# THE LINUX PROGRAMMING INTERFACE

A Linux and UNIX® System Programming Handbook

MICHAEL KERRISK

no starch press

# THE LINUX PROGRAMMING INTERFACE

**A Linux and UNIX System Programming Handbook**

**MICHAEL KERRISK**

no starch press

San Francisco

*Translated by: Kevin*

# 目录

前言.....17
主题.....17
目标读者.....17
Linux 和 UNIX.....18
使用和组织.....18
例子程序.....19
习题.....20
标准和可移植性.....20
Linux 内核和 C 库版本.....21
其它语言使用编程接口.....21
关于作者.....21
致谢.....21
许可.....22
网站和例子程序源代码.....22
反馈.....22
第 1 章 历史和标准.....23
1.1 UNIX 和 C 简史.....23
1.2 Linux 简史.....26
1.2.1 GNU 项目.....27
1.2.2 Linux 内核.....28
1.3 标准化.....33
1.3.1 C 编程语言.....33
1.3.2 第一个 POSIX 标准.....34
1.3.3 X/Open 公司和开放组织.....35
1.3.4 SUSv3 和 POSIX.1-2001.....36
1.3.5 SUSv4 和 POSIX.1-2008.....38
1.3.6 UNIX 标准时间线.....39

1.3.7 实现标准 .....40
1.3.8 Linux、标准、和 Linux 标准基础 .....41
1.4 小结 .....42
第 2 章 基础概念 .....44
2.1 操作系统的核心：内核 .....44
2.2 Shell .....47
2.3 用户和组 .....48
2.4 单一目录层次、目录、链接、和文件 .....49
2.5 文件 I/O 模型 .....53
2.6 程序 .....54
2.7 进程 .....54
2.8 内存映射 .....58
2.9 静态和共享库 .....59
2.10 进程间通信和同步 .....60
2.11 信号 .....61
2.12 线程 .....62
2.13 进程组和 shell 工作控制 .....62
2.14 会话、控制终端、和控制进程 .....63
2.15 伪终端 .....64
2.16 日期和时间 .....64
2.17 客户端-服务器体系架构 .....65
2.18 实时 .....66
2.19 /proc 文件系统 .....66
2.20 小结 .....67
第 3 章 系统编程概念 .....68
3.1 系统调用 .....68
3.2 库函数 .....71
3.3 标准 C 库；GNU C 库（glibc） .....72

3.4 系统调用和库函数的错误处理 ....73
3.5 本书示例程序的说明 ....76
3.5.1 命令行选项和参数 ....76
3.5.2 常用函数和头文件 ....76
3.6 可移植问题 ....87
3.6.1 特性测试宏 ....87
3.6.2 系统数据类型 ....90
3.6.3 各种可移植问题 ....93
3.7 小结 ....95
3.8 习题 ....96
第 4 章 文件 I/O：通用 I/O 模型 ....97
4.1 概述 ....97
4.2 I/O 的通用性 ....100
4.3 打开文件：open() ....100
4.3.1 open()的 flags 参数 ....103
4.3.2 open()的错误 ....107
4.3.3 creat()系统调用 ....108
4.4 读取文件：read() ....109
4.5 写入文件：write() ....110
4.6 关闭文件：close() ....111
4.7 改变文件偏移：lseek() ....111
4.8 通用 I/O 模型之外的操作：ioctl() ....117
4.9 小结 ....118
4.10 习题 ....118
第 5 章 文件 I/O：更多细节 ....119
5.1 原子性和竞争条件 ....119
5.2 文件控制操作：fcntl() ....123
5.3 打开文件状态标志 ....123

5.4 文件描述符和打开文件之间的关系 ..... 125
5.5 复制文件描述符 ..... 127
5.6 指定偏移位置的文件 I/O：pread()和 pwrite() ..... 130
5.7 Scatter-Gather I/O：readv()和 writev() ..... 131
5.8 截断文件：truncate()和 ftruncate() ..... 135
5.9 非阻塞 I/O ..... 136
5.10 大文件 I/O ..... 136
5.11 /dev/fd 目录 ..... 140
5.12 创建临时文件 ..... 141
5.13 小结 ..... 143
5.14 习题 ..... 144
第 6 章 进程 ..... 146
6.1 进程和程序 ..... 146
6.2 进程 ID 和父进程 ID ..... 147
6.3 进程内存布局 ..... 148
6.4 虚拟内存管理 ..... 149
6.5 堆栈和栈帧 ..... 149
6.6 命令行参数（argc, argv） ..... 149
6.7 环境列表 ..... 149
6.8 执行非局部跳转：setjmp()和 longjmp() ..... 149
6.9 小结 ..... 149
6.10 习题 ..... 149
第 7 章 内存分配 ..... 150
7.1 在堆上分配内存 ..... 150
7.1.1 调整 Program Break：brk()和 sbrk() ..... 150
7.1.2 在堆上分配内存：malloc()和 free() ..... 150
7.1.3 实现 malloc()和 free() ..... 150
7.1.4 在堆上分配内存的其它方法 ..... 150

7.2 在栈上分配内存：alloca()....150
7.3 小结....150
7.4 习题....150
第 8 章 用户和组....151
8.1 密码文件：/etc/passwd....151
8.2 阴影密码文件：/etc/shadow....151
8.3 组文件：/etc/group....151
8.4 获取用户和组信息....151
8.5 密码加密和用户认证....151
8.6 小结....151
8.7 习题....151
第 9 章 进程凭证....152
9.1 真实用户 ID 和真实组 ID....152
9.2 有效用户 ID 和有效组 ID....152
9.3 设置用户 ID 和设置组 ID....152
9.4 保存的设置用户 ID 和保存的设置组 ID....152
9.5 文件系统用户 ID 和文件系统组 ID....152
9.6 补充组 ID....152
9.7 获取和修改进程凭证....152
9.7.1 获取和修改真实、有效、和保存的设置 ID....153
9.7.2 获取和修改文件系统 ID....153
9.7.3 获取和修改补充组 ID....153
9.7.4 修改进程凭证调用小结....153
9.7.5 示例：显示进程凭证....153
9.8 小结....153
9.9 习题....153
第 10 章 时间....154
10.1 日历时间....154

10.2 时间转换函数 ..... 154
10.2.1 time_t 转换为可打印格式 ..... 154
10.2.2 time_t 和 Broken-Down 时间互相转换 ..... 154
10.2.3 Broken-Down 时间和可打印格式互相转换 ..... 154
10.3 时区 ..... 154
10.4 Locale ..... 154
10.5 更新系统时钟 ..... 154
10.6 软时钟（Jiffy） ..... 154
10.7 进程时间 ..... 154
10.8 小结 ..... 154
10.9 习题 ..... 154
第 11 章 系统限制和选项 ..... 155
11.1 系统限制 ..... 155
11.2 运行时获取系统限制（和选项） ..... 155
11.3 运行时获取文件相关的限制（和选项） ..... 155
11.4 不确定限制 ..... 155
11.5 系统选项 ..... 155
11.6 小结 ..... 155
11.7 习题 ..... 155
第 12 章 系统和进程信息 ..... 156
12.1 /proc 文件系统 ..... 156
12.1.1 获取进程的信息：/proc/PID ..... 156
12.1.2 /proc 下的系统信息 ..... 156
12.1.3 访问/proc 文件 ..... 156
12.2 系统标识：uname() ..... 156
12.3 小结 ..... 156
12.4 习题 ..... 156
第 13 章 文件 I/O 缓冲 ..... 157

13.1 文件 I/O 的内核缓冲：缓冲区缓存 ..... 157
13.2 stdio 库的缓冲 ..... 157
13.3 控制文件 I/O 的内核缓冲 ..... 157
13.4 I/O 缓冲小结 ..... 157
13.5 通知内核 I/O 模型 ..... 157
13.6 绕过缓冲区缓存：Direct I/O ..... 157
13.7 混合库函数和系统调用的文件 I/O ..... 157
13.8 小结 ..... 157
13.9 习题 ..... 157
第 14 章 文件系统 ..... 158
14.1 设备特殊文件（设备） ..... 158
14.2 磁盘和分区 ..... 158
14.3 文件系统 ..... 158
14.4 i-node ..... 158
14.5 虚拟文件系统（VFS） ..... 158
14.6 日志文件系统 ..... 158
14.7 单目录层次结构和挂载点 ..... 158
14.8 挂载和卸载文件系统 ..... 158
14.8.1 挂载文件系统：mount() ..... 158
14.8.2 卸载文件系统：umount()和 umount2() ..... 158
14.9 高级挂载特性 ..... 158
14.9.1 挂载文件系统至多个挂载点 ..... 158
14.9.2 堆叠多个挂载至相同挂载点 ..... 158
14.9.3 单个挂载选项的挂载标志 ..... 158
14.9.4 绑定挂载 ..... 159
14.9.5 递归绑定挂载 ..... 159
14.10 虚拟内存文件系统：tmpfs ..... 159
14.11 获取文件系统信息：statvfs() ..... 159

14.12 小结 ..... 159
14.13 习题 ..... 159
第 15 章 文件属性 ..... 160
15.1 获取文件信息：stat() ..... 160
15.2 文件时间戳 ..... 160
15.2.1 改变文件时间戳：utime()和 utimes() ..... 160
15.2.2 改变文件时间戳：utimensat()和 futimens() ..... 160
15.3 文件所属权 ..... 160
15.3.1 新文件的所属权 ..... 160
15.3.2 改变文件所属权：chown(), fchown(), lchown() ..... 160
15.4 文件权限 ..... 160
15.4.1 普通文件权限 ..... 160
15.4.2 目录权限 ..... 160
15.4.3 权限检查算法 ..... 160
15.4.4 检查文件可访问性：access() ..... 160
15.4.5 设置用户 ID，设置组 ID，粘滞位 ..... 160
15.4.6 进程文件模式创建掩码：umask() ..... 160
15.4.7 改变文件权限：chmod()和 fchmod() ..... 161
15.5 i-node 标志（ext2 扩展文件属性） ..... 161
15.6 小结 ..... 161
15.7 习题 ..... 161
第 16 章 扩展属性 ..... 162
16.1 概述 ..... 162
16.2 扩展属性实现细节 ..... 162
16.3 操作扩展属性的系统调用 ..... 162
16.4 小结 ..... 162
16.5 习题 ..... 162
第 17 章 访问控制列表 ..... 163

17.1 概述 ..... 163
17.2 ACL 权限检查算法 ..... 163
17.3 ACL 的长文本和短文本格式 ..... 163
17.4 ACL_MASK 入口和 ACL 组类 ..... 163
17.5 getfacl 和 setfacl 命令 ..... 163
17.6 默认 ACL 和文件创建 ..... 163
17.7 ACL 实现限制 ..... 163
17.8 ACL API ..... 163
17.9 小结 ..... 163
17.10 习题 ..... 163
第 18 章 目录和链接 ..... 164
18.1 目录和（硬）链接 ..... 164
18.2 符号（软）链接 ..... 164
18.3 创建和删除（硬）链接：link()和 unlink() ..... 164
18.4 文件重命名：rename() ..... 164
18.5 操作符号链接：symlink()和 readlink() ..... 164
18.6 创建和删除目录：mkdir()和 rmdir() ..... 164
18.7 删除文件或目录：remove() ..... 164
18.8 读取目录：opendir()和 readdir() ..... 164
18.9 遍历文件树：nftw() ..... 164
18.10 进程的当前工作目录 ..... 164
18.11 相对目录文件描述符操作 ..... 164
18.12 改变进程的根目录：chroot() ..... 164
18.13 解引用路径名：realpath() ..... 164
18.14 解析路径名字符串：dirname()和 basename() ..... 164
18.15 小结 ..... 165
18.16 习题 ..... 165
第 19 章 监控文件事件 ..... 166

19.1 概述 ..... 166
19.2 inotify API ..... 166
19.3 inotify 事件 ..... 166
19.4 读取 inotify 事件 ..... 166
19.5 队列限制和/proc 文件 ..... 166
19.6 监控文件事件的旧系统：dnotify ..... 166
19.7 小结 ..... 166
19.8 习题 ..... 166
第 20 章 信号：基础概念 ..... 167
20.1 概念和概述 ..... 167
20.2 信号类型和默认动作 ..... 167
20.3 改变信号配置：signal() ..... 167
20.4 信号处理器介绍 ..... 167
20.5 发送信号：kill() ..... 167
20.6 检查进程是否存在 ..... 167
20.7 发送信号的其它方法：raise()和 killpg() ..... 167
20.8 显示信号描述信息 ..... 167
20.9 信号集 ..... 167
20.10 信号掩码（阻塞信号递送） ..... 167
20.11 未决信号 ..... 167
20.12 信号没有排队 ..... 167
20.13 改变信号配置：sigaction() ..... 167
20.14 等待信号：pause() ..... 167
20.15 小结 ..... 168
20.16 习题 ..... 168
第 21 章 信号：信号处理器 ..... 169
21.1 设计信号处理器 ..... 169
21.1.1 信号没有排队（再论） ..... 169

21.1.2 可重入和异步信号安全的函数 ..... 169
21.1.3 全局变量和 sig_atomic_t 数据类型 ..... 169
21.2 终止信号处理器的其它方法 ..... 169
21.2.1 从信号处理器中执行非局部跳转 ..... 169
21.2.2 异常地终止进程：abort() ..... 169
21.3 在备用堆栈中处理信号：sigaltstack() ..... 169
21.4 SA_SIGINFO 标志 ..... 169
21.5 系统调用的中断和重启 ..... 169
21.6 小结 ..... 169
21.7 习题 ..... 169
第 22 章 信号：高级特性 ..... 170
22.1 Core Dump 文件 ..... 170
22.2 递送、配置、和处理的特殊情况 ..... 170
22.3 可中断和不可中断的进程睡眠状态 ..... 170
22.4 硬件产生的信号 ..... 170
22.5 同步和异步信号产生 ..... 170
22.6 定时和信号递送顺序 ..... 170
22.7 signal()的实现和可移植性 ..... 170
22.8 实时信号 ..... 170
22.8.1 发送实时信号 ..... 170
22.8.2 处理实时信号 ..... 170
22.9 使用掩码来等待信号：sigsuspend() ..... 170
22.10 同步等待信号 ..... 170
22.11 通过文件描述符接收信号 ..... 170
22.12 使用信号进行进程间通信 ..... 170
22.13 早期信号 API（System V 和 BSD） ..... 171
22.14 小结 ..... 171
22.15 习题 ..... 171

第 23 章 定时器和睡眠 ..... 172
23.1 间隔定时器 ..... 172
23.2 调度和定时器的精确度 ..... 172
23.3 设置阻塞操作的超时 ..... 172
23.4 固定间隔挂起执行（睡眠） ..... 172
23.4.1 低精度睡眠：sleep() ..... 172
23.4.2 高精度睡眠：nanosleep() ..... 172
23.5 POSIX 时钟 ..... 172
23.5.1 获取时钟的值：clock_gettime() ..... 172
23.5.2 设置时钟的值：clock_settime() ..... 172
23.5.3 获取特定进程或线程的时钟 ID ..... 172
23.5.4 增强的高精度睡眠：clock_nanosleep() ..... 172
23.6 POSIX 间隔定时器 ..... 172
23.6.1 创建定时器：timer_create() ..... 172
23.6.2 装备和解除定时器：timer_settime() ..... 172
23.6.3 获取定时器的当前值：timer_gettime() ..... 173
23.6.4 删除定时器：timer_delete() ..... 173
23.6.5 通过信号通知 ..... 173
23.6.6 定时器溢出 ..... 173
23.6.7 通过线程通知 ..... 173
23.7 通过文件描述符通知的定时器：timerfd API ..... 173
23.8 小结 ..... 173
23.9 习题 ..... 173
第 24 章 进程创建 ..... 174
第 25 章 进程结束 ..... 175
第 26 章 监控子进程 ..... 176
第 27 章 程序执行 ..... 177
第 28 章 进程创建和程序执行的更多细节 ..... 178

第 29 章 线程：介绍 .....179
第 30 章 线程：同步 .....180
第 31 章 线程：线程安全和线程存储 .....181
第 32 章 线程：线程取消 .....182
第 33 章 线程：更多细节 .....183
第 34 章 进程组、会话和任务控制 .....184
第 35 章 进程优先级和调度 .....185
第 36 章 进程资源 .....186
第 37 章 Daemon.....187
第 38 章 编写安全的特权程序 .....188
第 39 章 能力 .....189
第 40 章 登录会计 .....190
第 41 章 共享库基础 .....191
第 42 章 共享库高级特性 .....192
第 43 章 进程间通信简介 .....193
第 44 章 管道和 FIFO .....194
第 45 章 System V IPC 介绍.....195
第 46 章 System V 消息队列.....196
第 47 章 System V 信号量.....197
第 48 章 System V 共享内存.....198
第 49 章 内存映射 .....199
第 50 章 虚拟内存操作 .....200
第 51 章 POSIX IPC 介绍.....201
第 52 章 POSIX 消息队列.....202
第 53 章 POSIX 信号量.....203
第 54 章 POSIX 共享内存.....204
第 55 章 文件锁 .....205
第 56 章 Sockets：介绍 .....206

第 57 章 Sockets：UNIX Domain .....207
第 58 章 Sockets：TCP/IP 网络基础.....208
第 59 章 Sockets：Internet Domain.....209
第 60 章 Sockets：服务器设计 .....210
第 61 章 Sockets：高级主题 .....211
第 62 章 终端 .....212
第 63 章 可选 I/O 模型 .....213
第 64 章 伪终端 .....214
附录 A：跟踪系统调用 .....215
附录 B：解析命令行参数 .....216
附录 C：转换 NULL 指针.....217
附录 D：内核配置.....218
附录 E：更多信息来源.....219
附录 F：部分习题解答.....220
参考书目.....221
索引.....222

# 前言

## 主题

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

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

## 目标读者

本书主要面向以下读者：

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

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

# Linux 和 UNIX

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

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

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

# 使用和组织

你至少可以按两种方式来使用本书：

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

我把本书分为以下几部分：

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

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

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

# 例子程序

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

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

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

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

## 习题

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

## 标准和可移植性

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

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

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

## Linux 内核和 C 库版本

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

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

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

## 其它语言使用编程接口

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

## 关于作者

（略）

## 致谢

（略）

## 许可

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

## 网站和例子程序源代码

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

## 反馈

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

Michael Timothy Kerrisk

Munich, Germany and Christchurch, New Zealand

August 2010

mtk@man7.org

# 第 1 章 历史和标准

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

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

## 1.1 UNIX 和 C 简史

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

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

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

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

**UNIX 第一至第六版**

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

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

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

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

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

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

**BSD 和 System V 的诞生**

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

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

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

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

1993 年的最终发布版：4.4BSD。

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

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

## 1.2 Linux 简史

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

### 1.2.1 GNU 项目

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

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

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

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

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

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

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

## 1.2.2 Linux 内核

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

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

> 你是否怀念 minix-1.1 版时的日子？那时人们干劲十足，自己编写设备驱动程序。你是否手头正缺少一个很好的项目，并且非常渴望为符合自己的需要动手修改一个操作系统？当几乎所有的程序都能在 Minix 上运行时，你是否感到非常失望？不再有为了调通一个巧妙的程序而整夜不睡觉的夜猫子？那么本消息（邮件、公告）可能正是为你而发布的:-)。
>
> 正如我一个月前所提到的，我正在开发一个用于 AT-386 微机类似于 Minix 的操作系统。它目前已经达到了可用的程度(当然，能不能用还依赖于你的具体要求)，而且我很高兴把源代码拿出来广泛发布。目前它的版本是 0.02(加上已经编制好的(很小的)补丁程序，就是 0.03)，但是我已经在它上面成功地运行了 bash/gcc/gnu-make/gnu-sed/压缩程序等。

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

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

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

Linus

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

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

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

**BSD**

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

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

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

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

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

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

**Linux 内核版本号**

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

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

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

关于改善开发模型的谈论时不时都有进行，但是核心细节基本保持如下：

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

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

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

### 移植到其它硬件体系架构

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

**Linux 发行版**

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

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

## 1.3 标准化

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

### 1.3.1 C 编程语言

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

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

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

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

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

## 1.3.2 第一个 POSIX 标准

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

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

**POSIX.1 和 POSIX.2**

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

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

API。这样的操作系统才可以获得 POSIX.1 依从的证明。

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

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

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

**FIPS 151-1 和 FIPS 151-2**

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

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

## 1.3.3 X/Open 公司和开放组织

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

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

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

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

### 1.3.4 SUSv3 和 POSIX.1-2001

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

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

SUSv3 基本规范大概有 3700 页，分成以下四个部分：

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

含了 1123 个系统接口）。

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

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

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

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

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

**POSIX 依从、XSI 依从、和 XSI 扩展**

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

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

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

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

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

在后面章节中，当我们说 SUSv3 依从时，指的是 XSI 依从。

**未规定和软规定**

有时候我们会谈到某个接口在 SUSv3 中“未规定”或“软规定”

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

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

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

**遗留特性**

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

## 1.3.5 SUSv4 和 POSIX.1-2008

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

SUSv4 的变化比 SUSv3 要少很多。最重要的改变如下：

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

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

在本书的后面部分，我们会在相关主题被讨论时标注 SUSv4 的变化。

### 1.3.6 UNIX 标准时间线

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

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

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

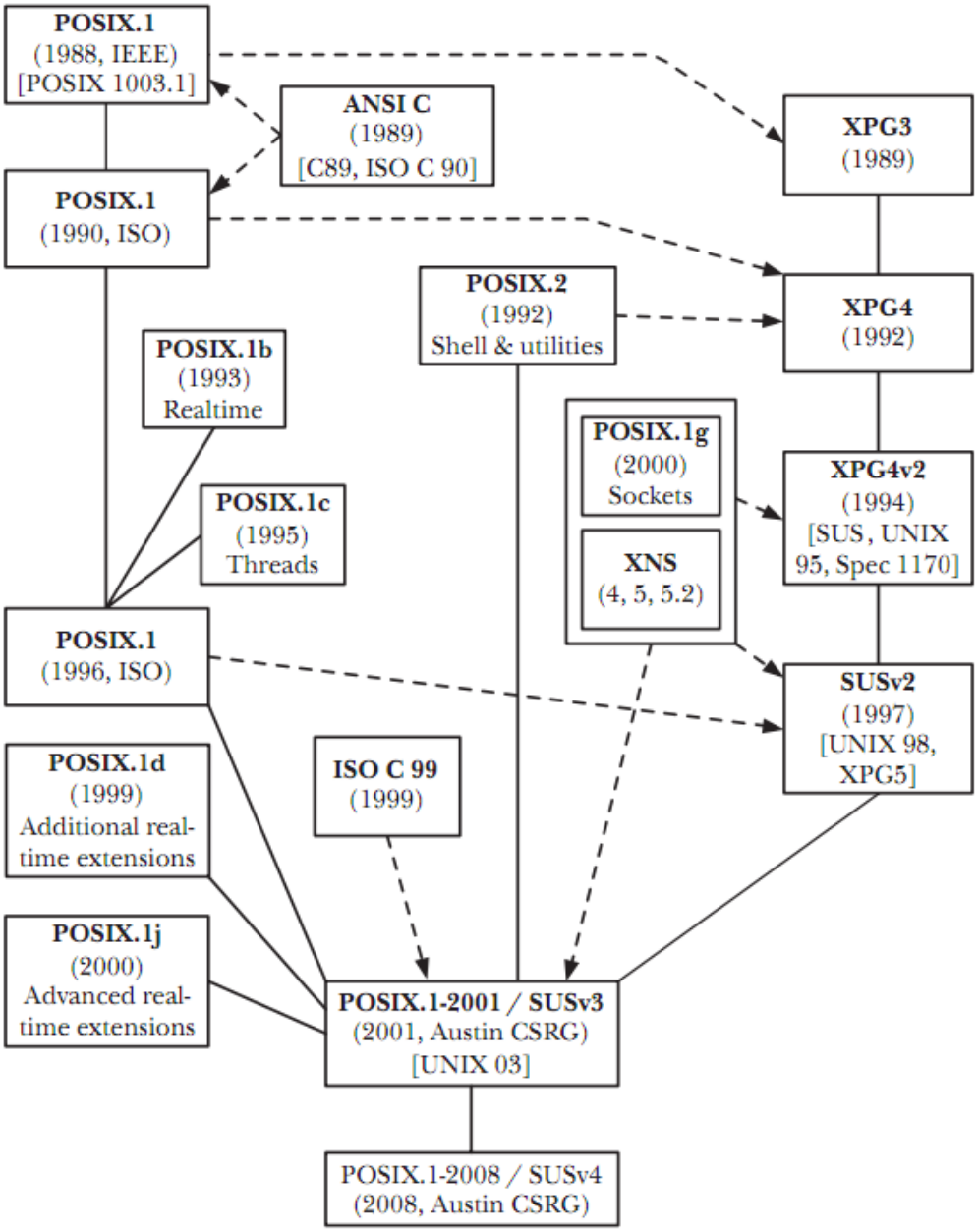


*图 1-1：各种 UNIX 和 C 标准之间的关系*

## 1.3.7 实现标准

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

### 1.3.8 Linux、标准、和 Linux 标准基础

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

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

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

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

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

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

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

## 1.4 小结

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

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

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

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

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

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

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

## 更多信息

（略）

# 第 2 章 基础概念

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

## 2.1 操作系统的核心：内核

操作系统这个术语通常表示两个不同的意思：

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

内核这个术语通常则代表第二种意思，本书所说的操作系统也是这种意思。

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

### 内核执行的任务

内核主要执行以下任务：

- 进程调度：计算机只有一个或少数中央处理单元（CPU）来执行程序指令。和其它 UNIX 系统一样，Linux 是抢先式多任务操作系统，多任务表示多个进程（正在运行的程序）可以同时在内存中，而且每个都可以使用 CPU。抢先式表示由内核进程调度器支配哪个进程获得 CPU，以及确定进程使用 CPU 的时间。
- 内存管理：虽然计算机内存容量在近十来年变得非常庞大，但软件的体积也相应地快速增长，因此物理内存（RAM）仍然是一种有限的资源，内核必须以公平和有效的方式使多个进程间共享物理内存。和多数现代操作系统一样，Linux 采用了虚拟内存管理机制（6.4 节），这个技术有两

个主要的优点：

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

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

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

### 内核模式和用户模式

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

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

### 进程 VS 内核对系统的视角

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

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

相比之下，运行系统的内核则知道和控制了所有一切。内核为系统中所有运行进程提供协助。内核决定哪个进程获得 CPU 访问权，什么时候获得，使用多长时间。内核维护一组进程数据结构，包含所有运行进程的所有信息，并根据进

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

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

## 2.2 Shell

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

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

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

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

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

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

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

## 2.3 用户和组

系统的每个用户都有唯一标识，用户可能属于某个或几个组。

### 用户

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

- 组 ID：数字的组 ID，用户加入的第一个组。
- home 目录：用户登录后的初始目录。
- 登录 shell：用来解释用户命令的 shell 名称。

这个密码记录可能还包含用户的密码，以加密形式存储。但是由于安全原因，

通常密码会存放在单独的 shadow 密码文件中，只对超级用户可读。

### 组

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

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

### 超级用户

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

## 2.4 单一目录层次、目录、链接、和文件

内核维护一个单一层次的目录结构，来组织系统中的所有文件。（这和 Microsoft Windows 明显不同，后者的每个磁盘分区都有自己的目录层次）。层次的最底部是 root 目录，名为“/”（斜线）。所有文件和目录都是 root 目录直接或间接的子目录。图 2-1 显示了这种文件结构的一个例子：

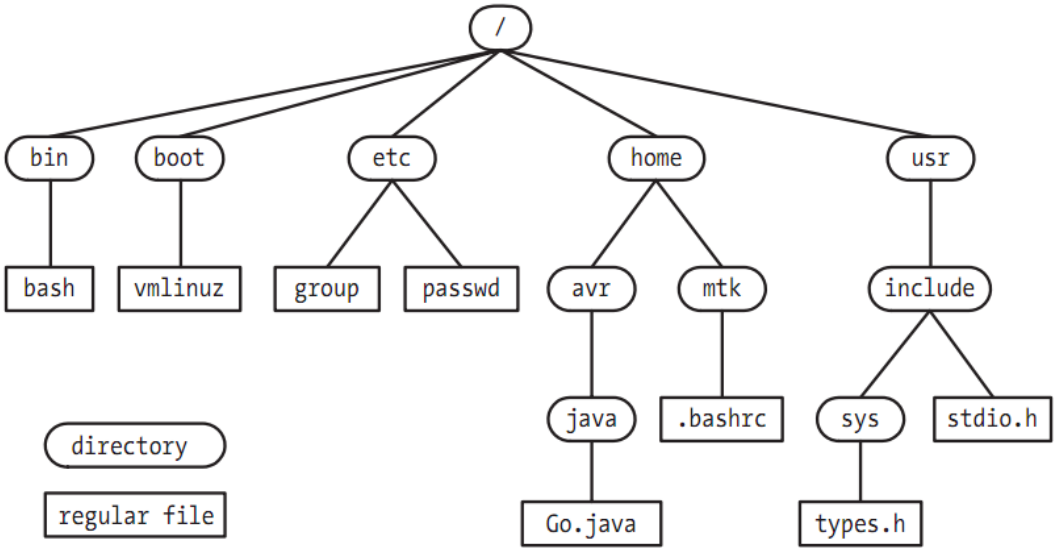


*图 2-1：Linux 单一目录层次的子集*

### 文件类型

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

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

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

### 符号链接

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

指针。如果符号链接的目标本身也是一个符号链接，那么这个过程可能产生递归。（内核强制解引用的数量限制，以避免环形符号链接）。如果符号链接引用的文件不存在，就称为 dangling 链接。

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

## 文件名

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

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

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

## 路径名

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

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

路径名描述了一个文件在单一目录层次架构中的具体位置，可以是绝对或相

对路径：

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

### 当前工作目录

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

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

### 文件所有权和权限

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

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

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

# 2.5 文件 I/O 模型

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

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

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

**文件描述符**

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

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

**stdio 库**

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

## 2.6 程序

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

**过滤器**

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

**命令行参数**

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

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

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

## 2.7 进程

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

从内核的视角来看，进程是内核必须为其共享许多计算机资源的实体。由于资源是有限的（如内存），内核一开始只分配一定的资源给进程，然后在进程的生命周期过程中，根据进程的需要和整个系统的负载情况，来调整这些分配。当进程终止时，进程使用的所有资源都会被回收，并提供给其它进程重新使用。其

它一些资源（如 CPU 和网络带宽），还必须在所有进程中公平地共享。

### 进程内存布局

进程逻辑上划分为以下部分，称为段（segment）：

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

### 进程创建和程序执行

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

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

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

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

### 进程 ID 和父进程 ID

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

### 进程终止和终止状态

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

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

### 进程用户和组标识符（凭证）

每个进程都有一组相关的用户 ID（UID）和组 ID（GID）。包括：

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

### 特权进程

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

特权进程创建的进程也拥有特权，例如由 root 用户启动的 login shell。另一

种使进程拥有特权的方法是通过设置用户 ID 机制，允许进程使用程序文件拥有者的身份执行该进程。

### 能力

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

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

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

### init 进程

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

### Daemon 进程

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

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

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

### 环境列表

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

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

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

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

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

### 资源限制

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

当新进程创建时，会继承父进程的资源限制设置。

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

## 2.8 内存映射

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

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

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

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

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

## 2.9 静态和共享库

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

### 静态库

静态库（有时候称为 archive）是早期 UNIX 系统唯一支持的库类型。静态库本质上是结构化的已编译对象模块。要使用静态库中的函数，我们在构建程序时使用链接命令来指定该库。链接器为应用程序引用的所有函数找到相应的静态库模块，然后从静态库中提取出所需的对象模块，并复制到最终的可执行文件中。我们称这样的程序是静态链接的。

每个静态链接的程序都从库中复制了需要的对象模块，这种方式导致了一些缺点。其中之一就是不同可执行文件中的对象代码重复浪费了磁盘空间。当使用

相同静态库的多个程序一起执行时，也浪费了内存空间；每个程序都会有相同的函数拷贝在内存中。此外如果库函数需要修改，那么在重新编译该函数并添加到静态库中后，所有使用该函数的应用都必须重新与库进行链接。

**共享库**

共享库是为了解决静态库的问题而设计的。

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

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

## 2.10 进程间通信和同步

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

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

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

- 信号，用来指示发生了某个事件。
- 管道（shell 用户熟知的“|”操作符）和 FIFO，用来在进程间传输数据。
- socket，用来在进程间传输数据，既可以在同一计算机中，也可以在通过网络连接的不同计算机中进行通信。
- 文件锁，允许进程锁住文件的某个区域，阻止其它进程读取和更新该区

域的文件内容。

- 消息队列，用来在不同进程间交换消息（数据包）。
- 信号量，用来同步进程间的动作。
- 共享内存，允许两个或多个进程共享一块内存。当一个进程修改共享内存的内容时，所有进程都可以立即看到这个修改。

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

## 2.11 信号

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

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

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

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

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

当进程接收到一个信号时，它可以根据不同的信号类型，采取以下动作：

- 进程忽略信号

- 进程被信号 kill
- 进程暂时挂起，稍后在收到特别的信号后再继续。

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

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

## 2.12 线程

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

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

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

## 2.13 进程组和 shell 工作控制

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

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

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

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

## 2.14 会话、控制终端、和控制进程

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

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

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

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

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

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

## 2.15 伪终端

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

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

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

## 2.16 日期和时间

进程一般会关心两种时间类型：

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

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

## 2.17 客户端-服务器体系架构

在本书的一些地方，我们会讨论客户端-服务器应用的设计和实现：

客户端-服务器应用分为两个组件：

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

有时候，客户端和服务器可能需要进行请求和返回的扩展对话。

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

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

服务器可以实现许多服务，例如：

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

把服务封装在一个服务器中有许多好处，例如：

- 高效，由服务器管理资源并提供服务，比在每台计算机中提供相同资源要便宜而且高效。
- 可控、协同、和安全，通过把资源（特别是信息资源）控制在单一位置，服务器可以控制资源的协同访问（如两个客户端不能同时更新相同的信息块），也可以使资源仅对选定客户端可用，提高安全性。
- 在多样环境中操作，在网络环境下，存在许多各不相同的客户端，服务器可以运行在不同的硬件和操作系统平台中。

## 2.18 实时

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

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

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

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

## 2.19 /proc 文件系统

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

/proc 是虚拟的文件系统，它以文件系统的文件和目录的方式，提供内核数据结构的访问接口。这样就可以轻松地查看或修改许多系统属性。另外有一些/proc/PID 形式的目录（PID 是进程 ID），允许我们查看系统每个运行进程的信息。

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

在我们讨论许多 Linux 编程接口的时候，我们会同时描述相关的/proc 文件。

12.1 节提供了/proc 文件系统的更多信息。没有任何标准对/proc 文件系统进行了定义，因此我们对其的讨论是特定于 Linux 的。

## 2.20 小结

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

# 第 3 章 系统编程概念

本章讲解系统编程的许多必备主题。首先介绍系统调用及其执行的详细步骤，然后考虑库函数及其与系统调用的区别，同时结合讲解（GNU）C 库。

当我们调用系统调用或库函数时，总是应该检查它的返回值，来确定调用是否成功。我们描述了如何检查函数返回值，并介绍了一组错误诊断函数，它们用在本书的多数示例代码中。

最后我们考察许多与可移植编程相关的问题，特别是使用 SUSv3 提供的特性测试宏和标准系统数据类型。

## 3.1 系统调用

系统调用是进入内核的受控入口点，允许进程请求内核代表进程执行某些动作。内核通过系统调用 API 为应用程序提供一系列服务。这些服务包括：创建新进程、执行 I/O 操作、创建进程间通信用的管道等等（syscalls(2)手册页列出了 Linux 的所有系统调用）。

在描述系统调用工作的细节之前，我们先看一些基本要点：

- 系统调用把处理器状态从用户模式切换到内核模式，这样 CPU 才能访问受保护的内核内存。
- 系统调用是固定的。每个系统调用都由一个唯一的数值标识（这个数值通常对应用不可见，应用使用系统调用的名字来标识）。
- 每个系统调用都可以有一组参数，指定用户空间和内核空间之间要传递的信息。

从编程的角度来看，调用系统调用和调用 C 函数是非常相似的。但是在幕后，执行系统调用需要许多步骤。为了解释系统调用的步骤，我们来看下 x86-32 硬件体系架构下系统调用的每一个步骤：

1. 应用程序通过调用 C 库的包装函数发起系统调用。

2. 包装函数必须把系统调用的所有参数传递给系统调用陷阱处理例程（马上讲到）。这些参数是通过堆栈传递给包装函数的，但是内核要求参数存放在特定的寄存器中。包装函数把参数拷贝到这些寄存器中。
3. 由于所有系统调用都以同样的方式进入内核，内核必须采用某种方法来标识不同的系统调用。因此包装函数会把系统调用数值也复制到特定的 CPU 寄存器（%eax）。
4. 包装函数执行一个 trap 机器指令（int 0x80），这样就会把处理器从用户模式切换到内核模式，并从系统 trap 向量的 0x80 位置开始执行代码。更加现代的 x86-32 体系架构实现了 sysenter 指令，比传统的 int 0x80 trap 指令提供更快的进入内核模式的方法。从 2.6 内核和 glibc 2.3.2 开始支持使用 sysenter。
5. 作为 trap 到 0x80 位置之后的响应，内核调用 system_call()例程（位于汇编文件 arch/i386/entry.S）来处理这个 trap。这个处理器：
   a) 把寄存器的值保存到内核堆栈中（6.5 节）。
   b) 检查系统调用数值的有效性。
   c) 调用适当的系统调用服务例程，使用系统调用数值作为索引，从系统调用服务例程表中得到（内核变量 sys_call_table）。如果系统调用服务例程需要参数，它会首先检查参数的有效性；例如，它会检查地址指向用户内存的合法位置。然后服务例程执行请求的任务，可能包括：修改参数指定地址的值，在用户内存和内核内存之间传输数据（例如 I/O 操作）。最后，服务例程返回一个结果状态给 system_call()例程。
   d) 从内核堆栈还原寄存器的值，并把系统调用返回值存放在堆栈中。
   e) 返回到包装函数，同时把处理器切回至用户模式。
6. 如果系统调用服务例程的返回值表示出现错误，包装函数就使用这个值设置全局变量 errno（3.4 节）。包装函数然后返回至调用方，提供一个整数返回值表示系统调用成功还是失败。

在 Linux 中，系统调用服务例程通常返回非负值表示成功，返回负值表示错误，这个错误值的绝对值就是相应的 errno 常量。当服务例程返回了负数值时，C 库包装函数把它取正，并复制给 errno，然后包装函数返回-1 表示出现了错误。

这个惯例假设系统调用服务例程在成功时不会返回负数值。但是对于少数几个服务例程这个假设并不成立。一般来说这并不存在问题，因为取反后的 errno 值也没有超过合法的负数返回值范围。但是这个惯例确实导致了一个问题：fcntl()系统调用的 F_GETOWN 操作，我们会在 63.3 节描述。

图 3-1 使用 execve()系统调用阐明了上面的步骤。在 Linux/x86-32 上，execve()的系统调用数值为 11（__NR_execve）。因此在 sys_call_table 向量中，条目 11 包含了 sys_execve()的地址，也就是这个系统调用的服务例程。（在 Linux 中，系统调用服务例程通常命名为 sys_xyz()，其中 xyz 就是正在讨论的这个系统调用）。

前面段落给出的信息已经超过了学习本书后面知识的需要。但是它说明了很重要的一点，即使是一个简单的系统调用，也需要完成许多工作，因此系统调用存在很小但仍然可观的开销。

作为系统调用开销的一个例子，我们来考虑 getppid()系统调用，它只是简单地返回调用进程的父进程 ID。在作者的 x86-32 Linux 2.6.25 系统中，调用 getppid()一千万次大约需要 2.2 秒才能完成，大约每次调用需要 0.3 微秒。在相同的系统中，一千万次 C 函数调用（简单地返回一个整数）只需要 0.11 秒，大约是调用 getppid()时间的二十分之一。当然，多数系统调用的开销比 getppid()要大得多。

从 C 程序的角度来看，调用 C 库包装函数和调用相应的系统调用服务例程是等价的，因此在本书的剩余部分，我们使用“调用系统调用 xyz()”来表示“调用包装函数来调用系统调用 xyz()”。

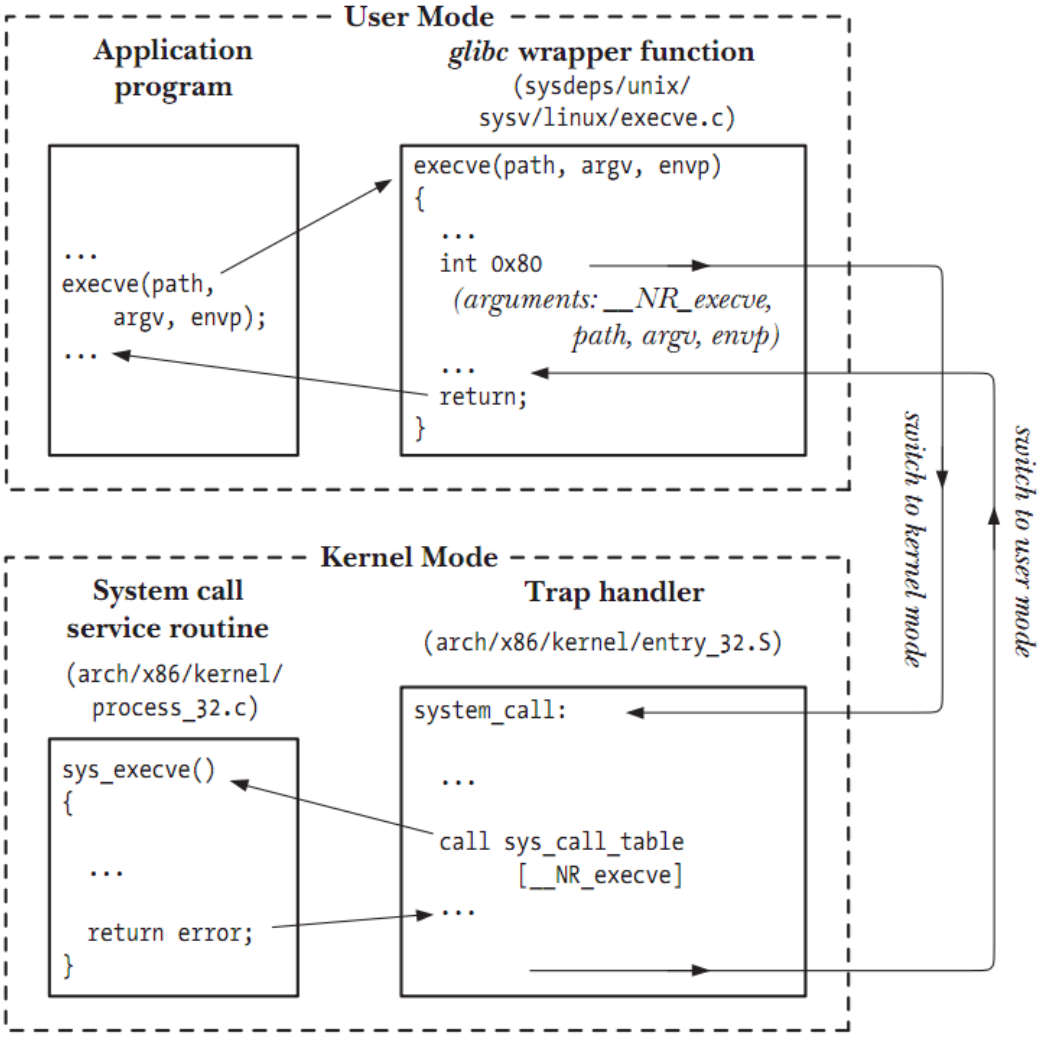


*图 3-1：执行系统调用的步骤*

附录 A 描述了 strace 命令，它可以用来跟踪应用发起的系统调用，可以作为调试程序时使用。

## 3.2 库函数

库函数指的就是标准 C 库的函数。这些函数的作用是各种各样的，包括打开文件、转换时间到人类可读的格式、以及比较两个字符串。

许多库函数不使用系统调用（例如字符串操作函数）。另外一些库函数则基于系统调用之上。例如 fopen()库函数使用 open()系统调用来打开文件。通常库函数设计用来提供比底层系统调用更加友好的接口。例如 printf()函数提供输出格式和数据缓冲，而 write()系统调用只是输出一块字节。类似地，malloc()和 free()

函数执行许多记录工作，使得分配和释放内存的任务比直接使用底层 brk()系统调用要简单得多。

## 3.3 标准 C 库；GNU C 库（glibc）

在不同 UNIX 实现中，标准 C 库也有不同的实现。Linux 中最常用的实现是 GNU C 库（glibc, http://www.gnu.org/software/libc/）。

**确定系统中 glibc 的版本**

有时候我们需要确定系统中 glibc 的版本。我们可以在 shell 中运行 glibc 共享库，就把它当成是一个可执行程序。当我们把这个库当作可执行文件运行时，它会显示许多信息，包括版本号：

```
$ /lib/libc.so.6
GNU C Library stable release version 2.10.1, by Roland McGrath et al.
Copyright (C) 2009 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
Compiled by GNU CC version 4.4.0 20090506 (Red Hat 4.4.0-4).
Compiled on a Linux >>2.6.18-128.4.1.el5<< system on 2009-08-19.
Available extensions:
        The C stubs add-on version 2.1.2.
        crypt add-on version 2.1 by Michael Glad and others
        GNU Libidn by Simon Josefsson
        Native POSIX Threads Library by Ulrich Drepper et al
        BIND-8.2.3-T5B
        RT using linux kernel aio
For bug reporting instructions, please see:
<http://www.gnu.org/software/libc/bugs.html>.
```

在某些 Linux 发行版中，GNU C 库没有安装在/lib/libc.so.6 路径下。有一个方法可以确定库的位置，就是对一个链接到 glibc 的可执行程序（多数可执行程序都会链接），运行 ldd（列出动态依赖）命令。然后我们就可以从库依赖列表中找到 glibc 共享库：

```
$ ldd myprog | grep libc
        libc.so.6 => /lib/tls/libc.so.6 (0x4004b000)
```

应用程序有两个办法可以确定 GNU C 库的版本：测试常量宏；或调用一个库函数。从 2.0 版本开始，glibc 定义了两个常量：__GLIBC__和__GLIBC_MINOR__，可以用来在编译时使用（#ifdef 语句）。在安装了 glibc 2.12 的系统中，这两个常量的值是 2 和 12。但是这些常量的作用有限，因为某个程序可以在这个系统编译，但拿到另一个系统去运行，两个系统安装了不同版本的 glibc。为了处理这个可能性，程序可以调用 gnu_get_libc_version()函数来确定系统当前使用的 glibc 的版本。

```
#include <gnu/libc-version.h>

const char *gnu_get_libc_version(void);
                            返回一个 null 结尾的静态字符串，包含 GNU C 库版本号
```

gnu_get_libc_version()函数返回一个字符串指针，例如”2.12”。

> 我们还可以使用 confstr()函数得到_CS_GNU_LIBC_VERSION 配置的值来获得版本信息。这个调用返回“glibc 2.12”形式的字符串。

## 3.4 系统调用和库函数的错误处理

几乎每个系统调用和库函数都会返回一个状态值，表示调用成功或失败。我们应该检查这个状态值来查看调用是否成功。如果调用失败了，就采取适当的措施——至少程序应该显示错误消息，警告发生了未预料的错误。

尽管为了节省打字时间，我们经常受诱惑而忽略这些检查（特别是许多 UNIX 和 Linux 例子程序都不检查返回值），这是负经济的作法。因为有时候我们没有对一个“不太可能失败”的系统调用检查返回值，结果就是浪费许多小时用于调试错误。

> 少数几个系统调用确实永远不会失败。例如 getpid()总是会成功地返回进程 ID，_exit()总是会终止一个进程。这种系统调用的返回值就不需要检查啦。

## 处理系统调用错误

每个系统调用的手册页都注明了可能的返回值，并显示哪些返回值表示错误。通常-1 表示错误，因此可以用以下代码检查系统调用的返回值：

```
fd = open(pathname, flags, mode);   /* system call to open a file */
if (fd == -1) {
    /* Code to handle the error */
}
...
if (close(fd) == -1) {
    /* Code to handle the error */
}
```

当系统调用失败时，它设置全局整型变量 errno 为一个正数值，标识具体发生的错误。包含<errno.h>头文件来提供 errno 的定义，以及许多错误数值的常量定义。所有错误的符号名都以 E 开头。每个手册页的 ERRORS 节列出了该系统调用可能的 errno 值。下面是一个简单的例子，使用 errno 来诊断系统调用错误：

```
cnt = read(fd, buf, numbytes);
if (cnt == -1) {
    if (errno == EINTR)
        fprintf(stderr, "read was interrupted by a signal\n");
    else {
        /* Some other error occurred */
    }
}
```

系统调用和库函数成功时不会重置 errno 为 0，因此如果之前的调用出现了错误，这个值会被设为非 0 值并保持到下次出错。此外 SUSv3 甚至还允许成功的函数调用设置 errno 为非 0 值（很少函数这样做）。因此当检查错误时，我们应该首先检查函数返回值是否表示出错，然后才检查 errno 来确定具体的错误原因。

少数系统调用（如 getpriority()）成功时返回-1 是合理的。要确定这种调用是否发生错误，我们需要在调用前设置 errno 为 0，调用完成后再检查 errno 值。如果调用返回-1 并且 errno 非 0，就发生了错误（有些库函数也是这样）。

系统调用失败后的常见动作是根据 errno 值打印一条错误消息。perror()和 strerror()库函数提供了这个功能。

perror()函数打印 msg 参数指向的字符串，紧跟着打印对应于当前 errno 值的错误消息。

```
#include <stdio.h>

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

处理系统调用失败的最简单方法如下：

```
fd = open(pathname, flags, mode);
if (fd == -1) {
    perror("open");
    exit(EXIT_FAILURE);
}
```

strerror()函数返回 errnum 参数指定的错误值相对应的错误字符串。

```
#include <string.h>

char *strerror(int errnum);
                                        返回对应于 errnum 的错误字符串
```

strerror()返回的字符串可能是静态分配的，这意味着它的值可能会被随后的 strerror()调用覆盖。

如果 errnum 指定了不被认可的错误数值，strerror()返回一个字符串“Unknown error nnn”。在某些其它实现中，strerror()在这种情况下返回 NULL。

由于 perror()和 strerror()函数是区域敏感的（10.4 节），错误描述会按本地语言显示出来。

### 处理库函数错误

不同的库函数返回不同的数据类型，并且以不同的值表示错误。（查看每个函数的手册页）。按我们的想法，库函数可以划分为以下几类：

- 有些库函数采用系统调用一样的方式返回错误：-1 返回值，并设置 errno 来指示错误。例如 remove()函数，用来移除一个文件（使用 unlink()系统调用）或目录（使用 rmdir()系统调用）。这类库函数的错误处理和系统调

用的错误处理一样。

- 有些库函数返回其它值表示错误，并且也设置 errno 来指示特定的错误。例如 fopen()返回 NULL 指针表示出错，并根据底层系统调用错误来设置 errno 的值。perror()和 strerror()函数可以用来诊断这种错误。
- 其它库函数完全不使用 errno。这种函数的错误诊断和处理要根据该函数的手册页描述来进行。对于这一类型的函数，使用 errno, perror(), strerror()来诊断错误是不行的。

## 3.5 本书示例程序的说明

在这一节中，我们描述本书示例程序经常采用的惯例和特性。

### 3.5.1 命令行选项和参数

本书的许多示例程序依赖于命令行选项和参数，来确定自己的行为。

传统的 UNIX 命令行选项包括一个起始连字符，一个字母标识选项，以及一个可选的参数。（GNU 实用工具提供一个扩展的选项语法，包括两个起始连字符，紧跟一个字符串标识选项，以及可选的参数）。要解析命令行选项，我们使用标准的 getopt()库函数（附录 B 描述）。

我们的每个有命令行选项的示例程序，都为用户提供了一个简单的帮助机制：如果以--help 选项调用，程序会显示命令行选项和参数的使用信息。

### 3.5.2 常用函数和头文件

多数示例程序都包含了同一个头文件，该文件包含常用的定义，以及一组常用的函数。我们在这一节讨论头文件和函数。

#### 常用头文件

清单 3-1 是本书几乎所有程序都要使用的头文件。这个头文件包含了许多其它头文件，提供给许多示例程序使用，定义了一个 Boolean 数据类型，定义了计

算最小和最大值的宏。使用这个头文件，我们的示例程序会更加短小简洁。

*清单 3-1：多数示例程序使用的头文件*

---------- lib/tlpi_hdr.h

```
#ifndef TLPI_HDR_H
#define TLPI_HDR_H      /* Prevent accidental double inclusion */
#include <sys/types.h>  /* Type definitions used by many programs */
#include <stdio.h>      /* Standard I/O functions */
#include <stdlib.h>     /* Prototypes of commonly used library functions,
                           plus EXIT_SUCCESS and EXIT_FAILURE constants */
#include <unistd.h>     /* Prototypes for many system calls */
#include <errno.h>      /* Declares errno and defines error constants */
#include <string.h>     /* Commonly used string-handling functions */
#include "get_num.h"    /* Declares our functions for handling numeric
                           arguments (getInt(), getLong()) */
#include "error_functions.h"  /* Declares our error-handling functions */
typedef enum { FALSE, TRUE } Boolean;
#define min(m,n) ((m) < (n) ? (m) : (n))
#define max(m,n) ((m) > (n) ? (m) : (n))
#endif
```

---------- lib/tlpi_hdr.h

## 错误诊断函数

为了简化示例程序的错误处理，我们使用错误诊断函数，如清单 3-2 所示：

*清单 3-2：常用错误处理函数声明*

---------- lib/error_functions.h

```
#ifndef ERROR_FUNCTIONS_H
#define ERROR_FUNCTIONS_H
void errMsg(const char *format, ...);
#ifdef __GNUC__
/* This macro stops 'gcc -Wall' complaining that "control reaches
       end of non-void function" if we use the following functions to
       terminate main() or some other non-void function. */
#define NORETURN __attribute__ ((__noreturn__))
#else
#define NORETURN
```

```
#endif
void errExit(const char *format, ...) NORETURN ;
void err_exit(const char *format, ...) NORETURN ;
void errExitEN(int errnum, const char *format, ...) NORETURN ;
void fatal(const char *format, ...) NORETURN ;
void usageErr(const char *format, ...) NORETURN ;
void cmdLineErr(const char *format, ...) NORETURN ;
#endif
```

---------- lib/error_functions.h

要诊断系统调用和库函数的错误，我们使用 errMsg(), errExit(), err_exit(), 和 errExitEN()。

```
#include "tlpi_hdr.h"

void errMsg(const char *format, ...);
void errExit(const char *format, ...);
void err_exit(const char *format, ...);
void errExitEN(int errnum, const char *format, ...);
```

errMsg()函数向标准错误打印一条消息。它的参数列表和 printf()是一样的，除了 errMsg()会自动添加一个换行字符到输出字符串中。errMsg()函数打印当前 errno 错误值对应的错误信息，包括错误名字，如 EPERM；加上 strerror()返回的错误描述；再随后是参数列表格式化后的输出字符串。

errExit()函数和 errMsg()类似，但同时还会终止程序，它调用了 exit()，或者如果环境变量 EF_DEMPCORE 设置为非空字符串，会调用 abort()来产生一个 core dump 文件，以便于调试（22.1 节解释 core dump 文件）。

err_exit()函数类似于 errExit()，但有以下两个区别：

- 它在打印错误消息之前不冲洗标准输出。
- 它通过_exit()而不是 exit()终止进程。这样会使进程不冲洗 stdio 缓冲区，也不调用退出处理器，直接终止进程。

err_exit()的区别的细节到第 25 章就会变得清晰，我们在那里会讨论_exit()和 exit()的区别，以及子进程如何对待 stdio 缓冲区和退出处理器。现在我们只说明

一点，父进程出现错误需要终止时，不应该冲洗子进程的 stdio 拷贝，也不应该调用 exit 处理器，这时候使用 err_exit()就非常有用了。

errExitEN()函数和 errExit()函数是相同的，但是它不打印当前 errno 错误对应的错误消息，它打印 errnum 指定的错误对应的错误消息。

我们主要在采用 POSIX 线程 API 的程序中使用 errExitEN()。和传统 UNIX 系统调用不一样（返回-1 表示错误），POSIX 线程函数通过返回错误数值来诊断错误（POSIX 线程函数返回 0 表示成功）。

我们可以使用下面代码来诊断 POSIX 线程函数：

```
errno = pthread_create(&thread, NULL, func, &arg);
if (errno != 0)
    errExit("pthread_create");
```

但是这样做不高效，因为 errno 宏在多线程环境下会扩展为一个函数调用，并且返回一个可以修改的左值。因此第次使用 errno 都会引起函数调用。errExitEN()函数允许我们编写更加高效地错误处理代码，等价于上面代码：

```
int s;
s = pthread_create(&thread, NULL, func, &arg);
if (s != 0)
    errExitEN(s, "pthread_create");
```

要诊断其它类型的错误，我们使用 fatal(), usageErr(), 和 cmdLineErr()。

```
#include "tlpi_hdr.h"

void fatal(const char *format, ...);
void usageErr(const char *format, ...);
void cmdLineErr(const char *format, ...);
```

fatal()函数用来诊断通用错误，包括不设置 errno 的库函数。它的参数列表和 printf()一样，除了它也会在输出末尾自动添加换行字符。它打印格式化后的输出到标准错误，然后和 errExit()一样终止程序。

usageErr()函数用来诊断命令行参数用法的错误。它的参数也和 printf()一样，并且打印字符串 Usage: 然后是格式化后的输出文本到标准错误，最后通过调用

exit()来终止程序。(有些示例程序提供扩展的 usageErr()函数，命名为 usageError())。

cmdLineErr()函数类似于 usageErr()，但是只用在命令行参数诊断中。

我们的错误诊断函数实现如清单 3-3 所示：

*清单 3-3：所有示例程序使用的错误处理函数*

---------- lib/error_functions.c

```
#include <stdarg.h>
#include "error_functions.h"
#include "tlpi_hdr.h"
#include "ename.c.inc"          /* Defines ename and MAX_ENAME */
#ifdef __GNUC__
__attribute__ ((__noreturn__))
#endif

static void
terminate(Boolean useExit3)
{
    char *s;
    /* Dump core if EF_DUMPCORE environment variable is defined and
       is a nonempty string; otherwise call exit(3) or _exit(2),
       depending on the value of 'useExit3'. */
    s = getenv("EF_DUMPCORE");
    if (s != NULL && *s != '\0')
        abort();
    else if (useExit3)
        exit(EXIT_FAILURE);
    else
        _exit(EXIT_FAILURE);
}

static void
outputError(Boolean useErr, int err, Boolean flushStdout,
        const char *format, va_list ap)
{
#define BUF_SIZE 500
    char buf[BUF_SIZE], userMsg[BUF_SIZE], errText[BUF_SIZE];
    vsnprintf(userMsg, BUF_SIZE, format, ap);
    if (useErr)
        snprintf(errText, BUF_SIZE, " [%s %s]",
```

```
                (err > 0 && err <= MAX_ENAME) ?
                ename[err] : "?UNKNOWN?", strerror(err));
    else
        snprintf(errText, BUF_SIZE, ":");
    snprintf(buf, BUF_SIZE, "ERROR%s %s\n", errText, userMsg);
    if (flushStdout)
        fflush(stdout);       /* Flush any pending stdout */
    fputs(buf, stderr);
    fflush(stderr);           /* In case stderr is not line-buffered */
}

void
errMsg(const char *format, ...)
{
    va_list argList;
    int savedErrno;
    savedErrno = errno;       /* In case we change it here */
    va_start(argList, format);
    outputError(TRUE, errno, TRUE, format, argList);
    va_end(argList);
    errno = savedErrno;
}

void
errExit(const char *format, ...)
{
    va_list argList;
    va_start(argList, format);
    outputError(TRUE, errno, TRUE, format, argList);
    va_end(argList);
    terminate(TRUE);
}

void
err_exit(const char *format, ...)
{
    va_list argList;
    va_start(argList, format);
    outputError(TRUE, errno, FALSE, format, argList);
    va_end(argList);
    terminate(FALSE);
}
```

```
void
errExitEN(int errnum, const char *format, ...)
{
    va_list argList;
    va_start(argList, format);
    outputError(TRUE, errnum, TRUE, format, argList);
    va_end(argList);
    terminate(TRUE);
}

void
fatal(const char *format, ...)
{
    va_list argList;
    va_start(argList, format);
    outputError(FALSE, 0, TRUE, format, argList);
    va_end(argList);
    terminate(TRUE);
}

void
usageErr(const char *format, ...)
{
    va_list argList;
    fflush(stdout);           /* Flush any pending stdout */
    fprintf(stderr, "Usage: ");
    va_start(argList, format);
    vfprintf(stderr, format, argList);
    va_end(argList);
    fflush(stderr);           /* In case stderr is not line-buffered */
    exit(EXIT_FAILURE);
}

void
cmdLineErr(const char *format, ...)
{
    va_list argList;
    fflush(stdout);           /* Flush any pending stdout */
    fprintf(stderr, "Command-line usage error: ");
    va_start(argList, format);
    vfprintf(stderr, format, argList);
```

```
    va_end(argList);
    fflush(stderr);          /* In case stderr is not line-buffered */
    exit(EXIT_FAILURE);
}
```

---------- lib/error_functions.c

enames.c.inc 文件如清单 3-4 所示。这个文件定义了一个字符串数组 ename，是每个可能的 errno 值对应的符号名。我们的错误处理函数使用这个数组来打印出某个错误数值对应的符号名。这是由于 strerror()并不打印错误的符号常量名。打印出符号名让我们可以更加容易地在手册页中找到错误原因。

注意 ename 数组的某些字符串是空的，对应的是那些未使用的错误值。此外有些字符串包含两个名字并用斜线分开，这对应于那些两个符号名拥有相同的错误数值。

*清单 3-4：Linux 错误名（x86-32 版）*

---------- lib/ename.c.inc

```
static char *ename[] = {
    /*   0 */ "",
    /*   1 */ "EPERM", "ENOENT", "ESRCH", "EINTR", "EIO", "ENXIO", "E2BIG",
    /*   8 */ "ENOEXEC", "EBADF", "ECHILD", "EAGAIN/EWOULDBLOCK", "ENOMEM",
    /*  13 */ "EACCES", "EFAULT", "ENOTBLK", "EBUSY", "EEXIST", "EXDEV",
    /*  19 */ "ENODEV", "ENOTDIR", "EISDIR", "EINVAL", "ENFILE", "EMFILE",
    /*  25 */ "ENOTTY", "ETXTBSY", "EFBIG", "ENOSPC", "ESPIPE", "EROFS",
    /*  31 */ "EMLINK", "EPIPE", "EDOM", "ERANGE", "EDEADLK/EDEADLOCK",
    /*  36 */ "ENAMETOOLONG", "ENOLCK", "ENOSYS", "ENOTEMPTY", "ELOOP", "",
    /*  42 */ "ENOMSG", "EIDRM", "ECHRNG", "EL2NSYNC", "EL3HLT", "EL3RST",
    /*  48 */ "ELNRNG", "EUNATCH", "ENOCSI", "EL2HLT", "EBADE", "EBADR",
    /*   54 */ "EXFULL", "ENOANO", "EBADRQC", "EBADSLT", "", "EBFONT",
"ENOSTR",
    /*  61 */ "ENODATA", "ETIME", "ENOSR", "ENONET", "ENOPKG", "EREMOTE",
    /*  67 */ "ENOLINK", "EADV", "ESRMNT", "ECOMM", "EPROTO", "EMULTIHOP",
    /*  73 */ "EDOTDOT", "EBADMSG", "EOVERFLOW", "ENOTUNIQ", "EBADFD",
    /*  78 */ "EREMCHG", "ELIBACC", "ELIBBAD", "ELIBSCN", "ELIBMAX",
    /*  83 */ "ELIBEXEC", "EILSEQ", "ERESTART", "ESTRPIPE", "EUSERS",
    /*  88 */ "ENOTSOCK", "EDESTADDRREQ", "EMSGSIZE", "EPROTOTYPE",
    /*  92 */ "ENOPROTOOPT", "EPROTONOSUPPORT", "ESOCKTNOSUPPORT",
```

```
    /*  95 */ "EOPNOTSUPP/ENOTSUP", "EPFNOSUPPORT", "EAFNOSUPPORT",
    /*  98 */ "EADDRINUSE", "EADDRNOTAVAIL", "ENETDOWN", "ENETUNREACH",
    /*  102  */  "ENETRESET",  "ECONNABORTED",  "ECONNRESET",  "ENOBUFS",
"EISCONN",
    /* 107 */ "ENOTCONN", "ESHUTDOWN", "ETOOMANYREFS", "ETIMEDOUT",
    /* 111 */ "ECONNREFUSED", "EHOSTDOWN", "EHOSTUNREACH", "EALREADY",
    /* 115 */ "EINPROGRESS", "ESTALE", "EUCLEAN", "ENOTNAM", "ENAVAIL",
    /* 120 */ "EISNAM", "EREMOTEIO", "EDQUOT", "ENOMEDIUM", "EMEDIUMTYPE",
    /* 125 */ "ECANCELED", "ENOKEY", "EKEYEXPIRED", "EKEYREVOKED",
    /* 129 */ "EKEYREJECTED", "EOWNERDEAD", "ENOTRECOVERABLE", "ERFKILL"
};

#define MAX_ENAME 132
```

---------- lib/ename.c.inc

## 解析数字命令行参数的函数

清单 3-5 中的头文件提供了两个函数的声明，这两个函数被用来解析命令行参数的整数值：getInt()和 getLong()。使用这些函数而不是 atoi(), atol()和 strtol()的主要优点是它们提供数值参数的基本验证检查功能。

```
#include "tlpi_hdr.h"

int getInt(const char *arg, int flags, const char *name);
long getLong(const char *arg, int flags, const char *name);
                                                    都返回 arg 转化为数值形式
```

getInt()和 getLong()函数把字符串 arg 转化为 int 或 long。如果 arg 不包含合法的整数字符串（只有数字和+-），则这些函数会打印一条错误消息，并终止程序。

如果 name 参数非 NULL，它会包含一个字符串标识 arg 中的参数。这个字符串被包含在这些函数的错误消息显示中。

flags 参数提供 getInt()和 getLong()函数的某些控制功能。默认这些函数都认为字符串包含带符号十进制整数。通过“|”一个或多个清单 3-5 中的 GN_*常量到 flags 中，我们可以选择其它进制，也可以限制数值的范围非负或大于 0。

getInt()和 getLong()函数的实现如清单 3-6。

*清单 3-5：get_num.c 的头文件*

---------- lib/get_num.h

```
#ifndef GET_NUM_H
#define GET_NUM_H

#define GN_NONNEG       01      /* Value must be >= 0 */
#define GN_GT_0         02      /* Value must be > 0 */
                               /* By default, integers are decimal */
#define GN_ANY_BASE   0100      /* Can use any base - like strtol(3) */
#define GN_BASE_8     0200      /* Value is expressed in octal */
#define GN_BASE_16    0400      /* Value is expressed in hexadecimal */

long getLong(const char *arg, int flags, const char *name);
int getInt(const char *arg, int flags, const char *name);

#endif
```

---------- lib/get_num.h

*清单 3-6：命令行数值参数解析函数*

---------- lib/get_num.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
#include <errno.h>
#include "get_num.h"

static void
gnFail(const char *fname, const char *msg, const char *arg, const char *name)
{
    fprintf(stderr, "%s error", fname);
    if (name != NULL)
        fprintf(stderr, " (in %s)", name);
    fprintf(stderr, ": %s\n", msg);
    if (arg != NULL && *arg != '\0')
        fprintf(stderr, "        offending text: %s\n", arg);

    exit(EXIT_FAILURE);
}
```

```
static long
getNum(const char *fname, const char *arg, int flags, const char *name)
{
    long res;
    char *endptr;
    int base;
    if (arg == NULL || *arg == '\0')
        gnFail(fname, "null or empty string", arg, name);
    base = (flags & GN_ANY_BASE) ? 0 : (flags & GN_BASE_8) ? 8 :
                    (flags & GN_BASE_16) ? 16 : 10;
    errno = 0;
    res = strtol(arg, &endptr, base);
    if (errno != 0)
        gnFail(fname, "strtol() failed", arg, name);
    if (*endptr != '\0')
        gnFail(fname, "nonnumeric characters", arg, name);
    if ((flags & GN_NONNEG) && res < 0)
        gnFail(fname, "negative value not allowed", arg, name);
    if ((flags & GN_GT_0) && res <= 0)
        gnFail(fname, "value must be > 0", arg, name);

    return res;
}

long
getLong(const char *arg, int flags, const char *name)
{
    return getNum("getLong", arg, flags, name);
}
int
getInt(const char *arg, int flags, const char *name)
{
    long res;
    res = getNum("getInt", arg, flags, name);
    if (res > INT_MAX || res < INT_MIN)
        gnFail("getInt", "integer out of range", arg, name);

    return (int) res;
}
```

---------- lib/get_num.c

## 3.6 可移植问题

在这一节，我们讨论编写可移植系统程序的主题。介绍 SUSv3 定义的特性测试宏和标准系统数据类型，然后再查看其它一些可移植问题。

### 3.6.1 特性测试宏

许多标准支配着系统调用和库函数 API 的行为（1.3 节）。其中一些标准由标准组织如开放组织（Single UNIX 规范）定义，另一些则由两个历史上重要的 UNIX 实现定义：BSD 和 System V Release 4（以及 System V 接口定义）。

有时候当编写可移植程序时，我们可能希望头文件只暴露特定标准的定义（常量、函数原型等）。我们可以在编译程序时定义一个或多个特性测试宏，来实现这个功能。我们在包含任何其它头文件之前定义这些宏：

```
#define _BSD_SOURCE 1
```

或者也可以使用 C 编译器的-D 选项：

```
$ cc -D_BSD_SOURCE prog.c
```

下面的特性测试宏由相关标准规定，因此在支持这些标准的所有系统中使用它们是可移植的：

_POSIX_SOURCE

如果定义了（无论什么值），暴露 POSIX.1-1990 和 ISO C（1990）标准的定义，这个宏已经被_POSIX_C_SOURCE 取代。

_POSIX_C_SOURCE

如果定义为 1，和_POSIX_SOURCE 的效果相同。如果定义为大于或等于 199309，同时还暴露 POSIX.1b（实时）定义。如果定义为大于或等于 199506，还暴露 POSIX.1c（线程）定义。如果定义为值 200112，同时还暴露 POSIX.1-2001 基础规

范定义（不包括 XSI 扩展）。（glibc 2.3.3 之前的版本，不把 200112 解释为 _POSIX_C_SOURCE）如果定义为值 200809，还暴露 POSIX.1-20008 基础规范（glibc 2.10 版本之前不能解释 200809 值）。

_XOPEN_SOURCE

如果定义（任何值），暴露 POSIX.1, POSIX.2, 和 X/Open（XPG4）定义。如果定义为 500 或更大，还暴露 SUSv2（UNIX98 和 XPG5）扩展。设置为 600 或更高，额外地暴露 SUSv3 XSI（UNIX03）扩展和 C99 扩展。（glibc 2.2 以前无法解释 600 值的_XOPEN_SOURCE）设置为 700 或更高，还同时暴露 SUSv4 XSI 扩展。（glibc 2.10 之前无法解释_XOPEN_SOURCE 为 700）。500，600 和 700 分别对应于 SUSv2，SUSv3 和 SUSv4，分别由于 X/Open 规范的 Issues 5，6 和 7 而得名。

下面这些特性测试宏是 glibc 专有的：

_BSD_SOURCE

如果定义（任何值），暴露 BSD 定义。定义这个宏同时也定义_POSIX_C_SOURCE 的值为 199506。显式地设置这个宏，当标准冲突时，会优先选择 BSD 定义。

_SVID_SOURCE

如果定义（任何值）。暴露 System V 接口定义（SVID）。

_GNU_SOURCE

如果定义（任何值），暴露前面所有宏提供的所有定义，以及许多 GNU 扩展。

当 GNU C 编译器不带特殊选项调用时，默认会定义：_POSIX_SOURCE，_POSIX_C_SOURCE = 200809（glibc 2.5 到 2.9 是 200112，glibc 2.4 及更早是 199506），_BSD_SOURCE，和_SVID_SOURCE。

如果定义了某个宏，或者编译器按标准模式调用（cc -ansi 或 cc -std=c99），

则只提供请求的定义。有一个例外：如果_POSIX_C_SOURCE 没有定义，并且编译器不以标准模式调用，则_POSIX_C_SOURCE 被定义 200809（或者上面所说的更早版本）。

定义多个宏是附加的，因此我们可以使用下面 cc 命令来显式地选择默认的宏设置：

```
$ cc -D_POSIX_SOURCE -D_POSIX_C_SOURCE=199506 \
                -D_BSD_SOURCE -D_SVID_SOURCE prog.c
```

<features.h>头文件和 feature_test_macros(7)手册页提供了更多信息，关于每个特性测试宏精确的数值及含义。

**_POSIX_C_SOURCE、_XOPEN_SOURCE、和 POSIX.1/SUS**

POSIX.1-2001/SUSv3 只规定了_POSIX_C_SOURCE 和_XOPEN_SOURCE 两个特性测试宏，并要求依从标准的应用分别把它们的值定义为 200112 和 600。定义_POSIX_C_SOURCE 为 200112 表示依从 POSIX.1-2001 基础规范（依从 POSIX，但不包括 XSI 扩展）。定义_XOPEN_SOURCE 为 600 则表示依从 SUSv3（XSI 依从，基础规范加 XSI 扩展）。类似的描述也适用于 POSIX.1-2008/SUSv4，后者要求这两个宏的值分别定义为 200809 和 700。

SUSv3 规定设置_XOPEN_SOURCE 为 600 时，同时也应该提供_POSIX_C_SOURCE 设置为 200112 的所有特性。因此应用依从 SUSv3（XSI）只需要设置_XOPEN_SOURCE。SUSv4 也做了类似的规定，设置_XOPEN_SOURCE 为 700 时也提供_POSIX_C_SOURCE 设置为 200809 的所有特性。

**函数原型和示例源代码中的特性测试宏**

手册页描述了要使特定常量定义或函数声明在头文件中可见，必须要设置的特性测试宏。

本书的所有示例源代码都编写成可以使用默认 GNU C 编译器选项或如下选项进行编译：

```
$ cc -std=c99 -D_XOPEN_SOURCE=600
```

本书中的每个函数原型都表示使用默认编译器选项或上面 cc 命令可以编译。手册页提供了每个函数定义需要的特性测试宏的更多精确描述。

## 3.6.2 系统数据类型

许多数据类型的实现都使用标准 C 类型来表示，例如进程 ID、用户 ID、和文件偏移量等。虽然直接使用 C 基本类型（如 int 和 long）来声明变量也可以，但这样做会降低跨 UNIX 系统的可移植性，原因如下：

- 基本类型的大小在不同 UNIX 实现中是不一样的（例如 long 可能是 4 个字节，也可能是 8 字节）。甚至相同 UNIX 系统的不同编译环境下类型大小也不一样。此外不同的实现可能使用不同的类型来表示相同的数据。例如进程 ID 在某个系统可能是 int，但在另一个系统却可能是 long。
- 即使在单一的 UNIX 实现中，不同版本用来表示某个数据的类型也可能不一样。显著的例子是 Linux 中的用户和组 ID，在 Linux 2.2 和更早的版本，这些值以 16 位表示，但在 Linux 2.4 及后面版本，它们却是 32 位的值。

为了避免这样的可移植问题，SUSv3 规定了许多标准系统数据类型，并要求 UNIX 实现正确地定义和使用这些类型。

每个类型都使用 C 语言的 typedef 特性定义。例如 pid_t 数据类型用来表示进程 ID，在 Linux/x86-32 下这个类型如下定义：

```
typedef int pid_t;
```

多数标准系统数据类型的名字都以“_t”结尾，而且多数定义在头文件<sys/types.h>中，少数其它则定义在另外的头文件中。

应用应该使用这些类型定义，来可移植地声明这些变量。例如下面声明允许应用在所有 SUSv3 依从的系统中正确地表示进程 ID：

```
pid_t mypid;
```

表 3-1 列出了我们在本书中会遇到的一些系统数据类型。对于表中某些类型，SUSv3 要求类型实现为“算术”类型。这意味着实现可以选择整数或浮点数（实数或复数）作为底层类型。

*表 3-1：部分系统数据类型*

| 数据类型 | SUSv3 类型要求 | 描述 |
|---|---|---|
| blkcnt_t | 带符号整数 | 文件块计数（15.1 节） |
| blksize_t | 带符号整数 | 文件块大小（15.1 节） |
| cc_t | 无符号整数 | 终端特殊字符（62.4 节） |
| clock_t | 整数或浮点 | 系统时间时钟滴答（10.7 节） |
| clockid_t | 算术类型 | POSIX.1b 时钟和定时器函数的时钟标识符（23.6 节） |
| comp_t | SUSv3 无定义 | 压缩的时钟滴答（28.1 节） |
| dev_t | 算术类型 | 设备数字，包括主和副数字（15.1 节） |
| DIR | 无类型要求 | 目录流（18.8 节） |
| fd_set | 结构体类型 | select()的文件描述符集（63.2.1 节） |
| fsblkcnt_t | 无符号整数 | 文件系统块计数（14.11 节） |
| fsfilcnt_t | 无符号整数 | 文件计数（14.11 节） |
| gid_t | 整数 | 数字的组标识符（8.3 节） |
| id_t | 整数 | 保存标识符的通用类型；至少足够大保存 pid_t, uid_t, gid_t |
| in_addr_t | 32 位无符号整数 | IPv4 地址（59.4 节） |
| in_port_t | 16 位无符号整数 | IP 端口号（59.4 节） |
| ino_t | 无符号整数 | 文件 i-node 数值（15.1 节） |
| key_t | 算术类型 | System V IPC 键（45.2 节） |
| mode_t | 整数 | 文件权限和类型（15.1 节） |
| mqd_t | 无类型要求，但不能 | POSIX 消息队列描述符 |

| | 是数组类型 | |
|---|---|---|
| msglen_t | 无符号整数 | System V 消息队列允许的字节数（46.4 节） |
| msgqnum_t | 无符号整数 | System V 消息队列消息计数（46.4 节） |
| nfds_t | 无符号整数 | poll()的文件描述符数量（63.2.2 节） |
| nlink_t | 整数 | 某个文件的硬链接计数（15.1 节） |
| off_t | 带符号整数 | 文件偏移量或大小（4.7 和 15.1 节） |
| pid_t | 带符号整数 | 进程 ID，进程组 ID，会话 ID（6.2、34.2、34.3 节） |
| ptrdiff_t | 带符号整数 | 两个指针的差 |
| rlim_t | 无符号整数 | 资源限制（36.2 节） |
| sa_family_t | 无符号整数 | socket 地址家族（56.4 节） |
| shmatt_t | 无符号整数 | System V 共享内存段的连接进程计数（48.8 节） |
| sig_atomic_t | 整数 | 可以被原子访问的数据类型（21.1.3 节） |
| siginfo_t | 结构体类型 | 信号来源相关的信息（21.4 节） |
| sigset_t | 整数或结构体类型 | 信号集（20.9 节） |
| size_t | 无符号整数 | 对象的字节大小 |
| socklen_t | 至少 32 位整数类型 | socket 地址结构体的字节大小（56.3 节） |
| speed_t | 无符号整数 | 终端行速度（62.7 节） |
| ssize_t | 带符号整数 | 字节大小或指示错误 |
| stack_t | 结构体类型 | 可选信号堆栈的描述（21.3 节） |
| suseconds_t | 带符号整数，允许范围[-1，1000000] | 微秒的时间间隔（10.1 节） |
| tcflag_t | 无符号整数 | 终端模式标志位掩码（62.2 节） |
| time_t | 整数或浮点 | 日历时间，从 Epoch 开始的秒数（10.1 节） |
| timer_t | 算术类型 | POSIX.1b 时间函数定时器标识符（23.6 节） |
| uid_t | 整数 | 数值的用户标识符（8.1 节） |

我们在后面讨论表 3-1 中的数据类型时，通常会说“[SUSv3 规定]某个类型是整数类型”。这意味着 SUSv3 要求这个类型定义为整数，但并没有要求特定的本地整数类型（如 short, int, long 等）。（通常我们不关心 Linux 到底使用什么本地数据类型来表示标准系统数据类型，因为可移植应用不应该依赖于此）。

**打印系统数据类型的值**

当我们要打印表 3-1 中系统数据类型的数值时（如 pid_t 和 uid_t），我们必须小心不要在 printf()中引入表示依赖。表示依赖的存在是由于 C 参数提升规则，会把 short 值转化为 int，但保持 int 和 long 类型参数不变。这意味着根据不同的系统数据类型，int 或 long 会传递给 printf()调用。但是由于 printf()无法在运行时确定其参数类型，调用方必须使用%d 或%ld 格式说明符来显式提供类型信息。问题是简单地在 printf()中指定说明符会引入表示依赖。通常的解决方案是使用%ld 说明符并且总是把相应的值强制转化为 long，如下：

```
pid_t mypid;

mypid = getpid();    /* Returns process ID of calling process */
printf("My PID is %ld\n", (long) mypid);
```

上面技术有一个例外。由于 off_t 数据类型在某些编译环境下是 long long，我们把 off_t 的值转化为 long long 并且使用%lld 说明符，5.10 节会再加描述。

## 3.6.3 各种可移植问题

在这一节，我们来考虑编写系统程序时，可能遇到的其它一些可移植问题。

**初始化和使用结构体**

每个 UNIX 实现都指定了一组标准结构体，用在许多系统调用和库函数中。例如 sembuf 结构体，它用来描述 semop()系统调用对信号量的操作：

```
struct sembuf {
```

```
    unsigned short sem_num;         /* Semaphore number */
    short          sem_op;          /* Operation to be performed */
    short          sem_flg;         /* Operation flags */
};
```

尽管 SUSv3 规定了结构体（如 sembuf），但意识到下面两点是很重要的：

- 通常并没有规定结构体中域定义的顺序。
- 某些情况下，额外的实现特定域可能会添加到这类结构体中。

因此下面这样使用结构体初始化是不可移植的：

```
struct sembuf s = { 3, -1, SEM_UNDO };
```

虽然这个初始化能够在 Linux 中工作，但其它实现中 sembuf 结构体的域定义顺序可能不同，这段代码将无法工作。要可移植地初始化结构体，我们必须显式地使用赋值语句，如下所示：

```
struct sembuf s;
s.sem_num = 3;
s.sem_op  = -1;
s.sem_flg = SEM_UNDO;
```

如果你使用 C99，那么可以采用结构体初始化的新语法来编写等价的初始化：

```
struct sembuf s = { .sem_num = 3, .sem_op = -1, .sem_flg = SEM_UNDO };
```

如果我们要把结构体的内容写到文件中，也需要考虑成员定义的顺序。为了实现可移植，我们不能简单地执行二进制写入。相反必须按特定顺序把结构体的域一个一个地写入到文件。

### 使用不是所有实现都提供的宏

某些情况下，可能不是所有 UNIX 实现都定义了某个宏。例如 WCOREDUMP() 宏（检查子进程是否产生了 core dump 文件）被广泛实现，但 SUSv3 却没有对其定义。因此这个宏可能在某些 UNIX 实现中不可用。为了处理这种可能性，我们

可以使用 C 预处理器的#ifdef 指令，如下面例子：

```
#ifdef WCOREDUMP
    /* Use WCOREDUMP() macro */
#endif
```

**跨实现所需头文件的差异**

在某些情况下，许多系统调用和库函数需要的头文件在不同 UNIX 实现上是不一样的。在本书中，我们会明确显示 Linux 和 SUSv3 的头文件要求。

本书的某些函数会包含一个特殊的头文件，伴随着注释/* For portability */。这表示头文件不是 Linux 或 SUSv3 所要求，而是由于某些其它（特别是老的系统）实现要求包含它，为了可移植我们需要包含它。

## 3.7 小结

系统调用允许进程向内核请求服务。相比用户空间的函数调用，即使是最简单的系统调用也需要大量的开销，因为系统必须临时切换到内核模式来执行系统调用，内核必须验证系统调用参数，并在用户内存和内核内存间传输数据。

标准 C 库提供许多函数，完成各种各样的任务。某些库函数采用系统调用来完成自己的工作；其它则完全在用户空间中执行任务。在 Linux 中，常用的标准 C 库实现是 glibc。

多数系统调用和库函数返回一个状态来表示调用是否成功。我们总是应该检查这个返回状态。

我们介绍了一组本书中使用的示例函数，这些函数执行的任务包括诊断错误和解析命令行参数。

我们讨论了许多能够帮助我们编写可移植系统程序的准则和技术。

当编译应用时，我们可以定义许多特性测试宏，来控制头文件暴露的定义。如果我们想确保程序依从于某个特定标准，特性测试宏就非常有用。

我们可以使用许多标准定义的系统数据类型，而不是系统本地 C 类型，来提高系统程序的可移植性。SUSv3 规定了许多系统数据类型，各个 UNIX 实现都应该支持，应用程序也应该采用这些系统数据类型。

## 3.8 习题

3-1. 当使用 Linux 特定的 reboot()系统调用来重启系统时，第二个参数 magic2 必须指定为某个魔法数字（例如 LINUX_REBOOT_MAGIC2）。这些数字的意义是什么？（把它们转换为十六进制会提供线索）。

# 第 4 章 文件 I/O：通用 I/O 模型

现在我们开始学习系统调用 API 最重要的部分之一。文件是很好的学习起点，因为它们是 UNIX 哲学的核心。本章关注于执行文件输入和输出的系统调用。

我们首先介绍文件描述符的概念，然后介绍构成通用 I/O 模型的系统调用。这些系统调用包括打开和关闭文件、读取和写入数据。

目前我们只关注于磁盘文件 I/O。但是本章讲解的多数材料都可以应用于后续章节，因为相同的系统调用被用来执行所有文件类型的 I/O 操作，例如管道和终端。

第 5 章扩展了本章的讨论，提供更多文件 I/O 的细节。另外一个文件 I/O 的重要方面（缓冲），也值得拥有自己的一章，第 13 章讲解了内核和 stdio 库的缓冲机制。

## 4.1 概述

所有执行 I/O 的系统调用都需要使用文件描述符，后者的类型是非负整数（通常很小）。文件描述符用来引用所有类型的打开文件，包括管道、FIFO、socket、终端、设备、和普通文件。每个进程都有自己的一组文件描述符。

通常多数应用程序都希望能够使用表 4-1 所列的三个标准文件描述符。这三个描述符由 shell 在程序启动前自动打开；更精确地说，是程序继承了 shell 的文件描述符，而 shell 保持这三个文件描述符总是打开（在交互式 shell 中，这三个文件描述符通常指向 shell 正在运行的终端）。如果命令行指定了 I/O 重定向，则 shell 确保在程序启动之前相应地修改文件描述符。

*表 4-1：标准文件描述符*

| 文件描述符 | 用途 | POSIX 名字 | stdio 流 |
|---|---|---|---|
| 0 | 标准输入 | STDIN_FILENO | stdin |
| 1 | 标准输出 | STDOUT_FILENO | stdout |

| | | | |
|---|---|---|---|
| 2 | 标准错误 | STDERR_FILENO | stderr |

在程序中要引用这些文件描述符，可以直接使用数值（0，1，2），更好的做法是使用<unistd.h>定义的 POSIX 标准名字。

> 尽管变量 stdin, stdout, stderr 最初引用进程的标准输入、标准输出、标准错误，但是可以使用 freopen()库函数把它们改为引用任何文件。freopen()在执行操作的同时，可能还会修改底层打开文件流的文件描述符。换句话说，比如对 stdout 执行 freopen()之后，再假设底层的文件描述符还是 1 就不再是安全的了。

下面是执行文件 I/O 的四个关键系统调用（编程语言和软件包通常间接地通过 I/O 库来使用这些系统调用）：

- `fd = open(pathname, flags, mode)` 打开 pathname 指定的文件，它返回一个文件描述符，在随后的调用中引用这个已经打开的文件。如果文件不存在，open()可以创建它，根据 flags 掩码参数的设置而定。flags 参数还指定了文件是打开读取、写入、还是二者均有。如果创建了一个新文件，mode 参数指定该文件的权限。如果 open()调用不创建文件，则忽略 mode 参数，可以省略。
- `numread = read(fd, buffer, count)` 最多从 fd 指向的文件中读取 count 字节，并存储读取的数据到 buffer。read()调用返回实际读取的字节数。如果没有更多的字节可以读取（如遇到文件尾），read()返回 0。
- `numwrite = write(fd, buffer, count)` 从 buffer 缓冲区中写入 count 字节到 fd 指向的文件。write()调用返回实际写入的字节数，可能小于 count。
- `status = close(fd)` 在所有 I/O 操作结束之后调用，可以释放 fd 文件描述符以及相关的内核资源。

在我们深入这些系统调用的细节之前，先提供清单 4-1 中一个简短的演示。这个程序是 cp 命令的简单版本。它复制第一个参数指向的现有文件到第二个参数指向的新文件。

我们可以如下使用清单 4-1 的程序：

```
$ ./copy oldfile newfile
```

*清单 4-1：使用 I/O 系统调用*

```
---------------------------------------------------------- fileio/copy.c
#include <sys/stat.h>
#include <fcntl.h>
#include "tlpi_hdr.h"

#ifndef BUF_SIZE /* Allow "cc -D" to override definition */
#define BUF_SIZE 1024
#endif

int
main(int argc, char *argv[])
{
    int inputFd, outputFd, openFlags;
    mode_t filePerms;
    ssize_t numRead;
    char buf[BUF_SIZE];

    if (argc != 3 || strcmp(argv[1], "--help") == 0)
        usageErr("%s old-file new-file\n", argv[0]);

    /* Open input and output files */
    inputFd = open(argv[1], O_RDONLY);
    if (inputFd == -1)
        errExit("opening file %s", argv[1]);

    openFlags = O_CREAT | O_WRONLY | O_TRUNC;
    filePerms = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP |
            S_IROTH | S_IWOTH; /* rw-rw-rw- */
    outputFd = open(argv[2], openFlags, filePerms);
    if (outputFd == -1)
        errExit("opening file %s", argv[2]);

    /* Transfer data until we encounter end of input or an error */
    while ((numRead = read(inputFd, buf, BUF_SIZE)) > 0)
    if (write(outputFd, buf, numRead) != numRead)
        fatal("couldn't write whole buffer");
    if (numRead == -1)
        errExit("read");
```

```
    if (close(inputFd) == -1)
        errExit("close input");
    if (close(outputFd) == -1)
        errExit("close output");

    exit(EXIT_SUCCESS);
}
----------------------------------------------------------  fileio/copy.c
```

## 4.2 I/O 的通用性

UNIX I/O 模型的显著特性之一就是通用 I/O 的概念。这意味着相同的四个系统调用——open(), read(), write(), close()——用来执行所有文件类型的 I/O 操作，包括终端设备等。因此如果使用这些系统调用编写了一个程序，那这个程序对任何文件都是可以工作的。例如，下面都是清单 4-1 程序的合理用法：

```
$ ./copy test test.old          复制普通文件
$ ./copy a.txt /dev/tty         复制普通文件到终端
$ ./copy /dev/tty b.txt         复制终端输入到普通文件
$ ./copy /dev/pts/16 /dev/tty   复制另一个终端的输入
```

UNIX 确保每种文件系统和设备驱动都实现了相同的 I/O 系统调用，来实现 I/O 的通用性。由于文件系统和设备的特定细节完全由内核处理，我们编写应用程序时可以忽略设备相关的因素。当需要访问文件系统或设备的特定特性时，可以使用 ioctl()系统调用（4.8 节），提供访问通用 I/O 模型之外特性的接口。

## 4.3 打开文件：open()

open()系统调用要么打开现有文件，要么打开一个新文件。

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

int open(const char *pathname, int flags, ... /* mode_t mode */);
                                        成功时返回文件描述符，失败返回-1
```

要打开的文件由 pathname 参数指定。如果 pathname 是一个符号链表，那么首先会解引用（找到实际的文件）。成功时 open()返回一个文件描述符，用来后续调用引用这个文件；如果发生错误，open()返回-1 并设置 errno 为合适的值。

flags 参数是一个位掩码，指定文件的访问模式，可以选择使用表 4-2 中的某个常量定义：

> 早期 UNIX 实现使用数字 0，1，2 而不是表 4-2 中显示的名字。多数现代 UNIX 实现都定义这些常量为这些值。因此我们可以认为 O_RDWR 不同于 O_RDONLY | O_WRONLY，后者的组合是一个逻辑错误。

*表 4-2：文件访问模式*

| 访问模式 | 描述 |
|---|---|
| O_RDONLY | 打开文件只能读取 |
| O_WRONLY | 打开文件只能写入 |
| O_RDWR | 打开文件同时读取和写入 |

当 open()用来创建新文件时，mode 位掩码参数指定文件的权限。（mode_t 数据类型在 SUSv3 中定义为整数类型）。如果 open()调用不指定 O_CREAT，mode 参数可以忽略。

我们会在后面 15.4 节详细地讨论文件权限，在那里我们会知道新文件的权限不仅仅依赖于 mode 参数，还同时依赖于进程的 umask（15.4.6 节），和父目录的默认访问控制列表（17.6 节）。到那里，我们还会说明 mode 参数可以指定为数值（通常是八进制），或者使用表 15-4 所列常量掩码的“位或”。

清单 4-2 显示了使用 open()的一个例子，使用了前面描述的 flags 位掩码：

*清单 4-2：open()的使用例子*

```
----------------------------------------------------------------------
    /* Open existing file for reading */
    fd = open("startup", O_RDONLY);
    if (fd == -1)
```

```
        errExit("open");

    /* Open new or existing file for reading and writing, truncating to zero
       bytes; file permissions read+write for owner, nothing for all others
    */
    fd = open("myfile", O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
    if (fd == -1)
        errExit("open");

    /* Open new or existing file for writing; writes should always
        append to end of file */
    fd = open("w.log", O_WRONLY | O_CREAT | O_TRUNC | O_APPEND,
            S_IRUSR | S_IWUSR);
    if (fd == -1)
        errExit("open");
-------------------------------------------------------------------------
```

**open()返回的文件描述符数值**

SUSv3 规定如果 open()成功，要确保使用进程未用文件描述符中最小数值的那个。我们可以使用这个特性来确保文件以某个特定的文件描述符打开。例如下面代码确保文件使用标准输入（文件描述符 0）来打开：

```
    if (close(STDIN_FILENO) == -1) /* Close file descriptor 0 */
        errExit("close");

    fd = open(pathname, O_RDONLY);
    if (fd == -1)
        errExit("open");
```

由于文件描述符 0 没有被使用，open()确保使用这个描述符来打开文件。在 5.5 节，我们会使用 dup2()和 fcntl()来达到类似的结果，但对所用文件描述符拥有更加灵活的控制。在那一节里，我们还展示了一个例子，演示控制打开文件使用的文件描述符有什么用处。

### 4.3.1 open()的 flags 参数

在清单 4-2 某些 open()的使用中，我们包含了文件访问模式之外的 flags 标志（O_CREAT, O_TRUNC, O_APPEND）。下面我们来详细讨论 flags 参数。表 4-3 总结了 flags 可以“或”的所有常量定义，最后一列表示这些常量是否 SUSv3 或 SUSv4 标准定义。

*表 4-3：open()调用 flags 参数的值*

| 标志 | 作用 | SUS? |
|---|---|---|
| O_RDONLY | 打开为只能读取 | v3 |
| O_WRONLY | 打开为只能写入 | v3 |
| O_RDWR | 打开为同时读写 | v3 |
| O_CLOEXEC | 设置“close-on-exec”标志（Linux 2.6.23 开始） | v4 |
| O_CREAT | 如果文件不存在则创建 | v3 |
| O_DIRECT | 文件 I/O 绕过缓冲区缓存 | |
| O_DIRECTORY | 如果 pathname 不是目录则失败 | v4 |
| O_EXCL | 和 O_CREAT 一起使用：创建唯一的文件 | v3 |
| O_LARGEFILE | 在 32 位系统中打开大文件 | |
| O_NOATIME | read()时不更新文件的最后访问时间（Linux 2.6.8 开始） | |
| O_NOCTTY | 不让 pathname 成为控制终端 | v3 |
| O_NOFOLLOW | 不解引用符号链接 | v4 |
| O_TRUNC | 截断现有文件为 0 长度 | v3 |
| O_APPEND | 写入总是追加至文件末尾 | v3 |
| O_ASYNC | I/O 可操作时产生一个信号 | |
| O_DSYNC | 提供同步的 I/O 数据完整性（Linux 2.6.33 开始） | v3 |
| O_NONBLOCK | 非阻塞模式打开 | v3 |
| O_SYNC | 使文件写入同步 | v3 |

表 4-3 中的常量可以分为以下几组：

- **文件访问模式标志**：包括前面讨论过的 O_RDONLY, O_WRONLY, 和 O_RDWR。可以使用 fcntl()的 F_GETFL 操作获得设置的值（5.3 节）。
- **文件创建标志**：这些标志显示在表 4-3 的第二部分。它们控制 open()调用的各种行为，以及控制随后进行的 I/O 操作的行为。这些标志不能修改也无法取得设置的值。
- **打开文件状态标志**：这些标志是表 4-3 最后那部分。它们可以使用 fcntl()的 F_GETFL 和 F_SETFL 操作来获得和修改（5.3 节）。这些标志有时候也简单地称为文件状态标志。

> 从内核 2.6.22 开始，目录/proc/PID/fdinfo 下的 Linux 特定文件可以被读取，来获取系统中任何进程的文件描述符信息。进程的每个打开文件描述符在这个目录下都有一个文件，文件名就是描述符的数值。该文件中的 pos 域显示了当前文件偏移（4.7 节）。flags 域是一个八进制数字，显示了文件访问模式标志和打开文件状态标志。（要解码这个数字，我们需要查看 C 库头文件中这些标志的数值）。

flags 常量的细节如下：

**O_APPEND**

写入总是追加到文件末尾。我们在 5.1 节讨论这个标志的重要性。

**O_ASYNC**

当文件描述符的 I/O 变得可操作时产生一个信号。这个特性叫做“信号驱动 I/O”，只在某些文件类型中可用，例如终端、FIFO、和 socket。（SUSv3 没有规定 O_ASYNC 标志，但多数 UNIX 实现都提供该标志或老的 FASYNC 标志）。在 Linux 中，调用 open()时指定 O_ASYNC 标志没有效果。要启用信号驱动 I/O，我们必须使用 fcntl()的 F_SETFL 操作来设置该标志（5.3 节）。（其它某些 UNIX 实现也是一样）。更多 O_ASYNC 标志的信息请参考 63.3 节。

**O_CLOEXEC（Linux 2.6.23 开始）**

为新打开的文件描述符启用 close-on-exec 标志（FD_CLOEXEC）。我们在 27.4 节描述 FD_CLOEXEC 标志。使用 O_CLOEXEC 标志允许程序避免使用额外的 fcntl()的 F_SETFD 操作来设置 close-on-exec 标志。也是多线程程序避免竞争

条件的重要手段。当一个线程打开文件描述符，然后试图设置 close-on-exec 标志，这时另一个线程调用 fork()然后再调用 exec()，就会出现竞争条件。（假设第二个线程调用 fork()和 exec()的时间，正好在第一个线程打开文件描述符和使用 fcntl()设置 close-on-exec 标志之间）。这种竞争条件会导致打开的文件描述符被无意地传递给不安全的程序。（我们会在 5.1 节更详细地讨论竞争条件）。

**O_CREAT**

如果文件不存在，就创建一个新的空白文件。即使文件以只读模式打开，这个标志也是有效的。如果我们指定 O_CREAT，就必须为 open()调用提供 mode 参数；否则新文件的权限会被设置为堆栈中的随机值。

**O_DIRECT**

允许文件 I/O 绕过缓冲区缓存。13.6 节详细描述了这个特性。必须设置_GNU_SOURCE 特性测试宏，才能在<fcntl.h>中启用这个常量的定义。

**O_DIRECTORY**

如果 pathname 不是目录则返回错误（errno 等于 ENOTDIR）。这个标志是专门为实现 opendir()（18.8 节）而特别设计的。必须定义_GNU_SOURCE 特性测试宏，才能在<fcntl.h>中启用这个常量定义。

**O_DSYNC（Linux 2.6.33 开始）**

按照同步 I/O 数据完整性的要求来执行文件写入。参考 13.3 节关于内核 I/O 缓冲的讨论。

**O_EXCL**

这个标志结合 O_CREAT 一起使用，表示如果文件已经存在，就不应该打开它；相反 open()应该失败，并设置 errno 为 EEXIST。换句话说，这个标志允许调用者确保由本进程创建该文件。检查是否已经存在和创建该文件的操作是“原子的”。我们会在 5.1 节讨论“原子”的概念。当 O_CREAT 和 O_EXCL 同时指定时，如果 pathname 是一个符号链接，则 open()也会失败（errno 为 EEXIST），SUSv3 规定了这个行为，这样特权应用可以在已知的位置创建文件，杜绝由于符号链接导致文件被创建在不同的位置（例如系统目录），

从而解决了这个安全问题。

**O_LARGEFILE**

支持打开大文件，这个标志用在 32 位系统操作大文件。尽管 SUSv3 并没有规定这个标志，其它一些 UNIX 实现也支持 O_LARGEFILE 标志。在 64 位 Linux 实现中（如 Alpha 和 IA-64），这个标志没有作用。更多信息请参考 5.10 节。

**O_NOATIME（Linux 2.6.8 开始）**

读取文件时不更新文件的最后访问时间（15.1 节描述的 st_atime 域）。要使用这个标志，调用进程的有效用户 ID 必须匹配文件拥有者，或者进程拥有特权（CAP_FOWNER）；否则 open()会以 EPERM 错误失败。（实际上，当使用 O_NOATIME 标志打开文件时，要匹配文件用户 ID 的并不是进程有效用户 ID，而是进程的文件系统用户 ID，9.5 节会详细描述）。O_NOATIME 标志是非标准的 Linux 扩展，要在<fcntl.h>中暴露它的定义，必须定义_GNU_SOURCE 特性测试宏。O_NOATIME 标志主要用于索引和备份程序，能够极大地降低磁盘活动，因为重复的磁盘来回 seek，不需要读取文件内容和更新文件 i-node 的最后访问时间（14.4 节）。使用 mount()的 MS_NOATIME 标志（14.8.1 节）和 FS_NOATIME 标志（15.5 节）也可以达到类似 O_NOATIME 的功能。

**O_NOCTTY**

如果被打开的文件是终端设备，则阻止它成为控制终端。34.4 节详细讨论控制终端。如果被打开的文件不是终端，则这个标志没有作用。

**O_NOFOLLOW**

通常如果 pathname 是一个符号链接，open()会自动进行解引用。但是如果指定了 O_NOFOLLOW 标志，则 pathname 是符号链接时 open()将失败（errno 设为 ELOOP）。这个标志非常有用，特别是特权程序，可以确保 open()不会解引用符号链接。要从<fcntl.h>中暴露这个标志的常量定义，必须定义_GNU_SOURCE 特性测试宏。

**O_NONBLOCK**

以非阻塞模式打开文件，请参考 5.9 节。

**O_SYNC**

以同步 I/O 模式打开文件。参考 13.3 节内核 I/O 缓冲的相关讨论。

**O_TRUNC**

如果文件已经存在并且是普通文件，则将其截断为 0 长度，并销毁所有数据。在 Linux 中，无论打开文件为读取还是写入，都会对文件进行截断（两种情况下，都必须拥有文件的写权限）。SUSv3 没有对 O_RDONLY 和 O_TRUNC 标志的组合做出规定，但多数 UNIX 实现的行为都和 Linux 一样。

## 4.3.2 open()的错误

如果打开文件时出现了错误，open()返回-1，并设置 errno 为错误原因。以下是可能出现的错误（上面描述 flags 参数时已经提到一些错误了）：

EACCES

文件的权限不允许调用进程以 flags 指定的模式打开文件，或者由于目录权限，文件不能被访问；或者文件不存在并且无法被创建。

EISDIR

指定的文件是目录，调用方试图以写入模式打开它，这个操作是不允许的。（换句话说，打开目录来读取有时候是有用的。我们在 18.11 节会提供相关的例子）。

EMFILE

进程达到允许打开文件描述符数量的资源限制（RLIMIT_NOFILE，36.3 节描述）。

ENFILE

已经达到系统级的打开文件描述符数量限制。

ENOENT

指定的文件不存在，又没有指定 O_CREAT 标志；或者指定了 O_CREAT，但 pathname 中的某个目录不存在；或者 pathname 是符号链接，指向不存在的路径（摇摆链接）。

EROFS

指定的文件在只读文件系统中，调用方却试图打开来写入。

ETXTBSY

指定的文件是可执行文件（程序），当前正在执行中。不允许修改（打开写入）正在运行程序关联的可执行文件。（我们必须首先终止这个程序，然后才能修改可执行文件）。

当我们后面讨论其它系统调用和库函数时，我们通常不会像上面这样列举出所有可能的错误（这个错误列表可以在系统调用和库函数相应的手册页找到）。在这里列出来有两个理由：首先，open()是本书详细讲解的第一个系统调用，上面列表说明系统调用或库函数可能由于各种原因失败；其次，open()为什么会失败本身也是非常有趣的，我们访问文件时需要考虑和检查各种情况。（上面列表是不完整的，更多失败原因请参考 open(2)手册页）。

## 4.3.3 creat()系统调用

在早期 UNIX 实现中，open()只有两个参数，而且不能用来创建新文件。使用 creat()系统调用来创建和打开新文件。

```
#include <fcntl.h>

int creat(const char *pathname, mode_t mode);
                                                    返回文件描述符；出错返回-1
```

creat()系统调用可以创建和打开 pathname 指定的新文件，或者如果文件已存在，则打开它并截断为 0 长度。creat()返回一个文件描述符，供后续系统调用使用。调用 creat()等价于下面 open()调用：

```
fd = open(pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);
```

由于 open()的 flags 参数提供了打开文件的更多控制（如指定 O_RDWR 而不是 O_WRONLY），目前 creat()已经废弃，不过在老程序中还可以见到它。

# 4.4 读取文件：read()

read()系统调用从描述符 fd 引用的文件中读取数据：

```
#include <unistd.h>

ssize_t read(int fd, void *buffer, size_t count);
                                    返回读取的字节数，EOF 返回 0，错误时返回-1
```

count 参数指定读取的最大字节数（size_t 数据类型是无符号整型）。buffer 参数指定内存缓冲区的地址，用来存放读取的数据。这个缓冲区至少必须有 count 字节大小。

> 系统调用并不为调用方分配缓冲区内存来返回信息。相反我们必须自己分配合适大小的内存缓冲区，并传递指针给系统调用。有些库函数则确实会分配内存缓冲区，用来返回信息给调用方。

成功调用 read()返回实际读取的字节数，如果遇到“end-of-file”则返回 0，错误时返回-1。ssize_t 数据类型是带符号整数类型，用来返回字节数，或错误时返回-1。

调用 read()可能读取少于请求的字节数。对于普通文件来说，可能的原因是我们已经接近文件末尾，只能读取剩余的字节。

当 read()应用于其它文件类型时（如管道、FIFO、socket、终端），也存在许多读取少于请求字节数的情况。例如默认情况下，read()从终端读取字符最多只到换行字符（\n）。我们在随后章节讲解其它文件类型时还会仔细考虑这些情况。

使用 read()从终端获取一系列输入字符，我们可以使用如下代码：

```
#define MAX_READ 20
char buffer[MAX_READ];

if (read(STDIN_FILENO, buffer, MAX_READ) == -1)
    errExit("read");

printf("The input data was: %s\n", buffer);
```

这段代码的输出可能会很奇怪，包含了实际输入字符串之外的许多字符。这是因为 read()不会在字符串的末尾自动增加 printf()函数需要的 null 终止字符。稍

微思考一下，我们就能明白 read 必须这么做，因为 read()可能用于任何文件读取任何数据。有时候输入可能是文本，其它情况下输入可能是二进制整数或 C 结构体等二进制形式。read()没有办法进行区分，因此不能简单地使用 C 的 null 终止字符惯例。如果输入缓冲区末尾要求有 null 终止字符，我们必须显式地手工增加。

```
char buffer[MAX_READ + 1];
ssize_t numRead;

numRead = read(STDIN_FILENO, buffer, MAX_READ);
if (numRead == -1)
    errExit("read");

buffer[numRead] = '\0';
printf("The input data was: %s\n", buffer);
```

由于 null 终止字符要求额外的一个字节内存，buffer 的大小必须至少是我们需要读取最大字符的长度加 1。

## 4.5 写入文件：write()

write()系统调用向打开文件写入数据。

```
#include <unistd.h>

ssize_t write(int fd, void *buffer, size_t count);
                                        返回实际写入的字节数，出错时返回-1
```

write()的参数类似于 read()：buffer 是待写入数据的地址；count 是 buffer 的字节数；fd 指向数据需要写入的文件。

成功时 write()返回实际写入的字节数，可能小于 count。对于磁盘文件来说，这种“部分写入”的可能原因是磁盘已满，或进程达到文件大小的资源限制。（36.3 节描述的 RLIMIT_FSIZE 限制）。

## 4.6 关闭文件：close()

close()系统调用关闭一个已打开的文件描述符，释放该文件描述符以供进程随后使用。当进程终止时，所有打开的文件描述符都会自动被关闭。

```
#include <unistd.h>

int close(int fd);
                                                成功时返回 0；失败返回-1
```

通常手工显式地关闭文件描述符是良好的实践，因为这样做可以提高代码应对以后修改的可读性和可靠性。此外文件描述符是消耗性资源，如果不关闭文件描述符，可能导致进程用完所有描述符。当编写长期运行需要处理许多文件的程序时（如 shell 或网络服务器），这就是一个特别重要的问题。

和其它系统调用一样，调用 close()也应该检查相关的错误，如下所示：

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

这样可以捕获到诸如关闭未打开文件描述符，两次关闭相同的文件描述符等错误。还能捕获到特定文件系统在关闭操作时诊断出来的错误条件。

> NFS（网络文件系统）会提供文件系统特定的错误。如果发生了 NFS 提交错误，意味着数据没有到达远程磁盘，这个错误就会通过 close()调用传送给应用。

## 4.7 改变文件偏移：lseek()

对于每个打开的文件，内核记录了一个文件偏移量，有时候也称为读写偏移或指针。这个偏移是下一个 read()或 write()将操作的文件位置。文件偏移是相对于文件起始的顺序字节位置，文件的第一个字节即是偏移 0。

当文件被打开时，文件偏移设置为文件的开头，并随着 read()和 write()调用自动调整为刚刚读取或写入的下一个字节。因此连续的 read()和 write()调用可以顺序地穿过一个文件。

lseek()系统调用可以调整文件描述符 fd 引用文件的偏移，设置为 offset 和

whence 指定的位置。

```
#include <unistd.h>

off_t lseek(int fd, off_t offset, int whence);
                                成功时返回新的文件偏移，出错返回-1
```

offset 参数指定字节数（SUSv3 规定 off_t 数据类型为带符号整数类型）。whence 参数指示 offset 的基点，可以取如下值之一：

SEEK_SET

文件偏移设为文件起始位置后的 offset 字节。

SEEK_CUR

文件偏移调整为相对当前文件偏移的 offset 字节。

SEEK_END

文件偏移设为文件大小加 offset。换句话说，offset 相对于文件末尾来解释。

图 4-1 显示了 whence 参数是如何被解释的。

在早期 UNIX 实现中，使用了 0，1，2 整数，而不是 SEEK*常量。老版本的 BSD 使用不同的名字：L_SET, L_INCR, L_XTND。

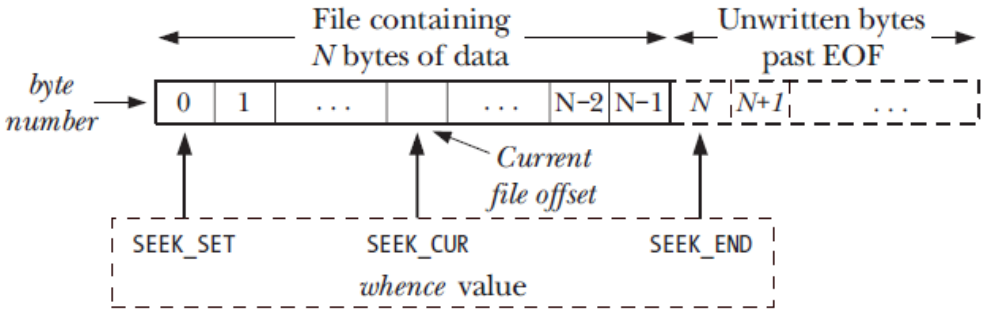


*图 4-1：解释 lseek()的 whence 参数*

如果 whence 是 SEEK_CUR 或 SEEK_END，offset 可以为正也可以为负；如果是 SEEK_SET，则 offset 必须非负。

成功调用 lseek()返回新的文件偏移。下面调用获取当前文件偏移，而不修改

文件偏移：

```
curr = lseek(fd, 0, SEEK_CUR);
```

有一些 UNIX 实现（不是 Linux）有非标准的 tell(fd)函数，作用和上面讲的 lseek 调用是一样的。

下面是 lseek()调用的一些例子，后面的注释描述了新的文件偏移：

```
lseek(fd, 0, SEEK_SET);      /* 文件起始位置 */
lseek(fd, 0, SEEK_END);      /* 文件末尾的下一个字节 */
lseek(fd, -1, SEEK_END);     /* 文件末尾 */
lseek(fd, -10, SEEK_CUR);    /* 当前位置的前 10 字节处 */
lseek(fd, 10000, SEEK_END); /* 文件末尾之后的 10001 字节 */
```

调用 lseek()只是简单地调整内核中文件描述符相关联的文件记录偏移值，不会引起任何物理设备访问操作。

我们在后面 5.4 节会更加详细地讨论文件偏移、文件描述符、和打开文件之间的关系。

不是所有文件类型都能使用 lseek()，对管道、FIFO、socket、或终端应用 lseek()是不允许的；lseek()会失败，并设置 errno 为 ESPIPE。反过来对某些可调整偏移的设备应用 lseek()是可以的，例如磁盘或磁带设备都可以 seek 到指定位置。

lseek()中的“l”表示最早 offset 参数和返回值都是 long。早期 UNIX 实现还提供一个 seek()系统调用，相应地使用 int 参数和返回值。

### 文件空洞

如果一个程序 seek 超过了文件末尾，然后又执行 I/O，会发生什么呢？这时候 read()会返回 0，表示 end-of-file。但是令人吃惊的是，超过文件末尾任意位置写入字节是可以的。

之前的文件末尾与新写入字节之间的空间被称为“文件空洞”。从编程的角度来看，空洞中是存在字节的，从空洞读取会返回全 0（null 字节）的数据。

不过文件空洞不会占用任何磁盘空间。文件系统不会为空洞分配任何磁盘块，除非之后有数据写入到空洞中。文件空洞的主要优点是稀疏文件会消耗更少的磁

盘空间，不需要为 null 字节分配实际的磁盘块。Core dump 文件（22.1 节）是包含大量空洞的典型例子。

> 文件空洞不消耗磁盘空间的说法需要稍微澄清。在多数文件系统中，文件空间是以块的单位来分配的（14.3 节）。块大小依赖于文件系统，不过通常 1024，2048 或 4096 字节。如果空洞存在于块中，而不是块的边界，那还是会分配一个完整的块来存储数据，对应于空洞的部分空间则填充 null 字节。

多数 UNIX 文件系统支持文件空洞的概念，但许多非 UNIX 本地文件系统（如 Microsoft VFAT）却不支持，在这些不支持文件空洞的文件系统中，显式的 null 字节会被写入到文件。

空洞的存在意味着文件的正常大小可能会大于它占用的磁盘存储空间（某些情况下会大很多）。向文件空洞写入字节会减少磁盘的可用空间，因为内核会分配块来存储这些数据，而文件的大小则不会改变。这种情况并不常见，但无论如何你需要知道有这么个问题。

> SUSv3 规定了一个函数 posix_fallocate(fd, offset, len)，可以确保在磁盘中为描述符 fd 引用的文件分配 offset 和 len 指定的空间。这允许应用确保之后的 write()操作不会因为磁盘空间耗尽而失败（如果有空洞这种情况可能会发生）。历史上 glibc 对这个函数的实现是在每个块中写入 0 字节。从内核 2.6.23 开始，Linux 提供了一个 fallocate()系统调用，可以更高效地确保分配必需的空间，新的 glibc posix_fallocate()实现就使用了这个系统调用。

14.4 节描述空洞在文件中是如何表示的，15.1 节描述 stat()系统调用，可以告诉我们文件的当前大小，以及为文件实际分配的块数量。

## 示例程序

清单 4-3 演示了使用 lseek()以及 read()和 write()的一个程序。第一个命令行参数是要打开的文件名，剩下的参数指定要对文件进行的 I/O 操作。每个操作都是一个字母以及相关的值（没有空格）：

- s(offset)：从文件起始 seek offset 字节。
- r(length)：从文件的当前偏移读取 length 字节，并显示为文本。
- R(length)：从文件的当前偏移读取 length 字节，并显示为十六进制。
- w(str)：写入 str 字符串到文件的当前偏移。

*清单 4-3：read(), write(), lseek()演示*

----------fileio/seek_io.c

```
#include <sys/stat.h>
#include <fcntl.h>
#include <ctype.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    size_t len;
    off_t offset;
    int fd, ap, j;
    char *buf;
    ssize_t numRead, numWritten;

    if (argc < 3 || strcmp(argv[1], "--help") == 0)
        usageErr(
            "%s file {r<length>|R<length>|w<string>|s<offset>}...\n",
            argv[0]);

    fd = open(argv[1], O_RDWR | O_CREAT,
        S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP |
        S_IROTH | S_IWOTH); /* rw-rw-rw- */
    if (fd == -1)
        errExit("open");

    for (ap = 2; ap < argc; ap++) {
        switch (argv[ap][0]) {

        case 'r': /* Display bytes at current offset, as text */
        case 'R': /* Display bytes at current offset, in hex */
            len = getLong(&argv[ap][1], GN_ANY_BASE, argv[ap]);
            buf = malloc(len);
            if (buf == NULL)
                errExit("malloc");

            numRead = read(fd, buf, len);
            if (numRead == -1)
                errExit("read");
```

```
            if (numRead == 0) {
                printf("%s: end-of-file\n", argv[ap]);
            } else {
                printf("%s: ", argv[ap]);

                for (j = 0; j < numRead; j++) {
                    if (argv[ap][0] == 'r')
                        printf("%c", isprint((unsigned char) buf[j]) ?
                            buf[j] : '?');
                    else
                        printf("%02x ", (unsigned int) buf[j]);
                }

                printf("\n");
            }
            free(buf);
            break;

        case 'w': /* Write string at current offset */
            numWritten = write(fd, &argv[ap][1], strlen(&argv[ap][1]));
            if (numWritten == -1)
                errExit("write");

            printf("%s: wrote %ld bytes\n", argv[ap], (long) numWritten);
            break;

        case 's': /* Change file offset */
            offset = getLong(&argv[ap][1], GN_ANY_BASE, argv[ap]);
            if (lseek(fd, offset, SEEK_SET) == -1)
                errExit("lseek");

            printf("%s: seek succeeded\n", argv[ap]);
            break;

        default:
            cmdLineErr("Argument must start with [rRws]: %s\n", argv[ap]);
        }
    }
    exit(EXIT_SUCCESS);
}
----------------------------------------------------fileio/seek_io.c
```

下面 shell 会话日志演示了清单 4-3 程序的使用，显示了当我们从文件空洞中读取字节时的情况：

```
$ touch tfile                          创建新的空白文件
$ ./seek_io tfile s100000 wabc         seek 到 100000 偏移，写入“abc”
s100000: seek succeeded
wabc: wrote 3 bytes
$ ls -l tfile                          检查文件大小
-rw-r--r-- 1 mtk users 100003 Feb 10 10:35 tfile
$ ./seek_io tfile s10000 R5            seek 到 10000，从空洞读取 5 字节
s10000: seek succeeded
R5: 00 00 00 00 00                     空洞中的字节全为 0
```

## 4.8 通用 I/O 模型之外的操作：ioctl()

ioctl()系统调用是一个通用机制，可以执行本章前面描述的通用 I/O 模型之外的文件和设备操作。

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

int ioctl(int fd, int request, ... /* argp */);
                                成功时返回 request 相关的值，出错返回-1
```

fd 参数是打开的文件描述符，也是 request 要执行控制操作的那个文件。设备相关头文件定义的常量可以传递给 request 参数。

上面的省略号（…）是标准 C 符号，表示 ioctl()的第三个参数 argp 可以是任何类型。request 参数的值允许 ioctl()确定自己需要的 argp 类型。通常 argp 是一个整型或结构体指针，有些情况下没有 argp 参数。

我们在后面章节会看到许多 ioctl()的使用（比如 15.5 节）。

> SUSv3 对 ioctl()的唯一规定是 STREAM 设备的控制操作（STREAM 机制是 System V 特性，Linux 主版本内核并不支持，不过已经开发了一些插件实现）。本书讨论的其它 ioctl()操作都不是 SUSv3 规范。但是 ioctl()调用很早的时候就一直是 UNIX 系统的一部分，因此我们后面讨论的一些 ioctl()操作在许多 UNIX 实现中都可用。在我们讨论每个 ioctl()操作时，会标明可移植性方面的问题。

## 4.9 小结

要对普通文件执行 I/O 操作，首先必须使用 open()获得文件描述符，然后使用 read()和 write()执行 I/O。在所有 I/O 操作结束之后，我们应该使用 close()释放文件描述符和相关的资源。这些系统调用可以用于所有文件类型的 I/O 操作。

通用 I/O 模型要求所有文件类型和设备驱动实现相同的 I/O 接口，意味着程序不需要为特定文件类型编写代码，相同的代码可以用于任何类型的文件。

对每个打开的文件，内核都维护了一个文件偏移，用于确定下一个 read 或 write 将操作的文件位置。read 和 write 会隐式地更新文件偏移，也可以使用 lseek()显式地调整文件偏移，设置为文件内的任何位置，也可以超过文件末尾。向超过文件末尾的位置写入数据会创建文件空洞。从文件空洞读取的数据全部为 0。

ioctl()系统调用提供标准文件 I/O 模型之外的设备和文件操作功能。

## 4.10 习题

4-1. tee 命令读取标准输入直到 end-of-file，并把输入复制到标准输出，以及命令行参数指定的文件中（我们在 44.7 节讨论 FIFO 时会展示该命令的使用例子）。请你使用 I/O 系统调用实现 tee 命令。默认情况下 tee 覆盖指定名字的现有文件。请你实现-a 命令行选项（tee –a file），使 tee 追加文本到已经存在文件的末尾。（参考附录 B 对 getopt()函数的描述，它可以用来解析命令行选项）。

4-2. 编写一个类似 cp 的程序，当用来复制包含空洞的文件时（null 字节序列），要求在目标文件中也创建相应的空洞。

# 第 5 章 文件 I/O：更多细节

在这一章，我们将扩展前一章对文件 I/O 的讨论。

我们将继续深入 open()系统调用的讨论，解释原子性的概念——系统调用执行的动作是单一不可中断的步骤。原子性是许多系统调用能够正确执行操作的必要条件。

我们还介绍另一个文件相关的系统调用，多用途的 fcntl()，并展现了它的一个用途：获取和设置打开文件状态标志。

接着我们学习内核用来表示文件描述符和打开文件的数据结构。理解这些结构体之间的关系，有助于我们随后章节解释文件 I/O 的许多微妙细节。基于这个模型，我们接下来解释怎样复制文件描述符。

然后我们考虑一些提供扩展读取和写入功能的系统调用。这些系统调用允许我们在文件的特定位置执行 I/O，而不修改文件偏移；以及同时读取和写入程序的多个缓冲区。

我们简要地介绍了非阻塞 I/O 的概念，并描述了一些支持大型文件 I/O 操作的扩展接口。

由于许多系统程序需要使用临时文件，我们也学习几个创建和使用临时文件的库函数，这些函数随机生成唯一的临时文件名。

## 5.1 原子性和竞争条件

原子性是我们讨论系统调用操作时经常遇到的一个概念。所有系统调用都是原子执行的。也就是说内核确保一个系统调用的所有步骤都能以单一操作完整地结束，不会被其它进程或线程中断。

原子性是某些操作能够成功完成的必要条件。特别值得一提的是，原子性使得我们避免了竞争条件（有时候称为竞争危险）。两个进程（或线程）同时操作共享资源时，如果执行结果依赖于进程获得 CPU 访问的具体顺序，就产生了竞争条件。

在接下来的几页里，我们查看两个文件 I/O 相关的竞争条件，并演示如何使

用 open()的标志来确保相关文件操作的原子性，从而消除这些竞争条件。

在后面 22.9 节讨论 sigsuspend()，和 24.4 节讨论 fork()时，我们还会再次遇到竞争条件。

### 互斥地创建文件

在 4.3.1 节，我们说过指定 O_EXCL 结合 O_CREAT 标志，如果文件已经存在则 open()将返回错误。这使得进程可以确保自己是新文件的创建者。检查文件是否存在，和创建该文件的过程是原子的。这一点非常重要，考虑清单 5-1 中的代码，我们没有使用 O_EXCL 标志（在这段代码中，我们显示了 getpid()返回的进程 ID，可以让我们区分同一个程序同时运行的不同结果）。

*清单 5-1：互斥打开文件的错误代码*

```
----------------------------------------fileio/bad_exclusive_open.c
fd = open(argv[1], O_WRONLY); /* Open 1: check if file exists */
if (fd != -1) { /* Open succeeded */
    printf("[PID %ld] File \"%s\" already exists\n",
        (long) getpid(), argv[1]);
    close(fd);
} else {
    if (errno != ENOENT) { /* Failed for unexpected reason */
        errExit("open");
    } else {
        /* WINDOW FOR FAILURE */
        fd = open(argv[1], O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
        if (fd == -1)
            errExit("open");

        printf("[PID %ld] Created file \"%s\" exclusively\n",
            (long) getpid(), argv[1]); /* MAY NOT BE TRUE! */
    }
}
----------------------------------------fileio/bad_exclusive_open.c
```

上面代码除了要烦琐地使用两次 open()调用之外，还包含一个 bug。假设进程调用了第一个 open()，此时文件并不存在；但是等到进程第二次调用 open()

时，其它进程已经创建了这个文件。如果内核调度器认为进程时间片已经用完并将 CPU 控制权交给其它进程，如图 5-1 所示；或者两个进程同时运行在多核处理器系统中，这种情况就很有可能发生。图 5-1 假设两个进程都执行清单 5-1 中的代码。在这种场景下，进程 A 最终可能错误地认为自己已经创建了该文件，因为第二个 open()无论文件是否存在都会返回成功。

虽然进程错误地认为自己是文件创建者的概率相对较低，但是无论如何这种可能性的存在，降低了代码的可靠性。操作结果依赖于两个进程的调度顺序，表明这是一个竞争条件。

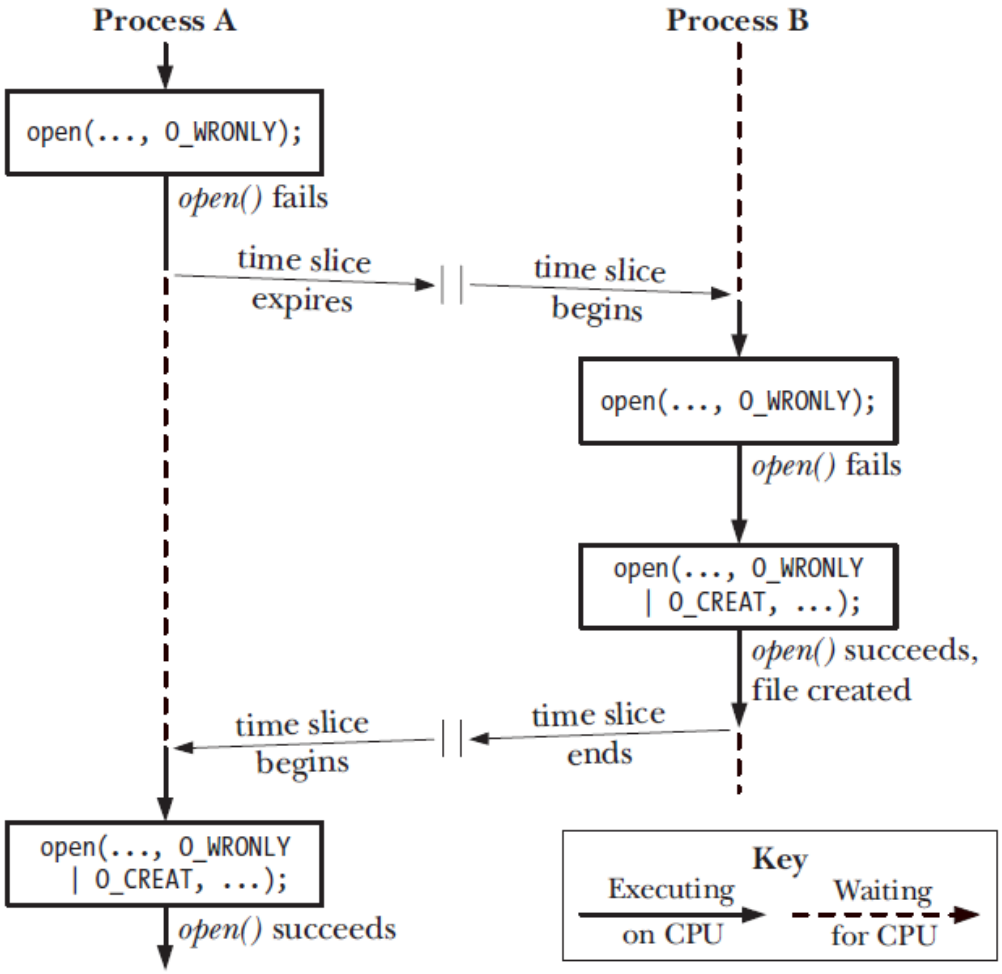


*图 5-1：互斥创建文件失败*

要演示上面代码存在的真正问题，我们可以替换清单 5-1 中已经注释掉的那行 WINDOW FOR FAILURE，在文件存在检查和创建文件之间，加入一段长时间的延迟代码：

```
printf("[PID %ld] File \"%s\" doesn't exist yet\n",
    (long) getpid(), argv[1]);
if (argc > 2) {       /* Delay between check and create */
    sleep(5);         /* Suspend execution for 5 seconds */
    printf("[PID %ld] Done sleeping\n", (long) getpid());
}
```

sleep()库函数挂起进程指定的秒数。我们在 23.4 节会讨论这个函数。

如果我们同时运行两次清单 5-1 中的程序，我们可以看到两个进程都声称自己已经成功创建该文件：

```
$ ./bad_exclusive_open tfile sleep &
[PID 3317] File "tfile" doesn't exist yet
[1] 3317
$ ./bad_exclusive_open tfile
[PID 3318] File "tfile" doesn't exist yet
[PID 3318] Created file "tfile" exclusively
$ [PID 3317] Done sleeping
[PID 3317] Created file "tfile" exclusively      [Not true]
```

两个进程都声称创建了该文件，是因为第一个进程检查文件是否存在，到创建文件之间被中断。使用一个 open()调用，指定 O_CREAT 和 O_EXCL 标志可以阻止这种情况的发生，确保检查和创建的步骤以原子操作进行（不可中断）。

### 追加数据至文件

另一个需要原子操作的例子是多个进程同时追加数据到相同文件（如全局日志文件）。要实现这个目的，我们可能会采用如下代码：

```
if (lseek(fd, 0, SEEK_END) == -1)
    errExit("lseek");
if (write(fd, buf, len) != len)
    fatal("Partial/failed write");
```

但是这段代码面临前面例子同样的问题。如果第一个进程在 lseek()和 write()之间被中断，而第二个进程也正好在执行这段代码，则两个进程都会设置文件偏移为相同位置，然后当第一个进程重新被调度运行时，它就会覆盖第二个进程已

经写入文件的数据。这里也存在一个竞争条件，因为结果依赖于这两个进程被调度的顺序。

要避免这个问题，就需要使 seek 到文件末尾下一个字节，和写入操作以原子方式进行。这就是以 O_APPEND 标志打开文件要实现的功能。

> 某些文件系统（如 NFS）不支持 O_APPEND。在这种情况下，内核会使用上面的非原子调用序列，结果就是刚刚描述过的可能污染文件。

## 5.2 文件控制操作：fcntl()

fcntl()系统调用可以对已打开的文件描述符执行许多控制操作。

```
#include <fcntl.h>

int fcntl(int fd, int cmd, ...);
                                成功时根据 cmd 返回，出错返回-1
```

cmd 参数可以指定大量的操作。下面几节我们会介绍其中的几个，其余则留待后面章节详述。

正如省略号所示，fcntl()的第三个参数可以是不同类型，有时候也可以省略。内核使用 cmd 参数的值来确定自己需要的参数值类型（如果有的话）。

## 5.3 打开文件状态标志

fcntl()的一个作用是获取和修改文件的访问模式和打开文件状态标志（也就是 open()调用中指定的 flags 参数值）。要获取这些值，我们为 cmd 指定 F_GETFL：

```
int flags, accessMode;

flags = fcntl(fd, F_GETFL); /* 第三个参数不需要 */
if (flags == -1)
    errExit("fcntl");
```

然后我们可以如下测试文件是否以同步写入模式打开：

```
if (flags & O_SYNC)
    printf("writes are synchronized\n");
```

SUSv3 要求只有 open()指定的状态标志和 fcntl() F_SETFL 指定的标志，才能设置到打开文件描述符中。但是 Linux 有一个地方偏离了这个规定：如果应用使用了 5.10 节打开大文件的技术进行编译，则 F_GETFL 获得的标志总是包含 O_LARGEFILE。

检查文件的访问模式则稍微复杂一些，因为 O_RDONLY(0), O_WRONLY(1), O_RDWR(2)常量并不对应于打开文件状态标志中的某个位。因此如果要检查访问模式，我们要用 O_ACCMODE 常量对 flags 进行掩码，然后再与上面三个常量检测是否相等：

```
accessMode = flags & O_ACCMODE;
if (accessMode == O_WRONLY || accessMode == O_RDWR)
    printf("file is writable\n");
```

我们可以使用 fcntl()的 F_SETFL 命令来修改某些打开文件状态标志。可以修改的标志包括：O_APPEND, O_NONBLOCK, O_NOATIME, O_ASYNC, O_DIRECT。试图修改其它标志会被忽略（某些 UNIX 实现允许 fcntl()修改其它标志，如 O_SYNC）。

以下情况使用 fcntl()修改打开文件状态标志特别有用：

- 文件不是由调用程序打开的，因此程序无法控制 open()时使用的 flags（如程序启动前打开的三个标准描述符）。
- 文件描述符不是使用 open()系统调用获得。例如 pipe()系统调用创建一个管道，并返回两个文件描述符分别引用管道的两端；socket()调用创建一个 socket 并返回一个文件描述符。

要修改打开文件的状态标志，我们先使用 fcntl()来获得现有标志，然后修改相应的位，最后再次调用 fcntl()来更新状态标志。因此要启用 O_APPEND 标志，我们可以编写如下代码：

```
int flags;
flags = fcntl(fd, F_GETFL);
if (flags == -1)
    errExit("fcntl");
flags |= O_APPEND;
if (fcntl(fd, F_SETFL, flags) == -1)
    errExit("fcntl");
```

## 5.4 文件描述符和打开文件之间的关系

到现在为止，看起来好像文件描述符和打开文件都是一一对应的。但是实际情况却并不是这样。可以有多个描述符引用同一个打开文件，而且这个特性非常有用。这些文件描述符可以由相同进程也可以由不同进程打开。

要理解这种关系，我们需要先学习内核维护的三个数据结构：

- 每个进程的文件描述符表；
- 系统级的打开文件说明表；
- 文件系统 i-node 表。

对于每个进程，内核都维护了一个打开文件描述符表。表中的每一项都记录了一个文件描述符的信息，包括：

- 一组控制文件描述符操作的标志（其实只有一个 close-on-exec 标志，27.4 节讨论）；
- 打开文件说明的引用。

内核维护了一个系统级的所有打开文件说明表（这个表有时候也叫做打开文件表，它的每一项称为打开文件句柄）。一个打开文件说明存储了打开文件相关的所有信息，包括：

- 当前文件偏移（由 open()和 write()更新，或者 lseek()显式修改）；
- 打开文件时指定的状态标志（open()的 flags 参数）；
- 文件访问模式（只读、只写、读写）；
- 信号驱动 I/O 相关的设置（63.3 节）；
- 该文件 i-node 对象的引用。

每个文件系统都有一个该文件系统内所有文件的 i-node 表。第 14 章会更加详细地讨论 i-node 结构体和文件系统。现在我们只需要知道每个文件的 i-node 包含了以下信息：

- 文件类型（如普通文件、socket、FIFO）和权限；
- 指向文件锁列表的一个指针；
- 文件的许多属性，包括不同类型文件操作相关的大小、时间戳等。

> 这里我们简单地说明一下磁盘中和内存中 i-node 的区别。磁盘中的 i-node 记录了文件的持久属性，如文件类型、权限、和时间戳等。当访问文件时，就会在内存中创建一份 i-node 拷贝，而内存中的 i-node 则记录了引用 i-node 的打开文件描述符数量，以及 i-node 所在设备的主要（major）和次要（minor）ID。内存 i-node 还记录了文件打开相关的许多短暂属性，如文件锁。

图 5-2 阐明了文件描述符、打开文件、和 i-node 之间的关系。在这幅图中，两个进程都有若干打开的文件描述符。

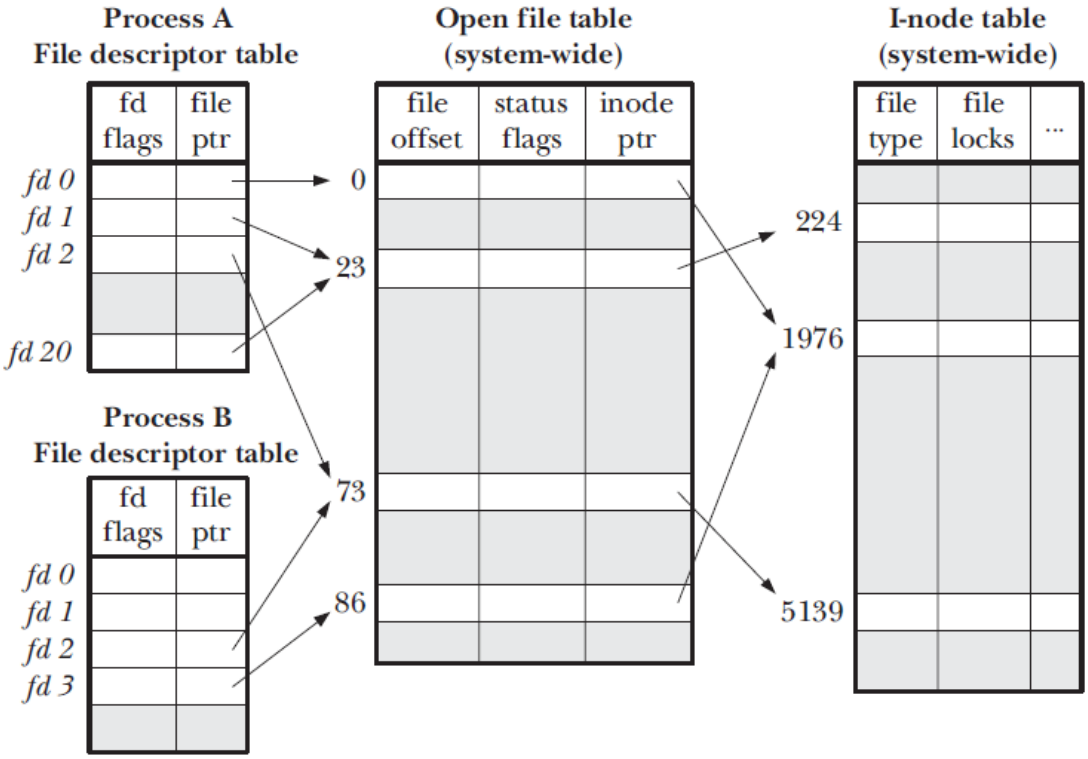


*图 5-2：文件描述符、打开文件、i-node 的关系*

在进程 A 中，描述符 1 和 20 都引用同一个打开文件（标签 23）。调用 dup(), dup2(), fcntl()可能会导致这种情况出现（5.5 节）。

进程 A 的描述符 2 和进程 B 的描述符 2 都引用同一个打开文件（标签 73）。这种情况发生的原因包括：调用 fork()（如进程 A 是进程 B 的父亲，反之亦然）；

或者一个进程使用 UNIX domain socket 把打开描述符传递给了另一个进程（61.13.3 节）。

最后，进程 A 的描述符 0 和进程 B 的描述符 3 引用不同的打开文件，但是这两个打开文件却又引用同一个 i-node 表项（标签 1976）——换句话说就是同一个文件。这是由于两个进程各自对相同文件调用了 open()，一个进程打开相同文件两次也会出现类似的情况。

我们可以从上面这些讨论得出以下结论：

- 引用同一个打开文件的两个不同文件描述符，共享相同的文件偏移。因此如果文件偏移被其中一个描述符改变（调用 read(), write(), lseek()），这个改变对其它文件描述符也是可见的。不管两个文件描述符属于同一个进程还是不同进程，结果都是一样的。
- 使用 fcntl() F_GETFL 和 F_SETFL 操作获取和修改打开文件状态标志时（O_APPEND, O_NONBLOCK, O_ASYNC），类似的范围规则同样适用。
- 相比之下，文件描述符标志（如 close-on-exec 标志）则是进程和文件描述符私有的。修改这些标志不会影响相同进程或其它进程的其它文件描述符。

## 5.5 复制文件描述符

使用（Bourne shell）I/O 重定向的语法 2>&1，可以通知 shell 我们想让标准错误（文件描述符 2）重定向到标准输出（文件描述符 1）发送的地方。因此下面命令会把标准输出和标准错误都重定向至文件 results.log（因为 shell 从左向右执行 I/O 重定向）：

```
$ ./myscript > results.log 2>&1
```

shell 通过复制文件描述符 2，让描述符 2 和描述符 1 引用同一个打开文件（和图 5-2 中进程 A 的描述符 1 和 20 引用同一个打开文件一样），来实现标准错误的重定向。dup()和 dup2()系统调可以完成这个任务。

注意 shell 简单地打开 results.log 文件两次（描述符 1 一次，描述符 2 一次）

是不能解决这个问题的。原因之一是两个文件描述符不能共享同一个文件偏移指针，因此会导致覆盖彼此的输出。另一个原因是文件可能不是磁盘文件。考虑下面命令，把标准错误和标准输出一起传送至同一管道。

```
$ ./myscript 2>&1 | less
```

dup()调用的 oldfd 参数是一个打开的文件描述符，dup()返回一个新的描述符，引用到同一个打开文件。新描述符确保是系统中未使用文件描述符最小的那个。

```
#include <unistd.h>

int dup(int oldfd);
                                成功时返回新的文件描述符，出错返回-1
```

假设我们有下面这个调用：

```
newfd = dup(1);
```

再假设 shell 按惯例已经替程序打开了文件描述符 0、1、2，并且没有使用其它描述符，上面 dup()调用会使用文件描述符 3 来复制描述符 1。

如果我们想要把上面的描述符复制为 2，就可以使用下面技术：

```
close(2);          /* 释放文件描述符 2 */
newfd = dup(1);    /* 重用文件描述符 2 */
```

上面代码只有描述符 0 已经打开才能正常工作。为了简化上面的代码，并且确保我们能够得到想要的文件描述符，可以使用 dup2()调用。

```
#include <unistd.h>

int dup2(int oldfd, int newfd);
                                成功时返回新的文件描述符，出错返回-1
```

dup2()系统调用使用 newfd 提供的描述符数值，来复制 oldfd 文件描述符。如果 newfd 指定的文件描述符已经打开，dup2()会先关闭它（关闭时出现的错误会被忽略；安全的编程实践是在调用 dup2()之前，显式地使用 close()来关闭 newfd）。

我们可以把前面调用 close()和 dup()的代码简化如下：

```
dup2(1, 2);
```

成功调用 dup2()返回新复制的描述符的数值（也就是 newfd 的值）。

如果 oldfd 是非法文件描述符，则 dup2()失败，设置错误 EBADF，并且不会关闭 newfd。如果 oldfd 是合法文件描述符，而 oldfd 和 newfd 的值相同，则 dup2()不做任何事情（不关闭 newfd，dup2()直接返回 newfd）。

fcntl()的 F_DUPFD 操作提供了复制文件描述符更加灵活的接口：

```
newfd = fcntl(oldfd, F_DUPFD, startfd);
```

这个调用使用大于或者等于 startfd，而且是尚未使用的最小文件描述符来复制 oldfd。如果我们想要确保新描述符的值在特定范围内，这个接口就非常有用。调用 dup()和 dup2()总是可以改写成调用 close()和 fcntl()，尽管前者更加简练（注意 dup()和 fcntl()返回的某些 errno 错误代码是不同的，请参考手册页）。

从图 5-2 中我们可以看到，复制文件描述符可以共享相同的文件偏移，以及相同的打开文件状态标志。但是新描述符仍然会有自己的一组文件描述符标志，并且 close-on-exec 标志（FD_CLOEXEC）也总是会被关闭。我们下面讨论的接口，则可以显式地控制新文件描述符的 close-on-exec 标志。

dup3()系统调用执行 dup2()同样的操作，但是增加了一个额外的参数 flags，用来修改系统调用行为的位掩码。

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

int dup3(int oldfd, int newfd, int flags);
                                        成功时返回新的文件描述符，出错返回-1
```

目前 dup3()只支持 O_CLOEXEC 一个标志，可以让内核为新文件描述符启用 close-on-exec 标志（FD_CLOEXEC）。这个标志的作用，我们已经在 4.3.1 节描述 open()的 O_CLOEXEC 标志时讨论过了。

dup3()是 Linux 2.6.27 引入的新系统调用，而且是 Linux 特定的。

从 Linux 2.6.24 开始，fcntl()增加了一个额外的复制文件描述符操作：F_DUPFD_CLOEXEC。这个标志和 F_DUPFD 做的事情一样，但是对新文件描述符设置 close-on-exec 标志（FD_CLOEXEC）。再次重申，这个操作和 open()的 O_CLOEXEC 标志一样，某些时候是非常有用的。F_DUPFD_CLOEXEC 不是 SUSv3 标准，在 SUSv4 中才有规定。

## 5.6 指定偏移位置的文件 I/O：pread()和 pwrite()

pread()和 pwrite()系统调用的操作和 read()和 write()是一样的，但是文件 I/O 是在 offset 指定的位置执行，而不是当前文件偏移。这两个调用都不会修改当前文件偏移。

```
#include <unistd.h>

ssize_t pread(int fd, void *buf, size_t count, off_t offset);
                                        返回读取的字节数；EOF 返回 0；出错返回-1

ssize_t pwrite(int fd, const void *buf, size_t count, off_t offset);
                                                返回写入的字节数；出错返回-1
```

调用 pread()相当于自动执行以下调用：

```
off_t orig;
orig = lseek(fd, 0, SEEK_CUR);   /* 保存当前偏移 */
lseek(fd, offset, SEEK_SET);
s = read(fd, buf, len);
lseek(fd, orig, SEEK_SET);       /* 还原原始文件偏移 */
```

pread()和 pwrite()的参数 fd 引用的文件都必须可以 seek（这个文件描述符允许执行 lseek()调用）。

这两个系统调用在多线程应用中特别有用。我们在第 29 章将看到，进程的所有线程都共享相同的文件描述符表。这意味着每个打开文件的文件偏移对所有线程来说是全局的，使用 pread()和 pwrite()，多个线程可以同时对相同文件描述符执行 I/O 操作，而不受其它线程改变文件偏移的影响。如果我们试图使用 lseek()

加 read()或 write()，那就会引入一个竞争条件，情况类似于我们在 5.1 节讨论的 O_APPEND 标志（多个进程的文件描述符引用相同打开文件时，pread()和 pwrite()系统调用同样可以避免竞争条件，一样非常有用）。

如果需要频繁地执行 lseek()然后执行文件 I/O，则 pread()和 pwrite()系统调用还能提供一定的性能优势。这是因为单个的 pread()（或 pwrite()）系统调用的开销比两个系统调用（lseek()和 read()或 write()）的开销要小。不过系统调用的开销通常相对实际执行 I/O 需要的时间几乎可以忽略。

## 5.7 Scatter-Gather I/O：readv()和 writev()

readv()和 writev()系统调用执行 scatter-gather I/O。

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

ssize_t readv(int fd, const struct iovec *iov, int iovcnt);
                                    返回读取的字节数，EOF 时返回 0，出错返回-1

ssize_t writev(int fd, const struct iovec *iov, int iovcnt);
                                                返回写入的字节数，出错返回-1
```

之前介绍的 read 和 write 操作都只允许一个缓冲区数据，这两个函数则只需一次调用就可以传输多个数据缓冲区。使用 iov 数组来定义要传输数据的缓冲区组合，整数 iovcnt 则指定了 iov 中的元素个数。iov 的每个元素都是一个结构体，定义如下：

```
struct iovec {
    void *iov_base;    /* 缓冲区的起始地址 */
    size_t iov_len;    /* 缓冲区要传输的字节数 */
};
```

SUSv3 允许实现对 iov 的元素个数设置限制。实现可以通过定义<limits.h>中的 IOV_MAX 来告知限制值，或者也可以通过调用 sysconf(_SC_IOV_MAX)在运行时动态返回该限制（我们在 11.2 节讨论 sysconf()函数）。SUSv3 要求这个限制值最少支持 16 个。Linux 定义 IOV_MAX 为 1024，对应于内核对这个向量大小的限制（内核定义的常量 UIO_MAXIOV）。

但是 glibc 对 readv()和 writev()的包装函数默默地做了一些额外的工作。如果由于 iovcnt 太大而导致系统调用失败，则包装函数会临时分配一个足够大的缓冲区来保存 iov 描述的所有项目，然后再执行 read()或 write()调用（参考下面的讨论：如何使用 write()来实现 writev()）。

图 5-3 显示了 iov、iovcnt 参数与缓冲区之间的关系

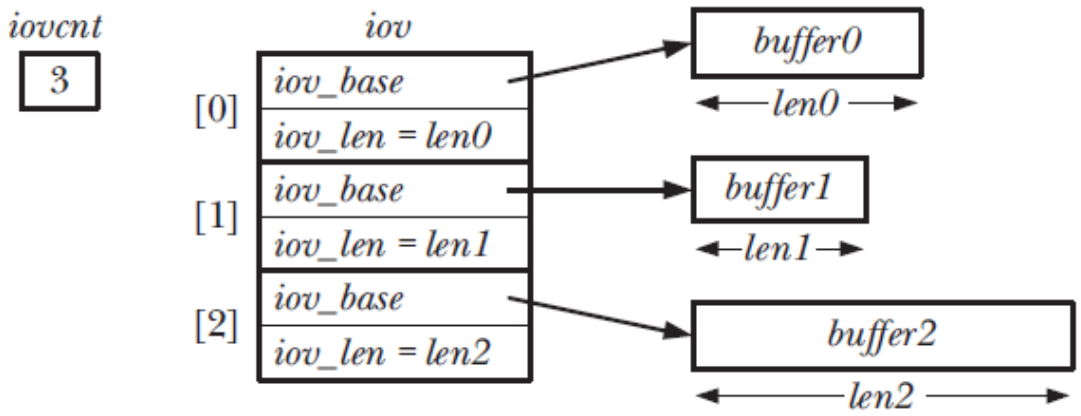


*图 5-3：iovec 数组和缓冲区的关联示例*

**Scatter 输入**

readv()系统调用执行 scatter 输入：从文件描述符 fd 引用的文件中读取连续的一段字节，并将这些字节放置（scatter）到 iov 指定的缓冲区中。readv()首先从 iov[0]开始填充，填满一个缓冲区再继续下一个。

readv()的一个重要属性是所有操作原子完成；也就是说，从调用进程的角度来看，内核在文件和用户内存中只执行一次数据传输。例如读取文件时，即使另一个进程（或线程）同时操作文件偏移，readv()也可以确保读取到连续的字节。

成功完成时 readv()返回读取的字节数，遇到 end-of-file 时返回 0。调用方必须检查返回值，以确认是否读取到请求的全部字节。如果没有足够的可用字节，就只有部分缓冲区会被填充，最后的缓冲区可能只被部分填充。

清单 5-2 演示了 readv()的使用。

> 例子程序使用前缀“t_”并跟随一个函数名（如清单 5-2 中的 t_readv.c），表示这个程序主要目的是演示某个系统调用或库函数的使用。

*清单 5-2：使用 readv()执行 scatter 输入*

```
----------------------------------------------------fileio/t_readv.c
#include <sys/stat.h>
#include <sys/uio.h>
#include <fcntl.h>
#include "tlpi_hdr.h"
```

```
int
main(int argc, char *argv[])
{
    int fd;
    struct iovec iov[3];
    struct stat myStruct; /* First buffer */
    int x; /* Second buffer */

    #define STR_SIZE 100
    char str[STR_SIZE]; /* Third buffer */
    ssize_t numRead, totRequired;

    if (argc != 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s file\n", argv[0]);

    fd = open(argv[1], O_RDONLY);
    if (fd == -1)
        errExit("open");

    totRequired = 0;

    iov[0].iov_base = &myStruct;
    iov[0].iov_len = sizeof(struct stat);
    totRequired += iov[0].iov_len;

    iov[1].iov_base = &x;
    iov[1].iov_len = sizeof(x);
    totRequired += iov[1].iov_len;

    iov[2].iov_base = str;
    iov[2].iov_len = STR_SIZE;
    totRequired += iov[2].iov_len;

    numRead = readv(fd, iov, 3);
    if (numRead == -1)
        errExit("readv");

    if (numRead < totRequired)
        printf("Read fewer bytes than requested\n");

    printf("total bytes requested: %ld; bytes read: %ld\n",
```

```
        (long) totRequired, (long) numRead);

    exit(EXIT_SUCCESS);
}
----------------------------------------------------fileio/t_readv.c
```

**Gather 输出**

writev()系统调用执行 gather 输出，它连接（gather）iov 指定的所有缓冲区的数据，并将其作为连续的顺序字节写入到 fd 引用的文件。缓冲区以 iov 数组顺序 gather，从 iov[0]缓冲区开始。

和 readv()一样，writev()也是原子完成，所有数据从用户内存到 fd 引用文件之间只执行一次传输操作。因此当写入到普通文件时，我们可以确保所有请求的数据都被连续地写入到文件中，而不会被其它进程（或线程）的写入操作打断。

和 write()一样，writev()也可能部分写入。因此我们必须检查 writev()的返回值，来确定是否所有请求字节都已被写入。

readv()和 writev()的主要优点是方便和速度。例如，我们可以把 writev()替换为以下方式：

- 分配单一大缓冲区，从进程地址空间复制要写入的所有数据到该缓冲区，然后执行 write()输出缓冲区的数据。
- 对每个缓冲区执行一次 write()调用。

第一种方式语义上和 writev()等价，但实现起来却不方便（也不高效），因为需要在用户空间分配缓冲区并复制数据。

第二种方式语义上和单一 writev()调用是不同的，因为多个 write()调用并不是原子执行。此外执行单一 writev()系统调用比执行多个 write()调用的开销也更小（参考 3.1 节关于系统调用的讨论）。

**指定偏移位置执行 scatter-gather I/O**

Linux 2.6.30 增加了两个新的系统调用，整合了 scatter-gather I/O 功能以及指定偏移位置执行 I/O 的能力：preadv()和 pwritev()。这两个系统调用不是标准的，

但在现代 BSD 系统中也可用。

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

ssize_t preadv(int fd, const struct iovec *iov, int iovcnt, off_t offset);
                                        返回读取的字节数，EOF 返回 0，出错返回-1。

ssize_t pwritev(int fd, const struct iovec *iov, int iovcnt, off_t offset);
                                        返回写入的字节数，出错返回-1。
```

preadv()和 pwritev()系统调用执行 readv()和 writev()相同的操作，只不过文件 I/O 在 offset 指定的位置进行（类似 pread()和 pwrite()）。

如果应用希望整合 scatter-gather I/O 的优点和指定位置执行 I/O 的能力，而不依赖于当前文件偏移（如多线程应用），这两个系统调用就非常有用。

## 5.8 截断文件：truncate()和 ftruncate()

truncate()和 ftruncate()系统调用设置文件的大小为 length 指定的值。

```
#include <unistd.h>

int truncate(const char *pathname, off_t length);
int ftruncate(int fd, off_t length);
                                        成功时都返回 0，出错返回-1
```

如果文件长度大于 length，超出的数据就会丢失。如果文件长度小于 length，则使用 null 字节序列或空洞对文件进行填充扩展。

这两个系统调用的区别在于如何指定文件。truncate()要求文件可访问和可写入，使用 pathname 字符串指定文件。如果 pathname 是符号链接，则首先解引用。ftruncate()系统调用则使用一个已经打开为写入的文件描述符，不会修改文件偏移。

如果 ftruncate()的 length 参数超出了当前文件大小，SUSv3 允许两种行为：要么扩展文件（Linux 就是这样），要么返回错误。XSI 依从的系统必须采用前一种行为。SUSv3 要求 truncate()在 length 大于文件长度时总是扩展文件。

truncate()是唯一一个可以直接修改文件内容，而不需要通过 open()（或者其它方式）获取文件描述符的系统调用。

## 5.9 非阻塞 I/O

打开文件时指定 O_NONBLOCK 标志有两个作用：

- 如果文件不能立即打开，open()将返回错误而不是阻塞。open()阻塞的典型例子是 FIFO（44.7 节）。
- 在成功调用 open()之后，所有的 I/O 操作也是非阻塞的。如果 I/O 系统调用不能立即完成，则要么执行部分数据传输，要么系统调用以 EAGAIN 或 EWOULDBLOCK 错误失败。具体返回哪个错误视系统调用而定。在 Linux 以及许多 UNIX 实现中，这两个错误常量是同义的。

非阻塞模式可以用于设备（如终端和伪终端）、管道、FIFO、和 socket 等。（由于管道和 socket 的文件描述符不是使用 open()获得，我们必须使用 fcntl()的 F_SETFL 操作来启用这个标志，5.3 节已经讨论过）。

对于普通文件通常忽略 O_NONBLOCK 标志，因为内核缓冲区缓存确保普通文件 I/O 不会阻塞，13.1 节会进一步讨论。不过当普通文件采用了强制文件锁（55.4 节）时，O_NONBLOCK 标志就确实会有作用。

我们会在第 44.9 节和第 63 章更详尽地讨论非阻塞 I/O。

历史上基于 System V 的系统提供 O_NDELAY 标志，语义类似于 O_NONBLOCK。主要区别在于 System V 的 write()如果不能完成或 read()没有可用输入，会返回 0 而不是出错。这个行为对于 read()来说是有问题的，因为我们无法区分 end-of-file 的情况。因此第一版 POSIX 标准引入了 O_NONBLOCK。某些 UNIX 实现继续提供旧语义的 O_NDELAY 标志。Linux 定义了 O_NDELAY 常量，但等价于 O_NONBLOCK。

## 5.10 大文件 I/O

off_t 数据类型用来保存文件偏移，通常实现为带符号长整型（必须使用带符号类型，因为-1 用来表示错误条件）。在 32 位体系架构中（如 x86-32）就意味着

文件大小限制为 $2^{31}$-1 字节（也就是 2GB）。

但是磁盘容量在很久以前就超过了这个限制，因此 32 位 UNIX 系统必须想办法实现大文件 I/O。由于这是许多实现共同面临的问题，UNIX 厂商联盟联合组成了 Large File Summit（LFS），来增强 SUSv2 规范以提供访问大文件的功能。我们在这一节概述 LFS 增强功能（完整的 LFS 规范完成于 1996 年，可以在 http://opengroup.org/platform/lfs.html 找到）。

Linux 从内核 2.4（同时要求 glibc 2.2 及以上）开始为 32 位系统提供 LFS 支持。此外相应的文件系统还必须支持大文件。多数 Linux 文件系统都支持大文件，但某些其它文件系统则不支持（著名的如 Microsoft 的 VFAT 和 NFSv2 都局限为 2GB，无论是否采用 LFS 扩展）。

> 由于 64 位体系架构（如 Alpha 和 IA-64）下的 long 型为 64 位，这些体系架构通常没有 LFS 试图解决的 2GB 文件限制的问题。无论如何，即使是在 64 位系统中，某些 Linux 文件系统的实现细节也可能导致文件的最大长度小于 $2^{63}$-1。多数情况下，这些限制比起磁盘大小要高许多，因此对文件大小并没有实践上的局限。

我们可以按两种方式来编写支持 LFS 功能的应用：

- 使用支持大文件的可选 API，LFS 设计这些 API 作为单一 UNIX 规范的“过渡型扩展”。因此依从 SUSv2 或 SUSv3 的系统不要求提供这些 API，但许多系统还是提供了它。这种方式目前已经废弃。
- 编译我们的程序时定义“_FILE_OFFSET_BITS”宏为 64，这是推荐的方式，因为它允许依从应用获得 LFS 功能的同时，不需要修改任何源代码。

**过渡型 LFS API**

要使用过渡型 LFS API，我们必须定义“_LARGEFILE64_SOURCE”特性测试宏，可在命令行中定义，或包含任何头文件之前定义。这个 API 提供了处理 64 位文件大小和偏移的功能函数。这些函数的名字和 32 位相应函数的名字一样，但是在函数名后面增加了 64 后缀。这些函数包括：fopen64(), open64(), lseek64(), truncate64(), stat64(), mmap64(), setrlimit64()等。（32 位版本的函数有些我们已经讨论过，其它在后续章节讨论）。

要访问大文件，我们只需要使用这些函数的 64 位版本。例如，要打开一个

大文件，我们可以编写如下代码：

```
fd = open64(name, O_CREAT | O_RDWR, mode);
if (fd == -1)
    errExit("open");
```

调用 open64()等价于调用 open()时指定 O_LARGEFILE 标志。使用 open()打开超过 2GB 的文件时，如果不指定这个标志就会返回错误。

除了上面提到的这些函数，LFS API 还增加了一些新的数据类型，包括：

- struct stat64：类似于 stat 结构体（15.1 节），允许使用大文件大小。
- off64_t：表示文件偏移的 64 位类型。

off64_t 数据类型用于 lseek64()函数中，如清单 5-3 所示。这个程序有两个命令行参数：要打开的文件名；和一个指定文件偏移的整数值。程序打开指定的文件，seek 到指定的文件偏移，然后写入一个字符串。下面 shell 会话显示了这个程序的使用，seek 到一个非常大的文件偏移（超过 10GB）然后写入一些字节：

```
$ ./large_file x 10111222333
$ ls -l x Check size of resulting file
-rw------- 1 mtk users 10111222337 Mar 4 13:34 x
```

*清单 5-3：访问大文件*

```
----------------------------------------------------fileio/large_file.c
#define _LARGEFILE64_SOURCE
#include <sys/stat.h>
#include <fcntl.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int fd;
    off64_t off;

    if (argc != 3 || strcmp(argv[1], "--help") == 0)
        usageErr("%s pathname offset\n", argv[0]);

    fd = open64(argv[1], O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
```

```
    if (fd == -1)
        errExit("open64");

    off = atoll(argv[2]);
    if (lseek64(fd, off, SEEK_SET) == -1)
        errExit("lseek64");

    if (write(fd, "test", 4) == -1)
        errExit("write");

    exit(EXIT_SUCCESS);
}
--------------------------------------------------fileio/large_file.c
```

**_FILE_OFFSET_BITS 宏**

获得 LFS 功能首选的方法是编译程序时定义宏_FILE_OFFSET_BITS 为 64。一种办法是在 C 编译器命令行选项中指定宏：

```
$ cc -D_FILE_OFFSET_BITS=64 prog.c
```

或者我们也可以在源代码包含任何头文件之前定义这个宏：

```
#define _FILE_OFFSET_BITS 64
```

这样就自动把所有相关的 32 位函数和数据类型转换为 64 位。例如调用 open()实际上被转换为调用 open64()，off_t 数据类型也被定义为足够 64 位长。换句话说，我们只需要重新编译现有程序就能处理大文件，无需修改源代码。

使用_FILE_OFFSET_BITS 明显比过渡型 LFS API 要简单许多，但这个方法对编写应用也有要求（例如显式地使用 off_t 来声明保存文件偏移的所有变量，而不能使用 C 语言的本地整数类型）。

LFS 规范并没有强制要求_FILE_OFFSET_BITS 宏，它只提到这个宏是指定 off_t 数据类型长度的可选方法。某些 UNIX 实现使用不同的特性测试宏来获得这个功能。

> 如果我们试图使用 32 位函数来访问大文件（例如没有设置_FILE_OFFSET_BITS 为 64 的程序），则可能会遇到 EOVERFLOW 错误。例如使用 32 位的 stat()（15.1 节）来获取超过 2GB 文件的信息时就会出现这个错误。

### 传递 off_t 值给 printf()

LFS 没有解决的问题之一就是：如何传递 off_t 值给 printf()调用。在 3.6.2 节，我们学习过显示预定义系统数据类型（如 pid_t 和 uid_t）的可移植方法，就是把值转化为 long，然后使用 printf()的%ld 说明符。但是如果我们采用了 LFS 扩展，这样做对于 off_t 数据类型是不够的，因为 off_t 可能定义为大于 long 的类型（通常是 long long）。因此要显示 off_t 类型的值，我们要把它转化为 long long，并使用 printf()的%lld 说明符，如下所示：

```
#define _FILE_OFFSET_BITS 64
off_t offset; /* Will be 64 bits, the size of 'long long' */

/* Other code assigning a value to 'offset' */

printf("offset=%lld\n", (long long) offset);
```

这个方法也适用于相关的 blkcnt_t 数据类型，它定义在 stat 结构体中（15.1 节详细讨论）。

> 如果我们在不同的编译模块中传递 off_t 或 stat 函数参数，则我们必须确保所有模块都使用相同的类型大小（要么都定义_FILE_OFFSET_BITS 为 64，要么都不设置这个宏的值）。

## 5.11 /dev/fd 目录

对于每个进程，内核都提供一个特殊的虚拟目录/dev/fd。这个目录包含的文件形态是：/dev/fd/*n*，其中 n 是一个数值，是该进程某个打开的文件描述符。因此/dev/fd/0 就是进程的标准输入（SUSv3 标准没有规定/dev/fd 特性，但许多 UNIX 实现都提供这个特性）。

打开/dev/fd 目录下的某个文件等价于复制相应的文件描述符。因此下面语句是等价的：

```
fd = open("/dev/fd/1", O_WRONLY);
fd = dup(1); /* 复制标准输出 */
```

open()调用的 flags 参数会被解析，因此我们应该小心地指定原有描述符相同

的访问模式。指定其它标志（如 O_CREAT），在这里是无意义的（被忽略）。

程序很少使用/dev/fd 目录下的文件。/dev/fd 的主要用途是 shell。许多用户级命令指定文件名参数，有时候我们需要把它们放进管道，并且设置参数之一为标准输入或标准输出。为了实现这个目的，有些程序（如 diff, ed, tar, comm）使用包含“-”的参数，来表示这个文件参数使用标准输入或标准输出代替。因此要比较 ls 的文件列表与之前创建的文件列表，我们可能会编写如下命令：

```
$ ls | diff - oldfilelist
```

这个方法有许多问题。首先，它要求每个程序对“-”进行特别的解释，但许多程序并不执行这种解析，它们只能工作于文件名参数，指定标准输入或标准输出时它们就无法工作。第二，有些程序解释“-”为标识命令行选项结尾的界定符。

使用/dev/fd 消除了这些问题，它允许把标准输入、标准输出、标准错误作为文件名指定给任何程序。因此我们可以如下重新编写上面命令：

```
$ ls | diff /dev/fd/0 oldfilelist
```

惯例上/dev/stdin, /dev/stdout, /dev/stderr 都是符号链接，分别指向/dev/fd/0, /dev/fd/1, /dev/fd/2。

## 5.12 创建临时文件

有些程序需要创建临时文件，这些文件只在程序运行时临时使用，当程序终止时应该自动删除这些文件。例如编译器在编译过程中就会创建许多临时文件。GNU C 库提供一系列库函数来创建临时文件（如此多变种的部分原因是继承许多其它 UNIX 实现的结果）。这里我们讨论其中的两个函数：mkstemp()和 tmpfile()。

mkstemp()函数根据调用方传递的模板来产生唯一的文件名，并打开该文件，然后返回一个可用于 I/O 系统调用的文件描述符。

```
#include <stdlib.h>
```

```
int mkstemp(char *template);
```
成功时返回文件描述符，出错返回-1

template 参数指定路径名，它的最后 6 个字符必须是“XXXXXX”，这 6 个字符会被替换为一个字符串，使得文件名唯一，同时替换后的字符串也通过 template 参数返回。由于 template 会被修改，它必须定义为字符数组，而不能定义为字符串常量。

mkstemp()函数为文件拥有者创建文件并设置为可读写权限（其它用户没有权限），并且以 O_EXCL 标志打开这个文件，确保调用方独占访问该文件。

通常临时文件打开之后我们会马上对其执行 unlink()系统调用（18.3 节，用来删除文件），因此我们可以如下使用 mkstemp()：

```
int fd;
char template[] = "/tmp/somestringXXXXXX";

fd = mkstemp(template);
if (fd == -1)
    errExit("mkstemp");
printf("Generated filename was: %s\n", template);
unlink(template);     /* Name disappears immediately, but the file
                         is removed only after close() */

/* Use file I/O system calls - read(), write(), and so on */

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

tmpnam(), tempnam(), mktemp()函数也可以用来生成唯一文件名。但是这些函数应该避免使用，因为它们可能导致应用产生安全漏洞。更多细节请参考这些函数的手册页。

tmpfile()函数创建一个唯一命名的临时文件，然后打开为读取和写入（这个文件以 O_EXCL 标志打开，确保其它进程不会创建相同名字的文件）。

```
#include <stdio.h>

FILE *tmpfile(void);
```
成功时返回文件指针，出错返回 NULL

成功时 tmpfile()返回文件流，可以用于其它的 stdio 库函数。临时文件在关闭时将自动删除，tmpfile()内部在打开文件后立即调用了 unlink()，来移除该文件。

## 5.13 小结

在本章的课程里，我们介绍了原子性的概念，以及其对某些系统调用正确操作至关重要的作用。特别是 open()的 O_EXCL 标志允许调用方确保自己是文件的创建者，而 open()的 O_APPEND 标志则确保多个进程向同一文件添加数据不会互相影响。

fcntl()系统调用执行一系列文件控制操作，包括改变打开文件状态标志和复制文件描述符。复制文件描述符也可以通过 dup()和 dup2()来实现。

我们查看了文件描述符、打开文件、文件 i-node 之间的关联，并且强调这三个对象所包含的信息是不同的。复制文件描述符会引用同一个打开文件，因此共享打开文件状态标志和文件偏移。

我们描述了一些能够扩展 read()和 write()功能的系统调用。pread()和 pwrite()系统调用在指定文件位置执行 I/O 操作，并且不改变文件偏移。readv()和 writev()执行 scatter-gather I/O；preadv()和 pwritev()调用则组合了 scatter-gather I/O 和指定文件位置 I/O 的功能。

truncate()和 ftruncate()系统调用可以用来减少文件大小，并丢弃超出的字节；也可以用来增加文件大小，并以 0 字节或文件空洞进行填充。

我们简短地介绍了非阻塞 I/O 的概念，后面章节会更加详细地讨论。

LFS 规范定义了一组大文件 I/O 扩展，可允许 32 位系统执行超过 32 位系统限制的大文件 I/O 操作。

/def/fd 虚拟目录下的文件允许进程通过文件描述符数值来访问自己的打开文件，这在 shell 命令中特别有用。

mkstemp()和 tmpfile()函数允许应用创建临时文件。

## 5.14 习题

5-1. 修改清单 5-3 中的程序，使用标准文件 I/O 系统调用（open()和 lseek()）以及 off_t 数据类型。把_FILE_OFFSET_BITS 宏设置为 64 然后编译程序，并测试该程序能够成功地创建大文件。

5-2. 编写一个程序，以 O_APPEND 标志打开一个现有文件，然后在每次写入数据之前 seek 到文件起始位置。写入文件中的数据会在哪里？为什么？

5-3. 这个练习用来演示为什么以 O_APPEND 标志打开文件以确保原子性是必需的。编写一个程序，它有三个命令行参数：

```
$ atomic_append filename num-bytes [x]
```

打开文件名指定的文件（如果不存在则创建），并使用 write()每次写入一个字节，一共向文件写入 num-bytes 字节。默认情况下程序要使用 O_APPEND 标志打开文件，但是如果指定了第三个命令行参数[x]，则忽略 O_APPEND 标志直接打开文件，此时文件应该在每次 write()之前调用 lseek(fd, 0, SEEK_END)。同时运行这个程序的两个实例，都不带 x 参数，向相同文件写入 1 百万字节：

```
$ atomic_append f1 1000000 & atomic_append f1 1000000
```

重复相同的步骤，让程序的两个实例写入另一个文件，但这次不使用 x 参数：

```
$ atomic_append f2 1000000 x & atomic_append f2 1000000 x
```

使用 ls –l 列出文件 f1 和 f2 的大小，并解释为何存在区别。

5-4. 使用 fcntl()（必要时也可使用 close()）来实现 dup()和 dup2()。（你可以忽略 dup2()和 fcntl()返回不同的 errno 错误值）。对于 dup2()，记住处理 oldfd 等于 newfd 的特殊情况。在这种情况下，你应该检查 oldfd 是否合法（可

以通过检查 fcntl(oldfd, F_GETFL)是否成功来实现），如果 oldfd 非法，函数应该返回-1 并设置 errno 为 EBADF。

5-5. 编写一个程序，验证复制的文件描述符共享相同的文件偏移和打开文件状态标志。

5-6. 在下面代码每次调用 write()之后，解释输出文件的内容是什么，为什么？

```
fd1 = open(file, O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
fd2 = dup(fd1);
fd3 = open(file, O_RDWR);
write(fd1, "Hello,", 6);
write(fd2, "world", 6);
lseek(fd2, 0, SEEK_SET);
write(fd1, "HELLO,", 6);
write(fd3, "Gidday", 6);
```

5-7. 使用 read()和 write()，以及 malloc 包中的适当函数（7.1.2 节），来实现 readv()和 writev()函数。

# 第 6 章 进程

在这一章，我们学习进程的组成，特别是进程虚拟内存的布局和内容。我们还讨论了一些进程的属性。在后续章节，我们会讨论更多的进程属性（如第 9 章的进程凭证，和第 35 章的进程优先级和调度）。在第 24 章到第 27 章，我们学习如何创建进程、终止进程、以及怎样执行新程序。

## 6.1 进程和程序

进程是正在执行的程序的一个实例。在这一节，我们详细讨论这个定义，并阐明程序和进程之间的区别。

程序是一个文件，包含运行时如何组织进程的一组信息。包括以下信息：

- 二进制格式标识：每个程序文件都包含一些元信息，描述该可执行文件的格式。内核根据这个信息来解释文件的剩余内容。历史上 UNIX 有两种广泛使用的可执行文件格式：原始的 a.out（“assembler output”）格式；和后来更为复杂的 COFF（Common Object File Format）格式。今天，多数 UNIX 实现（包括 Linux）都采用了可执行和链接格式（ELF），ELF 相比老的格式拥有许多优点。
- 机器语言指令：程序代码。
- 程序入口地址：这标识了程序开始执行的指令位置。
- 数据：程序文件包含的值，用来初始化变量以及字面常量（如字符串）。
- 符号和重定位表：描述了函数和变量的位置和名字。这些表有许多用途，包括调试和运行时符号解引用（动态链接）。
- 共享库和动态链接信息：程序文件包含一些域，列出了程序运行时需要使用的共享库，以及动态链接器装载这些库使用的路径。
- 其它信息：程序文件还包括许多其它信息，描述如何组织进程。

一个程序可以用来创建多个进程，或者反过来说，许多进程可以运行自同一个程序。

我们可以重新描述本章开头对进程的定义：进程是一个抽象实体，由内核定义，用来组织执行程序所分配的系统资源。

从内核的角度来看，进程由用户空间内存和许多内核数据结构组成，其中用户空间内存包含程序代码和它使用的变量；内核数据结构则维护了进程状态信息。内核记录的信息包括进程相关的许多数值标识符（ID）、虚拟内存表、打开文件描述符表、信号递送和处理信息表、进程资源使用量和限制、当前工作目录、以及其它许多信息。

## 6.2 进程 ID 和父进程 ID

每个进程都有一个进程 ID（PID），它是一个唯一标识系统中进程的正整数值。进程 ID 在一系列系统调用中使用和返回。例如 kill()系统调用（20.5 节）允许调用方向指定 ID 的进程发送一个信号。如果我们需要针对特定进程创建唯一标识，进程 ID 也非常有用。一个常见的例子是使用进程 ID 作为进程唯一文件名。

getpid()系统调用返回调用进程的进程 ID。

```
#include <unistd.h>

pid_t getpid(void);
                                                    总是成功返回调用进程 ID
```

SUSv3 规定 pid_t 数据类型为整数类型，用来存储进程 ID。

除了少数系统进程（如 init 的进程 ID 为 1），程序和进程 ID 之间没有必然联系（程序运行时的进程 ID 不会固定）。

Linux 内核限制进程 ID 小于等于 32767。当新进程创建时，会被顺序地赋予下一个可用的进程 ID。每当达到 32767 的限制，内核就重置进程 ID 计数器，因此进程 ID 又重新从最低整数值开始分配。

> 一旦达到 32767，进程 ID 计数器将重置为 300 而不是 1。这是因为许多低数值的进程 ID 已经被系统进程和 daemon 占用，搜索这个范围内未使用的进程 ID 纯粹是浪费时间。
>
> 在 Linux 2.4 及更早版本，进程 ID 的 32767 限制由内核常量 PID_MAX 定义。在 Linux 2.6 中又有了新的变化。进程 ID 默认的最大限制仍然是 32767，但这个限制现在可以通过 Linux 特定的/proc/sys/kernel/pid_max 文件来修改，这个文件的值是最大

进程 ID 的值加 1）。在 32 位平台中，这个文件的最大值是 32768；而在 64 位平台中，这个值最大可以调整为 $2^{22}$（大约 4 百万），这几乎可以满足任意数量的进程要求。

每个进程都有一个父进程（也就是创建该进程的进程）。进程可以使用 getppid()系统调用来获得父进程 ID。

```
#include <unistd.h>

pid_t getppid(void);
```

总是成功返回父进程 ID

实际上系统中所有进程的父进程属性组成了树形进程关系图。每个进程的父进程也有自己的父进程，并且依次递归最终回到进程 1（init），它是所有进程的祖先（可以使用 pstree 命令查看进程树）。

如果子进程由于父进程终止而变成“孤儿”进程，则子进程将自动由 init 进程收养，随后的 getppid()调用将返回 1（参考 26.2 节）。

任何进程的父进程都可以通过 Linux 特定的/proc/PID/status 文件中的 Ppid 域获得。

## 6.3 进程内存布局

每个进程分配的内存由许多部分组成，通常称为“段”。进程拥有如下段：

- text 段：

## 6.4 虚拟内存管理

## 6.5 堆栈和栈帧

## 6.6 命令行参数（argc, argv）

## 6.7 环境列表

## 6.8 执行非局部跳转：setjmp()和 longjmp()

## 6.9 小结

## 6.10 习题

# 第 7 章 内存分配

## 7.1 在堆上分配内存

### 7.1.1 调整 Program Break：brk()和 sbrk()

### 7.1.2 在堆上分配内存：malloc()和 free()

### 7.1.3 实现 malloc()和 free()

### 7.1.4 在堆上分配内存的其它方法

## 7.2 在栈上分配内存：alloca()

## 7.3 小结

## 7.4 习题

# 第 8 章 用户和组

8.1 密码文件：/etc/passwd

8.2 阴影密码文件：/etc/shadow

8.3 组文件：/etc/group

8.4 获取用户和组信息

8.5 密码加密和用户认证

8.6 小结

8.7 习题

# 第 9 章 进程凭证

## 9.1 真实用户 ID 和真实组 ID

## 9.2 有效用户 ID 和有效组 ID

## 9.3 设置用户 ID 和设置组 ID

## 9.4 保存的设置用户 ID 和保存的设置组 ID

## 9.5 文件系统用户 ID 和文件系统组 ID

## 9.6 补充组 ID

## 9.7 获取和修改进程凭证

### 9.7.1 获取和修改真实、有效、和保存的设置 ID

### 9.7.2 获取和修改文件系统 ID

### 9.7.3 获取和修改补充组 ID

### 9.7.4 修改进程凭证调用小结

### 9.7.5 示例：显示进程凭证

## 9.8 小结

## 9.9 习题

# 第 10 章 时间

## 10.1 日历时间

## 10.2 时间转换函数

### 10.2.1 time_t 转换为可打印格式

### 10.2.2 time_t 和 Broken-Down 时间互相转换

### 10.2.3 Broken-Down 时间和可打印格式互相转换

## 10.3 时区

## 10.4 Locale

## 10.5 更新系统时钟

## 10.6 软时钟（Jiffy）

## 10.7 进程时间

## 10.8 小结

## 10.9 习题

# 第 11 章 系统限制和选项

## 11.1 系统限制

## 11.2 运行时获取系统限制（和选项）

## 11.3 运行时获取文件相关的限制（和选项）

## 11.4 不确定限制

## 11.5 系统选项

## 11.6 小结

## 11.7 习题

# 第 12 章 系统和进程信息

## 12.1 /proc 文件系统

### 12.1.1 获取进程的信息：/proc/PID

### 12.1.2 /proc 下的系统信息

### 12.1.3 访问/proc 文件

## 12.2 系统标识：uname()

## 12.3 小结

## 12.4 习题

# 第 13 章 文件 I/O 缓冲

13.1 文件 I/O 的内核缓冲：缓冲区缓存

13.2 stdio 库的缓冲

13.3 控制文件 I/O 的内核缓冲

13.4 I/O 缓冲小结

13.5 通知内核 I/O 模型

13.6 绕过缓冲区缓存：Direct I/O

13.7 混合库函数和系统调用的文件 I/O

13.8 小结

13.9 习题

# 第 14 章 文件系统

## 14.1 设备特殊文件（设备）

## 14.2 磁盘和分区

## 14.3 文件系统

## 14.4 i-node

## 14.5 虚拟文件系统（VFS）

## 14.6 日志文件系统

## 14.7 单目录层次结构和挂载点

## 14.8 挂载和卸载文件系统

### 14.8.1 挂载文件系统：mount()

### 14.8.2 卸载文件系统：umount()和 umount2()

## 14.9 高级挂载特性

### 14.9.1 挂载文件系统至多个挂载点

### 14.9.2 堆叠多个挂载至相同挂载点

### 14.9.3 单个挂载选项的挂载标志

14.9.4 绑定挂载

14.9.5 递归绑定挂载

14.10 虚拟内存文件系统：tmpfs

14.11 获取文件系统信息：statvfs()

14.12 小结

14.13 习题

# 第 15 章 文件属性

## 15.1 获取文件信息：stat()

## 15.2 文件时间戳

### 15.2.1 改变文件时间戳：utime()和 utimes()

### 15.2.2 改变文件时间戳：utimensat()和 futimens()

## 15.3 文件所属权

### 15.3.1 新文件的所属权

### 15.3.2 改变文件所属权：chown(), fchown(), lchown()

## 15.4 文件权限

### 15.4.1 普通文件权限

### 15.4.2 目录权限

### 15.4.3 权限检查算法

### 15.4.4 检查文件可访问性：access()

### 15.4.5 设置用户 ID，设置组 ID，粘滞位

### 15.4.6 进程文件模式创建掩码：umask()

### 15.4.7 改变文件权限：chmod()和 fchmod()

## 15.5 i-node 标志（ext2 扩展文件属性）

## 15.6 小结

## 15.7 习题

# 第 16 章 扩展属性

## 16.1 概述

## 16.2 扩展属性实现细节

## 16.3 操作扩展属性的系统调用

## 16.4 小结

## 16.5 习题

# 第 17 章 访问控制列表

17.1 概述

17.2 ACL 权限检查算法

17.3 ACL 的长文本和短文本格式

17.4 ACL_MASK 入口和 ACL 组类

17.5 getfacl 和 setfacl 命令

17.6 默认 ACL 和文件创建

17.7 ACL 实现限制

17.8 ACL API

17.9 小结

17.10 习题

# 第 18 章 目录和链接

18.1 目录和（硬）链接

18.2 符号（软）链接

18.3 创建和删除（硬）链接：link()和 unlink()

18.4 文件重命名：rename()

18.5 操作符号链接：symlink()和 readlink()

18.6 创建和删除目录：mkdir()和 rmdir()

18.7 删除文件或目录：remove()

18.8 读取目录：opendir()和 readdir()

18.9 遍历文件树：nftw()

18.10 进程的当前工作目录

18.11 相对目录文件描述符操作

18.12 改变进程的根目录：chroot()

18.13 解引用路径名：realpath()

18.14 解析路径名字符串：dirname()和 basename()

## 18.15 小结

## 18.16 习题

# 第 19 章 监控文件事件

19.1 概述

19.2 inotify API

19.3 inotify 事件

19.4 读取 inotify 事件

19.5 队列限制和/proc 文件

19.6 监控文件事件的旧系统：dnotify

19.7 小结

19.8 习题

# 第 20 章 信号：基础概念

20.1 概念和概述

20.2 信号类型和默认动作

20.3 改变信号配置：signal()

20.4 信号处理器介绍

20.5 发送信号：kill()

20.6 检查进程是否存在

20.7 发送信号的其它方法：raise()和 killpg()

20.8 显示信号描述信息

20.9 信号集

20.10 信号掩码（阻塞信号递送）

20.11 未决信号

20.12 信号没有排队

20.13 改变信号配置：sigaction()

20.14 等待信号：pause()

## 20.15 小结

## 20.16 习题

# 第 21 章 信号：信号处理器

## 21.1 设计信号处理器

### 21.1.1 信号没有排队（再论）

### 21.1.2 可重入和异步信号安全的函数

### 21.1.3 全局变量和 sig_atomic_t 数据类型

## 21.2 终止信号处理器的其它方法

### 21.2.1 从信号处理器中执行非局部跳转

### 21.2.2 异常地终止进程：abort()

## 21.3 在备用堆栈中处理信号：sigaltstack()

## 21.4 SA_SIGINFO 标志

## 21.5 系统调用的中断和重启

## 21.6 小结

## 21.7 习题

# 第 22 章 信号：高级特性

22.1 Core Dump 文件

22.2 递送、配置、和处理的特殊情况

22.3 可中断和不可中断的进程睡眠状态

22.4 硬件产生的信号

22.5 同步和异步信号产生

22.6 定时和信号递送顺序

22.7 signal()的实现和可移植性

22.8 实时信号

22.8.1 发送实时信号

22.8.2 处理实时信号

22.9 使用掩码来等待信号：sigsuspend()

22.10 同步等待信号

22.11 通过文件描述符接收信号

22.12 使用信号进行进程间通信

## 22.13 早期信号 API（System V 和 BSD）

## 22.14 小结

## 22.15 习题

# 第 23 章 定时器和睡眠

## 23.1 间隔定时器

## 23.2 调度和定时器的精确度

## 23.3 设置阻塞操作的超时

## 23.4 固定间隔挂起执行（睡眠）

### 23.4.1 低精度睡眠：sleep()

### 23.4.2 高精度睡眠：nanosleep()

## 23.5 POSIX 时钟

### 23.5.1 获取时钟的值：clock_gettime()

### 23.5.2 设置时钟的值：clock_settime()

### 23.5.3 获取特定进程或线程的时钟 ID

### 23.5.4 增强的高精度睡眠：clock_nanosleep()

## 23.6 POSIX 间隔定时器

### 23.6.1 创建定时器：timer_create()

### 23.6.2 装备和解除定时器：timer_settime()

### 23.6.3 获取定时器的当前值：timer_gettime()

### 23.6.4 删除定时器：timer_delete()

### 23.6.5 通过信号通知

### 23.6.6 定时器溢出

### 23.6.7 通过线程通知

## 23.7 通过文件描述符通知的定时器：timerfd API

## 23.8 小结

## 23.9 习题

# 第 24 章 进程创建

# 第 25 章 进程结束

# 第 26 章 监控子进程

# 第 27 章 程序执行

# 第 28 章 进程创建和程序执行的更多细节

# 第 29 章 线程：介绍

# 第 30 章 线程：同步

# 第 31 章 线程：线程安全和线程存储

# 第 32 章 线程：线程取消

# 第 33 章 线程：更多细节

# 第 34 章 进程组、会话和任务控制

# 第 35 章 进程优先级和调度

# 第 36 章 进程资源

# 第 37 章 Daemon

# 第 38 章 编写安全的特权程序

# 第 39 章 能力

# 第 40 章 登录会计

# 第 41 章 共享库基础

# 第 42 章 共享库高级特性

# 第 43 章 进程间通信简介

# 第 44 章 管道和 FIFO

# 第 45 章 System V IPC 介绍

# 第 46 章 System V 消息队列

# 第 47 章 System V 信号量

# 第 48 章 System V 共享内存

# 第 49 章 内存映射

# 第 50 章 虚拟内存操作

# 第 51 章 POSIX IPC 介绍

# 第 52 章 POSIX 消息队列

# 第 53 章 POSIX 信号量

# 第 54 章 POSIX 共享内存

# 第 55 章 文件锁

# 第 56 章 Sockets：介绍

# 第 57 章 Sockets：UNIX Domain

# 第 58 章 Sockets：TCP/IP 网络基础

# 第 59 章 Sockets：Internet Domain

# 第 60 章 Sockets：服务器设计

# 第 61 章 Sockets：高级主题

# 第 62 章 终端

# 第 63 章 可选 I/O 模型

# 第 64 章 伪终端

# 附录 A：跟踪系统调用

# 附录 B：解析命令行参数

# 附录 C：转换 NULL 指针

# 附录 D：内核配置

# 附录 E：更多信息来源

# 附录 F：部分习题解答

# 参考书目

# 索引