当指定了该标志时，第三个参数的类型为mode_t，它是一个基本的系统数据类型，可以是任意类型的整数。我们遇到的问题是在BSD/OS上，它把该数据类型定义为unsigned short整数（占据16位）。既然该实现上的整数要占据32位，而且参数表中的所有短整数都被扩展成整数，那么C编译器会把这种类型的参数从16位扩展到32位。但是如果在va_arg调用中指定mode_t，那它将会在栈中走过16位后便指向下一个参数，然而本参数已被扩展为占据32位。为此我们必须定义自己的数据类型va_mode_t，它在BSD/OS下是整数，在其他系统下是类型mode_t。我们的unpipc.h头文件（图C-1）中的如下各行处理这个移植性问题：

```
#ifdef  __bsdi__
#define va_mode_t   int
#else
#define va_mode_t   mode_t
#endif
```

30　我们关掉mode变量中的用户执行位（S_IXUSR），其原因稍后解释。

my_pxmsg_mmap/mq_open.c

```
 1 #include    "unpipc.h"
 2 #include    "mqueue.h"
 3 #include    <stdarg.h>
 4 #define     MAX_TRIES   10    /* for waiting for initialization */
 5 struct mq_attr  defattr =
 6 { 0, 128, 1024, 0 };
 7 mqd_t
 8 mq_open(const char *pathname, int oflag, ...)
 9 {
10     int     i, fd, nonblock, created, save_errno;
11     long    msgsize, filesize, index;
12     va_list ap;
13     mode_t  mode;
14     int8_t *mptr;
15     struct stat statbuff;
16     struct mq_hdr   *mqhdr;
17     struct msg_hdr  *msghdr;
18     struct mq_attr  *attr;
19     struct mq_info  *mqinfo;
20     pthread_mutexattr_t mattr;
21     pthread_condattr_t  cattr;
22     created = 0;
23     nonblock = oflag & O_NONBLOCK;
24     oflag &= ~O_NONBLOCK;
25     mptr = (int8_t *) MAP_FAILED;
26     mqinfo = NULL;
27   again:
28     if (oflag & O_CREAT) {
29         va_start(ap, oflag);    /* init ap to final named argument */
30         mode = va_arg(ap, va_mode_t) & ~S_IXUSR;
31         attr = va_arg(ap, struct mq_attr *);
32         va_end(ap);
33             /* open and specify O_EXCL and user-execute */
34         fd = open(pathname, oflag | O_EXCL | O_RDWR, mode | S_IXUSR);
35         if (fd < 0) {
36             if (errno == EEXIST && (oflag & O_EXCL) == 0)
37                 goto exists;    /* already exists, OK */
38             else
39                 return((mqd_t) -1);
40         }
41         created = 1;
42             /* first one to create the file initializes it */
43         if (attr == NULL)
44             attr = &defattr;
45         else {
46             if (attr->mq_maxmsg <= 0 || attr->mq_msgsize <= 0) {
47                 errno = EINVAL;
48                 goto err;
49             }
50         }
```

my_pxmsg_mmap/mq_open.c

图5-21　mq_open函数：第一部分

创建一个新消息队列

33~34 按照由调用者指定的名字创建一个普通文件，并打开它的用户执行位。

处理潜在的竞争状态

35~40 要是我们只是打开该文件，内存映射其内容，并在调用者指定O_CREAT标志的前提下初始化映射文件（如稍后所述），就会碰到一个竞争状态。一个消息队列只在调用者指定了O_CREAT标志并且该消息队列原本不存在时才会由mq_open初始化。这意味着我们需要某种方法来检测消息队列是否存在。为此我们总是在open由调用者给定的内存映射文件时指定O_EXCL标志。但是只有调用者指定了O_EXCL标志时，来自open的EEXIST出错返回才会成为出自mq_open的错误。否则，如果open返回一个EEXIST错误，那说明由调用者给定的文件已经存在，我们就向前跳到图5-23，仿佛未曾指定过O_CREAT标志。

可能出现竞争状态是因为使用一个内存映射文件代表一个消息队列需要两个步骤来初始化一个新的消息队列：首先必须使用open创建该文件，其次必须初始化该文件的内容（稍后描述）。如果有两个线程（可以在同一进程或不同进程中）几乎同时调用mq_open，问题就会发生。一个线程创建该文件，然后系统在该线程完成初始化之前切换到第二个线程。第二个线程检测到该文件已存在（使用O_EXCL标志open），于是立即尝试使用该消息队列。不过只有在第一个线程初始化消息队列后，它才能被使用。我们使用该文件的用户执行位来指示该消息队列尚未初始化。该位只能由真正创建该文件的线程启用（使用O_EXCL标志检测是否由本线程创建了该文件），这个线程随后初始化该消息队列，并关掉用户执行位。我们在图10-43和图10-52中还将遇到类似的竞争状态。

109~111

检查属性

42~50 如果调用者给mq_open的最后一个参数指定了一个空指针，我们就使用图5-21开始处给出的默认属性：128个消息，每个消息1024字节。如果调用者指定了属性，我们就验证mq_maxmsg和mq_msgsize为正数。

mq_open函数的第二部分在图5-22中给出；它完成一个新队列的初始化。

my_pxmsg_mmap/mq_open.c

```
51              /* calculate and set the file size */
52          msgsize = MSGSIZE(attr->mq_msgsize);
53          filesize = sizeof(struct mq_hdr) + (attr->mq_maxmsg *
54                                              (sizeof(struct msg_hdr) + msgsize));
55          if (lseek(fd, filesize - 1, SEEK_SET) == -1)
56              goto err;
57          if (write(fd, "", 1) == -1)
58              goto err;
59
60              /* memory map the file */
61          mptr = mmap(NULL, filesize, PROT_READ | PROT_WRITE,
62                      MAP_SHARED, fd, 0);
63          if (mptr == MAP_FAILED)
64              goto err;
65
66              /* allocate one mq_info{} for the queue */
67          if ( (mqinfo = malloc(sizeof(struct mq_info))) == NULL)
68              goto err;
```

图5-22 mq_open函数第二部分：完成新队列的初始化

```
67          mqinfo->mqi_hdr = mqhdr = (struct mq_hdr *) mptr;
68          mqinfo->mqi_magic = MQI_MAGIC;
69          mqinfo->mqi_flags = nonblock;

70              /* initialize header at beginning of file */
71              /* create free list with all messages on it */
72          mqhdr->mqh_attr.mq_flags = 0;
73          mqhdr->mqh_attr.mq_maxmsg = attr->mq_maxmsg;
74          mqhdr->mqh_attr.mq_msgsize = attr->mq_msgsize;
75          mqhdr->mqh_attr.mq_curmsgs = 0;
76          mqhdr->mqh_nwait = 0;
77          mqhdr->mqh_pid = 0;
78          mqhdr->mqh_head = 0;
79          index = sizeof(struct mq_hdr);
80          mqhdr->mqh_free = index;
81          for (i = 0; i < attr->mq_maxmsg - 1; i++) {
82              msghdr = (struct msg_hdr *) &mptr[index];
83              index += sizeof(struct msg_hdr) + msgsize;
84              msghdr->msg_next = index;
85          }
86          msghdr = (struct msg_hdr *) &mptr[index];
87          msghdr->msg_next = 0;       /* end of free list */

88              /* initialize mutex & condition variable */
89          if ( (i = pthread_mutexattr_init(&mattr)) != 0)
90              goto pthreaderr;
91          pthread_mutexattr_setpshared(&mattr, PTHREAD_PROCESS_SHARED);
92          i = pthread_mutex_init(&mqhdr->mqh_lock, &mattr);
93          pthread_mutexattr_destroy(&mattr);  /* be sure to destroy */
94          if (i != 0)
95              goto pthreaderr;

96          if ( (i = pthread_condattr_init(&cattr)) != 0)
97              goto pthreaderr;
98          pthread_condattr_setpshared(&cattr, PTHREAD_PROCESS_SHARED);
99          i = pthread_cond_init(&mqhdr->mqh_wait, &cattr);
100         pthread_condattr_destroy(&cattr);   /* be sure to destroy */
101         if (i != 0)
102             goto pthreaderr;

103             /* initialization complete, turn off user-execute bit */
104         if (fchmod(fd, mode) == -1)
105             goto err;
106         close(fd);
107         return((mqd_t) mqinfo);
108     }
```

my_pxmsg_mmap/mq_open.c

图5-22（续）

设置文件大小

51~58　计算每个消息的大小，并向上舍入到下一个长整数大小的倍数。计算文件大小时会将在该文件开头分配的mq_hdr结构和在每个消息开头分配的msg_hdr结构所占的空间包括在内（图5-19）。使用lseek设置新创建文件的大小，然后往当前读写位置写入字节0。只调用ftruncate（13.3节）会更容易些，但是我们不能保证它可用来增长一个文件的大小。

内存映射该文件

59~63 使用mmap内存映射该文件。

分配mq_info结构

64~66 我们给mq_open的每次调用分配一个mq_info结构。然后初始化该结构。

初始化mq_hdr结构

67~68 初始化mq_hdr结构。设置消息链表的头（mqh_head）为0，把队列中所有消息加到空闲链表（mgh_free）中。

初始化互斥锁和条件变量

88~102 既然只要知道一个Posix消息队列的名字并具有足够的权限，任何进程都能共享它，那么我们必须以PTHEAD_PROCESS_SHARED属性初始化互斥锁和条件变量。为此我们首先调用pthread_mutexattr_init初始化一个互斥锁属性结构，再调用pthread_mutexattr_setpshared在该结构中设置进程间共享属性，然后调用pthread_mutex_init初始化互斥锁。对于条件变量也完成几乎同样的步骤。我们要小心地摧毁初始化了的互斥锁属性或条件变量属性，即使发生错误也这样，因为调用pthread_mutexattr_init或pthread_condattr_init可能分配了内存空间（习题7.3）。

112 ~ 113

关掉用户执行位

103~107 一旦消息队列已初始化，我们就关掉用户执行位。这样指示消息队列已初始化完毕。我们还close已内存映射的文件，因为映射到内存后就没有必要让它继续打开着（这样会占用一个描述符）。

图5-23给出了mq_open函数的最后一部分，它打开一个已存在的队列。

打开已存在的消息队列

109~115 我们是在O_CREAT标志未指定或O_CREAT指定了但消息队列已存在的条件下到达这里的。这两种情况下，我们将打开一个已存在的消息队列。我们open含有该消息队列的文件来读写，并使用mmap把该文件内存映射到当前进程的地址空间中。

就打开模式而言，我们的实现简化了。即使调用者指定了O_RDONLY，我们也必须给open和mmap指定读写访问，因为不可能从一个队列中读出一个消息而不改变该队列所在的内存映射文件。同样地，我们不可能往一个队列中写入一个消息而不读其内存映射文件。解决这个问题的方法之一是在mq_info结构中保存打开模式（O_RDONLY、O_WRONLY或O_RDWR），然后在各个函数中检查这个模式。例如，如果打开模式是O_WRONLY，mq_receive就应该失败。

验证消息队列已初始化

116~132 我们必须等待消息队列的初始化（以防多个线程几乎同时尝试创建同一个消息队列）。为此我们调用stat查看内存映射文件的权限（stat结构的st_mode成员）。如果用户执行位已关掉，那么消息队列已被初始化。

这段代码还处理另外一个可能的竞争状态。假设不同进程中的两个线程几乎同时打开同一个消息队列。第一个线程创建了由调用者给定的文件后阻塞在图5-22中的lseek调用中。第二个线程发现该文件已存在，于是跳转到exists标号处，在那儿它再次打开文件，然后阻塞。第一个线程再次运行，然而它在图5-22中的mmap调用失败（也许是因为它超过了自己的虚拟内存限制），于是跳转到err标号处unlink自己创建的文件。第二个线程继续运行，然而要是我们调用fstat而不是stat的话，该线程就可能在等待该文件初始化的for循环中超时。于是我们改为调用stat，而

且如果它返回一个文件不存在的错误，同时O_CREAT标志已指定，那么我们跳转到again标号处（图5-21）以再次创建该文件。这种可能的竞争状态也是我们在open调用中检查ENOENT错误的原因。

my_pxmsg_mmap/mq_open.c

```
109 exists:
110         /* open the file then memory map */
111     if ( (fd = open(pathname, O_RDWR)) < 0) {
112         if (errno == ENOENT && (oflag & O_CREAT))
113             goto again;
114         goto err;
115     }
116         /* make certain initialization is complete */
117     for (i = 0; i < MAX_TRIES; i++) {
118         if (stat(pathname, &statbuff) == -1) {
119             if (errno == ENOENT && (oflag & O_CREAT)) {
120                 close(fd);
121                 goto again;
122             }
123             goto err;
124         }
125         if ((statbuff.st_mode & S_IXUSR) == 0)
126             break;
127         sleep(1);
128     }
129     if (i == MAX_TRIES) {
130         errno = ETIMEDOUT;
131         goto err;
132     }
133     filesize = statbuff.st_size;
134     mptr = mmap(NULL, filesize, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
135     if (mptr == MAP_FAILED)
136         goto err;
137     close(fd);

138         /* allocate one mq_info{} for each open */
139     if ( (mqinfo = malloc(sizeof(struct mq_info))) == NULL)
140         goto err;

141     mqinfo->mqi_hdr = (struct mq_hdr *) mptr;
142     mqinfo->mqi_magic = MQI_MAGIC;
143     mqinfo->mqi_flags = nonblock;
144     return((mqd_t) mqinfo);
145 pthreaderr:
146     errno = i;
147 err:
148         /* don't let following function calls change errno */
149     save_errno = errno;
150     if (created)
151         unlink(pathname);
152     if (mptr != MAP_FAILED)
153         munmap(mptr, filesize);
154     if (mqinfo != NULL)
155         free(mqinfo);
156     close(fd);
157     errno = save_errno;
158     return((mqd_t) -1);
159 }
```

my_pxmsg_mmap/mq_open.c

图5-23　mq_open函数第三部分：打开已存在的队列

内存映射文件，分配并初始化mq_info结构

133~144 内存映射消息队列所在文件，然后关闭该文件的描述符。分配一个mq_info结构并将其初始化。返回值为指向这个新分配mq_info结构的指针。

处理错误

145~158 当在本函数中此前某处检测到一个错误时，程序将跳转到err标号处，同时errno已被设置成将由mq_open返回的值。这里我们要注意保证检测到错误后调用的用于清理的函数不影响将由本函数返回的errno值。

5.8.2 mq_close 函数

图5-24给出了我们的mq_close函数。

my_pxmsg_mmap/mq_close.c

```
 1 #include    "unpipc.h"
 2 #include    "mqueue.h"

 3 int
 4 mq_close(mqd_t mqd)
 5 {
 6     long    msgsize, filesize;
 7     struct mq_hdr   *mqhdr;
 8     struct mq_attr  *attr;
 9     struct mq_info  *mqinfo;

10     mqinfo = mqd;
11     if (mqinfo->mqi_magic != MQI_MAGIC) {
12         errno = EBADF;
13         return(-1);
14     }
15     mqhdr = mqinfo->mqi_hdr;
16     attr = &mqhdr->mqh_attr;

17     if (mq_notify(mqd, NULL) != 0)      /* unregister calling process */
18         return(-1);

19     msgsize = MSGSIZE(attr->mq_msgsize);
20     filesize = sizeof(struct mq_hdr) + (attr->mq_maxmsg *
21                                         (sizeof(struct msg_hdr) + msgsize));
22     if (munmap(mqinfo->mqi_hdr, filesize) == -1)
23         return(-1);

24     mqinfo->mqi_magic = 0;       /* just in case */
25     free(mqinfo);
26     return(0);
27 }
```

my_pxmsg_mmap/mq_close.c

114 ~ 116

图5-24 mq_close函数

取得指向各个结构的指针

10~16 验证参数的有效性后，取得指向内存映射区（mqhdr）和属性（在mq_hdr结构中）的指针。

注销调用进程

17~18 调用mq_notify注销队列的调用进程。如果该进程已注册，它就被注销，但是如果它未注册过，那也不返回错误。

撤销内存区映射，释放内存空间

19~25 给munmap计算待撤销内存映射文件的大小，释放mq_info结构所用的内存空间。为防

止本函数的调用者在该内存区被malloc重用之前继续使用已关闭的消息队列描述符，我们把魔数设置为0，这样我们的消息队列函数以后将检测到该错误。

注意，如果进程没有调用mq_close就终止，那么进程终止时发生同样的操作：内存映射文件被撤销映射，内存空间被释放。

5.8.3 mq_unlink函数

图5-25给出了我们的mq_unlink函数，它会删除与我们定义的某个消息队列相关联的名字。它只是调用Unix unlink函数。

my_pxmsg_mmap/mq_unlink.c

```
1 #include     "unpipc.h"
2 #include     "mqueue.h"

3 int
4 mq_unlink(const char *pathname)
5 {
6       if (unlink(pathname) == -1)
7           return(-1);
8       return(0);
9 }
```

my_pxmsg_mmap/mq_unlink.c

图5-25　mq_unlink函数

5.8.4 mq_getattr函数

图5-26给出了我们的mq_getattr函数，它返回调用者指定的队列的当前属性。

my_pxmsg_mmap/mq_getattr.c

```
1 #include     "unpipc.h"
2 #include     "mqueue.h"

3 int
4 mq_getattr(mqd_t mqd, struct mq_attr *mqstat)
5 {
6       int     n;
7       struct mq_hdr   *mqhdr;
8       struct mq_attr  *attr;
9       struct mq_info  *mqinfo;

10      mqinfo = mqd;
11      if (mqinfo->mqi_magic != MQI_MAGIC) {
12          errno = EBADF;
13          return(-1);
14      }
15      mqhdr = mqinfo->mqi_hdr;
16      attr = &mqhdr->mqh_attr;
17      if ( (n = pthread_mutex_lock(&mqhdr->mqh_lock)) != 0) {
18          errno = n;
19          return(-1);
20      }
21      mqstat->mq_flags = mqinfo->mqi_flags;   /* per-open */
22      mqstat->mq_maxmsg = attr->mq_maxmsg;    /* remaining three per-queue */
23      mqstat->mq_msgsize = attr->mq_msgsize;
24      mqstat->mq_curmsgs = attr->mq_curmsgs;

25      pthread_mutex_unlock(&mqhdr->mqh_lock);
26      return(0);
27 }
```

my_pxmsg_mmap/mq_getattr.c

图5-26　mq_getattr函数

获取队列的互斥锁

17~20 在取得属性前，必须获取由调用者指定的消息队列的互斥锁，以免另外某个线程中途修改它们。

5.8.5 mq_setattr 函数

图5-27给出了我们的mq_setattr函数，它给调用者指定的队列设置当前属性。

my_pxmsg_mmap/mq_setattr.c

```
 1 #include    "unpipc.h"
 2 #include    "mqueue.h"

 3 int
 4 mq_setattr(mqd_t mqd, const struct mq_attr *mqstat,
 5             struct mq_attr *omqstat)
 6 {
 7     int     n;
 8     struct mq_hdr   *mqhdr;
 9     struct mq_attr  *attr;
10     struct mq_info  *mqinfo;

11     mqinfo = mqd;
12     if (mqinfo->mqi_magic != MQI_MAGIC) {
13         errno = EBADF;
14         return(-1);
15     }
16     mqhdr = mqinfo->mqi_hdr;
17     attr = &mqhdr->mqh_attr;
18     if ( (n = pthread_mutex_lock(&mqhdr->mqh_lock)) != 0) {
19         errno = n;
20         return(-1);
21     }
22     if (omqstat != NULL) {
23         omqstat->mq_flags = mqinfo->mqi_flags;   /* previous attributes */
24         omqstat->mq_maxmsg = attr->mq_maxmsg;
25         omqstat->mq_msgsize = attr->mq_msgsize;
26         omqstat->mq_curmsgs = attr->mq_curmsgs;      /* and current status */
27     }
28     if (mqstat->mq_flags & O_NONBLOCK)
29         mqinfo->mqi_flags |= O_NONBLOCK;
30     else
31         mqinfo->mqi_flags &= ~O_NONBLOCK;

32     pthread_mutex_unlock(&mqhdr->mqh_lock);
33     return(0);
34 }
```

my_pxmsg_mmap/mq_setattr.c

图5-27 mq_setattr函数

返回当前属性

22~27 如果第三个参数是一个非空指针，那就在修改属性前返回先前的属性和当前的状态。

修改mq_flags

28~31 可使用本函数修改的唯一属性是mq_flags，我们将它放置在mq_info结构中。

5.8.6 mq_notify 函数

图5-28给出的mq_notify函数注册或注销所指定队列的调用进程。我们追踪当前已注册为

117～119 接收出自某个队列的通知的进程，办法是将它的进程ID存放在对应该队列的mq_hdr结构的mqh_pid成员中。对于一个给定队列，每次只有一个进程可注册。当一个进程注册自身时，它还把通过函数参数指定的sigevent结构保存到mqh_event结构中。

my_pxmsg_mmap/mq_notify.c

```
1 #include    "unpipc.h"
2 #include    "mqueue.h"
3 int
4 mq_notify(mqd_t mqd, const struct sigevent *notification)
5 {
6     int     n;
7     pid_t   pid;
8     struct mq_hdr  *mqhdr;
9     struct mq_info  *mqinfo;
10    mqinfo = mqd;
11    if (mqinfo->mqi_magic != MQI_MAGIC) {
12        errno = EBADF;
13        return(-1);
14    }
15    mqhdr = mqinfo->mqi_hdr;
16    if ( (n = pthread_mutex_lock(&mqhdr->mqh_lock)) != 0) {
17        errno = n;
18        return(-1);
19    }
20    pid = getpid();
21    if (notification == NULL) {
22        if (mqhdr->mqh_pid == pid) {
23            mqhdr->mqh_pid = 0; /* unregister calling process */
24        }                       /* no error if caller not registered */
25    } else {
26        if (mqhdr->mqh_pid != 0) {
27            if (kill(mqhdr->mqh_pid, 0) != -1 || errno != ESRCH) {
28                errno = EBUSY;
29                goto err;
30            }
31        }
32        mqhdr->mqh_pid = pid;
33        mqhdr->mqh_event = *notification;
34    }
35    pthread_mutex_unlock(&mqhdr->mqh_lock);
36    return(0);
37  err:
38    pthread_mutex_unlock(&mqhdr->mqh_lock);
39    return(-1);
40 }
```

my_pxmsg_mmap/mq_notify.c

图5-28　mq_notify函数

注销调用进程

20～24 如果第二个参数是个空指针，那么注销所指定队列的调用进程。可能让人奇怪的是，调用进程未曾被注册接收该队列的通知并未被指定为一种错误。120

注册调用进程

25～34 如果某个进程已被注册，我们就通过向它发送信号0（称为空信号（null signal））以检查它是否仍然存在。这么做只是执行普通的出错检查，而不发送任何信号，会在该进

程不存在时返回一个ESRCH错误。如果先前注册了的进程仍然存在，本函数就返回一个EBUSY错误。否则，保存调用进程的进程ID以及调用者的sigevent结构。

这里测试先前注册了的进程是否存在的方法是不完善的。该进程可能终止，而其进程ID却在以后某个时刻被重用。

5.8.7 mq_send函数

图5-29给出了我们的mq_send函数的前半部分。

my_pxmsg_mmap/mq_send.c

```
 1 #include    "unpipc.h"
 2 #include    "mqueue.h"
 3 int
 4 mq_send(mqd_t mqd, const char *ptr, size_t len, unsigned int prio)
 5 {
 6     int     n;
 7     long    index, freeindex;
 8     int8_t  *mptr;
 9     struct sigevent *sigev;
10     struct mq_hdr *mqhdr;
11     struct mq_attr *attr;
12     struct msg_hdr *msghdr, *nmsghdr, *pmsghdr;
13     struct mq_info *mqinfo;
14     mqinfo = mqd;
15     if (mqinfo->mqi_magic != MQI_MAGIC) {
16         errno = EBADF;
17         return(-1);
18     }
19     mqhdr = mqinfo->mqi_hdr;    /* struct pointer */
20     mptr = (int8_t *) mqhdr;    /* byte pointer */
21     attr = &mqhdr->mqh_attr;
22     if ( (n = pthread_mutex_lock(&mqhdr->mqh_lock)) != 0) {
23         errno = n;
24         return(-1);
25     }
26     if (len > attr->mq_msgsize) {
27         errno = EMSGSIZE;
28         goto err;
29     }
30     if (attr->mq_curmsgs == 0) {
31         if (mqhdr->mqh_pid != 0 && mqhdr->mqh_nwait == 0) {
32             sigev = &mqhdr->mqh_event;
33             if (sigev->sigev_notify == SIGEV_SIGNAL) {
34                 sigqueue(mqhdr->mqh_pid, sigev->sigev_signo,
35                          sigev->sigev_value);
36             }
37             mqhdr->mqh_pid = 0;         /* unregister */
38         }
39     } else if (attr->mq_curmsgs >= attr->mq_maxmsg) {
40             /* queue is full */
41         if (mqinfo->mqi_flags & O_NONBLOCK) {
42             errno = EAGAIN;
43             goto err;
44         }
45             /* wait for room for one message on the queue */
46         while (attr->mq_curmsgs >= attr->mq_maxmsg)
47             pthread_cond_wait(&mqhdr->mqh_wait, &mqhdr->mqh_lock);
48     }
```

my_pxmsg_mmap/mq_send.c

图5-29 mq_send函数：前半部分

初始化

14~29　取得指向我们将使用的各个结构的指针，获取访问调用者指定队列的互斥锁。检查待发送消息，确定其大小没有超过该队列的最大消息大小。

检查队列是否为空，若合适则发送通知

30~38　如果是在往一个空队列中放置消息，我们就检查是否有某个进程被注册为接收出自该队列的通知，并检查是否有某个线程阻塞在mq_receive调用中。对于后面那种检查而言，我们将看到图5-31和图5-32中的mq_receive函数维护着一个用来存放阻塞在空队列上的线程数的计数器（mqh_nwait）。如果该计数器的值为非零，我们就不向已注册了的进程发送任何通知。我们只处理SIGEV_SIGNAL这种通知方式，其信号通过调用sigqueue发出。已注册了的进程随后被注销。

调用sigqueue发送信号导致向信号处理程序传递的siginfo_t结构（5.7节）中si_code成员的值为SI_QUEUE，这是不正确的，正确的值应为SI_MESGQ。从用户进程中产生正确的si_code取决于实现。［IEEE 1996］第433页提到，要从一个用户函数库中产生该信号，需用到进入信号产生机制的一个隐藏接口。

检查队列是否填满

39~48　如果调用者指定的队列已填满，但是O_NONBLOCK标志已设置，我们就返回一个EAGAIN错误。否则，我们等待在条件变量mqh_wait上，该条件变量由我们的mq_receive函数在从某个填满的队列中读出一个消息时发给信号。

就mq_send调用在被某个由其调用进程捕获的信号中断时返回一个EINTR错误而言，我们的实现简化了。问题在于当信号处理程序返回时，pthread_cond_wait并不返回一个错误：它可能返回一个为0的值（这看来是次虚假的唤醒），也可能根本不返回。绕过这一问题的方法确实存在，但每种方法都不简单。

121 ~ 122

图5-30给出了mq_send函数的后半部分。至此我们已知道调用者指定的队列中有写入新消息的空间。

取得待用空闲块的索引

50~52　既然在调用者指定的队列初始化时创建的空闲消息数等于mq_maxmsg，我们就不应该有在空闲链表为空的前提下mq_curmsgs小于mq_maxmsg的状态。

复制消息

53~56　nmsghdr含有所映射内存区中用于存放待写入消息的位置的地址。该消息的优先级和长度存放在它的msg_hdr结构中，其内容则从调用者空间复制。

把新消息置于链表中正确位置

57~74　我们的链表中各消息的顺序是从开始处（mqh_head）的最高优先级到结束处的最低优先级。当一个新消息加入调用者指定的队列中，并且一个或多个同样优先级的消息已在该队列中时，这个新消息就加在最后一个优先级相同的消息之后。使用这样的排序方式后，mq_receive总是返回链表中的第一个消息（它是该队列上优先级最高的最早的消息）。当我们沿链表行进时，pmsghdr将含有链表中上一个消息的地址，因为它的msg_next值将含有该新消息的索引。

我们的做法在该队列中有大量消息时可能较慢，因为每次往该队列中写入一个消息时都得遍历大量的链表项。可以再维护一个索引，让它记住各个可能优先级的最后一个消息的位置。

my_pxmsg_mmap/mq_send.c

```
49          /* nmsghdr will point to new message */
50      if ( (freeindex = mqhdr->mqh_free) == 0)
51          err_dump("mq_send: curmsgs = %ld; free = 0", attr->mq_curmsgs);
52      nmsghdr = (struct msg_hdr *) &mptr[freeindex];
53      nmsghdr->msg_prio = prio;
54      nmsghdr->msg_len = len;
55      memcpy(nmsghdr + 1, ptr, len);      /* copy message from caller */
56      mqhdr->mqh_free = nmsghdr->msg_next;    /* new freelist head */

57          /* find right place for message in linked list */
58      index = mqhdr->mqh_head;
59      pmsghdr = (struct msg_hdr *) &(mqhdr->mqh_head);
60      while (index != 0) {
61          msghdr = (struct msg_hdr *) &mptr[index];
62          if (prio > msghdr->msg_prio) {
63              nmsghdr->msg_next = index;
64              pmsghdr->msg_next = freeindex;
65              break;
66          }
67          index = msghdr->msg_next;
68          pmsghdr = msghdr;
69      }
70      if (index == 0) {
71              /* queue was empty or new goes at end of list */
72          pmsghdr->msg_next = freeindex;
73          nmsghdr->msg_next = 0;
74      }
75          /* wake up anyone blocked in mq_receive waiting for a message */
76      if (attr->mq_curmsgs == 0)
77          pthread_cond_signal(&mqhdr->mqh_wait);
78      attr->mq_curmsgs++;

79      pthread_mutex_unlock(&mqhdr->mqh_lock);
80      return(0);

81  err:
82      pthread_mutex_unlock(&mqhdr->mqh_lock);
83      return(-1);
84  }
```

my_pxmsg_mmap/mq_send.c

图5-30 mq_send函数：后半部分

唤醒阻塞在mq_receive中的任何线程

75~77 如果在往该队列放置新消息前该队列为空，我们就调用pthread_cond_signal唤醒可能阻塞在mq_receive中的任何线程。

78 给当前在该队列中的消息数mq_curmsgs加1。

5.8.8 mq_receive函数

图5-31给出了我们的mq_receive函数的前半部分，它设置所需的指针、获取互斥锁，并验证调用者的缓冲区足以容纳最大的可能消息。

检查队列是否为空

30~40 如果由调用者指定的队列为空，而且O_NONBLOCK标志已设置，那就返回一个EAGAIN错误。否则，给该队列的mqh_nwait计数器加1，它由图5-29中的mq_send函数检查，检查前提是该队列为空，而且某个进程已注册成接收该队列的通知。然后等待在条件变量上，它由图5-29中的mq_send发送信号。

my_pxmsg_mmap/mq_receive.c

```
1 #include     "unpipc.h"
2 #include     "mqueue.h"
3 ssize_t
4 mq_receive(mqd_t mqd, char *ptr, size_t maxlen, unsigned int *priop)
5 {
6       int      n;
7       long     index;
8       int8_t  *mptr;
9       ssize_t len;
10      struct mq_hdr   *mqhdr;
11      struct mq_attr  *attr;
12      struct msg_hdr  *msghdr;
13      struct mq_info  *mqinfo;
14      mqinfo = mqd;
15      if (mqinfo->mqi_magic != MQI_MAGIC) {
16          errno = EBADF;
17          return(-1);
18      }
19      mqhdr = mqinfo->mqi_hdr;      /* struct pointer */
20      mptr = (int8_t *) mqhdr;      /* byte pointer */
21      attr = &mqhdr->mqh_attr;
22      if ( (n = pthread_mutex_lock(&mqhdr->mqh_lock)) != 0) {
23          errno = n;
24          return(-1);
25      }
26      if (maxlen < attr->mq_msgsize) {
27          errno = EMSGSIZE;
28          goto err;
29      }
30      if (attr->mq_curmsgs == 0) {        /* queue is empty */
31          if (mqinfo->mqi_flags & O_NONBLOCK) {
32              errno = EAGAIN;
33              goto err;
34          }
35              /* wait for a message to be placed onto queue */
36          mqhdr->mqh_nwait++;
37          while (attr->mq_curmsgs == 0)
38              pthread_cond_wait(&mqhdr->mqh_wait, &mqhdr->mqh_lock);
39          mqhdr->mqh_nwait--;
40      }
```

my_pxmsg_mmap/mq_receive.c

图5-31　mq_receive函数：前半部分

跟mq_send的实现一样，就mq_receive调用在被某个由其调用进程捕获的信号中断时返回一个EINTR错误而言，我们的实现简化了。

123～125

图5-32给出了我们的mq_receive函数的后半部分。至此我们已知道由调用者指定的队列中有一个消息准备返回给调用者。

给调用者返回消息

43～51　msghdr指向调用者指定队列中第一个消息的msg_hdr，它是我们要返回的。由该消息占据的空间变为空闲链表的新头。

唤醒阻塞在mq_send中的任何线程

52～54　如果该队列在我们从中取走待读出消息前是填满的，我们就调用pthread_cond_signal，以防某个线程阻塞在mq_send调用中等待一个消息的空间。

my_pxmsg_mmap/mq_receive.c

```
41      if ( (index = mqhdr->mqh_head) == 0)
42          err_dump("mq_receive: curmsgs = %ld; head = 0", attr->mq_curmsgs);

43      msghdr = (struct msg_hdr *) &mptr[index];
44      mqhdr->mqh_head = msghdr->msg_next;     /* new head of list */
45      len = msghdr->msg_len;
46      memcpy(ptr, msghdr + 1, len);       /* copy the message itself */
47      if (priop != NULL)
48          *priop = msghdr->msg_prio;

49          /* just-read message goes to front of free list */
50      msghdr->msg_next = mqhdr->mqh_free;
51      mqhdr->mqh_free = index;

52          /* wake up anyone blocked in mq_send waiting for room */
53      if (attr->mq_curmsgs == attr->mq_maxmsg)
54          pthread_cond_signal(&mqhdr->mqh_wait);
55      attr->mq_curmsgs--;

56      pthread_mutex_unlock(&mqhdr->mqh_lock);
57      return(len);

58    err:
59      pthread_mutex_unlock(&mqhdr->mqh_lock);
60      return(-1);
61  }
```

my_pxmsg_mmap/mq_receive.c

图5-32 mq_receive函数：后半部分

5.9 小结

Posix消息队列比较简单：mq_open创建一个新队列或打开一个已存在的队列，mq_close关闭队列，mq_unlink则删除队列名。往一个队列中放置消息使用mq_send，从一个队列中读出消息使用mq_receive。队列属性的查询与设置使用mq_getattr和mq_setattr，函数mq_notify则允许我们注册一个信号或线程，它们在有一个消息被放置到某个空队列上时发送（信号）或激活（线程）。队列中的每个消息被赋予一个小整数优先级，mq_receive每次被调用时总是返回最高优先级的最早消息。

126

rnq_notify的使用给我们引入了Posix实时信号，它们在SIGRTMIN和SIGRTMAX之间。当设置SA_SIGINFO标志来安装这些信号的处理程序时，（1）这些信号是排队的，（2）排了队的信号是以FIFO顺序递交的，（3）给信号处理程序传递两个额外的参数。

最后，我们使用内存映射I/O以及一个Posix互斥锁和一个Posix条件变量，以约500行C代码实现了Posix消息队列的大多数特性。该实现展示了处理新队列的创建中存在的一个竞争状态，我们在第10章中实现Posix信号量时将遇到同样的竞争状态。

习题

5.1 在介绍图5-5时我们说过，当创建新队列时，如果mq_open的*attr*参数非空，那么mq_maxmsg和mq_msgsize两个成员都必须指定。怎么做才能允许我们只指定其中的一个成员，而未指定的那个成员则采用系统的默认值？

5.2 修改图5-9中的程序，使得所讨论的信号在递交之时并不调用mq_notify。然后往相应的队列发送两

个消息，验证对应第二个消息的信号没有产生。为什么？

5.3 修改图5-9中的程序，使得所讨论的信号在递交之时并不从相应队列中读出消息。相反，处理程序只是调用mq_notify并输出接收到的信号。然后往该队列发送两个消息，验证对应第二个消息的信号没有产生。为什么？

5.4 在图5-17的第一个printf中，如果我们把那两个常值向整数的类型强制转换去掉，那么会发生什么情况？

5.5 如下修改图5-5中的程序：在调用mq_open之前，输出一个消息并sleep 30秒。在mq_open返回之后，输出另一个消息，sleep 30秒，然后调用mq_close。编译并运行该程序，指定一个较大的消息数（几十万个），最大消息大小则（譬如说）为10字节。其目的是创建一个大消息队列（数百万字节大小），然后查看消息队列的实现是否使用内存映射文件。在第一个30秒停顿期间，运行一个诸如ps这样的程序，看一看修改后程序的内存大小。mq_open返回后再做一遍。你能解释所发生的情况吗？

5.6 当mq_send的调用者指定一个0长度时，图5-30中的memcpy调用会发生什么？

5.7 此较消息队列和4.4节讲述的全双工管道。父子进程之间的双向通信需多少个消息队列？

5.8 图5-24中我们为什么不摧毁互斥锁和条件变量？

5.9 Posix说消息队列描述符不能是数组类型。为什么？

5.10 图5-14中的main函数在哪儿花费大部分时间？每次递交一个信号后发生什么？我们如何处理这种情形？ 127

5.11 不是所有实现都支持互斥锁和条件变量的PTHREAD_PROCESS_SHARED属性。重新编写5.8节中Posix消息队列的实现，使用Posix信号量（第10章）代替互斥锁和条件变量。

5.12 把5.8节中Posix消息队列的实现扩展成支持SIGEV_THREAD。 128

第6章

System V 消息队列

6.1 概述

System V消息队列使用消息队列标识符（message queue identifier）标识。具有足够特权（3.5节）的任何进程都可以往一个给定队列放置一个消息，具有足够特权的任何进程都可以从一个给定队列读出一个消息。跟Posix消息队列一样，在某个进程往一个队列中写入一个消息之前，不求另外某个进程正在等待该队列上一个消息的到达。

对于系统中的每个消息队列，内核维护一个定义在<sys/msg.h>头文件中的信息结构。

```
struct msqid_ds {
  struct ipc_perm   msg_perm;    /* read_write perms: Section 3.3 */
  struct msg       *msg_first;   /* ptr to first message on queue */
  struct msg       *msg_last;    /* ptr to last message on queue */
  msglen_t          msg_cbytes;  /* current # bytes on queue */
  msgqnum_t         msg_qnum;    /* current # of messages on queue */
  msglen_t          msg_qbytes;  /* max # of bytes allowed on queue */
  pid_t             msg_lspid;   /* pid of last msgsnd() */
  pid_t             msg_lrpid;   /* pid 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()
                                    (that changed the above) */
};
```

Unix 98不要求有msg_first、msg_last和msg_cbytes成员。然而普通的源自System V的实现中可找到这三个成员。很自然，将一个队列中的各个消息作为一个链表来维护的要求并不存在，但这却是msg_first和msg_last这两个成员所隐含的。就算提供了这两个指针，那么它们指向的是内核内存空间，对于应用来说基本上没有作用。

我们可以将内核中某个特定的消息队列画为一个消息链表，如图6-1所示。假设有一个具有三个消息的队列，消息长度分别为1字节、2字节和3字节，而且这些消息就是以这样的顺序写入该队列的。再假设这三个消息的类型（type）分别为100、200和300。

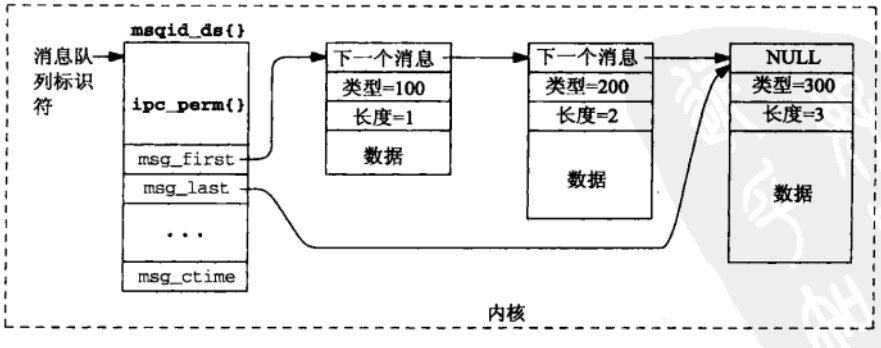


图6-1 内核中的System V消息队列结构

我们将在本章中查看操纵System V消息队列的函数，并使用消息队列实现4.2节中的文件服务器例子。

6.2 msgget函数

msgget函数用于创建一个新的消息队列或访问一个已存在的消息队列。

```
#include <sys/msg.h>
int msgget(key_t key, int oflag);
                                        返回：若成功则为非负标识符，若出错则为-1
```

返回值是一个整数标识符，其他三个msg函数就用它来指代该队列。它是基于指定的*key*产生的，而*key*既可以是ftok的返回值，也可以是常值IPC_PRIVATE，如图3-3所示。

*oflag*是图3-6中所示的读写权限值的组合。它还可以与IPC_CREAT或IPC_CREAT|IPC_EXCL按位或，如随图3-4的讨论所述。

130 当创建一个新消息队列时，msqid_ds结构的如下成员被初始化。

- msg_perm结构的uid和cuid成员被设置成当前进程的有效用户ID，gid和cgid成员被设置成当前进程的有效组ID。
- *oflag*中的读写权限位存放在msg_perm.mode中。
- msg_qnum、msg_lspid、msg_lrpid、msg_stime和msg_rtime被置为0。
- msg_ctime被设置成当前时间。
- msg_qbytes被设置成系统限制值。

6.3 msgsnd函数

使用msgget打开一个消息队列后，我们使用msgsnd往其上放置一个消息。

```
#include <sys/msg.h>
int msgsnd(int msqid, const void *ptr, size_t length, int flag);
                                        返回：若成功则为0，若出错则为-1
```

其中*msqid*是由msgget返回的标识符。*ptr*是一个结构指针，该结构具有如下的模板，它定义在<sys/msg.h>中。

```
struct msgbuf {
  long mtype;       /* message type, must be > 0 */
  char mtext[1];    /* message data */
};
```

消息类型必须大于0，因为对于msgrcv函数来说，非正的消息类型用作特殊的指示器，我们将在下一节讲述。

msgbuf结构定义中的名字mtext不大确切，消息的数据部分并不局限于文本。任何形式的数据都是允许的，无论是二进制数据还是文本。内核根本不解释消息数据的内容。

我们使用“模板”的说法描述这个结构，因为*ptr*所指向的只是一个含有消息类型的长整数，消息本身则紧跟在它之后（如果消息长度大于0字节）。不过大多数应用并不使用ms_qbuf结构的这个定义，因为其数据量（1个字节）通常是不够的。一个消息中的数据量并不存在编译时限

制（其系统限制则通常可由系统管理员修改），因此可不去声明一个数据量很大（比一个给定应用可能支持的数据还要大）的结构，而去定义一个上述的模板。大多数应用然后定义自己的消息结构，其数据部分根据应用的需要定义。

举例来说，如果某个应用需要交换由一个16位整数后跟一个8字节字符数组构成的消息，那 131
它可以定义自己的结构如下：

```
#define MY_DATA  8

typedef struct my_msgbuf {
  long     mtype;       /* message type */
  int16_t  mshort;      /* start of message data */
  char     mchar[MY_DATA];
} Message;
```

msgsnd的*length*参数以字节为单位指定待发送消息的长度。这是位于长整数消息类型之后的用户自定义数据的长度。该长度可以是0。在刚刚给出的例子中，长度可以传递成sizeof (Message) - sizeof(long)。

*flag*参数既可以是0，也可以是IPC_NOWAIT。IPC_NOWAIT标志使得msgsnd调用非阻塞（nonblocking）：如果没有存放新消息的可用空间，该函数就马上返回。这个条件可能发生的情形包括：

- 在指定的队列中已有太多的字节（对应该队列的msqid_ds结构中的msg_qbytes值）；
- 在系统范围存在太多的消息。

如果这两个条件中有一个存在，而且IPC_NOWAIT标志已指定，msgsnd就返回一个EAGAIN错误。如果这两个条件中有一个存在，但是IPC_NOWAIT标志未指定，那么调用线程被投入睡眠，直到：

- 具备存放新消息的空间；
- 由*msqid*标识的消息队列从系统中删除（这种情况下返回一个EIDRM错误）；
- 调用线程被某个捕获的信号所中断（这种情况下返回一个EINTR错误）。

6.4 msgrcv 函数

使用msgrcv函数从某个消息队列中读出一个消息。

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

ssize_t msgrcv(int msqid, void *ptr, size_t length, long type, int flag);
                                        返回：若成功则为读入缓冲区中数据的字节数，若出错则为-1
```

其中*ptr*参数指定所接收消息的存放位置。跟msgsnd一样，该指针指向紧挨在真正的消息数据之前返回的长整数类型字段（图4-26）。

*length*指定了由*ptr*指向的缓冲区中数据部分的大小。这是该函数能返回的最大数据量。该长度不包括长整数类型字段。 132

*type*指定希望从所给定的队列中读出什么样的消息。

- 如果*type*为0，那就返回该队列中的第一个消息。既然每个消息队列都是作为一个FIFO（先进先出）链表维护的，因此*type*为0指定返回该队列中最早的消息。
- 如果*type*大于0，那就返回其类型值为*type*的第一个消息。
- 如果*type*小于0，那就返回其类型值小于或等于*type*参数的绝对值的消息中类型值最小的

第一个消息。

考虑图6-1中所示的消息队列例子，它含有三个消息：

- 第一个消息的类型为100，长度为1；
- 第二个消息的类型为200，长度为2；
- 最后一个消息的类型为300，长度为3。

图6-2展示了为不同的*type*值返回的消息的类型。

type	所返回消息的类型
0	100
100	100
200	200
300	300
-100	100
-200	100
-300	100

图6-2　由msgrcv给不同的*type*值返回的消息

msgrcv的*flag*参数指定所请求类型的消息不在所指定的队列中时该做何处理。在没有消息可得的情况下，如果设置了*flag*中的IPC_NOWAIT位，msgrcv函数就立即返回一个ENOMSG错误。否则，调用者被阻塞到下列某个事件发生为止：

(1) 有一个所请求类型的消息可获取；

(2) 由*msqid*标识的消息队列被从系统中删除（这种情况下返回一个EIDRM错误）；

(3) 调用线程被某个捕获的信号所中断（这种情况下返回一个EINTR错误）。

*flag*参数中另有一位可以指定：MSG_NOERROR。当所接收消息的真正数据部分大于*length*参数时，如果设置了该位，msgrcv函数就只是截短数据部分，而不返回错误。否则，ms_grcv返回一个E2BIG错误。

133

成功返回时，msgrcv返回的是所接收消息中数据的字节数。它不包括也通过*ptr*参数返回的长整数消息类型所需的几个字节。

6.5　msgctl 函数

msgctl函数提供在一个消息队列上的各种控制操作。

```
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buff);
                                        返回：若成功则为0，若出错则为-1
```

msgctl函数提供3个命令。

IPC_RMID　从系统中删除由*msqid*指定的消息队列。当前在该队列上的任何消息都被丢弃。我们已在图3-7中看到过这种操作的例子。对于该命令而言，msgctl函数的第三个参数被忽略。

IPC_SET　给所指定的消息队列设置其msqid_ds结构的以下4个成员：msg_perm.uid、msg_perm.gid、msg_perm.mode和msg_qbytes。它们的值来自由*buff*参数指向的结构中的相应成员。

IPC_STAT （通过*buff*参数）给调用者返回与所指定消息队列对应的当前msqid_ds结构。

例子

图6-3中的程序创建一个消息队列，往该队列中放置一个含有1字节数据的消息，发出msgctl的IPC_STAT命令，使用system函数执行ipcs命令，最后使用msgctl的IPC_RMID命令删除该队列。

svmsg/ctl.c

```
 1 #include    "unpipc.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     msqid;
 6     struct msqid_ds info;
 7     struct msgbuf   buf;
 8     msqid = Msgget(IPC_PRIVATE, SVMSG_MODE | IPC_CREAT);
 9     buf.mtype = 1;
10     buf.mtext[0] = 1;
11     Msgsnd(msqid, &buf, 1, 0);
12     Msgctl(msqid, IPC_STAT, &info);
13     printf("read-write: %03o, cbytes = %lu, qnum = %lu, qbytes = %lu\n",
14            info.msg_perm.mode & 0777, (ulong_t) info.msg_cbytes,
15            (ulong_t) info.msg_qnum, (ulong_t) info.msg_qbytes);
16     system("ipcs -q");
17     Msgctl(msqid, IPC_RMID, NULL);
18     exit(0);
19 }
```

svmsg/ctl.c

图6-3 使用msgctl命令的例子

我们把1字节长度的消息写入所创建的队列中，这里只要使用定义在<sys/msg.h>中的标准msgbuf结构就行了。

执行该程序的结果如下：

```
solaris % ctl
read-write: 664, cbytes = 1, qnum = 1, qbytes = 4096
IPC status from <running system> as of Mon Oct 20 15:36:40 1997
T         ID      KEY         MODE        OWNER    GROUP
Message Queues:
q       1150   00000000  --rw-rw-r--  rstevens   other1
```

这与预期的一致。如3.2节所提，0是IPC_PRIVATE键的共同键值。执行本例子的系统上每个消息队列有4096字节的限制。既然我们写了一个1字节数据的消息，而且msg_cbytes的值为1，那么该限制显然只适用于消息的数据部分，而不包括与每个消息关联的长整数消息类型。

6.6 简单的程序

既然System V消息队列是随内核持续的，我们就可以编写一组小程序来操纵它们，以便观察效果。

6.6.1 msgcreate程序

图6-4给出了我们的msgcreate程序，它创建一个消息队列。

svmsg/msgcreate.c

```
 1 #include    "unpipc.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     int   c, oflag, mqid;
 6     oflag = SVMSG_MODE | IPC_CREAT;
 7     while ( (c = Getopt(argc, argv, "e")) != -1) {
 8         switch (c) {
 9         case 'e':
10             oflag |= IPC_EXCL;
11             break;
12         }
13     }
14     if (optind != argc - 1)
15         err_quit("usage: msgcreate [ -e ] <pathname>");
16     mqid = Msgget(Ftok(argv[optind], 0), oflag);
17     exit(0);
18 }
```

svmsg/msgcreate.c

图6-4　创建一个System V消息队列

9-12　我们允许使用-e命令行选项来指定IPC_EXCL标志。

16　把必须由用户作为命令行参数提供的路径名作为参数传递给ftok。导出的键由msgget转换成一个标识符（见习题6.1）。

6.6.2 msgsnd程序

图6-5给出了我们的msgsnd程序，它把一个指定了长度和类型的消息放置到某个队列中。

svmsg/msgsnd.c

```
 1 #include    "unpipc.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     mqid;
 6     size_t  len;
 7     long    type;
 8     struct msgbuf *ptr;
 9     if (argc != 4)
10         err_quit("usage: msgsnd <pathname> <#bytes> <type>");
11     len = atoi(argv[2]);
12     type = atoi(argv[3]);
13     mqid = Msgget(Ftok(argv[1], 0), MSG_W);
14     ptr = Calloc(sizeof(long) + len, sizeof(char));
15     ptr->mtype = type;
16     Msgsnd(mqid, ptr, len, 0);
17     exit(0);
18 }
```

svmsg/msgsnd.c

图6-5　往一个System V消息队列中加一个消息

我们先分配一个指向通用msgbuf结构的指针，然后根据待发送消息的大小调用calloc分配真正的结构（例如输出缓冲区）。calloc函数还把所分配的缓冲区初始化为0。 134 ~ 136

6.6.3 msgrcv 程序

图6-6给出了我们的msgrcv程序，它从一个队列中读出一个消息。-n命令行选项指定非阻塞，-t命令行选项指定msgrcv函数的*type*参数。

svmsg/msgrcv.c

```
 1 #include    "unpipc.h"
 2 #define MAXMSG  (8192 + sizeof(long))
 3 int
 4 main(int argc, char **argv)
 5 {
 6     int     c, flag, mqid;
 7     long    type;
 8     ssize_t n;
 9     struct msgbuf *buff;
10     type = flag = 0;
11     while ( (c = Getopt(argc, argv, "nt:")) != -1) {
12         switch (c) {
13         case 'n':
14             flag |= IPC_NOWAIT;
15             break;
16         case 't':
17             type = atol(optarg);
18             break;
19         }
20     }
21     if (optind != argc - 1)
22         err_quit("usage: msgrcv [ -n ] [ -t type ] <pathname>");
23     mqid = Msgget(Ftok(argv[optind], 0), MSG_R);
24     buff = Malloc(MAXMSG);
25     n = Msgrcv(mqid, buff, MAXMSG, type, flag);
26     printf("read %d bytes, type = %ld\n", n, buff->mtype);
27     exit(0);
28 }
```

svmsg/msgrcv.c

图6-6 从一个System V消息队列中读出一个消息

2 没有简单的方法用以确定一个消息的最大大小（我们将在6.10节讨论这个限制及其他限制），因此我们给它定义了自己的常值。

6.6.4 msgrmid 程序

要删除一个消息队列，我们以IPC_RMID命令调用msgctl，如图6-7所示。 137

svmsg/msgrmid.c

```
 1 #include    "unpipc.h"
 2 int
 3 main(int argc, char **argv)
```

图6-7 删除一个System V消息队列

```
4 {
5       int     mqid;

6       if (argc != 2)
7           err_quit("usage: msgrmid <pathname>");

8       mqid = Msgget(Ftok(argv[1], 0), 0);
9       Msgctl(mqid, IPC_RMID, NULL);

10      exit(0);
11 }
```

svmsg/msgrmid.c

图6-7（续）

6.6.5 例子

现在使用刚刚给出的四个程序。首先创建一个消息队列并往其中写入三个消息。

```
solaris % msgcreate /tmp/no/such/file
ftok error for pathname "/tmp/no/such/file" and id 0: No such file or directory
solaris % touch /tmp/test1
solaris % msgcreate /tmp/test1
solaris % msgsnd /tmp/test1 1 100
solaris % msgsnd /tmp/test1 2 200
solaris % msgsnd /tmp/test1 3 300
solaris % ipcs -qo
IPC status from <running system> as of Sat Jan 10 11:25:45 1998
T         ID      KEY         MODE        OWNER    GROUP CBYTES  QNUM
Message Quenes:
q        100   0x0000113e --rw-r--r-- rstevens   other1      6     3
```

我们首先使用一个不存在的路径名尝试创建一个消息队列。这个示例表明ftok的路径名参数必须存在。我们随后创建文件/tmp/test1，并使用该路径名创建一个消息队列。往该队列中放置三个消息：长度分别为1字节、2字节和3字节，类型分别为100、200和300（回想图6-1）。ipcs程序指出这三个消息总共构成该队列中的6个字节。

接着展示在不以FIFO顺序读出消息时msgrcv的*type*参数的使用。

```
solaris % msgrcv -t 200 /tmp/test1
read 2 bytes, type = 200
solaris % msgrcv -t -300 /tmp/test1
read 1 bytes, type = 100
solaris % msgrcv /tmp/test1
read 3 bytes, type = 300
solaris % msgrcv -n /tmp/test1
msgrcv error: No message of desired type
```

138

其中第一个例子请求读出类型为200的消息，第二个例子请求读出类型小于或等于300且是最小的消息，第三个例子请求读出该队列中的第一个消息。最后一次执行msgrcv程序用上了IPC_NOWAIT标志。

如果我们给msgrcv指定一个正的*type*参数，但是队列中不存在具有该类型的消息，那会发生什么？

```
solaris % ipcs -qo
IPC status from <running system> as of Sat Jan 10 11:37:01 1998
T         ID      KEY         MODE        OWNER    GROUP CBYTES  QNUM
Message Quenes:
q        100   0x0000113e --rw-r--r-- rstevens   other1      0     0
```

```
solaris % msgsnd /tmp/test1 1 100
solaris % msgrcv -t 999 /tmp/test1
^?                                          键入中断键终止程序执行
solaris % msgrcv -n -t 999 /tmp/test1
msgrcv error: No message of desired type
solaris % grep desired /usr/include/sys/errno.h
#define ENOMSG  35      /* No message of desired type */
solaris % msgrmid /tmp/test1
```

我们首先执行ipcs验证刚才的队列是空的，然后往其中放置一个长度为1字节、类型为100的消息。当请求读出一个类型为999的消息时，msgrcv程序阻塞（阻塞在msgrcv调用中），等待某个该类型的消息被放置到该队列中。我们用中断键终止该程序以中断其中的阻塞。接着在指定-n标志以防止阻塞的前提下重新执行msgrcv程序，结果看到返回ENOMSG错误。然后用我们的msgrmid程序从系统中删除该队列。我们也可以使用由系统提供的命令删除该队列。下面的命令指定的是消息队列标识符：

```
solaris % ipcrm -q 100
```

下面的命令指定的是消息队列键：

```
solaris % ipcrm -Q 0x113e
```

6.6.6 msgrcvid 程序

要访问一个System V消息队列，调用msgget并不是必须的：我们只需知道该消息队列的标识符（使用ipcs极易得到），并拥有该队列的读权限。图6-8是图6-6中msgrcv程序的简化版本。

svmsg/msgrcvid.c

```
 1 #include    "unpipc.h"
 2 #define MAXMSG  (8192 + sizeof(long))
 3 int
 4 main(int argc, char **argv)
 5 {
 6     int     mqid;
 7     ssize_t n;
 8     struct msgbuf *buff;
 9     if (argc != 2)
10         err_quit("usage: msgrcvid <mqid>");
11     mqid = atoi(argv[1]);
12     buff = Malloc(MAXMSG);
13     n = Msgrcv(mqid, buff, MAXMSG, 0, 0);
14     printf("read %d bytes, type = %ld\n", n, buff->mtype);
15     exit(0);
16 }
```

svmsg/msgrcvid.c

图6-8 只知道标识符时从一个System V消息队列中读

我们没有调用msgget，而是由调用者在命令行上指定消息队列标识符。下面是使用这种技巧的一个例子。

```
solaris % touch /tmp/testid
solaris % msgcreate /tmp/testid
solaris % msgsnd /tmp/testid 4 400
solaris % ipcs -qo
IPC status from <running system> as of Wed Mar 25 09:48:28 1998
```

```
T         ID      KEY        MODE        OWNER    GROUP CBYTES  QNUM
Message Queues:
q        150   0x0000118a --rw-r--r-- rstevens   other1       4     1
solaris % msgrcvid 150
read 4 bytes, type = 400
```

我们从ipcs的输出获得标识符为150，它就是我们的msgrcvid程序的命令行参数。

同样的特性也适用于System V信号量（习题11.1）和System V共享内存区（习题14.1）。

6.7　客户-服务器例子

现在我们把4.2节中的客户-服务器例子编写成使用两个消息队列。一个队列用于从客户到服务器的消息，另一个队列用于从服务器到客户的消息。

图6-9给出了我们svmsg.h头文件。其中包括了我们的标准头文件（unpipc.h），并定义了两个消息队列的键。

svmsgcliserv/svmsg.h

```
1 #include    "unpipc.h"
2 #define  MQ_KEY1 1234L
3 #define  MQ_KEY2 2345L
```

svmsgcliserv/svmsg.h

图6-9　使用消息队列的客户-服务器程序的头文件

图6-10给出了服务器程序的main函数。创建两个方向的消息队列，不过任何一个已经存在也没有关系，因为我们没有指定IPC_EXCL标志。server函数是图4-30中所示的版本，它调用的mesg_send和mesg_recv函数我们稍后给出。

svmsgcliserv/server_main.c

```
 1 #include    "svmsg.h"
 2 void    server(int, int);
 3 int
 4 main(int argc, char **argv)
 5 {
 6     int     readid, writeid;
 7     readid = Msgget(MQ_KEY1, SVMSG_MODE | IPC_CREAT);
 8     writeid = Msgget(MQ_KEY2, SVMSG_MODE | IPC_CREAT);
 9     server(readid, writeid);
10     exit(0);
11 }
```

svmsgcliserv/server_main.c

图6-10　使用消息队列的服务器程序main函数

图6-11给出了客户程序的main函数。我们打开两个方向的消息队列，随后调用图4-29中的client函数。该函数调用我们接下去给出的mesg_send和mesg_recv函数。

client和server函数都使用图4-25中所示的消息格式。这两个函数还调用我们的mesg_send和mesg_recv函数。图4-27和图4-28给出的这两个函数的版本调用了write和read，这对于管道和FIFO是有用的，但对于消息队列则需要重新编写它们。图6-12和图6-13给出了它们的新版本。注意，这两个函数的前后两个版本需传递的参数不变，因为第一个整数参数既可以含有一个整数描述符（用于访问管道或FIFO），也可以含有一个整数消息队列标识符。

svmsgcliserv/client_main.c

```
1 #include    "svmsg.h"
2 void    client(int, int);
3 int
4 main(int argc, char **argv)
5 {
6     int     readid, writeid;
7         /* assumes server has created the queues */
8     writeid = Msgget(MQ_KEY1, 0);
9     readid = Msgget(MQ_KEY2, 0);
10    client(readid, writeid);
11        /* now we can delete the queues */
12    Msgctl(readid, IPC_RMID, NULL);
13    Msgctl(writeid, IPC_RMID, NULL);
14    exit(0);
15 }
```

svmsgcliserv/client_main.c

图6-11 使用消息队列的客户程序main函数

svmsgcliserv/mesg_send.c

```
1 #include    "mesg.h"
2 ssize_t
3 mesg_send(int id, struct mymesg *mptr)
4 {
5     return(msgsnd(id, &(mptr->mesg_type), mptr->mesg_len, 0));
6 }
```

svmsgcliserv/mesg_send.c

图6-12 用于消息队列的mesg_send函数

svmsgcliserv/mesg_recv.c

```
1 #include    "mesg.h"
2 ssize_t
3 mesg_recv(int id, struct mymesg *mptr)
4 {
5     ssize_t n;
6     n = msgrcv(id, &(mptr->mesg_type), MAXMESGDATA, mptr->mesg_type, 0);
7     mptr->mesg_len = n;          /* return #bytes of data */
8     return(n);                   /* -1 on error, 0 at EOF, else >0 */
9 }
```

svmsgcliserv/mesg_recv.c

图6-13 用于消息队列的mesg_recv函数

6.8 复用消息

与一个队列中的每个消息相关联的类型字段提供了两个特性。

(1) 类型字段可用于标识消息，从而允许多个进程在单个队列上复用（multiplex）消息。举例来说，类型字段的某个值用于标识从各个客户到服务器的消息，对于每个客户均为唯一的另外某个值用于标识从服务器到各个客户的消息。每个客户的进程ID自然可用作对于每个客户均为唯一的类型字段。

(2) 类型字段可用作优先级字段。这允许接收者以不同于先进先出（FIFO）的某个顺序读出各个消息。使用管道或FIFO时，数据必须以写入的顺序读出。使用System V消息队列时，消息能够以任意顺序读出，只要跟与消息类型关联的值一致就行。而且我们可以指定IPC_NOWAIT标志调用msgrcv从某个队列中读出某个给定类型的任意消息，但是如果没有给定类型的消息存在，那就立即返回。

6.8.1 例子：每个应用一个队列

回想我们那个由一个服务器进程和单个客户进程构成的简单应用例子。使用管道和FIFO时，为在两个方向上交换数据需两个IPC通道，因为这两种类型的IPC是单向的。使用消息队列时，单个队列就够用，由每个消息的类型来标识该消息是从客户到服务器，还是从服务器到客户。

考虑更为复杂的情况：一个服务器带多个客户。这儿我们可以使用譬如说值为1的类型来指示从任意客户到服务器的消息。如果该客户将自己的进程ID作为消息的一部分传递，服务器就能把自己的消息发送给该客户，办法是把该客户的进程ID用作消息类型。每个客户然后把自己的进程ID指定为msgrcv的*type*参数。图6-14展示了单个消息队列是如何用于在多个客户和单个服务器之间复用消息的。

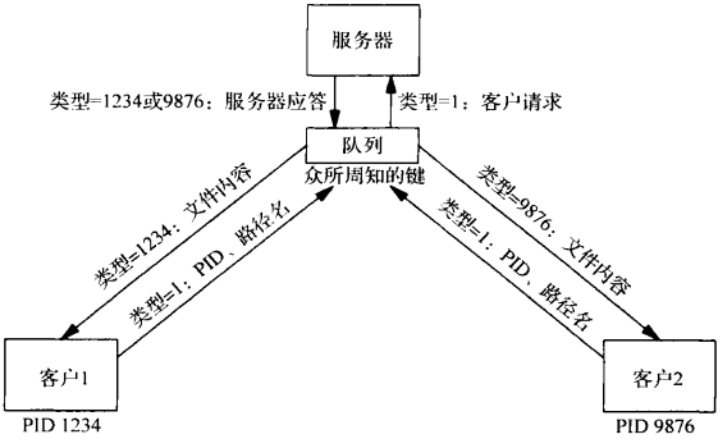


图6-14　在多个客户和单个服务器之间复用消息

当单个IPC通道同时由多个客户和单个服务器使用时，总是存在死锁的隐患。客户们可以填满消息队列（在本例子中），妨碍服务器发送应答。于是这些客户阻塞在msgsnd中，服务器也这样。可检测这种死锁的办法之一是，约定服务器对消息队列总是使用非阻塞写。

143

现在改用单个消息队列重新编写我们的客户-服务器例子程序，给每个方向的消息使用不同的消息类型。这些程序使用这样的约定：类型为1的消息是从客户到服务器的，所有其他消息有一个等于其客户进程ID的类型。该客户-服务器应用要求每个客户请求含有对应客户的进程ID以及所请求的路径名，这与我们在4.8节的做法类似。

图6-15给出了服务器程序的main函数。其中svmsg.h头文件在图6-9中给出。服务器只创建一个消息队列，如果它已经存在，那也没有关系。给server函数的两个参数所用的是同一个消息队列标识符。

svmsgmpx1q/server_main.c

```
1 #include    "svmsg.h"
2 void    server(int, int);
3 int
4 main(int argc, char **argv)
5 {
6     int     msqid;
7     msqid = Msgget(MQ_KEY1, SVMSG_MODE | IPC_CREAT);
8     server(msqid, msqid);        /* same queue for both directions */
9     exit(0);
10 }
```

svmsgmpx1q/server_main.c

图6-15 服务器程序main函数

server函数完成所有的服务器处理，在图6-16中给出。该函数是图4-23和图4-30的组合，其中图4-23是读出由一个进程ID和一个路径名构成的命令的FIFO服务器程序，图4-30则使用了我们的mesg_send和mesg_recv函数。注意，由某个客户发送的进程ID用作由服务器发送给该客户的所有消息的消息类型。这个server函数还是一个调用一次后永不返回的无限循环，每次循环读出一个客户请求并发回应答。这是个迭代服务器，如4.9节中讨论的那样。

svmsgmpx1q/server.c

```
1 #include    "mesg.h"
2 void
3 server(int readfd, int writefd)
4 {
5     FILE    *fp;
6     char    *ptr;
7     pid_t    pid;
8     ssize_t  n;
9     struct mymesg mesg;
10    for ( ; ; ) {
11            /* read pathname from IPC channel */
12        mesg.mesg_type = 1;
13        if ( (n = Mesg_recv(readfd, &mesg)) == 0) {
14            err_msg("pathname missing");
15            continue;
16        }
17        mesg.mesg_data[n] = '\0';    /* null terminate pathname */
18        if ( (ptr = strchr(mesg.mesg_data, ' ')) == NULL) {
19            err_msg("bogus request: %s", mesg.mesg_data);
20            continue;
21        }
22        *ptr++ = 0;              /* null terminate PID, ptr = pathname */
23        pid = atol(mesg.mesg_data);
24        mesg.mesg_type = pid; /* for messages back to client */
25        if ( (fp = fopen(ptr, "r")) == NULL) {
26                /* error: must tell client */
27            snprintf(mesg.mesg_data + n, sizeof(mesg.mesg_data) - n,
28                    ": can't open, %s\n", strerror(errno));
29            mesg.mesg_len = strlen(ptr);
```

图6-16 server函数

```
30              memmove(mesg.mesg_data, ptr, mesg.mesg_len);
31              Mesg_send(writefd, &mesg);

32          } else {
33                  /* fopen succeeded: copy file to IPC channel */
34              while (Fgets(mesg.mesg_data, MAXMESGDATA, fp) != NULL) {
35                  mesg.mesg_len = strlen(mesg.mesg_data);
36                  Mesg_send(writefd, &mesg);
37              }
38              Fclose(fp);
39          }

40              /* send a 0-length message to signify the end */
41          mesg.mesg_len = 0;
42          Mesg_send(writefd, &mesg);
43      }
44  }
```

svmsgmpx1q/server.c

图6-16（续）

图6-17给出了客户程序的main函数。客户打开唯一的消息队列，它必须已由服务器创建。

svmsgmpx1q/client_main.c

```
1  #include    "svmsg.h"

2  void    client(int, int);

3  int
4  main(int argc, char **argv)
5  {
6      int     msqid;

7          /* server must create the queue */
8      msqid = Msgget(MQ_KEY1, 0);

9      client(msqid, msqid);          /* same queue for both directions */

10     exit(0);
11 }
```

svmsgmpx1q/client_main.c

图6-17　客户程序main函数

图6-18所示的client函数为我们的客户完成所有处理。该函数是图4-24和图4-29的组合，其中图4-24发送一个进程ID后跟一个路径名，图4-29使用了我们的mesg_send和mesg_recv函数。注意从mesg_recv请求的消息类型等于当前客户的进程ID。

svmsgmpx1q/client.c

```
1  #include    "mesg.h"

2  void
3  client(int readfd, int writefd)
4  {
5      size_t   len;
6      ssize_t  n;
7      char    *ptr;
8      struct mymesg mesg;

9          /* start buffer with pid and a blank */
10     snprintf(mesg.mesg_data, MAXMESGDATA, "%ld ", (long) getpid());
11     len = strlen(mesg.mesg_data);
```

图6-18　client函数

```
12      ptr = mesg.mesg_data + len;
13          /* read pathname */
14      Fgets(ptr, MAXMESGDATA - len, stdin);
15      len = strlen(mesg.mesg_data);
16      if (mesg.mesg_data[len-1] == '\n')
17          len--;                  /* delete newline from fgets() */
18      mesg.mesg_len = len;
19      mesg.mesg_type = 1;
20          /* write PID and pathname to IPC channel */
21      Mesg_send(writefd, &mesg);
22          /* read from IPC, write to standard output */
23      mesg.mesg_type = getpid();
24      while ( (n = Mesg_recv(readfd, &mesg)) > 0)
25          Write(STDOUT_FILENO, mesg.mesg_data, n);
26  }
```

svmsgmpx1q/client.c

图6-18（续）

我们的client和server函数都使用图6-12和图6-13中的mesg_send和mesg_recv函数。

144 ~ 146

6.8.2 例子：每个客户一个队列

现在把前面的例子改成给去往服务器的所有客户请求使用一个队列，给每个客户使用一个队列接收去往各个客户的服务器应答。图6-19展示了这样的设计。

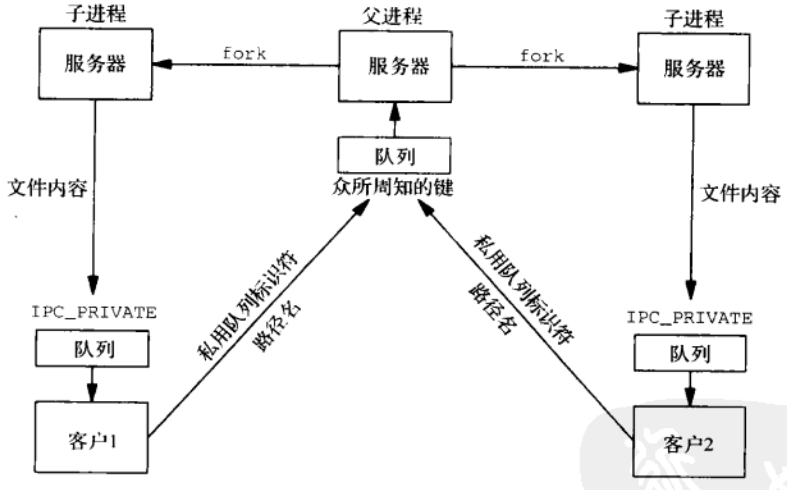


图6-19 每个服务器一个队列，每个客户一个队列

服务器的队列有一个对客户来说众所周知的键，但是各个客户以IPC_PRIVATE键创建各自的队列。这里并未随请求传递本进程ID，而是由每个客户把自己的私用队列的标识符传递给服务器，服务器把自己的应答发送到由客户指出的队列中。我们还以并发服务器模型编写这个服务器程序，给每个客户fork一次。

这种设计的潜在问题之一发生在某个客户中途死亡时，这种情况下它的私用队列中可能永远残留消息（或者至少到内核重新自举或某个用户显式地删除该队列为止）。

下列头文件和函数跟以前的版本一样：

- mesg.h头文件（图4-25）；
- svmsg.h头文件（图6-9）；
- 服务器程序main函数（图6-15）；
- mesg_send函数（图4-27）。

图6-20给出了我们的客户程序main函数，它与图6-17的差别很小。它打开服务器的众所周知队列（MQ_KEY1），然后用IPC_PRIVATE键创建自己的队列。这两个队列标识符成为client
147 函数（图6-21）的参数。当客户运行完时，它的私用队列被删除。

svmsgmpxnq/client_main.c

```
1 #include    "svmsg.h"
2 void    client(int, int);
3 int
4 main(int argc, char **argv)
5 {
6     int     readid, writeid;
7         /* server must create its well-known queue */
8     writeid = Msgget(MQ_KEY1, 0);
9         /* we create our own private queue */
10    readid = Msgget(IPC_PRIVATE, SVMSG_MODE | IPC_CREAT);
11    client(readid, writeid);
12        /* and delete our private queue */
13    Msgctl(readid, IPC_RMID, NULL);
14    exit(0);
15 }
```

svmsgmpxnq/client_main.c

图6-20　客户程序main函数

svmsgmpxnq/client.c

```
1 #include    "mesg.h"
2 void
3 client(int readid, int writeid)
4 {
5     size_t  len;
6     ssize_t n;
7     char    *ptr;
8     struct mymesg   mesg;
9         /* start buffer with msqid and a blank */
10    snprintf(mesg.mesg_data, MAXMESGDATA, "%d ", readid);
11    len = strlen(mesg.mesg_data);
12    ptr = mesg.mesg_data + len;
13        /* read pathname */
14    Fgets(ptr, MAXMESGDATA - len, stdin);
15    len = strlen(mesg.mesg_data);
16    if (mesg.mesg_data[len-1] == '\n')
17        len--;                  /* delete newline from fgets() */
18    mesg.mesg_len = len;
19    mesg.mesg_type = 1;
```

图6-21　client函数

```
20          /* write msqid and pathname to server's well-known queue */
21      Mesg_send(writeid, &mesg);

22          /* read from our queue, write to standard output */
23      while ( (n = Mesg_recv(readid, &mesg)) > 0)
24          Write(STDOUT_FILENO, mesg.mesg_data, n);
25  }
```

svmsgmpxnq/client.c

图6-21（续）

图6-21是client函数。该函数与图6-18几乎相同，但是作为请求的一部分传递的是本客户的私用队列标识符，而不是本客户的进程ID。mesg结构中的消息类型仍保留为1，因为它是两个方向的消息都使用的类型。

图6-23是server函数。与图6-16相比的主要变化是将这个函数编写为一个无限循环，每次循环给一个客户请求调用fork。

给SIGCHLD建立信号处理程序

10 既然是在给每个客户派生一个子进程，我们必须顾虑僵尸进程。UNPv1的5.9节和5.10节详细讨论了这一点。这里我们给SIGCHLD信号建立一个信号处理程序，这样当某个子进程终止时，我们的sig_chld（图6-22）就被调用。

12~18 服务器父进程阻塞在mesg_recv调用中，等待下一个客户消息的到达。

25~45 调用fork派生一个子进程，由子进程尝试打开所请求的文件，发回一个出错消息或该文件的内容。我们特意把fopen调用放在子进程而不是父进程中，以防该文件在某个远程文件系统上，因为要是这样，如果发生任何网络问题，文件的打开就可能花一段时间。

图6-22给出了SIGCHLD信号的处理程序。它复制自UNPv1的图5-11。

svmsgmpxnq/sigchldwaitpid.c

```
1 #include     "unpipc.h"

2 void
3 sig_chld(int signo)
4 {
5       pid_t   pid;
6       int     stat;

7       while ( (pid = waitpid(-1, &stat, WNOHANG)) > 0) ;
8       return;
9 }
```

svmsgmpxnq/sigchldwaitpid.c

图6-22 调用waitpid的SIGCHLD信号处理程序

每次调用我们的信号处理程序时，它就在一个循环中调用waitpid，以取得已终止的任何子进程的终止状态。该信号处理程序随后返回。这可能造成一个问题，因为父进程会把大部分时间花在阻塞在mesg_recv函数（图6-13）的msgrcv调用中。当我们的信号处理程序返回时，这个msgrcv调用就被中断。也就是说该函数将返回一个EINTR错误，如UNPv1的5.9节所述。

我们必须处理这个被中断的系统调用，图6-24给出Mesg_recv包裹函数的新版本。它允许有来自mesg_recv的EINTR错误（mesg_recv只是调用msgrcv），如果发生该错误，那就再次调用mesg_recv。

svmsgmpxnq/server.c

```
1 #include    "mesg.h"

2 void
3 server(int readid, int writeid)
4 {
5     FILE    *fp;
6     char    *ptr;
7     ssize_t  n;
8     struct mymesg   mesg;
9     void    sig_chld(int);

10    Signal(SIGCHLD, sig_chld);

11    for ( ; ; ) {
12            /* read pathname from our well-known queue */
13        mesg.mesg_type = 1;
14        if ( (n = Mesg_recv(readid, &mesg)) == 0) {
15            err_msg("pathname missing");
16            continue;
17        }
18        mesg.mesg_data[n] = '\0';      /* null terminate pathname */

19        if ( (ptr = strchr(mesg.mesg_data, ' ')) == NULL) {
20            err_msg("bogus request: %s", mesg.mesg_data);
21            continue;
22        }
23        *ptr++ = 0;                /* null terminate msgid, ptr = pathname */
24        writeid = atoi(mesg.mesg_data);

25        if (Fork() == 0) {         /* child */
26            if ( (fp = fopen(ptr, "r")) == NULL) {
27                    /* error: must tell client */
28                snprintf(mesg.mesg_data + n, sizeof(mesg.mesg_data) - n,
29                        ": can't open, %s\n", strerror(errno));
30                mesg.mesg_len = strlen(ptr);
31                memmove(mesg.mesg_data, ptr, mesg.mesg_len);
32                Mesg_send(writeid, &mesg);

33            } else {
34                    /* fopen succeeded: copy file to client's queue */
35                while (Fgets(mesg.mesg_data, MAXMESGDATA, fp) != NULL) {
36                    mesg.mesg_len = strlen(mesg.mesg_data);
37                    Mesg_send(writeid, &mesg);
38                }
39                Fclose(fp);
40            }

41                /* send a 0-length message to signify the end */
42            mesg.mesg_len = 0;
43            Mesg_send(writeid, &mesg);
44            exit(0);            /* child terminates */
45        }
46        /* parent just loops around */
47    }
48 }
```

svmsgmpxnq/server.c

150

图6-23　server函数

svmsgmpxnq/mesg_recv.c

```
10 ssize_t
11 Mesg_recv(int id, struct mymesg *mptr)
12 {
13     ssize_t n;

14     do {
15         n = mesg_recv(id, mptr);
16     } while (n == -1 && errno == EINTR);

17     if (n == -1)
18         err_sys("mesg_recv error");

19     return(n);
20 }
```

svmsgmpxnq/mesg_recv.c

图6-24 处理被中断系统调用的Mesg_recv包裹函数

6.9 消息队列上使用 select 和 poll

System V消息队列的问题之一是它们由各自的标识符而不是描述符标识。这意味着我们不能在消息队列上直接使用select或poll（UNPv1第6章）。

> 实际上有一个版本的Unix（即IBM的AIX）把select扩展成能够在描述符之外处理System V消息队列。不过这是不可移植的，只适用于AIX。

当有人想编写一个同时处理网络连接和IPC连接的服务器程序时，这种缺失的特性往往暴露出来。使用套接字API或XTI API（UNPv1）的网络通信使用的是描述符，因而允许使用select或poll。管道和FIFO也适合这两个函数，因为它们也是由描述符标识的。

解决该问题的办法之一是：让服务器创建一个管道，然后派生一个子进程，由子进程阻塞在msgrcv调用中。当有一个消息准备好被处理时，msgrcv返回，子进程接着从所指定的队列中读出该消息，并把该消息写入管道。服务器父进程当时可能在该管道以及一些网络连接上select。这种办法的负面效果是消息被处理了三次：一次是在子进程使用msgrcv读出时，一次是在子进程写入管道时，最后一次是在父进程从该管道中读出时。为避免这样的额外处理，父进程可以创建一个在它自身和子进程之间分享的共享内存区，然后把管道用作父子进程间的一种标志（习题12.5）。

> 在图5-14中我们给出了一种不需要fork就在Posix消息队列上间接使用select或poll的办法。在Posix消息队列上可以只使用单个进程的原因是它们提供了通知能力，当某个空队列上有一个消息到达时，这种通知能力将产生一个信号。System V消息队列不提供这种能力，因此必须fork一个子进程，由该子进程阻塞在msgrcv调用中。

151

与网络编程相比，System V消息队列另一个缺失的特性是无法窥探一个消息，而这是recv、recvform和recvmsg函数的MSG_PEEK标志提供的能力（UNPv1第356页[①]）。要是提供了这种能力，刚刚描述的（用于绕过select问题的）父子进程情形就可以做得更为有效，办法是让子进程指定msgrcv的窥探标志，当有一个消息准备好时就写1个字节到管道中，以通知父进程读出该消息。

① 此处为UNPv1第2版英文原版书页码，第3版为第388页。——编者注

6.10　消息队列限制

正如3.8节中所指出的那样，消息队列上往往存在某些系统限制。图6-25给出了两种实现上的这些值。第一栏是内容为所说明限制值的内核变量的传统SystemV名字。

名　字	说　明	DUnix 4.0B	Solaris 2.6
msgmax	每个消息的最大字节数	8 192	2 048
msgmnb	任何一个消息队列上的最大字节数	16 384	4 096
msgmni	系统范围的最大消息队列数	64	50
msgtql	系统范围的最大消息数	40	40

图6-25　System V消息队列的典型系统限制

许多源自SVR4的实现还有从它们的初始实现继承来的额外限制：msgssz和msgseg。msgssz往往是8，这是存放消息数据的“节（segment）”的大小（单位为字节）。一个21字节数据的消息将存放在3个这样的节中，其中最后一节的后3个字节未用上。msgseg是分配了的节数，往往是1 024。因历史原因，它一直存放在某个短整数中，因而必须小于32 768。所有消息数据的总共可用字节数是这两个变量的乘积，通常是8×1 024=8 192字节。

本节的目的是输出一些典型值，以辅助在代码移植上所作的计划。当一个系统运行大量使用消息队列的应用时，这些参数（或类似参数）的内核微调通常是必需的（关于内核参数微调已在3.8节中讲述过）。

例子

图6-26给出了确定图6-25中所示四个限制值的程序。

svmsg/limits.c

```
 1 #include    "unpipc.h"

 2 #define  MAX_DATA    64*1024
 3 #define  MAX_NMESG   4096
 4 #define  MAX_NIDS    4096
 5 int      max_mesg;

152
 6 struct mymesg {
 7   long    type;
 8   char    data[MAX_DATA];
 9 } mesg;

10 int
11 main(int argc, char **argv)
12 {
13     int     i, j, msqid, qid[MAX_NIDS];

14         /* first try and determine maximum amount of data we can send */
15     msqid = Msgget(IPC_PRIVATE, SVMSG_MODE | IPC_CREAT);
16     mesg.type = 1;
17     for (i = MAX_DATA; i > 0; i -= 128) {
18         if (msgsnd(msqid, &mesg, i, 0) == 0) {
19             printf("maximum amount of data per message = %d\n", i);
20             max_mesg = i;
21             break;
```

图6-26　确定System V消息队列上的系统限制

```
22          }
23          if (errno != EINVAL)
24              err_sys("msgsnd error for length %d", i);
25      }
26      if (i == 0)
27          err_quit("i == 0");
28      Msgctl(msqid, IPC_RMID, NULL);

29          /* see how many messages of varying size can be put onto a queue */
30      mesg.type = 1;
31      for (i = 8; i <= max_mesg; i *= 2) {
32          msqid = Msgget(IPC_PRIVATE, SVMSG_MODE | IPC_CREAT);
33          for (j = 0; j < MAX_NMESG; j++) {
34              if (msgsnd(msqid, &mesg, i, IPC_NOWAIT) != 0) {
35                  if (errno == EAGAIN)
36                      break;
37                  err_sys("msgsnd error, i = %d, j = %d", i, j);
38                  break;
39              }
40          }
41          printf("%d %d-byte messages were placed onto queue,", j, i);
42          printf(" %d bytes total\n", i*j);
43          Msgctl(msqid, IPC_RMID, NULL);
44      }

45          /* see how many identifiers we can "open" */
46      mesg.type = 1;
47      for (i = 0; i <= MAX_NIDS; i++) {
48          if ( (qid[i] = msgget(IPC_PRIVATE, SVMSG_MODE | IPC_CREAT)) == -1) {
49              printf("%d identifiers open at once\n", i);
50              break;
51          }
52      }
53      for (j = 0; j < i; j++)
54          Msgctl(qid[j], IPC_RMID, NULL);

55      exit(0);
56  }
```

svmsg/limits.c

图6-26（续）

153

确定最大消息大小

14~28 为确定最大消息大小，我们尝试发送一个含有65536字节数据的消息，如果失败了就尝试含有65408字节数据的消息，如此等等，直到msgsnd调用成功为止。

确定一个队列中可放置多少不同大小消息

29~44 从8字节消息开始查看一个给定队列上能放置多少消息。一旦确定该限制值，我们就删除其队列（丢弃其中的所有消息），并以16字节消息再次尝试。这样一直尝试到超过第一个步骤中确定的最大消息大小为止。我们预期较小的消息将在遇到每队列总消息数的限制，较大的消息将遇到每队列总字节数的限制。

确定同时可打开多少标识符

45~54 任何时刻能打开着的消息队列标识符最大数目通常存在一个系统限制。我们通过一直创建队列直到msgget失败来确定该限制。

我们先在Solaris 2.6下运行该程序，接着在Digital Unix 4.0B下运行，结果符合图6-25中所示的值。

```
solaris % limits
maximum amount of data per message = 2048
40 8-byte messages were placed onto queue, 320 bytes total
40 16-byte messages were placed onto queue, 640 bytes total
40 32-byte messages were placed onto queue, 1280 bytes total
40 64-byte messages were placed onto queue, 2560 bytes total
32 128-byte messages were placed onto queue, 4096 bytes total
16 256-byte messages were placed onto queue, 4096 bytes total
8 512-byte messages were placed onto queue, 4096 bytes total
4 1024-byte messages were placed onto queue, 4096 bytes total
2 2048-byte messages were placed onto queue, 4096 bytes total
50 identifiers open at once

alpha % limits
maximum amount of data per message = 8192
40 8-byte messages were placed onto queue, 320 bytes total
40 16-byte messages were placed onto queue, 640 bytes total
40 32-byte messages were placed onto queue, 1280 bytes total
40 64-byte messages were placed onto queue, 2560 bytes total
40 128-byte messages were placed onto queue, 5120 bytes total
40 256-byte messages were placed onto queue, 10240 bytes total
32 512-byte messages were placed onto queue, 16384 bytes total
16 1024-byte messages were placed onto queue, 16384 bytes total
8 2048-byte messages were placed onto queue, 16384 bytes total
4 4096-byte messages were placed onto queue, 16384 bytes total
2 8192-byte messages were placed onto queue, 16384 bytes total
63 identifiers open at once
```

Digital Unix下输出63个标识符的限制，而不是图6-25所示的64个，其原因在于已有一个系统守护进程在使用一个标识符。

154

6.11　小结

System V消息队列与Posix消息队列类似。新的应用程序应考虑使用Posix消息队列，不过大量的现有代码使用System V消息队列。然而把一个应用程序从使用System V消息队列重新编写成使用Posix消息队列不是件难事。Posix消息队列缺失的主要特性是从队列中读出指定优先级的消息的能力。这两种消息队列都不使用真正的描述符，从而造成在消息队列上使用select或poll的困难。

习题

6.1　修改图6-4中的程序以接受IPC_PRIVATE这样的路径名参数，若指定了这样的参数，就以一个私用键创建一个消息队列。这么一来，6.6节中的其余程序必须如何变动？

6.2　在图6-14中，我们为什么为发送给服务器的消息使用1这个类型？

6.3　在图6-14中，要是一个恶意的客户向服务器发送了许多消息，但它从来不去读服务器的应答，那么会发生什么？图6-19对于这种类型的客户有什么变动？

155

6.4　重新编写5.8节中Posix消息队列的实现，改用System V消息队列代替内存映射I/O。

第三部分

同　　步

第7章

互斥锁和条件变量

7.1 概述

从本章开始关于同步的讨论：怎样同步多个线程或多个进程的活动。为允许在线程或进程间共享数据，同步通常是必需的。互斥锁和条件变量是同步的基本组成部分。

互斥锁和条件变量出自Posix.1线程标准，它们总是可用来同步一个进程内的各个线程的。如果一个互斥锁或条件变量存放在多个进程间共享的某个内存区中，那么Posix还允许它用于这些进程间的同步。

> 这在Posix是个选项，在Unix 98却是必需的（参见图1-5中IPC类型为“进程间共享的互斥锁/条件变量”的那一行）。

本章中我们将介绍经典的生产者-消费者问题，并在解决该问题的方案中使用互斥锁和条件变量。对于本例子，我们使用多个线程而不是多个进程，因为让多个线程共享本问题中采用的公共数据缓冲区非常简单，而在多个进程间共享一个公共数据缓冲区却需要某种形式的共享内存区（将在第4部分中讲述）。我们将在第10章中提供使用信号量解决该问题的其他方案。

7.2 互斥锁：上锁与解锁

159

互斥锁指代相互排斥（mutual exclusion），它是最基本的同步形式。互斥锁用于保护临界区（critical region），以保证任何时刻只有一个线程在执行其中的代码（假设互斥锁由多个线程共享），或者任何时刻只有一个进程在执行其中的代码（假设互斥锁由多个进程共享）。保护一个临界区的代码的通常轮廓大体如下：

```
lock_the_mutex(...);
临界区
unlock_the_mutex(...);
```

既然任何时刻只有一个线程能够锁住一个给定的互斥锁，于是这样的代码保证任何时刻只有一个线程在执行其临界区中的指令。

Posix互斥锁被声明为具有pthread_mutex_t数据类型的变量。如果互斥锁变量是静态分配的，那么我们可以把它初始化成常值PTHREAD_MUTEX_INITIALIZER，例如：

```
static pthread_mutex_t  lock = PTHREAD_MUTEX_INITIALIZER;
```

如果互斥锁是动态分配的（例如通过调用malloc），或者分配在共享内存区中，那么我们必须在运行之时通过调用pthread_mutex_init函数来初始化它，如7.7节中所示。

> 你可能会碰到省略了初始化操作的代码，因为它所在的实现把初始化常值定义为0（而且静态分配的变量被自动地初始化为0）。不过这是不正确的代码。

下列三个函数给一个互斥锁上锁和解锁：

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mptr);
int pthread_mutex_trylock(pthread_mutex_t *mptr);
int pthread_mutex_unlock(pthread_mutex_t *mptr);
```

均返回：若成功则为0，若出错则为正的Exxx值

如果尝试给一个已由另外某个线程锁住的互斥锁上锁，那么pthread_mutex_lock将阻塞到该互斥锁解锁为止。pthread_mutex_trylock是对应的非阻塞函数，如果该互斥锁已锁住，它就返回一个EBUSY错误。

> 如果有多个线程阻塞在等待同一个互斥锁上，那么当该互斥锁解锁时，哪一个线程会开始运行呢？1003.1b-1993标准增加的特性之一是提供一个优先级调度选项。我们不讨论该领域，不过下述概括足以说明其内容：不同线程可被赋予不同的优先级，同步函数（互斥锁、读写锁、信号量）将唤醒优先级最高的被阻塞线程。[Butenhof 1997]的5.5节提供有关Posix.1实时调度特性的具体细节。

尽管我们说互斥锁保护的是临界区，实际上保护的是在临界区中被操纵的数据（data）。也就是说，互斥锁通常用于保护由多个线程或多个进程分享的共享数据（shared data）。 160

互斥锁是协作性（cooperative）锁。这就是说，如果共享数据是一个链表（举个例子），那么操纵该链表的所有线程都必须在实际操纵前获取该互斥锁。不过也没有办法防止某个线程不首先获取该互斥锁就操纵该链表。

7.3 生产者-消费者问题

同步中有一个称为生产者-消费者（producer-consumer）问题的经典问题，也称为有界缓冲区（bounded buffer）问题。一个或多个生产者（线程或进程）创建着一个个的数据条目，然后这些条目由一个或多个消费者（线程或进程）处理。数据条目在生产者和消费者之间是使用某种类型的IPC传递的。

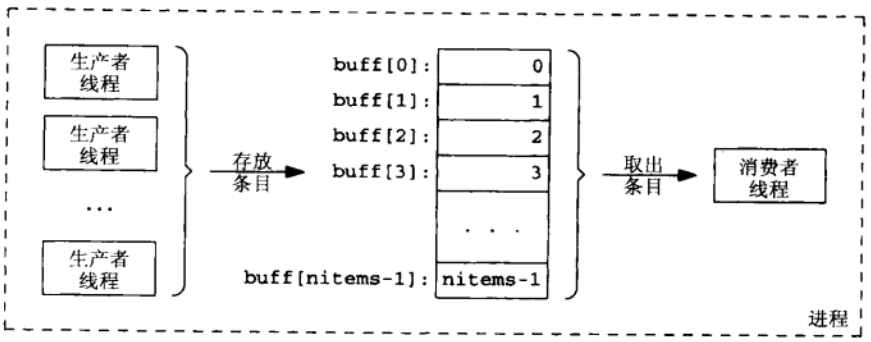
我们一直使用Unix管道处理这个问题。这就是说，如下的shell管道就是这样的问题：

```
grep pattern chapters.* | wc -l
```

grep是单个生产者，wc是单个消费者。Unix管道用作两者间的IPC形式。生产者和消费者间所需的同步是由内核以一定方式处理的，内核以这种方式处理生产者的write和消费者的read。如果生产者超前消费者（也就是管道被填满），内核就在生产者调用write时把它投入睡眠，直到管道中有空余空间。如果消费者超前生产者（也就是管道为空），内核就在消费者调用read时把它投入睡眠，直到管道中有一些数据为止。

这些类型的同步是隐式的（implicit）：也就是说生产者和消费者甚至不知道内核在执行同步。如果我们改用Posix消息队列或System V消息队列作为生产者和消费者间的IPC形式，那么内核仍然会处理同步。

然而当共享内存区用作生产者和消费者之间的IPC形式时，生产者和消费者必须执行某种类型的显式（explicit）同步。我们将使用互斥锁展示显式同步。图7-1展示了我们使用的例子。



161

图7-1　生产者-消费者例子：多个生产者线程、单个消费者线程

在单个进程中有多个生产者线程和单个消费者线程。整数数组buff含有被生产和消费的条目（也就是共享数据）。为简单起见，生产者只是把buff[0]设置为0，把buff[1]设置为1，如此等等。消费者只是沿着该数组行进，并验证每个数组元素都是正确的。

在第一个例子中，我们只关心多个生产者线程之间的同步。直到所有生产者线程都完成工作后，我们才启动消费者线程。图7-2是这个例子的main函数。

线程间共享的全局变量

4~12　这些变量是各个线程间共享的。我们把它们以及相应的互斥锁收集到一个名为shared的结构中，目的是为了强调这些变量只应该在拥有其互斥锁时访问。nput是buff数组中下一次存放的元素下标，nval是下一次存放的值（0、1、2，等等）。我们分配这个结构，并初始化其中用于生产者线程间同步的互斥锁。

> 我们将如本例子所做的那样一直努力地把共享数据和它们的同步变量（互斥锁、条件变量或信号量）收集到一个结构中，这是一个很好的编程技巧。然而在许多情况下共享数据是动态分配的，譬如说一个链表。我们可以把该链表的头以及该链表的同步变量存放到一个结构中（图5-20中的mq_hdr结构就是这么一回事），但是其他共享数据（该链表的其余部分）却不在该结构中。因此这种办法通常是不完善的。

命令行参数

19~22　第一个命令行参数指定生产者存放的条目数，下一个参数指定待创建生产者线程的数目。

设置并发级别

23　我们的set_concurrency函数告诉线程系统我们希望并发运行多少线程。在Solaris 2.6下，该函数只是调用thr_setconcurrency，当我们希望多个生产者线程中每一个都有执行机会时，这个函数是必需的。如果我们在Solaris下省略该调用，那么只有第一个生产者线程运行。在Digital Unix 4.0B下，我们的set_concurrency函数不做任何事（因为默认情况下，一个进程中的各个线程竞争使用处理器）。

> Unix 98需要一个名为pthread_setconcurrency的函数执行同样的功能。在把多个用户线程（使用pthread_create创建的对象）复用到较小的一组内核执行实体（例如内核线程）上的线程实现中，该函数是需要的。这些实现经常被称为多对少、两级或M对N实现。[butenhof 1997] 的5.6节详细讨论了用户线程和内核实体间的关系。

mutex/prodcons2.c

```
1 #include    "unpipc.h"

2 #define MAXNITEMS       1000000
3 #define MAXNTHREADS         100

4 int     nitems;                   /* read-only by producer and consumer */
5 struct {
6     pthread_mutex_t mutex;
7     int     buff[MAXNITEMS];
8     int     nput;
9     int     nval;
10 } shared = {
11     PTHREAD_MUTEX_INITIALIZER
12 };

13 void    *produce(void *), *consume(void *);

14 int
15 main(int argc, char **argv)
16 {
17     int     i, nthreads, count[MAXNTHREADS];
18     pthread_t   tid_produce[MAXNTHREADS], tid_consume;

19     if (argc != 3)
20         err_quit("usage: prodcons2 <#items> <#threads>");
21     nitems = min(atoi(argv[1]), MAXNITEMS);
22     nthreads = min(atoi(argv[2]), MAXNTHREADS);

23     Set_concurrency(nthreads);
24         /* start all the producer threads */
25     for (i = 0; i < nthreads; i++) {
26         count[i] = 0;
27         Pthread_create(&tid_produce[i], NULL, produce, &count[i]);
28     }

29         /* wait for all the producer threads */
30     for (i = 0; i < nthreads; i++) {
31         Pthread_join(tid_produce[i], NULL);
32         printf("count[%d] = %d\n", i, count[i]);
33     }

34         /* start, then wait for the consumer thread */
35     Pthread_create(&tid_consume, NULL, consume, NULL);
36     Pthread_join(tid_consume, NULL);

37     exit(0);
38 }
```

mutex/prodcons2.c

图7-2 main函数

创建生产者线程

24~28 创建生产者线程，每个线程执行produce。在tid_produce数组中保存每个线程的线程ID。传递给每个生产者线程的参数是指向count数组中某个元素的指针。我们首先把该计数器初始化为0，然后每个线程在每次往缓冲区中存放一个条目时给这个计数器加1。当一切生产完毕时，我们输出这个计数器数组各元素的值，以查看每个生产者线程分别存放了多少条目。

等待生产者线程，然后启动消费者线程

29~36 等待所有生产者线程终止，同时输出每个线程的计数器值，此后才启动单个的消费者

线程。这是我们（暂时）避免生产者和消费者之间的同步问题的办法。接着等待消费者完成，然后终止进程。

162～163 图7-3给出了这个例子所用的produce和consume函数。

mutex/prodcons2.c

```
39 void *
40 produce(void *arg)
41 {
42     for ( ; ; ) {
43         Pthread_mutex_lock(&shared.mutex);
44         if (shared.nput >= nitems) {
45             Pthread_mutex_unlock(&shared.mutex);
46             return(NULL);        /* array is full, we're done */
47         }
48         shared.buff[shared.nput] = shared.nval;
49         shared.nput++;
50         shared.nval++;
51         Pthread_mutex_unlock(&shared.mutex);
52         *((int *) arg) += 1;
53     }
54 }

55 void *
56 consume(void *arg)
57 {
58     int     i;

59     for (i = 0; i < nitems; i++) {
60         if (shared.buff[i] != i)
61             printf("buff[%d] = %d\n", i, shared.buff[i]);
62     }
63     return(NULL);
64 }
```

mutex/prodcons2.c

图7-3　produce和consume函数

产生数据条目

42～53　生产者的临界区由用来测试是否一切生产完毕的条件语句

```
if (shared.nput >= nitems)
```

和随后的三行

```
shared.buff[shared.nput] = shared.nval;
shared.nput++;
shared.nval++;
```

构成。

我们用一个互斥锁保护该临界区，同时保证在一切生产完毕的情况下给该互斥锁解锁。注意count元素的增加（通过指针arg）不属于该临界区，因为每个线程有各自的计数器（main函数中的count数组）。既然如此，我们就不把这行代码包括在由互斥锁锁住的临界区中，因为作为一个通用的编程原则，我们总是应该努力减少由一个互斥锁锁住的代码量。

消费者验证数组的内容

164　59～62　消费者只是验证buff数组中的每个条目是否正确，若发现错误则输出一个消息。正如我们先前所说，本函数只有一个实例在运行，而且是在所有生产者线程都完成之后，因此不需要任何同步。

指定一百万个条目和5个生产者线程运行上述程序，结果如下：

```
solaris % prodcons2 1000000 5
count[0] = 167165
count[1] = 249891
count[2] = 194221
count[3] = 191815
count[4] = 196908
```

正如我们所提，如果在Solaris 2.6下去掉set_concurrency调用，那么count[0]将变为1000000，其余计数器则都是0。

要是我们删除本例子中的互斥锁上锁，那么它将如期地失败。也就是说，消费者将检测出许多buff[i]不等于i的情况。我们还可以验证，如果只有一个生产者线程在运行，那么删除互斥锁上锁并没有影响。

7.4 对比上锁与等待

现在展示互斥锁用于上锁（locking）而不能用于等待（waiting）。我们把上一节中的生产者-消费者例子改为在所有生产者线程都启动后立即启动消费者线程。这样在生产者线程产生数据的同时，消费者线程就能处理它，而不是像图7-2中那样，消费者线程直到所有生产者线程都完成后才启动。但现在我们必须同步生产者和消费者，以确保消费者只处理已由生产者存放的数据条目。

图7-4给出了main函数。在main声明之前的所有行与图7-2中的一样。

mutex/prodcons3.c

```
14 int
15 main(int argc, char **argv)
16 {
17      int     i, nthreads, count[MAXNTHREADS];
18      pthread_t   tid_produce[MAXNTHREADS], tid_consume;

19      if (argc != 3)
20          err_quit("usage: prodcons3 <#items> <#threads>");
21      nitems = min(atoi(argv[1]), MAXNITEMS);
22      nthreads = min(atoi(argv[2]), MAXNTHREADS);

23          /* create all producers and one consumer */
24      Set_concurrency(nthreads + 1);
25      for (i = 0; i < nthreads; i++) {
26          count[i] = 0;
27          Pthread_create(&tid_produce[i], NULL, produce, &count[i]);
28      }
29      Pthread_create(&tid_consume, NULL, consume, NULL);

30          /* wait for all producers and the consumer */
31      for (i = 0; i < nthreads; i++) {
32          Pthread_join(tid_produce[i], NULL);
33          printf("count[%d] = %d\n", i, count[i]);
34      }
35      Pthread_join(tid_consume, NULL);

36      exit(0);
37 }
```

mutex/prodcons3.c

图7-4 main函数：启动生产者后立即启动消费者

24　给并发级别加1，把额外的消费者线程也计算在内。

25~29　创建生产者线程后，立即创建消费者线程。

produce函数没有变化，已在图7-3中给出。

图7-5给出了consume函数，它调用我们新定义的consume_wait函数。

mutex/prodcons3.c

```
54 void
55 consume_wait(int i)
56 {
57     for ( ; ; ) {
58         Pthread_mutex_lock(&shared.mutex);
59         if (i < shared.nput) {
60             Pthread_mutex_unlock(&shared.mutex);
61             return;              /* an item is ready */
62         }
63         Pthread_mutex_unlock(&shared.mutex);
64     }
65 }

66 void *
67 consume(void *arg)
68 {
69     int     i;

70     for (i = 0; i < nitems; i++) {
71         consume_wait(i);
72         if (shared.buff[i] != i)
73             printf("buff[%d] = %d\n", i, shared.buff[i]);
74     }
75     return(NULL);
76 }
```

mutex/prodcons3.c

图7-5　consume_wait和consume函数

消息者必须等待

166　71　consume函数的唯一变动是在从buff数组中取出下一个条目之前调用consume_wait。

等待生产者

57~64　我们的consume_wait函数必须等待到生产者产生了第i个条目。为检查这种条件，先给生产者的互斥锁上锁，再比较i和生产者的nput下标。我们必须在查看nput前获得互斥锁，因为某个生产者线程当时可能正处于更新该变量的过程中。

这里的基本问题是：当期待的条目尚未准备好时，我们能做些什么？图7-5中的做法是一次次地循环，每次给互斥锁解锁又上锁。这称为**轮转**（spinning）或**轮询**（polling），是一种对CPU时间的浪费。

我们也许可以睡眠很短的一段时间，但是不知道该睡眠多久。这儿所需的是另一种类型的同步，它允许一个线程（或进程）睡眠到发生某个事件为止。

7.5　条件变量：等待与信号发送

互斥锁用于上锁，条件变量则用于等待。这两种不同类型的同步都是需要的。

条件变量是类型为pthread_cond_t的变量，以下两个函数使用了这些变量。

```
#include <pthread.h>

int pthread_cond_wait(pthread_cond_t *cptr, pthread_mutex_t *mptr);

int pthread_cond_signal(pthread_cond_t *cptr);
```

均返回：若成功则为0，若出错则为正的Exxx值

其中第二个函数的名字中的“signal”一词指的不是Unix SIGxxx信号。

这两个函数所等待或由之得以通知的“条件”，其定义由我们选择：我们在代码中测试这种条件。

每个条件变量总是有一个互斥锁与之关联。我们调用pthread_cond_wait等待某个条件为真时，还会指定其条件变量的地址和所关联的互斥锁的地址。

我们通过重新编写上一节中的例子来解释条件变量的使用。图7-6给出了全局变量的声明。

mutex/prodcons6.c

```
 1 #include    "unpipc.h"

 2 #define MAXNITEMS        1000000
 3 #define MAXNTHREADS          100

 4         /* globals shared by threads */
 5 int     nitems;                 /* read-only by producer and consumer */
 6 int     buff[MAXNITEMS];
 7 struct {
 8     pthread_mutex_t     mutex;
 9     int     nput;               /* next index to store */
10     int     nval;               /* next value to store */
11 } put = {
12     PTHREAD_MUTEX_INITIALIZER
13 };

14 struct {
15     pthread_mutex_t mutex;
16     pthread_cond_t cond;
17     int     nready;             /* number ready for consumer */
18 } nready = {
19     PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER
20 };
```

mutex/prodcons6.c

图7-6 使用条件变量的生产者-消费者程序全局变量

把生产者变量和互斥锁收集到一个结构中

7~13 把互斥锁变量mutex以及与之关联的两个变量nput和nval收集到一个名为put的结构中。生产者使用这个结构。

把计数器、条件变量和互斥锁收集到一个结构中

14~20 下一个结构含有一个计数器、一个条件变量和一个互斥锁。我们把条件变量初始化为PTHREAD_COND_INITIALIZER。

main函数没有变动，已在图7-4中给出。

produce和consume函数变动了，在图7-7中给出。

往数组中放置下一个条目

50~58 当生产者往数组buff中放置一个新条目时，我们改用互斥锁put.mutex来为临界区上锁。

mutex/prodcons6.c

```
46 void *
47 produce(void *arg)
48 {
49     for ( ; ; ) {
50         Pthread_mutex_lock(&put.mutex);
51         if (put.nput >= nitems) {
52             Pthread_mutex_unlock(&put.mutex);
53             return(NULL);          /* array is full, we're done */
54         }
55         buff[put.nput] = put.nval;
56         put.nput++;
57         put.nval++;
58         Pthread_mutex_unlock(&put.mutex);

59         Pthread_mutex_lock(&nready.mutex);
60         if (nready.nready == 0)
61             Pthread_cond_signal(&nready.cond);
62         nready.nready++;
63         Pthread_mutex_unlock(&nready.mutex);

64         *((int *) arg) += 1;
65     }
66 }

67 void *
68 consume(void *arg)
69 {
70     int     i;

71     for (i = 0; i < nitems; i++) {
72         Pthread_mutex_lock(&nready.mutex);
73         while (nready.nready == 0)
74             Pthread_cond_wait(&nready.cond, &nready.mutex);
75         nready.nready--;
76         Pthread_mutex_unlock(&nready.mutex);

77         if (buff[i] != i)
78             printf("buff[%d] = %d\n", i, buff[i]);
79     }
80     return(NULL);
81 }
```

mutex/prodcons6.c

图7-7　produce和consume函数

通知消费者

59~64　给用来统计准备好由消费者处理的条目数的计数器nready.nready加1。在加1之前，如果该计数器的值为0，那就调用pthread_cond_signal唤醒可能正在等待其值变为非零的任意线程（如消费者）。现在可以看出与该计数器关联的互斥锁和条件变量的相互作用。该计数器是在生产者和消费者之间共享的，因此只有锁住与之关联的互斥锁（nready.mutex）时才能访问它。与之关联的条件变量则用于等待和发送信号。

消费者等待nready.nready变为非零

72~76　消费者只是等待计数器nready.nready变为非零。既然该计数器是在所有的生产者和消费者之间共享的，那么只有锁住与之关联的互斥锁（nready.mutex）时才能测试它的值。如果在锁住该互斥锁期间该计数器的值为0，我们就调用pthread_cond_wait进入睡眠。该函数原子地执行以下两个动作：

(1) 给互斥锁nready.mutex解锁；

(2) 把调用线程投入睡眠，直到另外某个线程就本条件变量调用pthread_cond_signal。

167 ~ 169

pthread_cond_wait在返回前重新给互斥锁nready.mutex上锁。因此当它返回并且我们发现计数器nready.nready不为0时，我们将把该计数器减1（前提是我们肯定已锁住了该互斥锁），然后给该互斥锁解锁。注意，当pthread_cond_wait返回时，我们总是再次测试相应条件成立与否，因为可能发生虚假的（spurious）唤醒：期待的条件尚不成立时的唤醒。各种线程实现都试图最大限度减少这些虚假唤醒的数量，但是仍有可能发生。

总的说来，给条件变量发送信号的代码大体如下：

```
struct {
  pthread_mutex_t  mutex;
  pthread_cond_t   cond;
  维护本条件的各个变量
} var = { PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER, ... };

Pthread_mutex_lock(&var.mutex);
设置条件为真
Pthread_cond_signal(&var.cond);
Pthread_mutex_unlock(&var.mutex);
```

在我们的例子中，用来维护条件的变量是一个整数计数器，设置条件的操作就是给该计数器加1。我们做了优化处理，即只有该计数器从0变为1时才发出条件变量信号。

测试条件并进入睡眠以等待该条件变为真的代码大体如下：

```
Pthread_mutex_lock(&var.mutex);
while (条件为假)
    Pthread_cond_wait(&var.cond, &var.mutex);
修改条件
Pthread_mutex_unlock(&var.mutex);
```

避免上锁冲突

在刚刚给出的代码片段以及图7-7中，pthread_cond_signal由当前锁住某个互斥锁的线程调用，而该互斥锁是与本函数将给它发送信号的条件变量关联的。我们可以设想下最坏情况，当该条件变量被发送信号后，系统立即调度等待在其上的线程，该线程开始运行，但立即停止，因为它没能获取相应的互斥锁。为避免这种上锁冲突，图7-7中的代码可作如下变动：

```
int  dosignal;

Pthread_mutex_lock(&nready.mutex);
dosignal = (nready.nready == 0);
nready.nready++;
Pthread_mutex_unlock(&neready.mutex);

if (dosignal)
    Pthread_cond_signal(&nready.cond);
```

在这儿，我们直到释放互斥锁nready.mutex后才给与之关联的条件变量nready.cond发送信号。Posix明确允许这么做：调用pthread_cond_signal的线程不必是与之关联的互斥锁的当前属主。不过Posix接着说：如果需要可预见的调度行为，那么调用pthread_cond_signal的线程必须锁住该互斥锁。

170

7.6　条件变量：定时等待和广播

通常pthread_cond_signal只唤醒等待在相应条件变量上的一个线程。在某些情况下一个线程认定有多个其他线程应被唤醒，这时它可调用pthread_cond_broadcast唤醒阻塞在相应条件变量上的所有线程。

> 有多个线程应唤醒的情形的例子之一发生在我们将在第8章中讲述的读出者与写入者问题中。当一个写入者完成访问并释放相应的锁后，它希望唤醒所有排着队的读出者，因为允许同时有多个读出者访问。
>
> 考虑条件变量信号单播发送与广播发送的一种候选（且更为安全的）方式是坚持使用广播发送。如果所有的等待者代码都编写确切，只有一个等待者需要唤醒，而且唤醒哪一个等待者无关紧要，那么可以使用为这些情况而优化的单播发送。所有其他情况下都必须使用广播发送。

```
#include <pthread.h>

int pthread_cond_broadcast(pthread_cond_t *cptr);

int pthread_cond_timedwait(pthread_cond_t *cptr, pthread_mutex_t *mptr,
                           const struct timespec *abstime);
```

均返回：若成功则为0，若出错则为正的Exxx值

pthread_cond_timedwait允许线程就阻塞时间设置一个限制值。*abstime*参数是一个timespec结构：

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

该结构指定这个函数必须返回时的系统时间，即便当时相应的条件变量还没有收到信号。如果发生这种超时情况，该函数就返回ETIMEDOUT错误。

时间值是绝对时间（absolute time），而不是时间差（time delta）。这就是说，*abstime*是该函数应该返回时刻的系统时间——自UTC时间1970年1月1日子时以来流逝的秒数和纳秒数。这与select、pselect和poll（UNPv1第6章）不同，它们都指定在将来的某个小数秒数，到时函数应该返回（select指定将来的微秒数，pselect指定将来的纳秒数，poll指定将来的毫秒数）。使用绝对时间而不是时间差的好处是：如果函数过早返回了（也许是因为捕获了某个信号），那么同一函数无需改变其参数中timespec结构的内容就能再次被调用。

171

7.7　互斥锁和条件变量的属性

本章中的互斥锁和条件变量例子把它们作为一个进程中的全局变量存放，它们用于该进程内各线程间的同步。我们用两个常值PTHREAD_MUTEX_INITIALIZER和PTHREAD_COND_INITIALIZER来初始化它们。由这种方式初始化的互斥锁和条件变量具备默认属性，不过我们还能以非默认属性初始化它们。

首先，互斥锁和条件变量是用以下函数初始化或摧毁的。

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *mptr, const pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mptr);
int pthread_cond_init(pthread_cond_t *cptr, const pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cptr);
                                        均返回：若成功则为0，若出错则为正的Exxx值
```

考虑互斥锁情况，*mptr*必须指向一个已分配的pthread_mutex_t变量，并由pthread_mutex_init函数初始化该互斥锁。由该函数第二个参数*attr*指向的pthread_mutexattr_t值指定其属性。如果该参数是个空指针，那就使用默认属性。

互斥锁属性的数据类型为pthread_mutexattr_t，条件变量属性的数据类型为pthread_condattr_t，它们由以下函数初始化或摧毁。

```
#include <pthread.h>
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
int pthread_condattr_init(pthread_condattr_t *attr);
int pthread_condattr_destroy(pthread_condattr_t *attr);
                                        均返回：若成功则为0，若出错则为正的Exxx值
```

一旦某个互斥锁属性对象或某个条件变量属性对象已被初始化，就通过调用不同函数启用或禁止特定的属性。举例来说，我们将在以后各章中使用的一个属性是：指定互斥锁或条件变量在不同进程间共享，而不是只在单个进程内的不同线程间共享。这个属性是用以下函数取得或存入的。 172

```
#include <pthread.h>
int pthread_mutexattr_getpshared(const pthread_mutexattr_t *attr, int *valptr);
int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr, int value);
int pthread_condattr_getpshared(const pthread_condattr_t *attr, int *valptr);
int pthread_condattr_setpshared(pthread_condattr_t *attr, int value);
                                        均返回：若成功则为0，若出错则为正的Exxx值
```

其中两个get函数返回在由*valptr*指向的整数中的这个属性的当前值，两个set函数则根据*value*的值设置这个属性的当前值。*value*的值可以是PTHREAD_PROCESS_PRIVATE或PTHREAD_PROCESS_SHARED。后者也称为进程间共享属性。

> 这个特性只在头文件<unistd.h>中定义了常值_POSIX_THREAD_PROCESS_SHARED时才得以支持。它在Posix.1中是可选特性，在Unix 98中却是必需的（图1-5）。

以下代码片段给出初始化一个互斥锁以便它能在进程间共享的过程：

```
pthread_mutex_t        *mptr;     /* pointer to the mutex in shared memory */
pthread_mutexattr_t     mattr;    /* mutex attribute datatype */
    . . .
    mptr = /* some value that points to shared memory */ ;
```

```
    Pthread_mutexattr_init(&mattr);
#ifdef _POSIX_THREAD_PROCESS_SHARED
    Pthread_mutexattr_setpshared(&mattr, PTHREAD_PROCESS_SHARED);
#else
# error this implementation does not support _POSIX_THREAD_PROCESS_SHARED
#endif
    Pthread_mutex_init(mptr, &mattr);
```

我们声明一个名为mattr的pthread_mutexattr_t数据类型的变量，把它初始化成互斥锁的默认属性，然后给它设置PTHREAD_PROCESS_SHARED属性，意思是该互斥锁将在进程间共享。pthread_mutex_init然后照此初始化该互斥锁。必须分配给该互斥锁的共享内存区空间大小为sizeof(pthread_mutex_t)。

用于给存放在共享内存区中供多个进程使用的一个条件变量设置PTHREAD_PROCESS_SHARED属性的一组语句跟用于互斥锁的语句几乎相同，只需把其中的5处mutex替换成cond。

173 我们已在图5-22中给出这些进程间共享的互斥锁和条件变量的例子。

持有锁期间进程终止

当在进程间共享一个互斥锁时，持有该互斥锁的进程在持有期间终止（也许是非自愿地）的可能总是有的。没有办法让系统在进程终止时自动释放所持有的锁。我们将会看到读写锁和Posix信号量也具备这种属性。进程终止时内核总是自动清理的唯一同步锁类型是fcntl记录锁（第9章）。使用System V信号量时，应用程序可以选择进程终止时内核是否自动清理某个信号量锁（将在11.3节中讨论的SEM_UNDO特性）。

一个线程也可以在持有某个互斥锁期间终止，起因是被另一个线程取消或自己去调用了pthread_exit。后者没什么可关注的，因为如果该线程调用pthread_exit自愿终止的话，它应该知道自己还持有一个互斥锁。如果是被另一个线程取消的情况，那么该线程可以安装将在被取消时调用的清理处理程序，如8.5节中所展示的那样。对于一个线程来说是致命的条件通常还导致整个进程的终止。举例来说，如果某个线程执行了一个无效指针访问，从而引发了SIGSEGV信号，那么一旦该信号未被捕获，整个进程就被它终止，我们于是回到了先前处理进程终止的条件上。

即使一个进程终止时系统会自动释放某个锁，那也可能解决不了问题。该锁保护某个临界区很可能是为了在执行该临界区代码期间更新某个数据。如果该进程在执行该临界区的中途终止，该数据处于什么状态呢？该数据处于不一致状态的可能性很大：举例来说，一个新条目也许只是部分插入某个链表中，要是该进程终止时内核仅仅把那个锁解开的话，使用该链表的下一个进程就可能发现它已损坏。

然而在某些例子中，让内核在进程终止时清理某个锁（若是信号量情况则为计数器）不成问题。例如，某个服务器可能使用一个System V信号量（打开其SEM_UNDO特性）来统计当前被处理的客户数。每次fork一个子进程时，它就把该信号量加1，当该子进程终止时，它再把该信号量减1。如果该子进程非正常终止，内核仍会把该计数器减1。9.7节给出了一个例子，说明内核在什么时候释放一个锁（不是我们刚讲的计数器）合适。那儿的守护进程一开始就在自己的某个数据文件上获得一个写入锁，然后在其运行期间一直持有该锁。如果有人试图启动该守护进程的另一个副本，那么新的副本将因为无法取得该写入锁而终止，从而确保该守护进程只有一个副本在一直运行。但是如果该守护进程不正常地终止了，那么内核会释放该写入锁，从而允许启动该守护进程的另一个副本。

7.8 小结

互斥锁用于保护代码临界区，从而保证任何时刻只有一个线程在临界区内执行。有时候一 174
个线程获得某个互斥锁后，发现自己需要等待某个条件变为真。如果是这样，该线程就可以等待在某个条件变量上。条件变量总是有一个互斥锁与之关联。把调用线程投入睡眠的pthread_cond_wait函数在这么做之前先给所关联的互斥锁解锁，以后某个时刻唤醒该线程前再给该互斥锁上锁。该条件变量由另外某个线程向它发送信号，而这个发送信号的线程既可以只唤醒一个线程（pthread_cond_signal），也可以唤醒等待相应条件变为真的所有线程（pthread_cond_broadcast）。

互斥锁和条件变量可以静态分配并静态初始化。它们也可以动态分配，那要求动态地初始化它们。动态初始化允许我们指定进程间共享属性，从而允许在不同进程间共享某个互斥锁或条件变量，其前提是该互斥锁或条件变量必须存放在由这些进程共享的内存区中。

习题

7.1 去掉图7-3中的互斥锁，验证这个例子在运行不止一个生产者线程的前提下会失败。

7.2 如果把图7-2中对消费者线程的Pthread_join调用去掉，那么会发生什么？

7.3 编写一个程序，在一个无限循环中只调用pthread_mutexattr_init和pthread_condattr_init。使用诸如ps这样的程序观察其进程的内存使用情况。发生了什么？现在加上合适的pthread_mutexattr_destroy和phtread_condattr_destory，再验证没有发生内存遗漏。

7.4 在图7-7中，生产者只在计数器nready.nready由0变为1时才调用pthread_cond_signal。为查看这种优化处理的效果，增设一个计数器，它在每次调用pthread_cond_signal时加1，当消费者完成时，在主线程中输出这个计数器的值。 175

第8章 读写锁

8.1 概述

互斥锁把试图进入我们称之为临界区的所有其他线程都阻塞住。该临界区通常涉及对由这些线程共享的一个或多个数据的访问或更新。然而有时候我们可以在读某个数据与修改某个数据之间作区分。

我们现在讲述读写锁（read-write lock），并在获取读写锁用于读和获取读写锁用于写之间作区分。这些读写锁的分配规则如下：

(1) 只要没有线程持有某个给定的读写锁用于写，那么任意数目的线程可以持有该读写锁用于读。

(2) 仅当没有线程持有某个给定的读写锁用于读或用于写时，才能分配该读写锁用于写。

换一种说法就是，只要没有线程在修改某个给定的数据，那么任意数目的线程都可以拥有该数据的读访问权。仅当没有其他线程在读或修改某个给定的数据时，当前线程才可以修改它。

某些应用中读数据比修改数据频繁，这些应用可从改用读写锁代替互斥锁中获益。任意给定时刻允许多个读出者存在提供了更高的并发度，同时在某个写入者修改数据期间保护该数据，以免任何其他读出者或写入者的干扰。

这种对于某个给定资源的共享访问也称为共享-独占（shared-exclusive）上锁，因为获取一个读写锁用于读称为共享锁（shared lock），获取一个读写锁用于写称为独占锁（exclusive lock）。有
177 关这种类型问题（多个读出者和一个写入者）的其他说法有读出者与写入者（readers and writers）问题以及多读出者-单写入者（readers-writer）锁。（最后一个说法的英文名称中，“readers”有意是复数，“writer”有意是单数，目的是强调这种问题的多个读出者与单个写入者本性。）

> 读写锁的一个日常类比是访问银行账户。多个线程可以同时读出某个账户的收支结余，但是一旦有一个线程需要更新某个给定收支结余，该线程就必须等待所有读出者完成该收支结余的读出，然后只允许该更新线程修改这个收支结余。直到更新完之前，任何读出者都不允许读该收支结余。
>
> 本章中描述的函数由Unix 98定义，因为读写锁不属于1996年Posix.1标准的一部分。这些函数是在1995年由一个称为Aspen Group的Unix厂家联合体开发的，同时开发的还有Posix.1未定义的其他扩充。有一个Posix工作组（1003.1j）正在开发包括读写锁在内的一组Pthread扩充，它们很有可能与本章中讲述的一样。

8.2 获取与释放读写锁

读写锁的数据类型为pthread_rwlock_t。如果这个类型的某个变量是静态分配的，那么

可通过给它赋常值PTHREAD_RWLOCK_INITIALIZER来初始化它。

pthread_rwlock_rdlock获取一个读出锁，如果对应的读写锁已由某个写入者持有，那就阻塞调用线程。pthread_rwlock_wrlock获取一个写入锁，如果对应的读写锁已由另一个写入者持有，或者已由一个或多个读出者持有，那就阻塞调用线程。pthread_rwlock_unlock释放一个读出锁或写入锁。

```
#include <pthread.h>
int pthread_rwlock_rdlock(pthread_rwlock_t *rwptr);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwptr);
int pthread_rwlock_unlock(pthread_rwlock_t *rwptr);
                                    均返回：若成功则为0，若出错则为正的Exxx值
```

下面两个函数尝试获取一个读出锁或写入锁，但是如果该锁不能马上取得，那就返回一个EBUSY错误，而不是把调用线程投入睡眠。

```
#include <pthread.h>
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwptr);
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwptr);
                                    均返回：若成功则为0，若出错则为正的Exxx值
```

178

8.3 读写锁属性

我们提到过，可通过给一个静态分配的读写锁赋常值PTHREAD_RWLOCK_INITIALIZER来初始化它。读写锁变量也可以通过调用pthread_rwlock_init来动态地初始化。当一个线程不再需要某个读写锁时，可以调用pthread_rwlock_destroy摧毁它。

```
#include <pthread.h>
int pthread_rwlock_init(pthread_rwlock_t *rwptr,
                        const pthread_rwlockattr_t *attr);
int pthread_rwlock_destroy(pthread_rwlock_t *rwptr);
                                    均返回：若成功则为0，若出错则为正的Exxx值
```

初始化某个读写锁时，如果*attr*是个空指针，那就使用默认属性。要赋予它非默认的属性，需使用下面两个函数。

```
#include <pthread.h>
int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
                                    均返回：若成功则为0，若出错则为正的Exxx值
```

数据类型为pthread_rwlockattr_t的某个属性对象一旦初始化，就通过调用不同的函数来启用或禁止特定属性。当前定义了的唯一属性是PTHREAD_PROCESS_SHARED，它指定相应的读写锁将在不同进程间共享，而不仅仅是在单个进程内的不同线程间共享。以下两个函数分别获取和设置这个属性。

```
#include <pthread.h>
int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *attr, int *valptr);
int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr, int value);
                                        均返回：若成功则为0，若出错则为正的Exxx值
```

第一个函数在由*valptr*指向的整数中返回该属性的当前值。第二个函数把该属性的当前值设置为*value*，其值或为PTHREAD_PROCESS_PRIVATE，或为PTHREAD_PROCESS_SHARED。

8.4　使用互斥锁和条件变量实现读写锁

179 只需使用互斥锁和条件变量就能实现读写锁。本节中我们将查看一种可能的实现。这个实现优先考虑等待着的写入者。这不是必需的，可以有其他实现方案。

> 本节和本章剩余各节含有高级主题，第一次阅读时你可暂时跳过去。
>
> 读写锁的其他实现也值得研究。[Butenhof 1997]的7.1.2节提供了一个优先考虑等待着的读出者的实现，并且包括取消处理（我们将稍后讨论）。[IEEE1996]的B.18.2.3.1节提供了另一个优先考虑等待着的写入者的实现，也包括取消处理。[Kleiman, Shah, and Smaalders 1996]第14章提供了一个优先考虑等待着的写入者的实现。本节给出的实现来自Doug Schmidt的ACE软件包http://www.cs.wustl.edu/~schmidt/ACE.html（适应性通信环境，Adaptive Communications Environment）。以上4个实现都使用互斥锁和条件变量。

8.4.1　pthread_rwlock_t 数据类型

图8-1给出我们的pthread_rwlock.h头文件，它定义了基本的pthread_rwlock_t数据类型和操作读写锁的各个函数的函数原型。通常情况下它们是在<pthread.h>头文件中。

my_rwlock/pthread_rwlock.h

```
 1 #ifndef __pthread_rwlock_h
 2 #define __pthread_rwlock_h

 3 typedef struct {
 4     pthread_mutex_t rw_mutex;      /* basic lock on this struct */
 5     pthread_cond_t rw_condreaders;     /* for reader threads waiting */
 6     pthread_cond_t rw_condwriters;     /* for writer threads waiting */
 7     int     rw_magic;              /* for error checking */
 8     int     rw_nwaitreaders;       /* the number waiting */
 9     int     rw_nwaitwriters;       /* the number waiting */
10     int     rw_refcount;
11         /* -1 if writer has the lock, else # readers holding the lock */
12 } pthread_rwlock_t;

13 #define RW_MAGIC    0x19283746

14         /* following must have same order as elements in struct above */
15 #define PTHREAD_RWLOCK_INITIALIZER{ PTHREAD_MUTEX_INITIALIZER, \
16             PTHREAD_COND_INITIALIZER, PTHREAD_COND_INITIALIZER, \
17             RW_MAGIC, 0, 0, 0 }

18 typedef int pthread_rwlockattr_t;      /* dummy; not supported */

19         /* function prototypes */
```

图8-1　pthread_rwlock_t数据类型的定义

```
20 int      pthread_rwlock_destroy(pthread_rwlock_t *);
21 int      pthread_rwlock_init(pthread_rwlock_t *, pthread_rwlockattr_t *);
22 int      pthread_rwlock_rdlock(pthread_rwlock_t *);
23 int      pthread_rwlock_tryrdlock(pthread_rwlock_t *);
24 int      pthread_rwlock_trywrlock(pthread_rwlock_t *);
25 int      pthread_rwlock_unlock(pthread_rwlock_t *);
26 int      pthread_rwlock_wrlock(pthread_rwlock_t *);
27         /* and our wrapper functions */
28 void     Pthread_rwlock_destroy(pthread_rwlock_t *);
29 void     Pthread_rwlock_init(pthread_rwlock_t *, pthread_rwlockattr_t *);
30 void     Pthread_rwlock_rdlock(pthread_rwlock_t *);
31 int      Pthread_rwlock_tryrdlock(pthread_rwlock_t *);
32 int      Pthread_rwlock_trywrlock(pthread_rwlock_t *);
33 void     Pthread_rwlock_unlock(pthread_rwlock_t *);
34 void     Pthread_rwlock_wrlock(pthread_rwlock_t *);

35 #endif   /* __pthread_rwlock_h */
```

my_rwlock/pthread_rwlock.h

图8-1（续）

3~13 我们的pthread_rwlock_t数据类型含有一个互斥锁、两个条件变量、一个标志及三个计数器。我们将从接下来给出的函数中看出所有这些成员的用途。无论何时检查或操纵该结构，我们都必须持有其中的互斥锁成员rw_mutex。该结构初始化成功后，标志成员rw_magic就被设置成RW_MAGIC。所有函数都测试该成员，以检查调用者是否向某个已初始化的读写锁传递了指针。该读写锁被摧毁时，这个成员就被置为0。

注意计数器成员之一rw_refcount总是指示着本读写锁的当前状态：-1表示它是一个写入锁（任意时刻这样的锁只能有一个），0表示它是可用的，大于0的值则意味着它当前容纳着那么多的读出锁。

14~17 给该数据类型定义静态初始化常值。

8.4.2 pthread_rwlock_init 函数

图8-2给出了我们的第一个函数pthread_rwlock_init，它动态初始化一个读写锁。

my_rwlock/pthread_rwlock_init.c

```
1 #include    "unpipc.h"
2 #include    "pthread_rwlock.h"

3 int
4 pthread_rwlock_init(pthread_rwlock_t *rw, pthread_rwlockattr_t *attr)
5 {
6     int     result;

7     if (attr != NULL)
8         return(EINVAL);  /* not supported */

9     if ( (result = pthread_mutex_init(&rw->rw_mutex, NULL)) != 0)
10        goto err1;
11    if ( (result = pthread_cond_init(&rw->rw_condreaders, NULL)) != 0)
12        goto err2;
13    if ( (result = pthread_cond_init(&rw->rw_condwriters, NULL)) != 0)
14        goto err3;
15    rw->rw_nwaitreaders = 0;
```

图8-2 pthread_rwlock_init函数：初始化一个读写锁

```
16      rw->rw_nwaitwriters = 0;
17      rw->rw_refcount = 0;
18      rw->rw_magic = RW_MAGIC;

19      return(0);

20    err3:
21      pthread_cond_destroy(&rw->rw_condreaders);
22    err2:
23      pthread_mutex_destroy(&rw->rw_mutex);
24    err1:
25      return(result);       /* an errno value */
26  }
```

my_rwlock/pthread_rwlock_init.c

图8-2（续）

7~8　我们不支持使用本函数给读写锁赋属性，因此检查其attr是否为一个空指针。

9~19　初始化由调用者指定其指针的读写锁结构中的互斥锁和两个条件变量成员。所有三个计数器成员都设置为0，rw_magic成员则设置为表示该结构已初始化完毕的值。

20~25　如果互斥锁或条件变量的初始化失败，那么小心地确保摧毁已初始化的对象，然后返回一个错误。

8.4.3　pthread_rwlock_destroy 函数

图8-3给出了我们的pthread_rwlock_destroy函数，它在所有线程（包括调用者在内）都不再持有也不试图持有某个读写锁的时候摧毁该锁。

my_rwlock/pthread_rwlock_destroy.c

```
 1 #include    "unpipc.h"
 2 #include    "pthread_rwlock.h"

 3 int
 4 pthread_rwlock_destroy(pthread_rwlock_t *rw)
 5 {
 6     if (rw->rw_magic != RW_MAGIC)
 7         return(EINVAL);
 8     if (rw->rw_refcount != 0 ||
 9         rw->rw_nwaitreaders != 0 || rw->rw_nwaitwriters != 0)
10         return(EBUSY);

11     pthread_mutex_destroy(&rw->rw_mutex);
12     pthread_cond_destroy(&rw->rw_condreaders);
13     pthread_cond_destroy(&rw->rw_condwriters);
14     rw->rw_magic = 0;

15     return(0);
16 }
```

my_rwlock/pthread_rwlock_destroy.c

图8-3　pthread_rwlock_destroy函数：摧毁一个读写锁

8~13　首先检查由调用者指定的读写锁已不在使用中，然后给其中的互斥锁和两个条件变量成员调用合适的摧毁函数。

8.4.4　pthread_rwlock_rdlock 函数

图8-4给出了我们的pthread_rwlock_rdlock函数。

my_rwlock/pthread_rwlock_rdlock.c

```
 1 #include    "unpipc.h"
 2 #include    "pthread_rwlock.h"

 3 int
 4 pthread_rwlock_rdlock(pthread_rwlock_t *rw)
 5 {
 6     int     result;

 7     if (rw->rw_magic != RW_MAGIC)
 8         return(EINVAL);

 9     if ( (result = pthread_mutex_lock(&rw->rw_mutex)) != 0)
10         return(result);

11         /* give preference to waiting writers */
12     while (rw->rw_refcount < 0 || rw->rw_nwaitwriters > 0) {
13         rw->rw_nwaitreaders++;
14         result = pthread_cond_wait(&rw->rw_condreaders, &rw->rw_mutex);
15         rw->rw_nwaitreaders--;
16         if (result != 0)
17             break;
18     }
19     if (result == 0)
20         rw->rw_refcount++;          /* another reader has a read lock */

21     pthread_mutex_unlock(&rw->rw_mutex);
22     return (result);
23 }
```

my_rwlock/pthread_rwlock_rdlock.c

图8-4 pthread_rwlock_rdlock函数：获取一个读出锁

9~10 无论何时操作pthread_rwlock_t类型的结构，都必须给其中的rw_mutex成员上锁。

11~18 如果（a）rw_refcount小于0（意味着当前有一个写入者持有由调用者指定的读写锁），或者（b）有线程正等着获取该读写锁的一个写入锁（rw_nwaitwriters大于0），那么我们无法获取该读写锁的一个读出锁。如果这两个条件中有一个为真，我们就把rw_nwaitreaders加1。并在rw_condreaders条件变量上调用pthread_cond_wait。我们稍后将看到，当给一个读写锁解锁时，首先检查是否有任何等待着的写入者，若没有则检查是否有任何等待着的读出者。如果有读出者在等待，那就向rw_condreaders条件变量广播信号。

19~20 取得读出锁后把rw_refcount加1。互斥锁旋即释放。

> 该函数中存在一个问题：如果调用线程阻塞在其中的pthread_cond_wait调用上并随后被取消，它就在仍持有互斥锁的情况下终止，于是rw_nwaitreaders计数器的值出错。图8-6中pthread_rwlock_wrlock函数的实现也存在同样的问题。我们将在8.5节纠正这些问题。

8.4.5 pthread_rwlock_tryrdlock函数

图8-5给出我们的pthread_rwlock_tryrdlock函数，它在尝试获取一个读出锁时并不阻塞。

11~14 如果当前有一个写入者持有调用者指定的读写锁，或者有线程在等待该读写锁的一个写入锁，那就返回EBUSY错误。否则，通过把rw_refcount加1获取该读写锁。

my_rwlock/pthread_rwlock_tryrdlock.c

```
 1 #include    "unpipc.h"
 2 #include    "pthread_rwlock.h"

 3 int
 4 pthread_rwlock_tryrdlock(pthread_rwlock_t *rw)
 5 {
 6     int     result;

 7     if (rw->rw_magic != RW_MAGIC)
 8         return(EINVAL);

 9     if ( (result = pthread_mutex_lock(&rw->rw_mutex)) != 0)
10         return(result);

11     if (rw->rw_refcount < 0 || rw->rw_nwaitwriters > 0)
12         result = EBUSY;           /* held by a writer or waiting writers */
13     else
14         rw->rw_refcount++;        /* increment count of reader locks */

15     pthread_mutex_unlock(&rw->rw_mutex);
16     return(result);
17 }
```

my_rwlock/pthread_rwlock_tryrdlock.c

图8-5　pthread_rwlock_tryrdlock函数：试图获取一个读出锁

8.4.6　pthread_rwlock_wrlock 函数

图8-6给出了我们的pthread_rwlock_wrlock函数。

my_rwlock/pthread_rwlock_wrlock.c

```
 1 #include    "unpipc.h"
 2 #include    "pthread_rwlock.h"

 3 int
 4 pthread_rwlock_wrlock(pthread_rwlock_t *rw)
 5 {
 6     int     result;

 7     if (rw->rw_magic != RW_MAGIC)
 8         return(EINVAL);

 9     if ( (result = pthread_mutex_lock(&rw->rw_mutex)) != 0)
10         return(result);

11     while (rw->rw_refcount != 0) {
12         rw->rw_nwaitwriters++;
13         result = pthread_cond_wait(&rw->rw_condwriters, &rw->rw_mutex);
14         rw->rw_nwaitwriters--;
15         if (result != 0)
16             break;
17     }
18     if (result == 0)
19         rw->rw_refcount = -1;

20     pthread_mutex_unlock(&rw->rw_mutex);
21     return(result);
22 }
```

my_rwlock/pthread_rwlock_wrlock.c

图8-6　pthread_rwlock_wrlock函数：获取一个写入锁

11~17　只要有读出者持有由调用者指定的读写锁的读出锁，或者有一个写入者持有该读写锁

的唯一写入锁（两者都是rw_refcount不为0的情况），调用线程就得阻塞。为此，我们把rw_nwaitwriters加1，然后在rw_condwriters条件变量上调用pthread_cond_wait。我们将看到，向该条件变量发送信号的前提是：它所在的读写锁被释放，并且有写入者正在等待。

18~19 取得写入锁后把rw_refcount置为-1。

8.4.7 pthread_rwlock_trywrlock 函数

图8-7给出了非阻塞版本的pthread_rwlock_trywrlock函数。

my_rwlock/pthread_rwlock_trywrlock.c

```
 1 #include    "unpipc.h"
 2 #include    "pthread_rwlock.h"

 3 int
 4 pthread_rwlock_trywrlock(pthread_rwlock_t *rw)
 5 {
 6     int     result;

 7     if (rw->rw_magic != RW_MAGIC)
 8         return(EINVAL);

 9     if ( (result = pthread_mutex_lock(&rw->rw_mutex)) != 0)
10         return(result);

11     if (rw->rw_refcount != 0)
12         result = EBUSY;       /* held by either writer or reader(s) */
13     else
14         rw->rw_refcount = -1; /* available, indicate a writer has it */

15     pthread_mutex_unlock(&rw->rw_mutex);
16     return(result);
17 }
```

my_rwlock/pthread_rwlock_trywrlock.c

图8-7 pthread_rwlock_trywrlock函数：试图获取一个写入锁

11~14 如果rw_refcount不为0，那么由调用者指定的读写锁或者由一个写入者持有，或者由一个或多个读出者持有（至于由哪个持有则无关紧要），因而返回一个EBUSY错误。否则，获取该读写锁的写入锁，并把rw_refcount置为-1。

8.4.8 pthread_rwlock_unlock 函数

图8-8给出了我们的最后一个函数pthread_rwlock_unlock。

11~16 如果rw_refcount当前大于0，那么有一个读出者（即调用线程）准备释放一个读出锁。如果rw_refcount当前为-1，那么有一个写入者（即调用线程）准备释放一个写入锁。

17~22 如果有一个写入者在等待，那么一旦由调用者指定的读写锁变得可用（也就是说它的引用计数变为0），就向rw_condwriters条件变量发送信号。我们知道只有一个写入者能够获取该读写锁，因此调用pthread_cond_signal来唤醒一个线程。如没有写入者在等待，但是有一个或多个读出者在等待，那就在rw_condreaders条件变量上调用pthread_ cond_broadcast，因为所有等待着的读出者都可以获取一个读出锁。注意，一旦有一个写入者在等待，我们就不给任何读出者授予读出锁，否则一个持续的读请求流可能永远阻塞某个等待着的写入者。由于这个原因，我们需要两个分开的if条件测试，而不能写成：

```
    /* give preference to waiting writers over waiting readers */
if (rw->rw_nwaitwriters > 0 && rw->rw_refcount == 0) {
    result = pthread_cond_signal(&rw->rw_condwriters);
} else if (rw->rw_nwaitreaders > 0)
    result = pthread_cond_broadcast(&rw->rw_condreaders);
```

my_rwlock/pthread_rwlock_unlock.c

```
 1 #include    "unpipc.h"
 2 #include    "pthread_rwlock.h"
 3 int
 4 pthread_rwlock_unlock(pthread_rwlock_t *rw)
 5 {
 6     int     result;
 7     if (rw->rw_magic != RW_MAGIC)
 8         return(EINVAL);
 9     if ( (result = pthread_mutex_lock(&rw->rw_mutex)) != 0)
10         return(result);
11     if (rw->rw_refcount > 0)
12         rw->rw_refcount--;          /* releasing a reader */
13     else if (rw->rw_refcount == -1)
14         rw->rw_refcount = 0;        /* releasing a reader */
15     else
16         err_dump("rw_refcount = %d", rw->rw_refcount);
17         /* give preference to waiting writers over waiting readers */
18     if (rw->rw_nwaitwriters > 0) {
19         if (rw->rw_refcount == 0)
20             result = pthread_cond_signal(&rw->rw_condwriters);
21     } else if (rw->rw_nwaitreaders > 0)
22         result = pthread_cond_broadcast(&rw->rw_condreaders);
23     pthread_mutex_unlock(&rw->rw_mutex);
24     return(result);
25 }
```

my_rwlock/pthread_rwlock_unlock.c

184～186

图8-8　pthread_rwlock_unlock函数：释放一个读出锁或写入锁

我们也可以省略对rw->rw_refcount的测试，不过那会导致在仍分配着读出锁的情况下还调用pthread_cond_signal，从而降低了效率。

8.5　线程取消

我们在随图8-4的说明中暗示了一个问题，即如果pthread_rwlock_rdlock的调用线程阻塞在其中的pthread_cond_wait调用上并随后被取消，它就在仍持有互斥锁的情况下终止。通过由对方调用函数pthread_cancel，一个线程可以被同一进程内的任何其他线程所取消（cancel），pthread_cancel的唯一参数就是待取消线程的线程ID。

```
#include <pthread.h>
int pthread_cancel(pthread_t tid);
                                        返回：若成功则为0，若出错则为正的Exxx值
```

举例来说，如果启动了多个线程以执行某个给定任务（譬如说在某个数据库中查找一个记录），那么首先完成任务的线程可使用线程取消功能取消其他线程。另一个例子是，当多个线程开始执行同一个任务时，如果其中某个线程发现一个错误，它和其他线程就有必要终止。

为处理被取消的可能情况，任何线程可以安装（压入）和删除（弹出）清理处理程序。

```
#include <pthread.h>
void pthread_cleanup_push(void (*function)(void *), void *arg);
void pthread_cleanup_pop(int execute);
```

这些处理程序就是发生以下情况时被调用的函数：

- 调用线程被取消（由某个线程调用pthread_cancel完成）；
- 调用线程自愿终止（或者通过调用pthread_exit，或者从自己的线程起始函数返回）。

清理处理程序可以恢复任何需要恢复的状态，例如给调用线程当前持有的任何互斥锁或信号量解锁。

pthread_cleanup_push的*function*参数是调用线程被取消时所调用的函数的地址，*arg*是它的单个参数。pthread_cleanup_pop总是删除调用线程的取消清理栈中位于栈顶的函数，而且如果*execute*不为0，那就调用该函数。

我们将随图15-31再次遇到线程取消，那时我们将看到，在某个过程调用正在处理期间如果客户终止，门服务器就被取消。

187

例子

说明上一节中我们的实现所存在问题的最简易方法是给出例子。图8-9给出了测试程序的时间线图，图8-10给出了程序本身。

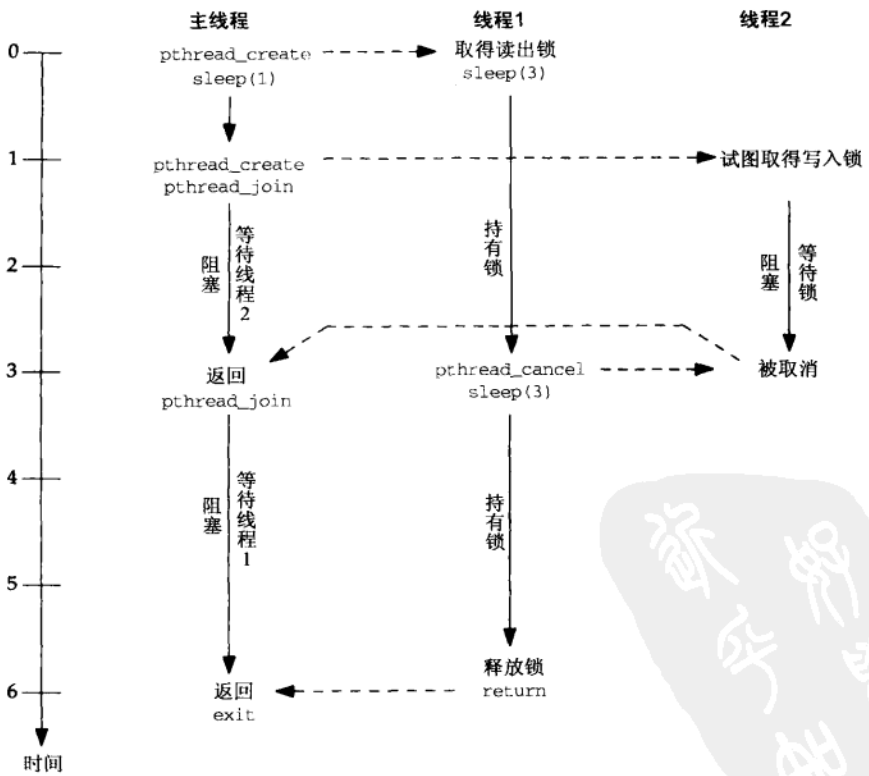


图8-9 图8-10中程序的时间线图

my_rwlock_cancel/testcancel.c

```
1 #include    "unpipc.h"
2 #include    "pthread_rwlock.h"

3 pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;
4 pthread_t tid1, tid2;
5 void   *thread1(void *), *thread2(void *);

6 int
7 main(int argc, char **argv)
8 {
9     void    *status;

10    Set_concurrency(2);
11    Pthread_create(&tid1, NULL, thread1, NULL);
12    sleep(1);                   /* let thread1() get the lock */
13    Pthread_create(&tid2, NULL, thread2, NULL);

14    Pthread_join(tid2, &status);
15    if (status != PTHREAD_CANCELED)
16        printf("thread2 status = %p\n", status);
17    Pthread_join(tid1, &status);
18    if (status != NULL)
19        printf("thread1 status = %p\n", status);

20    printf("rw_refcount = %d, rw_nwaitreaders = %d, rw_nwaitwriters = %d\n",
21           rwlock.rw_refcount, rwlock.rw_nwaitreaders,
22           rwlock.rw_nwaitwriters);
23    Pthread_rwlock_destroy(&rwlock);

24    exit(0);
25 }

26 void *
27 thread1(void *arg)
28 {
29    Pthread_rwlock_rdlock(&rwlock);
30    printf("thread1() got a read lock\n");
31    sleep(3);                   /* let thread2 block in pthread_rwlock_wrlock() */
32    pthread_cancel(tid2);
33    sleep(3);
34    Pthread_rwlock_unlock(&rwlock);
35    return(NULL);
36 }

37 void *
38 thread2(void *arg)
39 {
40    printf("thread2() trying to obtain a write lock\n");
41    Pthread_rwlock_wrlock(&rwlock);
42    printf("thread2() got a write lock\n");     /* should not get here */
43    sleep(1);
44    Pthread_rwlock_unlock(&rwlock);
45    return(NULL);
46 }
```

my_rwlock_cancel/testcancel.c

图8-10　展示线程取消的测试程序

创建两个线程

10~13　创建两个进程，第一个线程执行函数thread1，第二个线程执行函数thread2。创建第一个线程后睡眠1秒，以允许它获取一个读出锁。

等待线程终止

14~23 首先等待第二个线程，并验证其状态为PTHREAD_CANCEL。接着待等第一个线程，并验证其状态为一个空指针。然后输出pthread_rwlock_t类型读写锁结构中两个计数器成员的值，最后摧毁该读写锁。

thread1函数

26~36 第一个线程获取一个读出锁后睡眠3秒。这个停顿允许另一个线程（第二个线程）调用pthread_rwlock_wrlock并阻塞在其中的pthread_cond_wait调用中，因为在有一个读出锁活跃期间，是无法提供写入锁的。本线程然后调用pthread_cancel取消另一个线程，再睡眠3秒后释放所持有的读出锁，然后终止。

thread2函数

37~46 第二个线程试图获取一个写入锁（这是不可能取得的，因为第一个线程已经获取了一个读出锁）。本函数的其余部分不应该被执行。

如果使用上一节中给出的函数运行本测试程序，那么我们将得到如下结果：

```
solaris % testcancel
thread1() got a read lock
thread2() trying to obtain a write lock
```

而且肯定不返回shell提示符状态。该程序被挂起。发生如下步骤。

(1) 第二个线程调用pthread_rwlock_wrlock（图8-6），阻塞在其中的pthread_cond_wait调用中。

(2) 第一个线程中的sleep(3)返回，pthread_cancel被接着调用。

(3) 第二个线程被取消（这儿就是被终止）。当阻塞在某个条件变量等待中的一个线程被取消时，要再次取得与该条件变量关联的互斥锁，然后调用第一个线程取消清理处理程序。（我们尚未安装任何线程取消清理处理程序，但是所关联的互斥锁仍然在该线程被取消前再次取得。）因此，当第二个线程被取消时，它持有包含在读写锁中的互斥锁，而且图8-6中rw_nwaitwriters的值已被加1。

(4) 第一个线程调用pthread_rwlock_unlcok，但它永远阻塞在其中的pthread_mutex_lock调用中（图8-8），因为它想要持有的互斥锁仍然由已被取消的线程锁着。

要是我们去掉thread1函数中的pthread_rwlock_unlock调用，那么主线程将输出如下：

```
rw_refcount = 1, rw_nwaitreaders = 0, rw_nwaitwriters = 1
pthread_rwlock_destroy error: Device busy
```

第一个计数器是1，因为我们删除了pthread_rwlock_unlock调用，但是最后一个计数器也是1，因为它是由第二个线程在被取消前加1了的计数器。

这一问题纠正起来很简单。首先，给图8-4中的pthread_rwlock_rdlock函数增加两行代码（前有加号指示），它们把pthread_cond_wait调用括了起来：

```
      rw->rw_nwaitreaders++;
+     prhread_cleanup_push(rwlock_cancelrdwait, (void *) rw);
      result = pthread_cond_wait(&rw->rw_condreaders, &rw->rw_mutex);
+     pthread_cleanup_pop(0);
      rw->rw_nwaitreaders--;
```

第一行新代码建立一个清理处理程序（我们的rwlock_cancelrdwait函数），它的单个参数将是读写锁指针rw。如果pthread_cond_wait返回，第二行新代码就删除这个清理处理程序。pthread_cleanup_pop的值为0的单个参数指示不调用该处理程序。要是该参数不为0，那就先调用这个清理处理程序再删除它。

188 ~ 190

如果pthread_rwlock_rdlock的调用线程在阻塞于该函数中的pthread_cond_wait调用期间被取消，它就不会从该函数返回，而是会调用清理处理程序（在重新获取所关联的互斥锁之后，我们已在前面的第3步中提到过这一点）。

图8-11给出了我们的rwlock_cancelrdwait函数，它是我们为pthread rwlock_rdlock建立的清理处理程序。

my_rwlock_cancel/pthread_rwlock_rdlock.c

```
 3 static void
 4 rwlock_cancelrdwait(void *arg)
 5 {
 6     pthread_rwlock_t *rw;

 7     rw = arg;
 8     rw->rw_nwaitreaders--;
 9     pthread_mutex_unlock(&rw->rw_mutex);
10 }
```

my_rwlock_cancel/pthread_rwlock_rdlock.c

图8-11　rwlock_cancelrdwait函数：读出锁的清理处理程序

8~9　把rw_nwaitreaders计数器减1，并给互斥锁解锁。这是在调用pthread_cond_wait前建立的“状态”，其调用线程被取消时必须恢复到该状态。

对图8-6中pthread_rwlock_wrlock函数进行类似的修正。首先，在pthread_cond_wait调用前后各增加一行新代码：

```
    rw->rw_nwaitwriters++;
+   prhread_cleanup_push(rwlock_cancelwrwait, (void *) rw);
    result = pthread_cond_wait(&rw->rw_condwriters, &rw->rw_mutex);
+   pthread_cleanup_pop(0);
    rw->rw_nwaitwriters--;
```

图8-12给出了我们的rwlcok_cancelwrwait函数，它是清理写入锁请求的清理处理程序。

my_rwlock_cancel/pthread_rwlock_wrlock.c

```
 3 static void
 4 rwlock_cancelwrwait(void *arg)
 5 {
 6     pthread_rwlock_t *rw;

 7     rw = arg;
 8     rw->rw_nwaitwriters--;
 9     pthread_mutex_unlock(&rw->rw_mutex);
10 }
```

my_rwlock_cancel/pthread_rwlock_wrlock.c

图8-12　rwlock_cancelwrwait函数：写入锁的清理处理程序

8~9　给rw_nwaitwriters计数器减1，并给互斥锁解锁。

用这些新函数运行图8-10中的测试程序，结果是正确的：

```
solaris % testcancel
thread1() got a read lock
thread2() trying to obtain a write lock
rw_refcount = 0, rw_nwaitreaders = 0, rw_nwaitwriters = 0
```

三个计数器的值都是正确的，thread1从它的pthread_rwlock_unlock调用返回，而且pthread_rwlock_destroy不返回EBUSY错误。

本节仅仅是线程取消的一个概貌。它还有许多细节，参见［Butenhof 1997］的5.3节。

8.6 小结

与普通的互斥相比，当被保护数据的读访问比写访问更为频繁时，读写锁能提供更高的并发度。本章讲述的由Unix 98定义的读写锁函数或类似函数应出现在某个未来的Posix标准中。这些函数与第7章讲述的互斥锁函数类似。

读写锁可以只通过使用互斥锁和条件变量来实现，我们给出了一个实现例子。这个实现优先考虑等待着的写入者，但是有的实现优先考虑等待着的读出者。

线程可能在阻塞于pthread_cond_wait调用期间被取消，我们的读写锁实现允许看到这种情况的发生。我们使用线程取消清理处理程序解决了这个问题。

习题

8.1 修改8.4节中我们的读写锁实现，优先考虑读出者而不是写入者。

8.2 度量并比较8.4节中我们的读写锁实现和厂家提供的实现的性能。

191 ~ 192

第9章 记录上锁

9.1 概述

上一章讲述的读写锁是作为pthread_rwlock_t数据类型的变量在内存中分配的。当读写锁是在单个进程内的各个线程间共享时（默认情况），这些变量可以在那个进程内；当读写锁是在共享某个内存区的进程间共享时（假设初始化它们时指定了PTHREAD_PROCESS_SHARED属性），这些变量应该在该共享内存区中。

本章讲述读写锁的一种扩展类型，它可用于有亲缘关系或无亲缘关系的进程之间共享某个文件的读与写。被锁住的文件通过其描述符访问，执行上锁操作的函数是fcntl。这种类型的锁通常在内核中维护，其属主是由属主的进程ID标识的。这意味着这些锁用于不同进程间的上锁，而不是用于同一进程内不同线程间的上锁。

我们将在本章介绍序列号持续加1的例子。考虑来自Unix打印假脱机处理系统（BSD下使用lpr命令访问，System V下使用lp命令访问）的下述情形。把一个打印作业加到打印队列中（供另一个进程在以后某个时候打印）的进程必须给每个作业赋一个唯一的序列号。只是在该进程运行期间唯一的进程ID不能用作这个序列号，因为一个打印作业可能存在很长时间，期间早先把它加到打印队列中的进程的进程ID可能被重用。另外，一个给定进程可以往某个队列中加入多个打印作业，而每个作业都需要一个唯一的作业号。打印假脱机处理系统使用的技巧是：给每台打印机准备一个文件，它是只有一个单行的ASCII文本文件，其中含有待用的下一个序列号。需要给某个打印作业赋一个序列号的每个进程都得经历以下三个步骤。

193

(1) 读序列号文件。

(2) 使用其中的序列号。

(3) 给序列号加1并写回文件中。

问题是当某个进程在执行这三个步骤时，另一个进程可能在执行同样的三个步骤。这将导致混乱，如我们将在后面的一些例子中看到的那样。

> 我们刚刚叙述的是一个互斥问题。它可使用第7章讲述的互斥锁或第8章讲述的读写锁来解决。然而不同的是，我们假设各个进程彼此无亲缘关系，从而让使用这些技巧更为困难。我们可以让这些进程共享某个内存区（如本书第4部分所述），然后在该共享内存区中使用某种类型的同步变量，不过对于无亲缘关系的进程，fcntl记录上锁往往更易使用。另一个因素是，我们随行式打印机假脱机处理系统描述的问题，在互斥锁、条件变量和读写锁的可用之前许多年就存在。记录上锁是在20世纪80年代早期加到Unix中的，先于共享内存区和线程的开发。

我们所需的是：一个进程能够设置某个锁，以宣称没有其他进程能够访问相应的文件，直到第一个进程完成访问为止。图9-2给出了执行上述三个步骤的一个简单程序。函数my_lock和

my_unlock分别用于刚开始时给序列号文件上锁以及完成序列号更新时给该文件解锁。我们将给出这两个函数的多种实现。

20 每次循环输出序列号时同时输出正在运行的程序的名字（argv[0]），因为这个main函数与不同版本的上锁函数一块使用，而我们希望看到哪个版本在输出序列号。

输出进程ID需要把类型为pid_t的变量强制转换成long类型，然后使用%ld格式化串输出。其原因是，尽管pid_t是一个整数类型，但我们不知道它的大小（int或long），因此必须假设成最大的类型。要是我们假设它是int类型并使用%d格式化串，但是实际类型却为long，那么代码是错误的。

为展示不上锁的后果，图9-1提供了根本不上锁的两个“上锁”函数。

lock/locknone.c

```
1 void
2 my_lock(int fd)
3 {
4     return;
5 }

6 void
7 my_unlock(int fd)
8 {
9     return;
10 }
```

lock/locknone.c

194

图9-1 不上锁的函数

如果序列号文件中的序列号初始化为1，而且该程序只有一个副本在运行，那么结果如下：

```
solaris % locknone
locknone: pid = 15491, seq# = 1
locknone: pid = 15491, seq# = 2
locknone: pid = 15491, seq# = 3
locknone: pid = 15491, seq# = 4
locknone: pid = 15491, seq# = 5
locknone: pid = 15491, seq# = 6
locknone: pid = 15491, seq# = 7
locknone: pid = 15491, seq# = 8
locknone: pid = 15491, seq# = 9
locknone: pid = 15491, seq# = 10
locknone: pid = 15491, seq# = 11
locknone: pid = 15491, seq# = 12
locknone: pid = 15491, seq# = 13
locknone: pid = 15491, seq# = 14
locknone: pid = 15491, seq# = 15
locknone: pid = 15491, seq# = 16
locknone: pid = 15491, seq# = 17
locknone: pid = 15491, seq# = 18
locknone: pid = 15491, seq# = 19
locknone: pid = 15491, seq# = 20
```

注意main函数（图9-2）是在一个名为lockmain.c的文件中，但是当我们将它与不执行上锁的“上锁”函数（图9-1）一同编译和链接时，称结果的可执行文件为locknone。这是因为我们将提供my_lock和my_unlock这两个函数的其他实现，它们使用不同的上锁技巧，因此我们根据所用的上锁类型命名可执行文件。

lock/lockmain.c

```
1 #include    "unpipc.h"
2 #define SEQFILE "seqno"         /* filename */
3 void    my_lock(int), my_unlock(int);
4 int
5 main(int argc, char **argv)
6 {
7     int     fd;
8     long    i, seqno;
9     pid_t   pid;
10    ssize_t n;
11    char    line[MAXLINE + 1];
12    pid = getpid();
13    fd = Open(SEQFILE, O_RDWR, FILE_MODE);
14    for (i = 0; i < 20; i++) {
15        my_lock(fd);              /* lock the file */
16        Lseek(fd, 0L, SEEK_SET);  /* rewind before read */
17        n = Read(fd, line, MAXLINE);
18        line[n] = '\0';           /* null terminate for sscanf */
19        n = sscanf(line, "%ld\n", &seqno);
20        printf("%s: pid = %ld, seq# = %ld\n", argv[0], (long) pid, seqno);
21        seqno++;                  /* increment sequence number */
22        snprintf(line, sizeof(line), "%ld\n", seqno);
23        Lseek(fd, 0L, SEEK_SET);     /* rewind before write */
24        Write(fd, line, strlen(line));
25        my_unlock(fd);            /* unlock the file */
26    }
27    exit(0);
28 }
```

lock/lockmain.c

图9-2　文件上锁例子的main函数

把序列号重新初始化为1，然后在后台运行该程序两次，其结果如下：

```
solaris % locknone & locknone &
solaris % locknone: pid = 15498, seq# = 1
locknone: pid = 15498, seq# = 2
locknone: pid = 15498, seq# = 3
locknone: pid = 15498, seq# = 4
locknone: pid = 15498, seq# = 5
locknone: pid = 15498, seq# = 6
locknone: pid = 15498, seq# = 7
locknone: pid = 15498, seq# = 8
locknone: pid = 15498, seq# = 9
locknone: pid = 15498, seq# = 10
locknone: pid = 15498, seq# = 11
locknone: pid = 15498, seq# = 12
locknone: pid = 15498, seq# = 13
locknone: pid = 15498, seq# = 14
locknone: pid = 15498, seq# = 15
locknone: pid = 15498, seq# = 16
locknone: pid = 15498, seq# = 17
locknone: pid = 15498, seq# = 18
locknone: pid = 15498, seq# = 19
```

```
locknone: pid = 15498, seq# = 20          到本行为止一切正常
locknone: pid = 15499, seq# = 1           内核切换进程后开始出错
locknone: pid = 15499, seq# = 2
locknone: pid = 15499, seq# = 3
locknone: pid = 15499, seq# = 4
locknone: pid = 15499, seq# = 5
locknone: pid = 15499, seq# = 6
locknone: pid = 15499, seq# = 7
locknone: pid = 15499, seq# = 8
locknone: pid = 15499, seq# = 9
locknone: pid = 15499, seq# = 10
locknone: pid = 15499, seq# = 11
locknone: pid = 15499, seq# = 12
locknone: pid = 15499, seq# = 13
locknone: pid = 15499, seq# = 14
locknone: pid = 15499, seq# = 15
locknone: pid = 15499, seq# = 16
locknone: pid = 15499, seq# = 17
locknone: pid = 15499, seq# = 18
locknone: pid = 15499, seq# = 19
locknone: pid = 15499, seq# = 20
```

195 ~ 196

我们首先注意到shell提示符在程序输出第一行之前输出。这是正常的，而且在后台运行程序时经常见到。

前20行输出是正常的，它们由该程序的第一个实例（进程ID为15498）输出。但是该程序另一个实例（进程ID为15499）的第一行输出中却出现了问题：它输出一个值为1的序列号，表明它也许是由内核第一个启动，当它读完序列号文件（序列号值为1）后，内核切换另一个进程来运行。该进程直到另一个进程终止时才再次运行，它继续执行所用的值是在内核切换进程前已读出的值1。这不是我们所希望的。每个进程读出、加1然后写入序列号文件20次（从而恰好有40行输出），因此序列号的最终值应该是40。

我们需要某种方法以允许一个进程在执行前述三个步骤期间防止其他进程访问序列号文件。这就是说，考虑到其他进程，这三个步骤应作为一个原子操作（atomic operation）来执行。看这个问题的另一种方式是，图9-2中调用my_lock和调用my_unlock之间的几行代码构成一个临界区（critical region），如第7章中所述。

像刚才所示的那样在后台运行图9-2中程序的两个实例时，其输出是非确定的（nondeterministic）。每次运行该程序的两个实例时，不能保证得到同样的结果。如果早先所列的三个步骤因考虑到其他进程而原子地处理，从而产生40这个最终值，那么结果的不确定性不表示有错。但是如果这三个步骤不是原子地处理，往往会产生一个小于40的最终值，那就不正确了。举例来说，我们并不关心是第一个进程先把序列号从1递增到20，第二个进程接着从21递增到40，还是每个进程都运行刚好足够长时间，从而每次运行把序列号递增2（第一个进程输出1和2，然后第二个进程输出3和4，如此等等）。

非确定性并没有造成不正确。造成程序运行正确与否的是前述三个步骤是否原子地执行。然而非确定性往往使这些类型程序的调试更为困难。

9.2 对比记录上锁与文件上锁

Unix内核没有文件内的记录这一概念。任何关于记录的解释都是由读写文件的应用来进行的。然而Unix内核提供的上锁特性却用记录上锁（record locking）这一术语来描述。不过应用会指定文件中待上锁或解锁部分的字节范围（byte range）。这个字节范围是否跟同一文件内的

197 一个或多个逻辑记录有关联是应用的事。

Posix记录上锁定义了一个特殊的字节范围以指定整个文件，它的起始偏移为0（文件的开头），长度也为0。本章的讨论集中于记录上锁，文件上锁只是它的一个特例。

术语粒度（granularity）用于标记能被锁住的对象的大小。对于Posix记录上锁来说，粒度就是单个字节。通常情况下粒度越小，允许同时使用的用户数就越多。举例来说，假设有五个进程几乎同时访问一个给定的文件，其中三个是读出者，两个是写入者。再假设所有五个进程准备访问该文件中的不同记录，而且这五个请求中的每一个需花的时间几乎相同，譬如说1秒。如果上锁是在文件级别（可能的最粗粒度）上进行的，那么所有三个读出者可以同时访问它们各自的记录，但是那两个写入者必须等到所有读出者完成访问为止。然后其中一个写入者可以修改自己的记录，另一个写入者随后可以这么做。总的时间将大约是3秒。（当然我们在这些定时假设中忽略了许多细节。）但是如果上锁粒度是记录（可能的最细粒度），那么所有五个进程都能同时处理，因为它们各自访问的是不同的记录。于是总的时间只有1秒。

> 源自Berkeley的Unix实现支持给整个文件上锁或解锁的文件上锁（file locking），但没有给文件内的字节范围上锁或解锁的能力。文件上锁由flock函数提供。

历史

多年来Unix下的文件和记录上锁已应用了各种各样的技巧。诸如UUCP守护进程和行式打印机守护进程之类的早期程序所使用的各种技巧充分利用了文件系统实现上的特色。（我们将在9.8节讨论其中三个文件系统技巧。）然而这些技巧使用起来速度比较慢，因此20世纪80年代早期实现的数据库系统提出了使用更好技巧的要求。

第一个真正的文件和记录上锁是由John Bass于1980年加到Version 7中的，新增的一个系统调用名为locking。它提供强制性记录上锁，并为System III和Xenix的许多版本所沿用。（我们将在本章以后说明强制性上锁和劝告性上锁的区别，以及记录上锁和文件上锁的区别。）

4.2BSD于1983年通过flock函数提供了文件上锁（不是记录上锁）。1984年的/usr/group标准（X/Open的前身之一）定义了lockf函数，它只提供独占锁（即写入锁），而没有提供共享锁（即读出锁）。

System V Release 2（SVR2）于1984年通过fcntl函数提供了劝告性记录上锁。lockf函数也提供了，但它只是一个调用fcntl的库函数，而不是系统调用。（许多当前的系统仍然提供使用fcntl完成的lockf实现。）System V Release 3（SVR3）于1986年给fcntl增加了强制性记录上锁能力，它使用了文件的SGID权限位，我们将在9.5节中讨论。

1988年的Posix.1标准对fcntl函数的劝告性文件和记录上锁功能进行了标准化，这就是本章要讲述的内容。X/Open可移植性指南第3期（X/Open Portability Guide Issue 3，简称XPG3，1988
198 年）也指出记录上锁通过fcntl函数提供。

9.3 Posix fcntl 记录上锁

记录上锁的Posix接口是fcntl函数。

```
#include <fcntl.h>
int fcntl(int fd, int cmd, ... /* struct flock *arg */ );
```

返回：若成功则取决于*cmd*，若出错则为-1

用于记录上锁的*cmd*参数共有三个值。这三个命令要求第三个参数*arg*是指向某个flock结构的指针：

```
struct flock {
  short   l_type;     /* F_RDLCK, F_WRLCK, F_UNLCK */
  short   l_whence;   /* SEEK_SET, SEEK_CUR, SEEK_END */
  off_t   l_start;    /* relative starting offset in bytes */
  off_t   l_len;      /* #bytes; 0 means until end-of-file */
  pid_t   l_pid;      /* PID returned by F_GETLK */
};
```

这三个命令如下。

F_SETLK　获取（l_type成员为F_RDLCK或F_WRLCK）或释放（l_type成员为F_UNLCK）由*arg*指向的flock结构所描述的锁。

如果无法将该锁授予调用进程，该函数就立即返回一个EACCES或EAGAIN错误而不阻塞。

F_SETLKW　该命令与上一个命令类似，不过如果无法将所请求的锁授予调用进程，调用线程将阻塞到该锁能够授予为止。（该命令的名字中最后一个字母W意思是“等待（wait）”。）[①]

F_GETLK　检查由*arg*指向的锁以确定是否有某个已存在的锁会妨碍将新锁授予调用进程。如果当前没有这样的锁存在，由*arg*指向的flock结构的l_type成员就被置为F_UNLCK。否则，关于这个已存在锁的信息将在由*arg*指向的flock结构中返回（也就是说，该结构的内容由fcntl函数覆写），其中包括持有该锁的进程的进程ID。[②]

应清楚发出F_GETLK命令后紧接着发出F_SETLK命令不是一个原子操作。这就是说，如果我们发出F_GETLK命令，并且执行该命令的fcntl函数返回时置l_type成员为F_UNLCK，那么跟着立即发出F_SETLK命令不能保证其fcntl函数会成功返回。这两次调用之间可能有另外一个进程运行并获取了我们想要的锁。

提供F_GETLK命令的原因在于：当执行F_SETLK命令的fcntl函数返回一个错误时，导致该错误的某个锁的信息可由F_GETLK命令返回，从而允许我们确定是哪个进程锁住了所请求的文件区，以及上锁方式（读出锁或写入锁）。但是即使是这样的情形，F_GETLK命令也可能返回该文件区已解锁的信息，因为在F_SETLK和F_GETLK命令之间，该文件区可能被解锁。

flock结构描述锁的类型（读出锁或写入锁）以及待锁住的字节范围。跟lseek一样，起始 19

① 一个进程可以对某个文件的特定字节范围多次发出F_SETLK或F_SETLKW命令，每次成功与否取决于其他进程当时是否锁住该字节范围以及锁的类型，而与本进程先前是否锁住该字节范围无关。也就是说，后执行的F_SETLK或F_SETLKW命令覆盖先执行的针对同一字节范围的同样两个命令。另外，文件能否读写与相应的记录是否被其他进程锁住无关（前提是劝告性上锁），前者由文件访问权限完全决定。这就是说，一个进程有可能访问已被另一个进程独占地锁住的文件中的记录，不过彼此协作的进程应自觉地不去执行违反上锁要求的访问。——译者注

② 调用进程已持有的针对同一字节范围的锁不会妨碍它获取新锁，因为同一进程内，后执行的获取锁命令覆盖先执行的命令。举例来说，如果一个进程针对同一字节范围先后执行两个命令：F_SETLK（l_type成员为F_WRLCK）和F_GETLK（l_type成员为F_RDLCK），这两个命令之间无其他进程干扰，而且它们都执行成功，那么由F_GETLK返回的l_type成员是F_UNLCK而不是F_WRLCK。——译者注

字节偏移是作为一个相对偏移（l_start成员）伴随其解释（l_whence成员）指定的。l_whence成员有以下三个取值。

- SEEK_SET：l_start相对于文件的开头解释。
- SEEK_CUR：l_start相对于文件的当前字节偏移（即当前读写指针位置）解释。
- SEEK_END：l_start相对于文件的末尾解释。

l_len成员指定从该偏移开始的连续字节数。长度为0意思是“从起始偏移到文件偏移的最大可能值”。因此，锁住整个文件有两种方式。

(1) 指定l_whence成员为SEEK_SET，l_start成员为0，l_len成员为0。

(2) 使用lseek把读写指针定位到文件头，然后指定l_whence成员为SEEK_CUR，l_start成员为0，l_len成员为0。

第一种方式最常用，因为它只需一个函数调用（fcntl）而不是两个（另见习题9.10）。

fcntl记录上锁既可用于读也可用于写，对于一个文件的任意字节，最多只能存在一种类型的锁（读出锁或写入锁）。而且，一个给定字节可以有多个读出锁，但只能有一个写入锁。这跟我们在上一章讲述的读写锁是一致的。自然，当一个描述符不是打开来用于读时，如果我们对它请求一个读出锁，错误就会发生，同样，当一个描述符不是打开来用于写时，如果我们对它请求一个写入锁，错误也会发生。

对于一个打开着某个文件的给定进程来说，当它关闭该文件的所有描述符或它本身终止时，与该文件关联的所有锁都被删除。[①]锁不能通过fork由子进程继承。

> 在进程终止时由内核完成已有锁清理工作的特性只有fcntl记录上锁完全提供了，System V信号量则把它作为一个选项提供。我们讲述的其他同步技巧（互斥锁、条件变量、读写锁、Posix信号量）并不在进程终止时执行清理工作。我们已在7.7节末尾讨论过这一点。

记录上锁不应该同标准I/O函数库一块使用，因为该函数库会执行内部缓冲。当某个文件需上锁时，为避免问题，应对它使用read和write。

9.3.1 例子

现在回到图9-2中的例子，并把图9-1中的两个函数my_lock和my_unlock重新编写成使用Posix记录上锁。图9-3给出了这些函数。

200

注意，我们必须指定写入锁，以保证任何时刻只有一个进程更新序列号（见习题9.4）。在获取该锁时所指定的命令为F_SETLKW，因为如果该锁不可得，那么我们希望阻塞到它变为可得为止。

> 有了早先给出的flock结构的定义后，有人可能认为在my_lock中可以如下初始化我们的结构：
>
> ```
> static struct flock lock = { F_WRLCK, SEEK_SET, 0, 0, 0 };
> ```
>
> 然而这是错误的。Posix只定义在一个结构（例如flock）中的必需成员。各个实现可以以任意顺序排列这些成员，还可以增设特定于实现的成员。

① 确实如此，甚至于所关闭的描述符先前是在其文件已由本进程（通过该文件的另一个描述符）上锁后才打开也不例外。看来删除锁时关键的是进程ID，而不是引用同一文件的描述符数目及打开目的（只读、只写、读写）。既然锁跟进程ID紧密关联，它不能通过fork由子进程继承也就顺理成章，因为父子进程有不同的进程ID。
——译者注

lock/lockfcntl.c

```
1 #include    "unpipc.h"
2 void
3 my_lock(int fd)
4 {
5     struct flock lock;
6     lock.l_type = F_WRLCK;
7     lock.l_whence = SEEK_SET;
8     lock.l_start = 0;
9     lock.l_len = 0;           /* write lock entire file */
10    Fcntl(fd, F_SETLKW, &lock);
11 }
12 void
13 my_unlock(int fd)
14 {
15    struct flock lock;
16    lock.l_type = F_UNLCK;
17    lock.l_whence = SEEK_SET;
18    lock.l_start = 0;
19    lock.l_len = 0;           /* unlock entire file */
20    Fcntl(fd, F_SETLK, &lock);
21 }
```

lock/lockfcntl.c

图9-3 Posix fcntl上锁

我们不给出结果输出，但它看来是正确的。需认识到运行像图9-2这样的简单程序不足以告诉我们程序是否正常工作。如果输出像我们先前看到的那样是错误的，那么可以断言程序不正确，但是如果只运行它的两个副本，每个副本只循环20次，那么测试是不充分的。内核可能运行一个程序更新序列号20次，再运行另一个程序更新序列号20次。如果这两个进程中途不发生切换，我们就可能永远发现不了错误。更好的测试是：使用另外一个main函数运行图9-3中的函数，这个main函数给序列号加1譬如说1万次，每次循环时不再输出值。如果我们把序列号初始化为1，然后同时运行该程序的20个副本，那么序列号文件的最终值应该是200 001。

9.3.2 例子：简化用的宏

图9-3中，请求或释放一个锁需6行代码。我们必须分配一个结构，填写这个结构，然后调用fcntl。通过定义来自APUE的12.3节的以下7个宏，可以简化我们的程序：

```
#define read_lock(fd, offset, whence, len) \
                lock_reg(fd, F_SETLK, F_RDLCK, offset, whence, len)
#define readw_lock(fd, offset, whence, len) \
                 lock_reg(fd, F_SETLKW, F_RDLCK, offset, whence, len)
#define write_lock(fd, offset, whence, len) \
                 lock_reg(fd, F_SETLK, F_WRLCK, offset, whence, len)
#define writew_lock(fd, offset, whence, len) \
                  lock_reg(fd, F_SETLKW, F_WRLCK, offset, whence, len)
#define un_lock(fd, offset, whence, len) \
               lock_reg(fd, F_SETLK, F_UNLCK, offset, whence, len)

#define is_read_lockable(fd, offset, whence, len) \
                       !lock_test(fd, F_RDLCK, offset, whence, len)
#define is_write_lockable(fd, offset, whence, len) \
                        !lock_test(fd, F_WRLCK, offset, whence, len)
```

这些宏使用我们的lock_reg和lock_test函数，它们在图9-4和图9-5中给出。使用这些宏时，不必考虑flock结构和真正调用的函数。这些宏的前三个参数有意安排成跟lseek函数的前三个参数相同。

lib/lock_reg.c

```
 1 #include    "unpipc.h"

 2 int
 3 lock_reg(int fd, int cmd, int type, off_t offset, int whence, off_t len)
 4 {
 5     struct flock    lock;

 6     lock.l_type = type;       /* F_RDLCK, F_WRLCK, F_UNLCK */
 7     lock.l_start = offset;    /* byte offset, relative to l_whence */
 8     lock.l_whence = whence;   /* SEEK_SET, SEEK_CUR, SEEK_END */
 9     lock.l_len = len;         /* #bytes (0 means to EOF) */

10     return( fcntl(fd, cmd, &lock) );   /* -1 upon error */
11 }
```

lib/lock_reg.c

图9-4　调用fcntl获取或释放一个锁

lib/lock_test.c

```
 1 #include    "unpipc.h"

 2 pid_t
 3 lock_test(int fd, int type, off_t offset, int whence, off_t len)
 4 {
 5     struct flock  lock;

 6     lock.l_type = type;       /* F_RDLCK or F_WRLCK */
 7     lock.l_start = offset;    /* byte offset, relative to l_whence */
 8     lock.l_whence = whence;   /* SEEK_SET, SEEK_CUR, SEEK_END */
 9     lock.l_len = len;         /* #bytes (0 means to EOF) */

10     if (fcntl(fd, F_GETLK, &lock) == -1)
11         return(-1);           /* unexpected error */

12     if (lock.l_type == F_UNLCK)
13         return(0);            /* false, region not locked by another proc */
14     return(lock.l_pid);       /* true, return positive PID of lock owner */
15 }
```

lib/lock_test.c

图9-5　调用fcntl测试一个锁

我们还定义了两个包裹函数Lock_reg和Lock_test，它们在fcntl出错时输出一个错误并终止。另有7个同名但首字母大写的宏，它们调用这两个包裹函数。

使用这些宏，图9-3中的my_lock和my_unlock函数变为：

```
#define my_lock(fd)     (Writew_lock(fd, 0, SEEK_SET, 0))
#define my_unlock(fd)   (Un_lock(fd, 0, SEEK_SET, 0))
```

9.4　劝告性上锁

Posix记录上锁称为劝告性上锁（advisory locking）。其含义是内核维护着已由各个进程上锁的所有文件的正确信息，但是它不能防止一个进程写已由另一个进程读锁定的某个文件。类似地，它也不能防止一个进程读已由另一个进程写锁定的某个文件。一个进程能够无视一个劝告性锁而写一个读锁定文件，或者读一个写锁定文件，前提是该进程有读或写该文件的足够权限。

劝告性锁对于协作进程（cooperating processes）是足够了。网络编程中守护程序的编写是协作进程的一个例子：这些程序访问诸如序列号文件之类的共享资源，而且都在系统管理员的控制之下。只要含有序列号的真正文件不是任何进程都可写，那么在该文件被锁住期间，不理会劝告性锁的随意进程无法写它。

例子：非协作进程

通过运行我们的序列号程序的两个实例，就能展示Posix记录上锁是劝告性的，这两个实例是：使用图9-3中函数的lockfcntl，它在给序列号加1前先锁住文件，以及使用图9-1中函数的locknone，它不执行上锁。

202
↓
203

```
solaris % lockfcntl & locknone &
lockfcntl: pid = 18816, seq# = 1
lockfcntl: pid = 18816, seq# = 2
lockfcntl: pid = 18816, seq# = 3
lockfcntl: pid = 18816, seq# = 4
lockfcntl: pid = 18816, seq# = 5
lockfcntl: pid = 18816, seq# = 6
lockfcntl: pid = 18816, seq# = 7
lockfcntl: pid = 18816, seq# = 8
lockfcntl: pid = 18816, seq# = 9
lockfcntl: pid = 18816, seq# = 10
lockfcntl: pid = 18816, seq# = 11
locknone: pid = 18817, seq# = 11          切换进程；出错
locknone: pid = 18817, seq# = 12
locknone: pid = 18817, seq# = 13
locknone: pid = 18817, seq# = 14
locknone: pid = 18817, seq# = 15
locknone: pid = 18817, seq# = 16
locknone: pid = 18817, seq# = 17
locknone: pid = 18817, seq# = 18
lockfcntl: pid = 18816, seq# = 12         切换进程；出错
lockfcntl: pid = 18816, seq# = 13
lockfcntl: pid = 18816, seq# = 14
lockfcntl: pid = 18816, seq# = 15
lockfcntl: pid = 18816, seq# = 16
lockfcntl: pid = 18816, seq# = 17
lockfcntl: pid = 18816, seq# = 18
lockfcntl: pid = 18816, seq# = 19
lockfcntl: pid = 18816, seq# = 20
locknone: pid = 18817, seq# = 19          切换进程；出错
locknone: pid = 18817, seq# = 20
locknone: pid = 18817, seq# = 21
locknone: pid = 18817, seq# = 22
locknone: pid = 18817, seq# = 23
locknone: pid = 18817, seq# = 24
locknone: pid = 18817, seq# = 25
locknone: pid = 18817, seq# = 26
locknone: pid = 18817, seq# = 27
locknone: pid = 18817, seq# = 28
locknone: pid = 18817, seq# = 29
locknone: pid = 18817, seq# = 30
```

lockfcntl程序首先运行，但是在它执行将序列号从11增加到12的三个步骤期间（此间它持有整个文件的锁），内核切换进程，并且locknone程序运行。该新程序读出的序列号值是lockfcntl程序写回序列号文件之前的11。由lockfcntl程序持有的劝告性记录锁对locknone程序没有影响。

9.5　强制性上锁

有些系统提供另一种类型的记录上锁，称为强制性上锁（mandatory locking）。使用强制性锁后，内核检查每个read和write请求，以验证其操作不会干扰由某个进程持有的某个锁。对
204 于通常的阻塞式描述符，与某个强制性锁冲突的read或write将把调用进程投入睡眠，直到该锁释放为止。对于非阻塞式描述符，与某个强制性锁冲突的read或write将导致它们返回一个EAGAIN错误。

> Posix.1和Unix 98只定义劝告性上锁。然而源自System V的许多实现却同时提供劝告性上锁和强制性上锁。强制性记录上锁是随System V Release 3引入的。

为对某个特定文件施行强制性上锁，应满足：

- 组成员执行位必须关掉；
- SGID位必须打开。

注意，打开某个文件的SUID位而不打开它的用户执行位是没有意义的，同样，打开SGID位而不打开组成员执行位也没有意义。因此，以这种方式加上的强制性锁不会影响任何现有的用户软件。强制性上锁不需要新的系统调用。

在支持强制性记录上锁的系统上，ls命令查找权限位的这种特殊组合，并输出l或L以指示相应文件的强制性上锁是否启用。类似地，chmod命令接受l这个指示符以给某个文件启用强制性上锁。

例子

初看起来，使用强制性上锁应该解决非协作进程的问题，因为非协作进程对被锁住文件的任何read或write调用都将阻塞进程本身，直到该文件的锁被释放为止。不幸的是，定时问题相当复杂，这一点我们很容易展示。

要把我们使用fcntl的例子转换成使用强制性上锁，所需做的是修改seqno文件的权限位。我们还改用另一个版本的main函数，它的for循环次数取自第一个命令行参数（而不是使用常值20），每次循环时不再调用printf。

```
solaris % cat > seqno                          首先把序列号值初始化为1
1
^D                                             Ctrl+D是我们的终端文件结束符
solaris % ls -l seqno
-rw-r--r--   1 rstevens other1        2 Oct  7 11:24 seqno
solaris % chmod +l seqno                       启用强制性上锁
solaris % ls -l seqno
-rw-r-lr--   1 rstevens other1        2 Oct  7 11:24 seqno
```

现在在后台启动两个程序：loopfcntl使用fcntl上锁，loopnone不上锁。所指定的命令行参数为10 000，它是每个程序读出、加1再写入序列号的次数。

```
solaris % loopfcntl 10000 & loopnone 10000 &   在后台同时启动两个程序
solaris % wait                                 等待这两个后台作业的完成
solaris % cat seqno                            然后查看序列号
205 14378                                      出错：应该是20 001
```

每次运行这两个程序，最终的序列号通常在14 000和16 000之间。如果上锁像期望的那样工作的话，最终值应该总是为20 001。为查看错误发生位置，我们需要画出具体到每个步骤的时间线图，如图9-6所示。

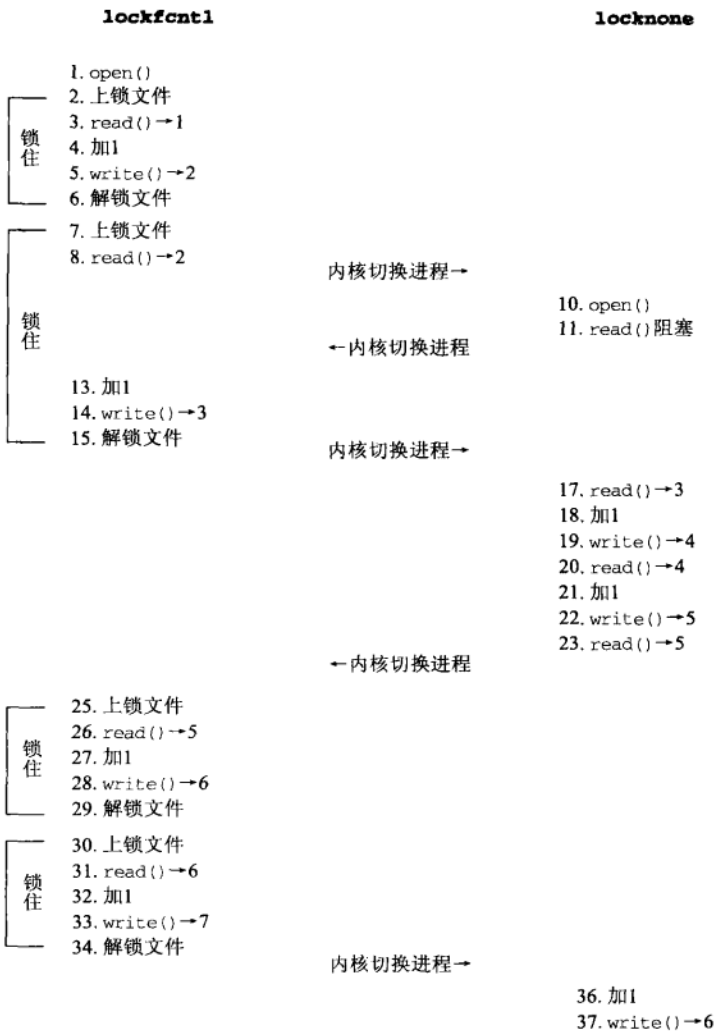


图9-6 loopfcntl和loopnone程序的时间线图

我们假设loopfcntl程序首先启动，执行图中所示前8个步骤。然后内核在loopfcntl持有序列号文件的一个记录锁期间切换进程。于是loopnone启动，但是它的第一个read阻塞了，因为它想从中读出序列号的文件有一个由另一个进程持有的未释放强制性锁。我们假设内核把进程切换回第一个程序，由它执行第13、14和15步。这是我们期待的行为：内核阻塞来自非协作进程的read，因为它试图读的文件由另一个进程锁着。

然后内核切换进程到locknone程序，由它执行第17~23步。这些步骤中的read和write是允许的，因为第一个程序已在第15步给序列号文件解锁。然而，当该程序在第23步read到值为5的序列号，接着内核切换到第一个进程时，问题就发生了。第一个进程接着给序列号值加1两次，然后在第二个进程运行第36步前存入一个值为7的序列号。但是第二个进程往序列号文件写入的值却为6，这是错误的。

我们从这个例子中看到的是，尽管强制性上锁阻止了非协作进程读一个已被锁住的文件（第11步），但是仍没有解决问题。问题出在当右边的进程处于更新序列号的三个步骤（第23、36和37步）期间时，左边的进程是允许更新序列号文件的（第25~34步）。如果有多个进程在更新一个文件，那么所有进程必须使用某种上锁形式协作。只要一个进程违规就可能引发大混乱。

9.6　读出者和写入者的优先级

在8.4节我们的读写锁实现中，优先考虑的是等待着的写入者而不是等待着的读出者。现在看看由fcntl记录上锁提供的解决读出者与写入者问题的办法的某些细节。我们想看到的是，当一个文件区已被锁住时，待处理的上锁请求是如何处理的，这是Posix未曾说明的。

9.6.1　例子：某个写入锁待处理期间的额外读出锁

我们问的第一个问题是：如果某个资源已经读锁定，并有一个写入锁请求在等待处理，那么是否允许有另一个读出锁？某些解决读出者与写入者问题的办法不允许在已有一个写入者等待着的情况下再增加一个读出者，因为要是不断允许新的读出请求的话，待处理的写入请求存在永远不被允许的可能性。

为测试fcntl记录上锁是如何处理这种情形的，我们编写一个测试程序，它获取某个完整文件的一个读出锁，然后fork两个子进程。第一个子进程首先尝试获取一个写入锁（它将阻塞，因为父进程已持有整个文件的一个读出锁），然后由第二个进程尝试获取一个读出锁。图9-7展示了这些请求的时间线图，图9-8给出了我们的测试程序。

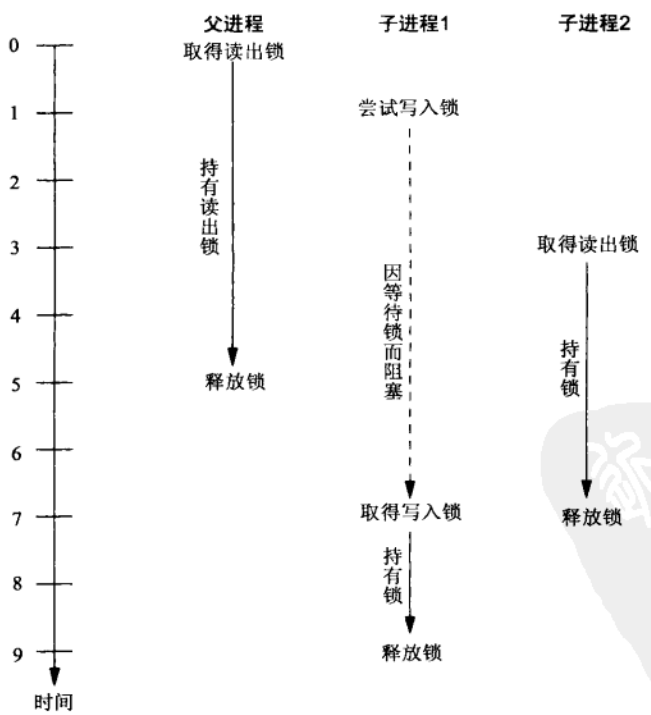


图9-7　确定有一个写入锁待处理期间是否允许有另一个读出锁

lock/test2.c

```
1 #include    "unpipc.h"
2 int
3 main(int argc, char **argv)
4 {
5     int     fd;
6     fd = Open("test1.data", O_RDWR | O_CREAT, FILE_MODE);
7     Read_lock(fd, 0, SEEK_SET, 0);     /* parent read locks entire file */
8     printf("%s: parent has read lock\n", Gf_time());
9     if (Fork() == 0) {
10            /* first child */
11        sleep(1);
12        printf("%s: first child tries to obtain write lock\n", Gf_time());
13        Writew_lock(fd, 0, SEEK_SET, 0);    /* this should block */
14        printf("%s: first child obtains write lock\n", Gf_time());
15        sleep(2);
16        Un_lock(fd, 0, SEEK_SET, 0);
17        printf("%s: first child releases write lock\n", Gf_time());
18        exit(0);
19    }
20    if (Fork() == 0) {
21            /* second child */
22        sleep(3);
23        printf("%s: second child tries to obtain read lock\n", Gf_time());
24        Readw_lock(fd, 0, SEEK_SET, 0);
25        printf("%s: second child obtains read lock\n", Gf_time());
26        sleep(4);
27        Un_lock(fd, 0, SEEK_SET, 0);
28        printf("%s: second child releases read lock\n", Gf_time());
29        exit(0);
30    }
31        /* parent */
32    sleep(5);
33    Un_lock(fd, 0, SEEK_SET, 0);
34    printf("%s: parent releases read lock\n", Gf_time());
35    exit(0);
36 }
```

lock/test2.c

图9-8 确定在有一个写入锁待处理期间是否允许有另一个读出锁

父进程打开文件并获取读出锁

6~8 父进程打开文件，并获取整个文件的读出锁。注意，我们调用read_lock（它不阻塞，但当无法取得锁时会返回一个错误）而不是readw_lock（它可能等待），因为预期该锁会立即取得。当取得该锁时，输出带有当前时间的一个消息（使用UNPv1第404页的gf_time函数）。

fork第一个子进程

9~19 创建第一个子进程，它睡眠1秒，然后阻塞，等待整个文件的一个写入锁。当取得该写入锁时，该进程持有它2秒后即释放它，然后终止。

fork第二个子进程

20~30 创建第二个子进程，它睡眠3秒以允许第一个子进程的写入锁处于待处理状态，然后尝试获取整个文件的一个读出锁。到时候我们就能凭readw_lock返回时输出的消息判

定，该读出锁是被排入请求队列了还是立即给予了。该锁持有4秒后被释放。

父进程持有读出锁5秒

31~35　父进程持有读出锁5秒后，释放该锁，然后终止。[①]

207 ~ 209

图9-7所示的时间线图是我们在Solaris 2.6、Digital Unix 4.0B和BSD/OS 3.1下看到的情形。也就是说，即使已有来自第一个子进程的一个待处理写入锁请求，第二个子进程请求的读出锁也是立即给予的。这么一来，只要连续不断地发出读出锁请求，写入者就可能因获取不了写入锁而“挨饿”。下面是程序的输出，我们在大的时间事件之间插入些空白行，以改善可读性：

```
alpha % test2
16:32:29.674453: parent has read lock

16:32:30.709197: first child tries to obtain write lock

16:32:32.725810: second child tries to obtain read lock
16:32:32.728739: second child obtain read lock

16:32:34.722282: parent releases read lock

16:32:36.729738: second child releases read lock
16:32:36.735597: first child obtains write lock

16:32:38.736938: first child releases write lock
```

9.6.2　例子：等待着的写入者是否比等待着的读出者优先

我们问的下一个问题是：等待着的写入者比等待着的读出者更优先吗？某些解决读出者与写入者问题的办法内置着这样的优先关系。

图9-9是我们的测试程序，图9-10是该测试程序的时间线图。

父进程创建文件并获取写入锁

6~8　父进程创建一个文件，并获取整个文件的一个写入锁。

fork第一个子进程

9~19　派生第一个子进程，它睡眠1秒后请求整个文件的一个写入锁。我们知道这将阻塞，因为父进程已获取整个文件的一个写入锁并且持有它5秒，不过我们期待父进程持有的锁释放时，本请求已排入队。

fork第二个子进程

20~30　派生第二个子进程，它睡眠3秒后请求整个文件的一个读出锁。当父进程释放持有的写入锁时，本请求也将排入队。

在Solaris 2.6和Digital Unix 4.0B下，我们看到第一个子进程的写入锁先于第二个子进程的读出锁取得，如图9-10所示。但是这不足以告诉我们写入锁比读出锁优先，因为其原因可能是内核以FIFO顺序准予上锁请求，而不管它们是读出锁还是写入锁。为验证之，我们创建另外一个与图9-9几乎相同的测试程序，不过读出锁请求是在第1秒发生，写入锁请求是在第3秒发生。前后两个测试程序表明，Solaris和Digital Unix是以FIFO顺序处理上锁请求的，而不管上锁请求的类型。这两个程序还表明，BSD/OS 3.1优先考虑读出请求。

210

① 为避免在子进程的输出中出现shell提示符，可以在第34行和第35行之间插入一行“wait(NULL); wait(NULL);”以等待两个子进程终止。图9-9、图12-10和图12-12中的程序也有类似情况。

lock/test3.c

```
1 #include    "unpipc.h"

2 int
3 main(int argc, char **argv)
4 {
5     int     fd;

6     fd = Open("test1.data", O_RDWR | O_CREAT, FILE_MODE);

7     Write_lock(fd, 0, SEEK_SET, 0);         /* parent write locks entire file */
8     printf("%s: parent has write lock\n", Gf_time());

9     if (Fork() == 0) {
10            /* first child */
11        sleep(1);
12        printf("%s: first child tries to obtain write lock\n", Gf_time());
13        Writew_lock(fd, 0, SEEK_SET, 0);    /* this should block */
14        printf("%s: first child obtains write lock\n", Gf_time());
15        sleep(2);
16        Un_lock(fd, 0, SEEK_SET, 0);
17        printf("%s: first child releases write lock\n", Gf_time());
18        exit(0);
19    }
20    if (Fork() == 0) {
21            /* second child */
22        sleep(3);
23        printf("%s: second child tries to obtain read lock\n", Gf_time());
24        Readw_lock(fd, 0, SEEK_SET, 0);
25        printf("%s: second child obtains read lock\n", Gf_time());
26        sleep(4);
27        Un_lock(fd, 0, SEEK_SET, 0);
28        printf("%s: second child releases read lock\n", Gf_time());
29        exit(0);
30    }
31    /* parent */
32    sleep(5);
33    Un_lock(fd, 0, SEEK_SET, 0);
34    printf("%s: parent releases write lock\n", Gf_time());
35    exit(0);
36 }
```

lock/test3.c

211

图9-9 测试写入者是否比读出者优先

下面是图9-9中程序的输出，我们就是以此构造出图9-10所示的时间线图的。

```
alpha % test3
16:34:02.810285: parent has write lock

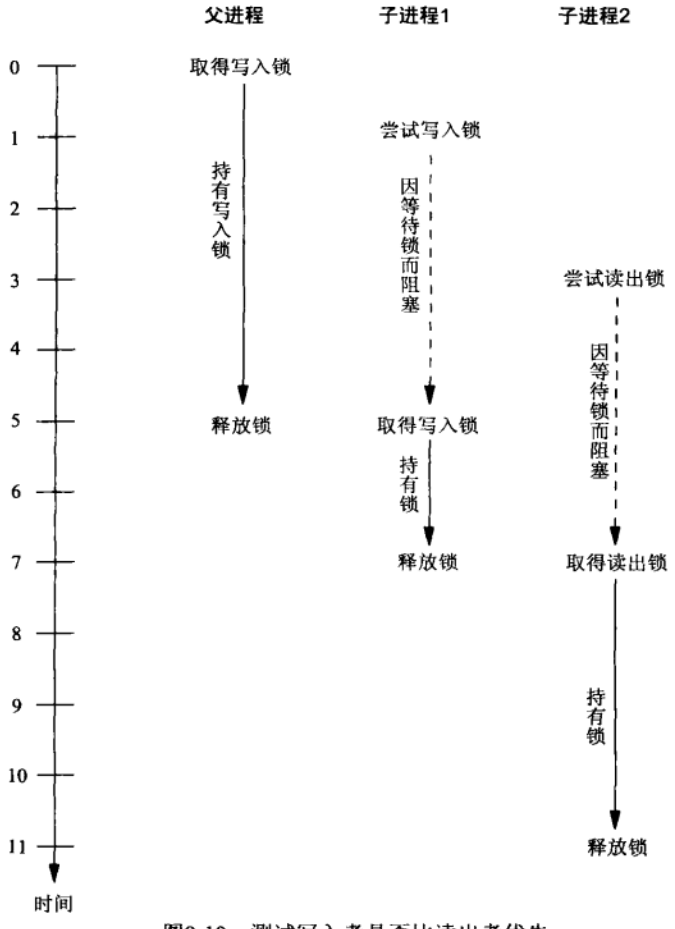
16:34:03.848166: first child tries to obtain write lock

16:34:05.861082: second child tries to obtain read lock


16:34:07.858393: parent releases write lock
16:34:07.865222: first child obtains write lock

16:34:09.865987: first child releases write lock
16:34:09.872823: second child obtains read lock

16:34:13.873822: secound child releases read lock
```



212

图9-10　测试写入者是否比读出者优先

9.7　启动一个守护进程的唯一副本

记录上锁的一个常见用途是确保某个程序（例如守护程序）在任何时刻只有一个副本在运行。图9-11给出了一个守护程序启动时将执行的代码片段。

打开一个文件并为其上锁

8~17　守护进程维护一个只有1行文本的文件，其中含有它的进程ID。它打开这个文件，必要的话创建之，然后请求整个文件的一个写入锁。如果没有取得该锁，我们就知道该程序有另一个副本在运行，于是输出一个出错消息并终止。

> 许多Unix系统让它们的守护进程把各自的进程ID写到一个文件中。Solaris 2.6在/etc目录下存放了其中一些文件。Digital Unix和BSD/OS则在/var/run目录下存放这些文件。

把本进程PID写入文件

18~21　把所打开的文件截为0，然后写入含有本进程PID的一行文本。截短该文件的原因是，

该程序先前的副本（譬如说在系统重新自举前执行的副本）可能有一个值为23456的进程ID，而本副本的进程ID为123。要是光写入那一行而不预先截短，那么文件内容将会是123\n6\n。尽管第一行仍然含有本进程的进程ID，避免该文件中出现第二行的可能却更为清晰，更不易引起混淆。

lock/onedaemon.c

```
1 #include    "unpipc.h"
2 #define PATH_PIDFILE    "pidfile"
3 int
4 main(int argc, char **argv)
5 {
6     int     pidfd;
7     char    line[MAXLINE];
8         /* open the PID file, create if nonexistent */
9     pidfd = Open(PATH_PIDFILE, O_RDWR | O_CREAT, FILE_MODE);
10        /* try to write lock the entire file */
11    if (write_lock(pidfd, 0, SEEK_SET, 0) < 0) {
12        if (errno == EACCES || errno == EAGAIN)
13            err_quit("unable to lock %s, is %s already running?",
14                     PATH_PIDFILE, argv[0]);
15        else
16            err_sys("unable to lock %s", PATH_PIDFILE);
17    }
18        /* write my PID, leave file open to hold the write lock */
19    snprintf(line, sizeof(line), "%ld\n", (long) getpid());
20    Ftruncate(pidfd, 0);
21    Write(pidfd, line, strlen(line));
22    /* then do whatever the daemon does ... */
23    pause();
24 }
```

lock/onedaemon.c

图9-11 确保某个程序只有一个副本在运行

下面是图9-11中程序的一个测试结果：

```
solaris % onedaemon &                          启动第一个副本
[1]       22388
solaris % cat pidfile                          检查写入文件中的PID
22388
solaris % onedaemon                            然后尝试启动第二个副本
unable to lock pidfile, is onedaemon already running?
```

一个守护进程还有其他方法防止自身另一个副本启动，譬如说可能使用信号量。本节所示的方法的优势在于，许多守护程序都编写成向某个文件写入本进程ID，而且如果某个守护进程过早崩溃了，那么内核会自动释放它的记录锁。

9.8 文件作锁用

Posix.1保证，如果以O_CREAT（若文件不存在则创建它）和O_EXCL（独占打开）标志调用open函数，那么一旦该文件已经存在，该函数就返回一个错误。而且考虑到其他进程的存在，检查该文件是否存在和创建该文件（如果它还不存在）必须是原子的。因此，我们可以把以这

种技巧创建的文件作为锁使用。Posix.1保证任何时候只有一个进程能够创建这样的文件（也就是获取锁），释放这样的锁只需unlink该文件。

图9-12给出了使用这种技巧的上锁函数的一个版本。如果open成功，我们就持有与所创建文件对应的锁，于是my_lock函数可以返回。返回前还close该文件，因为我们并不需要它的描述符：该文件的存在本身代表锁，至于它是否打开则无关紧要。如果open返回一个EEXIST错误，那么该文件已经存在，于是我们再次尝试open。

lock/lockopen.c

```
 1 #include    "unpipc.h"
 2 #define LOCKFILE    "/tmp/seqno.lock"
 3 void
 4 my_lock(int fd)
 5 {
 6     int     tempfd;
 7     while ( (tempfd = open(LOCKFILE, O_RDWR|O_CREAT|O_EXCL, FILE_MODE)) < 0){
 8         if (errno != EEXIST)
 9             err_sys("open error for lock file");
10         /* someone else has the lock, loop around and try again */
11     }
12     Close(tempfd);              /* opened the file, we have the lock */
13 }
14 void
15 my_unlock(int fd)
16 {
17     Unlink(LOCKFILE);           /* release lock by removing file */
18 }
```

lock/lockopen.c

图9-12　使用指定O_CREAT和O_EXCL标志的open实现的锁函数

这种技巧存在以下三个问题。

(1) 如果当前持有该锁的进程没有释放它就终止，那么其文件名并未删除。对付这个问题有一些特别的技巧，例如检查该文件的最近访问时间，如果它有一段大于某个确定数量的时间未曾访问，那就假设它已被遗忘，不过这些技巧没有一个是完善的。另一个技巧是把持有该锁的进程的进程ID写入其锁文件中，这样其他进程可以读出该进程ID，并检查该进程是否仍在运行。这也是不完善的，因为进程ID在过一段时间后会被重用。

这种情形对fcntl记录上锁而言不成问题，因为当某个进程终止时，由它持有的任何记录锁都自动释放。

(2) 如果另外某个进程已打开了锁文件，那么当前进程只是在一个无限循环中一次又一次地调用open。这称为轮询，是对CPU时间的一种浪费。一种替换技巧是sleep 1秒，然后再次尝试open。（我们在图7-5中看到了同样的问题。）

如果使用fcntl记录上锁，这就不成问题，前提是想要持有该锁的进程指定FSETLKW命令。内核将把该进程投入睡眠，直到该锁可用，然后唤醒它。

(3) 调用open和unlink创建和删除一个额外的文件涉及文件系统的访问，这通常比调用fcntl两次（一次用于获取锁，一次用于释放锁）所花时间长得多。测量在我们的程序中给序列号加1共1000次的循环所花的执行时间，发现fcntl记录上锁比调用open和unlink快75倍。

Unix文件系统的另外两个技巧也用于提供特殊的上锁。第一个技巧是：如果新链接的名字已经存在，那么link函数将失败。为获取一个锁，首先创建一个唯一的临时文件，其路径名中

含有调用进程的进程ID（如果不同进程中的线程间以及同一进程内的线程间都需要上锁，那么所含的是进程ID和线程ID的某种组合）。然后以待建立锁文件的众所周知路径名调用link函数创建这个临时文件的一个链接。如果创建成功，该临时路径名就可以unlink掉。当调用线程使用完该锁时，只需unlink其众所周知的路径名就可以解锁。如果link失败返回EEXIST错误，调用线程就得重新尝试（类似于图9-12中的做法）。这种技巧的要求之一是：临时文件路径名和锁文件众所周知的路径名必须都存在于同一文件系统中，因为多数版本的Unix不允许硬链接（link函数的结果）跨越不同的文件系统。

213 ~ 215

第二种技巧基于：如果待打开的文件已经存在，打开时指定了O_TRUNC标志，而且调用进程不具备写访问权限，那么open调用将返回一个错误。为获取一个锁，我们在指定O_CREAT|O_WRONLY|O_TRUNC标志并置mode参数为0（即新文件不打开任何权限位）的前提下调用open。如果调用成功，我们就拥有了该锁，以后使用完该锁后只需unlink其路径名。如果open调用失败返回EACCES错误，那么调用线程必须重新尝试（类似于图9-12中的做法）。需要注意的是，这种技巧在调用线程具备超级用户特权时不起作用。

从这些例子得出的教训是：应该使用fcntl记录上锁。然而你有可能碰到使用这些老式上锁技巧的代码，它们通常存在于fcntl上锁尚未广泛得以实现之前编写的程序中。

9.9 NFS上锁

NFS就是网络文件系统，它在TCPv1第29章中讨论。作为对NFS的一种扩展，NFS的大多数实现支持fcntl记录上锁。Unix系统通常以两个额外的守护进程支持NFS记录上锁，它们是lockd和statd。当某个进程调用fcntl以获取一个锁，而且内核检测出其描述符引用通过NFS安装的某个文件系统上的一个文件时，本地的lockd就向服务器的lockd发送这个请求。statd守护进程跟踪着持有锁的各个客户，它与lockd交互以提供NFS上锁的崩溃恢复功能。

我们可以预期NFS文件的记录上锁比本地文件的记录上锁花的时间长，因为获取与释放每一个锁都需要网络通信。为测试NFS记录上锁，我们只需修改图9-2中由SEQFILE指定的文件名。测量我们的程序使用fcntl记录上锁执行10 000次循环所需的时间，发现本地文件的记录上锁比NFS文件的记录上锁快了约80倍。还要留意的是，当序列号文件在某个通过NFS安装的文件系统上时，记录上锁和序列号的读写都涉及网络通信。

> 防止误解的说明：NFS记录上锁多年来一直是个问题，它差不多是由不理想的实现所导致的。尽管主要的Unix厂家已最终清理了它们各自的实现，通过NFS使用fcntl记录上锁对于许多实现来说仍然是一个严重的问题。我们不会在这个问题上偏袒一方而贬低另一方，只是指出fcntl记录上锁在NFS上也应该起作用，不过实际成功与否取决于实现的质量，客户端和服务器端都有质量要求。

9.10 小结

fcntl记录上锁提供了对一个文件的劝告性或强制性上锁功能，而我们是通过该文件打开着的描述符来访问它的。这些锁用于不同进程间的上锁，而不是同一进程内不同线程间的上锁。术语“记录”是个不确切的名字，因为Unix内核没有文件内记录的概念。更好的称谓是“范围上锁（range locking）”，因为我们上锁或解锁的是文件内的一个字节范围。这类记录上锁几乎都

216

用作协作进程之间的劝告性锁，因为即使是强制性上锁也会导致不一致的数据，正如我们所示。

使用fcntl记录上锁时，等待着的读出者优先还是等待着的写入者优先没有保证，这也是我们在第8章中看到过的读写锁的情形。如果这对于某个应用来说很重要，那就编写并运行9.6节中开发的类似测试程序，或者给该应用提供满足所需优先关系的专用读写锁实现（如我们在8.4节所做的那样）。

习题

9.1　从图9-2和图9-1构造locknone程序，在自己的系统上运行多次。验证这个没有任何上锁能力的程序工作不正确，而且结果是非确定的。

9.2　把图9-2中的程序修改成不对标准输出进行缓冲。这样的修改有什么效果？

9.3　继续上一道习题，这次改为调用putchar逐个输出字符，而不是调用printf。这样的修改有什么效果？

9.4　把图9-3中my_lock函数使用的写入锁改为读出锁。会发生什么？

9.5　把loopmain.c程序中的open调用改为同时指定O_NONBLOCK标志。构造loopfcntlnonb程序，同时运行它的两个实例。结果有什么变化吗？为什么？

9.6　继续上一道习题，这次使用非阻塞版本的loopmain.c构造loopnonenonb程序（使用locknone.c文件，它不进行上锁操作）。启用seqno文件的强制性上锁。同时运行本程序的一个实例以及来自上一道习题的loopfcntlnonb程序的一个实例。会发生什么？

9.7　构造loopfcntl程序，从某个shell脚本在后台运行它10次。这10个实例的每一个应指定一个值为10 000的命令行参数。首先在使用劝告性上锁的前提下给这个shell脚本计时，然后把seqno文件的权限改为启用强制性上锁。强制性上锁对性能有什么影响？

9.8　在图9-8和图9-9中，我们为什么调用fork创建子进程，而不是调用pthread_create创建线程？

9.9　在图9-11中，我们调用ftruncate把文件的大小置为0字节。为什么不改为简单地给open指定O_TRUNC标志？

9.10　如果我们要编写一个使用fcntl记录上锁的线程化应用程序，那么在指定上锁的起始字节偏移量时，应使用SEEK_SET、SEEK_CUR还是SEEK_END呢？为什么？

217

第10章 Posix信号量

10.1 概述

信号量（semaphore）是一种用于提供不同进程间或一个给定进程的不同线程间同步手段的原语。本书讨论三种类型的信号量。

- Posix有名信号量：使用Posix IPC名字（2.2节）标识，可用于进程或线程间的同步。
- Posix基于内存的信号量：存放在共享内存区中，可用于进程或线程间的同步。
- System V信号量（第11章）：在内核中维护，可用于进程或线程间的同步。

我们暂时只考虑不同进程间的同步。首先考虑二值信号量（binary semaphore）：其值或为0或为1的信号量。图10-1展示了这种信号量。

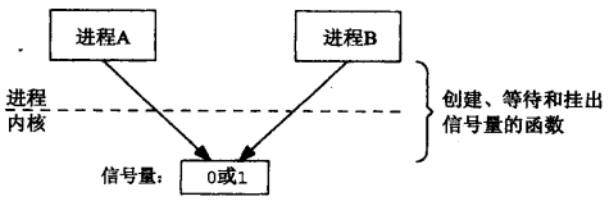


图10-1 由两个进程使用的一个二值信号量

图中画出该信号量是由内核来维护的（这对于System V信号量是正确的），其值可以是0或1。

Posix信号量不必在内核中维护。另外，Posix信号量是由可能与文件系统中的路径名对应的名字来标识的。因此，图10-2是Posix有名信号量的更为实际的图示。

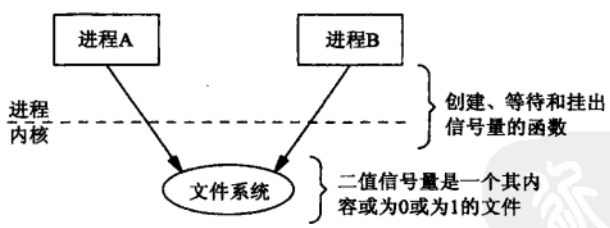


图10-2 由两个进程使用的一个Posix有名二值信号量

我们必须就图10-2作一个限定：尽管Posix有名信号量是由可能与文件系统中的路径对应的名字来标识的，但是并不要求它们真正存放在文件系统内的某个文件中。举例来说，嵌入式实时系统可能使用这样的名字来标识信号量，但是真正的信号量值却存放在内核中的某个地方。然而，如果信号量的实现用到了映射文件（我们将在10.15节展示这样的一个实现），那么信号量的真正值确实出现在某个文件中，而该文件是映射到所有让该信号量打开着的进程的地址空间的。

在图10-1和图10-2中，我们注出了一个进程可以在某个信号量上执行的三种操作。

(1) 创建（create）一个信号量。这还要求调用者指定初始值，对于二值信号量来说，它通常是1，但也可以是0。

(2) 等待（wait）一个信号量。该操作会测试会这个信号量的值，如果其值小于或等于0，那就等待（阻塞），一旦其值变为大于0就将它减1。这个过程可以用如下的伪代码来总结：

```
while (semaphore_value <= 0)
    ;     /* wait; i.e., block the thread or process */
semaphore_value--;
/* we have the semaphore */
```

这里的基本要求是：考虑到访问同一信号量的其他线程或进程，在while语句中测试该信号量的值和其后将它减1（如果该值大于0）这两个步骤必须作为一个原子操作完成。（这是20世纪80年代中期System V信号量在内核中实现的原因之一。这样一来信号量操作成为内核中的系统调用，于是保证相对其他进程的原子性变得容易起来。）

220 本操作还有其他常用名字：最初Edsger Dijkstra称它为P操作，代表荷兰语单词proberen（意思是尝试）。它也称为递减（down，因为信号量的值被减掉1）或上锁（lock），不过我们使用Posix术语等待（wait）。

(3) 挂出（post）一个信号量。该操作将信号量的值加1，可以用如下的伪代码来总结：

```
semaphore_value++;
```

如果有一些进程阻塞着等待该信号量的值变为大于0，其中一个进程现在就可能被唤醒。与刚刚给出的等待伪代码一样，考虑到访问同一信号量的其他进程，挂出操作也必须是原子的。

本操作还有其他常用名字：最初称为V操作，代表荷兰语单词verhogen（意思是增加）。它也称为递增（up，因为信号量的值被加上1）、解锁（unlock）或发信号（signal）。我们使用Posix术语挂出（post）。

显而易见，真正的信号量代码比我们给出的等待和挂出操作的伪代码有更多的细节，也就是如何将等待某个给定信号量的所有进程排队，然后如何唤醒一个（可能是很多进程中的一个）正在等待某个给定信号量被挂出的进程。所幸的是这些细节是由实现来处理的。

注意，上面给出的伪代码并没有假定使用其值仅为0或1的二值信号量。它们适用于其值初始化为任意非负值的信号量。这样的信号量称为计数信号量（counting semaphore）。计数信号量通常初始化为某个值N，指示可用的资源（譬如说缓冲区）数。本章我们将同时展示二值信号量和计数信号量的例子。

> 我们往往在二值信号量和计数信号量之间进行区分，这样做是为了我们自己的教学目的。在实现信号量的代码中，这两者间并没有差别。

二值信号量可用于互斥目的，就像互斥锁一样。图10-3给出了一个例子。

初始化互斥锁；	初始化信号量为1；
pthread_mutex_lock(&mutex);	sem_wait(&sem);
临界区	临界区
pthread_mutex_unlock(&mutex);	sem_post(&sem);

图10-3　比较解决互斥问题的互斥锁和信号量

我们把信号量初始化为1，sem_wait调用等待其值变为大于0，然后将它减1，sem_post调用则将其值加1（从0变为1），然后唤醒阻塞在sem_wait调用中等待该信号量的任何线程。

221 除可以像互斥锁那样使用外，信号量还有一个互斥锁没有提供的特性：互斥锁必须总是由

锁住它的线程解锁，信号量的挂出却不必由执行过它的等待操作的同一线程执行。我们可以使用两个二值信号量和第7章中生产者-消费者问题的一个简化版本提供展示这种特性的一个例子。图10-4展示了往某个共享缓冲区中放置一个条目的一个生产者以及取走该条目的一个消费者。为简单起见，假设该缓冲区只容纳一个条目。

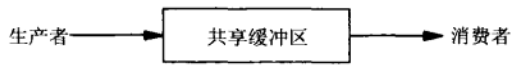


图10-4 使用一个共享缓冲区的简单生产者-消费者问题

图10-5给出了生产者和消费者程序的伪代码。

```
            生产者                                    消费者
把信号量get初始化为0;
把信号量put初始化为1;

for ( ; ; ) {                                 for ( ; ; ) {
 sem_wait(&put);                                   sem_wait(&get);
 把数据放入缓冲区                                   处理缓冲区中的数据
 sem_post(&get);                                   sem_post(&put);
}                                             }
```

图10-5 简单的生产者-消费者程序的伪代码

信号量put控制生产者是否可以往共享缓冲区中放置一个条目，信号量get控制消费者是否可以从共享缓冲区中取走一个条目。按时间顺序发生的步骤如下所述。

(1) 生产者初始化缓冲区和两个信号量。

(2) 假设消费者接着运行。它阻塞在sem_wait调用中，因为get的值为0。

(3) 一段时间后生产者接着运行。当它调用sem_wait后，put的值由1减为0，于是生产者往缓冲区中放置一个条目，然后它调用sem_post，把get的值由0增为1。既然有一个线程（即消费者）阻塞在该信号量上等待其值变为正数，该线程将被标记成准备好运行（ready_to_run）。但是假设生产者继续运行，生产者随后会阻塞在for循环顶部的sem_wait调用中，因为put的值为0。生产者必须等待到消费者腾空缓冲区。

(4) 消费者从sem_wait调用中返回，它将get信号量的值由1减为0。接着它会处理缓冲区中的数据，然后调用sem_post，把put的值由0增为1。既然有一个线程（生产者）阻塞在该信号量上等待其值变为正数，该线程将被标记成准备好运行。但是假设消费者继续运行，消费者随后会阻塞在for循环顶部的sem_wait调用中，因为get的值为0。

(5) 生产者从sem_wait调用中返回，于是把数据放入缓冲区中，上述情形循环继续。

我们假设每次调用sem_post时，即使当时有一个进程正在等待并随后被标记成准备好运行，调用者也继续运行。是调用者继续运行还是刚变成准备好状态的线程运行无关紧要（你应该假设对立的情形，并说服自己接受这个事实）。

下面列出信号量、互斥锁和条件变量之间的三个差异。

(1) 互斥锁必须总是由给它上锁的线程解锁，信号量的挂出却不必由执行过它的等待操作的同一线程执行。这是我们的例子刚展示过的。

(2) 互斥锁要么被锁住，要么被解开（二值状态，类似于二值信号量）。

(3) 既然信号量有一个与之关联的状态（它的计数值），那么信号量挂出操作总是被记住。然而当向一个条件变量发送信号时，如果没有线程等待在该条件变量上，那么该信号将丢失。作为这种特性的一个例子，考虑图10-5，不过假设第一次通过生产者的循环时，消费者还没有

调用sem_wait。生产者仍可以往缓冲区放置数据条目，然后在get信号量上调用sem_post（把它的值由0增为1），接着阻塞在put信号量上的sem_wait调用中。一段时间以后，消费者可能进入它的for循环，在get信号量上调用sem_wait，其结果是将该信号量的值减1（由1减为0），于是消费者接着处理缓冲区。

> Posix.1基本原理一文声称，有了互斥锁和条件变量还提供信号量的原因是："本标准提供信号量的主要目的是提供一种进程间同步方式。这些进程可能共享也可能不共享内存区。互斥锁和条件变量是作为线程间的同步机制说明的，这些线程总是共享（某个）内存区。这两者都是已广泛使用了多年的同步范式。每组原语都特别适合于特定的问题。"我们将在10.15节看到，使用互斥锁和条件变量实现具有随内核持续性的计数信号量需要约300行C代码，应用程序不应该各自从头编写这300行C代码。尽管信号量的意图在于进程间同步，互斥锁和条件变量的意图则在于线程间同步，但是信号量也可用于线程间，互斥锁和条件变量也可用于进程间。我们应该使用适合具体应用的那组原语。

我们提到过Posix提供两类信号量：有名（named）信号量和基于内存的（memory-based）的信号量，后者也称为无名（unnamed）信号量。图10-6比较了这两类信号量使用的函数。

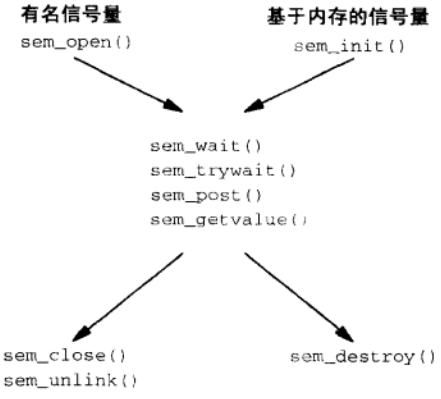


图10-6　用于Posix信号量的函数调用

图10-2展示了一个Posix有名信号量。图10-7则展示了某个进程内由两个线程共享的一个Posix基于内存的信号量。

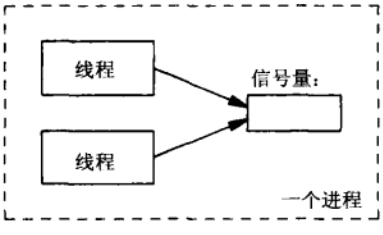


图10-7　由一个进程内的两个线程共享的基于内存的信号量

图10-8展示了某个共享内存区（第4部分）中由两个进程共享的一个Posix基于内存的信号量。图中画出该共享内存区同时属于这两个进程的地址空间。

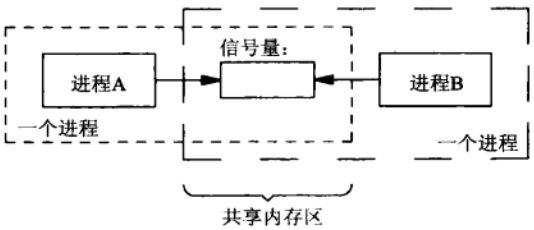


图10-8 由两个进程共享、处于共享内存区中的基于内存的信号量

223
224

本章中我们首先讲述Posix有名信号量，然后讲述Posix基于内存的信号量。我们回到7.3节中的生产者-消费者问题，将它扩展成允许多个生产者和一个消费者，最后是多个生产者和多个消费者。我们然后指出，多个缓冲区的常用I/O技巧只是生产者-消费者问题的一个特例。

我们给出Posix有名信号量的三种实现：第一种实现使用FIFO，第二种实现使用内存映射I/O以及互斥锁和条件变量，第三种实现使用System V信号量。

10.2 sem_open、sem_close 和 sem_unlink 函数

函数sem_open创建一个新的有名信号量或打开一个已存在的有名信号量。有名信号量总是既可用于线程间的同步，又可用于进程间的同步。

```
#include <semaphore.h>

sem_t *sem_open(const char *name, int oflag, ...
                /* mode_t mode, unsigned int value */ );
```

返回：若成功则为指向信号量的指针，若出错则为SEM_FAILED

我们已在2.2节中描述过有关*name*参数的规则。

*oflag*参数可以是0、O_CREAT或O_CREAT|O_EXCL，如2.3节所述。如果指定了O_CREAT标志，那么第三个和第四个参数是需要的：其中*mode*参数指定权限位（图2-4），*value*参数指定信号量的初始值。该初始值不能超过SEM_VALUE_MAX（这个常值必须至少为32767）。二值信号量的初始值通常为1，计数信号量的初始值则往往大于1。

如果指定了O_CREAT（而没有指定O_EXCL），那么只有当所需的信号量尚未存在时才初始化它。不过所需信号量已存在条件下指定O_CREAT并不是一个错误。该标志的意思仅仅是“如果所需信号量尚未存在，那就创建并初始化它”。但是所需信号量已存在条件下指定O_CREAT | O_EXCL却是一个错误。

sem_open的返回值是指向某个sem_t数据类型的指针。该指针随后用作sem_close、sem_wait、sem_trywait、sem_post以及sem_getvalue的参数。

用SEM_FAILED这个返回值来指示错误比较奇怪。使用空指针也许更为合理。后来形成Posix标准的那些早期草案指定使用-1这个返回值来指示出错，许多实现于是定义

```
#define  SEM_FAILED  ((sem_t *)(-1))
```

当使用sem_open创建或打开某个信号量时，Posix.1未就与该信号量关联的权限位做过多少说明。实际上从图2-3和前面的讨论可注意到，当打开一个有名信号量时，我们甚至没有在*oflag*参数中指定O_RDONLY、O_WRONLY或O_RDWR标志。本书中的例子所用的两个系统（Digital Unix 4.0B和Solaris 2.6）都要求对某个已存在的信号量具有读访问和写访问权限，这

样对它的sem_open才能成功。其原因也许是信号量的挂出与等待操作都需要读出并修改信号量的值。这两种实现上，不具备读访问或写访问某个已存在信号量的权限都将导致sem_open函数返回一个EACCES错误（“Permission denied（访问权限不符）”）。

225

使用sem_open打开的有名信号量，使用sem_close将其关闭。

```
#include <semaphore.h>
int sem_close(sem_t *sem);
```

返回：若成功则为0，若出错则为-1

一个进程终止时，内核还对其上仍然打开着的所有有名信号量自动执行这样的信号量关闭操作。不论该进程是自愿终止的（通过调用exit或_exit）还是非自愿地终止的（通过向它发送一个Unix信号），这种自动关闭都会发生。

关闭一个信号量并没有将它从系统中删除。这就是说，Posix有名信号量至少是随内核持续的：即使当前没有进程打开着某个信号量，它的值仍然保持。

有名信号量使用sem_unlink从系统中删除。

```
#include <semaphore.h>
int sem_unlink(const char *name);
```

返回：若成功则为0，若出错则为-1

每个信号量有一个引用计数器记录当前的打开次数（就像文件一样），sem_unlink类似于文件I/O的unlink函数：当引用计数还是大于0时，*name*就能从文件系统中删除，然而其信号量的析构（不同于将它的名字从文件系统中删除）却要等到最后一个sem_close发生时为止。

10.3　sem_wait 和 sem_trywait 函数

sem_wait函数测试所指定信号量的值，如果该值大于0，那就将它减1并立即返回。如果该值等于0，调用线程就被投入睡眠中，直到该值变为大于0，这时再将它减1，函数随后返回。我们以前提到过，考虑到访问同一信号量的其他线程，“测试并减1”操作必须是原子的。

```
#include <semaphore.h>
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
```

226

均返回：若成功则为0，若出错则为-1

sem_wait和sem_trywait的差别是：当所指定信号量的值已经是0时，后者并不将调用线程投入睡眠。相反，它返回一个EAGAIN错误。

如果被某个信号中断，sem_wait就可能过早地返回，所返回的错误为EINTR。

10.4　sem_post 和 sem_getvalue 函数

当一个线程使用完某个信号量时，它应该调用sem_post。就像10.1节中讨论过的那样，本函数把所指定信号量的值加1，然后唤醒正在等待该信号量值变为正数的任意线程。

```
#include <semaphore.h>
int sem_post(sem_t *sem);
int sem_getvalue(sem_t *sem, int *valp);
```

均返回：若成功则为0，若出错则为-1

sem_getvalue在由*valp*指向的整数中返回所指定信号量的当前值。如果该信号量当前已上锁，那么返回值或为0，或为某个负数，其绝对值就是等待该信号量解锁的线程数。

我们现在看到了互斥锁、条件变量和信号量之间的更多差别。首先，互斥锁必须总是由给它上锁的线程解锁。信号量没有这种限制：一个线程可以等待某个给定信号量（譬如说将该信号量的值由1减为0，这跟给该信号量上锁一样），而另一个线程可以挂出该信号量（譬如说将该信号量的值由0增为1，这跟给该信号量解锁一样）。

其次，既然每个信号量有一个与之关联的值，它由挂出操作加1，由等待操作减1，那么任何线程都可以挂出一个信号（譬如说将它的值由0增为1），即使当时没有线程在等待该信号量值变为正数也没有关系。然而，如果某个线程调用了pthread_cond_signal，不过当时没有任何线程阻塞在pthread_cond_wait调用中，那么发往相应条件变量的信号将丢失。

最后，在各种各样的同步技巧（互斥锁、条件变量、读写锁、信号量）中，能够从信号处理程序中安全调用的唯一函数是sem_post。

> 这三个差异点不应该被解释成作者对于信号量的偏袒。我们已看过的所有同步原语（互斥锁、条件变量、读写锁、信号量以及记录上锁）都有它们各自的位置。对于一个给定应用我们已有很多选择，因而需要了解各种原语之间的差别。还要从刚刚列出的比较中意识到的是，互斥锁是为上锁而优化的，条件变量是为等待而优化的，信号量既可用于上锁，也可用于等待，因而可能导致更多的开销和更高的复杂性。

227

10.5 简单的程序

我们现在提供一些在Posix有名信号量上操作的简单程序，目的是更多地了解它们的功能与实现。由于Posix有名信号量至少具有随内核的持续性，因此我们可以跨多个程序操纵它们。

10.5.1 semcreate 程序

图10-9中的程序创建一个有名信号量，允许的命令行选项有指定独占创建的-e和指定一个初始值（默认值1以外的值）的-i。

pxsem/semcreate.c

```
1 #include     "unpipc.h"
2 int
3 main(int argc, char **argv)
4 {
5       int     c, flags;
6       sem_t   *sem;
7       unsigned int value;
8       flags = O_RDWR | O_CREAT;
9       value = 1;
```

图10-9 创建一个有名信号量

```
10      while ( (c = Getopt(argc, argv, "ei:")) != -1) {
11          switch (c) {
12          case 'e':
13              flags |= O_EXCL;
14              break;
15          case 'i':
16              value = atoi(optarg);
17              break;
18          }
19      }
20      if (optind != argc - 1)
21          err_quit("usage: semcreate [ -e ] [ -i initalvalue ] <name>");
22      sem = Sem_open(argv[optind], flags, FILE_MODE, value);
23      Sem_close(sem);
24      exit(0);
25  }
```

pxsem/semcreate.c

图10-9（续）

创建信号量

22　既然总是指定O_CREAT标志，我们调用sem_open时必须提供四个参数。不过最后两个参数只有所需信号量尚未存在时才由sem_open使用。

关闭信号量

23　我们调用sem_close，不过要是省掉了这个调用，当相应进程终止时，所创建的信号量也被关闭（所占用的系统资源随之释放）。

228

10.5.2　semunlink 程序

图10-10中的程序删除一个有名信号量的名字。

pxsem/semunlink.c

```
1 #include     "unpipc.h"
2 int
3 main(int argc, char **argv)
4 {
5       if (argc != 2)
6           err_quit("usage: semunlink <name>");
7       Sem_unlink(argv[1]);
8       exit(0);
9 }
```

pxsem/semunlink.c

图10-10　删除一个有名信号量的名字

10.5.3　semgetvalue 程序

图10-11中的简单程序打开一个有名信号量，取得它的当前值，然后输出该值。

打开信号量

9　当我们去打开一个一定存在的信号量时，sem_open的第二个参数为0，因为我们不指定O_CREAT，也没有其他O_*xxx*常值需指定。

pxsem/semgetvalue.c

```
1 #include    "unpipc.h"

2 int
3 main(int argc, char **argv)
4 {
5     sem_t   *sem;
6     int     val;

7     if (argc != 2)
8         err_quit("usage: semgetvalue <name>");

9     sem = Sem_open(argv[1], 0);
10    Sem_getvalue(sem, &val);
11    printf("value = %d\n", val);

12    exit(0);
13 }
```

pxsem/semgetvalue.c

图10-11 取得并输出一个信号量的值

10.5.4 semwait 程序

图10-12中的程序打开一个有名信号量，调用sem_wait（如果该信号量的当前值小于或等于0，该调用就阻塞，结束阻塞后该调用将信号量的值减1），取得并输出该信号量的当前值，然后永远阻塞在一个pause调用中。

pxsem/semwait.c

```
1 #include    "unpipc.h"

2 int
3 main(int argc, char **argv)
4 {
5     sem_t   *sem;
6     int     val;

7     if (argc != 2)
8         err_quit("usage: semwait <name>");

9     sem = Sem_open(argv[1], 0);
10    Sem_wait(sem);
11    Sem_getvalue(sem, &val);
12    printf("pid %ld has semaphore, value = %d\n", (long) getpid(), val);

13    pause();                    /* blocks until killed */
14    exit(0);
15 }
```

pxsem/semwait.c

图10-12 等待一个信号量并输出它的值

10.5.5 sempost 程序

图10-13中的程序挂出一个有名信号量（即把它的值加1），然后取得并输出该信号量的值。

pxsem/sempost.c

```
1 #include    "unpipc.h"
2 int
3 main(int argc, char **argv)
4 {
5     sem_t   *sem;
6     int     val;
7     if (argc != 2)
8         err_quit("usage: sempost <name>");
9     sem = Sem_open(argv[1], 0);
10    Sem_post(sem);
11    Sem_getvalue(sem, &val);
12    printf("value = %d\n", val);
13    exit(0);
14 }
```

pxsem/sempost.c

229 ~ 230

图10-13　挂出一个信号量

10.5.6　例子

我们首先在Digital Unix 4.0B下创建一个有名信号量，然后输出它的（默认）值。

```
alpha % semcreate /tmp/test1
alpha % ls -l /tmp/test1
-rw-r--r--   1 rstevens system      264 Nov 13 08:51 /tmp/test1
alpha % semgetvalue /tmp/test1
value = 1
```

跟Posix消息队列一样，本系统创建一个位于文件系统中的文件，它对应于我们给所创建的有名信号量指定的名字。

现在等待该信号量，然后中止持有该信号量锁的程序。

```
alpha % semwait /tmp/test1
pid 9702 has semaphore, value = 0          sem_wait返回后的值
^?                                         键入中断键以中止程序
alpha % semgetvalue /tmp/test1
value = 0                                  值仍然为0
```

本例子展示了我们早先提到过的两个特性。首先，信号量的值是随内核持续的。这就是说，从上一个例子创建该信号量到本例子，尽管期间没有程序打开着该信号量，其值（等于1）却由内核维持着。其次，当我们中止持有信号量锁的semwait程序时，该信号量的值并不改变。这就是说，当持有某个信号量锁的进程没有释放它就终止时，内核并不给该信号量解锁。这跟记录锁不一样，我们在第9章中说过，当持有某个记录锁的进程没有释放它就终止时，内核自动释放它。

我们接着展示Digital Unix的信号量实现使用负的信号量值指示等待该信号量解锁的进程数。

```
alpha % semgetvalue /tmp/test1
value = 0                                  值仍然是来自上一个例子的0

alpha % semwait /tmp/test1 &               在后台启动一个semwait程序
[1]     9718                               它阻塞，等待信号量

alpha % semgetvalue /tmp/test1
value = -1                                 有一个进程在等待信号量

alpha % semwait /tmp/test1 &               在后台启动另一个semwait程序
```

```
[2]     9727                                    它也阻塞，等待信号量

alpha % semgetvalue /tmp/test1
value = -2                                      有两个进程在等待信号量

alpha % sempost /tmp/test1                      现在挂出信号量
value = -1                                      sem_post返回后的值
pid 9718 has semaphore, value = -1              来自第一个semwait程序的输出

alpha % sempost /tmp/test1                      再次挂出信号量
value = 0
pid 9727 has semaphore, value = 0               来自第二个semwait程序的输出
```

231

当信号量值为-2时，我们执行sempost程序，于是该值经加1后变为-1，同时有一个原本阻塞在sem_wait调用中的进程返回。

现在改为在Solaris 2.6下执行同样的例子，目的是查看它们在信号量实现上的差异。

```
solaris % semcreate /test2
solaris % ls -l /tmp/.*test2*
-rw-r--r--   1 rstevens other1      48 Nov 13 09:11 /tmp/.SEMDtest2
-rw-rw-rw-   1 rstevens other1       0 Nov 13 09:11 /tmp/.SEMLtest2
solaris % semgetvalue / test2
value = 1
```

跟Posix消息队列一样，Solaris系统在/tmp目录下创建若干文件，作为文件名后缀的是所指定信号量的名字。我们看到第一个文件的权限与我们调用sem_open时指定的权限相对应，至于第二个文件，我们猜想其用于上锁。

下面验证当持有某个信号量锁的进程没有释放该锁就终止时，内核没有自动挂出该信号量。

```
solaris % semwait /test2
pid 4133 has semaphore, value = 0
^?                                              键入中断键
solaris % semgetvalue /test2
value = 0                                       值仍然为0
```

接着展示Solaris的信号量实现在有进程等待着某个信号量的时候如何处理该信号量的值。

```
solaris % semgetvalue /test2
value = 0                                       值仍然是来自上一个例子的0

solaris % semwait /test2 &                      在后台启动一个semwait程序
[1]     4257                                    它阻塞，等待信号量

solaris % semgetvalue /test2
value = 0                                       本实现不使用负值

solaris % semwait /test2 &                      在后台启动另一个semwait程序
[2]     4263

solaris % semgetvalue /test2
value = 0                                       值仍然为0，不过有两个进程在等着

solaris % sempost /test2                        现在挂出信号量
pid 4257 has semaphore, value = 0               来自第一个semwait程序的输出
value = 0

solaris % sempost /test2
pid 4263 has semaphore, value = 0               来自第二个semwait程序的输出
value = 0
```

与前面在Digital Unix下的输出相比，这个输出中的一个差别是在挂出信号量的时机：看起来等待着的进程优于挂出信号量的进程运行。

232

10.6　生产者-消费者问题

我们在7.3节中讲述了生产者-消费者问题，并展示了一些解决方案，具体解决多个生产者线程填写由单个消费者线程处理的一个数组时的同步问题。

(1) 在第一个方案中（7.3节），消费者是在生产者完成后启动的，因此使用单个互斥锁（来同步各个生产者）就能解决同步问题。

(2) 在下一个方案中（7.5节），消费者在生产者完成之前启动，因此解决同步问题需要一个互斥锁（来同步各个生产者）加上一个条件变量及其互斥锁（来同步生产者和消费者）。

现在对生产者-消费者问题进行扩展，把共享缓冲区用作一个环绕缓冲区：生产者填写最后一项（buff[NBUFF-1]）后，回过头来填写第一项（buff[0]），消费者也同样这么做。这么一来增加了又一个同步问题，即生产者不能走到消费者的前面。我们仍然假设生产者和消费者都是线程，不过它们也可以是进程，前提是存在某种在进程间共享缓冲区的方法（例如我们将在第4部分介绍的共享内存区）。

当共享缓冲区作为一个环绕缓冲区考虑时，必须由代码来维持以下三个条件。

(1) 当缓冲区为空时，消费者不能试图从其中去除一个条目。

(2) 当缓冲区填满时，生产者不能试图往其中放置一个条目。

(3) 共享变量可能描述缓冲区的当前状态（下标、计数和链表指针等），因此生产者和消费者的所有缓冲区操纵都必须保护起来，以避免竞争状态。

接下来我们给出的使用信号量的方案展示了三种不同类型的信号量。

(1) 名为mutex的二值信号量保护两个临界区：一个是往共享缓冲区中插入一个数据条目（由生产者执行），另一个是从共享缓冲区中移走一个数据条目（由消费者执行）。用作互斥锁的二值信号量初始化为1。（显然，我们可以使用真正的互斥锁代替这样的二值信号量。见习题10.10。）

(2) 名为nempty的计数信号量统计共享缓冲区中的空槽位数。该信号量初始化为缓冲区中的槽位数（NBUFF）。

(3) 名为nstored的计数信号量统计共享缓冲区中已填写的槽位数。该信号量初始化为0，
233 因为缓冲区一开始是空的。

图10-14展示了程序完成初始化时我们的缓冲区及两个计数信号量的状态。我们给未用的数组元素标以阴影。

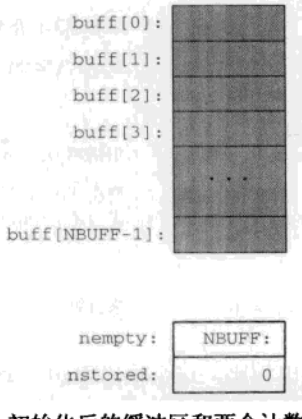


图10-14　初始化后的缓冲区和两个计数信号量

在我们的例子中，生产者只是把0~(NLOOP-1)存放到共享缓冲区中(buff[0] = 0，buff[1] = 1，等等)，并把该缓冲区用作一个环绕缓冲区。消费者从该缓冲区取出这些整数，并验证它们是正确的，若有错误则输出到标准输出上。

图10-15展示了在生产者往共享缓冲区放置了3个条目之后，但在消息者从该缓冲区取走其中任何条目之前该缓冲区和两个计数信号量的状态。

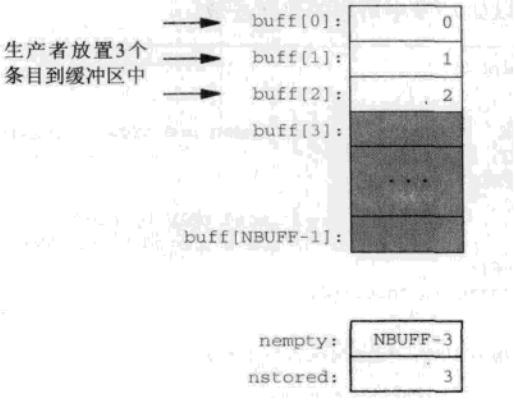


图10-15 生产者放置3个条目到缓冲区后的缓冲区和计数信号量

我们接着假设消费者从共享缓冲区中移走一个条目，图10-16展示这时的状态。

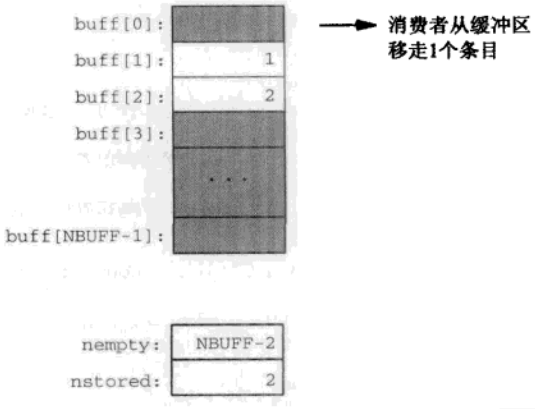


图10-16 消费者从缓冲区移走第一个条目后的缓冲区和计数信号量

图10-17中的main函数创建前述三个信号量，创建两个线程，等待这两个线程的完成，然后删除这些信号量。

全局变量

6~10 可存放NBUFF个条目的缓冲区以及三个信号量指针是生产者线程和消费者线程共享的全局变量。正如第7章中所述，我们把它们收集到一个结构中，以强调这些信号量是用于同步对共享缓冲区的访问的。

创建信号量

19~25 创建三个信号量，它们的名字首先传递给我们的px_ipc_name函数。指定O_EXCL标志，

因为每个信号量都需要初始化为正确的值。如果这三个信号量因先前本程序运行中止而没有全部删除，那么我们可以在创建之前给每个信号量调用sem_unlink，并忽略任何错误。另一种方法是，检查指定了O_EXCL标志的sem_open是否返回一个EEXIST错误，若是则调用sem_unlink，然后再调用一次sem_open，不过这么一来就更复杂了。如果我们需要验证本程序只有一个副本在运行（这一步可在尝试创建任何信号量之前完成），那么可以如9.7节中所述的那样去做。

pxsem/prodcons1.c

```
 1 #include    "unpipc.h"

 2 #define NBUFF    10
 3 #define SEM_MUTEX    "mutex"        /* these are args to px_ipc_name() */
 4 #define SEM_NEMPTY   "nempty"
 5 #define SEM_NSTORED "nstored"

 6 int      nitems;                    /* read-only by producer and consumer */
 7 struct {                            /* data shared by producer and consumer */
 8   int    buff[NBUFF];
 9   sem_t *mutex, *nempty, *nstored;
10 } shared;

11 void    *produce(void *), *consume(void *);

12 int
13 main(int argc, char **argv)
14 {
15     pthread_t   tid_produce, tid_consume;

16     if (argc != 2)
17         err_quit("usage: prodcons1 <#items>");
18     nitems = atoi(argv[1]);

19         /* create three semaphores */
20     shared.mutex = Sem_open(Px_ipc_name(SEM_MUTEX), O_CREAT | O_EXCL,
21                             FILE_MODE, 1);
22     shared.nempty = Sem_open(Px_ipc_name(SEM_NEMPTY), O_CREAT | O_EXCL,
23                              FILE_MODE, NBUFF);
24     shared.nstored = Sem_open(Px_ipc_name(SEM_NSTORED), O_CREAT | O_EXCL,
25                               FILE_MODE, 0);

26         /* create one producer thread and one consumer thread */
27     Set_concurrency(2);
28     Pthread_create(&tid_produce, NULL, produce, NULL);
29     Pthread_create(&tid_consume, NULL, consume, NULL);

30         /* wait for the two threads */
31     Pthread_join(tid_produce, NULL);
32     Pthread_join(tid_consume, NULL);

33         /* remove the semaphores */
34     Sem_unlink(Px_ipc_name(SEM_MUTEX));
35     Sem_unlink(Px_ipc_name(SEM_NEMPTY));
36     Sem_unlink(Px_ipc_name(SEM_NSTORED));
37     exit(0);
38 }
```

pxsem/prodcons1.c

图10-17　生产者-消费者问题信号量解决方案的main函数

创建两个线程

26~29　创建两个线程，一个作为生产者，一个作为消费者。不给这两个线程传递任何参数。

30~36 主线程然后等待这两个线程的终止，接着删除一开始创建的三个信号量。

我们还可以给每个线程调用sem_close，不过进程终止时这会自动发生。然而删除一个有名信号量的名字却必须显式地完成。

图10-18给出了produce和comsume函数。

pxsem/prodcons1.c

```
39 void *
40 produce(void *arg)
41 {
42     int     i;

43     for (i = 0; i < nitems; i++) {
44         Sem_wait(shared.nempty);     /* wait for at least 1 empty slot */
45         Sem_wait(shared.mutex);
46         shared.buff[i % NBUFF] = i;     /* store i into circular buffer */
47         Sem_post(shared.mutex);
48         Sem_post(shared.nstored);    /* 1 more stored item */
49     }
50     return(NULL);
51 }

52 void *
53 consume(void *arg)
54 {
55     int     i;

56     for (i = 0; i < nitems; i++) {
57         Sem_wait(shared.nstored);       /* wait for at least 1 stored item */
58         Sem_wait(shared.mutex);
59         if (shared.buff[i % NBUFF] != i)
60             printf("buff[%d] = %d\n", i, shared.buff[i % NBUFF]);
61         Sem_post(shared.mutex);
62         Sem_post(shared.nempty);        /* 1 more empty slot */
63     }
64     return(NULL);
65 }
```

pxsem/prodcons1.c

图10-18 produce和consume函数

生产者等待到缓冲区中有一个条目的空间

44 生产者在nempty信号量上调用sem_wait，等待缓冲区中有存放另一个条目的可用空间为止。首次执行本行语句时，该信号量的值将从NBUFF变为NBUFF-1。

生产者在缓冲区中存放条目

45~48 在往缓冲区中存放新条目之前，生产者必须获取mutex信号量。在我们的例子中，生产者只是往下标为i % NBUFF的数组元素中存放一个值，因此不需要描述缓冲区状态的共享变量（也就是说我们不使用每次往缓冲区中放置一个条目就得更新状态的链表）。这样的话，获取和释放mutex信号量实际上没有必要。不过我们还是给出了，因为这种类型的问题（更新由多个线程共享的一个缓冲区）通常需要这么做。

往缓冲区中存放当前条目后，释放mutex信号量（其值由0变为1），并挂出nstored信号量。第一次执行这个语句时，nstored的值将从初始值0变为1。

消费者等待nstored信号量

57~62 当nstored信号量的值大于0时，缓冲区中已有那么多的条目待处理。消费者从缓冲区

235～237

中取出一个条目并验证它的值是正确的，不过这样的缓冲区访问是用mutex信号量保护起来的。之后消费者挂出nempty信号量，告诉生产者又有一个空槽位可用了。

死锁

如果我们错误地对换了消费者函数（图10-18）中两个Sem_wait调用的顺序，那会发生什么呢？假设生产者首先启动（跟习题10.1的解答中所假设的一样），那么它将往缓冲区中存放NBUFF个条目，从而把nempty信号量的值从NBUFF递减为0，把nstored信号量的值从0递增为NBUFF。至此，生产者阻塞在Sem_wait(shared.nempty)调用中，因为缓冲区满，其上没有存放另一个条目的空槽位可用。

消费者启动，验证缓冲区中第一批NBUFF个条目的正确性。这个过程把nstored信号量的值从NBUFF递减为0，把nempty信号量的值从0递增为NBUFF。消费者接着在调用Sem_wait(shared.mutex)之后阻塞在Sem_wait(shared.nstored)调用中。生产者可以恢复执行了，因为nempty的值现已大于0，然而生产者接着调用的是Sem_wait(shared.mutex)，于是阻塞。

这种现象就是死锁（dead lock）。生产者在等待mutex信号量，但是消费者却持有该信号量并在等待nstored信号量。然而生产者只有获取了mutex信号量才能挂出nstored信号量。这就是使用信号量的问题之一：要是编写代码时出了差错，程序就不能正确工作。

Posix允许sem_wait检测死锁并返回EDEADLK错误，但是运行本例子所用的系统（Solaris 2.6和Digital Unix 4.0B）都不能检测这种错误。

10.7　文件上锁

现在回到第9章中的序列号问题，我们提供了使用Posix有名信号量实现的my_lock和my_unlock函数。图10-19给出了这两个函数。

lock/lockpxsem.c

```
1 #include    "unpipc.h"
2 #define LOCK_PATH    "pxsemlock"
3 sem_t       *locksem;
4 int         initflag;
5 void
6 my_lock(int fd)
7 {
8     if (initflag == 0) {
9         locksem = Sem_open(Px_ipc_name(LOCK_PATH), O_CREAT, FILE_MODE, 1);
10        initflag = 1;
11    }
12    Sem_wait(locksem);
13 }
14 void
15 my_unlock(int fd)
16 {
17    Sem_post(locksem);
18 }
```

lock/lockpxsem.c

图10-19　使用Posix有名信号量的文件上锁

这两个函数采用一个作为劝告性文件锁使用的信号量，当首先调用my_lock函数时，该信号量的值被初始化为1。为获取该文件锁，我们调用sem_wait；为释放该锁，我们调用sem_post。

10.8 sem_init和sem_destroy函数

本章此前的内容处理的是Posix有名信号量。这些信号量由一个*name*参数标识，它通常指代文件系统中的某个文件。然而Posix也提供基于内存的信号量，它们由应用程序分配信号量的内存空间（也就是分配一个sem_t数据类型的内存空间），然后由系统初始化它们的值。

```
#include <semaphore.h>

int sem_init(sem_t *sem, int shared, unsigned int value);
                                                    返回：若出错则为-1

int sem_destroy(sem_t *sem);
                                                    返回：若成功则为0，若出错则为-1
```

基于内存的信号量是由sem_init初始化的。*sem*参数指向应用程序必须分配的sem_t变量。如果*shared*为0，那么待初始化的信号量是在同一进程的各个线程间共享的，否则该信号量是在进程间共享的。当*shared*为非零时，该信号量必须存放在某种类型的共享内存区中，而即将使用它的所有进程都要能访问该共享内存区。跟sem_open一样，*value*参数是该信号量的初始值。

使用完一个基于内存的信号量后，我们调用sem_destroy摧毁它。

sem_open不需要类似于*shared*的参数或类似于PTHREAD_PROCESS_SHARED的属性（第7章中讲述的互斥锁和条件变量可使用该属性），因为有名信号量总是可以在不同进程间共享的。

注意，基于内存的信号量不使用任何类似于O_CREAT标志的东西，也就是说，sem_init总是初始化信号量的值。因此，对于一个给定的信号量，我们必须小心保证只调用sem_init一次。（习题10.2展示了对于有名信号量的这个差别。）对一个已初始化过的信号量调用sem_init，其结果是未定义的。

你得确保理解sem_open和sem_init之间的下述基本差异。前者返回一个指向某个sem_t变量的指针，该变量由（sem_open）函数本身分配并初始化。后者的第一个参数是一个指向某个sem_t变量的指针，该变量由调用者分配，然后由（sem_init）函数初始化。

Posix.1警告说，对于一个基于内存的信号量，只有sem_init的*sem*参数指向的位置可用于访问该信号量，使用它的sem_t数据类型副本访问时结果未定义。

sem_init出错时返回-1，但成功时并不返回0。这确实有些奇怪，Posix.1基本原理一文中有一个注解说，将来的某个修订版可能指定调用成功时返回0。

238 ~ 239

当不需要使用与有名信号量关联的名字时，可改用基于内存的信号量。彼此无亲缘关系的不同进程需使用信号量时，通常使用有名信号量。其名字就是各个进程标识信号量的手段。

我们在图1-3中说过，基于内存的信号量至少具有随进程的持续性，然而它们真正的持续性却取决于存放信号量的内存区的类型。只要含有某个基于内存信号量的内存区保持有效，该信号量就一直存在。

- 如果某个基于内存的信号量是由单个进程内的各个线程共享的（sem_init的*shared*的参数为0），那么该信号量具有随进程的持续性，当该进程终止时它也消失。
- 如果某个基于内存的信号量是在不同进程间共享的（sem_init的*shared*参数为1），那么该信号量必须存放在共享内存区中，因而只要该共享内存区仍然存在，该信号量也就继

续存在。从图1-3可以看出，Posix共享内存区和System V共享内存区都具有随内核的持续性。这意味着服务器可以创建一个共享内存区，在该共享内存区中初始化一个Posix基于内存的信号量，然后终止。一段时间后，一个或多个客户可打开该共享内存区，访问存放在其中的基于内存的信号量。

小心，下面的代码并不像预期的那样工作。

```
sem_t  mysem;

    Sem_init(&mysem, 1, 0);      /* 2nd arg of 1 -> shared between processes */
    if (Fork() == 0) {           /* child */
       . . .
       Sem_post(&mysem);
    }
    Sem_wait(&mysem);            /* parent; wait for child */
```

问题在于信号量mysem不在共享内存区中，正确的代码见10.12节。fork出来的子进程通常不共享父进程的内存空间。子进程是在父进程内存空间的副本上启动的，它跟共享内存区不是一回事。我们将在本书第4部分详细讨论共享内存区。

240

例子

作为一个例子，我们把图10-17和图10-18中的生产者-消费者例子程序转换成使用基于内存的信号量。图10-20给出了这个程序。

pxsem/prodcons2.c

```
 1 #include    "unpipc.h"
 2 #define NBUFF     10
 3 int     nitems;                    /* read-only by producer and consumer */
 4 struct {                           /* data shared by producer and consumer */
 5     int     buff[NBUFF];
 6     sem_t   mutex, nempty, nstored;         /* semaphores, not pointers */
 7 } shared;
 8 void    *produce(void *), *consume(void *);
 9 int
10 main(int argc, char **argv)
11 {
12     pthread_t   tid_produce, tid_consume;
13     if (argc != 2)
14         err_quit("usage: prodcons2 <#items>");
15     nitems = atoi(argv[1]);
16         /* initialize three semaphores */
17     Sem_init(&shared.mutex, 0, 1);
18     Sem_init(&shared.nempty, 0, NBUFF);
19     Sem_init(&shared.nstored, 0, 0);
20     Set_concurrency(2);
21     Pthread_create(&tid_produce, NULL, produce, NULL);
22     Pthread_create(&tid_consume, NULL, consume, NULL);
23     Pthread_join(tid_produce, NULL);
24     Pthread_join(tid_consume, NULL);
```

图10-20　使用基于内存信号量的生产者-消费者程序

```
25      Sem_destroy(&shared.mutex);
26      Sem_destroy(&shared.nempty);
27      Sem_destroy(&shared.nstored);
28      exit(0);
29  }

30  void *
31  produce(void *arg)
32  {
33      int     i;

34      for (i = 0; i < nitems; i++) {
35          Sem_wait(&shared.nempty);       /* wait for at least 1 empty slot */
36          Sem_wait(&shared.mutex);
37          shared.buff[i % NBUFF] = i;     /* store i into circular buffer */
38          Sem_post(&shared.mutex);
39          Sem_post(&shared.nstored);      /* 1 more stored item */
40      }
41      return(NULL);
42  }

43  void *
44  consume(void *arg)
45  {
46      int     i;

47      for (i = 0; i < nitems; i++) {
48          Sem_wait(&shared.nstored);      /* wait for at least 1 stored item */
49          Sem_wait(&shared.mutex);
50          if (shared.buff[i % NBUFF] != i)
51              printf("buff[%d] = %d\n", i, shared.buff[i % NBUFF]);
52          Sem_post(&shared.mutex);
53          Sem_post(&shared.nempty);       /* 1 more empty slot */
54      }
55      return(NULL);
56  }
```

pxsem/prodcons2.c

241

图10-20（续）

分配信号量

6 本程序所用的三个信号量现在声明为三个sem_t数据类型的变量，而不是以前的三个sem_t数据类型指针。

调用sem_init

16~27 把原来的sem_open调用改为sem_init，sem_unlink调用改为sem_destroy。这几个sem_destroy调用实际上没有必要，因为程序马上就结束了。

其余变动是在所有的sem_wait和sem_post调用中传递指向三个信号量变量的指针。

10.9 多个生产者，单个消费者

10.6节中的生产者-消费者方案解决的是经典的单个生产者单个消费者问题。对它作修改后允许有多个生产者和单个消费者。我们从图10-20使用基于内存信号量的方案着手。图10-21给出了全局变量和main函数。

pxsem/prodcons3.c

```
 1 #include    "unpipc.h"
 2 #define NBUFF        10
 3 #define MAXNTHREADS 100
 4 int     nitems, nproducers;            /* read-only by producer and consumer */
 5 struct {                               /* data shared by producers and consumer */
 6     int     buff[NBUFF];
 7     int     nput;
 8     int     nputval;
 9     sem_t   mutex, nempty, nstored;        /* semaphores, not pointers */
10 } shared;
11 void   *produce(void *), *consume(void *);
12 int
13 main(int argc, char **argv)
14 {
15     int     i, count[MAXNTHREADS];
16     pthread_t tid_produce[MAXNTHREADS], tid_consume;
17     if (argc != 3)
18         err_quit("usage: prodcons3 <#items> <#producers>");
19     nitems = atoi(argv[1]);
20     nproducers = min(atoi(argv[2]), MAXNTHREADS);
21         /* initialize three semaphores */
22     Sem_init(&shared.mutex, 0, 1);
23     Sem_init(&shared.nempty, 0, NBUFF);
24     Sem_init(&shared.nstored, 0, 0);
25         /* create all producers and one consumer */
26     Set_concurrency(nproducers + 1);
27     for (i = 0; i < nproducers; i++) {
28         count[i] = 0;
29         Pthread_create(&tid_produce[i], NULL, produce, &count[i]);
30     }
31     Pthread_create(&tid_consume, NULL, consume, NULL);
32         /* wait for all producers and the consumer */
33     for (i = 0; i < nproducers; i++) {
34         Pthread_join(tid_produce[i], NULL);
35         printf("count[%d] = %d\n", i, count[i]);
36     }
37     Pthread_join(tid_consume, NULL);
38     Sem_destroy(&shared.mutex);
39     Sem_destroy(&shared.nempty);
40     Sem_destroy(&shared.nstored);
41     exit(0);
42 }
```

pxsem/prodcons3.c

图10-21　创建多个生产者线程的main函数

全局变量

4　全局变量nitems是所有生产者生产的总条目数，nproducers是生产者线程的总数。它们都是根据命令行参数设置的。

共享的结构

5~10　在shared结构中定义了两个新变量：nput和nputval。nput是下一个待存入值的缓冲

区项的下标（按NBUFF求模），nputval则是下一个待存入缓冲区的值。这两个变量来自图7-2和图7-3中的解决方案。它们用于同步多个生产者线程。 242 / 243

新的命令行参数

17~20 两个新的命令行参数分别指定待存入缓冲区的总条目数以及待创建生产者线程的总数。

创建所有线程

21~41 初始化各个信号量，创建所有的生产者线程和唯一的消费者线程。然后等待所有线程终止。这段代码与图7-2几乎相同。

图10-22给出了由每个生产者线程执行的produce函数。

pxsem/prodcons3.c

```
43 void *
44 produce(void *arg)
45 {
46     for ( ; ; ) {
47         Sem_wait(&shared.nempty);       /* wait for at least 1 empty slot */
48         Sem_wait(&shared.mutex);

49         if (shared.nput >= nitems) {
50             Sem_post(&shared.nempty);
51             Sem_post(&shared.mutex);
52             return(NULL);        /* all done */
53         }

54         shared.buff[shared.nput % NBUFF] = shared.nputval;
55         shared.nput++;
56         shared.nputval++;

57         Sem_post(&shared.mutex);
58         Sem_post(&shared.nstored);      /* 1 more stored item */
59         *((int *) arg) += 1;
60     }
61 }
```

pxsem/prodcons3.c

图10-22 所有生产者线程都执行的函数

生产者线程间的互斥

49~53 与图10-18相比的改变是，当所有线程总共往缓冲区中放置了nitems个值后，循环即终止。注意，能同时获取nempty信号量的生产者线程可能有多个，但每个时刻只有一个生产者线程能获取mutex信号量。这么一来，变量nput和nputval就不会同时受不止一个生产者线程的修改。

生产者线程的终止

50~51 我们必须仔细处理生产者线程的终止。最后一个条目生产出来后，每个生产者线程都执行循环顶端的如下一行语句：

```
Sem_wait(&shared.nempty);   /* wait for at least 1 empty slot */
```

它将nempty信号量减1。然而每个生产者线程在终止前必须给该信号量加1，因为它在最后一次走过循环时并没有往缓冲区中存入一个条目。即将终止的生产者线程还得释放mutex信号量，以允许其他生产者线程继续运行。要是线程终止时我们没有给nempty信号量加1，而且生产者线程数大于缓冲区槽位数（譬如说14个生产者线程和10个缓冲区槽位），那么多余的线程（4个）将永远阻塞，等待nempty信号量，从而永 244

远终止不了。[①]

图10-23中的consume函数只是验证缓冲区中每个项都是正确的，如果检测到错误就输出一个消息。

pxsem/prodcons3.c

```
62 void *
63 consume(void *arg)
64 {
65     int     i;

66     for (i = 0; i < nitems; i++) {
67         Sem_wait(&shared.nstored);      /* wait for at least 1 stored item */
68         Sem_wait(&shared.mutex);

69         if (shared.buff[i % NBUFF] != i)
70             printf("error: buff[%d] = %d\n", i, shared.buff[i % NBUFF]);

71         Sem_post(&shared.mutex);
72         Sem_post(&shared.nempty);       /* 1 more empty slot */
73     }
74     return(NULL);
75 }
```

pxsem/prodcons3.c

图10-23　唯一的消费者线程执行的函数

这个唯一的消费者线程的终止条件非常简单——它只需统计已消费的条目数。

10.10　多个生产者，多个消费者

对于生产者-消费者问题的进一步修改是允许多个生产者和多个消费者。具有多个消费者是否有意义取决于具体应用。作者看到过使用这种技巧的两个应用程序。

(1) 一个把IP地址转换成对应主机名的程序。每个消费者取一个IP地址，调用gethostbyaddr（UNPv1的9.6节），然后往某个文件中添加得出的主机名。由于每次调用gethostbyaddr所花时间可能不一样，因此缓冲区中IP地址的顺序通常与各个消费者线程存入结果的文件中的主机名顺序不一致。这种情形的优势在于多个gethostbyaddr调用（每个调用可能得花数秒）可以并行地发生：每个消费者线程一个调用。

> 这里假设gethostbyaddr是一个可重入版本的函数，然而不是所有实现都具备这个属性。要是可重入版本的gethostbyaddr函数不可用，候选方法之一就是将缓冲区存放在共享内存区中，并改用多个进程代替多个线程。

245

(2) 一个读出UDP数据报，对它们进行操作后把结果写入某个数据库的程序。每个数据报由一个消费者线程处理，为重叠可能很花时间的每个数据报的处理，需要有多个消费者线程。尽管由消费者线程们写入数据库中的数据报顺序通常不同于原来的数据报顺序，数据库中的记录排序功能却能处理顺序问题。

图10-24给出了全局变量。

① 试想，消费者消费掉生产者产生的所有数据时，nempty和nstore的值肯定恢复成生产-消费过程尚未开始时的值（分别为NBUFF和0）。也就是说生产-消费过程的结束状态与开始状态是一致的。此后所有生产者线程都执行第47行的Sem_wait，其中有nempty个线程返回，其余线程则阻塞。未阻塞的线程要是不执行第50行的Sem_post，已阻塞的线程将永远阻塞下去。永远阻塞的线程数于是为生产-消费过程结束时的nempty=nproducers-NBUFF。——译者注

pxsem/prodcons4.c

```
 1 #include        "unpipc.h"
 2 #define NBUFF         10
 3 #define MAXNTHREADS 100
 4 int     nitems, nproducers, nconsumers;          /* read-only */
 5 struct {                        /* data shared by producers and consumers */
 6     int     buff[NBUFF];
 7     int     nput;               /* item number: 0, 1, 2, ... */
 8     int     nputval;            /* value to store in buff[] */
 9     int     nget;               /* item number: 0, 1, 2, ... */
10     int     ngetval;            /* value fetched from buff[] */
11   sem_t     mutex, nempty, nstored;         /* semaphores, not pointers */
12 } shared;
13 void    *produce(void *), *consume(void *);
```

pxsem/prodcons4.c

图10-24 全局变量

全局变量和共享的结构

4~12 消费者线程数现在是一个全局变量，它是根据一个命令行参数设置的。我们还往shared结构中加了另外两个变量：nget和ngetval。nget是任意一个消费者线程待取出的下一个条目的编号，ngetval则存放相应的值。

图10-25给出的main函数已修改成创建多个消费者线程。

pxsem/prodcons4.c

```
14 int
15 main(int argc, char **argv)
16 {
17     int     i, prodcount[MAXNTHREADS], conscount[MAXNTHREADS];
18     pthread_t tid_produce[MAXNTHREADS], tid_consume[MAXNTHREADS];
19     if (argc != 4)
20         err_quit("usage: prodcons4 <#items> <#producers> <#consumers>");
21     nitems = atoi(argv[1]);
22     nproducers = min(atoi(argv[2]), MAXNTHREADS);
23     nconsumers = min(atoi(argv[3]), MAXNTHREADS);
24         /* initialize three semaphores */
25     Sem_init(&shared.mutex, 0, 1);
26     Sem_init(&shared.nempty, 0, NBUFF);
27     Sem_init(&shared.nstored, 0, 0);
28         /* create all producers and all consumers */
29     Set_concurrency(nproducers + nconsumers);
30     for (i = 0; i < nproducers; i++) {
31         prodcount[i] = 0;
32         Pthread_create(&tid_produce[i], NULL, produce, &prodcount[i]);
33     }
34     for (i = 0; i < nconsumers; i++) {
35         conscount[i] = 0;
36         Pthread_create(&tid_consume[i], NULL, consume, &conscount[i]);
37     }
38         /* wait for all producers and all consumers */
39     for (i = 0; i < nproducers; i++) {
40         Pthread_join(tid_produce[i], NULL);
41         printf("producer count[%d] = %d\n", i, prodcount[i]);
42     }
43     for (i = 0; i < nconsumers; i++) {
```

图10-25 创建多个生产者和多个消费者的main函数

```
44          Pthread_join(tid_consume[i], NULL);
45          printf("consumer count[%d] = %d\n", i, conscount[i]);
46      }

47      Sem_destroy(&shared.mutex);
48      Sem_destroy(&shared.nempty);
49      Sem_destroy(&shared.nstored);
50      exit(0);
51  }
```

pxsem/prodcons4.c

图10-25（续）

19~23　增设一个新的命令行选项，由它指定待创建消费者线程的总数。我们必须分配一个数组（tid_consume）以保存所有消费者的线程ID，再分配一个数组（conscount）以保存每个消费者线程处理的条目数，作为诊断计数。

24~50　创建多个生产者线程和多个消费者线程，然后等待它们的完成。

246 ~ 247

我们的生产者函数含有图10-22中没有的一个新行。当所有的生产者线程完成生产工作时，前面标以加号的那一行是新加的：

```
    if (shared.nput >= nitems) {
+       Sem_post(&shared.nstored);      /* let consumers terminate */
        Sem_post(&shared.nempty);
        Sem_post(&shared.mutex);
        return(NULL);               /* all done */
    }
```

在处理生产者线程和消费者线程的终止时，我们仍得小心。缓冲区中所有条目都被消费掉之后，每个消费者线程将阻塞在如下调用中：

```
Sem_wait(&shared.nstored);      /* wait for at least 1 stored item */
```

我们让每个生产者线程给nstored信号量加1以给各个消费者线程解阻塞，以此让消费者们看到生产者们已完成生产工作。

图10-26给出了我们的消费者函数。

pxsem/prodcons4.c

```
72  void *
73  consume(void *arg)
74  {
75      int     i;

76      for ( ; ; ) {
77          Sem_wait(&shared.nstored);      /* wait for at least 1 stored item */
78          Sem_wait(&shared.mutex);

79          if (shared.nget >= nitems) {
80              Sem_post(&shared.nstored);
81              Sem_post(&shared.mutex);
82              return(NULL);       /* all done */
83          }
84          i = shared.nget % NBUFF;
85          if (shared.buff[i] != shared.ngetval)
86              printf("error: buff[%d] = %d\n", i, shared.buff[i]);
87          shared.nget++;
88          shared.ngetval++;

89          Sem_post(&shared.mutex);
90          Sem_post(&shared.nempty);       /* 1 more empty slot */
91          *((int *) arg) += 1;
92      }
93  }
```

pxsem/prodcons4.c

图10-26　所有消费者线程都执行的函数

消费者线程的终止

79~83 新的消费者函数必须比较nget和nitems，以确定所有消费者线程完成消费工作的时刻（类似于生产者函数）。缓冲区中最后一个条目被消费掉之后，各个消费者线程阻塞，等待nstored信号量变为大于0。这么一来，每个生产者线程终止时，应给nstored加1以让一个消费者线程终止。①

248

10.11 多个缓冲区

在处理一些数据的典型程序中，我们可以找到一个如下形式的循环：

```
while ( (n = read(fdin, buff, BUFFSIZE)) > 0) {
    /* process the data */
    write(fdout, buff, n);
}
```

举例来说，处理文本文件的许多程序读入一行输入，对它进行处理，然后写出一行输出。对于文本文件，read和write调用往往被替换成对标准I/O函数fgets和fputs的调用。

图10-27展示了实现这种操作的一种方法，其中名为reader的函数从输入文件读入数据，名为writer的函数往输出文件写出数据。总共使用一个缓冲区。

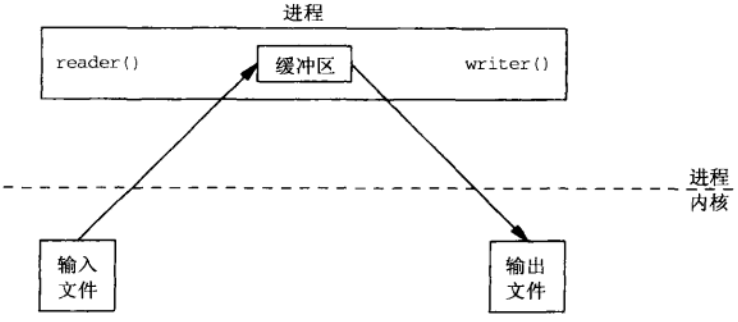


图10-27 由一个进程把数据读入某个缓冲区，再从该缓冲区写出

图10-28给出了整个操作的时间线图。我们在时间线的左边按从上到下的时间增长顺序标出了数值，时间单位则是某个任意值。我们假设一个读操作花5个单位时间，一个写操作花7个单位时间，读和写之间的处理花2个单位时间。

我们可以把这个应用修改成在两个线程间分割读写操作，如图10-29所示。这儿我们使用两个线程，因为各个线程自动共享同一个全局缓冲区。我们也可以把复制操作分割到两个进程中，不过那将需要使用还没有讨论的共享内存区。

① 生产-消费过程刚结束时，nempty和nstored恢复为生产-消费过程尚未开始时的值（分别为NBUFF和0）。生产者函数produce中新增的Sem_post行使得只要有一个生产者线程终止，nstored的值就大于0（当所有生产者线程都终止时，nstored的值将变为nproducers）。由于消费者函数consume中第77行的Sem_wait和第80行的Sem_post匹配成对，因此只要nstored大于0，不论有多少个消费者线程，都能一个也不阻塞地全部终止。所有消费者线程都终止时，所有生产者线程不一定都已终止，但至少有一个已终止。——译者注

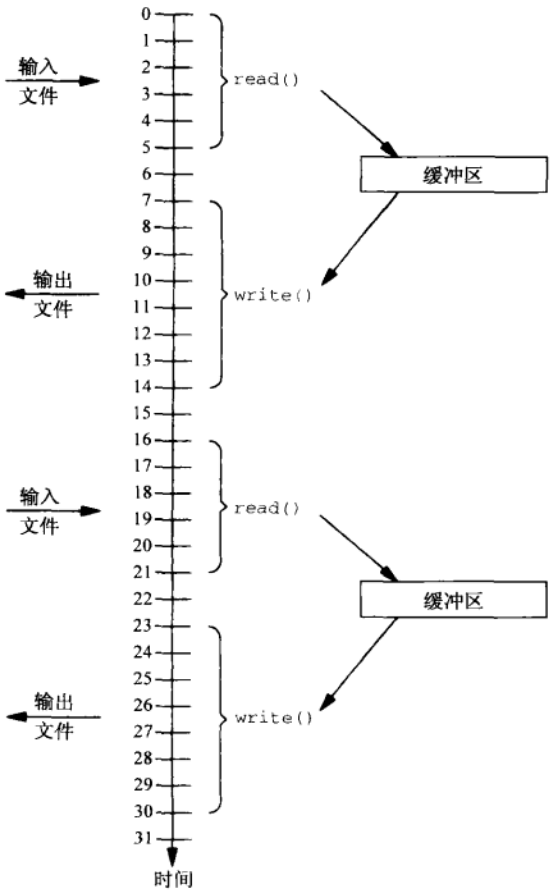


图10-28　由一个进程把数据读入某个缓冲区，再从该缓冲区写出

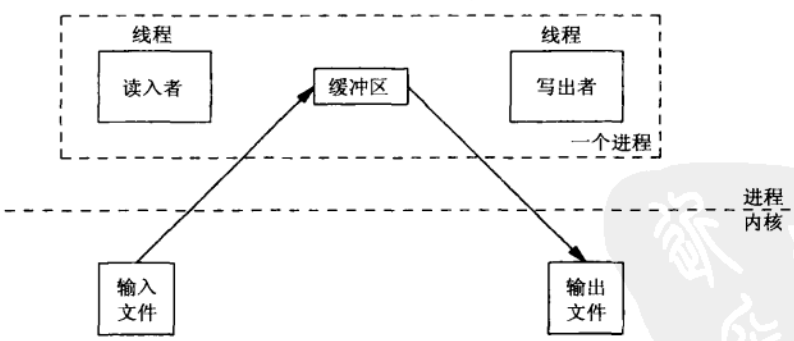


图10-29　把文件复制操作分割到两个线程中

249
~
250

把读写操作分割到两个线程（或两个进程）中还需要线程（或进程）间某种形式的通知。当缓冲区准备好写出时，读入者线程必须通知写出者线程；同样，当缓冲区准备好重新读入时，写出者线程必须通知读入者线程。图10-30给出了这种操作的时间线图。

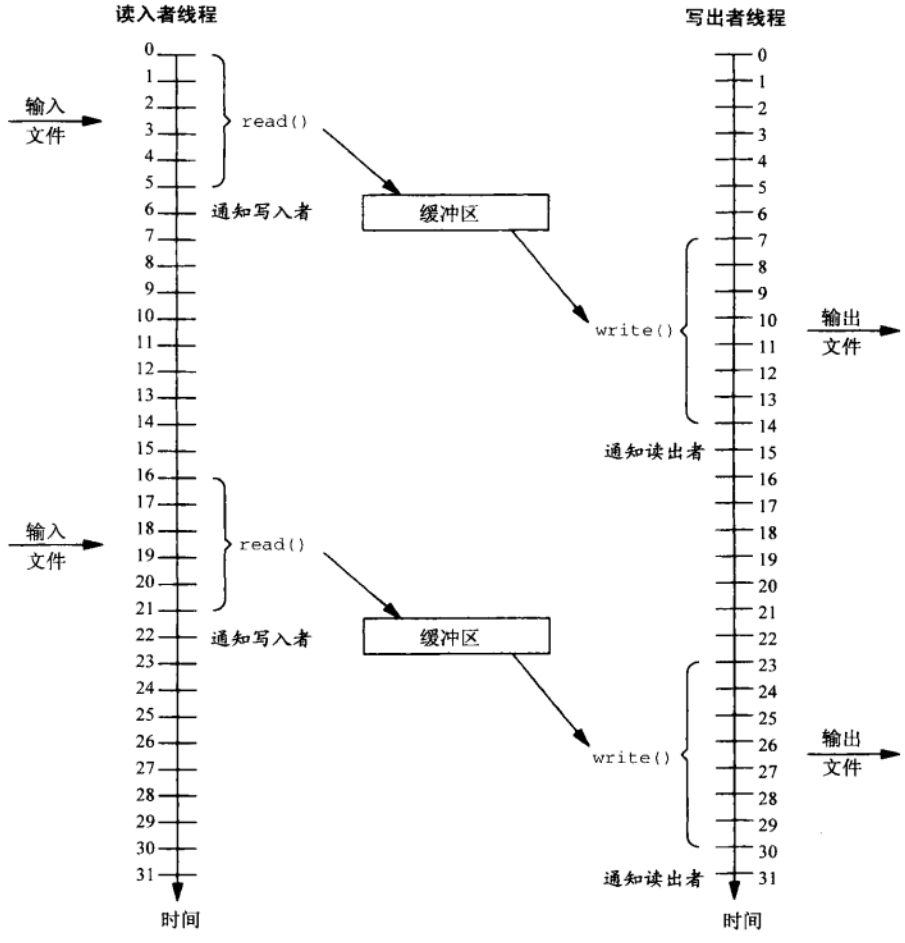


图10-30 把文件复制操作分割到两个线程中

我们假设处理缓冲区中数据以及通知对方线程花两个单位时间。需特别注意的是，把读和写分割到两个线程中并不影响完成整个操作所需时间。我们没有得到任何速度上的优势，只是把整个操作分割到两个线程（或进程）中。

我们在这些时间线图中忽略了许多细微点。例如，大多数Unix内核检测出对一个文件的顺序读后就为读进程执行对下一个磁盘块的异步超前读（read ahead）。这可以改善执行这种类型操作所花的称为“时钟时间（clock time）”的实际时间量。我们还忽略了其他进程对于我们的读入者线程和写出者线程的影响以及内核调度算法的效果。

接下去我们可以把文件复制应用修改成使用两个线程（或进程）和两个缓冲区。这就是经典的双缓冲（double buffering）方案，如图10-31所示。 251

图中画出读入者线程正在往第一个缓冲区中读入数据，写出者线程正在从第二个缓冲区中写出数据。这两个缓冲区随后就在这两个线程间来回切换。

图10-32展示了双缓冲方案的时间线图。读入者首先读入缓冲区1，然后通知写出者缓冲区1准备好处理。读入者随后开始读入缓冲区2，其间写出者在从缓冲区1中写出数据。

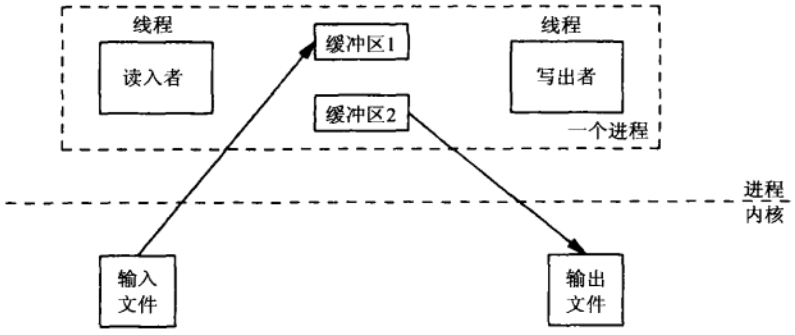


图10-31　使用两个缓冲区把文件复制操作分割到两个线程中

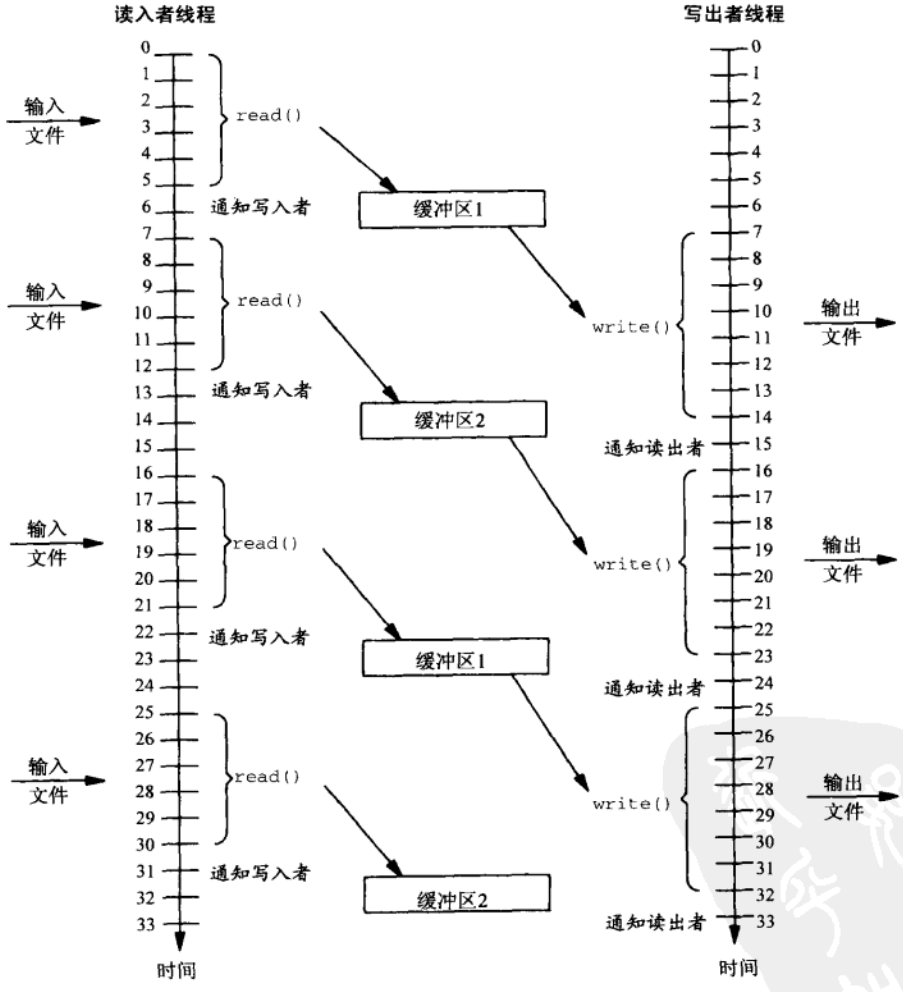


图10-32　双缓冲方案的时间线图

注意，我们不能走得快于最慢的操作，在本例子中就是写操作。服务器完成前两个读操作

后，不得不额外等待2个单位时间：写操作所花时间（7）与读操作所花时间（5）之差。然而对于我们这个假想的例子来说，双缓冲方案所花的总时钟时间几乎只有单缓冲方案的一半。

还要注意，写出操作现在是在尽可能快地发生着，每两个写操作间仅有2个单位时间作为分隔，而图10-28和图10-30中却有9个单位时间作为分隔。这种状况有助于某些像磁带驱动器这样的设备，这些设备在尽可能快地往它们写入数据的条件下会动作得更快（这称为鱼贯（streaming）模式）。

有关双缓冲问题需注意的有趣之事是，它仅仅是生产者-消费者问题的一个特例。

现在开始把以前的生产者-消费者程序修改为处理多个缓冲区。我们从图10-20中使用基于内存信号量的方案入手。新方案能处理任意数量的缓冲区（由NBUFF定义），而不只是个双缓冲方案。图10-33给出了全局变量和main函数。

pxsem/mycat2.c

```
 1 #include    "unpipc.h"

 2 #define NBUFF   8

 3 struct {                          /* data shared by producer and consumer */
 4     struct {
 5         char    data[BUFFSIZE];   /* a buffer */
 6         ssize_t n;                /* count of #bytes in the buffer */
 7     } buff[NBUFF];                /* NBUFF of these buffers/counts */
 8     sem_t   mutex, nempty, nstored;         /* semaphores, not pointers */
 9 } shared;

10 int      fd;                     /* input file to copy to stdout */
11 void    *produce(void *), *consume(void *);

12 int
13 main(int argc, char **argv)
14 {
15     pthread_t   tid_produce, tid_consume;

16     if (argc != 2)
17         err_quit("usage: mycat2 <pathname>");

18     fd = Open(argv[1], O_RDONLY);

19         /* initialize three semaphores */
20     Sem_init(&shared.mutex, 0, 1);
21     Sem_init(&shared.nempty, 0, NBUFF);
22     Sem_init(&shared.nstored, 0, 0);

23         /* one producer thread, one consumer thread */
24     Set_concurrency(2);
25     Pthread_create(&tid_produce, NULL, produce, NULL);    /* reader thread */
26     Pthread_create(&tid_consume, NULL, consume, NULL);    /* writer thread */

27     Pthread_join(tid_produce, NULL);
28     Pthread_join(tid_consume, NULL);

29     Sem_destroy(&shared.mutex);
30     Sem_destroy(&shared.nempty);
31     Sem_destroy(&shared.nstored);
32     exit(0);
33 }
```

pxsem/mycat2.c

图10-33 全局变量和main函数

声明NBUFF个缓冲区

2~9　新的shared结构含有另一个名为buff的结构的数组，而这个新结构含有一个缓冲区和它的当前字节计数。

打开输入文件

18　命令行参数是我们将把其内容复制到标准输出的文件的路径名。

图10-34给出了produce和consume函数。

pxsem/mycat2.c

```
34 void *
35 produce(void *arg)
36 {
37     int     i;

38     for (i = 0; ; ) {
39         Sem_wait(&shared.nempty);       /* wait for at least 1 empty slot */

40         Sem_wait(&shared.mutex);
41             /* critical region */
42         Sem_post(&shared.mutex);

43         shared.buff[i].n = Read(fd, shared.buff[i].data, BUFFSIZE);
44         if (shared.buff[i].n == 0) {
45             Sem_post(&shared.nstored);   /* 1 more stored item */
46             return(NULL);
47         }
48         if (++i >= NBUFF)
49             i = 0;                /* circular buffer */

50         Sem_post(&shared.nstored);       /* 1 more stored item */
51     }
52 }

53 void *
54 consume(void *arg)
55 {
56     int     i;

57     for (i = 0; ; ) {
58         Sem_wait(&shared.nstored);      /* wait for at least 1 stored item */

59         Sem_wait(&shared.mutex);
60             /* critical region */
61         Sem_post(&shared.mutex);

62         if (shared.buff[i].n == 0)
63             return(NULL);
64         Write(STDOUT_FILENO, shared.buff[i].data, shared.buff[i].n);
65         if (++i >= NBUFF)
66             i = 0;             /* circular buffer */

67         Sem_post(&shared.nempty);       /* 1 more empty slot */
68     }
69 }
```

pxsem/mycat2.c

图10-34　produce和consume函数

空临界区

40~42　本例子中由mutex锁住的临界区是空的。要是数据缓冲区是在一个链表上维护的，那么这儿就是我们从该链表中移出某个缓冲区的地方，把该操作放在临界区中是为了避免

与消费者对该链表的操纵发生冲突。然而在我们的例子中，生产者只是使用下一个缓冲区，而且生产者线程只有一个，因此没有什么东西需保护以避免消费者干扰。我们仍然给出了mutex的上锁和解锁，目的是强调本代码的其他修改版本中也许需要这些。

读入数据并给nstored信号量加1

43~49 生产者每获得一个空闲缓冲区就调用read。read返回后生产者将nstored信号量加1，告诉消费者该缓冲区准备好写出。如果read返回0（文件结束符），生产者就在给nstored信号量加1后返回。

消费者线程

57~68 消费者线程取出缓冲区中数据并把它们的内容写到标准输出。碰到长度为0的一个缓冲区表示已到达文件尾。跟生产者函数一样，由mutex保护着的临界区也是空的。

我们在UNPv1的22.3节中开发了一个使用多个缓冲区的例子。在那个例子中，生产者是SIGIO信号处理程序，消费者是主处理循环（do_echo函数）。在生产者和消费者之间共享的变量是nqueue计数器。消费者每次检查或修改该计数器时都阻塞SIGIO信号以防止它产生。 252 ~ 255

10.12 进程间共享信号量

进程间共享基于内存信号量的规则很简单：信号量本身（其地址作为sem_init第一个参数的sem_t数据类型变量）必须驻留在由所有希望共享它的进程所共享的内存区中，而且sem_init的第二个参数必须为1。

这些规则与进程间共享互斥锁、条件变量或读写锁的规则类似：同步对象本身（pthread_mutex_t变量、pthread_cond_t变量或pthread_rwlock_t变量）必须驻留在由所有希望共享它的进程所共享的内存区中，而且该对象必须以PTHREAD_PROCESS_SHARED属性初始化。

至于有名信号量，不同进程（不论彼此间有无亲缘关系）总是能够访问同一个有名信号量，只要它们在调用sem_open时指定相同的名字就行。即使对于某个给定名字的sem_open调用在每个调用进程中可能返回不同的指针，使用该指针的信号量函数（例如sem_post和sem_wait）所引用的仍然是同一个有名信号量。

如果我们在调用sem_open返回指向某个sem_t数据类型变量的指针后接着调用fork，情况又会怎么样呢？Posix.1中有关fork函数的描述这么说："在父进程中打开的任何信号量仍应在子进程中打开。"这意味着如下形式的代码是正确的：

```
sem_t  *mutex;      /* global pointer that is copied across the fork() */

    ...
        /* parent creates named semaphore */
    mutex = Sem_open(Px_ipc_name(NAME), O_CREAT | O_EXCL, FILE_MODE, 0);

    if ( (childpid = Fork()) == 0) {
            /* child */
        ...
        Sem_wait(mutex);
        ...
    }
        /* parent */
    ...
    Sem_post(mutex);
    ...
```

我们必须仔细搞清什么时候可以或者不可以在不同进程间共享某个信号量的原因：一个信号量的状态可能包含在它本身的sem_t数据类型中，但它还可能使用其他信息（例如文件描
256 述符）。我们将在下一章看到，一个进程用于描述一个System V信号量的唯一句柄是由semget返回的它的整数标识符。知道这个标识符的任何进程都可以访问该信号量。System V信号量的所有状态都包含在内核中，它们的整数标识符只是告诉内核具体引用哪个信号量。

10.13　信号量限制

Posix定义了两个信号量限制：

SEM_NSEMS_MAX　一个进程可同时打开着的最大信号量数（Posix要求至少为256）；

SEM_VALUE_MAX　一个信号量的最大值（Posix要求至少为32767）。

这两个常值通常定义在<unistd.h>头文件中，也可在运行时通过调用sysconf函数获取，如下面的例子所示。

例子：semsysconf 程序

图10-35中的程序调用sysconf输出信号量的两个由实现定义的限制值。

pxsem/semsysconf.c

```
1 #include     "unpipc.h"
2 int
3 main(int argc, char **argv)
4 {
5       printf("SEM_NSEMS_MAX = %ld, SEM_VALUE_MAX = %ld\n",
6               Sysconf(_SC_SEM_NSEMS_MAX), Sysconf(_SC_SEM_VALUE_MAX));
7       exit(0);
8 }
```

pxsem/semsysconf.c

图10-35　调用sysconf获取信号量限制值

在我们的两个系统上执行该程序的结果如下：

```
solaris % semsysconf
SEM_NSEMS_MAX = 2147483647, SEM_VALUE_MAX = 2147483647

alpha % semsysconf
SEM_NSEMS_MAX = 256, SEM_VALUE_MAX = 32767
```

10.14　使用 FIFO 实现信号量

现在提供一个使用FIFO完成的Posix有名信号量的实现。每个有名信号量作为一个使用同一
257 名字的FIFO来实现，该FIFO中的非负字节数代表该信号量的当前值。sem_post函数往该FIFO中写入1个字节，sem_wait函数则从该FIFO中读出一个字节（我们希望如果该FIFO为空，那就阻塞调用线程）。sem_open函数在指定了O_CREAT标志的前提下创建待打开的FIFO，然后（不论是否指定了O_CREAT）打开该FIFO两次（一次用于只读，一次用于只写），如果该FIFO是新创建的，那就往其中写入作为信号量初始值指定的那个数目的字节。

本节和本章其余各节含有你第一次阅读时可能希望暂时跳过的高级主题内容。

我们首先在图10-36中给出semaphore.h头文件，它定义了基本的sem_t数据类型。

my_pxsem_fifo/semaphore.h

```
1     /* the fundamental datatype */
2 typedef struct {
3     int     sem_fd[2];        /* two fds: [0] for reading, [1] for writing */
4     int     sem_magic;        /* magic number if open */
5 } sem_t;

6 #define SEM_MAGIC   0x89674523

7 #ifdef  SEM_FAILED
8 #undef  SEM_FAILED
9 #define SEM_FAILED  ((sem_t *)(-1))    /* avoid compiler warnings */
10 #endif
```

my_pxsem_fifo/semaphore.h

图10-36 semaphore.h头文件

sem_t数据类型

1~5 我们的信号量数据结构包含两个描述符，一个用于从实现它的FIFO中读，一个用于往实现它的FIFO中写。为与管道保持一致，我们将这两个描述符存放在某个仅有两个元素的数组中，第一个元素存放读描述符，第二个元素存放写描述符。

一旦该结构初始化，sem_magic成员将含有常值SEM_MAGIC。接受一个sem_t指针作为参数的每个函数都检查该值，从而判定该指针确实指向一个已初始化的信号量结构。当关闭一个信号量时，其sem_t数据结构中的这个成员将被置为0。这种技巧尽管不大完善，却有助于检查一些编程错误。

10.14.1 sem_open 函数

图10-37给出了我们的sem_open函数，它创建一个新的信号量或打开一个已存在的信号量。

创建一个新的信号量

13~17 如果调用者指定了O_CREAT标志，我们就知道需要的是四个而不是两个参数。我们调用va_start初始化ap变量，让它指向最后一个命名过的参数（oflag）。然后使用ap和系统实现提供的va_arg函数获取第三和第四个参数的值。我们已随图5-21描述过可变参数表和我们的va_mode_t数据类型。

创建新的FIFO

18~23 创建一个新的FIFO，其名字为调用者指定给信号量的名字。跟4.6节中讨论过的一样，如果该FIFO已经存在，mkfifo函数将返回一个EEXIST错误。如果sem_open的调用者没有指定O_EXCL标志，那么这种错误无关紧要，不过我们不想以后在本函数中再次初始化该FIFO，于是关掉O_CREAT标志。

分配sem_t数据类型，打开FIFO分别用于读和写

25~37 为一个sem_t数据类型分配空间，它将含有两个描述符。打开新创建或已存在的FIFO两次，一次用于只读，一次用于只写。我们不想阻塞在这两个open调用的任何一个之中，因此在打开该FIFO只读时指定了O_NONBLOCK标志（回想图4-21）。当打开FIFO只写时，我们也指定O_NONBLOCK标志，不过是为了将来检测溢出（例如试图往该管道中写入多于PIPE_BUF个字节）。打开该FIFO两次之后，关掉只读描述符上的非阻塞标志。

初始化新创建信号量的值

38~42 如果创建了一个新的信号量，我们就通过向实现它的FIFO逐个写入总共value个字节来初始化它的值。如果这个初始值超过了系统实现给定的PIPE_BUF限制，那么该FIFO填满后再次调用的write将返回一个EAGAIN错误。

my_pxsem_fifo/sem_open.c

```
1 #include    "unpipc.h"
2 #include    "semaphore.h"
3 #include    <stdarg.h>          /* for variable arg lists */
4 sem_t   *
5 sem_open(const char *pathname, int oflag, ... )
6 {
7     int     i, flags, save_errno;
8     char    c;
9     mode_t  mode;
10    va_list ap;
11    sem_t   *sem;
12    unsigned int  value;
13    if (oflag & O_CREAT) {
14        va_start(ap, oflag);       /* init ap to final named argument */
15        mode = va_arg(ap, va_mode_t);
16        value = va_arg(ap, unsigned int);
17        va_end(ap);
18        if (mkfifo(pathname, mode) < 0) {
19            if (errno == EEXIST && (oflag & O_EXCL) == 0)
20                oflag &= ~O_CREAT;/* already exists, OK */
21            else
22                return(SEM_FAILED);
23        }
24    }
25    if ( (sem = malloc(sizeof(sem_t))) == NULL)
26        return(SEM_FAILED);
27    sem->sem_fd[0] = sem->sem_fd[1] = -1;
28    if ( (sem->sem_fd[0] = open(pathname, O_RDONLY | O_NONBLOCK)) < 0)
29        goto error;
30    if ( (sem->sem_fd[1] = open(pathname, O_WRONLY | O_NONBLOCK)) < 0)
31        goto error;
32        /* turn off nonblocking for sem_fd[0] */
33    if ( (flags = fcntl(sem->sem_fd[0], F_GETFL, 0)) < 0)
34        goto error;
35    flags &= ~O_NONBLOCK;
36    if (fcntl(sem->sem_fd[0], F_SETFL, flags) < 0)
37        goto error;
38    if (oflag & O_CREAT) {         /* initialize semaphore */
39        for (i = 0; i < value; i++)
40            if (write(sem->sem_fd[1], &c, 1) != 1)
41                goto error;
42    }
43    sem->sem_magic = SEM_MAGIC;
44    return(sem);
45 error:
46    save_errno = errno;
47    if (oflag & O_CREAT)
48        unlink(pathname);          /* if we created FIFO */
49    close(sem->sem_fd[0]);         /* ignore error */
50    close(sem->sem_fd[1]);         /* ignore error */
51    free(sem);
52    errno = save_errno;
53    return(SEM_FAILED);
54 }
```

my_pxsem_fifo/sem_open.c

图10-37　sem_open函数

10.14.2 sem_close 函数

图10-38给出了我们的sem_close函数。

my_pxsem_fifo/sem_close.c

```
 1 #include    "unpipc.h"
 2 #include    "semaphore.h"

 3 int
 4 sem_close(sem_t *sem)
 5 {
 6     if (sem->sem_magic != SEM_MAGIC) {
 7         errno = EINVAL;
 8         return(-1);
 9     }
10     sem->sem_magic = 0;          /* in case caller tries to use it later */
11     if (close(sem->sem_fd[0]) == -1 || close(sem->sem_fd[1]) == -1) {
12         free(sem);
13         return(-1);
14     }
15     free(sem);
16     return(0);
17 }
```

my_pxsem_fifo/sem_close.c

图10-38 sem_close函数

11~15 close由调用者指定的信号量结构中的两个FIFO描述符，free分配给这个sem_t数据类型的空间。

10.14.3 sem_unlink 函数

图10-39给出了我们的sem_unlink函数，由它删除与某个信号量关联的名字。它只是调用Unix的unlink函数。

my_pxsem_fifo/sem_unlink.c

```
1 #include    "unpipc.h"
2 #include    "semaphore.h"

3 int
4 sem_unlink(const char *pathname)
5 {
6     return(unlink(pathname));
7 }
```

my_pxsem_fifo/sem_unlink.c

图10-39 sem_unlink函数

10.14.4 sem_post 函数

图10-40给出了我们的sem_post函数，由它将某个信号量的值加1。

11~12 往与由调用者指定的信号量相关联的FIFO中写入一个任意字节。如果该FIFO先前是空的，那么这个写入操作将唤醒阻塞在该FIFO上的read调用中等待一个数据字节的任意进程。

my_pxsem_fifo/sem_post.c

```
 1 #include    "unpipc.h"
 2 #include    "semaphore.h"

 3 int
 4 sem_post(sem_t *sem)
 5 {
 6     char    c;

 7     if (sem->sem_magic != SEM_MAGIC) {
 8         errno = EINVAL;
 9         return(-1);
10     }
11     if (write(sem->sem_fd[1], &c, 1) == 1)
12         return(0);
13     return(-1);
14 }
```

my_pxsem_fifo/sem_post.c

258 ~ 261

图10-40　sem_post函数

10.14.5　sem_wait函数

图10-41给出了我们的最后一个函数sem_wait。

my_pxsem_fifo/sem_wait.c

```
 1 #include    "unpipc.h"
 2 #include    "semaphore.h"

 3 int
 4 sem_wait(sem_t *sem)
 5 {
 6     char    c;

 7     if (sem->sem_magic != SEM_MAGIC) {
 8         errno = EINVAL;
 9         return(-1);
10     }
11     if (read(sem->sem_fd[0], &c, 1) == 1)
12         return(0);
13     return(-1);
14 }
```

my_pxsem_fifo/sem_wait.c

图10-41　sem_wait函数

11~12　从与由调用者指定的信号量相关联的FIFO中read 1个字节，如果该FIFO已为空，那就阻塞。

我们没有实现sem_trywait函数，但是通过启用与其信号量关联的FIFO的非阻塞标志后再调用read，就能做到。我们也没有实现sem_getvalue函数。当调用stat或fstat函数时，某些实现会返回当前在某个管道或FIFO中的字节数，它是作为所返回stat结构的st_size成员给出的。然而Posix并不保证这一点，因而是不可移植的。下一节中将给出这两个Posix信号量函数的实现。

10.15　使用内存映射I/O实现信号量

我们现在提供一个使用内存映射I/O以及Posix互斥锁和条件变量完成的Posix有名信号量的

实现。[IEEE 1996]的B.11.3节（基本原理部分）中也提供了一个类似的实现。

> 我们将在第12章和第13章中讨论内存映射I/O。你可能希望跳过本节，到阅读完那两章后再返回来。

我们首先在图10-42中给出自己的semaphore.h头文件，它定义了基本的sem_t数据类型。

my_pxsem_mmap/semaphore.h

```
1         /* the fundamental datatype */
2 typedef struct {
3     pthread_mutex_t sem_mutex;          /* lock to test and set semaphore value */
4     pthread_cond_t  sem_cond;           /* for transition from 0 to nonzero */
5     unsigned int sem_count;             /* the actual semaphore value */
6     int     sem_magic;                  /* magic number if open */
7 } sem_t;

8 #define SEM_MAGIC   0x67458923

9 #ifdef  SEM_FAILED
10 #undef  SEM_FAILED
11 #define SEM_FAILED  ((sem_t *)(-1))    /* avoid compiler warnings */
12 #endif
```

my_pxsem_mmap/semaphore.h

图10-42 semaphore.h头文件

sem_t数据类型

1~7 我们的信号量数据结构含有一个互斥锁、一个条件变量和包含本信号量当前值的一个无符号整数。正如随图10-36讨论的那样，一旦该结构已被初始化，其sem_magic成员将含有SEM_MAGIC值。

10.15.1 sem_open函数

图10-43给出了我们的sem_open函数的前半部分，它会创建一个新的信号量或打开一个已存在的信号量。

my_pxsem_mmap/sem_open.c

```
1 #include    "unpipc.h"
2 #include    "semaphore.h"

3 #include    <stdarg.h>              /* for variable arg lists */
4 #define     MAX_TRIES   10          /* for waiting for initialization */

5 sem_t *
6 sem_open(const char *pathname, int oflag, ... )
7 {
8     int     fd, i, created, save_errno;
9     mode_t  mode;
10    va_list ap;
11    sem_t  *sem, seminit;
12    struct stat statbuff;
13    unsigned int value;
14    pthread_mutexattr_t mattr;
15    pthread_condattr_t  cattr;

16    created = 0;
17    sem = MAP_FAILED;               /* [sic] */
18 again:
```

图10-43 sem_open函数：前半部分

```
19      if (oflag & O_CREAT) {
20          va_start(ap, oflag);        /* init ap to final named argument */
21          mode = va_arg(ap, va_mode_t) & ~S_IXUSR;
22          value = va_arg(ap, unsigned int);
23          va_end(ap);

24              /* open and specify O_EXCL and user-execute */
25          fd = open(pathname, oflag | O_EXCL | O_RDWR, mode | S_IXUSR);
26          if (fd < 0) {
27              if (errno == EEXIST && (oflag & O_EXCL) == 0)
28                  goto exists;        /* already exists, OK */
29              else
30                  return(SEM_FAILED);
31          }
32          created = 1;
33              /* first one to create the file initializes it */
34              /* set the file size */
35          bzero(&seminit, sizeof(seminit));
36          if (write(fd, &seminit, sizeof(seminit)) != sizeof(seminit))
37              goto err;

38              /* memory map the file */
39          sem = mmap(NULL, sizeof(sem_t), PROT_READ | PROT_WRITE,
40                     MAP_SHARED, fd, 0);
41          if (sem == MAP_FAILED)
42              goto err;

43              /* initialize mutex, condition variable, and value */
44          if ( (i = pthread_mutexattr_init(&mattr)) != 0)
45              goto pthreaderr;
46          pthread_mutexattr_setpshared(&mattr, PTHREAD_PROCESS_SHARED);
47          i = pthread_mutex_init(&sem->sem_mutex, &mattr);
48          pthread_mutexattr_destroy(&mattr);  /* be sure to destroy */
49          if (i != 0)
50              goto pthreaderr;

51          if ( (i = pthread_condattr_init(&cattr)) != 0)
52              goto pthreaderr;
53          pthread_condattr_setpshared(&cattr, PTHREAD_PROCESS_SHARED);
54          i = pthread_cond_init(&sem->sem_cond, &cattr);
55          pthread_condattr_destroy(&cattr);   /* be sure to destroy */
56          if (i != 0)
57              goto pthreaderr;

58          if ( (sem->sem_count = value) > sysconf(_SC_SEM_VALUE_MAX)) {
59              errno = EINVAL;
60              goto err;
61          }
62              /* initialization complete, turn off user-execute bit */
63          if (fchmod(fd, mode) == -1)
64              goto err;
65          close(fd);
66          sem->sem_magic = SEM_MAGIC;
67          return(sem);
68      }
```

my_pxsem_mmap/sem_open.c

图10-43（续）

处理可变参数表

14~23 如果调用者指定了O_CREAT标志，我们就知道需要的是四个而不是两个参数，我们已随图5-21讲述过可变参数表和我们的va_mode_t数据类型。接着关掉mode变量中的用户执行位（S_IXUSR），其原因稍后讲述。

创建一个新的信号量，并处理潜在的竞争状态

24~32 创建一个新文件，其名字为调用者命名信号量所用的名字，同时打开它的用户执行位。在调用者指定了O_CREAT标志的前提下，如果我们只是打开该文件，内存映射其内容，然后初始化sem_t结构的三个成员，那就会碰到一个竞争状态。我们已随图5-21描述过该竞争状态，这儿所用的处理技巧跟那儿给出的一样。在图10-52中我们还会碰到一个类似的竞争状态。

设置文件大小

33~37 通过往其中写入一个用0填充的结构来设置新创建文件的大小。既然知道刚创建的文件的大小为0，于是我们调用write而不是ftruncate来设置文件大小，因为正如我们在13.3节中所注，当一个普通文件的大小有待增长时，Posix并不保证ftruncate能够工作。

内存映射文件

38~42 调用mmap对新创建的文件进行内存映射。该文件将含有所创建信号量sem_t数据结构的当前值，不过既然已内存映射了该文件，我们就只通过由mmap返回的指针来访问它，而从不调用read或write。

初始化sem_t数据结构

3~57 初始化内存映射文件中的sem_t数据结构的三个成员：互斥锁、条件变量、信号量的值。既然Posix有名信号量可由知道其名字并有足够权限的任意进程共享，因此在初始化互斥锁和条件变量成员时，我们必须指定PTHREAD_PROCESS_SHARED属性。对于互斥锁成员的做法是：首先调用pthread_mutexattr_init初始化它的属性，然后调用pthread_mutexattr_setpshared在该属性结构中设置进程间共享属性，最后调用pthread_mutex_init初始化该互斥锁。对于条件变量成员也有几乎相同的做法。我们小心地保证在出错情况下摧毁这些属性对象。

初始化信号量值

58~61 最后存入信号量的初始值，我们还把该值与通过调用sysconf（10.13节）获取的最大允许值作比较。

关掉用户执行位

62~67 完成信号量的初始化后，关掉用户执行位。它指示信号量已初始化完毕的状态。接着close已内存映射了的文件，因为已没有保持它继续打开着的必要了。

图10-44给出了我们的sem_open函数的后半部分。在这里处理前面提到过的竞争状态的技巧与图5-23中所用的技巧相同。

打开已存在的信号量

69~78 我们是在调用者没有指定O_CREAT标志或者尽管指定了但所需信号量已存在的条件下到达这儿的。无论哪种情况下，我们都要打开一个已存在的信号量。我们open含有所需sem_t数据类型的文件用于读和写，并把该文件内存映射到当前进程的地址空间中（使用mmap）。

262 ~ 265

my_pxsem_mmap/sem_open.c

```
69  exists:
70    if ( (fd = open(pathname, O_RDWR)) < 0) {
71        if (errno == ENOENT && (oflag & O_CREAT))
72            goto again;
73        goto err;
74    }
75    sem = mmap(NULL, sizeof(sem_t), PROT_READ | PROT_WRITE,
76               MAP_SHARED, fd, 0);
77    if (sem == MAP_FAILED)
78        goto err;

79        /* make certain initialization is complete */
80    for (i = 0; i < MAX_TRIES; i++) {
81        if (stat(pathname, &statbuff) == -1) {
82            if (errno == ENOENT && (oflag & O_CREAT)) {
83                close(fd);
84                goto again;
85            }
86            goto err;
87        }
88        if ((statbuff.st_mode & S_IXUSR) == 0) {
89            close(fd);
90            sem->sem_magic = SEM_MAGIC;
91            return(sem);
92        }
93        sleep(1);
94    }
95    errno = ETIMEDOUT;
96    goto err;

97  pthreaderr:
98    errno = i;
99  err:
100       /* don't let munmap() or close() change errno */
101   save_errno = errno;
102   if (created)
103       unlink(pathname);
104   if (sem != MAP_FAILED)
105       munmap(sem, sizeof(sem_t));
106   close(fd);
107   errno = save_errno;
108   return(SEM_FAILED);
109 }
```

my_pxsem_mmap/sem_open.c

图10-44　sem_open函数：后半部分

现在可以看出为什么Posix.1声称："访问信号量的副本将产生未定义的结果"。当使用内存映射I/O来实现有名信号量时，信号量（即sem_t数据类型）会内存映射到让它打开着的所有进程的地址空间中。这是由打开该有名信号量的每个进程调用sem_open来完成的。任意一个进程对该信号量所做的变动（例如改变它的计数值）由所有其他进程通过内存映射当场看到。要是我们去制造某个sem_t数据结构的私用副本，那么该副本将不再为所有进程所共享。即使我们可能认为它在工作（针对它的信号量函数也许不会给出错误，至少在调用sem_close之前有这种可能，而sem_close是要撤销映射的，因此关闭私用副本肯定失败），本进程与其他进程间也不会发生同步。然而从图1-6可注意到，父进程中的内存映射区穿越fork后继续在子进程中存留，因此由内核完成的从父进程到子进程穿越fork进行的信号量复制是可行的。

确保信号量已初始化

79~96 我们必须等待刚才打开的信号量完成初始化（以防多个线程几乎同时尝试创建同一个信号量）。为此，我们调用stat查看内存映射文件的权限（stat结构的st_mode成员）。如果它的用户执行位已关掉，那么该信号量已完成初始化。

出错返回

97~108 当出错时，我们小心地保证不改变errno的值。

10.15.2 sem_close函数

图10-45给出了我们的sem_close函数，它只是对先前映射的内存区调用munmap。要是调用者继续使用由以前的sem_open调用返回并作为参数传递给本函数的指针，那么它将收到一个SIGSEGV信号。

my_pxsem_mmap/sem_close.c

```
1 #include    "unpipc.h"
2 #include    "semaphore.h"

3 int
4 sem_close(sem_t *sem)
5 {
6     if (sem->sem_magic != SEM_MAGIC) {
7         errno = EINVAL;
8         return(-1);
9     }
10    if (munmap(sem, sizeof(sem_t)) == -1)
11        return(-1);

12    return(0);
13 }
```

my_pxsem_mmap/sem_close.c

图10-45 sem_close函数

10.15.3 sem_unlink函数

图10-46给出了我们的sem_unlink函数，由它删除与某个信号量关联的名字。它仅仅调用Unix的unlink函数。

my_pxsem_mmap/sem_unlink.c

```
1 #include    "unpipc.h"
2 #include    "semaphore.h"

3 int
4 sem_unlink(const char *pathname)
5 {
6     if (unlink(pathname) == -1)
7         return(-1);
8     return(0);
9 }
```

my_pxsem_mmap/sem_unlink.c

图10-46 sem_unlink函数

10.15.4 sem_post函数

图10-47给出了我们的sem_post函数，它会给某个信号量的值加1，如果此前该信号量的值为0，那就唤醒因等待该信号量而阻塞的任意线程。

my_pxsem_mmap/sem_post.c

```
 1 #include    "unpipc.h"
 2 #include    "semaphore.h"

 3 int
 4 sem_post(sem_t *sem)
 5 {
 6     int     n;

 7     if (sem->sem_magic != SEM_MAGIC) {
 8         errno = EINVAL;
 9         return(-1);
10     }
11     if ( (n = pthread_mutex_lock(&sem->sem_mutex)) != 0) {
12         errno = n;
13         return(-1);
14     }
15     if (sem->sem_count == 0)
16         pthread_cond_signal(&sem->sem_cond);
17     sem->sem_count++;
18     pthread_mutex_unlock(&sem->sem_mutex);
19     return(0);
20 }
```

my_pxsem_mmap/sem_post.c

图10-47　sem_post函数

11~18　在修改由调用者指定的信号量的值之前，首先得获取它的互斥锁。如果该信号量的值将从0增为1，那就调用pthread_cond_signal唤醒等待它的任意一个线程。

267 ~ 268

10.15.5　sem_wait函数

图10-48给出了我们的sem_wait函数，它等待某个信号量的值超过0。

my_pxsem_mmap/sem_waitt.c

```
 1 #include    "unpipc.h"
 2 #include    "semaphore.h"

 3 int
 4 sem_wait(sem_t *sem)
 5 {
 6     int     n;

 7     if (sem->sem_magic != SEM_MAGIC) {
 8         errno = EINVAL;
 9         return(-1);
10     }
11     if ( (n = pthread_mutex_lock(&sem->sem_mutex)) != 0) {
12         errno = n;
13         return(-1);
14     }
15     while (sem->sem_count == 0)
16         pthread_cond_wait(&sem->sem_cond, &sem->sem_mutex);
17     sem->sem_count--;
18     pthread_mutex_unlock(&sem->sem_mutex);
19     return(0);
20 }
```

my_pxsem_mmap/sem_wait.c

图10-48　sem_wait函数

11~18　在修改由调用者指定的信号量的值之前，首先得获取它的互斥锁。如果该信号量的值

为0，那就让调用线程睡眠在一个pthread_cond_wait调用中，等待另外某个线程为该信号量的条件变量调用pthread_cond_signal，到那时该信号量的值将由那个线程从0增为1。该值一旦大于0，本函数就将它减1，然后释放关联的互斥锁。

10.15.6 sem_trywait函数

图10-49给出了我们的sem_trywait函数，它是sem_wait函数的非阻塞版本。

my_pxsem_mmap/sem_trywait.c

```
 1 #include    "unpipc.h"
 2 #include    "semaphore.h"

 3 int
 4 sem_trywait(sem_t *sem)
 5 {
 6     int     n, rc;

 7     if (sem->sem_magic != SEM_MAGIC) {
 8         errno = EINVAL;
 9         return(-1);
10     }
11     if ( (n = pthread_mutex_lock(&sem->sem_mutex)) != 0) {
12         errno = n;
13         return(-1);
14     }
15     if (sem->sem_count > 0) {
16         sem->sem_count--;
17         rc = 0;
18     } else {
19         rc = -1;
20         errno = EAGAIN;
21     }
22     pthread_mutex_unlock(&sem->sem_mutex);
23     return(rc);
24 }
```

my_pxsem_mmap/sem_trywait.c

图10-49 sem_trywait函数

11~22 获取由调用者指定的信号量的互斥锁后检查它的值。如果该值大于0，那就将它减1并返回0。否则返回值为-1，同时置errno为EAGAIN。

10.15.7 sem_getvalue函数

图10-50给出了我们的最后一个函数sem_getvalue，它会返回某个信号量的当前值。

my_pxsem_mmap/sem_getvalue.c

```
 1 #include    "unpipc.h"
 2 #include    "semaphore.h"

 3 int
 4 sem_getvalue(sem_t *sem, int *pvalue)
 5 {
 6     int     n;

 7     if (sem->sem_magic != SEM_MAGIC) {
 8         errno = EINVAL;
 9         return(-1);
```

图10-50 sem_getvalue函数

```
10      }
11      if ( (n = pthread_mutex_lock(&sem->sem_mutex)) != 0) {
12          errno = n;
13          return(-1);
14      }
15      *pvalue = sem->sem_count;
16      pthread_mutex_unlock(&sem->sem_mutex);
17      return(0);
18  }
```

my_pxsem_mmap/sem_getvalue.c

269 ~ 270

图10-50（续）

11~16　获取由调用者指定的信号量的互斥锁后返回它的值。

从本节提供的实现可以看出，使用信号量比使用互斥锁和条件变量要简单。

10.16　使用System V信号量实现Posix信号量

我们现在提供使用System V信号量完成的Posix有名信号量的又一个实现。既然较早的System V信号量实现与较新的Posix信号量实现相比更为普遍，那么本实现允许应用程序在操作系统还不支持Posix信号量的情况下就开始使用它们。

> 我们将在第11章中讨论System V信号量。你可能希望暂时跳过本节，到阅读完那一章之后再返回来。

我们首先在图10-51中给出semaphore.h头文件，它定义了基本的sem_t数据类型。

my_pxsem_svsem/semaphore.h

```
1         /* the fundamental datatype */
2  typedef struct {
3      int      sem_semid;        /* the System V semaphore ID */
4      int      sem_magic;        /* magic number if open */
5  } sem_t;

6  #define SEM_MAGIC    0x45678923

7  #ifdef  SEM_FAILED
8  #undef  SEM_FAILED
9  #define SEM_FAILED   ((sem_t *)(-1))    /* avoid compiler warnings */
10 #endif

11 #ifndef SEMVMX
12 #define SEMVMX  32767             /* historical System V max value for sem */
13 #endif
```

my_pxsem_svsem/semaphore.h

图10-51　semaphore.h头文件

sem_t数据类型

1~5　我们使用仅由一个成员构成的System V信号量集来实现Posix有名信号量。这个信号量数据结构包含对应的System V信号量ID和一个魔数（我们已随图10-36讨论过魔数的用途）。

10.16.1　sem_open函数

图10-52给出了我们的sem_open函数的前半部分，该函数创建一个新的信号量或打开一个

已存在的信号量。

my_pxsem_svsem/sem_open.c

```
 1 #include    "unpipc.h"
 2 #include    "semaphore.h"

 3 #include    <stdarg.h>        /* for variable arg lists */
 4 #define     MAX_TRIES   10    /* for waiting for initialization */

 5 sem_t *
 6 sem_open(const char *pathname, int oflag, ... )                    271
 7 {
 8     int     i, fd, semflag, semid, save_errno;
 9     key_t   key;
10     mode_t  mode;
11     va_list ap;
12     sem_t   *sem;
13     union semun arg;
14     unsigned int value;
15     struct semid_ds seminfo;
16     struct sembuf   initop;

17         /* no mode for sem_open() w/out O_CREAT; guess */
18     semflag = SVSEM_MODE;
19     semid = -1;

20     if (oflag & O_CREAT) {
21         va_start(ap, oflag);       /* init ap to final named argument */
22         mode = va_arg(ap, va_mode_t);
23         value = va_arg(ap, unsigned int);
24         va_end(ap);

25             /* convert to key that will identify System V semaphore */
26         if ( (fd = open(pathname, oflag, mode)) == -1)
27             return(SEM_FAILED);
28         close(fd);
29         if ( (key = ftok(pathname, 0)) == (key_t) -1)
30             return(SEM_FAILED);

31         semflag = IPC_CREAT | (mode & 0777);
32         if (oflag & O_EXCL)
33             semflag |= IPC_EXCL;

34             /* create the System V semaphore with IPC_EXCL */
35         if ( (semid = semget(key, 1, semflag | IPC_EXCL)) >= 0) {
36                 /* success, we're the first so initialize to 0 */
37             arg.val = 0;
38             if (semctl(semid, 0, SETVAL, arg) == -1)
39                 goto err;
40                 /* then increment by value to set sem_otime nonzero */
41             if (value > SEMVMX) {
42                 errno = EINVAL;
43                 goto err;
44             }
45             initop.sem_num = 0;
46             initop.sem_op  = value;
47             initop.sem_flg = 0;
48             if (semop(semid, &initop, 1) == -1)
49                 goto err;
50             goto finish;

51         } else if (errno != EEXIST || (semflag & IPC_EXCL) != 0)
52             goto err;
53         /* else fall through */
54     }
```

my_pxsem_svsem/sem_open.c 272

图10-52 sem_open函数：前半部分

创建一个新的信号量，处理可变长度参数表

20~24　如果调用者指定了O_CREAT标志，我们就知道需要的是四个而不是两个参数。我们已随图5-21描述过可变长度参数表的处理和我们的va_mode_t数据类型。

创建辅助文件，将其路径名映射成System V IPC键

25~30　创建一个普通文件，其路径名为调用者命名Posix信号量所用的名字。创建该文件的目的只是为了有一个路径名可供ftok用来标识该信号量。调用者给该信号量指定的*oflag*参数可以是O_CREAT或O_CREAT | O_EXCL，它用在打开辅助文件的open调用中。这样，如果该文件尚未存在，那就创建它；如果该文件已存在而且调用者指定了O_EXCL标志，那就返回一个错误。接着关闭该文件的描述符，因为该文件的唯一用途是作为ftok的参数，由ftok将其路径名转换成一个System V IPC键（3.2节）。

创建仅有一个成员的System V信号量集

31~33　把O_CREAT和O_EXCL这两个常值转换成对应的System V IPC_*xxx*常值后，调用semget创建一个仅由单个成员构成的System V信号量集。我们总是指定IPC_EXCL标志，目的是为了确定该System V信号量是否存在。

初始化信号量

34~50　11.2节将叙述初始化System V信号量时存在的一个基本问题，11.6节将给出避免其中潜在的竞争状态的代码。这儿使用类似的技巧。尝试创建那个System V信号量的第一个线程（回想一下，我们调用semget时总是指定IPC_EXCL）将使用semctl的SETVAL命令将它初始化为0，然后调用semop把它设置成由调用者指定的初始值。该信号量的sem_otime值保证由semget初始化为0，然后由创建者的semop调用设置成某个非零值。这样一来，发现该信号量已经存在的任何其他线程一旦看到该信号量的sem_otime值不为0，就可以肯定它已完成初始化。

检查初始值

40~44　我们检查由调用者指定的初始值，因为System V信号量通常作为unsigned short整数存放（见11.1节中的sem结构），有一个32767的最大值（11.7节），Posix信号量则通常作为普通整数存放，允许的值可能更大（10.13节）。有些System V实现把常值SEMVMX定义成最大的信号量值，如果系统没有定义该常值，我们就在图10-51中把它定义为32767。

51~53　如果所需的System V信号量已经存在，而且调用者没有指定O_EXCL，那就不算出错。这种情形下，代码将落入用于打开（而不是创建）已存在信号量的那部分。

273　图10-53给出了我们的sem_open函数的后半部分。

打开已存在的信号量

55~63　对于已存在的待打开Posix信号量（调用者未指定O_CREAT标志或者尽管指定了该标志，但所需信号量已存在），我们使用semget打开与之对应的System V信号量。注意，当O_CREAT标志未指定时，sem_open并没有*mode*参数，然而即使待打开的是一个已存在的信号量，semget也需要一个与*mode*参数等价的参数。在本函数的开始处，我们给变量semflag赋了个默认值（来自我们的unpipc.h头文件的SVSEM_MODE常值），它在调用者未指定O_CREAT标志时传递给semget。

等待信号量完成初始化

64~72　接着通过以IPC_STAT命令循环调用semctl，等待该信号量的sem_otime值变为非零来验证它已初始化完毕。

my_pxsem_svsem/sem_open.c

```
55      /*
56       * (O_CREAT not secified) or
57       * (O_CREAT without O_EXCL and semaphore already exists).
58       * Must open semaphore and make certain it has been initialized.
59       */
60      if ( (key = ftok(pathname, 0)) == (key_t) -1)
61          goto err;
62      if ( (semid = semget(key, 0, semflag)) == -1)
63          goto err;

64      arg.buf = &seminfo;
65      for (i = 0; i < MAX_TRIES; i++) {
66          if (semctl(semid, 0, IPC_STAT, arg) == -1)
67              goto err;
68          if (arg.buf->sem_otime != 0)
69              goto finish;
70          sleep(1);
71      }
72      errno = ETIMEDOUT;
73  err:
74      save_errno = errno;         /* don't let semctl() change errno */
75      if (semid != -1)
76          semctl(semid, 0, IPC_RMID);
77      errno = save_errno;
78      return(SEM_FAILED);

79  finish:
80      if ( (sem = malloc(sizeof(sem_t))) == NULL)
81          goto err;

82      sem->sem_semid = semid;
83      sem->sem_magic = SEM_MAGIC;
84      return(sem);
85  }
```

my_pxsem_svsem/sem_open.c

图10-53 sem_open函数：后半部分

出错返回

274

73~78 发生错误时，我们小心地保证不改变errno的值。

分配sem_t数据类型

79~84 为一个sem_t数据类型分配空间，把所创建或打开的System V信号量的ID存放到该结构中。本函数的返回值就是指向该sem_t数据类型的指针。

10.16.2 sem_close 函数

图10-54给出了我们的sem_close函数，它仅仅调用free释放早先为sem_t数据类型动态分配的内存空间。

my_pxsem_svsem/sem_close.c

```
1 #include      "unpipc.h"
2 #include      "semaphore.h"

3 int
4 sem_close(sem_t *sem)
5 {
6       if (sem->sem_magic != SEM_MAGIC) {
```

图10-54 sem_close函数

```
 7          errno = EINVAL;
 8          return(-1);
 9      }
10      sem->sem_magic = 0;          /* just in case */

11      free(sem);
12      return(0);
13  }
```

my_pxsem_svsem/sem_close.c

图10-54（续）

10.16.3　sem_unlink函数

图10-55给出了我们的sem_unlink函数，它删除与我们的某个Posix信号量关联的辅助文件和System V信号量。

my_pxsem_svsem/sem_unlink.c

```
 1 #include    "unpipc.h"
 2 #include    "semaphore.h"

 3 int
 4 sem_unlink(const char *pathname)
 5 {
 6      int     semid;
 7      key_t   key;

 8      if ( (key = ftok(pathname, 1)) == (key_t) -1)
 9          return(-1);
10      if (unlink(pathname) == -1)
11          return(-1);
12      if ( (semid = semget(key, 1, SVSEM_MODE)) == -1)
13          return(-1);
14      if (semctl(semid, 0, IPC_RMID) == -1)
15          return(-1);
16      return(0);
17  }
```

my_pxsem_svsem/sem_unlink.c

图10-55　sem_unlink函数

获取与路径名关联的System V键

8~16　ftok把调用者指定的路径名转换成一个System V IPC键。unlink函数删除与该路径名同名的辅助文件。（我们现在就删除该文件，这样可避免另外某个函数返回一个错误。[①]）semget打开关联的System V信号量，接着由semctl的IPC_RMID命令删除该信号量。

10.16.4　sem_post函数

图10-56给出了我们的sem_post函数，它会给某个信号量的值加1。

11~16　以单一操作调用semop，把由调用者指定的信号量的值加1。

① 在调用线程执行本函数期间，系统可能切换针对同一信号量调用sem_open函数的另一个线程来运行。要是把本函数改为先删除关联的System V信号量，再删除辅助文件，那么一旦线程切换发生在这两个删除操作之间，执行sem_open的线程将发现所需的Posix信号量已存在（因为其辅助文件尚未删除），但是与之关联的System V信号量却打不开（因为已被删除了），于是出错。——译者注

my_pxsem_svsem/sem_post.c

```
 1 #include    "unpipc.h"
 2 #include    "semaphore.h"

 3 int
 4 sem_post(sem_t *sem)
 5 {
 6     struct sembuf op;

 7     if (sem->sem_magic != SEM_MAGIC) {
 8         errno = EINVAL;
 9         return(-1);
10     }
11     op.sem_num = 0;
12     op.sem_op = 1;
13     op.sem_flg = 0;
14     if (semop(sem->sem_semid, &op, 1) < 0)
15         return(-1);
16     return(0);
17 }
```

my_pxsem_svsem/sem_post.c

图10-56 sem_post函数

10.16.5 sem_wait 函数

图10-57给出了我们的sem_wait函数，它等待某个信号量的值超过0。

my_pxsem_svsem/sem_wait.c

```
 1 #include    "unpipc.h"
 2 #include    "semaphore.h"

 3 int
 4 sem_wait(sem_t *sem)
 5 {
 6     struct sembuf op;

 7     if (sem->sem_magic != SEM_MAGIC) {
 8         errno = EINVAL;
 9         return(-1);
10     }
11     op.sem_num = 0;
12     op.sem_op = -1;
13     op.sem_flg = 0;
14     if (semop(sem->sem_semid, &op, 1) < 0)
15         return(-1);
16     return(0);
17 }
```

my_pxsem_svsem/sem_wait.c

图10-57 sem_wait函数

11~16 以单一操作调用semop，把由调用者指定的信号量的值减1。

10.16.6 sem_trywait 函数

图10-58给出了我们的sem_trywait函数，它是sem_wait函数的非阻塞版本。

13 与图10-57中sem_wait函数的唯一差别是把sem_ftg成员指定为IPC_NOWAIT。如果不阻塞调用线程就完成不了所指定的操作，那么semop的返回值将是EAGAIN错误，这也是sem_trywait非得阻塞才能完成等待操作时必须返回的错误。

my_pxsem_svsem/sem_trywait.c

```
 1 #include    "unpipc.h"
 2 #include    "semaphore.h"

 3 int
 4 sem_trywait(sem_t *sem)
 5 {
 6     struct sembuf   op;

 7     if (sem->sem_magic != SEM_MAGIC) {
 8         errno = EINVAL;
 9         return(-1);
10     }
11     op.sem_num = 0;
12     op.sem_op = -1;
13     op.sem_flg = IPC_NOWAIT;
14     if (semop(sem->sem_semid, &op, 1) < 0)
15         return(-1);
16     return(0);
17 }
```

my_pxsem_svsem/sem_trywait.c

图10-58　sem_trywait函数

10.16.7　sem_getvalue函数

图10-59给出了我们的最后一个函数sem_getvalue，它返回某个信号量的当前值。

my_pxsem_svsem/sem_getvalue.c

```
 1 #include    "unpipc.h"
 2 #include    "semaphore.h"

 3 int
 4 sem_getvalue(sem_t *sem, int *pvalue)
 5 {
 6     int     val;

 7     if (sem->sem_magic != SEM_MAGIC) {
 8         errno = EINVAL;
 9         return(-1);
10     }
11     if ( (val = semctl(sem->sem_semid, 0, GETVAL)) < 0)
12         return(-1);
13     *pvalue = val;
14     return(0);
15 }
```

my_pxsem_svsem/sem_getvalue.c

图10-59　sem_getvalue函数

11~14　由调用者指定的信号量的当前值使用semctl的GETVAL命令获取。

10.17　小结

Posix信号量是计数信号量，它提供以下三种基本操作：

(1) 创建一个信号量；

(2) 等待一个信号量的值变为大于0，然后将它的值减1；

(3) 给一个信号量的值加1，并唤醒等待该信号量的任意线程，以此挂出该信号量。

Posix信号量可以是有名的，也可以是基于内存的。有名信号量总是能够在不同进程间共享，

基于内存的信号量则必须在创建时指定成是否在进程间共享。这两类信号量的持续性也有差别：有名信号量至少有随内核的持续性，基于内存的信号量则具有随进程的持续性。

生产者-消费者问题是演示信号量的经典例子。本章中，解决这个问题的第一个方案只有一个生产者线程和一个消费者线程，下一个方案允许多个生产者线程和单个消费者线程，最后一个方案则允许多个消费者线程。我们接着指出，双缓冲这个经典问题仅仅是生产者-消费者问题的一个特例，该问题涉及的生产者和消费者都是单个的。

本章最后提供了Posix信号量的三种示例实现。使用FIFO完成的第一种实现是最简单的，因为内核提供的read和write函数处理了不少同步需求。第二种实现使用内存映射I/O完成，这与5.8节中提供的Posix消费队列的实现类似，其中的同步使用互斥锁和条件变量进行。最后一种实现使用System V信号量完成，它同时提供了访问System V信号量的一个更简单的接口。

275
2
278

习题

10.1 如下修改10.6节中的produce和consume函数。首先，对换消费者程序中两个Sem_wait的顺序以引发死锁（正如10.6节中讨论的那样）。其次，在每个Sem_wait调用之前加一个printf调用，以指出哪个线程（生产者或消费者）在等待哪个信号量。在这些Sem_wait调用之后再加一个printf调用，以指出取得了相应信号量的线程。把缓冲区的数目减少到2，然后构造并运行该程序，以验证它会导致死锁。

10.2 假设启动图9-2中的程序的4个副本，而该程序调用的my_lock函数来自图10-19：

```
% lockpxsem & lockpxsem & lockpxsem & lockpxsem &
```

这4个进程都以值为0的initflag启动，因此每个进程调用sem_open时都指定了O_CREAT标志。这样可行吗？

10.3 上一道习题中，要是有一个进程在调用my_lock之后但在调用my_unlock之前终止，那么会发生什么？

10.4 在图10-37中，要是我们没有把那两个描述符都初始化为-1，那么会发生什么？

10.5 在图10-37中，我们为什么先保存errno的值，稍后再恢复，而不是把那两个close调用改为：

```
if (sem->fd[0] >= 0)
    close(sem->fd[0]);
if (sem->fd[1] >= 0)
    close(sem->fd[1]);
```

10.6 如果有两个进程几乎同时调用我们的sem_open的FIFO实现版本（图10-37），而且都指定O_CREAT标志和初始值5，那么会发生什么？相应的FIFO有可能被（不正确地）初始化为10吗？

10.7 对于图10-43和图10-44，我们描述过当有两个进程几乎同时尝试创建一个信号量时可能存在的一种竞争状态。然而在上一道习题的解答中，我们说图10-37不存在竞争状态。请解释。

10.8 Posix.1规定了sem_wait的一个可选功能：检测自己已被一个捕获的信号中断并返回EINTR错误。编写一个测试程序，判定你的系统上的实现是否进行这种检测。

再针对我们的几种实现运行你的测试程序，它们有使用FIFO的（10.14节）、使用内存映射I/O的（10.15节）和使用System V信号量的（10.16节）。

10.9 我们的3种sem_post实现中，哪种是异步信号安全的（图5-10）？

10.10 修改10.6节中使用的生产者-消费者解决方案，将mutex变量改为使用pthread_mutex_t数据类型，而不是使用信号量。在性能上发生了可测量到的变化了吗？

10.11 比较有名信号量（图10-17和图10-18）和基于内存的信号量（图10-20）的定时结果。

279

第11章 System V信号量

11.1 概述

我们在第10章中讲述信号量的概念时，首先讨论的是

- 二值信号量（binary semaphore）：其值或为0或为1的信号量。这与互斥锁（第7章）类似，若资源被锁住则信号量值为0，若资源可用则信号量值为1。

接着把这种信号量扩展为

- 计数信号量（counting semaphore）：其值在0和某个限制值（对于Posix信号量，该值必须至少为32767）之间的信号量。我们使用这些信号量在生产者-消费者问题中统计资源，信号量的值就是可用资源数。

这两种类型的信号量中，等待（wait）操作都等待信号量的值变为大于0，然后将它减1。挂出（post）操作则只是将信号量的值加1，从而唤醒正在等待该信号量值变为大于0的任意线程。

System V信号量通过定义如下概念给信号量增加了另外一级复杂度。

- 计数信号量集（set of counting semaphores）：一个或多个信号量（构成一个集合），其中每个都是计数信号量。每个集合的信号量数存在一个限制，一般在25个的数量级上（11.7节）。当我们谈论“System V信号量”时，所指的是计数信号量集。当我们谈论“Posix信号量”时，所指的是单个计数信号量。

对于系统中的每个信号量集，内核维护一个如下的信息结构，它定义在<sys/sem.h>头文件中。

```
struct semid_ds {
  struct ipc_perm   sem_perm;    /* operation permission struct */
  struct sem       *sem_base;    /* ptr to array of semaphores in set */
  ushort            sem_nsems;   /* # of semaphores in set */
  time_t            sem_otime;   /* time of last semop() */
  time_t            sem_ctime;   /* time of creation or last IPC_SET */
};
```

其中的ipc_perm结构已在3.3节描述过，它含有当前这个特定信号量的访问权限。

sem结构是内核用于维护某个给定信号量的一组值的内部数据结构。一个信号量集的每个成员由如下这个结构描述：

```
struct sem {
  ushort_t semval;     /* semaphore value, nonnegative */
  short    sempid;     /* PID of last successful semop(), SETVAL, SETALL */
  ushort_t semncnt;    /* # awaiting semval > current value */
  ushort_t semzcnt;    /* # awaiting semval = 0 */
};
```

注意sem_base含有指向某个sem结构数组的指针：当前信号量集中的每个信号量对应其中一个数组元素。

除维护一个信号量集内每个信号量的实际值之外，内核还给该集合中每个信号量维护另外三个信息：对其值执行最后一次操作的进程的进程ID、等待其值增长的进程数计数以及等待其值变为0的进程数计数。

Unix 98声称上述这个结构是匿名的。我们给出的名字sem来自历史上的System V实现。

我们可以把内核中的某个特定信号量图解成指向一个sem结构数组的一个semid_ds结构。如果该信号量在其集合中有两个成员，那么我们将有图11-1所示的图解。该图中变量sem_nsems的值为2，另外该集合的两个成员分别用下标[0]和[1]标记。

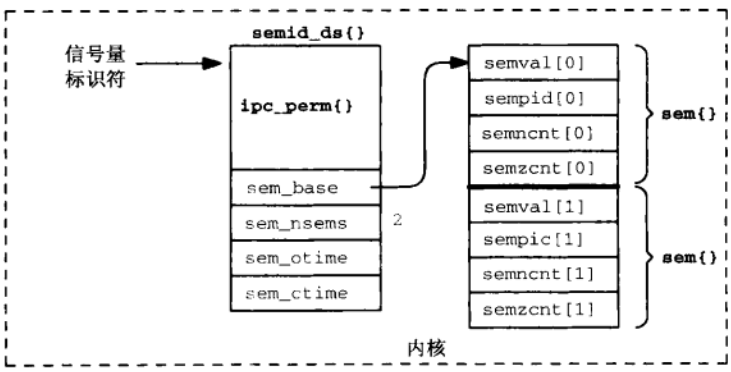


图11-1 由两组值构成的某个信号量集的内核数据结构

11.2 semget 函数

semget函数创建一个信号量集或访问一个已存在的信号量集。

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

int semget(key_t key, int nsems, int oflag);
```

返回：若成功则为非负标识符，若出错则为-1

返回值是一个称为信号量标识符（semaphore identifier）的整数，semop和semctl函数将使用它。

*nsems*参数指定集合中的信号量数。如果我们不创建一个新的信号量集，而只是访问一个已存在的集合，那就可以把该参数指定为0。一旦创建完一个信号量集，我们就不能改变其中的信号量数。

*oflag*值是图3-6中给出的SEM_R和SEM_A常值的组合。其中R代表“读（read）”，A代表“改（alter）”。它们还可以与IPC_CREAT或IPC_CREAT | IPC_EXCL按位或，如随图3-4所作的讨论。

当实际操作为创建一个新的信号量集时，相应的semid_ds结构的以下成员将被初始化。

- sem_perm结构的uid和cuid成员被置为调用进程的有效用户ID，gid和cgid成员被置为调用进程的有效组ID。
- oflag参数中的读写权限位存入sem_perm.mode。
- sem_otime被置为0，sem_ctime则被置为当前时间。

- sem_nsems被置为*nsems*参数的值。
- 与该集合中每个信号量关联的各个sem结构并不初始化。这些结构是在以SET_VAL或SETALL命令调用semctl时初始化的。

信号量值的初始化

282 ~ 283 出现在本书1990年版所含源代码中的注释不正确地声称，当实际操作为创建一个新的信号量集时，semget会将该集合中各个信号量的值初始化为0。尽管有些系统确实把新的信号量集内各个信号量的值初始化为0，这一点却不能保证做到。实际上早期的System V实现根本不对信号量值进行初始化，存放新创建信号量集的那部分内存空间最近一次使用时的值就是各个信号量的初始值。

semget的大多数手册页面根本不就实际操作为创建一个新的信号量集时各个信号量的初始值应为多少说些什么。X/Open XPG3可移植性指南（1989年）和Unix 98纠正了这个忽略行为，明确地陈述semget并不初始化各个信号量的值，这个初始化必须通过以SET_VAL命令（设置集合中一个值）或SETALL命令（设置集合中所有值）调用semctl（我们稍后描述）来完成。

System V信号量的设计中，创建一个信号量集（semget）并将它初始化（semctl）需两次函数调用是一个致命的缺陷。一个不完备的解决方案是：在调用semget时指定IPC_CREAT | IPC_EXCL标志，这样只有一个进程（首先调用semget的那个进程）创建所需信号量，该进程随后初始化该信号量，其他进程会收到来自semget的一个EEXIST错误，于是再次调用semget，不过这次调用既不指定IPC_CREAT标志，也不指定IPC_EXCL标志。

然而竞争状态依然存在。假设有两个进程几乎同时尝试创建并初始化一个只有单个成员的信号量集，两者都执行如下几行标了号的代码：

```
1    oflag = IPC_CREAT | IPC_EXCL | SVSEM_MODE;
2    if ( (semid = semget(key, 1, oflag)) >= 0) {
             /* success, we are the first, so initialize */
3        arg.val = 1;
4        Semctl(semid, 0, SETVAL, arg);

5    } else if (errno == EEXIST) {
             /* already exists, just open */
6        semid = Semget(key, 1, SVSEM_MODE);

7    } else
8        err_sys("semget error");

9    Semop(semid, ...);      /* decrement the semaphore by 1 */
```

那么可能发生如下情形：

(1) 第一个进程执行第1~3行，然后被内核阻止执行；

(2) 内核启动第二个进程，它执行第1、2、5、6、9行。

尽管成功创建该信号量的第一个进程将是初始化该信号量的唯一进程，但是由于它完成创建和初始化操作需花两个步骤，因此内核有可能在这两个步骤之间把上下文切换到另一个进程。这个新切换来运行的进程随后可以使用该信号量（上述代码片段的第9行），但是该信号量的值尚未由第一个进程初始化。当第二个进程执行第9行时，该信号量的值是不确定的。

幸运的是存在绕过这个竞争状态的方法。当semget创建一个新的信号量集时，其semid_ds结构的sem_otime成员保证被置为0。（System V手册已陈述这个事实很长时间，XPG3和Unix 98 284 标准也这么说。）该成员只是在semop调用成功时才被设置为当前值。因此，上面例子中的第二

个进程再次成功地调用semget（上述代码片段的第6行）后，必须以IPC_STAT命令调用semctl。它然后等待sem_otime变为非零值，到时就可断定该信号量已被初始化，而且对它进行初始化的那个进程已成功地调用semop。这意味着创建该信号量的那个进程必须初始化它的值，而且必须在任何其他进程可以使用该信号量之前调用semop。我们在图10-52和图11-7中展示了使用这种技巧的例子。

Posix有名信号量通过让单个函数（sem_open）创建并初始化信号量来避免上述问题。而且即使指定O_CREAT标志，信号量也只是在尚未存在的前提下才被初始化。

这个潜在的竞争状态是否构成问题还取决于应用程序。有些应用程序（例如图10-21中的生产者-消费者程序）由单个进程创建并初始化信号量。这种情形下不会存在竞争状态。但是在其他应用程序（例如图10-19中的文件上锁例子程序）中，创建并初始化信号量的并不是单个进程：第一个打开信号量的进程必须创建并初始化它，而且必须避免竞争状态。

11.3 semop 函数

使用semget打开一个信号量集后，对其中一个或多个信号量的操作就使用semop函数来执行。

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

int semop(int semid, struct sembuf *opsptr, size_t nops);
```

返回：若成功则为0，若出错则为-1

其中*opsptr*指向一个如下结构的数组：

```
struct sembuf {
  short   sem_num;      /* semaphore number: 0, 1, ..., nsems-1 */
  short   sem_op;       /* semaphore operation: <0, 0, >0 */
  short   sem_flg;      /* operation flags: 0, IPC_NOWAIT, SEM_UNDO */
};
```

*nops*参数指出由*opsptr*指向的sembuf结构数组中元素的数目。该数组中的每个元素给目标信号量集内某个特定的信号量指定一个操作。这个特定的信号量由sem_num指定，0代表第一个元素，1代表第二个元素，依次类推，直到nsems-1，其中nsems是目标信号量集内成员信号量的数目（也就是创建该集合时传递给semget的第二个参数）。

我们仅仅保证sembuf结构含有所给出的三个成员。它还可能含有其他成员，而且各成员并不保证以我们给出的顺序排序。这意味着我们绝不能静态地初始化这种结构，例如：

```
struct sembuf   ops[2] = {
  0, 0, 0,              /* wait for [0] to be 0 */
  0, 1, SEM_UNDO        /* then increment [0] by 1 */
};
```

而是必须使用运行时初始化方法，例如：

```
struct sembuf   ops[2];
ops[0].sem_num = 0;         /* wait for [0] to be 0 */
ops[0].sem_op = 0;
ops[0].sem_flg = 0;
ops[1].sem_num = 0;         /* then increment [0] by 1 */
ops[1].sem_op = 1;
ops[1].sem_flg = SEM_UNDO;
```

由内核保证传递给semop函数的操作数组（opsptr）被原子地执行。内核或者完成所有指定的操作，或者什么操作都不做。我们将在11.5节中给出这个特性的一个例子。

每个特定的操作是由sem_op的值确定的，它可以是负数、0或正数。在稍后给出的讨论中，我们将使用如下术语。

- semval：信号量的当前值（图11-1）。
- semncnt：等待semval变为大于其当前值的线程数（图11-1）。
- semzcnt：等待semval变为0的线程数（图11-1）。
- semadj：所指定信号量针对调用进程的调整值。只有在对应本操作的sembuf结构的sem_flg成员中指定SEM_UNDO标志后，semadj才会更新。这是一个概念性的变量，它由内核为在其某个信号量操作中指定了SEM_UNDO标志的各个进程维护，不必存在名为semadj的结构成员。[①]
- 使得一个给定信号量操作非阻塞的方法是，在对应的sembuf结构的sem_flg成员中指定IPC_NOWAIT标志。在指定了该标志，并且如果不把调用线程投入睡眠就完成不了这个给定操作的情况下，semop将返回一个EAGAIN错误。
- 当一个线程被投入睡眠以等待某个信号量操作完成之时（我们将看到该线程既可等待这个信号量的值变为0，也可等待它变为大于0），如果它捕获了一个信号，那么其信号处理程序的返回将中断引起睡眠的semop函数，该函数于是返回一个EINTR错误。按照UNPv1第124页的术语定义，semop是需被所捕获的信号中断的慢系统调用（slow system call）。
- 当一个线程被投入睡眠以等待某个信号量操作完成之时，如果该信号量被另外某个线程或进程从系统中删除，那么引起睡眠的semop函数将返回一个EIDRM错误，表示“identifier removed（标识符已删除）”。

现在我们基于每个具体指定的sem_op操作的三类可能值——正数、0或负数——来描述semop的操作。

286 (1) 如果sem_op是正数，其值就加到semval上。这对应于释放由某个信号量控制的资源。如果指定了SEM_UNDO标志，那就从相应信号量的semadj值中减掉sem_op的值。

(2) 如果sem_op是0，那么调用者希望等待到semval变为0。如果semval已经是0，那就立即返回。

如果semval不为0，相应信号量的semzcnt值就加1，调用线程则被阻塞到semval变为0（到那时，相应信号量的semzcnt值再减1）。前面已经提到，如果指定了IPC_NOWAIT标志，调用线程就不会被投入睡眠。如果某个被捕获的信号中断了引起睡眠的semop函数，或者相应的信号量被删除了，那么该函数将过早地返回一个错误。

(3) 如果sem_op是负数，那么调用者希望等待semval变为大于或等于sem_op的绝对值。这对应于分配资源。

如果semval大于或等于sem_op的绝对值，那就从semval中减掉sem_op的绝对值。如果指定了SEM_UNDO标志，那么sem_op的绝对值就加到相应信号量的semadj值上。

如果semval小于sem_op的绝对值，相应信号量的semncnt值就加1，调用线程则被阻塞到

① 调用进程终止时，semadj加到相应信号量的semval之上。要是调用进程对某个信号量的全部操作都指定SEM_UNDO标志，那么该进程终止后，该信号量的值就会变得像根本没有运行过该进程一样，这就是复旧（undo）的本意。——译者注

semval变为大于或等于sem_op的绝对值。到那时该线程将被解阻塞，还将从semval中减掉sem_op的绝对值，相应信号量的semncnt值将减1。如果指定了SEM_UNDO标志，那么sem_op的绝对值将加到相应信号量的semadj值上。前面已经提到，如果指定了IPC_NOWAIT标志，调用线程就不会被投入睡眠。另外，如果某个被捕获的信号中断了引起睡眠的sem_op函数，或者相应的信号量被删除了，那么该函数将过早地返回一个错误。

比较一下这些操作和Posix信号量允许的操作，可看到Posix信号量只允许-1（sem_wait）和+1（sem_post）这两个操作。System V信号量允许信号量的值增长或减少不光是1，而且允许等待信号量的值变为0。与较为简单的Posix信号量相比，这些更为一般化的操作以及System V信号量可以有一组值的事实造成了System V信号量的复杂性。

11.4 semctl 函数

semctl函数对一个信号量执行各种控制操作。

```
#include <sys/sem.h>
int semctl(int semid, int semnum, int cmd, ... /* union semun arg */ );
                                        返回：若成功则为非负值（见正文），若出错则为-1
```

287

第一个参数*semid*标识其操作待控制的信号量集，第二个参数*semnum*标识该信号量集内的某个成员（0、1等等，直到*nsems*-1）。*semnum*值仅仅用于GETVAL、SETVAL、GETNCNT、GETZCNT和GETPID命令。

第四个参数是可选的，取决于第三个参数*cmd*（参见下面给出的联合中的注释）。

```
union semun {
  int              val;    /* used for SETVAL only */
  struct semid_ds  *buf;   /* used for IPC_SET and IPC_STAT */
  ushort           *array; /* used for GETALL and SETALL */
};
```

这个联合并没有出现在任何系统头文件中，因而必须由应用程序声明。（我们在图C.1中给出的unpipc.h头文件中定义了它。）它是按值传递的，而不是按引用传递的。也就是说作为参数的是这个联合的真正值，而不是指向它的指针。

不幸的是，有些系统（FreeBSD和Linux）在<sys/sem.h>头文件中定义了这个联合，从而使编写可移植代码变得困难。尽管由这个系统头文件来声明semun联合确实有理由，Unix 98还是声称它必须由应用程序显式声明。

System V支持下列*cmd*值。除非另外声明，否则返回值为0表示成功，返回值为-1表示出错。

GETVAL　把semval的当前值作为函数返回值返回。既然信号量决不会是负数（semval被声明成一个unsigned short整数），那么成功的返回值总是非负数。

SETVAL　把semval值设置为*arg.val*。如果操作成功，那么相应信号量在所有进程中的信号量调整值（semadj）将被置为0。

GETPID　把sempid的当前值作为函数返回值返回。

GETNCNT　把semncnt的当前值作为函数返回值返回。

GETZCNT　把semzcnt的当前值作为函数返回值返回。

GETALL　返回所指定信号量集内每个成员的semval值。这些值通过*arg.array*指针返回，函数本身的返回值则为0。注意，调用者必须分配一个unsigned short整数数组，该数组要足够容纳所指定信号量集内所有成员的semval值的，然后把*arg.array*设置成指向这个数组。

SETALL　设置所指定信号量集中每个成员的semval值。这些值是通过*arg.array*指针指定的。

IPC_RMID　把由*semid*指定的信号量集从系统中删除掉。

IPC_SET　设置所指定信号量集的semid_ds结构中的以下三个成员：sem_perm.uid、sem_perm.gid和sem_perm.mode，这些值来自由*arg.buf*参数指向的结构中的相应成员。semid_ds结构中的sem_ctime成员也被设置成当前时间。

288

IPC_STAT　（通过*arg.buf*参数）返回所指定信号量集当前的semid_ds结构。注意，调用者必须首先分配一个semid_ds结构，并把*arg.buf*设置成指向这个结构。

11.5　简单的程序

既然System V信号量具有随内核的持续性，于是我们可以通过编写一组操纵它们的程序并查看这些程序的运行结果来展示它们的用法。信号量的值将由内核从我们的某个程序维持到下一个程序。

11.5.1　semcreate程序

图11-2给出了我们的第一个程序，它只是创建一个System V信号量集。-e命令行选项指定IPC_EXCL标志，该集合中信号量的数目必须由最后一个命令行参数指定。

svsem/semcreate.c

```
 1 #include    "unpipc.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     c, oflag, semid, nsems;

 6     oflag = SVSEM_MODE | IPC_CREAT;
 7     while ( (c = Getopt(argc, argv, "e")) != -1) {
 8         switch (c) {
 9         case 'e':
10             oflag |= IPC_EXCL;
11             break;
12         }
13     }
14     if (optind != argc - 2)
15         err_quit("usage: semcreate [ -e ] <pathname> <nsems>");
16     nsems = atoi(argv[optind + 1]);

17     semid = Semget(Ftok(argv[optind], 0), nsems, oflag);
18     exit(0);
19 }
```

svsem/semcreate.c

图11-2　semcreate程序

11.5.2 semrmid 程序

图11-3给出的下一个程序会从系统中删除一个信号量集。删除该集合通过调用semctl函数执行IPC_RMID命令完成。

svsem/semrmid.c

```
1 #include    "unpipc.h"
2 int
3 main(int argc, char **argv)
4 {
5     int     semid;
6     if (argc != 2)
7         err_quit("usage: semrmid <pathname>");
8     semid = Semget(Ftok(argv[1], 0), 0, 0);
9     Semctl(semid, 0, IPC_RMID);
10    exit(0);
11 }
```

svsem/semrmid.c

图11-3 semrmid函数

11.5.3 semsetvalues 程序

图11-4给出的semsetvalues程序设置某个信号量集中的所有值。

svsem/semsetvalues.c

```
1 #include    "unpipc.h"
2 int
3 main(int argc, char **argv)
4 {
5     int     semid, nsems, i;
6     struct semid_ds seminfo;
7     unsigned short  *ptr;
8     union semun arg;
9     if (argc < 2)
10        err_quit("usage: semsetvalues <pathname> [ values ... ]");
11        /* first get the number of semaphores in the set */
12    semid = Semget(Ftok(argv[1], 0), 0, 0);
13    arg.buf = &seminfo;
14    Semctl(semid, 0, IPC_STAT, arg);
15    nsems = arg.buf->sem_nsems;
16        /* now get the values from the command line */
17    if (argc != nsems + 2)
18        err_quit("%d semaphores in set, %d values specified", nsems, argc-2);
19        /* allocate memory to hold all the values in the set, and store */
20    ptr = Calloc(nsems, sizeof(unsigned short));
21    arg.array = ptr;
22    for (i = 0; i < nsems; i++)
23        ptr[i] = atoi(argv[i + 2]);
24    Semctl(semid, 0, SETALL, arg);
25    exit(0);
26 }
```

svsem/semsetvalues.c

图11-4 semsetvalues函数

取得集合中信号量的数目

11~15　使用semget获取所指定信号量集的信号量ID之后，发出一个semctl的IPC_STAT命令取得该信号量的semid_ds结构。其中sem_nsems成员就是该集合中信号量的数目。

设置所有的值

19~24　分配一个unsigned short整数数组的内存空间，每个集合成员对应一个数组元素，然后把它们的值从命令行复制到数组中。接着由一个semctl的SETALL命令设置该信号量集内所有成员信号量的值。

11.5.4　semgetvalues 程序

图11-5给出的semgetvalues程序取得并输出某个信号量集中的所有值。

svsem/semgetvalues.c

```
 1 #include    "unpipc.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     semid, nsems, i;
 6     struct semid_ds seminfo;
 7     unsigned short  *ptr;
 8     union semun arg;

 9     if (argc != 2)
10         err_quit("usage: semgetvalues <pathname>");

11         /* first get the number of semaphores in the set */
12     semid = Semget(Ftok(argv[1], 0), 0, 0);
13     arg.buf = &seminfo;
14     Semctl(semid, 0, IPC_STAT, arg);
15     nsems = arg.buf->sem_nsems;

16         /* allocate memory to hold all the values in the set */
17     ptr = Calloc(nsems, sizeof(unsigned short));
18     arg.array = ptr;

19         /* fetch the values and print */
20     Semctl(semid, 0, GETALL, arg);
21     for (i = 0; i < nsems; i++)
22         printf("semval[%d] = %d\n", i, ptr[i]);

23     exit(0);
24 }
```

svsem/semgetvalues.c

图11-5　semgetvalues程序

取得集合中信号量的数目

11~15　使用semget获取所指定信号量集的信号量ID之后，发出一个semctl的IPC_STAT命令取得该信号量的semid_ds结构。其中sem_nsems成员就是该集合中信号量的数目。

取得所有的值

16~22　分配一个unsigned short整数数组的内存空间，每个集合成员对应一个数组元素，然后发出一个semctl的GETALL命令获取该信号量集内所有成员信号量的值。接着输出所有的值。

11.5.5 semops 程序

图11-6给出的semops程序对某个信号量集执行一数组的操作。

svsem/semops.c

```
1 #include    "unpipc.h"
2 int
3 main(int argc, char **argv)
4 {
5     int     c, i, flag, semid, nops;
6     struct sembuf   *ptr;
7     flag = 0;
8     while ( (c = Getopt(argc, argv, "nu")) != -1) {
9         switch (c) {
10        case 'n':
11            flag |= IPC_NOWAIT;         /* for each operation */
12            break;
13        case 'u':
14            flag |= SEM_UNDO;           /* for each operation */
15            break;
16        }
17    }
18    if (argc - optind < 2)              /* argc - optind = #args remaining */
19        err_quit("usage: semops [ -n ] [ -u ] <pathname> operation ...");
20    semid = Semget(Ftok(argv[optind], 0), 0, 0);
21    optind++;
22    nops = argc - optind;
23        /* allocate memory to hold operations, store, and perform */
24    ptr = Calloc(nops, sizeof(struct sembuf));
25    for (i = 0; i < nops; i++) {
26        ptr[i].sem_num = i;
27        ptr[i].sem_op = atoi(argv[optind + i]);      /* <0, 0, or >0 */
28        ptr[i].sem_flg = flag;
29    }
30    Semop(semid, ptr, nops);
31    exit(0);
32 }
```

svsem/semops.c

图11-6 semops程序

命令行选项

7~19 -n选项给每个操作指定IPC_NOWAIT标志，-u选项给每个操作指定SEM_UNDO标志。注意，semop函数允许我们给sembuf结构的每个成员（也就是针对信号量集内每个成员的操作）指定一组不同的标志，但是为了简单起见，我们让这些命令行选项给所有成员信号量各自的指定操作统一指定相应标志。

给各个操作分配内存空间

20~29 使用semget打开所指定的信号量集后，分配一个sembuf结构数组，命令行中指定的每个操作对应一个数组元素。与前两个程序不同，本程序允许用户指定少于相应信号量集内成员数目的操作个数。

执行各个操作

30 semop在所指定信号量集上执行刚才创建的操作数组。

11.5.6 例子

现在演示刚刚给出的5个程序，以查看System V信号量的某些特性。

```
solaris % touch /tmp/rich
solaris % semcreate -e /tmp/rich 3
solaris % semsetvalues /tmp/rich 1 2 3
solaris % semgetvalues /tmp/rich
semval[0] = 1
semval[1] = 2
semval[2] = 3
```

我们首先创建一个名为/tmp/rich的文件，它将（通过ftok）用于标识本例子所用的信号量集。semcreate创建一个共有三个成员的信号量集。semsetvalues把它们的值分别设置为1、2和3，这些值随后由semgetvalues输出。

我们接着演示在一个信号量集上执行一组操作的原子性。

```
solaris % semops -n /tmp/rich -1 -2 -4
semctl error: Resource temporarily unavailable
solaris % semgetvalues /tmp/rich
semval[0] = 1
semval[1] = 2
semval[2] = 3
```

我们指定了非阻塞标志（-n）和三个操作，每个操作分别减少刚创建信号量集内的某个值。第一个操作可以执行（我们可以从该集合值为1的第一个成员中减掉1），第二个操作也可以执行（我们可以从该集合值为2的第二个成员中减掉2），但是第三个操作却无法执行（我们不能从该集合值为3的第三个成员中减掉4）。既然最后一个操作不能执行，而且指定了非阻塞标志，那么将返回一个EAGAIN错误。（要是未曾指定非阻塞标志，我们的程序就只是阻塞。）我们接着验证该集合中没有值变动过。尽管前两个操作可以执行，但是由于最后一个操作不能执行，因此这三个操作都不执行。semop的原子性意味着要么所有操作都执行，要么一个操作都不执行。

290 ~ 293

下面演示System V信号量的SEM_UNDO属性。

```
solaris % semsetvalues /tmp/rich 1 2 3          设置成已知值
solaris % semops -u /tmp/rich -1 -2 -3          给每个操作指定SEM_UNDO标志
solaris % semgetvalues /tmp/rich
semval[0] = 1                                   当semops终止时所有变动都被取消
semval[1] = 2
semval[2] = 3
solaris % semops /tmp/rich -1 -2 -3             不指定SEM_UNDO标志
solaris % semgetvalues /tmp/rich
semval[0] = 0                                   变动未被取消
semval[1] = 0
semval[2] = 0
```

我们首先使用semsetvalues把三个信号量值重新置为1、2和3，然后使用semops指定-1、-2和-3三个操作。这导致所有三个值都变为0，但是既然我们给semops程序指定了-u选项，那么所有三个操作都被指定了SEM_UNDO标志。这么一来，这三个成员信号量的semadj值就分别被置为1、2和3。后来当semops程序终止时，这三个semadj值就分别加到三个成员信号量的当前值（全为0）上，导致它们的最终值分别变为1、2和3，这一点我们用semgetvalues程序验证了。我们接着再次执行semops程序，不过这次不指定-u选项，其结果是当semops程序终止时，所有三个成员信号量的值都保持为0，而不回复到开始执行semops程序时的值。

11.6 文件上锁

我们可以提供图10-19中my_lock和my_unlock函数的另一个版本，它们使用System V信号量实现。图11-7给出了这个版本。

lock/locksvsem.c

```
1 #include    "unpipc.h"
2 #define LOCK_PATH   "/tmp/svsemlock"
3 #define MAX_TRIES   10
4 int     semid, initflag;
5 struct sembuf   postop, waitop;
6 void
7 my_lock(int fd)
8 {
9     int     oflag, i;
10    union semun arg;
11    struct semid_ds seminfo;
12    if (initflag == 0) {
13        oflag = IPC_CREAT | IPC_EXCL | SVSEM_MODE;
14        if ( (semid = semget(Ftok(LOCK_PATH, 0), 1, oflag)) >= 0) {
15                /* success, we're the first so initialize */
16            arg.val = 1;
17            Semctl(semid, 0, SETVAL, arg);
18        } else if (errno == EEXIST) {
19                /* someone else has created; make sure it's initialized */
20            semid = Semget(Ftok(LOCK_PATH, 0), 1, SVSEM_MODE);
21            arg.buf = &seminfo;
22            for (i = 0; i < MAX_TRIES; i++) {
23                Semctl(semid, 0, IPC_STAT, arg);
24                if (arg.buf->sem_otime != 0)
25                    goto init;
26                sleep(1);
27            }
28            err_quit("semget OK, but semaphore not initialized");
29        } else
30            err_sys("semget error");
31  init:
32        initflag = 1;
33        postop.sem_num = 0;       /* and init the two semop() structures */
34        postop.sem_op  = 1;
35        postop.sem_flg = SEM_UNDO;
36        waitop.sem_num = 0;
37        waitop.sem_op  = -1;
38        waitop.sem_flg = SEM_UNDO;
39    }
40    Semop(semid, &waitop, 1);     /* down by 1 */
41 }
42 void
43 my_unlock(int fd)
44 {
45    Semop(semid, &postop, 1);     /* up by 1 */
46 }
```

lock/locksvsem.c

图11-7 使用System V信号量实现的文件上锁

首先尝试独占创建

13~17 我们必须保证只有单个进程初始化文件上锁信号量，因此给semget指定了IPC_CREAT

IPC_EXCL标志。如果创建成功，当前进程就调用semctl将该信号量的值初始化为1。如果有多个进程几乎同时调用我们的my_lock函数，那么只有一个进程会创建出文件上锁信号量（假设它尚未存在），接着初始化该信号量的也是这个进程。

信号量已存在，那就打开它

18~20　对于其他进程来说，第一个semget调用将返回一个EEXIST错误，它们于是再次调用semget，不过这次不指定IPC_CREAT|IPC_EXCL标志。

等待信号量被初始化

21~28　我们遇到了11.2节中讲解System V信号量的初始化时讨论过的同一竞争状态。为避免该竞争状态，发现文件上锁信号量已存在的任何进程都必须以IPC_STAT命令调用semctl，以查看该信号量的sem_otime值。如果该值不为0，我们就知道创建该信号量的进程已对它初始化，并已调用semop（semop调用在本函数末尾）。如果该信号量的sem_otime值仍为0（这种情况应该非常罕见），我们就sleep 1秒再尝试。我们限制了尝试次数，避免发生永久睡眠。

初始化sembuf结构

33~38　我们早先提及，sembuf结构中各成员的排列顺序没有保证，因此不能静态地初始化它们。当一个进程首次调用my_lock时，我们分配两个这样的结构，并在运行时填写它们。我们指定了SEM_UNDO标志，这样的话如果某个进程在持有锁期间终止了，内核会释放该锁（见习题10.3）。

在首次使用时创建一个信号量很容易（每个进程尝试创建它，但是如果它已经存在，那就忽略所产生的错误），然而在所有进程都完成后将它删除要困难得多。在使用序列号文件分配作业号的打印机守护进程例子中，信号量将一直存在下去。但是其他应用程序可能希望一旦需要上锁的文件被删除，其信号量也被删除。对于这种情况，使用记录锁也许比使用信号量更好。

11.7　信号量限制

跟System V消息队列一样，System V信号量也有特定的系统限制，其中大部分源自最初的System V实现（3.8节）。图11-8展示了这些限制。第一栏是含有相应限制值的内核变量的传统System V名字。

名　字	说　明	DUnix 4.0B	Solaris 2.6
semmni	系统范围最大信号量集数	16	10
semmsl	每个信号量集最大信号量数	25	25
semmns	系统范围最大信号量数	400	60
semopm	每个semop调用最大操作数	10	10
semmnu	系统范围最大复旧结构数①		30
semume	每个复旧结构最大复旧项数	10	10
semvmx	任何信号量的最大值	32767	32767
semaem	最大退出时调整（adjust-on-exit）值	16384	16384

图11-8　System V信号量的典型限制值

①每个复旧（undo）结构对应一个进程。——译者注