****

**THE LINUX PROGRAMMING INTERFACE**

**A Linux and UNIX System Programming Handbook**

**MICHAEL KERRISK**

no starch press

San Francisco

*Translated by: Kevin*

***本资料仅供学习所用，请于下载后24小时内删除，否则引起的任何后果均由您自己承担。本书版权归原作者所有，如果您喜欢本书，请购买正版支持作者。***

目录

[前言 36](#_Toc303865951)

[主题 36](#_Toc303865952)

[目标读者 36](#_Toc303865953)

[Linux和UNIX 37](#_Toc303865954)

[使用和组织 37](#_Toc303865955)

[例子程序 38](#_Toc303865956)

[习题 39](#_Toc303865957)

[标准和可移植性 39](#_Toc303865958)

[Linux内核和C库版本 40](#_Toc303865959)

[其它语言使用编程接口 40](#_Toc303865960)

[关于作者 40](#_Toc303865961)

[致谢 40](#_Toc303865962)

[许可 41](#_Toc303865963)

[网站和例子程序源代码 41](#_Toc303865964)

[反馈 41](#_Toc303865965)

[第1章 历史和标准 42](#_Toc303865966)

[1.1 UNIX和C简史 42](#_Toc303865967)

[1.2 Linux简史 45](#_Toc303865968)

[1.2.1 GNU项目 46](#_Toc303865969)

[1.2.2 Linux内核 47](#_Toc303865970)

[1.3 标准化 52](#_Toc303865971)

[1.3.1 C编程语言 52](#_Toc303865972)

[1.3.2 第一个POSIX标准 53](#_Toc303865973)

[1.3.3 X/Open公司和开放组织 54](#_Toc303865974)

[1.3.4 SUSv3和POSIX.1-2001 55](#_Toc303865975)

[1.3.5 SUSv4和POSIX.1-2008 57](#_Toc303865976)

[1.3.6 UNIX标准时间线 58](#_Toc303865977)

[1.3.7 实现标准 59](#_Toc303865978)

[1.3.8 Linux、标准、和Linux标准基础 60](#_Toc303865979)

[1.4 小结 61](#_Toc303865980)

[第2章 基础概念 63](#_Toc303865981)

[2.1 操作系统的核心：内核 63](#_Toc303865982)

[2.2 Shell 66](#_Toc303865983)

[2.3 用户和组 67](#_Toc303865984)

[2.4 单一目录层次、目录、链接、和文件 68](#_Toc303865985)

[2.5 文件I/O模型 72](#_Toc303865986)

[2.6 程序 73](#_Toc303865987)

[2.7 进程 73](#_Toc303865988)

[2.8 内存映射 77](#_Toc303865989)

[2.9 静态和共享库 78](#_Toc303865990)

[2.10 进程间通信和同步 79](#_Toc303865991)

[2.11 信号 80](#_Toc303865992)

[2.12 线程 81](#_Toc303865993)

[2.13 进程组和shell工作控制 81](#_Toc303865994)

[2.14 会话、控制终端、和控制进程 82](#_Toc303865995)

[2.15 伪终端 83](#_Toc303865996)

[2.16 日期和时间 83](#_Toc303865997)

[2.17 客户端-服务器体系架构 84](#_Toc303865998)

[2.18 实时 85](#_Toc303865999)

[2.19 /proc文件系统 85](#_Toc303866000)

[2.20 小结 86](#_Toc303866001)

[第3章 系统编程概念 87](#_Toc303866002)

[3.1 系统调用 87](#_Toc303866003)

[3.2 库函数 90](#_Toc303866004)

[3.3 标准C库；GNU C库（glibc） 91](#_Toc303866005)

[3.4 系统调用和库函数的错误处理 92](#_Toc303866006)

[3.5 本书示例程序的说明 95](#_Toc303866007)

[3.5.1 命令行选项和参数 95](#_Toc303866008)

[3.5.2 常用函数和头文件 95](#_Toc303866009)

[3.6 可移植问题 106](#_Toc303866010)

[3.6.1 特性测试宏 106](#_Toc303866011)

[3.6.2 系统数据类型 109](#_Toc303866012)

[3.6.3 各种可移植问题 112](#_Toc303866013)

[3.7 小结 114](#_Toc303866014)

[3.8 习题 115](#_Toc303866015)

[第4章 文件I/O：通用I/O模型 116](#_Toc303866016)

[4.1 概述 116](#_Toc303866017)

[4.2 I/O的通用性 119](#_Toc303866018)

[4.3 打开文件：open() 119](#_Toc303866019)

[4.3.1 open()的flags参数 122](#_Toc303866020)

[4.3.2 open()的错误 126](#_Toc303866021)

[4.3.3 creat()系统调用 127](#_Toc303866022)

[4.4 读取文件：read() 128](#_Toc303866023)

[4.5 写入文件：write() 129](#_Toc303866024)

[4.6 关闭文件：close() 130](#_Toc303866025)

[4.7 改变文件偏移：lseek() 130](#_Toc303866026)

[4.8 通用I/O模型之外的操作：ioctl() 136](#_Toc303866027)

[4.9 小结 137](#_Toc303866028)

[4.10 习题 137](#_Toc303866029)

[第5章 文件I/O：更多细节 138](#_Toc303866030)

[5.1 原子性和竞争条件 138](#_Toc303866031)

[5.2 文件控制操作：fcntl() 142](#_Toc303866032)

[5.3 打开文件状态标志 142](#_Toc303866033)

[5.4 文件描述符和打开文件之间的关系 144](#_Toc303866034)

[5.5 复制文件描述符 146](#_Toc303866035)

[5.6 指定偏移位置的文件I/O：pread()和pwrite() 149](#_Toc303866036)

[5.7 Scatter-Gather I/O：readv()和writev() 150](#_Toc303866037)

[5.8 截断文件：truncate()和ftruncate() 154](#_Toc303866038)

[5.9 非阻塞I/O 155](#_Toc303866039)

[5.10 大文件I/O 155](#_Toc303866040)

[5.11 /dev/fd目录 159](#_Toc303866041)

[5.12 创建临时文件 160](#_Toc303866042)

[5.13 小结 162](#_Toc303866043)

[5.14 习题 163](#_Toc303866044)

[第6章 进程 165](#_Toc303866045)

[6.1 进程和程序 165](#_Toc303866046)

[6.2 进程ID和父进程ID 166](#_Toc303866047)

[6.3 进程内存布局 167](#_Toc303866048)

[6.4 虚拟内存管理 171](#_Toc303866049)

[6.5 堆栈和堆栈帧 174](#_Toc303866050)

[6.6 命令行参数（argc, argv） 175](#_Toc303866051)

[6.7 环境列表 178](#_Toc303866052)

[6.8 执行非局部goto：setjmp()和longjmp() 185](#_Toc303866053)

[6.9 小结 192](#_Toc303866054)

[6.10 习题 192](#_Toc303866055)

[第7章 内存分配 194](#_Toc303866056)

[7.1 在堆上分配内存 194](#_Toc303866057)

[7.1.1 调整Program Break：brk()和sbrk() 194](#_Toc303866058)

[7.1.2 在堆上分配内存：malloc()和free() 195](#_Toc303866059)

[7.1.3 实现malloc()和free() 200](#_Toc303866060)

[7.1.4 在堆上分配内存的其它方法 204](#_Toc303866061)

[7.2 在栈上分配内存：alloca() 207](#_Toc303866062)

[7.3 小结 208](#_Toc303866063)

[7.4 习题 208](#_Toc303866064)

[第8章 用户和组 209](#_Toc303866065)

[8.1 密码文件：/etc/passwd 209](#_Toc303866066)

[8.2 阴影密码文件：/etc/shadow 211](#_Toc303866067)

[8.3 组文件：/etc/group 211](#_Toc303866068)

[8.4 获取用户和组信息 213](#_Toc303866069)

[8.5 密码加密和用户认证 219](#_Toc303866070)

[8.6 小结 223](#_Toc303866071)

[8.7 习题 223](#_Toc303866072)

[第9章 进程凭证 224](#_Toc303866073)

[9.1 实际用户ID和实际组ID 224](#_Toc303866074)

[9.2 有效用户ID和有效组ID 224](#_Toc303866075)

[9.3 设置用户ID和设置组ID 225](#_Toc303866076)

[9.4 保存的设置用户ID和保存的设置组ID 227](#_Toc303866077)

[9.5 文件系统用户ID和文件系统组ID 228](#_Toc303866078)

[9.6 附加组ID 229](#_Toc303866079)

[9.7 获取和修改进程凭证 229](#_Toc303866080)

[9.7.1 获取和修改实际、有效、和保存的设置ID 229](#_Toc303866081)

[9.7.2 获取和修改文件系统ID 235](#_Toc303866082)

[9.7.3 获取和修改附加组ID 236](#_Toc303866083)

[9.7.4 修改进程凭证调用汇总 238](#_Toc303866084)

[9.7.5 示例：显示进程凭证 240](#_Toc303866085)

[9.8 小结 242](#_Toc303866086)

[9.9 习题 242](#_Toc303866087)

[第10章 时间 244](#_Toc303866088)

[10.1 日历时间 244](#_Toc303866089)

[10.2 时间转换函数 245](#_Toc303866090)

[10.2.1 time\_t转换为可打印格式 246](#_Toc303866091)

[10.2.2 time\_t和分解时间互相转换 247](#_Toc303866092)

[10.2.3 分解时间和可打印格式互相转换 249](#_Toc303866093)

[10.3 时区 257](#_Toc303866094)

[10.4 Locale 260](#_Toc303866095)

[10.5 更新系统时钟 264](#_Toc303866096)

[10.6 软时钟（Jiffy） 266](#_Toc303866097)

[10.7 进程时间 266](#_Toc303866098)

[10.8 小结 271](#_Toc303866099)

[10.9 习题 271](#_Toc303866100)

[第11章 系统限制和选项 272](#_Toc303866101)

[11.1 系统限制 273](#_Toc303866102)

[11.2 运行时获取系统限制（和选项） 277](#_Toc303866103)

[11.3 运行时获取文件相关的限制（和选项） 279](#_Toc303866104)

[11.4 不确定限制 281](#_Toc303866105)

[11.5 系统选项 282](#_Toc303866106)

[11.6 小结 284](#_Toc303866107)

[11.7 习题 285](#_Toc303866108)

[第12章 系统和进程信息 286](#_Toc303866109)

[12.1 /proc文件系统 286](#_Toc303866110)

[12.1.1 获取进程的信息：/proc/PID 287](#_Toc303866111)

[12.1.2 /proc下的系统信息 289](#_Toc303866112)

[12.1.3 访问/proc文件 291](#_Toc303866113)

[12.2 系统标识：uname() 293](#_Toc303866114)

[12.3 小结 295](#_Toc303866115)

[12.4 习题 296](#_Toc303866116)

[第13章 文件I/O缓冲 297](#_Toc303866117)

[13.1 文件I/O的内核缓冲：缓冲区缓存 297](#_Toc303866118)

[13.2 stdio库的缓冲 301](#_Toc303866119)

[13.3 控制文件I/O的内核缓冲 304](#_Toc303866120)

[13.4 I/O缓冲小结 309](#_Toc303866121)

[13.5 向内核建议I/O模型 310](#_Toc303866122)

[13.6 绕过缓冲区缓存：Direct I/O 311](#_Toc303866123)

[13.7 混合库函数和系统调用的文件I/O 314](#_Toc303866124)

[13.8 小结 316](#_Toc303866125)

[13.9 习题 316](#_Toc303866126)

[第14章 文件系统 318](#_Toc303866127)

[14.1 设备特殊文件（设备） 318](#_Toc303866128)

[14.2 磁盘和分区 320](#_Toc303866129)

[14.3 文件系统 321](#_Toc303866130)

[14.4 i-node 324](#_Toc303866131)

[14.5 虚拟文件系统（VFS） 326](#_Toc303866132)

[14.6 日志文件系统 327](#_Toc303866133)

[14.7 单目录层次结构和挂载点 329](#_Toc303866134)

[14.8 挂载和卸载文件系统 331](#_Toc303866135)

[14.8.1 挂载文件系统：mount() 332](#_Toc303866136)

[14.8.2 卸载文件系统：umount()和umount2() 339](#_Toc303866137)

[14.9 高级挂载特性 341](#_Toc303866138)

[14.9.1 挂载文件系统至多个挂载点 341](#_Toc303866139)

[14.9.2 堆叠多个挂载至相同挂载点 342](#_Toc303866140)

[14.9.3 单次挂载的挂载标志选项 342](#_Toc303866141)

[14.9.4 绑定挂载 343](#_Toc303866142)

[14.9.5 递归绑定挂载 344](#_Toc303866143)

[14.10 虚拟内存文件系统：tmpfs 346](#_Toc303866144)

[14.11 获取文件系统信息：statvfs() 347](#_Toc303866145)

[14.12 小结 349](#_Toc303866146)

[14.13 习题 350](#_Toc303866147)

[第15章 文件属性 351](#_Toc303866148)

[15.1 获取文件信息：stat() 351](#_Toc303866149)

[15.2 文件时间戳 352](#_Toc303866150)

[15.2.1 改变文件时间戳：utime()和utimes() 352](#_Toc303866151)

[15.2.2 改变文件时间戳：utimensat()和futimens() 352](#_Toc303866152)

[15.3 文件拥有者 352](#_Toc303866153)

[15.3.1 新文件的拥有者 352](#_Toc303866154)

[15.3.2 改变文件拥有者：chown(), fchown(), lchown() 353](#_Toc303866155)

[15.4 文件权限 353](#_Toc303866156)

[15.4.1 普通文件权限 353](#_Toc303866157)

[15.4.2 目录权限 353](#_Toc303866158)

[15.4.3 权限检查算法 353](#_Toc303866159)

[15.4.4 检查文件可访问性：access() 353](#_Toc303866160)

[15.4.5 设置用户ID，设置组ID，粘滞位 353](#_Toc303866161)

[15.4.6 进程文件模式创建掩码：umask() 353](#_Toc303866162)

[15.4.7 改变文件权限：chmod()和fchmod() 354](#_Toc303866163)

[15.5 i-node标志（ext2扩展文件属性） 354](#_Toc303866164)

[15.6 小结 354](#_Toc303866165)

[15.7 习题 354](#_Toc303866166)

[第16章 扩展属性 355](#_Toc303866167)

[16.1 概述 355](#_Toc303866168)

[16.2 扩展属性实现细节 355](#_Toc303866169)

[16.3 操作扩展属性的系统调用 355](#_Toc303866170)

[16.4 小结 355](#_Toc303866171)

[16.5 习题 355](#_Toc303866172)

[第17章 访问控制列表 356](#_Toc303866173)

[17.1 概述 356](#_Toc303866174)

[17.2 ACL权限检查算法 356](#_Toc303866175)

[17.3 ACL的长文本和短文本格式 356](#_Toc303866176)

[17.4 ACL\_MASK入口和ACL组类 356](#_Toc303866177)

[17.5 getfacl和setfacl命令 356](#_Toc303866178)

[17.6 默认ACL和文件创建 356](#_Toc303866179)

[17.7 ACL实现限制 356](#_Toc303866180)

[17.8 ACL API 356](#_Toc303866181)

[17.9 小结 356](#_Toc303866182)

[17.10 习题 356](#_Toc303866183)

[第18章 目录和链接 357](#_Toc303866184)

[18.1 目录和（硬）链接 357](#_Toc303866185)

[18.2 符号（软）链接 357](#_Toc303866186)

[18.3 创建和删除（硬）链接：link()和unlink() 357](#_Toc303866187)

[18.4 文件重命名：rename() 357](#_Toc303866188)

[18.5 操作符号链接：symlink()和readlink() 357](#_Toc303866189)

[18.6 创建和删除目录：mkdir()和rmdir() 357](#_Toc303866190)

[18.7 删除文件或目录：remove() 357](#_Toc303866191)

[18.8 读取目录：opendir()和readdir() 357](#_Toc303866192)

[18.9 遍历文件树：nftw() 357](#_Toc303866193)

[18.10 进程的当前工作目录 357](#_Toc303866194)

[18.11 相对目录文件描述符操作 357](#_Toc303866195)

[18.12 改变进程的根目录：chroot() 357](#_Toc303866196)

[18.13 解引用路径名：realpath() 357](#_Toc303866197)

[18.14 解析路径名字符串：dirname()和basename() 357](#_Toc303866198)

[18.15 小结 358](#_Toc303866199)

[18.16 习题 358](#_Toc303866200)

[第19章 监控文件事件 359](#_Toc303866201)

[19.1 概述 359](#_Toc303866202)

[19.2 inotify API 359](#_Toc303866203)

[19.3 inotify事件 359](#_Toc303866204)

[19.4 读取inotify事件 359](#_Toc303866205)

[19.5 队列限制和/proc文件 359](#_Toc303866206)

[19.6 监控文件事件的旧系统：dnotify 359](#_Toc303866207)

[19.7 小结 359](#_Toc303866208)

[19.8 习题 359](#_Toc303866209)

[第20章 信号：基础概念 360](#_Toc303866210)

[20.1 概念和概述 360](#_Toc303866211)

[20.2 信号类型和默认动作 360](#_Toc303866212)

[20.3 改变信号配置：signal() 360](#_Toc303866213)

[20.4 信号处理器介绍 360](#_Toc303866214)

[20.5 发送信号：kill() 360](#_Toc303866215)

[20.6 检查进程是否存在 360](#_Toc303866216)

[20.7 发送信号的其它方法：raise()和killpg() 360](#_Toc303866217)

[20.8 显示信号描述信息 360](#_Toc303866218)

[20.9 信号集 360](#_Toc303866219)

[20.10 信号掩码（阻塞信号递送） 360](#_Toc303866220)

[20.11 未决信号 360](#_Toc303866221)

[20.12 信号没有排队 360](#_Toc303866222)

[20.13 改变信号配置：sigaction() 360](#_Toc303866223)

[20.14 等待信号：pause() 360](#_Toc303866224)

[20.15 小结 361](#_Toc303866225)

[20.16 习题 361](#_Toc303866226)

[第21章 信号：信号处理器 362](#_Toc303866227)

[21.1 设计信号处理器 362](#_Toc303866228)

[21.1.1 信号没有排队（再论） 362](#_Toc303866229)

[21.1.2 可重入和异步信号安全的函数 362](#_Toc303866230)

[21.1.3 全局变量和sig\_atomic\_t数据类型 362](#_Toc303866231)

[21.2 终止信号处理器的其它方法 362](#_Toc303866232)

[21.2.1 从信号处理器中执行非局部跳转 362](#_Toc303866233)

[21.2.2 异常地终止进程：abort() 362](#_Toc303866234)

[21.3 在备用堆栈中处理信号：sigaltstack() 362](#_Toc303866235)

[21.4 SA\_SIGINFO标志 362](#_Toc303866236)

[21.5 系统调用的中断和重启 362](#_Toc303866237)

[21.6 小结 362](#_Toc303866238)

[21.7 习题 362](#_Toc303866239)

[第22章 信号：高级特性 363](#_Toc303866240)

[22.1 Core Dump文件 363](#_Toc303866241)

[22.2 递送、配置、和处理的特殊情况 363](#_Toc303866242)

[22.3 可中断和不可中断的进程睡眠状态 363](#_Toc303866243)

[22.4 硬件产生的信号 363](#_Toc303866244)

[22.5 同步和异步信号产生 363](#_Toc303866245)

[22.6 定时和信号递送顺序 363](#_Toc303866246)

[22.7 signal()的实现和可移植性 363](#_Toc303866247)

[22.8 实时信号 363](#_Toc303866248)

[22.8.1 发送实时信号 363](#_Toc303866249)

[22.8.2 处理实时信号 363](#_Toc303866250)

[22.9 使用掩码来等待信号：sigsuspend() 363](#_Toc303866251)

[22.10 同步等待信号 363](#_Toc303866252)

[22.11 通过文件描述符接收信号 363](#_Toc303866253)

[22.12 使用信号进行进程间通信 363](#_Toc303866254)

[22.13 早期信号API（System V和BSD） 364](#_Toc303866255)

[22.14 小结 364](#_Toc303866256)

[22.15 习题 364](#_Toc303866257)

[第23章 定时器和睡眠 365](#_Toc303866258)

[23.1 间隔定时器 365](#_Toc303866259)

[23.2 调度和定时器的精确度 365](#_Toc303866260)

[23.3 设置阻塞操作的超时 365](#_Toc303866261)

[23.4 固定间隔挂起执行（睡眠） 365](#_Toc303866262)

[23.4.1 低精度睡眠：sleep() 365](#_Toc303866263)

[23.4.2 高精度睡眠：nanosleep() 365](#_Toc303866264)

[23.5 POSIX时钟 365](#_Toc303866265)

[23.5.1 获取时钟的值：clock\_gettime() 365](#_Toc303866266)

[23.5.2 设置时钟的值：clock\_settime() 365](#_Toc303866267)

[23.5.3 获取特定进程或线程的时钟ID 365](#_Toc303866268)

[23.5.4 增强的高精度睡眠：clock\_nanosleep() 365](#_Toc303866269)

[23.6 POSIX间隔定时器 365](#_Toc303866270)

[23.6.1 创建定时器：timer\_create() 365](#_Toc303866271)

[23.6.2 装备和解除定时器：timer\_settime() 365](#_Toc303866272)

[23.6.3 获取定时器的当前值：timer\_gettime() 366](#_Toc303866273)

[23.6.4 删除定时器：timer\_delete() 366](#_Toc303866274)

[23.6.5 通过信号通知 366](#_Toc303866275)

[23.6.6 定时器溢出 366](#_Toc303866276)

[23.6.7 通过线程通知 366](#_Toc303866277)

[23.7 通过文件描述符通知的定时器：timerfd API 366](#_Toc303866278)

[23.8 小结 366](#_Toc303866279)

[23.9 习题 366](#_Toc303866280)

[第24章 进程创建 367](#_Toc303866281)

[24.1 fork(), exit(), wait(), execve()概述 367](#_Toc303866282)

[24.2 创建新进程：fork() 367](#_Toc303866283)

[24.2.1 父子进程文件共享 367](#_Toc303866284)

[24.2.2 fork()的内存语义 367](#_Toc303866285)

[24.3 vfork()系统调用 367](#_Toc303866286)

[24.4 fork()之后的竞争条件 367](#_Toc303866287)

[24.5 通过信号同步来避免竞争条件 367](#_Toc303866288)

[24.6 小结 367](#_Toc303866289)

[24.7 习题 367](#_Toc303866290)

[第25章 进程结束 368](#_Toc303866291)

[25.1 终止进程：\_exit()和exit() 368](#_Toc303866292)

[25.2 进程终止的细节 368](#_Toc303866293)

[25.3 Exit处理器 368](#_Toc303866294)

[25.4 fork()、stdio缓冲区、\_exit()之间的交互 368](#_Toc303866295)

[25.5 小结 368](#_Toc303866296)

[25.6 习题 368](#_Toc303866297)

[第26章 监控子进程 369](#_Toc303866298)

[26.1 等待子进程 369](#_Toc303866299)

[26.1.1 wait()系统调用 369](#_Toc303866300)

[26.1.2 waitpid()系统调用 369](#_Toc303866301)

[26.1.3 等待状态值 369](#_Toc303866302)

[26.1.4 进程从信号处理器中终止 369](#_Toc303866303)

[26.1.5 waitid()系统调用 369](#_Toc303866304)

[26.1.6 wait3()和wait4()系统调用 369](#_Toc303866305)

[26.2 孤儿进程（Orphan）和僵尸进程（Zombie） 369](#_Toc303866306)

[26.3 SIGCHLD信号 369](#_Toc303866307)

[26.3.1 创建SIGCHLD处理器 369](#_Toc303866308)

[26.3.2 子进程停止时递送SIGCHLD 370](#_Toc303866309)

[26.3.3 忽略死亡的子进程 370](#_Toc303866310)

[26.4 小结 370](#_Toc303866311)

[26.5 习题 370](#_Toc303866312)

[第27章 程序执行 371](#_Toc303866313)

[27.1 执行新程序：execve() 371](#_Toc303866314)

[27.2 exec()库函数 371](#_Toc303866315)

[27.2.1 PATH环境变量 371](#_Toc303866316)

[27.2.2 指定程序参数为列表 371](#_Toc303866317)

[27.2.3 传递调用方环境给新程序 371](#_Toc303866318)

[27.2.4 执行描述符引用的文件：fexecve() 371](#_Toc303866319)

[27.3 解释器脚本 371](#_Toc303866320)

[27.4 文件描述符和exec() 371](#_Toc303866321)

[27.5 信号和exec() 371](#_Toc303866322)

[27.6 执行shell命令：system() 371](#_Toc303866323)

[27.7 实现system() 371](#_Toc303866324)

[27.8小结 371](#_Toc303866325)

[27.9 习题 371](#_Toc303866326)

[第28章 进程创建和程序执行的更多细节 372](#_Toc303866327)

[28.1 进程会计 372](#_Toc303866328)

[28.2 clone()系统调用 372](#_Toc303866329)

[28.2.1 clone()的flags参数 372](#_Toc303866330)

[28.2.2 clone子进程的waitpid()扩展 372](#_Toc303866331)

[28.3 进程创建的速度 372](#_Toc303866332)

[28.4 exec()和fork()对进程属性的影响 372](#_Toc303866333)

[28.5 小结 372](#_Toc303866334)

[28.6 习题 372](#_Toc303866335)

[第29章 线程：介绍 373](#_Toc303866336)

[29.1 概述 373](#_Toc303866337)

[29.2 pthread API的背景细节 373](#_Toc303866338)

[29.3 线程创建 373](#_Toc303866339)

[29.4 线程终止 373](#_Toc303866340)

[29.5 线程ID 373](#_Toc303866341)

[29.6 等待线程终止 373](#_Toc303866342)

[29.7 分离线程 373](#_Toc303866343)

[29.8 线程属性 373](#_Toc303866344)

[29.9 线程VS进程 373](#_Toc303866345)

[29.10 小结 373](#_Toc303866346)

[29.11 习题 373](#_Toc303866347)

[第30章 线程：同步 374](#_Toc303866348)

[30.1 保护共享变量访问：Mutex 374](#_Toc303866349)

[30.1.1 静态分配的Mutex 374](#_Toc303866350)

[30.1.2 Mutex加锁和解锁 374](#_Toc303866351)

[30.1.3 Mutex的性能 374](#_Toc303866352)

[30.1.4 Mutex死锁 374](#_Toc303866353)

[30.1.5 动态初始化Mutex 374](#_Toc303866354)

[30.1.6 Mutex属性 374](#_Toc303866355)

[30.1.7 Mutex类型 374](#_Toc303866356)

[30.2状态变化通知：条件变量 374](#_Toc303866357)

[30.2.1 静态分配的条件变量 374](#_Toc303866358)

[30.2.2 通知和等待条件变量 374](#_Toc303866359)

[30.2.3 测试条件变量的Predicate 374](#_Toc303866360)

[30.2.4 示例程序：等待任意线程终止 374](#_Toc303866361)

[30.2.5 动态分配的条件变量 374](#_Toc303866362)

[30.3 小结 375](#_Toc303866363)

[30.4 习题 375](#_Toc303866364)

[第31章 线程：线程安全和线程存储 376](#_Toc303866365)

[31.1 线程安全（再论可重入） 376](#_Toc303866366)

[31.2 一次性初始化 376](#_Toc303866367)

[31.3 线程特定数据 376](#_Toc303866368)

[31.3.1 库函数中的线程特定数据 376](#_Toc303866369)

[31.3.2 线程特定数据API概述 376](#_Toc303866370)

[31.3.3 线程特定数据API细节 376](#_Toc303866371)

[31.3.4 使用线程特定数据API 376](#_Toc303866372)

[31.3.5 线程特定数据的实现限制 376](#_Toc303866373)

[31.4 线程本地存储 376](#_Toc303866374)

[31.5 小结 376](#_Toc303866375)

[31.6 习题 376](#_Toc303866376)

[第32章 线程：取消线程 377](#_Toc303866377)

[32.1 取消线程 377](#_Toc303866378)

[32.2 取消状态和类型 377](#_Toc303866379)

[32.3 取消点 377](#_Toc303866380)

[32.4 测试线程取消 377](#_Toc303866381)

[32.5 清理处理器 377](#_Toc303866382)

[32.6 异步取消 377](#_Toc303866383)

[32.7 小结 377](#_Toc303866384)

[第33章 线程：更多细节 378](#_Toc303866385)

[33.1 线程堆栈 378](#_Toc303866386)

[33.2 线程和信号 378](#_Toc303866387)

[33.2.1 UNIX信号模型如何映射到线程 378](#_Toc303866388)

[33.2.2 操作线程信号掩码 378](#_Toc303866389)

[33.2.3 向线程发送信号 378](#_Toc303866390)

[33.2.4 理智地处理异步信号 378](#_Toc303866391)

[33.3 线程和进程控制 378](#_Toc303866392)

[33.4 线程实现模型 378](#_Toc303866393)

[33.5 POSIX线程的Linux实现 378](#_Toc303866394)

[33.5.1 LinuxThreads 378](#_Toc303866395)

[33.5.2 NPTL 378](#_Toc303866396)

[33.5.3 选择哪个线程实现？ 378](#_Toc303866397)

[33.6 pthread API的高级特性 378](#_Toc303866398)

[33.7 小结 378](#_Toc303866399)

[33.8 习题 379](#_Toc303866400)

[第34章 进程组、会话和任务控制 380](#_Toc303866401)

[34.1 概述 380](#_Toc303866402)

[34.2 进程组 380](#_Toc303866403)

[34.3 会话 380](#_Toc303866404)

[34.4 控制终端和控制进程 380](#_Toc303866405)

[34.5 前台和后台进程组 380](#_Toc303866406)

[34.6 SIGHUP信号 380](#_Toc303866407)

[34.6.1 shell对SIGHUP的处理 380](#_Toc303866408)

[34.6.2 SIGHUP和控制进程的终止 380](#_Toc303866409)

[34.7 任务控制 380](#_Toc303866410)

[34.7.1 在shell中使用任务控制 380](#_Toc303866411)

[34.7.2 实现任务控制 380](#_Toc303866412)

[34.7.3 处理任务控制信号 380](#_Toc303866413)

[34.7.4 孤儿进程组（再论SIGHUP） 380](#_Toc303866414)

[34.8 小结 380](#_Toc303866415)

[34.9 习题 381](#_Toc303866416)

[第35章 进程优先级和调度 382](#_Toc303866417)

[35.1 进程优先级（Nice值） 382](#_Toc303866418)

[35.2 实时进程调度概述 382](#_Toc303866419)

[35.2.1 SCHED\_RR策略 382](#_Toc303866420)

[35.2.2 SCHED\_FIFO策略 382](#_Toc303866421)

[35.2.3 SCHED\_BATCH和SCHED\_IDLE策略 382](#_Toc303866422)

[35.3 实时进程调度API 382](#_Toc303866423)

[35.3.1 实时优先级范围 382](#_Toc303866424)

[35.3.2 修改和获取策略和优先级 382](#_Toc303866425)

[35.3.3 放弃 CPU 382](#_Toc303866426)

[35.3.4 SCHED\_RR时间片 382](#_Toc303866427)

[35.4 CPU亲和力 382](#_Toc303866428)

[35.5 小结 382](#_Toc303866429)

[35.6 习题 382](#_Toc303866430)

[第36章 进程资源 383](#_Toc303866431)

[36.1 进程资源使用 383](#_Toc303866432)

[36.2 进程资源限制 383](#_Toc303866433)

[36.3 特定资源限制的细节 383](#_Toc303866434)

[36.4 小结 383](#_Toc303866435)

[36.5 习题 383](#_Toc303866436)

[第37章 Daemon 384](#_Toc303866437)

[37.1 概述 384](#_Toc303866438)

[37.2 创建Daemon 384](#_Toc303866439)

[37.3 Daemon编写指南 384](#_Toc303866440)

[37.4 使用SIGHUP来重新初始化Daemon 384](#_Toc303866441)

[37.5 使用syslog记录日志和错误信息 384](#_Toc303866442)

[37.5.1 概述 384](#_Toc303866443)

[37.5.2 syslog API 384](#_Toc303866444)

[37.5.3 /etc/syslog.conf文件 384](#_Toc303866445)

[37.6 小结 384](#_Toc303866446)

[37.7 习题 384](#_Toc303866447)

[第38章 编写安全的特权程序 385](#_Toc303866448)

[38.1 是否需要设置用户ID和设置组ID的程序？ 385](#_Toc303866449)

[38.2 以最小权限执行操作 385](#_Toc303866450)

[38.3 执行程序时要小心 385](#_Toc303866451)

[38.4 避免暴露敏感信息 385](#_Toc303866452)

[38.5 限制进程 385](#_Toc303866453)

[38.6 小心信号和竞争条件 385](#_Toc303866454)

[38.7 执行文件操作和文件I/O的陷阱 385](#_Toc303866455)

[38.8 不要相信输入和环境 385](#_Toc303866456)

[38.9 小心缓冲区溢出 385](#_Toc303866457)

[38.10 小心拒绝服务攻击 385](#_Toc303866458)

[38.11 检查返回状态和安全地失败 385](#_Toc303866459)

[38.12 小结 385](#_Toc303866460)

[38.13 习题 385](#_Toc303866461)

[第39章 能力 386](#_Toc303866462)

[39.1 能力的基本原理 386](#_Toc303866463)

[39.2 Linux能力 386](#_Toc303866464)

[39.3 进程和文件能力 386](#_Toc303866465)

[39.3.1 进程能力 386](#_Toc303866466)

[39.3.2 文件能力 386](#_Toc303866467)

[39.3.3 进程允许和有效能力集的作用 386](#_Toc303866468)

[39.3.4 文件允许和有效能力集的作用 386](#_Toc303866469)

[39.3.5 进程和文件可继承能力集的作用 386](#_Toc303866470)

[39.3.6 shell中查看和赋予文件能力 386](#_Toc303866471)

[39.4 现代的能力实现 386](#_Toc303866472)

[39.5 exec()时进程能力的转化 386](#_Toc303866473)

[39.5.1 能力边界集 386](#_Toc303866474)

[39.5.2 保留root语义 386](#_Toc303866475)

[39.6 改变用户ID对进程能力的影响 386](#_Toc303866476)

[39.7 编程改变进程能力 387](#_Toc303866477)

[39.8 创建能力唯一环境 387](#_Toc303866478)

[39.9 发现程序所需的能力 387](#_Toc303866479)

[39.10 没有文件能力的老内核和系统 387](#_Toc303866480)

[39.11 小结 387](#_Toc303866481)

[39.12 习题 387](#_Toc303866482)

[第40章 登录会计 388](#_Toc303866483)

[40.1 utmp和wtmp文件概述 388](#_Toc303866484)

[40.2 utmpx API 388](#_Toc303866485)

[40.3 utmpx结构体 388](#_Toc303866486)

[40.4 从utmp和wtmp文件中获取信息 388](#_Toc303866487)

[40.5 获取登录名称：getlogin() 388](#_Toc303866488)

[40.6 为登录会话更新utmp和wtmp文件 388](#_Toc303866489)

[40.7 lastlog文件 388](#_Toc303866490)

[40.8 小结 388](#_Toc303866491)

[40.9 习题 388](#_Toc303866492)

[第41章 共享库基础 389](#_Toc303866493)

[41.1 对象库 389](#_Toc303866494)

[41.2 静态库 389](#_Toc303866495)

[41.3 共享库概述 389](#_Toc303866496)

[41.4 创建和使用共享库 – A First Pass 389](#_Toc303866497)

[41.4.1 创建共享库 389](#_Toc303866498)

[41.4.2 位置无关的代码 389](#_Toc303866499)

[41.4.3 使用共享库 389](#_Toc303866500)

[41.4.4 共享库Soname 389](#_Toc303866501)

[41.5 使用共享库的有用工具 389](#_Toc303866502)

[41.6 共享库版本和命名惯例 389](#_Toc303866503)

[41.7 安装共享库 389](#_Toc303866504)

[41.8 兼容VS不兼容库 389](#_Toc303866505)

[41.9 升级共享库 389](#_Toc303866506)

[41.10 在对象文件中指定库搜索目录 389](#_Toc303866507)

[41.11 运行时查找共享库 390](#_Toc303866508)

[41.12 运行时符号解析 390](#_Toc303866509)

[41.13 使用静态库而不是共享库 390](#_Toc303866510)

[41.14 小结 390](#_Toc303866511)

[41.15 习题 390](#_Toc303866512)

[第42章 共享库高级特性 391](#_Toc303866513)

[42.1 动态装载库 391](#_Toc303866514)

[42.1.1 打开共享库：dlopen() 391](#_Toc303866515)

[42.1.2 诊断错误：dlerror() 391](#_Toc303866516)

[42.1.3 获取符号地址：dlsym() 391](#_Toc303866517)

[42.1.4 关闭共享库：dlclose() 391](#_Toc303866518)

[42.1.5 获取已装载符号的信息：dladdr() 391](#_Toc303866519)

[42.1.6 在主程序中访问符号 391](#_Toc303866520)

[42.2 控制符号可见性 391](#_Toc303866521)

[42.3 链接器版本脚本 391](#_Toc303866522)

[42.3.1 使用版本脚本控制符号可见性 391](#_Toc303866523)

[42.3.2 符号版本 391](#_Toc303866524)

[42.4 初始化和终止化函数 391](#_Toc303866525)

[42.5 预装载共享库 391](#_Toc303866526)

[42.6 监控动态链接器：LD\_DEBUG 391](#_Toc303866527)

[42.7 小结 392](#_Toc303866528)

[42.8 习题 392](#_Toc303866529)

[第43章 进程间通信概述 393](#_Toc303866530)

[43.1 IPC机制分类 393](#_Toc303866531)

[43.2 通信机制 393](#_Toc303866532)

[43.3 同步机制 393](#_Toc303866533)

[43.4 IPC机制对比 393](#_Toc303866534)

[43.5 小结 393](#_Toc303866535)

[43.6 习题 393](#_Toc303866536)

[第44章 管道和FIFO 394](#_Toc303866537)

[44.1 概述 394](#_Toc303866538)

[44.2 创建和使用管道 394](#_Toc303866539)

[44.3 管道作为进程同步的方法 394](#_Toc303866540)

[44.4 使用管道连接过滤器 394](#_Toc303866541)

[44.5 使用管道与shell命令交互：popen() 394](#_Toc303866542)

[44.6 管道和stdio缓冲区 394](#_Toc303866543)

[44.7 FIFO 394](#_Toc303866544)

[44.8 使用FIFO的客户端-服务器应用 394](#_Toc303866545)

[44.9 非阻塞I/O 394](#_Toc303866546)

[44.10 管道和FIFO的read()和write()语义 394](#_Toc303866547)

[44.11 小结 394](#_Toc303866548)

[44.12 习题 394](#_Toc303866549)

[第45章 System V IPC介绍 395](#_Toc303866550)

[45.1 API概述 395](#_Toc303866551)

[45.2 IPC Key 395](#_Toc303866552)

[45.3 相关的数据结构和对象权限 395](#_Toc303866553)

[45.4 IPC标识符和客户端-服务器应用 395](#_Toc303866554)

[45.5 System V IPC被调用时采用的算法 395](#_Toc303866555)

[45.6 ipcs和ipcrm命令 395](#_Toc303866556)

[45.7 获取所有IPC对象列表 395](#_Toc303866557)

[45.8 IPC的限制 395](#_Toc303866558)

[45.9 小结 395](#_Toc303866559)

[45.10 习题 395](#_Toc303866560)

[第46章 System V消息队列 396](#_Toc303866561)

[46.1 创建或打开消息队列 396](#_Toc303866562)

[46.2 交换消息 396](#_Toc303866563)

[46.2.1 发送消息 396](#_Toc303866564)

[46.2.2 接收消息 396](#_Toc303866565)

[46.3 消息队列控制操作 396](#_Toc303866566)

[46.4 消息队列相关的数据结构 396](#_Toc303866567)

[46.5 消息队列的限制 396](#_Toc303866568)

[46.6 显示系统中所有消息队列 396](#_Toc303866569)

[46.7 客户端-服务器消息队列编程 396](#_Toc303866570)

[46.8 使用消息队列的文件服务器应用 396](#_Toc303866571)

[46.9 System V消息队列的缺点 396](#_Toc303866572)

[46.10 小结 396](#_Toc303866573)

[46.11 习题 396](#_Toc303866574)

[第47章 System V信号量 397](#_Toc303866575)

[47.1 概述 397](#_Toc303866576)

[47.2 创建或打开信号量 397](#_Toc303866577)

[47.3 信号量控制操作 397](#_Toc303866578)

[47.4 信号量相关的数据结构 397](#_Toc303866579)

[47.5 信号量初始化 397](#_Toc303866580)

[47.6 信号量操作 397](#_Toc303866581)

[47.7 处理多个阻塞的信号量操作 397](#_Toc303866582)

[47.8 信号量撤消值 397](#_Toc303866583)

[47.9 实现二进制信号量协议 397](#_Toc303866584)

[47.10 信号量的限制 397](#_Toc303866585)

[47.11 System V信号量的缺点 397](#_Toc303866586)

[47.12 小结 397](#_Toc303866587)

[47.13 习题 397](#_Toc303866588)

[第48章 System V共享内存 398](#_Toc303866589)

[48.1 概述 398](#_Toc303866590)

[48.2 创建或打开共享内存段 398](#_Toc303866591)

[48.3 使用共享内存 398](#_Toc303866592)

[48.4 示例：通过共享内存传输数据 398](#_Toc303866593)

[48.5 共享内存在虚拟内存中的位置 398](#_Toc303866594)

[48.6 在共享内存中存储指针 398](#_Toc303866595)

[48.7 共享内存控制操作 398](#_Toc303866596)

[48.8 共享内存相关的数据结构 398](#_Toc303866597)

[48.9 共享内存的限制 398](#_Toc303866598)

[48.10 小结 398](#_Toc303866599)

[48.11 习题 398](#_Toc303866600)

[第49章 内存映射 399](#_Toc303866601)

[49.1 概述 399](#_Toc303866602)

[49.2 创建映射：mmap() 399](#_Toc303866603)

[49.3 解除映射区域：munmap() 399](#_Toc303866604)

[49.4 文件映射 399](#_Toc303866605)

[49.4.1 私有文件映射 399](#_Toc303866606)

[49.4.2 共享文件映射 399](#_Toc303866607)

[49.4.3 边界情况 399](#_Toc303866608)

[49.4.4 内存保护和文件访问模式的相互作用 399](#_Toc303866609)

[49.5 同步映射区域：msync() 399](#_Toc303866610)

[49.6 额外的mmap()标志 399](#_Toc303866611)

[49.7 匿名映射 399](#_Toc303866612)

[49.8 重新映射区域：mremap() 399](#_Toc303866613)

[49.9 MAP\_NORESERVE和交换空间过量使用 399](#_Toc303866614)

[49.10 MAP\_FIXED标志 399](#_Toc303866615)

[49.11 非线性映射：remap\_file\_pages() 400](#_Toc303866616)

[49.12 小结 400](#_Toc303866617)

[49.13 习题 400](#_Toc303866618)

[第50章 虚拟内存操作 401](#_Toc303866619)

[50.1 改变内存保护：mprotect() 401](#_Toc303866620)

[50.2 内存锁：mlock()和mlockall() 401](#_Toc303866621)

[50.3 确定内存所在：mincore() 401](#_Toc303866622)

[50.4 建议未来的内存使用模式：madvise() 401](#_Toc303866623)

[50.5 小结 401](#_Toc303866624)

[50.6 习题 401](#_Toc303866625)

[第51章 POSIX IPC介绍 402](#_Toc303866626)

[51.1 API概述 402](#_Toc303866627)

[51.2 比较System V IPC和POSIX IPC 402](#_Toc303866628)

[51.3 小结 402](#_Toc303866629)

[第52章 POSIX消息队列 403](#_Toc303866630)

[52.1 概述 403](#_Toc303866631)

[52.2 打开、关闭、删除消息队列 403](#_Toc303866632)

[52.3 描述符和消息队列的关联 403](#_Toc303866633)

[52.4 消息队列属性 403](#_Toc303866634)

[52.5 交换消息 403](#_Toc303866635)

[52.5.1 发送消息 403](#_Toc303866636)

[52.5.2 接收消息 403](#_Toc303866637)

[52.5.3 带超时的消息发送和接收 403](#_Toc303866638)

[52.6 消息通知 403](#_Toc303866639)

[52.6.1 通过信号接收通知 403](#_Toc303866640)

[52.6.2 通过线程接收通知 403](#_Toc303866641)

[52.7 Linux特定特性 403](#_Toc303866642)

[52.8 消息队列的限制 403](#_Toc303866643)

[52.9 比较POSIX和System V消息队列 403](#_Toc303866644)

[52.10 小结 404](#_Toc303866645)

[52.11 习题 404](#_Toc303866646)

[第53章 POSIX信号量 405](#_Toc303866647)

[53.1 概述 405](#_Toc303866648)

[53.2 命名信号量 405](#_Toc303866649)

[53.2.1 打开命名信号量 405](#_Toc303866650)

[53.2.2 关闭信号量 405](#_Toc303866651)

[53.2.3 删除命名信号量 405](#_Toc303866652)

[53.3 信号量操作 405](#_Toc303866653)

[53.3.1 等待信号量 405](#_Toc303866654)

[53.3.2 公告信号量 405](#_Toc303866655)

[53.3.3 获取信号量的当前值 405](#_Toc303866656)

[53.4 未命名信号量 405](#_Toc303866657)

[53.4.1 初始化未命名信号量 405](#_Toc303866658)

[53.4.2 销毁未命名信号量 405](#_Toc303866659)

[53.5 与其它同步技术对比 405](#_Toc303866660)

[53.6 信号量的限制 405](#_Toc303866661)

[53.7 小结 406](#_Toc303866662)

[53.8 习题 406](#_Toc303866663)

[第54章 POSIX共享内存 407](#_Toc303866664)

[54.1 概述 407](#_Toc303866665)

[54.2 创建共享内存对象 407](#_Toc303866666)

[54.3 使用共享内存对象 407](#_Toc303866667)

[54.4 删除共享内存对象 407](#_Toc303866668)

[54.5 各种共享内存API的对比 407](#_Toc303866669)

[54.6 小结 407](#_Toc303866670)

[54.7 习题 407](#_Toc303866671)

[第55章 文件锁 408](#_Toc303866672)

[55.1 概述 408](#_Toc303866673)

[55.2 flock()文件锁 408](#_Toc303866674)

[55.2.1 锁继承和释放的语义 408](#_Toc303866675)

[55.2.2 flock()的限制 408](#_Toc303866676)

[55.3 fcntl()记录锁 408](#_Toc303866677)

[55.3.1 死锁 408](#_Toc303866678)

[55.3.2 示例：交互式的锁程序 408](#_Toc303866679)

[55.3.3 示例：锁函数库 408](#_Toc303866680)

[55.3.4 锁限制和性能 408](#_Toc303866681)

[55.3.5 锁继承和释放的语义 408](#_Toc303866682)

[55.3.6 锁饥饿和排队锁请求的优先级 408](#_Toc303866683)

[55.4 强制锁 408](#_Toc303866684)

[55.5 /proc/locks文件 408](#_Toc303866685)

[55.6 只运行程序的一个实例 408](#_Toc303866686)

[55.7 老的锁技术 409](#_Toc303866687)

[55.8 小结 409](#_Toc303866688)

[55.9 习题 409](#_Toc303866689)

[第56章 Sockets：介绍 410](#_Toc303866690)

[56.1 概述 410](#_Toc303866691)

[56.2 创建Socket：socket() 410](#_Toc303866692)

[56.3 绑定Socket到地址：bind() 410](#_Toc303866693)

[56.4 通用Socket地址结构体：struct sockaddr 410](#_Toc303866694)

[56.5 流Socket 410](#_Toc303866695)

[56.5.1 监听进来的连接：listen() 410](#_Toc303866696)

[56.5.2 接受连接：accept() 410](#_Toc303866697)

[56.5.3 连接到端Socket：connect() 410](#_Toc303866698)

[56.5.4 流Socket I/O 410](#_Toc303866699)

[56.5.5 终止连接：close() 410](#_Toc303866700)

[56.6 数据报Socket 411](#_Toc303866701)

[56.6.1 交换数据报：recvfrom()和sendto() 411](#_Toc303866702)

[56.6.2 对数据报Socket使用connect() 411](#_Toc303866703)

[56.7 小结 411](#_Toc303866704)

[第57章 Sockets：UNIX Domain 412](#_Toc303866705)

[57.1 UNIX Domain Socket地址：struct sockaddr\_un 412](#_Toc303866706)

[57.2 UNIX Domain中的流Socket 412](#_Toc303866707)

[57.3 UNIX Domain中的数据报Socket 412](#_Toc303866708)

[57.4 UNIX Domain Socket权限 412](#_Toc303866709)

[57.5 创建一对互连的Socket：socketpair() 412](#_Toc303866710)

[57.6 Linux抽象Socket命名空间 412](#_Toc303866711)

[57.7 小结 412](#_Toc303866712)

[57.8 习题 412](#_Toc303866713)

[第58章 Sockets：TCP/IP网络基础 413](#_Toc303866714)

[58.1 因特网 413](#_Toc303866715)

[58.2 网络协议和分层 413](#_Toc303866716)

[58.3 数据链路层 413](#_Toc303866717)

[58.4 网络层：IP 413](#_Toc303866718)

[58.5 IP地址 413](#_Toc303866719)

[58.6 传输层 413](#_Toc303866720)

[58.6.1 端口号 413](#_Toc303866721)

[58.6.2 用户数据报协议（UDP） 413](#_Toc303866722)

[58.6.3 传输控制协议（TCP） 413](#_Toc303866723)

[58.7 Requests For Comments（RFC） 413](#_Toc303866724)

[58.8 小结 413](#_Toc303866725)

[第59章 Sockets：Internet Domain 414](#_Toc303866726)

[59.1 Internet Domain Socket 414](#_Toc303866727)

[59.2 网络字节序 414](#_Toc303866728)

[59.3 数据表示 414](#_Toc303866729)

[59.4 Internet Socket地址 414](#_Toc303866730)

[59.5 主机和服务转换函数概述 414](#_Toc303866731)

[59.6 inet\_pton()和inet\_ntop()函数 414](#_Toc303866732)

[59.7 客户端-服务器例子（数据报Socket） 414](#_Toc303866733)

[59.8 域名系统（DNS） 414](#_Toc303866734)

[59.9 /etc/services文件 414](#_Toc303866735)

[59.10 协议无关的主机和服务转换 414](#_Toc303866736)

[59.10.1 getaddrinfo()函数 414](#_Toc303866737)

[59.10.2 释放addrinfo列表：freeaddrinfo() 414](#_Toc303866738)

[59.10.3 诊断错误：gai\_strerror() 414](#_Toc303866739)

[59.10.4 getnameinfo()函数 414](#_Toc303866740)

[59.11 客户端-服务器例子（流Socket） 415](#_Toc303866741)

[59.12 一个Internet Domain Socket库 415](#_Toc303866742)

[59.13 已废弃的主机和服务转换API 415](#_Toc303866743)

[59.13.1 inet\_aton()和inet\_ntoa()函数 415](#_Toc303866744)

[59.13.2 gethostbyname()和gethostbyaddr()函数 415](#_Toc303866745)

[59.13.3 getservbyname()和getservbyport()函数 415](#_Toc303866746)

[59.14 UNIX vs Internet Domain Socket 415](#_Toc303866747)

[59.15 更多信息 415](#_Toc303866748)

[59.16 小结 415](#_Toc303866749)

[59.17 习题 415](#_Toc303866750)

[第60章 Sockets：服务器设计 416](#_Toc303866751)

[60.1 迭代和并发服务器 416](#_Toc303866752)

[60.2 迭代UDP echo服务器 416](#_Toc303866753)

[60.3 并发TCP echo服务器 416](#_Toc303866754)

[60.4 其它并发服务器设计 416](#_Toc303866755)

[60.5 inetd（Internet Superserver）Daemon 416](#_Toc303866756)

[60.6 小结 416](#_Toc303866757)

[60.7 习题 416](#_Toc303866758)

[第61章 Sockets：高级主题 417](#_Toc303866759)

[61.1 流Socket的部分读取和写入 417](#_Toc303866760)

[61.2 shutdown()系统调用 417](#_Toc303866761)

[61.3 Socket特定的I/O系统调用：recv()和send() 417](#_Toc303866762)

[61.4 sendfile()系统调用 417](#_Toc303866763)

[61.5 获取Socket地址 417](#_Toc303866764)

[61.6 深入TCP 417](#_Toc303866765)

[61.6.1 TCP段的格式 417](#_Toc303866766)

[61.6.2 TCP序号和确认 417](#_Toc303866767)

[61.6.3 TCP状态机和状态转换图 417](#_Toc303866768)

[61.6.4 TCP建立连接 417](#_Toc303866769)

[61.6.5 TCP终止连接 417](#_Toc303866770)

[61.6.6 对TCP Socket调用shutdown() 417](#_Toc303866771)

[61.6.7 TIME\_WAIT状态 417](#_Toc303866772)

[61.7 监控Socket：netstat 417](#_Toc303866773)

[61.8 使用tcpdump来监控TCP流量 418](#_Toc303866774)

[61.9 Socket选项 418](#_Toc303866775)

[61.10 SO\_REUSEADDR Socket选项 418](#_Toc303866776)

[61.11 多个accept()的标志和选项继承 418](#_Toc303866777)

[61.12 TCP vs UDP 418](#_Toc303866778)

[61.13 高级特性 418](#_Toc303866779)

[61.13.1 带外（Out-of-Band）数据 418](#_Toc303866780)

[61.13.2 sendmsg()和recvmsg()系统调用 418](#_Toc303866781)

[61.13.3 传递文件描述符 418](#_Toc303866782)

[61.13.4 获取发送方凭证 418](#_Toc303866783)

[61.13.5 顺序包Socket 418](#_Toc303866784)

[61.13.6 SCTP和DCCP传输层协议 418](#_Toc303866785)

[61.14 小结 418](#_Toc303866786)

[61.15 习题 418](#_Toc303866787)

[第62章 终端 419](#_Toc303866788)

[62.1 概述 419](#_Toc303866789)

[62.2 获取和修改终端属性 419](#_Toc303866790)

[62.3 stty命令 419](#_Toc303866791)

[62.4 终端特殊字符 419](#_Toc303866792)

[62.5 终端标志 419](#_Toc303866793)

[62.6 终端I/O模式 419](#_Toc303866794)

[62.6.1 Canonical模式 419](#_Toc303866795)

[62.6.2 非Canonical模式 419](#_Toc303866796)

[62.6.3 Cooked, Cbreak, Raw模式 419](#_Toc303866797)

[62.7 终端行速（Bit Rate） 419](#_Toc303866798)

[62.8 终端行控制 419](#_Toc303866799)

[62.9 终端窗口大小 419](#_Toc303866800)

[62.10 终端标识 419](#_Toc303866801)

[62.11 小结 419](#_Toc303866802)

[62.12 习题 420](#_Toc303866803)

[第63章 可选I/O模型 421](#_Toc303866804)

[63.1 概述 421](#_Toc303866805)

[63.1.1 Level触发和Edge触发通知 421](#_Toc303866806)

[63.1.2 通过可选I/O模型使用非阻塞I/O 421](#_Toc303866807)

[63.2 I/O多路复用 421](#_Toc303866808)

[63.2.1 select()系统调用 421](#_Toc303866809)

[63.2.2 poll()系统调用 421](#_Toc303866810)

[63.2.3 什么时候文件描述符准备好？ 421](#_Toc303866811)

[63.2.4 比较select()和poll() 421](#_Toc303866812)

[63.2.5 select()和poll()的问题 421](#_Toc303866813)

[63.3 信号驱动I/O 421](#_Toc303866814)

[63.3.1 什么时候通知“I/O可用”？ 421](#_Toc303866815)

[63.3.2 信号驱动I/O使用精要 421](#_Toc303866816)

[63.4 epoll API 421](#_Toc303866817)

[63.4.1 创建epoll实例：epoll\_create() 421](#_Toc303866818)

[63.4.2 修改epoll兴趣列表：epoll\_ctl() 422](#_Toc303866819)

[63.4.3 等待事件：epoll\_wait() 422](#_Toc303866820)

[63.4.4 深入epoll语义 422](#_Toc303866821)

[63.4.5 epoll及I/O复用的性能对比 422](#_Toc303866822)

[63.4.6 Edge触发通知 422](#_Toc303866823)

[63.5 等待信号和文件描述符 422](#_Toc303866824)

[63.5.1 pselect()系统调用 422](#_Toc303866825)

[63.5.2 Self-Pipe技巧 422](#_Toc303866826)

[63.6 小结 422](#_Toc303866827)

[63.7 习题 422](#_Toc303866828)

[第64章 伪终端 423](#_Toc303866829)

[64.1 概述 423](#_Toc303866830)

[64.2 UNIX 98 伪终端 423](#_Toc303866831)

[64.2.1 打开未使用Master：posix\_openpt() 423](#_Toc303866832)

[64.2.2 修改Slave拥有者和权限：grantpt() 423](#_Toc303866833)

[64.2.3 解锁Slave：unlockpt() 423](#_Toc303866834)

[64.2.4 获取Slave的名字：ptsname() 423](#_Toc303866835)

[64.3 打开Master：ptyMasterOpen() 423](#_Toc303866836)

[64.4 连接进程到伪终端：ptyFork() 423](#_Toc303866837)

[64.5 伪终端I/O 423](#_Toc303866838)

[64.6 实现script(1) 423](#_Toc303866839)

[64.7 终端属性和窗口大小 423](#_Toc303866840)

[64.8 BSD伪终端 423](#_Toc303866841)

[64.9 小结 423](#_Toc303866842)

[64.10 习题 423](#_Toc303866843)

[附录A：跟踪系统调用 425](#_Toc303866844)

[附录B：解析命令行参数 426](#_Toc303866845)

[附录C：转换NULL指针 427](#_Toc303866846)

[附录D：内核配置 428](#_Toc303866847)

[附录E：更多信息来源 429](#_Toc303866848)

[附录F：部分习题解答 430](#_Toc303866849)

[参考书目 431](#_Toc303866850)

[索引 432](#_Toc303866851)

# 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](mailto: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：
   1. 把寄存器的值保存到内核堆栈中（6.5节）。
   2. 检查系统调用数值的有效性。
   3. 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。
   4. 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。
   5. 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，但不能是数组类型 | POSIX消息队列描述符 |
| 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  O\_WRONLY  O\_RDWR | 打开为只能读取  打开为只能写入  打开为同时读写 | v3  v3  v3 |
| O\_CLOEXEC  O\_CREAT  O\_DIRECT  O\_DIRECTORY  O\_EXCL  O\_LARGEFILE  O\_NOATIME  O\_NOCTTY  O\_NOFOLLOW  O\_TRUNC | 设置“close-on-exec”标志（Linux 2.6.23开始）  如果文件不存在则创建  文件I/O绕过缓冲区缓存  如果pathname不是目录则失败  和O\_CREAT一起使用：创建唯一的文件  在32位系统中打开大文件  read()时不更新文件的最后访问时间（Linux 2.6.8开始）  不让pathname成为控制终端  不解引用符号链接  截断现有文件为0长度 | v4  v3  v4  v3  v3  v4  v3 |
| O\_APPEND  O\_ASYNC  O\_DSYNC  O\_NONBLOCK  O\_SYNC | 写入总是追加至文件末尾  I/O可操作时产生一个信号  提供同步的I/O数据完整性（Linux 2.6.33开始）  非阻塞模式打开  使文件写入同步 | v3  v3  v3  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231-1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263-1。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. 这个习题用来演示为什么以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222（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：包含进程运行程序的机器语言指令。text段只读，因此进程不能通过坏指针值来修改自己的指令。由于许多进程可能运行同一个程序，于是text段被设置为可共享，一份程序代码可以被映射到所有进程的虚拟地址空间中。
* 已初始化数据段：包含已经显式初始化的全局和静态变量。在程序装载到内存中时从可执行文件中读取这些变量的值。
* 未初始化数据段：包含没有显式初始化的全局和静态变量。在程序开始运行之前，系统把这个段中的所有内存初始化为0。由于历史原因，这个段常被称为bss段，这个名字起源于老的汇编器速记“block started by symbol”。把已初始化和未初始化的全局和静态变量放在不同的段中，主要原因是程序存储在磁盘中时，未初始化数据是不需要分配空间的。相反，可执行文件只需要记录未初始化数据段的位置和大小，由程序装载器在运行时分配这个空间。
* 堆栈：是动态增长和缩小的包含堆栈帧的段。当前调用的每个函数都会分配一个堆栈帧。帧存储了函数的本地变量（自动变量）、参数、和返回值。6.5节会更加详细地讨论堆栈帧。
* 堆：是可以在运行时（为变量）动态分配内存的区域。堆的顶端被称为program break。

已初始化和未初始化数据段也称为“用户初始化数据段”和“零初始化数据段”，虽然描述性更强，不过较少使用。

size命令显示二进制可执行文件的text段、已初始化数据段、未初始化数据段（bss）的大小。

清单6-1显示了各种类型的C变量，并以注释指明了该变量所在的段。这些注释假设未使用编译器优化，应用二进制接口的所有参数也都传递至堆栈。在实践中，优化的编译器可能会把常用变量分配在寄存器中，或者把某个变量完全优化掉。此外，某些ABI要求函数参数和返回值通过寄存器而不是堆栈来传递。无论如何，这个例子说明了C变量和进程段之间的映射。

*清单6-1：程序变量在进程内存段中的位置*

--------------------------------------------------proc/mem\_segments.c

#include <stdio.h>

#include <stdlib.h>

char globBuf[65536]; /\* 未初始化数据段 \*/

int primes[] = { 2, 3, 5, 7 }; /\* 已初始化数据段 \*/

static int

square(int x) /\* 分配在square()的堆栈帧中 \*/

{

int result; /\* 分配在square()的堆栈帧中 \*/

result = x \* x;

return result; /\* 返回值通过寄存器传递 \*/

}

static void

doCalc(int val) /\* 分配在doCalc()的堆栈帧中 \*/

{

printf("The square of %d is %d\n", val, square(val));

if (val < 1000) {

int t; /\* 分配在doCalc()的堆栈帧中 \*/

t = val \* val \* val;

printf("The cube of %d is %d\n", val, t);

}

}

int

main(int argc, char \*argv[]) /\* 分配在main()的堆栈帧中 \*/

{

static int key = 9973; /\* 已初始化数据段 \*/

static char mbuf[10240000]; /\* 未初始化数据段 \*/

char \*p; /\* 分配在main()的堆栈帧中 \*/

p = malloc(1024); /\* 指向堆中的内存 \*/

doCalc(key);

exit(EXIT\_SUCCESS);

}

--------------------------------------------------proc/mem\_segments.c

尽管SUSv3并没有规定，多数UNIX实现（包括Linux）中的C程序环境都提供三个全局符号：etext, edata, end。这些符号可以用来在程序中分别获得程序text, 已初始化数据段末尾，未初始化数据段末尾的下一个字节地址。要使用这些符号，我们需要如下显式地声明它们：

extern char etext, edata, end;

/\* For example, &etext gives the address of the end

of the program text / start of initialized data \*/

图6-1显示了x86-32体系架构下各个内存段的分布。图片顶部标记为argv, environ的空间保存了程序命令行参数（C中可以使用main()函数的argv参数）和进程环境列表（马上就会讨论）。不同的内核配置和程序链接选项，会导致这个图中的二进制地址可能变化。灰色区域表示进程虚拟地址空间中的非法区域；也就是还没有创建页表的区域（参考下面关于虚拟内存管理的讨论）。

我们在48.5节会更加详细地讨论进程内存布局，在那里我们将考虑共享内存和共享库在进程的虚拟内存中如何放置。



*图6-1：Linux/x86-32典型的进程内存布局*

## 6.4 虚拟内存管理

前面讨论了进程的内存布局，但我们实际上讨论的是虚拟内存的布局。由于理解虚拟内存对于我们后续的许多主题非常有帮助（如fork()系统调用、共享内存、映射文件等），现在让我们来讨论一些虚拟内存的细节。

和多数现代内核一样，Linux也采用了虚拟内存管理技术。这个技术的主要目的是通过利用多数程序的一个典型特点：引用的局部性，来高效地使用CPU和RAM（物理内存）。多数程序都拥有两种局部性：

* 空间局部性：是程序倾向于访问最近访问过的地址附近的内存（由于指令的顺序处理，以及有时候数据结构的顺序处理）。
* 时间局部性：是程序倾向于在不久的将来还将访问刚刚访问过的相同的内存地址（由于循环）。

引用局部性的结果就是在执行程序时，可以只在RAM中维护程序的部分地址空间。

虚拟内存设计将每个程序使用的内存划分为固定大小的很小单元，称为“页”。相应地RAM也被划分为一系列相同大小的页帧。在任何时候，只需要程序的某些页驻留在物理内存页帧中；这些页就组成了驻留集。程序不使用的页拷贝维护在swap区域中（磁盘中的一个保留区域，用来补充计算机的RAM），并且只有在需要时才会被装载进物理内存中。当进程引用一个当前没有驻留在物理内存中的页时，就会触发page fault，这时候内核挂起进程的执行，同时把页从磁盘中装载进内存。

在x86-32中，页大小为4096字节。某些其它Linux实现使用更大的页大小。例如Alpha的页大小为8192字节，而IA-64则拥有可变的页大小，通常默认为16384字节。程序可以通过调用sysconf(\_SC\_PAGESIZE)来确定系统虚拟内存页大小，11.2节会详细讨论。

为了支持这样的组织结构，内核为每个进程维护了页表（图6-2）。页表描述了进程虚拟地址空间中（进程可用的所有虚拟内存页）的每个页的位置。页表中的每个条目要么表示虚拟页在RAM中的位置，要么表示虚拟页目前在磁盘中。



*图6-2：虚拟内存概观*

不是所有的进程虚拟地址空间的地址范围都需要页表项。通常大部分虚拟地址空间都未使用，因此没有必要维护相应的页表项。如果进程试图访问没有相应页表项的地址，就会接收到一个SIGSEGV信号。

进程的合法虚拟地址范围会随着生命周期而变化，内核会为进程分配和取消页（以及页表项）。例如以下这些情况：

* 由于堆栈向下增长超过之前达到的限制；
* 当在堆中分配和释放内存时，使用brk(), sbrk(), 或malloc系列函数提高program break（第7章）；
* 使用shmat()和shmdt()来附加和分离System V共享内存区域（第48章）；
* 使用mmap()和munmap()创建和解除内存映射（第49章）。

实现虚拟内存要求硬件支持分页内存管理单元（PMMU）。PMMU转化每个虚拟内存地址引用到相应的物理内存地址，以及在虚拟内存地址相应的页不在RAM中时向内核产生一个page fault。

虚拟内存管理分离了进程的虚拟地址空间和RAM的物理地址空间。这样做有许多优点：

* 进程之间以及与内核全部隔离，因此一个进程不能读取和修改另一个进程（或内核）的内存。通过使每个进程的页表项指向不同的物理页面（或交换区域）来实现。
* 当需要时，两个或多个进程可以共享内存。内核通过使不同进程的页表项指向相同RAM页面来实现。内存共享有两种常见的场景：
  + 多个进程执行相同程序，可以共享一个（只读的）程序代码拷贝。这种类型的共享在多个进程执行相同程序文件时会隐式地自动执行（或者装载相同的共享库时）。
  + 进程可以使用shmget()和mmap()系统调用来显式地请求与其它进程共享内存区域。主要目的是实现进程间通信。
* 促进了内存保护的实现。页表项可以打上标记，来表示相应的页内容可读、可写、可执行、或其它组合。当多个进程共享RAM页面时，可以对该内存为不同进程指定不同的保护；例如一个进程对该页只读访问，另一个则拥有读写访问。
* 程序员以及许多工具（如编译器和链接器），不需要关注程序在RAM中的物理布局。
* 由于只有部分程序需要驻留在内存中，程序装载和运行会更加快速。此外，进程的内存占用（虚拟大小）也可以大大超过RAM的容量。

虚拟内存管理的最后一个优点是由于每个进程都使用了更少的RAM，因此同时可以有更多的进程运行在RAM中。这通常会提高CPU的使用率，因为这样增加了任何时候CPU都至少有一个进程可以执行的可能性。

## 6.5 堆栈和堆栈帧

堆栈随着函数调用和返回自动线性增长和缩小。对于x86-32体系架构下的Linux来说（以及多数其它Linux和UNIX实现），堆栈存在于内存的高端位置并且向下增长（朝向堆）。有一个特殊的寄存器：堆栈指针，记录了当前堆栈的顶部。每次调用函数时，都会在堆栈上分配一个帧，这个帧在函数返回时删除。

尽管堆栈向下增长，我们仍然称为堆栈顶部增长，因为抽象地讲实事就是这样。堆栈的实际增长方向是（硬件）实现相关的。HP PA-RISC的Linux实现就使用了向上增长的堆栈。

从虚拟内存的术语来讲，堆栈段随着分配栈帧而增大，但是在多数实现中，这些帧删除时不会减小堆栈的大小（这些内存在新的堆栈帧分配时重用）。当我们讨论堆栈段增长和缩小时，我们将从逻辑角度来考虑，帧会在堆栈中增加和删除。

有时候术语用户堆栈会用来区分我们下面讨论的内核堆栈。内核堆栈是内核为每个进程在内核内存中维护的一段内存区域，用作执行系统调用时内部函数执行所使用的堆栈（内核不能使用用户堆栈，因为后者驻留在未保护用户内存中）。

每个（用户）堆栈帧都包含以下信息：

* 函数参数和本地变量：在C语言中，这些称为自动变量，因为它们在函数调用时自动创建，当函数返回时则自动删除（因为堆栈帧不存在了），这也是自动和静态（以及全局）变量的主要语义区别：后者永久地存在，并且与执行函数无关。
* 调用链接信息：每个函数都需要使用寄存器，如程序计数器，指向下一条要执行的机器语言指令。每次一个函数调用另一个函数时，这些寄存器的拷贝都会保存在调用函数的堆栈帧中，这样当函数返回时，调用函数才能适当地恢复原先的寄存器值。

由于函数可以调用另一个函数，堆栈上可能会存在多个帧（如果一个函数递归地调用自己，堆栈上也会有多个帧）。考虑清单6-1的代码，在square()函数执行过程中，堆栈将包含图6-3中所示的帧。



*图6-3：进程堆栈示例*

## 6.6 命令行参数（argc, argv）

每个C程序都有一个main()函数，这是程序开始执行的入口点。当程序执行时，可以通过main()函数的两个参数来访问命令行参数。第一个参数int argc表示命令行参数的个数。第二个参数char \*argv[]是一个命令行参数的指针数组，每个指针都是null结束的字符串。第一个字符串argv[0]惯例上是程序本身的名字。argv中的指针列表由一个NULL指针终止（也就是说argv[argc]等于NULL）。

由于argv[0]包含了调用程序的名字，这一点可以用来实现一个有用的技巧。我们可以对同一个程序创建多个链接，然后让程序查看argv[0]，根据不同的程序名字来执行不同的动作。gzip, gunzip, zcat命令就是采用了这个技术，这三个命令其实都链接到同一个可执行文件。（如果我们采用了这个技术，我们必须小心地处理用户可能会用其它非预期的名字来调用程序的可能性）。

图6-4显示了执行清单6-2中的程序时，argc和argv关联的数据结构。在这个图中，我们显示了每个字符串最后的null终止字节（\0）。



*图6-4：necho hello world命令的argc和argv值*

清单6-2中的程序显示自己的命令行参数，每个参数一行输出，前面字符串表示了当前打印的是哪个参数。

*清单6-2：打印命令行参数*

-----------------------------------------------------------proc/necho.c

#include "tlpi\_hdr.h"

int

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

{

int j;

for (j = 0; j < argc; j++)

printf("argv[%d] = %s\n", j, argv[j]);

exit(EXIT\_SUCCESS);

}

-----------------------------------------------------------proc/necho.c

由于argv列表由NULL终止，我们可以把清单6-2中的程序代码改写为如下形式，仅仅在每行输出命令行参数：

char \*\*p;

for (p = argv; \*p != NULL; p++)

puts(\*p);

argc/argv机制的一个局限是这些变量只在main()函数中作为参数可用。要可移植地使命令行参数在其它函数中也可用，我们要么将argv作为参数传递给这些函数，要么设置一个全局变量指向argv。

也有一些不可移植的方法，可以在程序的任何位置访问部分或所有这些信息：

* 任何进程的命令行参数都可以通过Linux特定的/proc/PID/cmdline文件读取到，每个参数都以null字节终止。（程序可以通过/proc/self/cmdline来访问自己的命令行参数）。
* GNU C库提供两个全局变量，可以在程序的任何位置用来获取调用程序的名字（也就是命令行的第一个参数）。第一个全局变量是program\_invocation\_name，是调用程序的完整路径名；第二个全局变量是program\_invocation\_short\_name，是程序的简短名字（不包含任何目录前缀）。这两个变量的声明可以通过包含头文件<errno.h>，并定义\_GNU\_SOURCE宏来获得。

如图6-1所示，argv和environ数组，以及它们指向的字符串，都存放在一个连续的内存区域中，就在进程堆栈的上方（我们在下一节讨论environ环境列表）。这个区域能够存储的总字节数有一个上限。SUSv3规定使用ARG\_MAX常量（定义在<limits.h>中），或者调用sysconf(\_SC\_ARG\_MAX)来确定这个上限（我们在11.2节讨论sysconf()）。SUSv3要求ARG\_MAX至少\_POSIX\_ARG\_MAX字节（4096）。多数UNIX实现都允许高出4096许多的值。至于实现如何计算ARG\_MAX限制和多余字节（如null终止字节，对齐字节，以及argv和environ指针数组），SUSv3没有进行定义。

在Linux中，ARG\_MAX历史上固定为32页（Linux/x86-32上就是131072字节），并且包括了多余字节的空间。从内核2.6.23开始，argv和environ能够使用的总空间可以使用RLIMIT\_STACK资源限制来控制，而且允许argv和environ使用大出许多的限制值。这个限制在execve()调用时强制计算为RLIMIT\_STACK软资源限制的四分之一。更多细节请参考execve()手册页。

许多程序（包括本书的几个例子）使用getopt()库函数来解析命令行选项（以“-”开始的参数）。我们在附录B讨论getopt()函数。

## 6.7 环境列表

每个进程都有一个关联的字符串数组，称为环境列表，或者简单地称为环境。每个字符串的格式都是name=value。因此环境描述了一组name-value，可以用来保存任何信息。列表中的name通常称为环境变量。

当新进程创建时，会继承父进程环境的一份拷贝。这是最原始但也是常用的进程间通信方式——环境提供了一种父进程向子进程传递信息的机制。由于子进程在创建时获得父进程的环境拷贝，这种信息传递是单向并且单次的。在子进程创建完成之后，父子进程都可以修改自己的环境，而且这些修改不会互相影响。

环境变量的常见用途是shell，通过把值放在shell的环境中，shell可以确保这些值会被传递到自己执行用户命令而创建的进程中。例如环境变量SHELL设置为shell程序本身的路径。许多程序解析这个变量，用在程序需要执行shell时使用的shell路径名。

某些库函数允许设置环境变量来修改自己的行为。这允许用户控制使用这些函数的应用的行为，而无需修改应用的代码，也不需要重新链接相应的库。一个典型的例子是getopt()函数（附录B），它的行为可以通过设置POSIXLY\_CORRECT环境变量来修改。

在多数shell中，可以使用export命令来添加环境值：

$ SHELL=/bin/bash 创建一个shell变量

$ export SHELL 把变量放到shell进程的环境中

在bash和Korn shell中，可以简化为：

$ export SHELL=/bin/bash

在C shell中，则需要使用setenv命令：

% setenv SHELL /bin/bash

上面这些命令永久地增加一个值到shell的环境中，然后这个环境被shell创建的所有子进程继承。在任何时候，都可以使用unset命令（C shell中使用unsetenv）删除一个环境变量。

在Bourne shell及其后裔（如bash和Korn shell）中，下面语法可以用来为单次执行程序增加环境变量，而不影响shell（以及后续命令）的环境：

$ NAME=value program

这样就只为program程序增加环境变量，如果需要，也可以在程序名前面使用多个赋值语句（空格分隔）。

printenv命令显示当前环境列表。下面是它的示例输出：

$ printenv

LOGNAME=mtk

SHELL=/bin/bash

HOME=/home/mtk

PATH=/usr/local/bin:/usr/bin:/bin:.

TERM=xterm

我们会在后续章节适当的时候说明上面多数环境变量的作用（也可以参考environ手册页）。

从上面的输出，我们可以看到环境列表并没有排序；列表中字符串的顺序由最方便的实现方式而定。通常这不会导致问题，因为我们一般只是访问环境中的单个变量，而不是访问顺序的一组变量。

任何进程的环境列表都可以通过Linux特定的/proc/PID/environ文件获得，每个NAME=value字符串都以null字节终止。

**程序中访问环境**

在C程序中，可以通过全局变量char \*\*environ来访问环境列表。（C运行时启动代码定义了这个变量，并向它赋予了环境列表的位置）。和argv一样，environ也指向一个NULL终止的指针列表，每个指针指向一个null终止的字符串。图6-5显示了上面printenv命令的环境列表数据结构。



*图6-5：进程环境列表数据结构示例*

清单6-3中的程序访问了environ，列出进程环境的所有值。这个程序可以获得和printenv命令一样的输出。程序中的循环依赖于使用指针遍历environ。虽然我们也可以把environ视作数组（和清单6-2使用argv一样），但这样做不如指针遍历自然，因为环境列表中的项目并没有特定的顺序，也没有变量（对应于argc）来指定环境列表的大小。（由于类似的原因，我们没有在图6-5中标注environ数组的元素个数）。

*清单6-3：显示进程环境*

----------------------------------------------------proc/display\_env.c

#include "tlpi\_hdr.h"

extern char \*\*environ;

int

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

{

char \*\*ep;

for (ep = environ; \*ep != NULL; ep++)

puts(\*ep);

exit(EXIT\_SUCCESS);

}

----------------------------------------------------proc/display\_env.c

另一种访问环境列表的方法是在main()函数中声明第三个参数：

int main(int argc, char \*argv[], char \*envp[])

可以把第三个参数envp当成environ一样处理，不过区别在于envp的作用域只在main()函数中。尽管这个特性在UNIX系统已经被广泛实现，我们应该避免使用它，因为除了作用域的局限，SUSv3也没有对其进行规范。

getenv()函数从进程环境中获取单个值。

|  |
| --- |
| #include <stdlib.h>  char \*getenv(const char \*name);  返回环境值字符串指针，如果变量不存在则返回NULL |

给定一个环境变量的名字，getenv()返回一个指针指向相应的值字符串。因此在我们之前的环境示例中，如果指定name参数为SHELL，则返回/bin/bash。如果指定name的环境变量不存在，getenv()返回NULL。

使用getenv()时请注意以下可移植性问题：

* SUSv3规定应用不应该修改getenv()返回的字符串。因为这个字符串实际上是环境的一部分（name=value字符串中的value部分）。（多数实现都是这样做的）。如果我们需要修改环境变量的值，我们可以使用setenv()或putenv()函数（下面讨论）。
* SUSv3允许getenv()实现返回静态分配的缓冲区字符串，因此后续getenv(), setenv(), putenv(), unsetenv()调用都可能覆盖该缓冲区。尽管glibc没有使用静态缓冲区来实现getenv()，可移植程序如果需要保存getenv()返回的字符串，就应该在下一次调用这些函数之前，将字符串复制到另一个缓冲区中。

**修改环境**

有时候进程需要修改自己的环境，比如修改环境后使其对随后创建的所有子进程都可见；另一种可能是我们希望设置一个变量，使其对将要装载到本进程内存的新程序也可见（“exec”），在这种情况下，环境就不仅仅是进程间通信机制，还是程序间通信的一种方式（这一点在第27章将会更加清晰，我们在那里详细解释exec()函数怎样允许在同一个进程中使用新程序替换自己）。

putenv()函数增加一个新变量到调用进程的环境，或者修改现有变量的值。

|  |
| --- |
| #include <stdlib.h>  int putenv(char \*string);  成功时返回0；出错返回非0 |

string参数是一个字符串指针，格式为name=value。在调用putenv()之后，这个字符串就成为环境的一部分。换句话说，putenv()并不会复制string指向的字符串，而是直接将environ的某个元素设置为指向string的位置。因此如果我们随后修改了string指向的字节，也会直接影响到进程的环境。由于这个原因，string不能定义为自动变量（堆栈中分配的字符串数组），因为这样的内存区域在函数返回后可能会被覆盖。

注意putenv()在出错时返回非0，而不是返回-1。

glibc的putenv()实现提供一个非标准的扩展。如果string不包含等号（=），将会从环境列表中删除string指定的环境变量。

setenv()函数是另一个增加变量到环境的方法，是putenv()的替代方案。

|  |
| --- |
| #include <stdlib.h>  int setenv(const char \*name, const char \*value, int overwrite);  成功时返回0；出错返回-1 |

setenv()函数通过为name=value分配内存缓冲区，来创建一个新的环境变量，并将name和value指向的字符串复制到该缓冲区中。注意我们不需要（实际上必须不）在name末尾或value开头提供等号，因为setenv()在将环境变量添加到环境列表中时会自动添加等号字符。

如果name指定的变量已经存在，并且overwrite的值为0，setenv()函数不会改变环境；如果overwrite非0，则总是改变环境。

setenv()复制参数的事实，意味着和putenv()不一样，我们随后修改name和value指向的字符串内容，不会影响环境。同时也意味着setenv()的参数使用自动变量不会引起任何问题。

unsetenv()函数从环境中删除name指定的变量。

|  |
| --- |
| #include <stdlib.h>  int unsetenv(const char \*name);  成功时返回0；出错时返回-1 |

和setenv()一样，name不能包含等号。

setenv()和unsetenv()都源自BSD，不如putenv()使用广泛。原始的POSIX.1标准和SUSv2都没有规定，但SUSv3最终包含了这两个函数的定义。

在glibc 2.2.2之前的版本，unsetenv()的原型定义为返回void。这也是早期BDS实现对unsetenv()采用的原型定义，而且某些UNIX实现目前仍然保持这种BSD原型。

有时候我们需要清除整个环境，然后再重新创建。例如我们可能要以安全方式执行设置用户ID的程序（38.8节）。我们可以通过赋值environ为NULL来清除整个环境：

environ = NULL;

这也正是clearenv()库函数执行的操作。

|  |
| --- |
| #define \_BSD\_SOURCE /\* Or: #define \_SVID\_SOURCE \*/  #include <stdlib.h>  int clearenv(void)  成功时返回0；出错时返回非0 |

在某些情况下，使用setenv()和clearenv()可能导致程序的内存泄漏。我们在上面已经说过setenv()会分配内存缓冲区，然后放进环境中。当我们调用clearenv()时，它并不会释放这个缓冲区（也没办法做到，因为clearenv()并不知道缓冲区的存在）。程序反复地调用这两个函数就可能会大量地泄漏内存。在实践中一般不会有问题，因为程序通常只在启动时调用clearenv()一次，以删除从前辈（也就是调用exec执行自己的那个程序）那里继承过来的所有环境。

许多UNIX实现都提供clearenv()，但是SUSv3并没有规定。SUSv3规定如果应用直接修改environ，正如clearenv()所做的那样，则setenv(), unsetenv(), 和getenv()的行为是未定义的。（阻止依从应用直接修改环境，能够允许实现完全控制自己实现环境变量所使用的数据结构）。SUSv3允许应用清除环境的唯一途径是获得环境变量的完整列表（获取environ的所有名字），然后调用unsetenv()逐个地删除环境变量。

**示例程序**

清单6-4演示了本节讨论的所有函数的使用。程序首先清除环境，然后把所有命令行参数添加到环境中。如果命令行没有指定GREET变量，程序会增加这个变量定义；删除名为BYE的变量定义；最后打印当前环境列表。下面是这个程序运行时产生的输出示例：

$ ./modify\_env "GREET=Guten Tag" SHELL=/bin/bash BYE=Ciao

GREET=Guten Tag

SHELL=/bin/bash

$ ./modify\_env SHELL=/bin/sh BYE=byebye

SHELL=/bin/sh

GREET=Hello world

如果我们把environ赋为NULL（清单6-4中调用clearenv()），则下面循环将会失败，因为\*environ是非法的：

for (ep = environ; \*ep != NULL; ep++)

puts(\*ep);

不过如果setenv()或putenv()发现environ是NULL，它们会创建一个新的环境列表并将environ指向它，这样上面循环就可以正确执行了。

*清单6-4：修改进程环境*

-----------------------------------------------------proc/modify\_env.c

#define \_GNU\_SOURCE /\* To get various declarations from <stdlib.h> \*/

#include <stdlib.h>

#include "tlpi\_hdr.h"

extern char \*\*environ;

int

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

{

int j;

char \*\*ep;

clearenv(); /\* Erase entire environment \*/

for (j = 1; j < argc; j++)

if (putenv(argv[j]) != 0)

errExit("putenv: %s", argv[j]);

if (setenv("GREET", "Hello world", 0) == -1)

errExit("setenv");

unsetenv("BYE");

for (ep = environ; \*ep != NULL; ep++)

puts(\*ep);

exit(EXIT\_SUCCESS);

}

-----------------------------------------------------proc/modify\_env.c

## 6.8 执行非局部goto：setjmp()和longjmp()

setjmp()和longjmp()库函数执行非局部goto。非局部术语表示goto的目标是当前执行函数之外的某个位置。

和许多编程语言一样，C也有goto语句，滥用goto会导致阅读和维护程序非常困难，不过少数情况下也可以使程序更加简单和快速。

C语言goto的一个限制是不能从当前函数跳到另一个函数，不过有时候这种功能会非常有用。考虑以下错误处理的常见场景：在一个嵌套层次非常深的函数调用中，我们遇到了一个错误，需要放弃当前任务来处理错误，我们需要从多个函数调用中返回，然后在某个高层函数（可能是main()）中继续执行。要实现这个功能，我们可以让每个函数都返回一个状态值，由调用方检查并适当地进行处理。在很多情况下，这是处理这种问题很好的办法。但是某些情况下如果我们能从嵌套函数中直接跳转到某个高层的调用函数，那代码就会简单许多。这正是setjmp()和longjmp()所提供的功能。

|  |
| --- |
| #include <setjmp.h>  int setjmp(jmp\_buf env);  初始调用时返回0；通过longjmp()返回时函数返回非0  void longjmp(jmp\_buf env, int val); |

调用setjmp()建立一个目标，用于后面longjmp()执行跳转。建立的这个目标也正是在程序中setjmp()调用发生的地方。从编程的角度来看，在调用longjmp()之后，就好像我们又第二次从setjmp()调用中返回。我们通过setjmp()返回的整数值来区分初始返回和第二次“返回”。setjmp()初始返回0，而后面的“伪”返回则是longjmp()函数的val参数所指定的值。通过使用不同的val参数值，我们就可以从程序的不同位置跳转到相同目标，并加以区分。

由于longjmp()的参数val指定为0，会导致setjmp()的“伪”返回无法区别于初始返回。因此如果val指定为0，则longjmp()总是使用1代替。

两个函数使用的env参数是允许跳转能够完成的胶水。setjmp()调用保存许多当前进程环境的信息到env中。随后longjmp()调用也必须指定相同的env变量来执行“伪”返回。由于setjmp()和longjmp()调用处于不同的函数（否则我们就直接使用goto了），env需要定义为全局变量，或者通过函 数参数进行传递。

除了其它一些信息，调用setjmp()时env还保存了程序计数器寄存器以及堆栈指针寄存器的拷贝。这些信息允许随后longjmp()调用完成两个关键步骤：

* 调用longjmp()的函数，和之前调用setjmp()的函数，二者之间所有函数的栈帧将被丢弃。这个过程有时候被称为“unwinding the stack”，通过重置堆栈指针寄存器为env参数中保存的值来完成。
* 重置程序计数器寄存器，这样程序才能从初始setjmp()调用的位置继续执行。同样这也是通过env中保存的值来完成的。

**示例程序**

清单6-5的程序演示了setjmp()和longjmp()的使用。这个程序使用setjmp()初始调用设置了一个跳转目标。随后的switch检查setjmp()返回值以确定是初始调用还是longjmp()调用返回。当返回值为0时（意味着初始调用），我们调用f1()，然后根据argc的值，要么立即调用longjmp()，要么继续调用f2()。如果进入f2()函数，则立即调用longjmp()。不管是哪个longjmp()，都会把我们跳转回setjmp()的位置。我们在两个longjmp()调用中使用了不同的val参数，因此main()函数中的switch语句可以确定是哪个函数跳转过来的，并打印一条恰当的消息。

当我们不带任何参数运行清单6-5的程序时，输出如下：

$ ./longjmp

Calling f1() after initial setjmp()

We jumped back from f1()

指定命令行参数则会导致从f2()中跳转：

$ ./longjmp x

Calling f1() after initial setjmp()

We jumped back from f2()

*清单6-5：演示setjmp()和longjmp()的使用*

--------------------------------------------------------proc/longjmp.c

#include <setjmp.h>

#include "tlpi\_hdr.h"

static jmp\_buf env;

static void

f2(void)

{

longjmp(env, 2);

}

static void

f1(int argc)

{

if (argc == 1)

longjmp(env, 1);

f2();

}

int

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

{

switch (setjmp(env)) {

case 0: /\* This is the return after the initial setjmp() \*/

printf("Calling f1() after initial setjmp()\n");

f1(argc); /\* Never returns... \*/

break; /\* ... but this is good form \*/

case 1:

printf("We jumped back from f1()\n");

break;

case 2:

printf("We jumped back from f2()\n");

break;

}

exit(EXIT\_SUCCESS);

}

--------------------------------------------------------proc/longjmp.c

**使用setjmp()的限制**

SUSv3和C99规定只能在下面上下文中调用setjmp()：

* 作为选择或迭代语句的完整控制表达式（if, switch, while等）；
* 作为一元!（非）操作符的操作数，同时这个表达式则作为选择或迭代语句的完整控制表达式；
* 作为比较操作（==, !=, <, 等等）的一部分，其它操作数是整数常量表达式，并且组合起来的表达式必须是选择或迭代语句的完整控制表达式；
* 作为独立的函数调用，不嵌入到任何表达式中。

注意C赋值语句并不在上面列表中。下面代码没有遵循标准规定：

s = setjmp(env); /\* WRONG! \*/

规定以上限制主要是因为setjmp()实现为传统函数，这样就不能保证拥有足够的信息，来为setjmp()所在表达式保存所有寄存器和临时堆栈位置，longjmp()调用没有办法正确地执行还原操作。因此setjmp()只允许在不需要临时存储的简单表达式中调用。

**滥用longjmp()**

如果env缓冲区定义为全局变量（典型做法），则可能会出现以下执行步骤：

1. 函数x()使用setjmp()建立一个跳转目标，并保存在全局变量env中。
2. 函数x()返回退出。
3. 函数y()使用env调用longjmp()。

这是一个严重错误！我们不能使用longjmp()跳转到一个已经返回的函数。想想longjmp()对堆栈执行的操作（它试图回绕并不存在的堆栈帧），这样操作的结果明显是错误的。如果我们比较幸运，我们的程序会立即崩溃。不过根据当前堆栈的状态，也可能会有其它结果，包括无穷调用-返回循环，甚至是程序貌似真的跳转到一个已经返回的函数。（在多线程程序中存在类似的longjmp()滥用，那就是在不同线程中调用longjmp()）。

SUSv3规定在嵌套信号处理器中调用longjmp()，则程序的行为是未定义的。

**优化型编译器的问题**

优化型编译器可能会重新排列程序的指令顺序，并且存储某些变量到CPU寄存器而不是RAM中。这种优化通常依赖于运行时控制流，并直接反映到程序的文法结构。由于setjmp()和longjmp()执行的跳转操作在运行时建立和执行，并且不映射到程序的文法结构，编译器的优化器在执行优化时没有办法把它们考虑进来。此外某些ABI实现的语义要求longjmp()恢复之前setjmp()保存的CPU寄存器。这意味着优化后的变量可能在调用longjmp()之后出现错误的值。我们可以通过清单6-6中的例子来检验这种行为。

*清单6-6：演示编译器优化与longjmp()的相互影响*

----------------------------------------------------proc/setjmp\_vars.c

#include <stdio.h>

#include <stdlib.h>

#include <setjmp.h>

static jmp\_buf env;

static void

doJump(int nvar, int rvar, int vvar)

{

printf("Inside doJump(): nvar=%d rvar=%d vvar=%d\n", nvar, rvar, vvar);

longjmp(env, 1);

}

int

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

{

int nvar;

register int rvar; /\* Allocated in register if possible \*/

volatile int vvar; /\* See text \*/

nvar = 111;

rvar = 222;

vvar = 333;

if (setjmp(env) == 0) { /\* Code executed after setjmp() \*/

nvar = 777;

rvar = 888;

vvar = 999;

doJump(nvar, rvar, vvar);

} else { /\* Code executed after longjmp() \*/

printf("After longjmp(): nvar=%d rvar=%d vvar=%d\n", nvar,

rvar, vvar);

}

exit(EXIT\_SUCCESS);

}

----------------------------------------------------proc/setjmp\_vars.c

当我们正常地编译上面程序时，可以看到预期的输出：

$ cc -o setjmp\_vars setjmp\_vars.c

$ ./setjmp\_vars

Inside doJump(): nvar=777 rvar=888 vvar=999

After longjmp(): nvar=777 rvar=888 vvar=999

但是当我们使用优化编译，就会得到下面非预期的结果：

$ cc -O -o setjmp\_vars setjmp\_vars.c

$ ./setjmp\_vars

Inside doJump(): nvar=777 rvar=888 vvar=999

After longjmp(): nvar=111 rvar=222 vvar=999

我们可以看到在调用longjmp()之后，nvar和rvar都被重置为调用setjmp()时的值。这是因为优化器执行的代码重新组织，在调用longjmp()之后造成了混淆。任何局部变量都是优化的候选者，因此都可能遇到这个问题；这包括指针变量和所有简单类型变量，如char, int, float, long等。

我们可以通过定义变量为volatile来阻止代码的重新组织，定义为volatile将通知优化器不进行优化。在前面程序中，当我们使用volatile定义变量vvar时，即使是优化编译程序，也能得到正确的结果。

因为不同的编译器执行不同类型的优化，可移植程序应该对setjmp()函数使用的所有本地变量采用volatile关键字。

如果我们对GNU C编译器指定-Wextra（额外警告信息）选项，则编译setjmp\_vars.c程序时会产生以下有用的警告信息：

$ cc -Wall -Wextra -O -o setjmp\_vars setjmp\_vars.c

setjmp\_vars.c: In function `main':

setjmp\_vars.c:17: warning: variable `nvar' might be clobbered by `longjmp' or `vfork'

setjmp\_vars.c:18: warning: variable `rvar' might be clobbered by `longjmp' or `vfork'

**尽量避免setjmp()和longjmp()**

如果说goto能够使程序难以阅读，则非局部goto的严重程度将数倍于此，因为它可以在程序中任意两个函数间转移控制。由于这个原因，我们应该谨慎地使用setjmp()和longjmp()。通常改进设计和编码来避免这两个函数的使用是值得的，因为程序将更加可读，而且更加可移植。当我们讨论信号时，还将再次学习这些函数的变种（sigsetjmp()和siglongjmp()，21.2.1节讨论），有时候对于编写信号处理器是非常有用的。

## 6.9 小结

每个进程有唯一的进程ID，并维护了父进程ID的一条记录。

进程的虚拟内存被逻辑地划分为若干段：text，已初始化和未初始化数据段，堆栈，堆。

堆栈包含一系列帧，每次函数调用时增加几个帧，每次函数返回时删除几个帧。每个帧都包含一个函数调用的局部变量、函数参数、调用链接等信息。

程序调用时提供的命令行参数可以通过main()函数的argc和argv参数来访问。惯例上argv[0]包含程序名。

每个进程都会获得父进程环境列表的一份拷贝，由name-value组成。进程可以通过全局变量environ和许多库函数来访问和修改环境变量。

setjmp()和longjmp()函数提供非局部goto的能力，可以从一个函数跳转到另一个函数（unwinding the stack）。为了避免编译器优化的问题，我们需要在使用这两个函数时定义变量为volatile。非局部goto可能导致程序难以阅读和维护，应该尽量避免使用。

## 6.10 习题

6-1. 编译清单6-1中的程序（mem\_segments.c），并且使用ls –l列出其大小。尽管程序包含一个数组（mbuf）大约10MB大小，但可执行文件的大小却远小于此，为什么？

6-2. 编写一个程序，检验调用longjmp()跳转到一个已经返回的函数时，会发生什么。

6-3. 使用getenv()和putenv()，以及直接修改environ，来实现setenv()和unsetenv()。unsetenv()应该检查环境变量是否存在多重定义，并且全部移除（这也是glibc的unsetenv()的行为）。

# 第7章 内存分配

许多系统程序需要为动态数据结构（如链表和二叉树）分配内存，而这些内存分配依赖的信息只能在运行时获得。本章描述在堆和堆栈上分配内存的函数。

## 7.1 在堆上分配内存

进程可以通过增加堆的大小来分配内存，堆是进程虚拟内存中紧靠着未初始化数据段的一个连续的大小可变的段，并且随着内存分配和释放而增大和缩小（119页的图6-1）。堆的当前界线称为program break。

C程序通常使用malloc家族函数来分配内存，我们马上会讨论。但是我们首先描述brk()和sbrk()，它们是malloc系列函数的基础。

### 7.1.1 调整Program Break：brk()和sbrk()

改变堆的大小（分配或释放内存）实际上只是简单地通知内核调整进程的program break。最初program break正好越过未初始化数据段的末尾（&end相同的位置，如图6-1所示）。

在增加program break之后，程序就可以访问新分配区域内的任何地址，但是此时还没有分配物理内存页面。内核在进程首次访问这些页面的地址时自动分配新的物理页面。

传统的UNIX系统提供两个系统调用来操作program break，这两个系统调用在Linux上也可用：brk()和sbrk()。尽管程序很少直接使用这些系统调用，理解它们能够让我们更清楚地认识内存分配的工作原理。

|  |
| --- |
| #include <unistd.h>  int brk(void \*end\_data\_segment);  成功时返回0；出错返回-1  void \*sbrk(intptr\_t increment);  成功时返回之前的program break；出错返回(void\*) -1 |

brk()系统调用设置program break为end\_data\_segment指定的位置。由于虚拟内存以页为单位进行分配，end\_data\_segment会自动舍入到下一页的边界。

试图设置program break小于它的初始值（低于&end）通常都会导致未预料的行为，例如访问不存在的初始化或未初始化数据段时产生的段错误（20.2节讨论的SIGSEGV信号）。program break可以设置的上限值依赖于许多因素，包括：进程的数据段大小资源限制（RLIMIT\_DATA，36.3节描述）；内存映射、共享内存段、和共享库的位置。

调用sbrk()会调整program break，对其增加increment（在Linux中，sbrk()是基于brk()实现的库函数）。intptr\_t类型用来声明increment是整数类型。成功时sbrk()返回之前的program break地址。换句话说，如果我们增加了program break，则返回值是一个指向新分配内存块起始位置的指针。

sbrk(0)返回program break的当前位置而不修改它。这对我们跟踪堆大小非常有用，例如监控所有内存分配行为。

SUSv2规定了brk()和sbrk()（标记为LEGACY（遗留））。SUSv3删除了它们的规定。

### 7.1.2 在堆上分配内存：malloc()和free()

通常C程序使用malloc家族函数来分配和释放堆内存。这些函数相比brk()和sbrk()有许多优点，特别是：

* 标准化为C语言的组成部分；
* 在多线程程序中更容易使用；
* 提供简单的接口，允许以小单元分配内存块；
* 允许我们单独地释放内存块，使用链表进行维护并且在将来的内存分配中循环使用。

malloc()函数从堆中分配size字节，并返回新分配内存块起始位置的指针。分配的内存是未初始化的。

|  |
| --- |
| #include <stdlib.h>  void \*malloc(size\_t size);  成功时返回分配内存的指针；出错时返回NULL |

由于malloc()返回void \*，我们可以把它赋值给任意类型的C指针。malloc()返回的内存块总是对齐于适当的边界，可以适用于任何C数据结构。实践中在多数体系架构下分配内存对齐于8字节或16字节边界。

SUSv3规定调用malloc(0)可以返回NULL；也可以返回一小块内存，随后可以使用free()进行释放。在Linux中，malloc(0)采用了后一种方式。

如果内存分配失败（例如达到program break最大资源限制），则malloc()返回NULL，并设置errno来指示错误。尽管分配内存失败的可能性非常小，我们调用malloc()以及后面马上要讨论的相关函数时，都应该检查它们的错误返回。

free()函数释放ptr参数指向的内存块，该内存块必须是之前通过malloc()或其它堆内存分配函数所得到。

|  |
| --- |
| #include <stdlib.h>  void free(void \*ptr); |

通常free()不会降低program break，而是把这块内存添加到一个自由内存块列表中，供随后的malloc()使用。这样做的理由如下：

* 被释放的内存一般在堆的中间位置而不是末尾，因此不能简单地降低program break。
* 大大减少了程序调用sbrk()的次数（如3.1节所述，系统调用有很小但仍然不可忽略的开销）。
* 在许多情况下，降低break对分配大量内存的程序并没有好处，因为这些程序倾向于反复地分配和释放内存，而不是一次性完全释放并且继续运行很长时间。

如果free()的参数是NULL指针，则调用不做任何事（换句话说，传递NULL给free()并不是错误）。

调用free()之后对ptr的任何使用（例如再次传递给free()函数）都是错误的，会导致不可预料的结果。

**例子程序**

清单7-1中的程序可以用来说明free()对program break的影响。这个程序根据可选的命令行参数，来分配多个内存块，然后再释放其中的某些内存块。

前面两个命令行参数指定内存分配的数量和大小。第三个命令行参数指定循环的步长，用于释放内存块。如果我们指定1（忽略该参数时的默认值），则程序释放所有内存块；指定2则释放每两次分配的块；依此类推。第四和第五个命令行参数指定我们想要释放的块范围。如果忽略这两个参数，则所有分配的块都将被释放（根据第三个参数指定的步长）。

*清单7-1：演示释放内存对program break的影响*

---------------------------------------------memalloc/free\_and\_sbrk.c

#include "tlpi\_hdr.h"

#define MAX\_ALLOCS 1000000

int

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

{

char \*ptr[MAX\_ALLOCS];

int freeStep, freeMin, freeMax, blockSize, numAllocs, j;

printf("\n");

if (argc < 3 || strcmp(argv[1], "--help") == 0)

usageErr("%s num-allocs block-size [step [min [max]]]\n", argv[0]);

numAllocs = getInt(argv[1], GN\_GT\_0, "num-allocs");

if (numAllocs > MAX\_ALLOCS)

cmdLineErr("num-allocs > %d\n", MAX\_ALLOCS);

blockSize = getInt(argv[2], GN\_GT\_0 | GN\_ANY\_BASE, "block-size");

freeStep = (argc > 3) ? getInt(argv[3], GN\_GT\_0, "step") : 1;

freeMin = (argc > 4) ? getInt(argv[4], GN\_GT\_0, "min") : 1;

freeMax = (argc > 5) ? getInt(argv[5], GN\_GT\_0, "max") : numAllocs;

if (freeMax > numAllocs)

cmdLineErr("free-max > num-allocs\n");

printf("Initial program break: %10p\n", sbrk(0));

printf("Allocating %d\*%d bytes\n", numAllocs, blockSize);

for (j = 0; j < numAllocs; j++) {

ptr[j] = malloc(blockSize);

if (ptr[j] == NULL)

errExit("malloc");

}

printf("Program break is now: %10p\n", sbrk(0));

printf("Freeing blocks from %d to %d in steps of %d\n",

freeMin, freeMax, freeStep);

for (j = freeMin - 1; j < freeMax; j += freeStep)

free(ptr[j]);

printf("After free(), program break is: %10p\n", sbrk(0));

exit(EXIT\_SUCCESS);

}

---------------------------------------------memalloc/free\_and\_sbrk.c

以下面命令行运行该程序，将分配1000块内存，并以步长2释放相应的块：

$ ./free\_and\_sbrk 1000 10240 2

下面的输出显示在这些内存块释放后，program break保持不变，还是分配所有内存块之后达到的位置：

Initial program break: 0x804a6bc

Allocating 1000\*10240 bytes

Program break is now: 0x8a13000

Freeing blocks from 1 to 1000 in steps of 2

After free(), program break is: 0x8a13000

下面命令行释放除最后一个之外的所有内存块。同样program break保持高位不变：

$ ./free\_and\_sbrk 1000 10240 1 1 999

Initial program break: 0x804a6bc

Allocating 1000\*10240 bytes

Program break is now: 0x8a13000

Freeing blocks from 1 to 999 in steps of 1

After free(), program break is: 0x8a13000

但是如果我们释放堆顶部的一组内存块，则program break会相应地减小，表示free()使用sbrk()降低了program break。下面命令释放最后的500内存块：

$ ./free\_and\_sbrk 1000 10240 1 500 1000

Initial program break: 0x804a6bc

Allocating 1000\*10240 bytes

Program break is now: 0x8a13000

Freeing blocks from 500 to 1000 in steps of 1

After free(), program break is: 0x852b000

在这种情况下，free()函数（glibc）能够识别出堆顶部的整个区域被释放，因为在释放块时，free()会将相邻的释放块合并到一起（合并是为了避免大量小碎片的存在，可能这些碎片都无法满足下一次malloc()的要求）。

glibc的free()函数只在堆顶部的已释放块足够大时，才会调用sbrk()来降低program break，“足够大”通过malloc包的参数来控制（典型值是128KB）。这样可以减少sbrk()调用的次数（也就是brk()系统调用的次数）。

**要不要free()？**

当进程终止时，它的所有内存都将返还给系统，包括使用malloc分配的堆内存。对于那些分配内存之后，一直使用这些内存到终止的程序，通常可以忽略free()，由系统的这个行为来自动释放内存。这对于分配许多内存块的程序特别有用，因为增加许多free()要浪费不少CPU时间，而且编码也更加复杂。

尽管依赖于进程终止来自动释放内存对很多程序是可接受的，也有许多理由要求程序显式地手工释放已分配内存：

* 显式调用free()可以增强程序的可读性和维护性，有助于将来的程序修改。
* 如果我们使用malloc调试库（下面会描述）来查找程序的内存泄漏，则任何没有显式释放的内存都会被报告为内存泄漏。这对于我们找出真正的泄漏增加了复杂性。

### 7.1.3 实现malloc()和free()

尽管malloc()和free()提供了直观的接口，使用也比brk()和sbrk()更加简单，但是在使用它们时仍然可能会犯严重的编程错误。理解malloc()和free()的实现，能让我们洞察这些错误产生的原因，并知道如何避免这些错误。

malloc()的实现非常简单。首先它扫描之前由free()释放的内存块，以找到大于或等于所请求的内存块（这里的扫描可以采用各种策略，如first-fit和best-fit）。如果块大小刚好适合，就把它返回给调用方；如果块大了，就分离出两个块，返回大小合适的那个块，并剩下另一个小块在自由列表中。

如果自由列表中没有足够大小的块，则malloc()调用sbrk()分配更多的内存。为了降低调用sbrk()的次数，malloc()会以更大的单元来增大program break（虚拟内存页大小的某个倍数），而不是仅仅增大用户请求的内存数。然后malloc()把剩余的内存放到自由列表中。

下面来看看free()的实现，现在事情变得更为有趣了。当free()把一块内存放到自由列表中时，它怎么知道这块内存的大小呢？这里有一个技巧，当malloc()分配内存块时，它会分配额外的整数字节来保存块大小。这个整数在内存块的开始位置；实际上返回给调用方的地址指向这个长度的后面字节，如图7-1所示：



*图7-1：malloc()返回的内存块*

当内存块被放在自由列表（双向链表）中时，free()使用块本身的字节来把块添加到列表中，如图7-2所示：



*图7-2：自由列表中的一个块*

随着块被释放和重新分配，自由列表中的块会混杂着已分配，正在使用的内存，如图7-3所示：



*图7-3：堆包含已分配块和自由列表*

C语言允许我们创建指向堆任意位置的指针，并修改指针指向位置的内容，包括free()和malloc()维护的长度、上一个自由块、下一个自由块。结合上文的讨论，我们就能明白为什么C语言的内存和指针可能引入极度隐晦的程序Bug。例如错误的指针，我们偶然地增大了已分配内存块前面的某个长度值，随后我们释放该内存块时，free()就会记录错误的内存块大小到自由列表，后面的malloc()可能会重新分配这个内存块，导致程序指针指向两个以为是独立的内存块，而实际上这两个内存块却是重叠的。我们还可以想象很多内存Bug的情况出来。

要避免这类错误，我们应该遵守以下规则：

* 在我们分配一块内存之后，必须非常小心不要访问超出该内存块范围的任何字节。例如错误的指针运算、或循环语句更新内存块时的off-by-one错误。
* 不要多次释放同一块已分配内存。在Linux的glibc环境下，通常会导致段违例错误（SIGSEGV信号）。这样还算是好的，因为它警告我们犯了编程错误。但是更常见的情况是，释放相同内存多次会导致未预料的行为。
* 决不能free()不是通过malloc库中某个函数分配的内存指针。
* 如果我们编写长时间运行的程序（例如shell或网络daemon进程），这样的程序需要反复的分配内存，则我们需要确保内存使用完之后得到正确地释放。否则堆将持续地增长，直到我们到达可用虚拟内存的最大限制，到那时所有分配内存的请求都将失败。这种情况被称为内存泄漏。

**malloc调试的工具和库**

未能遵守上述规则，可能产生非常晦涩和难以重现的Bug。使用glibc提供的malloc调试工具或者malloc调试库，可以更好地发现这一类型的Bug。

glibc提供以下malloc调试工具：

* mtrace()和muntrace()函数：允许程序打开和关闭内存分配调用跟踪。这两个函数与MALLOC\_TRACE环境变量结合使用，后者定义为跟踪信息要写入的文件。调用mtrace()时，函数会检查文件路径及文件是否可以打开为写入；如果可以打开写入，则malloc库的所有函数调用都会被跟踪并记录在该文件中。这个文件不太容易人类可读，因此提供一个mtrace脚本来分析该文件并生成可读的汇总信息。由于安全方面的原因，设置用户ID和设置组ID的程序调用mtrace()将被忽略。
* mcheck()和mprobe()函数：允许程序对已分配内存块执行一致性检查。例如捕获试图往超出已分配内存块末尾位置写入的错误。这些函数提供的功能与下面讨论的malloc调试库有一定的重叠。采用了这两个函数的程序必须使用cc -lmcheck选项与mcheck库链接。
* MALLOC\_CHECK\_ 环境变量：类似于mcheck()和mprobe()的功能。（两种技术的一个显著区别是，使用MALLOC\_CHECK\_ 不需要修改和重新编译程序）。通过设置这个环境变量为不同的整数值，我们可以控制程序如何响应内存分配错误。可以设置的值包括：0，表示忽略错误；1，表示打印错误诊断到stderr；2，表示调用abort()来终止程序。不是所有的内存分配和释放错误都能够通过MALLOC\_CHECK\_ 来检测到；它只能发现常见的内存错误。不过这个技术的特点是快速、容易使用、而且运行时开销相比使用malloc调试库也要低很多。由于安全方面的原因，设置用户ID和设置组ID的程序将忽略MALLOC\_CHECK\_ 设置。

上面这些特性的更多信息可以参考glibc手册。

malloc调试库提供标准malloc库相同的API，但是做了额外的工作来捕获内存分配的bug。要使用这样的调试库，我们要把程序链接到调试库而不是标准C库的malloc包。由于这些库通常都带来运行时开销、以及增大内存消耗的代价，我们只能在调试阶段使用这些库，生产系统的应用还是要链接到标准malloc包。常用的malloc调试库包括：

Electric Fence (<http://www.perens.com/FreeSoftware/>)；

dmalloc (<http://dmalloc.com/>)；

Valgrind (<http://valgrind.org/>)；

Insure++ (<http://www.parasoft.com/>)。

Valgrind和Insure++除了检测堆相关的分配，还能够检测许多其它类型的Bug。细节信息请参考各自的网站。

**控制和监控malloc包**

glibc手册描述了一系列非标准的函数，可以用来监控和控制malloc包的内存分配函数，包括以下：

* mallopt()函数可以修改各种参数，控制malloc()所使用的算法。例如有一个参数指定sbrk()缩小堆之前，自由列表必须存在的最小可释放空间大小。另一个参数指定从堆中分配的块大小的上限，超过这个限制的内存块将使用mmap()系统调用来分配（参考49.7节）。
* mallinfo()函数返回一个结构体，包含malloc()分配的所有内存的各种统计信息。

许多UNIX实现提供不同版本的mallopt()和mallinfo()，但是不同实现提供的接口可能会有不同，因此这些函数是不可移植的。

### 7.1.4 在堆上分配内存的其它方法

除了malloc()，C库还提供其它一些在堆上分配内存的函数，下面我们就来详细讨论它们。

**使用calloc()和realloc()分配内存**

calloc()函数分配相同元素的一个数组。

|  |
| --- |
| #include <stdlib.h>  void \*calloc(size\_t numitems, size\_t size);  成功时返回分配内存的指针，出错返回NULL |

numitems参数指定要分配的元素个数，size参数则指定每个元素的大小。calloc()分配了适当大小的内存块之后，返回指向块首地址的指针（如果无法完成内存分配则返回NULL）。和malloc()不一样，calloc()初始化已分配内存为0。

下面是calloc()的使用示例：

struct { /\* Some field definitions \*/ } myStruct;

struct myStruct \*p;

p = calloc(1000, sizeof(struct myStruct));

if (p == NULL)

errExit("calloc");

realloc()函数用来调整（通常是扩大）已分配内存块的大小，这块内存必须是通过malloc包分配得来的。

|  |
| --- |
| #include <stdlib.h>  void \*realloc(void \*ptr, size\_t size);  成功时返回已分配内存的指针，出错返回NULL |

ptr参数是要调整大小的内存块指针，size参数指定新内存块的大小。

成功时realloc()返回调整大小后的内存块地址，这个地址可能和调用之前的内存块地址不同。出错时realloc()返回NULL，并且不会改变ptr指向的内存块（SUSv3要求这个特性）。

当realloc()增大已分配内存块的大小时，它不会初始化额外分配的字节。

使用calloc()和realloc()分配的内存都应该使用free()来释放。

调用realloc(ptr, 0)等同于调用free(ptr)+malloc(0)。如果ptr为NULL，则realloc()等同于调用malloc(size)。

如果我们增加内存块的大小，如果自由列表存在紧邻而且足够大的内存块， realloc()会尝试合并内存块。如果内存块在堆的末尾，则realloc()会扩展堆。如果内存块在堆的中间，而且没有足够大的紧邻内存空间，realloc()就会分配新的内存块并复制所有数据到该内存。最后一种情况很常见而且非常消耗CPU。通常我们应该尽量少用realloc()。

由于realloc()可能会重新分配内存块，我们必须使用realloc()的返回值来引用该内存块。我们可以采用如下代码来重新分配ptr指向的内存块：

nptr = realloc(ptr, newsize);

if (nptr == NULL) {

/\* Handle error \*/

} else { /\* realloc() succeeded \*/

ptr = nptr;

}

在这个例子中，我们没有把realloc()的返回值直接赋给ptr，因为如果realloc()失败，ptr会被设置为NULL，造成现有内存块泄漏。

由于realloc()可能会移动内存块，调用realloc()之前所有引用内存块的任何指针，在realloc()调用之后都可能不再合法。内存块内唯一保证合法的地址引用是块首地址加偏移量而得到的指针。我们在48.6节会更详细地讨论这个指针。

**分配对齐的内存：memalign()和posix\_memalign()**

memalign()和posix\_memalign()函数用来分配对齐于特定边界的内存，这个特性对某些应用很有用（例如清单13-1）。

|  |
| --- |
| #include <malloc.h>  void \*memalign(size\_t boundary, size\_t size);  成功时返回已分配内存指针；出错返回NULL |

memalign()函数分配size字节内存，地址对齐于boundary的倍数，后者必须是2的幂。函数的返回值是已分配的内存块地址。

memalign()函数不是所有UNIX实现都可用。多数提供memalign()函数的UNIX实现要求包含<stdlib.h>而不是<malloc.h>，才能得到该函数的声明。

SUSv3没有规定memalign()，但是规定了一个类似的函数，名为posix\_memalign()。这个函数是标准委员会最近新增的，因此只有少数UNIX实现提供。

|  |
| --- |
| #include <stdlib.h>  int posix\_memalign(void \*\*memptr, size\_t alignment, size\_t size);  成功时返回0；出错时返回正的错误值 |

posix\_memalign()与memalign()函数有两个方面不同：

* 已分配内存的地址通过memptr返回
* 内存对齐于alignment的倍数，而且必须是2的幂 乘以 sizeof(void\*)（多数硬件体系架构下为4或8字节）。

同样还需要注意这个函数的返回值，出错时不是返回-1，而是返回错误数值（也就是通常通过errno返回的正整数值）。

假设sizeof(void\*)等于4，我们要使用posix\_memalign()来分配65536字节内存，并对齐于4096字节边界，代码如下：

int s;

void \*memptr;

s = posix\_memalign(&memptr, 1024 \* sizeof(void \*), 65536);

if (s != 0)

/\* Handle error \*/

使用memalign()或posix\_memalign()分配的内存块也应该使用free()来释放。

在某些UNIX实现中，memalign()分配的内存块可能无法调用free()，因为memalign()的实现使用了malloc()来分配内存块，然后返回内存块中适合于对齐要求的指针。glibc的memalign()实现没有这个问题。

## 7.2 在栈上分配内存：alloca()

和malloc包里的函数一样，alloca()函数也用于动态分配内存。但是alloca()不从堆中分配，而是通过增大栈帧的大小，从堆栈中获取内存。由于调用函数的栈帧在堆栈的最顶部，因此这个方法是可行的。栈帧上面有空间可以扩展，简单地修改堆栈指针的值就能完成这个操作。

|  |
| --- |
| #include <alloca.h>  void \*alloca(size\_t size);  返回分配的内存块指针 |

size参数指定要在堆栈中分配的字节数。alloca()函数返回指向已分配内存块的指针。

我们不需要（实际上不可以）调用free()来释放alloca()分配的内存。同样我们也不可能调用realloc()来调整alloca()分配的内存块的大小。

尽管alloca()并不是SUSv3标准规定，多数UNIX实现都提供了该函数，因此一定程度上是可移植的。

老版本的glibc，以及其它一些UNIX实现（主要是BSD衍生系统），要求包含<stdlib.h>而不是<alloca.h>来获得alloca()函数的声明。

如果调用alloca()导致了堆栈溢出，则程序的行为是未定义的。而且我们调用alloca()无法得到NULL返回值来显示产生了错误。（实际上在这种情况下，我们可能会收到SIGSEGV信号，更多细节请参考21.3节）。

注意我们不能在函数参数列表中使用alloca()，如下面例子所示：

func(x, alloca(size), z); /\* WRONG! \*/

因为这样alloca()分配的堆栈空间会出现在函数参数的中间（而函数参数需要存储在栈帧的固定位置）。我们必须使用下面代码：

void \*y;

y = alloca(size);

func(x, y, z);

使用alloca()来分配内存比malloc()有几个优点。其中之一是alloca()分配内存比malloc()要快，因为编译器使用内联代码来实现alloca()，直接调整堆栈指针。此外alloca()也不需要维护自由内存块列表。

alloca()的另一个优点是它分配的内存在栈帧被移除时自动释放，也就是调用alloca()的函数返回时。这是因为函数返回时会重置堆栈指针寄存器的值为上一个帧的末尾（假设堆栈向下增长，重置为当前帧起始位置的上面）。由于我们不需要在函数的所有返回路径都处理分配内存的释放，函数代码能够更加简洁。

如果我们使用了longjmp()（6.8节）或siglongjmp()（21.2.1节），来执行非局部跳转，则alloca()特别有用。在这种情况下，如果我们在跳转函数中使用malloc()来分配内存，则非常困难甚至不可能避免内存泄漏。相反alloca()完全避免了这个问题，因为随着这些调用解开堆栈，分配的内存也会自动释放。

## 7.3 小结

使用malloc家族函数，进程可以在堆上动态地分配和释放内存。在考虑这些函数的实现中，我们看到程序中错误地处理已分配内存块，可能导致的各种问题，然后我们讨论了一些能够帮助定位这种错误的调试工具。

alloca()函数在堆栈上分配内存，这些内存在调用alloca()的函数返回时将自动释放。

## 7.4 习题

7-1. 修改清单7-1中的程序（free\_and\_sbrk.c），在每次执行malloc()之后打印当前program break的值。指定一个小的分配块大小来运行该程序。这能够演示malloc()并不会在每次调用时都采用sbrk()来调整program break，而是定期地分配更大的内存块，然后从中返回小内存块给调用方。

7-2. （高级）实现malloc()和free()。

# 第8章 用户和组

每个用户都有唯一的登录名和相关联的数值用户标识（UID）。用户可以属于一个或多个组。每个组同样也有唯一的名字和组标识（GID）。

用户和组ID的主要目的是确定各种系统资源的所有权，以及控制进程获得的权限来访问这些资源。例如每个文件都属于一个特定的用户和组，每个进程也有若干用户和组ID，用来确定谁拥有该进程以及拥有何种权限来访问文件（更多细节请参考第9章）。

本章我们讨论用来定义系统中用户和组的系统文件，然后描述从这些文件获取信息的库函数。最后以crypt()函数结束我们这一章的讨论，该函数用来加密和验证登录密码。

## 8.1 密码文件：/etc/passwd

在系统密码文件/etc/passwd中，每一行对应于系统中的一个用户账户。每一行都由几个域组成，并以冒号（:）分隔，如下所示：

mtk:x:1000:100:Michael Kerrisk:/home/mtk:/bin/bash

这些域依次为：

* 登录名：这是用户登录时必须输入的唯一名字，通常也称为用户名。我们可以认为登录名是人类可读的标识（符号），对应于数值的用户标识（下面描述）。类似ls的程序显示文件拥有者时（ls -l），会显示文件相关联的用户名而不是数值用户ID。
* 加密的密码：这个域包含13字符的加密密码，我们在8.5节会更加详细地讨论。如果密码域包含了其它字符（不是13字符的字符串），则该账户被禁止登录，因为这样的字符串代表非法的加密密码。但是请注意如果启用了阴影密码（一般都已经启用），则密码域将会被忽略。在这种情况下，/etc/passwd文件的密码域惯例上设置为字母x（当然任何其它非空字符串也都是可以的），而加密密码实际上存储在阴影密码文件（8.2节）。如果/etc/passwd的密码域为空，则表示该账户登录不需要密码（即使阴影密码已经启用）。

这里我们假设密码使用Data Encryption Standard（DES）加密，由于历史原因仍然被UNIX密码加密机制广泛使用。我们可以使用其它加密算法来替代DES，如MD5产生128位消息摘要（哈希的一种）。这个值在密码（或阴影）文件中存储为34字符的字符串。

* 用户ID（UID）：该用户的数值ID。如果这个域的值为0，表示该账户拥有超级权限。通常只有一个这样的账户，登录名为root。在Linux 2.2及之前版本，用户ID为16位，允许0到65535范围的取值；在Linux 2.4及以后，用户ID存储为32位，因此允许大得多的范围。

密码文件可以拥有多个用户ID相同的记录（虽然并不常见），允许相同用户ID使用多个登录名。这样就允许多个用户使用不同的密码来访问相同的资源（如文件）。不同的登录名可以关联到不同的组ID集。

* 组ID（GID）：用户的第一个组的数值ID。该用户的其它组信息定义在系统组文件中。
* 注释：这个域保存了关于该用户的文本信息。许多程序会显示这个文本，如finger。
* Home目录：用户登录之后的初始目录。这个域同时也成为HOME环境变量的值。
* Login shell：这是用户登录之后控制权将转移至的程序。通常这是某一个shell程序，例如bash，但实际上也可以是任意程序。如果这个域为空，则默认为/bin/sh，也就是Bourne shell。这个域同时也成为SHELL环境变量的值。

在单机系统中，/etc/passwd文件存储了所有的密码信息。但是如果我们使用了Network Information System（NIS）或Lightweight Directory Access Protocol（LDAP）来分布密码到网络环境，部分或全部信息则会存储在远程系统中。只要程序采用本章下面描述的函数（getpwnam(), getpwuid()等）来访问密码信息，则NIS或LDAP对于应用将是透明的。同样的描述也适用于下面讨论的阴影文件和组文件。

## 8.2 阴影密码文件：/etc/shadow

历史上UNIX系统在/etc/passwd文件中维护了所有用户信息，包括加密密码，这导致了安全问题。因为许多非特权系统工具需要读取权限来访问密码文件的其它信息，因此该文件必须对所有用户可读。这同时也为密码破解程序打开了一扇门，暴力破解程序可以尝试大量可能的密码（例如标准字典或人名）进行加密，来查看结果是否匹配用户的加密密码。阴影密码文件/etc/shadow正是用来防止这一类攻击。主要的思路是所有非敏感的用户信息都存储在公开可读的密码文件中，而加密密码则由阴影密码文件维护，而只有特权程序才能读取阴影密码文件。

除了登录名（用来匹配密码文件中相对应的用户记录）、和加密密码，阴影密码文件还包含其它一些安全相关的域。这些域的详细信息可参考shadow手册页。我们在这里主要关注加密密码域，我们在8.5节讨论crypt()库函数时，会更加详细地讨论加密密码域。

SUSv3没有规定阴影密码，也不是所有UNIX实现都提供这个特性。

## 8.3 组文件：/etc/group

按组来管理用户，对于许多系统管理操作，特别是控制用户访问文件和其它系统资源，是非常有用的。

用户可以属于多个组，用户密码文件项的组ID域定义了用户属于的第一个组，而组文件则定义了用户属于的其它组。之所以分开两个文件来定义用户所属的组，主要是历史原因造成的。在早期的UNIX实现中，一个用户同时只能从属于一个组。用户最早的组在登录时由密码文件中的组ID域决定，随后可以使用newgrp命令来修改，而且要求用户提供组密码（如果组采用了密码保护）。4.2BSD引入了同时属于多个组的概念，并且随后POSIX.1-1990对其进行了标准化。在这种设计中，组文件列出了每个用户从属的额外组（groups命令显示了shell进程所属的组；或者如果提供了一个或多个用户名作为命令行参数，该命令显示这些用户所属的组）。

组文件/etc/group每一行包含了系统中的一个组。每一行由四个域组成，冒号分隔，如下所示：

users:x:100:

jambit:x:106:claus,felli,frank,harti,markus,martin,mtk,paul

这四个域的含义按顺序如下：

* 组名：这个组的名字，和密码文件中的登录名一样，我们可以认为组名是用户可读的标识（符号），对应于数值的组标识（GID）。
* 加密密码：这个域包含可选的组密码。随着多个组从属关系的出现，当今UNIX系统已经很少使用组密码了。无论如何，我们可以为组指定密码（特权用户使用passwd命令）。如果用户不是某个组的成员，newgrp会要求提供组密码，才会开户一个新的属于该组的shell。如果启用了阴影密码，则这个域将被忽略（这种情况下，传统上我们使用字母x，不过任何字符串，甚至是空字符串都是允许的），加密的组密码实际上存储在阴影组文件/etc/gshadow中，该文件只能由特权用户和程序访问。组密码采用类似于用户密码的加密方式（8.5节）。
* 组ID（GID）：这是数值的组ID。通常有一个组ID为0，称为root组。在Linux 2.2及以前，组ID为16位的值，允许0到65535的取值范围；在Linux 2.4及以后，组ID存储为32位。
* 用户列表：这是逗号分隔的用户名列表，这里列出的所有用户都属于这个组。（这个列表包含用户名而不是用户ID，因为密码文件中的用户ID并不需要唯一）。

假设用户avr从属于users, staff, teach组，而且密码文件中有如下记录：

avr:x:1001:100:Anthony Robins:/home/avr:/bin/bash

则组文件中会有以下记录：

users:x:100:

staff:x:101:mtk,avr,martinl

teach:x:104:avr,rlb,alc

密码记录的第四个域包含组ID 100，表示用户属于users组。用户所属的其它组则在组文件相关的组记录中列出了用户avr。

## 8.4 获取用户和组信息

在这一节，我们讨论从密码文件、阴影密码文件、和组文件中获取信息的库函数，以及扫描这些文件中所有记录的方法。

**从密码文件中获取记录**

getpwnam()和getpwuid()函数从密码文件中获取记录。

|  |
| --- |
| #include <pwd.h>  struct passwd \*getpwnam(const char \*name);  struct passwd \*getpwuid(uid\_t uid);  成功时都返回一个指针，出错返回NULL；  参考“没有找到”情况下的相关描述 |

name参数指定登录名，getpwnam()函数返回指向下面结构体的指针，包含了密码记录项的相应信息：

struct passwd {

char \*pw\_name; /\* 登录名（用户名） \*/

char \*pw\_passwd; /\* 加密密码 \*/

uid\_t pw\_uid; /\* 用户ID \*/

gid\_t pw\_gid; /\* 组ID \*/

char \*pw\_gecos; /\* 注释（用户信息） \*/

char \*pw\_dir; /\* 初始工作目录（home） \*/

char \*pw\_shell; /\* 登录shell \*/

};

SUSv3没有定义passwd结构体的pw\_gecos和pw\_passwd域，但是在所有UNIX实现中这两个域都可用。只有阴影密码没有启用时，pw\_passwd域才包含合法的信息。（从编程的角度来看，确定系统是否启用阴影密码，最简单的方法是成功调用getpwnam()之后，再调用getspnam()，检查后者是否为相同用户返回阴影密码记录）。其它UNIX实现还在这个结构体中提供了额外的非标准的域。

getpwuid()函数返回的信息和getpwnam()完全一样，只不过使用uid参数提供的数值用户ID来进行查找。

getpwnam()和getpwuid()都返回一个静态分配的结构体指针。这个结构体会被后续的这些调用所覆盖（还包括下面讨论的getpwent()函数）。

由于这些函数返回静态分配的内存指针，getpwnam()和getpwuid()都不是可重入的。实际上情况比这还要复杂，因为返回的passwd结构体还包含其它信息的指针（例如pw\_name域），而这些指针也是静态分配的。（我们在21.1.2节讨论可重入）。getgrnam()和getgrgid()函数（马上就要讨论）同样也面临相同的情况。

SUSv3规定了一组可重入的函数：getpwnam\_r(), getpwuid\_r(), getgrnam\_r(), getgrgid\_r()，这些函数需要提供passwd或group结构体以及额外的缓冲区作为参数，用来保存返回的passwd或group结构体，以及保存相关的域数据。这个额外的缓冲区要求的字节大小，可以使用sysconf(\_SC\_GETPW\_R\_SIZE\_MAX)或sysconf(\_SC\_GETGR\_R\_SIZE\_MAX)来获取。这些函数的相关信息请参考手册页。

根据SUSv3的规定，如果无法找到匹配的passwd记录，getpwnam()和getpwuid()将返回NULL，并且不修改errno（如果发生了错误则会修改errno）。这意味着我们可以采用下面代码来区分“没有找到”还是发生了其它错误：

struct passwd \*pwd;

errno = 0;

pwd = getpwnam(name);

if (pwd == NULL) {

if (errno == 0)

/\* Not found \*/;

else

/\* Error \*/;

}

但是许多UNIX实现在这一点并没有遵循SUSv3，如果找不到匹配的passwd记录，这些函数返回NULL，并且设置errno为非0值，例如ENOENT或ESRCH。glibc 2.7之前，在这种情况下会产生ENOENT错误；但是从2.7版本开始，glibc严格遵守了SUSv3要求。不同实现的这种差异，部分原因是POSIX.1-1990没有要求这些函数在错误时设置errno，并允许函数在“没有找到”时也设置errno。结论是使用这些函数时如果要严格区分错误和“没有找到”，实际上是不可移植的。

**从组文件获取记录**

getgrnam()和getgrgid()函数从组文件中获取记录。

|  |
| --- |
| #include <grp.h>  struct group \*getgrnam(const char \*name);  struct group \*getgrgid(gid\_t gid);  成功时都返回指针，出错返回NULL；  参考关于“没有找到”情况下的相关讨论 |

getgrnam()函数通过组名查找组信息，而getgrgid()函数则通过组ID进行查找。两个函数都返回以下结构体的指针：

struct group {

char \*gr\_name; /\* 组名 \*/

char \*gr\_passwd; /\* 加密密码（如果没有启用阴影）\*/

gid\_t gr\_gid; /\* 组ID \*/

char \*\*gr\_mem; /\* NULL终止的数组，包含/etc/group文件中列出的成员名 \*/

};

group结构体的gr\_passwd域不属于SUSv3规范，但在多数UNIX实现中都可用。

正如上面密码函数的相关讨论，这个结构体也会被后续函数调用所覆盖。

如果这两个函数无法找到匹配的组记录，这两个函数的行为和上面讨论的getpwnam()和getpwuid()函数一样。

**示例程序**

本节讨论的这些函数最常见的一个用途就是用户名（或组名）与数值ID之间的互相转换。清单8-1演示了这样的转换，提供了四个函数：userNameFromId(), userIdFromName(), groupNameFromId(), groupIdFromName()。为了方便调用，userIdFromName()和groupIdFromName()还允许name参数为数值ID字符串；在这种情况下，字符串被直接转换为数字并返回给调用方。本书后面的一些例子程序将会采用这里的四个函数。

*清单8-1：转换用户和组ID与用户和组名*

-----------------------------------------users\_groups/ugid\_functions.c

#include <pwd.h>

#include <grp.h>

#include <ctype.h>

#include "ugid\_functions.h" /\* Declares functions defined here \*/

char \* /\* Return name corresponding to 'uid', or NULL on error \*/

userNameFromId(uid\_t uid)

{

struct passwd \*pwd;

pwd = getpwuid(uid);

return (pwd == NULL) ? NULL : pwd->pw\_name;

}

uid\_t /\* Return UID corresponding to 'name', or -1 on error \*/

userIdFromName(const char \*name)

{

struct passwd \*pwd;

uid\_t u;

char \*endptr;

if (name == NULL || \*name == '\0') /\* On NULL or empty string \*/

return -1; /\* return an error \*/

u = strtol(name, &endptr, 10); /\* As a convenience to caller \*/

if (\*endptr == '\0') /\* allow a numeric string \*/

return u;

pwd = getpwnam(name);

if (pwd == NULL)

return -1;

return pwd->pw\_uid;

}

char \* /\* Return name corresponding to 'gid', or NULL on error \*/

groupNameFromId(gid\_t gid)

{

struct group \*grp;

grp = getgrgid(gid);

return (grp == NULL) ? NULL : grp->gr\_name;

}

gid\_t /\* Return GID corresponding to 'name', or -1 on error \*/

groupIdFromName(const char \*name)

{

struct group \*grp;

gid\_t g;

char \*endptr;

if (name == NULL || \*name == '\0') /\* On NULL or empty string \*/

return -1; /\* return an error \*/

g = strtol(name, &endptr, 10); /\* As a convenience to caller \*/

if (\*endptr == '\0') /\* allow a numeric string \*/

return g;

grp = getgrnam(name);

if (grp == NULL)

return -1;

return grp->gr\_gid;

}

-----------------------------------------users\_groups/ugid\_functions.c

**扫描密码和组文件的所有记录**

setpwent()、getpwent()、和endpwent()函数用来顺序扫描密码文件中的记录。

|  |
| --- |
| #include <pwd.h>  struct passwd \*getpwent(void);  成功时返回passwd结构体指针，流末尾或错误时返回NULL  void setpwent(void);  void endpwent(void); |

getpwent()函数从密码文件中一个一个地返回记录，当没有更多记录时返回NULL（或者发生了错误）。第一次调用getpwent()函数时自动打开密码文件。当我们操作完成之后，调用endpwent()来关闭密码文件。

我们可以使用下面代码来遍历整个密码文件，并打印例如名和用户ID：

struct passwd \*pwd;

while ((pwd = getpwent()) != NULL)

printf("%-8s %5ld\n", pwd->pw\_name, (long) pwd->pw\_uid);

endpwent();

endpwent()调用是必须的，这样随后的getpwent()调用（可能在程序的其它部分或我们调用的某个库）才能重新打开密码文件并从起始位置开始。如果我们已经读取了部分密码文件记录，也可以使用setpwent()函数从起始位置重新开始。

getgrent(), setgrent(), endgrent()函数对组文件执行类似的操作。这里我们忽略这三个函数的原型介绍，因为它们和上面讨论的密码文件函数是一样的；更多细节请参考相应的手册页。

**从阴影密码文件中获取记录**

下面函数扫描阴影密码文件的所有记录，并获取每条记录的信息。

|  |
| --- |
| #include <shadow.h>  struct spwd \*getspnam(const char \*name);  成功时返回指针，没有找到或出错时返回NULL  struct spwd \*getspent(void);  成功时返回指针，流末尾或出错时返回NULL  void setspent(void);  void endspent(void); |

我们不详细描述这些函数，因为它们执行的操作类似于相应的密码文件函数。（SUSv3没有规定这些函数，也不是所有UNIX实现都可用）。

getspnam()和getspent()函数返回结构体spwd类型的指针，spwd结构体的组成如下：

struct spwd {

char \*sp\_namp; /\* Login name (username) \*/

char \*sp\_pwdp; /\* Encrypted password \*/

/\* Remaining fields support "password aging", an optional

feature that forces users to regularly change their

passwords, so that even if an attacker manages to obtain

a password, it will eventually cease to be usable. \*/

long sp\_lstchg; /\* Time of last password change

(days since 1 Jan 1970) \*/

long sp\_min; /\* Min. number of days between password changes \*/

long sp\_max; /\* Max. number of days before change required \*/

long sp\_warn; /\* Number of days beforehand that user is

warned of upcoming password expiration \*/

long sp\_inact; /\* Number of days after expiration that account

is considered inactive and locked \*/

long sp\_expire; /\* Date when account expires

(days since 1 Jan 1970) \*/

unsigned long sp\_flag; /\* Reserved for future use \*/

};

我们在清单8-2中演示getspname()的使用。

## 8.5 密码加密和用户认证

有些应用要求用户验证自己的身份。验证通常都是以用户名（登录名）和密码来完成。应用可以维护自己的用户名和密码数据库以实现验证操作。不过有时候，允许用户使用/etc/passwd和/etc/shadow定义的标准用户名和密码，可能会更加方便。（在本节剩余的部分，我们假设系统启用了阴影密码，因此加密密码存放在/etc/shadow文件中）。那些提供远程系统登录的网络应用，例如ssh和ftp是典型的这一类程序。这些应用必须像标准login程序那样验证用户名和密码。

由于安全方面的原因，UNIX系统的密码加密使用不可逆加密算法，这就意味着没有办法从加密密码重新创建原始密码。因此验证某个密码是否合法，唯一的办法是使用相同的算法对其进行加密，并查看加密后的结果与/etc/shadow中存储的值是否匹配，加密算法则封装在crypt()函数中。

|  |
| --- |
| #define \_XOPEN\_SOURCE  #include <unistd.h>  char \*crypt(const char \*key, const char \*salt);  成功时返回静态分配的加密密码字符串；出错时返回NULL |

crypt()算法需要一个key（密码），最多8个字符，并且对其应用Data Encryption Standard（DES）算法的一个变种。salt参数是两个字节的字符串，用来混淆（变化）算法，使得破解加密密码更加地困难。函数返回静态分配的指针，加密密码为13字符的字符串。

salt参数和加密密码都由64字符集[a-zA-Z0-9/.]组成。因此salt参数可以导致加密算法变换为64 \* 64 = 4096种不同的方式。这意味着破解者需要检查4096种加密版本的字典才能检测加密结果。

crypt()返回的加密密码的前两个字节包含了原始的salt值。这意味着当加密候选密码时，我们可以从/etc/shadow中存储的加密密码的值得到适当的salt值。（passwd程序在加密新密码时会生成随机salt值）。实际上，crypt()函数忽略salt字符串前两位之后的所有字符。因此我们可以把加密密码直接传递给salt参数。

在Linux中要使用crypt()，我们必须以-lcrypt选项编译程序，这样程序才会链接到crypt库。

**示例程序**

清单8-2演示了怎样使用crypt()来验证用户。这个程序首先读取一个用户名，然后获得相应的密码记录和阴影密码记录（如果存在）。如果找不到相应的密码记录，或者程序没有权限读取阴影密码文件（要求超级特权或shadow组成员），程序打印一条错误消息然后退出。然后程序使用getpass()函数读取用户的密码。

|  |
| --- |
| #define \_BSD\_SOURCE  #include <unistd.h>  char \*getpass(const char \*prompt);  成功时返回静态分配的输入密码字符串；出错时返回NULL |

getpass()函数首先禁用屏幕显示和终端所有特殊字符处理（例如中断字符，通常是Control-C）。（我们会在第62章解释如何修改这些终端设置）。然后getpass()打印prompt字符串，并读取一行输入，并返回去除末尾换行字符的null终止字符串作为函数结果。（这个返回字符串是静态分配的，因此会被随后的getpass()调用覆盖）。在返回之前，getpass()会还原终端至原来的状态。

使用getpass()读取密码之后，清单8-2的程序然后使用crypt()对其进行加密，并将加密字符串与阴影密码文件中的相应记录进行比较，从而验证密码。如果密码匹配，则显示用户ID，如下所示：

**$ su** （需要特权来读取阴影密码文件）

Password:

**# ./check\_password**

Username: **mtk**

Password: （我们输入密码，字符不会显示）

Successfully authenticated: UID=1000

清单8-2的程序使用sysconf(\_SC\_LOGIN\_NAME\_MAX)来分配用户名字符串数组，这是当前系统的用户名最大大小。我们在11.2节讨论sysconf()的用法。

*清单8-2：使用阴影密码文件来验证用户*

-----------------------------------------users\_groups/check\_password.c

#define \_BSD\_SOURCE /\* Get getpass() declaration from <unistd.h> \*/

#define \_XOPEN\_SOURCE /\* Get crypt() declaration from <unistd.h> \*/

#include <unistd.h>

#include <limits.h>

#include <pwd.h>

#include <shadow.h>

#include "tlpi\_hdr.h"

int

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

{

char \*username, \*password, \*encrypted, \*p;

struct passwd \*pwd;

struct spwd \*spwd;

Boolean authOk;

size\_t len;

long lnmax;

lnmax = sysconf(\_SC\_LOGIN\_NAME\_MAX);

if (lnmax == -1) /\* If limit is indeterminate \*/

lnmax = 256; /\* make a guess \*/

username = malloc(lnmax);

if (username == NULL)

errExit("malloc");

printf("Username: ");

fflush(stdout);

if (fgets(username, lnmax, stdin) == NULL)

exit(EXIT\_FAILURE); /\* Exit on EOF \*/

len = strlen(username);

if (username[len - 1] == '\n')

username[len - 1] = '\0'; /\* Remove trailing '\n' \*/

pwd = getpwnam(username);

if (pwd == NULL)

fatal("couldn't get password record");

spwd = getspnam(username);

if (spwd == NULL && errno == EACCES)

fatal("no permission to read shadow password file");

if (spwd != NULL) /\* If there is a shadow password record \*/

pwd->pw\_passwd = spwd->sp\_pwdp; /\* Use the shadow password \*/

password = getpass("Password: ");

/\* Encrypt password and erase cleartext version immediately \*/

encrypted = crypt(password, pwd->pw\_passwd);

for (p = password; \*p != '\0'; )

\*p++ = '\0';

if (encrypted == NULL)

errExit("crypt");

authOk = strcmp(encrypted, pwd->pw\_passwd) == 0;

if (!authOk) {

printf("Incorrect password\n");

exit(EXIT\_FAILURE);

}

printf("Successfully authenticated: UID=%ld\n", (long) pwd->pw\_uid);

/\* Now do authenticated work... \*/

exit(EXIT\_SUCCESS);

}

-----------------------------------------users\_groups/check\_password.c

清单8-2说明了一个重要的安全要点。读取密码的程序应该立即对密码进行加密，并马上从内存中擦除未加密版本的密码。这样可以最小化程序因产生core dump文件而暴露密码的可能性。

未加密密码还可能由于其它原因被暴露。例如包含密码的虚拟内存页面被交换出去，则特权程序就可以读取交换文件而获知密码。另外拥有足够权限的进程还可以读取/dev/mem（虚拟设备，描述计算机物理内存为顺序字节流），从而尝试发现未加密密码。

getpass()在SUSv2中被标记为遗留函数，标准认为该函数的名字容易令人误解，而且它提供的功能本身也很容易自己实现。SUSv3已经移除了getpass()的定义。无论如何，多数UNIX实现都提供了getpass()函数。

## 8.6 小结

每个用户都有唯一的登录名和相关联的数值用户ID。用户可以属于一个或多个组，每个组也有唯一的名字和相关联的数值标识。这些标识的主要目的在于确定各种系统资源的属主（例如文件），以及访问这些资源的权限。

用户名和ID定义在/etc/passwd文件中，该文件还包含用户的其它信息。用户的组成员关系定义在/etc/passwd和/etc/group文件中。此外/etc/shadow是只有特权程序可读的阴影密码文件，用来分离/etc/passwd文件中的敏感密码信息。系统提供了许多库函数，用来获取这些文件的信息。

crypt()函数按标准的login程序使用的方式来加密密码，当需要验证用户时，这个函数非常有用。

## 8.7 习题

8-1. 当我们执行下面代码时，即使两个用户拥有不同的ID，程序仍然两次显示相同的数字。为什么？

printf("%ld %ld\n", (long) (getpwnam("avr")->pw\_uid),

(long) (getpwnam("tsr")->pw\_uid));

8-2. 使用setpwent(), getpwent(), endpwent()实现getpwnam()。

# 第9章 进程凭证

每个进程都有一组相关联的数值用户标识（UID）和组标识（GID），有时候也称为进程凭证。这些标识如下：

* 实际用户ID和组ID；
* 有效用户ID和组ID；
* 保存的设置用户ID和保存的设置组ID；
* 文件系统用户ID和组ID（Linux特定）；
* 附加组ID。

在这一章，我们详细地描述这些进程标识的作用，以及获取和修改这些标识的系统调用和库函数。我们还讨论特权和非特权进程的概念，以及设置用户ID和设置组ID机制的用法，允许以指定用户或组的权限来运行程序。

## 9.1 实际用户ID和实际组ID

实际用户ID和组ID标识了进程属于的用户和组。login shell是登录进程的组成部分，从用户密码文件/etc/passwd的第三和第四个域获得自己的实际用户和组ID。当新进程被创建时（例如shell执行某个程序），会从父进程继承这些标识。

## 9.2 有效用户ID和有效组ID

在多数UNIX实现中（Linux稍微不同，9.5节描述），有效用户ID和组ID，与附加组ID结合使用，来决定授予进程执行各种操作时的权限（如文件系统）。例如进程访问文件和System V进程间通信（IPC）对象时，这些标识就确定了进程的权限，同时这些资源也有相关联的用户和组ID，用于确定它们的拥有者。我们在20.5节还将看到，内核使用有效用户ID来确定某个进程是否能向另一个进程发送信号。

有效用户ID为0（0是root用户的ID）的进程拥有超级用户的所有权限。这样的进程被称为特权进程，很多系统调用只能被特权进程执行。

在第39章，我们会讨论Linux对能力的实现，这个机制能够将授予超级用户的权限划分为许多独立的单元，并单独启用或禁用。

通常有效用户和组ID与相应的实际ID相同，但是有两种办法可以使有效ID获得不同的值。一是使用9.7节讨论的系统调用；二是执行设置用户ID和设置组ID的程序。

## 9.3 设置用户ID和设置组ID

设置用户ID的程序允许进程获得正常情况下没有的权限，通过设置进程的有效用户ID为可执行文件的用户ID（拥有者）。设置组ID的程序为进程的有效组ID执行类似的操作。（术语设置用户ID程序和设置组ID程序有时候也简称为设置UID程序和设置GID程序）。

和所有文件一样，可执行程序文件也有一个相关联的用户ID和组ID，定义了该文件的拥有者。此外可执行文件还有两个特殊的权限位：设置用户ID和设置组ID位。（实际上每个文件都有这两个权限位，但是这两个权限位只有和可执行文件结合起来才有效）。这些权限位通过chmod命令来进行设置。非特权用户可以为自己拥有的文件设置这两个权限位。特权用户（CAP\_FOWNER）可以为任何文件设置这两个权限位。下面是一个例子：

$ su

Password:

# ls -l prog

-rwxr-xr-x 1 root root 302585 Jun 26 15:05 prog

# chmod u+s prog 启用设置用户ID权限位

# chmod g+s prog 启用设置组ID权限位

如上所示，程序可以同时设置这两个权限位，不过通常我们不会这样做。当ls -l显示程序的权限时，如果启用了设置用户ID和设置组ID权限位，则使用字母s来替代字母x，表示该文件的可执行权限：

# ls -l prog

-rwsr-sr-x 1 root root 302585 Jun 26 15:05 prog

当运行设置用户ID程序时，内核会将进程的有效用户ID设置为可执行文件的用户ID。运行设置组ID程序时，进程的有效组ID也会被设置为文件的组ID。使用这个方法来修改进程的有效用户ID或有效组ID，可以使进程获得正常情况下没有的权限。例如某个可执行文件的拥有者是root（超级用户），并且启用了设置用户ID权限位，则程序运行时进程将获得超级用户权限。

设置用户ID和设置组ID程序也可以修改进程的有效ID为root之外的用户。例如为了提供受保护文件（或者其它系统资源）的访问，我们可能会创建特殊用途的用户（或组）ID，用来控制访问文件（或资源）的权限。于是设置用户ID和设置组ID程序就可以修改有效用户ID为这个特定的ID。这样程序就能够访问特定的文件或资源，而又不需要授予超级用户的全部权限。

有时候我们使用set-user-ID-root来区分root和其它用户拥有的设置用户ID程序，根据具体的用户来授予进程相应的权限。

由于38.3节描述的相关原因，在Linux中，设置用户ID和设置组ID权限位对于shell脚本没有任何作用。

Linux中常用的设置用户ID程序包括：passwd，用来修改用户的密码；mount和umount，挂载和卸载文件系统；su，允许用户以不同的用户ID运行shell。设置组ID的程序如wall，用于向tty组拥有的所有终端写入一条消息（通常每个终端都属于这个组）。

在8.5节，我们说明清单8-2的程序需要以root来运行才能访问/etc/shadow文件。我们可以把它改成set-user-ID-root程序，这样任何用户都可以运行这个程序，如下所示：

$ su

Password:

# chown root check\_password 使root拥有这个程序

# chmod u+s check\_password 启用设置用户ID位

# ls -l check\_password

-rwsr-xr-x 1 root users 18150 Oct 28 10:49 check\_password

# exit

$ whoami 这是非特权login

mtk

$ ./check\_password 但是我们可以使用这个程序

Username: avr 访问shadow密码文件

Password:

Successfully authenticated: UID=1001

设置用户ID/设置组ID技术是非常有用和强大的工具，但是如果设计不合理，也可能导致应用的安全漏洞。在第38章，我们列出了一组良好的实践准则，当我们编写设置用户ID和设置组ID程序时应该遵守。

## 9.4 保存的设置用户ID和保存的设置组ID

保存的设置用户ID和保存的设置组ID设计给设置用户ID程序和设置组ID程序使用。当程序执行时，会发生以下步骤：

1. 如果可执行文件启用了设置用户ID（设置组ID）权限位，则进程的有效用户（组）ID和可执行文件的拥有者相同。如果设置用户ID（设置组ID）没有启用，则不修改进程的有效用户（组）ID。
2. 保存的设置用户ID和保存的设置组ID从相应的有效ID中复制而来。无论是否启用可执行文件的设置用户（组）ID，这个复制都会发生。

举个例子说明上述步骤的效果，假设进程的实际用户ID、有效用户ID、保存的设置用户ID都是1000，执行root拥有的设置用户ID程序（用户ID为0）。在exec之后，进程的用户ID会作如下修改：

real=1000 effective=0 saved=0

许多系统调用允许设置用户ID程序改变自己的有效用户ID，可以在实际用户ID和保存的设置用户ID之间切换。同时还有类似的系统调用允许设置组ID程序修改自己的有效组ID。以这种方式，程序可以临时丢弃和重新获取被执行文件的拥有者的权限。正如我们在38.2节将详细描述的那样，如果设置用户ID和设置组ID程序不需要以特权ID执行操作，就应该以非特权ID（如实际ID）来执行，这是安全编程的实践。

保存的设置用户ID和保存的设置组ID有时候也被引用为保存的用户ID和保存的组ID。

保存的设置ID是System V发明的，并被POSIX采纳。BSD 4.4之前的发布版不提供保存的设置ID。最初的POSIX.1标准以可选的方式支持这些ID，但随后标准（1988年的FIPS 151-1）强制要求这些ID的支持。

## 9.5 文件系统用户ID和文件系统组ID

在Linux中，这是文件系统用户和组ID，而不是有效用户和组ID。用来（结合附加组ID）确定执行文件系统操作时的权限（如打开文件、修改文件属主、修改文件权限）。（有效ID仍然使用，和其它UNIX实现一样，目的如前所述）。

通常情况下，文件系统用户和组ID等于相应的有效ID（因此通常也等于相应的实际ID）。此外，无论有效用户或组ID是否改变，通过系统调用或执行设置用户（组）ID程序，相应的文件系统ID也同样修改为相同的值。由于文件系统ID以这种方式追随有效ID，意味着Linux在检查特权和权限时和其它UNIX实现的行为是完全一样的。文件系统ID不同于相应的有效ID，这时候Linux就和其它UNIX实现不同，但只有在我们使用Linux特定的系统调用setfsuid()和setfsgid()，显式地修改ID之后。

为什么Linux提供文件系统ID？在什么情况下有效ID和文件系统ID会不同呢？其实主要是历史原因。文件系统ID最早出现于Linux 1.2，在那个版本的内核中，如果进程的有效用户ID匹配另一个进程的实际或有效用户ID，进程就可以向那个进程发送信号。这影响到了某些程序，例如Linux NFS（网络文件系统）服务程序，它需要访问文件，就需要有效ID等于客户端进程的有效ID。但是如果NFS服务器修改了自己的有效用户ID，它就可能受到其它非特权用户进程的信号攻击。为了阻止这种可能性，设计了独立的文件系统用户和组ID。通过保持有效ID未修改，但修改文件系统ID，NFS服务器可以假装成其它用户来访问文件，而又不会受到用户进程的信号攻击。

从内核2.0开始，Linux采纳了SUSv3对发送信号权限的强制规定，而这些规则与目标进程的有效用户ID无关（参考20.5节）。因此文件系统ID特性实际上已经不再需要，但是对现有软件仍保持兼容。

由于文件系统ID是历史产物，而且它们的值通常与相应的有效ID相同，在本书的剩余部分，我们会讨论许多文件权限的检查，以及设置新文件的权限，这些都以进程的有效ID来讨论。尽管在Linux中检查文件系统权限时实际上使用的是进程的文件系统ID，但在实践中，文件系统ID和有效ID并没有实际差别。

## 9.6 附加组ID

附加组ID是一组进程属于的额外的组。新进程从父进程继承这些ID。login shell则从系统组文件中获得附加组ID。正如上面描述的，这些ID结合有效和文件系统ID一起使用，用来确定访问文件系统、System V IPC对象、以及其它系统资源的权限。

## 9.7 获取和修改进程凭证

Linux提供一系列系统调用和库函数来获取和修改不同的用户和组ID，只有部分API是由SUSv3规定的。但是在没有规定的API中，有一些广泛使用于UNIX实现中，还有少数是Linux特定的。我们在讨论每个接口时会标注可移植问题。在本章的末尾部分，表9-1汇总了所有接口改变进程凭证时进行的操作。

除了接下来要讨论的系统调用，我们也可以通过Linux特定的/proc文件系统来确定任何进程的凭证，/proc/PID/status文件的Uid、Gid、Groups行分别提供了这些信息，其中Uid和Gid都按实际、有效、保存设置、和文件系统的顺序列出相应的ID。

在接下来的几节，我们使用传统的特权进程定义，也就是有效用户ID为0的进程。但是Linux把超级用户的权限划分为不同的能力，第39章将会描述。有两种能力对于我们的讨论是相关的，会影响所有修改进程用户和组ID的系统调用：

* CAP\_SETUID能力允许进程任意地改变用户ID
* CAP\_SETGID能力允许进程任意地改变组ID

### 9.7.1 获取和修改实际、有效、和保存的设置ID

在接下来的讨论中，我们描述获取和修改实际、有效、保存的设置ID的系统调用。有几个系统调用可以完成这些任务，在某些情况下它们的功能互相重叠，因为这些系统调用起源于不同的UNIX实现。

**获取实际和有效ID**

getuid()和getgid()系统调用分别返回调用进程的实际用户ID和实际组ID。geteuid()和getegid()系统调用则分别返回有效ID。这些系统调用总是成功。

|  |
| --- |
| #include <unistd.h>  uid\_t getuid(void);  返回调用进程的实际用户ID  uid\_t geteuid(void);  返回调用进程的有效用户ID  gid\_t getgid(void);  返回调用进程的实际组ID  gid\_t getegid(void);  返回调用进程的有效组ID |

**修改有效ID**

setuid()系统调用修改调用进程的有效用户ID为uid参数指定的值（也可能修改实际用户ID和保存的设置用户ID）。setgid()系统调用则为相应的组ID执行类似的操作。

|  |
| --- |
| #include <unistd.h>  int setuid(uid\_t uid);  int setgid(gid\_t gid);  成功时都返回0；出错时都返回-1 |

进程使用setuid()和setgid()怎样修改进程凭证，依赖于进程是否拥有特权（有效用户ID等于0）。以下规则应用于setuid()：

1. 当非特权进程调用setuid()时，只有进程的有效用户ID被修改。此外有效用户ID只能被修改为实际用户ID或保存的设置用户ID（试图违反这个规定会得到EPERM错误）。这意味着对于非特权用户，这个调用只对执行设置用户ID程序时有用，因为如果执行普通程序，进程的实际用户ID、有效用户ID、和保存的设置用户ID本来就拥有相同的值。在某些源自BSD的实现中，非特权进程调用setuid()或setgid()与其它UNIX实现有不同的语义：调用修改实际、有效、和保存的设置ID（为当前实际或有效ID的值）。
2. 当特权进程以非0参数执行setuid()时，则实际用户ID、有效用户ID、保存的设置用户ID都被设置为uid参数指定的值。这是一个不可逆的过程，一旦特权进程按这种方式修改了自己的凭证，它立即失去所有特权因此不能再次调用setuid()来重新设置ID为0。如果这不是你想要的结果，则应该使用seteuid()或setreuid()，我们马上就会讨论它们。

控制setgid()修改组ID的规则也类似，只不过使用setgid()并提供group参数。使用setgid()时，规则1是完全符合的。规则2则稍有不同，因为修改组ID并不会导致进程丧失特权（特权由有效用户ID确定），特权程序可以使用setgid()自由地修改组ID为任何需要的值。

当set-user-ID-root程序的当前有效用户ID为0，如果需要丢弃所有特权时，可以把有效用户ID和保存的设置用户ID修改为实际用户ID，这是首选的方法：

if (setuid(getuid()) == -1)

errExit("setuid");

非root用户拥有的设置用户ID程序也可以使用setuid()来切换有效用户ID为实际用户ID和保存的设置用户ID，9.4节已经讨论过相关的安全原因。不过这里推荐使用seteuid()，因为它拥有相同的作用，无论设置用户ID程序是否root拥有。

进程可以使用seteuid()来修改自己的有效用户ID为euid指定的值，使用setegid()来修改有效组ID为egid指定的值。

|  |
| --- |
| #include <unistd.h>  int seteuid(uid\_t euid);  int setegid(gid\_t egid);  成功时都返回0，出错时都返回-1 |

进程使用seteuid()和setegid()修改有效ID时遵循以下规则：

1. 非特权进程只能修改有效ID为相应的实际或保存设置ID（换句话说，对于非特权进程，seteuid()和setegid()的作用与setuid()和setgid()相同，除了上面说过的BSD可移植性问题）。
2. 特权进程可以修改有效ID为任意值。如果特权进程使用seteuid()修改自己的有效用户ID为非0值，则进程将不再是特权（但可以通过上一节的规则重新获得特权）。

使用seteuid()是设置用户ID和设置组ID程序临时丢弃并稍后重获特权的首选方法，下面是一个例子：

euid = geteuid(); /\* 保存最初的有效用户ID(等于保存的设置用户ID) \*/

if (seteuid(getuid()) == -1) /\* 丢弃特权 \*/

errExit("seteuid");

if (seteuid(euid) == -1) /\* 重新获得特权 \*/

errExit("seteuid");

seteuid()和setegid()最初起源于BSD，现在已经被SUSv3标准化，在多数UNIX实现中都可用。

**修改实际和有效ID**

setreuid()系统调用允许调用进程独立地修改实际和有效用户ID。setregid()系统调用则修改实际和有效组ID。

|  |
| --- |
| #include <unistd.h>  int setreuid(uid\_t ruid, uid\_t euid);  int setregid(gid\_t rgid, gid\_t egid);  成功时都返回0；出错时都返回-1 |

这些系统调用的第一个参数是新的实际ID，第二个参数是新的有效ID。如果我们只想修改其中一个，可以为另一个参数指定-1。

setreuid()和setregid()最初起源于BSD，现在已经被SUSv3标准化，并且在多数UNIX实现中都可用。

和本节描述的其它系统调用一样，使用setreuid()和setregid()时也遵循相应的规则。下面我们描述setreuid()的规则，setregid()也类似，区别我们会标注出来：

1. 非特权进程只能修改实际用户ID为当前的实际（不修改）或有效用户ID。有效用户ID只能修改为当前的实际用户ID，或有效用户ID（不修改），或保存的设置用户ID。

SUSv3没有规定非特权进程是否可以使用setreuid()修改实际用户ID为当前的实际用户ID、有效用户ID、保存的设置用户ID，因此对实际用户ID进行修改的具体细节在不同实现可能存在差别。

SUSv3对于setregid()的行为描述则稍有不同：非特权进程可以设置实际组ID为保存的设置组ID，或者设置有效组ID为实际组ID或保存的设置组ID。同样，修改的细节在不同实现中也可能存在差别。

1. 特权进程可以任意修改ID。
2. 对于特权和非特权进程，如果以下任意条件成立，保存的设置用户ID都会被修改为（新的）有效用户ID：
   1. ruid不是-1（表示实际用户ID将被设置，即使是设置为现有的值）
   2. 有效用户ID被设置为不同于调用之前的实际用户ID

反过来说，如果进程使用setreuid()只修改有效用户ID为当前的实际用户ID，则保存的设置用户ID保持不变，后续的setreuid()或seteuid()可以重新还原有效用户ID为保存的设置用户ID。（SUSv3没有规定setreuid()和setregid()对保存的设置ID的作用，但SUSv4则规定了这里描述的行为）。

第三个规则提供了设置用户ID程序永久丢弃特权的方法，使用如下代码：

setreuid(getuid(), getuid());

set-user-ID-root进程如果要同时修改用户和组凭证为任意值，应该先调用setregid()然后调用setreuid()。如果调用顺序反过来，则setregid()将会失败，因为调用setreuid()之后进程已经不再拥有特权。如果使用setresuid()和setresgid()（下面马上讨论），也必须以同样的顺序调用。

**获取实际、有效、保存的设置ID**

在多数UNIX实现中，进程不能直接获取（或更新）保存的设置用户ID和保存的设置组ID。但是Linux提供了两个（非标准）系统调用允许我们这样做：getresuid()和getresgid()。

|  |
| --- |
| #define \_GNU\_SOURCE  #include <unistd.h>  int getresuid(uid\_t \*ruid, uid\_t \*euid, uid\_t \*suid);  int getresgid(gid\_t \*rgid, gid\_t \*egid, gid\_t \*sgid);  成功时都返回0；出错时都返回-1 |

getresuid()系统调用返回调用进程的实际用户ID、有效用户ID、保存的设置用户ID，返回值存储在三个指针参数中。getresgid()系统调用则返回相应的组ID。

**修改实际、有效、保存的设置ID**

setresuid()系统调用允许调用进程独立地修改所有三个用户ID，新的ID值由三个参数指定。setresgid()则相应地修改三个组ID。

|  |
| --- |
| #define \_GNU\_SOURCE  #include <unistd.h>  int setresuid(uid\_t ruid, uid\_t euid, uid\_t suid);  int setresgid(gid\_t rgid, gid\_t egid, gid\_t sgid);  成功时都返回0；出错时都返回-1 |

如果我们不想修改所有的凭证，可以为参数指定-1，则相应的ID将不会改变。例如，下面调用等同于seteuid(x)：

setresuid(-1, x, -1);

setresuid()修改凭证时遵循以下规则（setresgid()也类似）：

1. 非特权进程只能设置实际用户ID、有效用户ID、保存的设置用户ID为当前的实际用户ID、有效用户ID、或保存的设置用户ID。
2. 特权进程可以任意修改实际用户ID、有效用户ID、保存的设置用户ID。
3. 无论setresuid()如何修改这些ID，文件系统用户ID总是和有效用户ID（可能是修改后的）相同。

调用setresuid()和setresgid()拥有要么全有要么全无的效果，要么所有ID都成功修改，要么不修改任何一个。（本章描述的修改多个ID的系统调用都采用这种工作方式）。

尽管setresuid()和setresgid()提供了修改进程凭证最直接的API，但是在应用程序中我们一般不采用它们；因为SUSv3没有规定这两个调用，而且只在少数其它UNIX实现中可用。

### 9.7.2 获取和修改文件系统ID

前面讨论的所有系统调用在修改进程的有效用户或组ID时，同时也会修改相应的文件系统ID。如果要单独地修改文件系统ID，我们必须使用两个Linux特定的系统调用：setfsuid()和setfsgid()：

|  |
| --- |
| #include <sys/fsuid.h>  int setfsuid(uid\_t fsuid);  总是返回之前的文件系统用户ID  int setfsgid(gid\_t fsgid);  总是返回之前的文件系统组ID |

setfsuid()系统调用修改进程的文件系统用户ID为fsuid指定的值，setfsgid()系统调用则修改文件系统组ID为fsgid指定的值。

同样这两个系统调用在修改ID时也遵循特定的规则，setfsgid()的规则类似于setfsuid()，如下：

1. 非特权进程只能修改文件系统用户ID为当前的实际用户ID、有效用户ID、文件系统用户ID（不修改）、或保存的设置用户ID。
2. 特权进程可以修改文件系统用户ID为任意值。

这些调用的实现某种程度上讲是很粗糙的。首先，没有相应的系统调用来获取文件系统ID的当前值；此外，这些系统调用也没有错误检查；如果非特权进程试图修改文件系统ID为非法的值，只是简单地忽略该调用。这些系统调用的返回值是之前的文件系统ID，无论成功或者失败。因此我们可以使用这两个系统调用来获得当前的文件系统ID，但是只能尝试去修改文件系统ID（要么成功要么失败）。

即使在Linux中，也不再需要使用setfsuid()和setfsgid()系统调用，因此在需要移植到其它UNIX实现的应用中应该避免使用它们。

### 9.7.3 获取和修改附加组ID

getgroups()系统调用返回调用进程属于的所有组，并存放在grouplist数组中。

|  |
| --- |
| #include <unistd.h>  int getgroups(int gidsetsize, gid\_t grouplist[]);  成功时返回grouplist中的组ID数目，出错返回-1 |

在Linux和多数UNIX实现中，getgroups()只是简单地返回调用进程的附加组ID。但是SUSv3还允许实现在返回的grouplist中包含调用进程的有效组ID。

调用程序必须分配grouplist数组并通过gidsetsize指定它的大小。成功完成时，getgroups()返回存放在grouplist中的组ID数目。

如果进程的组数量超过了gidsetsize，getgroups()返回错误（EINVAL）。为了避免这个问题，我们可以使grouplist数组大于常量NGROUPS\_MAX（定义在<limits.h>中），这个常量是进程可能拥有的组数量的最大值，同时还可以处理包含有效组ID的可移植问题。因此我们可以如下定义grouplist：

gid\_t grouplist[NGROUPS\_MAX + 1];

在Linux内核2.6.4之前，NGROUPS\_MAX的值为32。从内核2.6.4开始，NGROUPS\_MAX的值为65536。

应用还可以在运行时按以下方法确定NGROUPS\_MAX限制的值：

* 调用sysconf(\_SC\_NGROUPS\_MAX)。（我们在11.2节描述sysconf()）
* 从只读的Linux特定的/proc/sys/kernel/ngroups\_max文件中读取限制值，这个文件从内核2.6.4开始提供。

或者应用也可以调用getgroups()并指定gidsetsize为0，在这种情况下，grouplist不会修改，但是返回值将是进程所属的所有组的数量。

使用这些运行时技术获得大小之后，就可以动态分配grouplist数组，然后调用getgroups()系统调用。

特权进程可以使用setgroups()和initgroups()来修改附加组ID集。

|  |
| --- |
| #define \_BSD\_SOURCE  #include <grp.h>  int setgroups(size\_t gidsetsize, const gid\_t \*grouplist);  int initgroups(const char \*user, gid\_t group);  成功时都返回0；出错时都返回-1 |

setgroups()系统调用使用grouplist数组指定的ID集来替换调用进程的附加组ID。gidsetsize参数指定grouplist数组中的组ID数量。

initgroups()函数扫描/etc/groups并构建user用户所属所有组的列表，然后初始化调用进程的附加组ID。此外，group指定的组ID也被添加到进程的附加组ID集。

initgroups()的主要用途是那些创建登录会话的程序，例如login，在执行用户的login shell之前需要设置各种进程属性。这类程序通常读取密码文件中用户记录的组ID域，这有一点令人迷惑，因为密码文件中的组ID实际上不是附加组，相反它定义了login shell的初始实际用户ID、有效用户ID、和保存的设置用户ID。无论如何，这是initgroups()的典型使用方式。

尽管setgroups()和initgroups()系统调用并不是SUSv3规范，但是在所有UNIX实现中都可用。

### 9.7.4 修改进程凭证调用汇总

表9-1汇总了各种修改进程凭证的系统调用和库函数的作用。

图9-1以图形方式提供了表9-1相同的信息。这个图显示了修改用户ID时发生的事情，但是同样适用于修改组ID。



*图9-1：修改进程用户ID函数的作用*

*表9-1：修改进程凭证接口汇总*

|  |  |  |  |
| --- | --- | --- | --- |
| **接口** | **作用和效果** | | **可移植性** |
| **非特权进程** | **特权进程** |
| setuid(u)  setgid(g) | 修改有效ID为当前的实际或保存设置ID | 修改实际、有效、保存设置ID为任意值 | SUSv3规定；BSD派生系统有不同的语义 |
| seteuid(e)  setegid(e) | 修改有效ID为当前的实际或保存设置ID | 修改有效ID为任意值 | SUSv3规定 |
| setreuid(r, e)  setregid(r, e) | （独立地）修改实际ID为当前的实际或有效ID，修改有效ID为当前的实际、有效、保存设置ID | （独立地）修改实际和有效ID为任意值 | SUSv3规定，但是不同实现执行的操作可能会有不同 |
| setresuid(r, e, s)  setresgid(r, e, s) | （独立地）修改实际、有效、保存设置ID为当前的实际、有效、或保存设置ID | （独立地）修改实际、有效、保存设置ID为任意值 | SUSv3没有规定，少数UNIX实现可用 |
| setfsuid(u)  setfsgid(u) | 修改文件系统用户ID为当前的实际、有效、文件系统、保存设置ID | 修改文件系统ID为任意值 | Linux特定 |
| setgroups(n, l) | 非特权进程无法调用 | 设置附加组ID为任意值 | SUSv3没有规定，但是所有UNIX实现都可用 |

注意以下对表9-1的补充信息：

* glibc实现的seteuid()（实现为setresuid(-1, e, -1)）和setegid()（实现为setregid(-1, 3)）允许有效ID设置为当前已经拥有的值，但这不是SUSv3所规定。如果有效用户ID设置为非当前实际用户ID，setegid()实现还会修改保存的设置组ID。（SUSv3没有规定setegid()修改保存的设置组ID）。
* 特权和非特权进程调用setreuid()和setregid()时，如果r不是-1，或者e指定为不同于实际ID的值，则保存的设置用户ID或保存的设置组ID也同时被修改为（新的）有效ID。（SUSv3没有规定setreuid()和setregid()修改保存的设置ID）。
* 只要有效用户（组）ID被修改，Linux特定的文件系统用户（组）ID也将被修改为相同的值。
* 调用setresuid()总是修改文件系统用户ID为有效用户ID，无论调用是否修改了有效用户ID。调用setresgid()对文件系统组ID也有类似的效果。

### 9.7.5 示例：显示进程凭证

清单9-1的程序使用前面讨论的系统调用和库函数，获取进程所有的用户和组ID，然后显示出来。

*清单9-1：显示进程的所有用户和组ID*

-----------------------------------------------------proccred/idshow.c

#define \_GNU\_SOURCE

#include <unistd.h>

#include <sys/fsuid.h>

#include <limits.h>

#include "ugid\_functions.h" /\* userNameFromId() & groupNameFromId() \*/

#include "tlpi\_hdr.h"

#define SG\_SIZE (NGROUPS\_MAX + 1)

int

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

{

uid\_t ruid, euid, suid, fsuid;

gid\_t rgid, egid, sgid, fsgid;

gid\_t suppGroups[SG\_SIZE];

int numGroups, j;

char \*p;

if (getresuid(&ruid, &euid, &suid) == -1)

errExit("getresuid");

if (getresgid(&rgid, &egid, &sgid) == -1)

errExit("getresgid");

/\* Attempts to change the file-system IDs are always ignored

for unprivileged processes, but even so, the following

calls return the current file-system IDs \*/

fsuid = setfsuid(0);

fsgid = setfsgid(0);

printf("UID: ");

p = userNameFromId(ruid);

printf("real=%s (%ld); ", (p == NULL) ? "???" : p, (long) ruid);

p = userNameFromId(euid);

printf("eff=%s (%ld); ", (p == NULL) ? "???" : p, (long) euid);

p = userNameFromId(suid);

printf("saved=%s (%ld); ", (p == NULL) ? "???" : p, (long) suid);

p = userNameFromId(fsuid);

printf("fs=%s (%ld); ", (p == NULL) ? "???" : p, (long) fsuid);

printf("\n");

printf("GID: ");

p = groupNameFromId(rgid);

printf("real=%s (%ld); ", (p == NULL) ? "???" : p, (long) rgid);

p = groupNameFromId(egid);

printf("eff=%s (%ld); ", (p == NULL) ? "???" : p, (long) egid);

p = groupNameFromId(sgid);

printf("saved=%s (%ld); ", (p == NULL) ? "???" : p, (long) sgid);

p = groupNameFromId(fsgid);

printf("fs=%s (%ld); ", (p == NULL) ? "???" : p, (long) fsgid);

printf("\n");

numGroups = getgroups(SG\_SIZE, suppGroups);

if (numGroups == -1)

errExit("getgroups");

printf("Supplementary groups (%d): ", numGroups);

for (j = 0; j < numGroups; j++) {

p = groupNameFromId(suppGroups[j]);

printf("%s (%ld) ", (p == NULL) ? "???" : p, (long) suppGroups[j]);

}

printf("\n");

exit(EXIT\_SUCCESS);

}

-----------------------------------------------------proccred/idshow.c

## 9.8 小结

每个进程都有一组相关联的用户和组ID（凭证）。实际ID定义了进程的拥有者。在多数UNIX实现中，有效ID用来确定进程访问系统资源（如文件）时的权限。但是在Linux中，访问文件时将使用文件系统ID来确定权限，而有效ID则用于其它权限检查。（因为文件系统ID通常与有效ID相同，在检查访问文件权限时Linux与其它UNIX实现的行为是一致的）。进程的附加组ID是进程属于的额外组集合，目的也是用于权限检查。许多系统调用和库函数允许进程获取和修改自己的用户和组ID。

当设置用户ID程序运行时，进程的有效用户ID被修改为文件的拥有者。这个机制允许用户假定自己的身份（特权用户或其它用户），从而以其它用户ID来运行特定的程序。同样，设置组ID程序修改进程的有效组ID为程序所属的组。保存的设置用户ID和保存的设置组ID允许设置用户ID和设置组ID程序临时丢弃然后重新获取特权。

用户ID 0很特殊，通常root用户拥有这个ID。有效用户ID为0的进程是特权进程（进程调用各种系统调用时，可以免除许多权限检查，例如可以任意修改进程的各种用户和组ID）。

## 9.9 习题

9-1. 假设在每次执行下面调用之前，进程的初始用户ID都为：real=1000 effective=0 saved=0 file-system=0。请问在执行以下调用之后用户ID的状态如何？

a) setuid(2000);

b) setreuid(–1, 2000);

c) seteuid(2000);

d) setfsuid(2000);

e) setresuid(–1, 2000, 3000);

9-2. 进程拥有以下用户ID，请问进程是否拥有特权，请解释。

real=0 effective=1000 saved=1000 file-system=1000

9-3. 使用setgroups()和获取密码和组文件信息的库函数（参考8.4节），实现initgroups()。记住进程必须拥有特权才能调用setgroups()。

9-4. 如果进程的用户ID全部为X，执行用户Y的设置用户ID程序（Y非0），则进程的凭证将被设置为如下：

real=X effective=Y saved=Y

（我们忽略了文件系统用户ID，因为它和有效用户ID一样）。分别使用setuid(), seteuid(), setreuid(), setresuid()，可以执行以下操作：

1. 挂起和重获设置用户ID身份（也就是切换有效用户ID为实际用户ID，然后再切回保存的设置用户ID）。
2. 永久地丢弃设置用户ID身份（也就是确保有效用户ID和保存的设置用户ID都设置为实际用户ID）。

（这个练习还需要使用getuid()和geteuid()来获取进程的实际和有效用户ID）。注意上面列出来的某些系统调用，可能无法执行某些操作。

9-5. 重复上面的练习，但进程执行set-user-ID-root程序，拥有以下初始凭证：

real=X effective=0 saved=0

# 第10章 时间

在程序中，我们可能会对以下两种时间感兴趣：

* 真实时间：这是以某个标准时间点（日历时间）或某个固定点（一般是起点）来测量进程的生命（逝去时间或时钟时间）。获取日历时间是很有用的，例如数据库或文件的时间戳。测量逝去时间对于程序同样是很有用的，特别是那些执行定期动作或定期观测外部输入设备的程序。
* 进程时间：这是进程使用的CPU时间总量。测量进程时间对于检查或优化程序或算法的性能非常有用。

多数计算机体系架构都有内建的硬件时钟，允许内核测量真实和进程时间。在本章，我们讨论处理这两种时间的系统调用，以及在人类可读和内部表示的时间之间转换的库函数。由于人类可读的时间格式依赖于地理位置及语言和文化惯例，对于这些时间格式我们需要学习时区和区域。

## 10.1 日历时间

无论是什么地理位置，UNIX系统内部对时间的表示，都是从Epoch开始持续的秒数；也就是1970年1月1日午夜，协调世界时间（UTC，以前叫格林威治平均时间GMT）。这个日期大致就是UNIX系统诞生之日。日历时间存储在类型time\_t中，SUSv3规定为整数类型。

在32位Linux系统中，time\_t是带符号整数，可以表示的日期范围是1901-12-13 20:45:52到2038-01-19 03:14:07。（SUSv3没有规定负数time\_t的含义）。因此目前的许多32位UNIX系统理论上面临2038年问题，如果基于未来的日期进行计算的话，还可能在2038年之前就碰到这个问题。不过实际上这可能不会成为真正的问题，因为到2038年多数UNIX系统都将是64位或者更高。不过32位嵌入式系统，比桌面硬件拥有长得多的生命周期，可能仍然会面临这个问题。此外，这个问题对于继续使用32位time\_t格式的遗留数据和应用也仍然存在。

gettimeofday()系统调用在tv指向的缓冲区中返回日历时间。

|  |
| --- |
| #include <sys/time.h>  int gettimeofday(struct timeval \*tv, struct timezone \*tz);  成功时返回0；出错时返回-1 |

tv参数是指向以下结构体的指针：

struct timeval {

time\_t tv\_sec; /\* 1970-01-01 00:00:00 UTC开始的秒数 \*/

suseconds\_t tv\_usec; /\* 额外的微秒（长整型）\*/

};

尽管tv\_usec域提供了微秒精度，但实际精度取决于具体的体系架构实现。在现代x86-32系统中（如奔腾系统拥有时间戳计数寄存器，在每个CPU时钟周期递增一次），gettimeofday()确实提供微秒精度。

gettimeofday()的tz参数是一个历史产物。在早期UNIX实现中，它用来获取系统的时区信息。现在这个参数已经废弃并且应该指定为NULL。

time()系统调用返回从Epoch开始过去的秒数（等同于gettimeofday()返回的tv参数的tv\_sec域）。

|  |
| --- |
| #include <time.h>  time\_t time(time\_t \*timep);  返回从Epoch开始过去的秒数，出错时返回(time\_t)-1 |

如果timep参数不是NULL，则秒数还会放在timep指向的位置。

由于time()在两个地方返回相同的值，而且唯一可能的错误是timep参数地址非法（EFAULT），因此我们通常简单地使用以下调用形式（不进行错误检查）：

t = time(NULL);

time()和gettimeofday()本质上作用是一样的，出现两个系统调用主要是历史原因。早期UNIX实现提供time()。4.3BSD增加了更高精度的gettimeofday()系统调用。现在time()作为系统调用已经是多余的；它可能被实现为库函数，内部调用gettimeofday()。

## 10.2 时间转换函数

图10-1显示了在time\_t和其它时间格式之间转换的函数，包括可打印的时间格式。这些函数帮助我们处理了复杂的细节，例如时区转换、夏令时（DST）、以及区域等问题。（我们在10.3节描述时区，10.4节描述区域）。



*图10-1：获取和转换日历时间的函数*

### 10.2.1 time\_t转换为可打印格式

ctime()函数提供了一种简单的转换time\_t为可打印格式的方法。

|  |
| --- |
| #include <time.h>  char \*ctime(const time\_t \*timep);  成功时返回指向静态分配的字符串，以换行和\0结尾  出错时返回NULL |

在timep中指定一个time\_t指针，ctime()返回26字节的字符串，包含标准格式的日期和时间，如下所示：

Wed Jun 8 14:22:34 2011

这个字符串包含换行和null终止符。ctime()函数在执行转换时自动计算本地时区和DST设置。（我们在10.3节解释如何确定这些设置）。返回的字符串是静态分配的，后续的ctime()调用会覆盖它。

SUSv3规定调用ctime(), gmtime(), localtime(), asctime()中的任何一个都可能覆盖之前任何函数返回的静态分配的结构体。换句话说，这些函数可能共享同一个返回字符串数组和tm结构体，而且在某些glibc版本中也确实是这样实现的。如果我们需要在不同函数的多个调用之间维护返回信息，我们必须复制到本地。

ctime\_r()提供了可重入版本的ctime()。（我们在21.1.2节讨论可重入）。这个函数允许调用方指定一个额外的参数，指向调用方分配的缓冲区，用于返回时间字符串。本章提及的其它可重入函数也都类似。

### 10.2.2 time\_t和分解时间互相转换

gmtime()和localtime()函数转换time\_t为分解（Broken-Down）时间。分解时间被存放在静态分配的结构体中，并作为函数结果返回。

|  |
| --- |
| #include <time.h>  struct tm \*gmtime(const time\_t \*timep);  struct tm \*localtime(const time\_t \*timep);  成功时返回静态分配的分解时间结构体指针，出错时返回NULL |

gmtime()函数根据相应的UTC转换日历时间为分解时间。（字母gm表示Greenwich Mean Time）。相反，localtime()则会计算时区和DST设置，并返回系统本地的分解时间。

这些函数的可重入版本分别是gmtime\_r()和localtime\_r()。

函数返回的tm结构体分解时间为单个部分，包括日期和时间域。这个结构体定义如下：

struct tm {

int tm\_sec; /\* 秒(0-60) \*/

int tm\_min; /\* 分钟(0-59) \*/

int tm\_hour; /\* 小时(0-23) \*/

int tm\_mday; /\* 月的天数(1-31) \*/

int tm\_mon; /\* 月份(0-11) \*/

int tm\_year; /\* 从1900开始的年数 \*/

int tm\_wday; /\* 周的天数(星期天 = 0)\*/

int tm\_yday; /\* 年的天数(0-365; 1月1日 = 0)\*/

int tm\_isdst; /\* 夏令时标志

> 0: DST 有效;

= 0: DST 无效;

< 0: DST 信息不可用 \*/

};

tm\_sec域最大可以是60（而不是59），用来计算润秒，偶尔应用于调整日历的Astronomically Exact Year（也称为Tropical）。

如果定义了\_BSD\_SOURCE测试宏，则glibc对tm结构体的定义还包含两个额外的域，用于描述更多的时间信息。第一个是long int tm\_gmtoff，包含表示falls east UTC时间的秒数。第二个是const char \*tm\_zone，是时区名的简称（例如CEST代表Central European Summer Time）。SUSv3没有规定这两个域，而且它们只在少数其它UNIX实现中可用（主要是BSD系列）。

mktime()函数处理本地时间表示的分解时间，将其转换为time\_t值，并作为函数结果返回。调用方使用tm结构体指针timeptr提供分解时间。在转换的过程中，tm\_wday和tm\_yday域将被忽略。

|  |
| --- |
| #include <time.h>  time\_t mktime(struct tm \*timeptr);  成功时返回对应于timeptr的Epoch开始的秒数；出错时返回(time\_t) -1 |

mktime()函数可能会修改timeptr指向的结构体。至少会根据其它输入域确保正确地设置tm\_wday和tm\_yday域。

此外，mktime()不要求tm结构体的其它域严格地限制于前面描述的范围内。对于每个超出范围的域，mktime()会调整该域的值，适当地设置该域，并调整其它域。mktime()先调整这些域的值，然后更新tm\_wday和tm\_yday域，并计算和返回time\_t值。

例如输入的tm\_sec域为123，则在函数返回时，该域的值为3，并且tm\_min域会相应的增加2。（如果tm\_min又超出范围，则tm\_min的值也会被调整，并增加到tm\_hour域，依此类推）。这些调整甚至会应用于负数值，例如tm\_sec指定为-1，表示上一分钟的59秒。这个特性非常有用，允许我们对分解时间执行日期和时间计算。

mktime()在执行转换时会使用时区设置，此外根据tm\_isdst域的值，可能会或不会使用DST设置：

* 如果tm\_isdst为0：按标准时间对待（忽略DST，即使当前时间的DST有效）。
* 如果tm\_isdst大于0：按DST时间处理（DST有效，即使当前时间并不是DST）。
* 如果tm\_isdst小于0：尝试根据当前时间来确定DST是否有效，这通常是我们希望使用的值。

完成时（无论tm\_isdst最初的设置为何值），mktime()都会正确地设置tm\_isdst域，如果DST有效则为正值，DST无效则设置为0。

### 10.2.3 分解时间和可打印格式互相转换

在这一节，我们讨论分解时间和可打印格式的互相转换。

**转换分解时间为可打印格式**

asctime()函数的参数为指向分解时间结构体的指针，返回静态分配的字符串，时间格式与ctime()函数相同。

|  |
| --- |
| #include <time.h>  char \*asctime(const struct tm \*timeptr);  成功时返回静态分配的字符串，以换行和\0结尾；出错时返回NULL |

相比ctime()，本地时区设置对asctime()没有作用，因为asctime()转换的分解时间要么已经通过localtime()本地化，要么是gmtime()返回的UTC时间。

和ctime()一样，我们也无法控制asctime()函数返回的字符串格式。

asctime()的可重入版本是asctime\_r()

清单10-1演示了asctime()的使用，以及本章目前为止讨论过的所有时间转换函数。这个程序获取当前日历时间，然后使用这些时间转换函数，并显示它们的返回结果。下面是程序的运行结果示例，在Munich, Germany，Central European Time，比UTC早一个小时：

$ date

Tue Dec 28 16:01:51 CET 2010

$ ./calendar\_time

Seconds since the Epoch (1 Jan 1970): 1293548517 (about 40.991 years)

gettimeofday() returned 1293548517 secs, 715616 microsecs

Broken down by gmtime():

year=110 mon=11 mday=28 hour=15 min=1 sec=57 wday=2 yday=361 isdst=0

Broken down by localtime():

year=110 mon=11 mday=28 hour=16 min=1 sec=57 wday=2 yday=361 isdst=0

asctime() formats the gmtime() value as: Tue Dec 28 15:01:57 2010

ctime() formats the time() value as: Tue Dec 28 16:01:57 2010

mktime() of gmtime() value: 1293544917 secs

mktime() of localtime() value: 1293548517 secs 3600 secs ahead of UTC

*清单10-1：获取和转换日历时间*

--------------------------------------------------time/calendar\_time.c

#include <locale.h>

#include <time.h>

#include <sys/time.h>

#include "tlpi\_hdr.h"

#define SECONDS\_IN\_TROPICAL\_YEAR (365.24219 \* 24 \* 60 \* 60)

int

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

{

time\_t t;

struct tm \*gmp, \*locp;

struct tm gm, loc;

struct timeval tv;

t = time(NULL);

printf("Seconds since the Epoch (1 Jan 1970): %ld", (long) t);

printf(" (about %6.3f years)\n", t / SECONDS\_IN\_TROPICAL\_YEAR);

if (gettimeofday(&tv, NULL) == -1)

errExit("gettimeofday");

printf(" gettimeofday() returned %ld secs, %ld microsecs\n",

(long) tv.tv\_sec, (long) tv.tv\_usec);

gmp = gmtime(&t);

if (gmp == NULL)

errExit("gmtime");

gm = \*gmp; /\* Save local copy, since \*gmp may be modified

by asctime() or gmtime() \*/

printf("Broken down by gmtime():\n");

printf(" year=%d mon=%d mday=%d hour=%d min=%d sec=%d ", gm.tm\_year,

gm.tm\_mon, gm.tm\_mday, gm.tm\_hour, gm.tm\_min, gm.tm\_sec);

printf("wday=%d yday=%d isdst=%d\n",

gm.tm\_wday, gm.tm\_yday, gm.tm\_isdst);

locp = localtime(&t);

if (locp == NULL)

errExit("localtime");

loc = \*locp; /\* Save local copy \*/

printf("Broken down by localtime():\n");

printf(" year=%d mon=%d mday=%d hour=%d min=%d sec=%d ",

loc.tm\_year, loc.tm\_mon, loc.tm\_mday,

loc.tm\_hour, loc.tm\_min, loc.tm\_sec);

printf("wday=%d yday=%d isdst=%d\n\n",

loc.tm\_wday, loc.tm\_yday, loc.tm\_isdst);

printf("asctime() formats the gmtime() value as: %s", asctime(&gm));

printf("ctime() formats the time() value as: %s", ctime(&t));

printf("mktime() of gmtime() value: %ld secs\n", (long) mktime(&gm));

printf("mktime() of localtime() value: %ld secs\n",

(long) mktime(&loc));

exit(EXIT\_SUCCESS);

}

--------------------------------------------------time/calendar\_time.c

strftime()函数也是转换分解时间为可打印格式，但是给我们提供更多精确的控制。timeptr参数指向分解时间结构体，strftime()把相应的日期时间字符串，以及null终止符放在outstr指向的缓冲区中。

|  |
| --- |
| #include <time.h>  size\_t strftime(char \*outstr, size\_t maxsize, const char \*format,  const struct tm \*timeptr);  成功时返回outstr中存放的字节数（不包括null终止符）；  出错时返回0 |

outstr返回的字符串，根据format参数进行格式化。maxsize参数指定outstr缓冲区的最大可用空间。和ctime(), asctime()不一样，strftime()不在返回字符串中增加换行字符（除非format中指定）。

成功时strftime()返回outstr中存放的字节数，不包括null终止符。如果转换后的字符串总长度（包括null终止符）超过了maxsize字节，则strftime()返回0表示出现错误，此时outstr中的内容是不确定的。

strftime()的format参数是类似于printf()的字符串。以%开始的字符是转换格式说明符，根据不同的说明符，将使用相应的日期和时间替换。函数提供了非常丰富的转换说明符，表10-1列出了其中的一个子集（完整的列表，请参考strftime(3)手册页）。除非单独标注，所有转换说明符都由SUSv3标准化。

%U和%W说明符都生成当年的周数。%U周数目按第一个周日计算，数值为1，前面的部分周则为0；如果周日刚好是该年的第一天，则没有为0的周，当年的最后一天则是第53周。%W周数目按相同的方式计算，但是使用周一而不是周日。

我们在本书的各种演示程序中需要显示当前时间。因此我们提供了currTime()函数，返回一个以strftime()函数格式化的字符串。

|  |
| --- |
| #include "curr\_time.h"  char \*currTime(const char \*format);  成功时返回静态分配的字符串指针；出错时返回NULL |

清单10-2显示了currTime()函数的实现。

*表10-1：挑选的strftime()转换说明符*



*清单10-2：返回当前时间字符串的函数*

------------------------------------------------------time/curr\_time.c

#include <time.h>

#include "curr\_time.h" /\* Declares function defined here \*/

#define BUF\_SIZE 1000

/\* Return a string containing the current time formatted according to

the specification in 'format' (see strftime(3) for specifiers).

If 'format' is NULL, we use "%c" as a specifier (which gives the

date and time as for ctime(3), but without the trailing newline).

Returns NULL on error. \*/

char \*

currTime(const char \*format)

{

static char buf[BUF\_SIZE]; /\* Nonreentrant \*/

time\_t t;

size\_t s;

struct tm \*tm;

t = time(NULL);

tm = localtime(&t);

if (tm == NULL)

return NULL;

s = strftime(buf, BUF\_SIZE, (format != NULL) ? format : "%c", tm);

return (s == 0) ? NULL : buf;

}

------------------------------------------------------time/curr\_time.c

**转换可打印格式为分解时间**

strptime()函数的功能与strftime()相反，它转换日期+时间字符串为分解时间。

|  |
| --- |
| #define \_XOPEN\_SOURCE  #include <time.h>  char \*strptime(const char \*str, const char \*format, struct tm \*timeptr);  成功时返回指向str中下一个未处理字符的指针；出错返回NULL |

strptime()函数使用format指定的说明符，来解析str提供的日期时间字符串，并把转换后的分解时间存放在timeptr指向的结构体中。

成功时，strptime()返回指向str中下一个未处理字符的指针。（如果字符串包含额外的信息，需要调用程序继续处理，则这个返回非常有用）。如果不能匹配完整的格式字符串，strptime()返回NULL表示错误。

strptime()的format参数类似于scanf(3)。它包含以下几种字符：

* 以%开始的转换说明符；
* 空白字符，匹配输入字符串中的0个或多个空白；
* 非空白字符（也不是%），必须完全匹配输入字符串的相同字符。

转换说明符类似于strftime()（表10-1）。主要的差别在于这里的说明符更加通用。例如%a和%A都接受周天的名字，不管是简称还是全称；%d和%e可以用来读取当月的天数，单个字符时可以有前缀0，或者没有前缀0。此外忽略大小写；例如May和MAY是等价的月名。%%用来匹配输入字符串中的%。strptime(3)手册页提供了更多细节。

glibc的strptime()实现，不会修改tm结构体中那些format没有设置的域。这意味着我们可以采用一系列的strptime()调用从多个输入字符串来构造单个tm结构体，例如日期字符串和时间字符串。SUSv3允许这个行为，但也没有要求这个行为，因此我们在其它UNIX实现中不能依赖它。在可移植应用中，我们必须确保str和format包含的输入会正确地设置tm结构体的所有域，或者确保tm结构体在调用strptime()之前适当地初始化。多数情况下，使用memset()将tm全部初始化为0就足够了，但是要知道在glibc和许多其它时间转换函数的实现中，tm\_mday域的值为0表示上一个月的最后一天。最后，注意strptime()永远不会设置tm结构体的tm\_isdst域。

GNU C库还提供两个其它的函数，作用和strptime()类似：getdate()（SUSv3规定，广泛可用）和可重入版本getdate\_r()（SUSv3未规定，少数其它UNIX实现可用）。我们不讨论这两个函数，因为它们采用外部文件（由环境变量DATEMSK标识）来指定扫描日期的格式，这使得函数难于使用，同时在设置用户ID程序中还可能导致安全缺陷。

清单10-3演示了strptime()和strftime()的使用。这个程序接收命令行参数，包含日期和时间，并使用strptime()将其转换为分解时间。然后再使用strftime()进行反向转换并显示其结果。程序最多接受三个参数，前面两个是必须的。第一个参数是包含日期和时间的字符串；第二个参数是strptime()用来解析第一个参数的格式说明符。可选的第三个参数是strftime()用来反转的格式说明符，如果忽略了这个参数，则使用默认的格式字符串。（我们在10.4节讨论这里使用的setlocale()函数）。下面是在shell中使用这个程序的示例：

$ ./strtime "9:39:46pm 1 Feb 2011" "%I:%M:%S%p %d %b %Y"

calendar time (seconds since Epoch): 1296592786

strftime() yields: 21:39:46 Tuesday, 01 February 2011 CET

下面用法类似，但是我们显式地指定strftime()的格式说明符：

$ ./strtime "9:39:46pm 1 Feb 2011" "%I:%M:%S%p %d %b %Y" "%F %T"

calendar time (seconds since Epoch): 1296592786

strftime() yields: 2011-02-01 21:39:46

*清单10-3：获取和转换日历时间*

---------------------------------------------------------time/strtime.c

#define \_XOPEN\_SOURCE

#include <time.h>

#include <locale.h>

#include "tlpi\_hdr.h"

#define SBUF\_SIZE 1000

int

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

{

struct tm tm;

char sbuf[SBUF\_SIZE];

char \*ofmt;

if (argc < 3 || strcmp(argv[1], "--help") == 0)

usageErr("%s input-date-time in-format [out-format]\n", argv[0]);

if (setlocale(LC\_ALL, "") == NULL)

errExit("setlocale"); /\* Use locale settings in conversions \*/

memset(&tm, 0, sizeof(struct tm)); /\* Initialize 'tm' \*/

if (strptime(argv[1], argv[2], &tm) == NULL)

fatal("strptime");

tm.tm\_isdst = -1; /\* Not set by strptime(); tells mktime()

to determine if DST is in effect \*/

printf("calendar time (seconds since Epoch): %ld\n",

(long) mktime(&tm));

ofmt = (argc > 3) ? argv[3] : "%H:%M:%S %A, %d %B %Y %Z";

if (strftime(sbuf, SBUF\_SIZE, ofmt, &tm) == 0)

fatal("strftime returned 0");

printf("strftime() yields: %s\n", sbuf);

exit(EXIT\_SUCCESS);

}

---------------------------------------------------------time/strtime.c

## 10.3 时区

不同的国家（有时候甚至是同一个国家的不同区域）拥有不同的时区和DST状态。输入和输出时间的程序必须考虑运行系统的时区和DST状态。幸运的是，C库为我们处理了所有细节。

**时区定义**

时区信息很多并且易变，因此程序或库不应对时区信息硬编码，而应由系统按标准格式在文件中维护时区信息。

这个文件存放在目录/usr/share/zoneinfo中。该目录下的每个文件都包含特定国家或区域的时区信息。这些文件根据所描述的时区来命名，因此我们可以用类似EST（US东部标准时间）、CET（欧洲中部时间）、UTC、Turkey、Iran这样的名字来找到相应的文件。此外子目录还可以用来层次化地组织相关的时区。例如在Pacific子目录中，我们可以找到Auckland、Port\_Moresby、Galapagos等文件。当我们在程序中指定时区时，实际上指定的是某个时区文件在目录中的相对路径。

系统的本地时间由时区文件/etc/localtime文件定义，通常链接到/usr/share/zoneinfo下面的某个文件。

时区文件的格式在tzfile手册页中有描述。时区文件使用zic（zone information compiler）来创建。zdump命令可以根据某个特定的时区文件来显示时间。

**为程序指定时区**

如果运行程序时需要指定时区，我们可以设置TZ环境变量，格式为冒号(:)加上/usr/share/zoneinfo下面定义的某个时区名。设置时区之后，会自动影响到ctime(), localtime(), mktime(), strftime()函数。

这四个函数都使用tzset()函数来获取当前的时区设置，后者会初始化三个全局变量：

char \*tzname[2]; /\* 时区或DST时区的名字 \*/

int daylight; /\* 非0表示DST时区 \*/

long timezone; /\* UTC和本地标准时间的秒差 \*/

tzset()函数首先检查TZ环境变量，如果这个环境变量没有设置，则时区将初始化为/etc/localtime定义的默认时区。如果TZ环境变量定义为某个不能匹配系统时区文件的值，或者是空字符串，则使用UTC。TZDIR环境变量（GNU的非标准扩展）可以设置为自定义时区目录，用来替代默认的/usr/share/zoneinfo目录。

我们可以通过运行清单10-4的程序来查看TZ环境变量的作用。第一次运行时，我们看到输出对应于系统的默认时区（欧洲中央时区CET）。第二次运行时，我们指定时区为新西兰，当前为夏令时，比CET快12个小时。

$ ./show\_time

ctime() of time() value is: Tue Feb 1 10:25:56 2011

asctime() of local time is: Tue Feb 1 10:25:56 2011

strftime() of local time is: Tuesday, 01 Feb 2011, 10:25:56 CET

$ TZ=":Pacific/Auckland" ./show\_time

ctime() of time() value is: Tue Feb 1 22:26:19 2011

asctime() of local time is: Tue Feb 1 22:26:19 2011

strftime() of local time is: Tuesday, 01 February 2011, 22:26:19 NZDT

*清单10-4：演示时区和locale的作用*

-------------------------------------------------------time/show\_time.c

#include <time.h>

#include <locale.h>

#include "tlpi\_hdr.h"

#define BUF\_SIZE 200

int

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

{

time\_t t;

struct tm \*loc;

char buf[BUF\_SIZE];

if (setlocale(LC\_ALL, "") == NULL)

errExit("setlocale"); /\* Use locale settings in conversions \*/

t = time(NULL);

printf("ctime() of time() value is: %s", ctime(&t));

loc = localtime(&t);

if (loc == NULL)

errExit("localtime");

printf("asctime() of local time is: %s", asctime(loc));

if (strftime(buf, BUF\_SIZE, "%A, %d %B %Y, %H:%M:%S %Z", loc) == 0)

fatal("strftime returned 0");

printf("strftime() of local time is: %s\n", buf);

exit(EXIT\_SUCCESS);

}

-------------------------------------------------------time/show\_time.c

SUSv3定义了两种通用的方法来设置TZ环境变量。正如上面描述，TZ可以设置为冒号加字符串，以具体实现相关的方式来标识时区，通常就是时区描述文件的路径（Linux和其它UNIX实现允许忽略冒号，但SUSv3并没有规定；为了提高可移植性，我们应该包含冒号）。

SUSv3详细地规定了TZ的另一种设置方法，在这个方法中，我们把TZ设置为以下格式的字符串：

std offset [ dst [ offset ][ , start-date [ /time ] , end-date [ /time ]]]

上面的空格只是为了看起来更加清晰，设置TZ时不能包含空格。中括号表示可选组件。

std和dst组件是标识标准和DST时区的字符串，例如CET和CEST分别是欧洲中央时间和欧洲夏令时间。offset则指定转换本地时间为UTC时使用的正或负的调整值。最后的四个组件描述标准时间转换为DST时，所使用的规则。

日期可以指定为许多形式，其中之一是Mm.n.d。这表示天d（0=星期天，6=星期六）周n（1至5，5总是表示最后n天）月m（1至12）。如果忽略时间，则默认为02:00:00（2AM）。

下面是欧洲中央时区的TZ定义，标准时间比UTC快一个小时，DST比UTC早两个小时：

TZ="CET-1:00:00CEST-2:00:00,M3.5.0,M10.5.0

我们忽略了DST时间逆向转换的规格说明，因为它默认发生于02:00:00。当然上面这个TZ明显没有下面这个Linux特定的定义清晰：

TZ=":Europe/Berlin"

## 10.4 Locale

世界上有几千种语言，而且有很大一部分已经广泛使用于计算机系统中。此外不同的国家还使用不同的信息显示惯例，例如数字、货币数额、日期、时间等。例如在多数欧洲国家，使用逗号而不是小数点，来分离数字的整数和小数部分，而且多数国家使用的日期格式也不同于美国使用的MM/DD/YY。SUSv3对locale的定义是：语言和文化惯例相关的用户环境子集。

理想情况下，所有设计运行于多个地区的程序，都应该处理locale，按用户首选的语言和格式来显示和输入信息。这也就组成了复杂的国际化。在理想世界里，我们可以编写程序一次，然后无论在哪里运行，执行I/O时都能自动处理locale，也就是执行本地化的工作。国际化一个程序有时候是非常耗时的任务，尽管我们可以使用各种工具来简化国际化的难度。glibc等程序库也提供帮助国际化的工具和设施。

术语Internationalization通常简称为I18N，表示I加上18个字母再加上N。除了更容易写之外，I18N还能避免英国英语和美国英语之间的拼写差异。

**Locale定义**

和时区信息一样，Locale信息也倾向于繁杂和易变。因此程序和库也不存储Locale信息，而由系统在文件中以标准格式维护这些信息。

Locale信息维护在目录层次/usr/share/locale（某些发行版使用/usr/lib/locale）中。每个子目录包含特定locale的信息。这些子目录命名格式如下：

language[\_territory[.codeset]][@modifier]

language是两个字母的ISO语言代码，territory是两个字母的ISO国家代码。codeset指定字符编码集。modifier指定一个名字，用于区分多个相同语言、国家、字符集的locale目录。de\_DE.utf-8@euro就是一个完整的locale目录名示例，代表：德语、德国、UTF-8字符编码，采用欧元作为货币单位。

locale目录命名格式中的中括号表示可以忽略。通常名字只包含语言和国家。因此en\_US目录表示（说英语）的美国locale，而fr\_CH目录则表示（说法语）的瑞士。

当我们在程序中指定使用的locale时，实际上指定的是/usr/share/locale目录下某个子目录名。如果程序指定的locale无法完全匹配/usr/share/locale下面的子目录，C库会丢弃指定locale的某些组成来搜索部分匹配，顺序如下：

1. codeset
2. 规格化的codeset
3. territory
4. modifier

规格化的codeset是对codeset移除所有非字母数字，把所有字母转换为小写，并加上iso前缀的结果字符串。规格化的目的在于处理codeset名的大写和标点等变种。

举个例子说明这个剥离过程，如果程序指定locale为fr\_CH.utf-8，但是不存在这个名字的子目录。接下来将会匹配fr\_CH目录，如果fr\_CH仍然不存在，则匹配fr目录。如果fr目录依旧不存在，则setlocale()函数将会报告错误。

在每个locale子目录中，存放了一组标准文件，规定该locale使用的惯例，如表10-2所示。

*表10-2：locale子目录的内容*

|  |  |
| --- | --- |
| **文件名** | **作用** |
| LC\_CTYPE | 这个文件包含字符分类（参考isalpha(3)）和大小写转换的规则 |
| LC\_COLLATE | 这个文件包含字符集的校对规则 |
| LC\_MONETARY | 这个文件包含货币数值（参考localeconv(3)和<locale.h>）的格式化规则 |
| LC\_NUMERIC | 这个文件包含非货币数值（参考localeconv(3)和<locale.h>）的格式化规则 |
| LC\_TIME | 这个文件包含日期和时间的格式化规则 |
| LC\_MESSAGES | 这个目录下的文件规定用于确认和否认（yes/no）响应的格式和值 |

请注意以下对上表的额外补充信息：

* LC\_COLLATE文件定义了一组规则，描述某个字符集中的字符如何排序（也就是字符集的字母顺序）。这些规则决定了strcoll()和strxfrm()函数的操作。即使是同样起源于拉丁语系的语言，也并不遵守相同的排序规则。例如几种欧洲语言在字母Z之后还有几个额外的字母。其它特殊情况还包括西班牙的两个字母序列ll，排序紧随字母l；以及德语元音变音字符，如ä，对应于ae的排序。
* LC\_MESSAGES目录是程序显示消息时使用的国际化机制。多数国际化程序消息可以使用消息分类（参考catopen(3)和catgets(3)手册页）或者GNU gettext API（<http://www.gnu.org/>）来完成。

系统中实际定义的locale可能存在差异。SUSv3并没有对此做任何要求，只要求必须定义POSIX标准locale。这个locale镜像了UNIX系统对locale的历史行为。因此它基于ASCII字符集，使用英语来命名日期和月份，以及yes/no响应。这个locale的货币和数值没有定义。

locale命令显示当前locale环境相关的信息（在shell中）。locale -a命令列出系统定义的所有locale集。

**程序中指定locale**

setlocale()函数用来查询和设置程序的当前locale。

|  |
| --- |
| #include <locale.h>  char \*setlocale(int category, const char \*locale);  成功时返回新设置或当前locale的字符串（通常是静态分配的）；出错返回NULL |

category参数选择locale的哪个部分需要设置或查询，指定为一组常量中的某个，名字和表10-2中所列一样。例如我们可以设置locale的时间显示为德国，而设置locale的货币显示为US美元。不过我们通常都是使用LC\_ALL来指定修改locale的所有方面。

使用setlocale()设置locale有两种不同的方法。locale参数可以指定为系统中定义的某个locale字符串（也就是/usr/lib/locale目录下的某个子目录名），例如de\_DE或en\_US。或者我们也可以指定locale为空字符串，表示从环境变量中取得locale的设置：

setlocale(LC\_ALL, "");

我们必须调用setlocale之后，程序才会认识并访问到locale环境变量。如果忽略这个调用，则这些环境变量对程序是没有任何作用的。

当运动调用了setlocale(LC\_ALL, "");的程序时，我们可以使用一组环境变量来控制程序的locale属性，变量名同样对应于表10-2中所列：LC\_CTYPE, LC\_COLLATE, LC\_MONETARY, LC\_NUMERIC, LC\_TIME, LC\_MESSAGES。另外我们还可以使用LC\_ALL或LANG环境变量来指定设置整个locale。如果上面所说的多个变量都设置，则LC\_ALL优先于所有其它LC\_\*环境变量，而LANG则拥有最低的优先级。因此我们可以使用LANG来为所有分类设置默认locale，然后再使用单独的LC\_\*环境变量设置locale的具体方面。

setlocale()返回locale的category类别相应的字符串值（通常是静态分配的）。如果我们只想查询当前locale设置而不修改它，可以指定locale参数为NULL。

locale设置控制了许多GNU/Linux工具，以及许多glibc函数的操作，如strftime()和strptime()（10.2.3节），在不同locale下时，清单10-4的程序中这些函数的结果是不一样的：

$ LANG=de\_DE ./show\_time German locale

ctime() of time() value is: Tue Feb 1 12:23:39 2011

asctime() of local time is: Tue Feb 1 12:23:39 2011

strftime() of local time is: Dienstag, 01 Februar 2011, 12:23:39 CET

下面演示了LC\_TIME优先于LANG：

$ LANG=de\_DE LC\_TIME=it\_IT ./show\_time German and Italian locales

ctime() of time() value is: Tue Feb 1 12:24:03 2011

asctime() of local time is: Tue Feb 1 12:24:03 2011

strftime() of local time is: martedì, 01 febbraio 2011, 12:24:03 CET

下面则演示了LC\_ALL优先于LC\_TIME：

$ LC\_ALL=fr\_FR LC\_TIME=en\_US ./show\_time French and US locales

ctime() of time() value is: Tue Feb 1 12:25:38 2011

asctime() of local time is: Tue Feb 1 12:25:38 2011

strftime() of local time is: mardi, 01 février 2011, 12:25:38 CET

## 10.5 更新系统时钟

现在我们来讨论两个更新系统时钟的接口：settimeofday()和adjtime()。应用程序很少需要使用这些接口（因为系统时间通常由网络时间协议daemon等工具维护），而且修改系统时间需要调用方拥有特权（CAP\_SYS\_TIME）。

settimeofday()系统调用执行gettimeofday()（10.1节讨论过了）相反的操作：设置系统的日历时间为timeval结构体tv指定的值。

|  |
| --- |
| #define \_BSD\_SOURCE  #include <sys/time.h>  int settimeofday(const struct timeval \*tv, const struct timezone \*tz);  成功时返回0；出错返回-1 |

和gettimeofday()一样，tz参数也已经废弃，这个参数总是应该指定为NULL。

tv.tv\_usec域的微秒并不意味着我们能够按微秒精度来控制系统时钟，因为系统时钟周期可能大于一微秒。

尽管SUSv3没有规定settimeofday()，它在其它UNIX实现中也广泛可用。

Linux还提供stime()系统调用，用于设置系统时钟。settimeofday()和stime()的区别在于后者只允许按秒的精度来设置日历时间。和time(), gettimeofday()一样，存在stime()和settimeofday()两个调用也是历史原因，后者由4.3BSD增加。

调用settimeofday()剧烈地改变系统时间，可能对其它应用（如make、使用时间戳的数据库系统、时间戳日志文件）产生不良影响，因为这些应用依赖于单调增长的系统时钟。由于这个原因，当需要微量调整时间（如几秒），通常优先使用adjtime()库函数，该函数会将系统时钟逐步地调整到需要的值。

|  |
| --- |
| #define \_BSD\_SOURCE  #include <sys/time.h>  int adjtime(struct timeval \*delta, struct timeval \*olddelta);  成功时返回0；出错时返回-1 |

delta参数指向一个timeval结构体，指定需要调整的秒数和微秒数。如果这个值是正的，则系统时钟的每一秒都将增加小量的值，直到增加够指定的时间数量。如果delta值为负，系统时钟将以同样的方式减缓。

Linux/x86-32的时钟调整数量是1/2000秒（43.2秒每天）。

调用adjtime()时，可能还有上一个未完成的时钟调整正在进行。在这种情况下，剩余的未调整时间数量将在olddelta指向的timeval结构体中返回。如果我们对这个值不感兴趣，可以指定olddelta为NULL。反过来，如果我们只想知道当前未调整完成的时间数量，而不修改时钟，我们可以指定delta参数为NULL。

尽管SUSv3并没有规定adjtime()，它在多数UNIX实现中都可用。

在Linux中，adjtime()使用更加通用（也更复杂）的Linux特定系统调用adjtimex()来实现。后者也被用于网络时间协议（NTP）daemon。更多信息请参考Linux源代码、Linux adjtimex(2)手册页、和NTP规范（[Mills, 1992]）

## 10.6 软时钟（Jiffy）

本书中讨论的各种时间相关的系统调用，其精度受限于系统软时钟的精度，软时钟以jiffies为单位来测量时间。jiffy的大小在内核源代码中由HZ常量定义。这也是内核按roundrobin time-sharing调度算法分配CPU给进程的时间单位（35.1节）。

在Linux/x86-32内核2.4以前（包括2.4），软时钟为100赫兹，也就是1个jiffy就是10毫秒。

由于CPU速度自Linux最早实现以来已经大幅增长，在内核2.6.0之后，Linux/x86-32的软时钟提升为1000赫兹。更高的软时钟带来的优点是定时器能够以更高的精度操作，时间测量的精度也更高。但是随意地提升时钟速率也是不行的，因为每个时钟中断都要消耗少量的CPU时间，而这些时间CPU无法用来执行进程。

内核开发者之间对于时钟率的争论，最终导致软时钟成为内核可配置项（在Processor type and features, Timer frequency下）。从内核2.6.13开始，时钟率可以设置为100、250（默认）、1000赫兹，对应的jiffy分别为10、4、1毫秒。从内核2.6.20开始，又提供另一个频率：300赫兹，能够整除于两种常见视频帧速率：25帧每秒（PAL）和30帧每秒（NTSC）。

## 10.7 进程时间

进程时间是进程自创建起使用的CPU时间总量。为了便于记录，内核把CPU时间划分为以下两个部分：

* 用户CPU时间：花费在用户模式下执行的时间数量。有时候也称为虚拟时间，对于程序来说，这也是它能够访问CPU的时间。
* 系统CPU时间：花费在内核模式下执行的时间数量。这是内核执行系统调用、或代替程序执行其它任务所花费的时间（如页面中断服务）。

有时候，我们称进程时间为进程消耗的总CPU时间。

当我们在shell中运行一个程序时，可以使用time(1)命令来获取进程时间、以及运行程序的实际时间：

$ time ./myprog

real 0m4.84s

user 0m1.030s

sys 0m3.43s

times()系统调用可以获取进程的时间信息，并返回在buf指向的结构体中。

|  |
| --- |
| #include <sys/times.h>  clock\_t times(struct tms \*buf);  成功时返回时钟tick数量；出错时返回(clock\_t)-1 |

tms结构体的组成如下：

struct tms {

clock\_t tms\_utime; /\* 调用方使用的用户CPU时间 \*/

clock\_t tms\_stime; /\* 调用方使用的系统CPU时间 \*/

clock\_t tms\_cutime; /\* 所有子进程（已经等待）的用户CPU时间 \*/

clock\_t tms\_cstime; /\* 所有子进程（已经等待）的系统CPU时间 \*/

};

tms结构体的前两个域返回调用进程目前为止使用的用户和系统CPU时间。后两个域返回调用进程的子进程消耗的CPU时间，所有已经终止并且父进程已经调用过wait()的子进程所消耗的CPU时间。

tms结构体的四个域都是clock\_t类型，这是一个整数类型，用来按时钟tick测量时间。我们可以调用sysconf(\_SC\_CLK\_TCK)来获取每秒的时钟tick数量，然后除一下就能得到秒数（我们在11.2节讨论sysconf()）。

在多数Linux硬件体系架构下，sysconf(\_SC\_CLK\_TCK)返回数值100，对应于内核常量USER\_HZ。但是USER\_HZ可以在少数体系架构下定义为其它值，例如Alpha和IA-64。

成功时times()返回从过去的某个任意时间点开始，已经逝去的（实际）时间的时钟tick数量。SUSv3故意不规定这个任意的时间点，只是声明在调用进程的生命周期内都保持不变即可。因此times()返回值唯一可移植的用途是测量进程执行期内的时间，通过与另一个times()返回值进行计算而得。但是即使这样使用times()的返回值也是不可靠的，因为它可能超出clock\_t的范围而溢出，这时候这个值又重新从0开始（也就是后一个times()的返回值小于前一个times()的返回值）。测量已经逝去时间的可靠方法是使用gettimeofday()（10.1节）。

在Linux中，我们可以指定buf为NULL；在这种情况下，times()只是简单地返回函数结果，但这样是不可移植的。SUSv3没有规定buf为NULL的用法，而且许多UNIX实现要求这个参数不能为NULL。

clock()函数提供更简单的接口获取进程时间。它返回调用进程使用的CPU时间总量（用户加系统CPU时间）。

|  |
| --- |
| #include <time.h>  clock\_t clock(void);  成功时返回调用进程使用的CPU时间总量，按CLOCKS\_PER\_SEC测量  出错时返回(clock\_t)-1 |

clock()的返回值按CLOCKS\_PER\_SEC单位计量，因此我们必须拿返回值除以CLOCKS\_PER\_SEC来得到进程使用的CPU时间的秒数。无论底层的软时钟（10.6节）精度是多少，POSIX.1规定CLOCKS\_PER\_SEC固定为1百万。当然clock()的精度肯定是受限于系统软时钟。

尽管clock()的返回类型clock\_t与times()使用的数据类型相同，这两个接口采用的计量单位却是不同的。这是由于POSIX.1和C编程语言标准对clock\_t定义的冲突导致的，属于历史原因。

尽管CLOCKS\_PER\_SEC固定为1百万，SUSv3却注明这个常量在非XSI依从系统中可以是整型变量，因此我们不能可移植地把它当作编译期常量（例如我们不能在#ifdef预处理语句中使用CLOCKS\_PER\_SEC）。由于它可能被定义为长整型（1000000L），我们要可移植地使用printf()打印它的值，就应该总是把这个值先转换为long，再打印输出（3.6.2节）。

SUSv3注明clock()应该返回“进程使用的处理器时间”，因此可以按不同的方式解释。在某些UNIX实现中，clock()返回值包含所有已经等待的子进程的CPU时间，不过Linux并没有这样做。

**示例程序**

清单10-5的程序演示了本节讨论的函数的使用。displayProcessTimes()函数打印调用方提供的消息，然后使用clock()和times()获取并显示进程时间。主程序首次调用displayProcessTimes()后，会执行一个循环调用getppid()来消耗一些CPU时间，然后再次调用displayProcessTimes()来查看循环消耗的CPU时间。当我们使用这个程序调用getppid()一千万次时，输出如下：

$ ./process\_time 10000000

CLOCKS\_PER\_SEC=1000000 sysconf(\_SC\_CLK\_TCK)=100

At program start:

clock() returns: 0 clocks-per-sec (0.00 secs)

times() yields: user CPU=0.00; system CPU: 0.00

After getppid() loop:

clock() returns: 2960000 clocks-per-sec (2.96 secs)

times() yields: user CPU=1.09; system CPU: 1.87

*清单10-5：获取进程CPU时间*

----------------------------------------------------time/process\_time.c

#include <sys/times.h>

#include <time.h>

#include "tlpi\_hdr.h"

static void /\* Display 'msg' and process times \*/

displayProcessTimes(const char \*msg)

{

struct tms t;

clock\_t clockTime;

static long clockTicks = 0;

if (msg != NULL)

printf("%s", msg);

if (clockTicks == 0) { /\* Fetch clock ticks on first call \*/

clockTicks = sysconf(\_SC\_CLK\_TCK);

if (clockTicks == -1)

errExit("sysconf");

}

clockTime = clock();

if (clockTime == -1)

errExit("clock");

printf(" clock() returns: %ld clocks-per-sec (%.2f secs)\n",

(long) clockTime, (double) clockTime / CLOCKS\_PER\_SEC);

if (times(&t) == -1)

errExit("times");

printf(" times() yields: user CPU=%.2f; system CPU: %.2f\n",

(double) t.tms\_utime / clockTicks,

(double) t.tms\_stime / clockTicks);

}

int

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

{

int numCalls, j;

printf("CLOCKS\_PER\_SEC=%ld sysconf(\_SC\_CLK\_TCK)=%ld\n\n",

(long) CLOCKS\_PER\_SEC, sysconf(\_SC\_CLK\_TCK));

displayProcessTimes("At program start:\n");

numCalls = (argc > 1) ?

getInt(argv[1], GN\_GT\_0, "num-calls") : 100000000;

for (j = 0; j < numCalls; j++)

(void) getppid();

displayProcessTimes("After getppid() loop:\n");

exit(EXIT\_SUCCESS);

}

----------------------------------------------------time/process\_time.c

## 10.8 小结

实际时间对应于日常生活中的时间。当实际时间从某个标准点计量时，我们就称之为日历时间，与经过时间相对，后者则从进程生命周期的某个点（通常是起点）来计量。

进程时间是进程使用的CPU时间总量，划分为用户和系统CPU时间两部分。

许多系统调用允许我们获取和设置系统时钟值（日历时间，从Epoch开始计量的秒数），许多库函数允许在日历时间和其它时间格式之间转换，包括分解时间和人类可读的字符串格式。对这类转换的讨论带我们进一步讨论了locale和国际化。

使用和显示时间与日期是许多应用很重要的一部分，在本书后面章节，我们需要频繁地使用本章讨论的函数。我们在第23章还会再讨论时间测量。

**更多信息**

[Love, 2010]中可以找到Linux内核如何测量时间的详细信息。

时区和国际化的详细讨论可以在GNU C库手册中找到（<http://www.gnu.org/>）。SUSv3文档也详细地描述了locale。

## 10.9 习题

10-1. 假设系统调用sysconf(\_SC\_CLK\_TCK)返回100，假设times()返回的clock\_t是无符号32位整数，需要多久clock\_t会重新从0开始？对clock()返回的CLOCKS\_PER\_SEC值做同样的计算，结果如何？

# 第11章 系统限制和选项

每个UNIX实现都对各种系统特性和资源设置了限制，并且提供（或者选择不提供）各种标准定义的选项。常见例子如下：

* 进程同时能保持多少打开文件？
* 系统是否支持实时信号？
* 类型int的变量能存储的最大值是什么？
* 程序的参数列表最多能有多大？
* 路径名最长多少？

虽然我们可以在应用中硬编码这些限制和选项，但这样做降低了可移植性，因为限制和选项可能由于以下原因而变化：

* 跨UNIX实现：尽管在某个系统中限制和选项可能是固定的，但在不同的UNIX实现间，限制和选项的值却可能完全不同。例如int能够存储的最大值。
* 在某个实现的运行时：例如内核可能重新配置并修改某个限制。此外应用也可能在这个系统上编译，但在拥有不同限制和选项的另一个系统中运行。
* 文件系统不同：例如传统的System V文件系统只允许文件名最长14字节，而传统的BSD文件系统和多数Linux文件系统则允许文件名最长255字节。

由于系统限制和选项会影响应用的行为，可移植应用需要一种机制，来确定限制的值，以及某个选项是否支持。C编程语言标准和SUSv3提供两种主要的方法，允许应用获取这些信息：

* 某些限制和选项可以在编译期确定。例如int的最大值由硬件体系架构和编译器设计决定。这一类限制可以记录在头文件中。
* 其它限制和选项可能在运行时变化。这种情况下，SUSv3定义了三个函数：sysconf(), pathconf(), fpathconf()。应用可以调用这三个函数来检查系统限制和选项。

SUSv3规定了一系列限制，要求依从系统实现。SUSv3同时还规定了一组选项，特定系统可能提供，也可能不提供这些选项。我们在本章讨论部分限制和选项，并且在后续章节适当的位置讨论相关的限制和选项。

## 11.1 系统限制

SUSv3对于每个标准规定的限制，都要求所有实现对该限制支持一个最小值。多数情况下，这个最小值作为常量定义在<limits.h>，名字前缀为\_POSIX\_，通常还包含字符串\_MAX；因此名字的组成格式为\_POSIX\_XXX\_MAX。

如果应用对每个限制都严格使用SUSv3规定的最小值，那么应用就能移植到标准的所有实现系统中。但是这样做阻止应用利用系统提供的更高限制值。因此通常在运行时，使用系统的<limits.h>、sysconf()、pathconf()确定限制值，是更佳选择。

每个限制都有一个名字，对应于最小值名，但是缺少了\_POSIX\_前缀。系统可以在<limits.h>中定义这个常量来指示当前系统该限制的值。如果定义了限制常量，这个值至少总是拥有上面讨论过的最小值（XXX\_MAX >= \_POSIX\_XXX\_MAX）。

SUSv3把限制划分为三种类型：运行时不变值、路径名可变值、运行时可增大值。在以下段落中，我们讨论这些不同的类型，并提供一些示例。

**运行时不变值（可能不确定）**

运行时不变值是那些定义在<limits.h>中，并且对于实现是固定的限制值。但是这个值可能是不确定的（例如依赖于可用的内存空间），这时候<limits.h>将会漏掉该限制的值定义。在这种情况下（包括<limits.h>已经定义限制值的情况），应用可以使用sysconf()在运行时确定该限制的值。

MQ\_PRIO\_MAX限制就是一个运行时不变值。52.5.1节将说明，这是POSIX消息队列的消息优先级限制。SUSv3定义的\_POSIX\_MQ\_PRIO\_MAX常量值为32，表示所有依从标准的实现都必须为该限制至少提供这个值。这意味着我们可以保证所有依从实现都允许消息优先级范围0到31。某个UNIX实现可以设置一个更高的限制值，并在<limits.h>中定义常量MQ\_PRIO\_MAX为这个值。例如在Linux中，MQ\_PRIO\_MAX定义为32768，这个值也可以在运行时使用以下调用确定：

lim = sysconf(\_SC\_MQ\_PRIO\_MAX);

**路径名可变值**

路径名可变值是那些与路径名相关的限制（文件、目录、终端等等）。每个限制在具体的实现系统中可能是常量，但在不同的文件系统中可能变化。当路径名限制可能变化时，应用可以使用pathconf()和fpathconf()来确定限制的值。

NAME\_MAX就是一个路径名可变值的限制。这个限制定义了特定文件系统的文件名的最大长度。SUSv3定义常量\_POSIX\_NAME\_MAX的值为14（受到老的System V文件系统限制），这就是依从实现必须提供的文件名最小长度。实现可以定义NAME\_MAX为更高的限制值，此时特定文件系统的这个限制值可以通过以下调用获得：

lim = pathconf(directory\_path, \_PC\_NAME\_MAX)

directory\_path是文件系统中某个目录的路径。

**运行时可增大值**

运行时可增大值的限制对于特定实现拥有固定最小值，所有依从系统都对该限制提供至少这个最小值，但是特定的系统可能在运行时增大这个限制的值，应用可以使用sysconf()获取系统对该限制支持的实际大小。

NGROUPS\_MAX就是一个运行时可增大值的限制，这个限制规定了一个进程能够同时拥有的附加组ID的最大数量（9.6节）。SUSv3定义了相应的最小值，\_POSIX\_NGROUPS\_MAX为8。在运行时应用可以使用sysconf(\_SC\_NGROUPS\_MAX)来获取这个限制的实际值。

**部分SUSv3限制汇总**

表11-1列出了SUSv3定义的与本书相关的部分限制，其它限制在后续章节也会相应地介绍。

*表11-1：部分SUSv3限制*

|  |  |  |  |
| --- | --- | --- | --- |
| **限制名称**  **（<limits.h>）** | **最小值** | **sysconf()/pathconf()名字**  **（<unistd.h>）** | **描述** |
| ARG\_MAX | 4096 | \_SC\_ARG\_MAX | 可以提供给exec()的参数（argv）和环境（environ）的最大字节数（6.7和27.2.3节） |
| 无 | 无 | \_SC\_CLK\_TCK | times()的计量单位 |
| LOGIN\_NAME\_MAX | 9 | \_SC\_LOGIN\_NAME\_MAX | 登录名（包括null终止符）的最大长度 |
| OPEN\_MAX | 20 | \_SC\_OPEN\_MAX | 进程能够同时打开的文件描述符最大数量，比最大可用描述符大1（36.2节） |
| NGROUPS\_MAX | 8 | \_SC\_NGROUPS\_MAX | 进程能够同时属于的附加组的最大数量（9.7.3节） |
| 无 | 1 | \_SC\_PAGESIZE | 虚拟内存页的大小（与\_SC\_PAGE\_SIZE相同） |
| RTSIG\_MAX | 8 | \_SC\_RTSIG\_MAX | 不同实时信号的最大数量（22.8节） |
| SIGQUEUE\_MAX | 32 | \_SC\_SIGQUEUE\_MAX | 实时信号排队的最大数量（22.8节） |
| STREAM\_MAX | 8 | \_SC\_STREAM\_MAX | 同时能够打开的stdio流的最大数量 |
| NAME\_MAX | 14 | \_PC\_NAME\_MAX | 文件名的最大字节数，不包括null终止字节 |
| PATH\_MAX | 256 | \_PC\_PATH\_MAX | 路径名的最大字节数，包括null终止符 |
| PIPE\_BUF | 512 | \_PC\_PIPE\_BUF | 能够向管道和FIFO原子写入的最大字节数（44.1节） |

表11-1的第一列给出了限制名，可能在<limits.h>中定义为常量，表示某个特定实现的限制值。第二列是SUSv3定义的限制最小值（也定义在<limits.h>中）。多数情况下，每个最小值都以前缀\_POSIX\_定义为常量。例如\_POSIX\_RTSIG\_MAX（值为8）表示SUSv3对相应的RTSIG\_MAX限制要求的最小值。第3列是运行时调用sysconf()或pathconf()时使用的常量名，用来获取相应限制的实现值。\_SC\_前缀的常量用于sysconf()；\_PC\_前缀的常量则用于pathconf()和fpathconf()。

注意以下对表11-1的补充信息：

* getdtablesize()函数用来确定进程文件描述符限制（OPEN\_MAX），但是已经被废弃。SUSv2标记该函数为遗留，SUSv3则移除了该函数的定义。
* getpagesize()函数用来确定系统页大小（\_SC\_PAGESIZE），但是已经被废弃。SUSv2标记该函数为遗留，SUSv3则移除了该函数的定义。
* <stdio.h>中定义的常量FOPEN\_MAX，等同于STREAM\_MAX。
* NAME\_MAX不包含null终止符，而PATH\_MAX则包含null终止符。早期POSIX.1标准没有规定PATH\_MAX是否包含null终止符，虽然新标准对NAME\_MAX和PATH\_MAX的规定确实不统一，但是规定PATH\_MAX包含终止符能够最大限度地保持兼容性，这样那些为路径只分配PATH\_MAX个字节的应用也能继续遵循标准的规定。

**在shell中确定限制和选项：getconf**

在shell中，我们可以使用getconf命令来获取限制和选项的值。这个命令通用的格式如下：

$ getconf variable-name [ pathname ]

variable-name标识限制，必须是SUSv3标准规定的限制名，例如ARG\_MAX或NAME\_MAX。当限制与路径名相关时，我们必须指定一个路径名作为命令的第二个参数，如下所示：

$ getconf ARG\_MAX

131072

$ getconf NAME\_MAX /boot

255

## 11.2 运行时获取系统限制（和选项）

sysconf()函数允许应用在运行时获取系统限制的值。

|  |
| --- |
| #include <unistd.h>  long sysconf(int name);  成功时返回name指定限制的值  限制不确定或出错时返回-1 |

name参数是某个定义在<unistd.h>中的\_SC\_\*常量，表11-1列出了常用的几个。sysconf()函数返回限制相应的值。

如果限制无法确定，sysconf()返回-1，出错时函数也返回-1（唯一的错误是EINVAL，表示name非法）。要区分限制无法确定还是函数出错，我们必须在调用函数前设置errno为0；如果函数返回-1，并且设置了errno，就表示函数发生了错误。

sysconf(), pathconf(), fpathconf()的返回值总是long。在SUSv3 sysconf()的rationale主题中，注明曾考虑返回字符串作为可能的返回值，但由于实现和使用的复杂性而最终被拒绝。

清单11-1演示了sysconf()的使用，这个程序显示各种系统限制。在Linux 2.6.31/x86-32系统中运行该程序得到如下输出：

$ ./t\_sysconf

\_SC\_ARG\_MAX: 2097152

\_SC\_LOGIN\_NAME\_MAX: 256

\_SC\_OPEN\_MAX: 1024

\_SC\_NGROUPS\_MAX: 65536

\_SC\_PAGESIZE: 4096

\_SC\_RTSIG\_MAX: 32

*清单11-1：使用sysconf()*

-----------------------------------------------------syslim/t\_sysconf.c

#include "tlpi\_hdr.h"

static void /\* Print 'msg' plus sysconf() value for 'name' \*/

sysconfPrint(const char \*msg, int name)

{

long lim;

errno = 0;

lim = sysconf(name);

if (lim != -1) { /\* Call succeeded, limit determinate \*/

printf("%s %ld\n", msg, lim);

} else {

if (errno == 0) /\* Call succeeded, limit indeterminate \*/

printf("%s (indeterminate)\n", msg);

else /\* Call failed \*/

errExit("sysconf %s", msg);

}

}

int

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

{

sysconfPrint("\_SC\_ARG\_MAX: ", \_SC\_ARG\_MAX);

sysconfPrint("\_SC\_LOGIN\_NAME\_MAX: ", \_SC\_LOGIN\_NAME\_MAX);

sysconfPrint("\_SC\_OPEN\_MAX: ", \_SC\_OPEN\_MAX);

sysconfPrint("\_SC\_NGROUPS\_MAX: ", \_SC\_NGROUPS\_MAX);

sysconfPrint("\_SC\_PAGESIZE: ", \_SC\_PAGESIZE);

sysconfPrint("\_SC\_RTSIG\_MAX: ", \_SC\_RTSIG\_MAX);

exit(EXIT\_SUCCESS);

}

-----------------------------------------------------syslim/t\_sysconf.c

SUSv3要求sysconf()对某个限制返回的值在调用进程的生命周期内都保持常量。例如我们可以认为\_SC\_PAGESIZE返回的值在进程运行过程中不会改变。

在Linux中，上面关于进程生命周期内限制值不变的说法存在一些（合理的）例外。进程可以使用setrlimit()（36.2节）来修改许多进程资源限制，进而影响到sysconf()报告的限制值：RLIMIT\_NOFILE确定进程可以打开的文件数量（\_SC\_OPEN\_MAX）；RLIMIT\_NPROC是进程能够创建的进程数量（\_SC\_CHILD\_MAX，SUSv3并没有规定这个限制）；RLIMIT\_STACK是Linux 2.6.23开始，确定进程命令行参数和环境允许的空间大小（\_SC\_ARG\_MAX；细节请参考execve手册页）。

## 11.3 运行时获取文件相关的限制（和选项）

pathconf()和fpathconf()函数允许应用在运行时获取文件相关的限制值。

|  |
| --- |
| #include <unistd.h>  long pathconf(const char \*pathname, int name);  long fpathconf(int fd, int name);  成功时都返回name指定限制的值  限制无法确定或出错时都返回-1 |

pathconf()和fpathconf()的唯一区别在于参数指定文件或目录。pathconf()的参数是路径名；而fpathconf()的参数则是文件描述符（之前打开）。

name参数是<unistd.h>中定义的某个\_PC\_\*常量，表11-1中列出了其中常用的几个。表11-2则提供了表11-1中\_PC\_\*常量的更多细节。

这两个函数都返回相应限制的值。我们可以按sysconf()一样的方式来区分限制无法确定和出错。

和sysconf()不一样的是，SUSv3不要求pathconf()和fpathconf()返回的值在进程生命周期内保持不变。因为文件系统在进程运行过程中可能被卸载并以不同的属性重新挂载。

*表11-2：部分pathconf() \_PC\_\*常量的细节*

|  |  |
| --- | --- |
| **常量** | **说明** |
| \_PC\_NAME\_MAX | 针对目录，将返回目录中文件的限制值。传递其它文件类型的行为是未定义的。 |
| \_PC\_PATH\_MAX | 针对目录，将返回该目录相对路径名的最大长度。传递其它文件类型的行为是未定义的。 |
| \_PC\_PIPE\_BUF | 针对FIFO或管道，将返回引用文件的限制值。  针对目录，将返回该目录下创建的FIFO的限制值。传递其它文件类型的行为是未定义的。 |

清单11-2显示了fpathconf()的使用，获取标准输入指向的文件的各种限制值。当我们运行这个程序时，指定标准输入为ext2文件系统的某个目录，将得到以下结果：

$ ./t\_fpathconf < .

\_PC\_NAME\_MAX: 255

\_PC\_PATH\_MAX: 4096

\_PC\_PIPE\_BUF: 4096

*清单11-2：使用fpathconf()*

---------------------------------------------------syslim/t\_fpathconf.c

#include "tlpi\_hdr.h"

static void /\* Print 'msg' plus value of fpathconf(fd, name) \*/

fpathconfPrint(const char \*msg, int fd, int name)

{

long lim;

errno = 0;

lim = fpathconf(fd, name);

if (lim != -1) { /\* Call succeeded, limit determinate \*/

printf("%s %ld\n", msg, lim);

} else {

if (errno == 0) /\* Call succeeded, limit indeterminate \*/

printf("%s (indeterminate)\n", msg);

else /\* Call failed \*/

errExit("fpathconf %s", msg);

}

}

int

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

{

fpathconfPrint("\_PC\_NAME\_MAX: ", STDIN\_FILENO, \_PC\_NAME\_MAX);

fpathconfPrint("\_PC\_PATH\_MAX: ", STDIN\_FILENO, \_PC\_PATH\_MAX);

fpathconfPrint("\_PC\_PIPE\_BUF: ", STDIN\_FILENO, \_PC\_PIPE\_BUF);

exit(EXIT\_SUCCESS);

}

---------------------------------------------------syslim/t\_fpathconf.c

## 11.4 不确定限制

偶尔我们会遇到某些系统限制没有定义为系统限制常量（如PATH\_MAX），此时sysconf()或pathconf()会告知我们该限制（如\_PC\_PATH\_MAX）是不确定的。在这种情况下，我们可以采用以下策略之一：

* 编写需要在多个UNIX实现间可移植的应用时，我们可以采用SUSv3规定的最小限制值。也就是那些格式为 \_POSIX\_\*\_MAX的常量，11.1节已经讨论过了。有时候这个方法并不可行，因为标准规定的最小值不切实际的低，例如\_POSIX\_PATH\_MAX和\_POSIX\_OPEN\_MAX。
* 某些情况下，忽略限制检查是实践可行的解决办法，此时我们可以执行相关的系统或库函数调用。（这个办法也可以应用到11.5节讨论的某些SUSv3选项中）。如果系统或函数调用失败，并且errno指示某个系统限制超出范围，我们就可以修改相应的应用行为（减少限制值）并重试。例如多数UNIX实现都对每个进程实时信号排队的数量设置了限制，一旦达到这个限制，试图发送信号（使用sigqueue()）将会以EAGAIN错误失败。在这种情况下，发送进程只需要简单地重试，也可以适当地延迟一定时间再发送。类似地，打开名字太长的文件会得到ENAMETOOLONG错误，应用可以缩短文件名再重试。
* 我们在编写自己的程序或函数时，可以推断或估计限制值。调用sysconf()或pathconf()之后，如果发现限制值不确定，我们就返回一个“合理猜测”的值。这个办法虽然不完美，但在实践中通常是可行的。
* 我们可以采用类似GNU Autoconf的工具，这是一个可扩展的工具，能够确定各种系统特性和限制是否存在以及相应的值。Autoconf程序根据自己确定的信息来生成头文件，然后C程序可以包含并使用这些头文件。Autoconf的更多信息可以在<http://www.gnu.org/software/autoconf/>找到。

## 11.5 系统选项

除了规定各种系统资源的限制，SUSv3还规定了UNIX实现需要支持的各种选项。包括实时信号、POSIX共享内存、任务控制、POSIX线程等。通常SUSv3不强制要求UNIX实现支持这些选项（少数例外）。SUSv3允许实现在编译期或运行时报告是否支持某个特性。

UNIX实现可以在<unistd.h>中定义相应的常量，在编译期声明自己支持某个特定的SUSv3选项。这样的常量都有特定前缀，表明该选项起源于哪个标准（例如 \_POSIX\_ 或 \_XOPEN\_）。

如果定义了某个选项常量，它可以拥有以下值：

* 值为-1表示*不支持该选项*。在这种情况下，该选项相关的头文件、数据类型、函数接口都不需要定义。我们需要使用#if预处理器指令来处理这种选项的可移植性。
* 值为0表示*可能支持该选项*。应用必须在运行时检查，以确定是否支持该选项。
* 值大于0表示*支持该选项*。该选项相关的所有头文件、数据类型、函数接口都已经定义，并且符合标准规定的行为。在多数情况下，SUSv3要求这个值定义为200112L，对应于SUSv3被通过为正式标准的年月（SUSv4的值则为200809L）。

当常量定义为0时，应用可以使用sysconf(), pathconf(), fpathconf()函数在运行时检查是否支持该选项。传递给这些函数的name参数通常和相应的编译期常量相同，只不过前缀替换为 \_SC\_ 或 \_PC\_。实现至少必须提供必要的头文件、常量、函数接口，来执行运行时检查。

SUSv3对于未定义选项常量的含义不太清晰，没有明确说明未定义常量与常量定义为0（可能支持该选项），还是常量定义为-1（不支持该选项）相同。标准委员会随后解决了这个问题，SUSv4标准明确规定未定义选项常量，等同于选项常量定义为-1（不支持）。

表11-3列出了SUSv3规定的部分选项。第一列是选项的编译期常量名（定义在<unistd.h>中），以及相应的sysconf()（\_SC\_\*）或pathconf()（\_PC\_\*）名字参数。注意以下特定选项的补充信息：

* SUSv3要求某些选项，编译期常量的值总是大于0。历史上这些选项是可选的，但是今天已经强制要求实现。表11-3中以 + 号标注这样的选项（另外SUSv4把SUSv3中的很多可选选项都变成强制要求）。
* 对于某些选项，编译期常量的值不能是-1。换句话说，实现必须支持该选项，或者必须在运行时检查选项是否支持。这些选项在表中以字符 \* 进行标注。

*表11-3：部分SUSv3选项*

|  |  |  |
| --- | --- | --- |
| **选项（常量）名**  **sysconf()/pathconf()名** | **描述** | **标注** |
| \_POSIX\_ASYNCHRONOUS\_IO  (\_SC\_ASYNCHRONOUS\_IO) | 异步I/O |  |
| \_POSIX\_CHOWN\_RESTRICTED  (\_PC\_CHOWN\_RESTRICTED) | 只有特权进程才能使用chown()和fchown()来改变文件的用户ID和组ID为任意值（15.3.2节） | \* |
| \_POSIX\_JOB\_CONTROL  (\_SC\_JOB\_CONTROL) | 任务控制（34.7节） | + |
| \_POSIX\_MESSAGE\_PASSING  (\_SC\_MESSAGE\_PASSING) | POSIX消息队列（52章） |  |
| \_POSIX\_PRIORITY\_SCHEDULING  (\_SC\_PRIORITY\_SCHEDULING) | 进程调度（35.3节） |  |
| \_POSIX\_REALTIME\_SIGNALS  (\_SC\_REALTIME\_SIGNALS) | 实时信号扩展（22.8节） |  |
| \_POSIX\_SAVED\_IDS  (none) | 进程拥有设置用户ID和保存的设置组ID（9.4节） | + |
| \_POSIX\_SEMAPHORES  (\_SC\_SEMAPHORES) | POSIX信号量（53章） |  |
| \_POSIX\_SHARED\_MEMORY\_OBJECTS  (\_SC\_SHARED\_MEMORY\_OBJECTS) | POSIX共享内存对象（54章） |  |
| \_POSIX\_THREADS  (\_SC\_THREADS) | POSIX线程 |  |
| \_XOPEN\_UNIX  (\_SC\_XOPEN\_UNIX) | 支持XSI扩展（1.3.4节） |  |

## 11.6 小结

SUSv3规定了实现必须实施的限制，以及实现可能支持的系统选项。

通常我们不能将系统限制和选项硬编码到程序中，因为不同实现的系统限制和选项可能不同，甚至相同系统也可能不同（例如运行时改变或跨文件系统）。因此SUSv3规定了实现报告自己对限制和选项是否支持的方法。对于多数限制，SUSv3规定了最小值，所有实现都必须支持。此外每个实现可以在编译期（通过<limits.h>或<unistd.h>中的常量）来声明特定限制和选项的值。同时通过调用sysconf(), pathconf(), fpathconf()函数，应用可以在运行时获得限制和选项的值。某些情况下，使用上述办法可能无法确定特定限制的值，对于这种不确定限制，我们必须借助于特别的技术来确定限制的值（11.4节）。

**更多信息（懒得翻译了）**

Chapter 2 of [Stevens & Rago, 2005] and Chapter 2 of [Gallmeister, 1995] cover similar ground to this chapter. [Lewine, 1991] also provides much useful (although now slightly outdated) background. Some information about POSIX options with notes on glibc and Linux details can be found at http://people.redhat.com/drepper/posix-option-groups.html. The following Linux manual pages are also relevant: sysconf(3), pathconf(3), feature\_test\_macros(7), posixoptions(7), and standards(7).

The best sources of information (though sometimes difficult reading) are the relevant parts of SUSv3, particularly Chapter 2 from the Base Definitions (XBD), and the specifications for <unistd.h>, <limits.h>, sysconf(), and fpathconf(). [Josey, 2004] provides guidance on the use of SUSv3.

## 11.7 习题

11-1. 在其它UNIX实现中运行清单11-1中的程序。

11-2. 在其它文件系统中运行清单11-2中的程序。

# 第12章 系统和进程信息

本章我们讨论访问各种系统和进程信息的方法，主要的焦点是讨论/proc文件系统。同时我们还描述uname()系统调用，用来获取各种系统标识。

## 12.1 /proc文件系统

在老的UNIX实现中，通常不能很容易地分析（或修改）内核属性，很难对类似下面这样的问题作出解答：

* 系统中有多少进程正在运行？谁拥有这些进程？
* 进程打开了哪些文件？
* 哪些文件被锁定，哪些进程拥有锁？
* 系统正在使用什么样的sockets？

有些老的UNIX实现通过允许特权程序深入内核内存中的数据结构，来解决这个问题。但是这个办法带来了各种问题，特别是它要求程序员理解内核数据结构的专业知识，而这些结构可能在不同内核版本中有变化，从而要求依赖于这些数据结构的程序重新编写。

为了提供访问内核信息更加容易的方法，许多现代UNIX实现提供了/proc虚拟文件系统。这个文件系统在/proc目录下，包含许多文件，暴露了各种内核信息，并允许进程方便地使用普通文件I/O系统调用来读取这些信息，某些情况下也可以修改某些信息。/proc文件系统被称为虚拟，是因为子目录和文件并没有存放在磁盘中。相反内核在进程访问它们的时候实时创建这些文件和子目录。

在这一节，我们概述/proc文件系统。在本书后面，我们会在各章主题相关的时候讨论特定的/proc文件。尽管许多UNIX实现都提供/proc文件系统，SUSv3却并没有对其做出规定，本书讨论的细节都特定于Linux。

### 12.1.1 获取进程的信息：/proc/PID

对于系统中的每一个进程，内核都提供一个相应的目录，命名为/proc/PID，其中PID是进程的ID。在这个目录里有许多子目录和文件，包含了进程的所有相关信息。例如，我们可以获取init进程的相关信息，init进程的ID总是1，因此我们需要读取/proc/1目录下的文件。

/proc/PID目录下的许多文件中有一个命名为status，该文件提供了进程的许多重要信息：

$ cat /proc/1/status

Name: init 进程运行的命令名

State: S (sleeping) 进程状态

Tgid: 1 线程组ID（传统的PID，getpid()）

Pid: 1 实际上是线程ID（gettid()）

PPid: 0 父进程ID

TracerPid: 0 跟踪进程的PID（没有跟踪则为0）

Uid: 0 0 0 0 实际、有效、保存设置、文件系统用户ID

Gid: 0 0 0 0 实际、有效、保存设置、文件系统组ID

FDSize: 256 当前已分配的 #of文件描述符slots

Groups: 附加组ID

VmPeak: 852 kB 虚拟内存大小峰值

VmSize: 724 kB 虚拟内存当前大小

VmLck: 0 kB 锁定内存

VmHWM: 288 kB resident set大小峰值

VmRSS: 288 kB resident set当前大小

VmData: 148 kB 数据段大小

VmStk: 88 kB 堆栈大小

VmExe: 484 kB 文本段（可执行代码）大小

VmLib: 0 kB 共享库代码大小

VmPTE: 12 kB 页表大小（2.6.10起）

Threads: 1 本线程的线程组#of threads

SigQ: 0/3067 当前/最大排队的信号（since 2.6.12起）

SigPnd: 0000000000000000 线程挂起的信号

ShdPnd: 0000000000000000 进程挂起的信号（since 2.6起）

SigBlk: 0000000000000000 阻塞的信号

SigIgn: fffffffe5770d8fc 忽略的信号

SigCgt: 00000000280b2603 捕获的信号

CapInh: 0000000000000000 可中断capabilities

CapPrm: 00000000ffffffff 权限capabilities

CapEff: 00000000fffffeff 有效能力capabilities

CapBnd: 00000000ffffffff Capability边界集 （2.6.26起）

Cpus\_allowed: 1 允许的CPU，掩码（2.6.24起）

Cpus\_allowed\_list: 0 同上，列表格式（2.6.26起）

Mems\_allowed: 1 允许的内存结点，掩码（2.6.24起）

Mems\_allowed\_list: 0 同上，列表格式（2.6.26起）

voluntary\_ctxt\_switches: 6998 自愿上下文切换（2.6.23起）

nonvoluntary\_ctxt\_switches: 107 非自愿上下文件切换（2.6.23起）

Stack usage: 8 kB 高位堆栈用量（2.6.32起）

上面是从内核2.6.32获取的输出。正如括号中注释的从内核版本起那样，这个文件的格式是随着时间而变化的，后续内核版本会增加新的域（少数情况下还会移除域）。除了上面标注的Linux 2.6变化，Linux 2.4增加了Tgid, TracerPid, FDSize, Threads域。

/proc下的文件随着时间而变化，意味着我们使用/proc的文件时，必须采用以下基本策略：当文件内容包含多个记录时，我们需要防御性地进行解析，也就是通过匹配特定字符串（例如PPid）来找到相应的行，而不是直接通过行号来匹配特定的记录。

表12-1列出了/proc/PID目录下比较重要的一些文件。

*表12-1：/proc/PID目录下的一些文件*

|  |  |
| --- | --- |
| **文件** | **描述（进程属性）** |
| cmdline | 命令行参数，\0分隔 |
| cwd | 当前工作目录的符号链接 |
| environ | 环境列表，格式为NAME=value，\0分隔 |
| exe | 被执行文件的符号链接 |
| fd | 目录，包含进程所有打开文件的符号链接 |
| maps | 内存映射 |
| mem | 进程虚拟内存（执行I/O前必须lseek()到合法的偏移位置） |
| mounts | 进程的挂载点 |
| root | root目录的符号链接 |
| status | 各种信息（如进程ID、凭证、内存使用、信号等） |
| task | 进程的每个线程都有一个子目录（Linux 2.6） |

**/proc/PID/fd目录**

/proc/PID/fd目录下对进程打开的每个文件描述符都有一个符号链接。每个符号链接的名字都是对应的描述符数字；例如/proc/1968/1是进程1968的标准输出的符号链接。更多信息请参考5.11节。

进程访问自己的/proc/PID目录有一个便捷的方法，那就是使用符号链接/proc/self。

**线程：/proc/PID/task目录**

Linux 2.4增加了线程组的概念，以支持POSIX线程模型。由于线程组中线程的某些属性是有区别的，Linux 2.4在/proc/PID目录下增加了task子目录。对于进程的每个线程，内核都提供一个/proc/PID/task/TID子目录，其中TID就是线程ID。（等同于线程中调用gettid()返回的值）。

在每个/proc/PID/task/TID子目录下，有一组和/proc/PID下完全相同的文件和目录。由于线程共享许多属性，对于进程中的每个线程，这些文件中的许多信息都是相同的。但是在适当的地方，这些文件也显示了各个线程的区别。例如在线程组的/proc/task/TID/status文件中，每个线程的State, Pid, SigPnd, SigBlk, CapInh, CapPrm, CapEff, CapBnd域就是不一样的。

### 12.1.2 /proc下的系统信息

/proc下的许多文件和子目录提供了系统级信息的访问。图12-1显示了其中的一部分。

图12-1中显示的许多文件在本书其它地方都有描述。表12-2汇总了/proc子目录的一般用途。



*图12-1：/proc下的部分文件和子目录*

*表12-2：部分/proc子目录的用途*

|  |  |
| --- | --- |
| **目录** | **目录中文件暴露的信息** |
| /proc | 各种系统信息 |
| /proc/net | 网络和sockets的状态信息 |
| /proc/sys/fs | 文件系统相关设置 |
| /proc/sys/kernel | 各种通常的内核设置 |
| /proc/sys/net | 网络和sockets设置 |
| /proc/sys/vm | 内存管理设置 |
| /proc/sysvipc | System V IPC对象相关的信息 |

### 12.1.3 访问/proc文件

我们通常使用shell脚本来访问/proc下的文件（多数/proc文件包含多行信息，使用Python或Perl等脚本语言可以非常容易地解析）。例如我们可以使用以下shell命令来修改和查看/proc文件的内容：

# echo 100000 > /proc/sys/kernel/pid\_max

# cat /proc/sys/kernel/pid\_max

100000

当然，使用普通文件I/O系统调用的程序也可以访问/proc文件。访问这些文件时会有如下限制：

* 某些/proc文件是只读的；它们的存在只为显示内核的某些信息，不能用来修改这些信息。/proc/PID目录下的多数文件都是只读的。
* 某些/proc文件只能被文件拥有者读取（或者特权进程）。例如/proc/PID下的所有文件都被相应进程的所有者拥有，其中的某些文件（如/proc/PID/environ），只能被文件拥有者读取。
* 除了/proc/PID子目录，/proc下的多数文件都属于root，只有root才能修改这些文件。

**访问/proc/PID下的文件**

/proc/PID目录是易变的。当进程被创建时，就会生成相应ID的/proc/PID目录；进程终止时，这个目录也会相应地消失。这意味着如果我们要访问某个特定的/proc/PID目录，我们在打开目录下的文件的时候，必须处理进程终止，相应地/proc/PID目录被删除的情况。

**示例程序**

清单12-1演示了如何读取和修改/proc文件。这个程序读取并显示/proc/sys/kernel/pid\_max的内容。如果提供了命令行参数，程序就使用这个值来更新该文件。这个文件（Linux 2.6新增）指定了进程ID的上限（6.2节）。下面是使用程序的示例：

$ su Privilege is required to update pid\_max file

Password:

# ./procfs\_pidmax 10000

Old value: 32768

/proc/sys/kernel/pid\_max now contains 10000

*清单12-1：访问/proc/sys/kernel/pid\_max*

------------------------------------------------sysinfo/procfs\_pidmax.c

#include <fcntl.h>

#include "tlpi\_hdr.h"

#define MAX\_LINE 100

int

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

{

int fd;

char line[MAX\_LINE];

ssize\_t n;

fd = open("/proc/sys/kernel/pid\_max", (argc > 1) ? O\_RDWR : O\_RDONLY);

if (fd == -1)

errExit("open");

n = read(fd, line, MAX\_LINE);

if (n == -1)

errExit("read");

if (argc > 1)

printf("Old value: ");

printf("%.\*s", (int) n, line);

if (argc > 1) {

if (write(fd, argv[1], strlen(argv[1])) != strlen(argv[1]))

fatal("write() failed");

system("echo /proc/sys/kernel/pid\_max now contains "

"`cat /proc/sys/kernel/pid\_max`");

}

exit(EXIT\_SUCCESS);

}

------------------------------------------------sysinfo/procfs\_pidmax.c

## 12.2 系统标识：uname()

uname()系统调用可以获取主机系统的各种标识信息，并返回结果保存在utsbuf指向的结构体中。

|  |
| --- |
| #include <sys/utsname.h>  int uname(struct utsname \*utsbuf);  成功时返回0；出错时返回-1 |

utsbuf参数指向utsname结构体，定义如下：

#define \_UTSNAME\_LENGTH 65

struct utsname {

char sysname[\_UTSNAME\_LENGTH]; /\* 系统名 \*/

char nodename[\_UTSNAME\_LENGTH]; /\* 网络结点名 \*/

char release[\_UTSNAME\_LENGTH]; /\* 系统发布级别 \*/

char version[\_UTSNAME\_LENGTH]; /\* 发布版本级别 \*/

char machine[\_UTSNAME\_LENGTH]; /\* 系统运行的硬件环境 \*/

#ifdef \_GNU\_SOURCE /\* 下面Linux特定 \*/

char domainname[\_UTSNAME\_LENGTH]; /\* 主机的NIS域名 \*/

#endif

};

SUSv3规定了uname()，但是对utsname结构体各个域的长度未定义，只是要求字符串以null字符终止。在Linux中，这些域的长度均为65字节，包括了null终止字节。在某些UNIX实现中，这些域的长度更短；在另外一些（如Solaris）UNIX中，域的长度则为257字节。

utsname结构体的sysname, release, version, machine域由内核自动设置。

在Linux中，/proc/sys/kernel目录下有三个只读文件ostype, osrelease, version，分别包含了sysname, release, version三个域相同的信息。另一个文件/proc/version则包含了所有这三个文件的信息，同时还包括内核编译级的信息（例如执行编译的用户名、执行编译的主机名、编译使用的gcc版本）

nodename域返回网络结点名，也就是sethostname()系统调用设置的值（细节请参考相应的手册页）。通常这个名字就是系统的DNS域名的主机前缀。

domainname域返回setdomainname()系统调用设置的值（细节请参考相应的手册页）。这是主机的网络信息服务（NIS）域名（和主机的DNS域名不是一回事）。

gethostname()系统调用是sethostname()的逆向调用，能够获取系统主机名。系统主机名还可以通过hostname(1)命令和Linux特定的/proc/hostname文件来查看和设置。

getdomainname()系统调用与setdomainname()相反，用来获取NIS域名。NIS域名也可以使用domainname(1)命令和Linux特定的/proc/domainname文件来查看和设置。

应用程序很少使用sethostname()和setdomainname()系统调用。通常主机名和NIS域名由启动脚本在系统启动时建立。

清单12-2中的程序显示uname()返回的信息。下面是运行这个程序时的示例输出：

$ ./t\_uname

Node name: tekapo

System name: Linux

Release: 2.6.30-default

Version: #3 SMP Fri Jul 17 10:25:00 CEST 2009

Machine: i686

Domain name:

*清单12-2：使用uname()*

------------------------------------------------------sysinfo/t\_uname.c

#define \_GNU\_SOURCE

#include <sys/utsname.h>

#include "tlpi\_hdr.h"

int

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

{

struct utsname uts;

if (uname(&uts) == -1)

errExit("uname");

printf("Node name: %s\n", uts.nodename);

printf("System name: %s\n", uts.sysname);

printf("Release: %s\n", uts.release);

printf("Version: %s\n", uts.version);

printf("Machine: %s\n", uts.machine);

#ifdef \_GNU\_SOURCE

printf("Domain name: %s\n", uts.domainname);

#endif

exit(EXIT\_SUCCESS);

}

------------------------------------------------------sysinfo/t\_uname.c

## 12.3 小结

/proc文件系统向应用程序暴露了各种内核信息。每个/proc/PID子目录都拥有一组文件和子目录，提供进程ID为PID的各种信息。/proc下面的另外一些文件和目录则暴露了系统级的信息，程序可以读取这些信息，某些情况下还可以修改其中的某些信息。

uname()系统调用允许我们获取UNIX实现和机器的类型等信息。

**更多信息**

关于/proc文件系统的更多信息，可以参考proc(5)手册页、内核源文件Documentation/filesystems/proc.txt、以及Documentation/sysctl目录下的各种文件。

## 12.4 习题

12-1. 编写一个程序，命令行参数指定一个用户名，列出该用户运行的所有进程ID和进程的命令名。（你可以使用清单8-1中的userIdFromName()函数）。我们可以检查系统中所有/proc/PID/status文件的Name:和Uid:行。遍历系统中所有的/proc/PID目录，需要使用readdir，我们在18.8节讨论。确保你的程序正确地处理了以下特殊情况：一开始/proc/PID目录存在，当打开/proc/PID/status文件时该目录却已经消失（进程终止）。

12-2. 编写一个程序绘制一棵树，显示系统中所有进程的父子关系层次架构，最终能够回到init进程。对于每个进程，程序应该显示进程ID和被执行的命令。程序的输出需要类似于pstree(1)产生的输出，当然不一定要和pstree一样复杂。每个进程的父进程可以通过读取/proc/PID/status文件的PPid来获得。小心地处理以下可能性：执行所有/proc/PID目录时父进程终止（因此/proc/PID目录消失）。

12-3. 编写一个程序，列出打开了特定路径文件的所有进程。你可以通过检查所有/proc/PID/fd/\*符号链接来完成这个任务，你需要循环嵌套调用readdir来扫描所有/proc/PID目录，然后是每个/proc/PID目录下的所有/proc/PID/fd项。要读取/proc/PID/fd/n符号链接的内容，你需要使用readlink()，18.5节对其进行了描述。

# 第13章 文件I/O缓冲

为了提高速度和效率，I/O系统调用（内核）和I/O标准C库函数（stdio函数）在操作磁盘文件时会缓冲数据。在这一章，我们讨论这两种类型的缓冲，并考虑它们如何影响应用的性能。我们还会讨论影响和禁止这两种缓冲的各种技术，然后讨论Direct I/O技术，可以用来在某些情况下绕过内核缓冲。

## 13.1 文件I/O的内核缓冲：缓冲区缓存

在读写磁盘文件时，read()和write()系统调用并不会直接发起磁盘访问。相反只是简单地在用户空间缓冲区和内核buffer cache缓冲区之间复制数据。例如下面调用从用户空间内存缓冲区，向内核空间缓冲区传输了3个字节数据：

write(fd, "abc", 3);

这时候write()返回，然后在后面某个时候，内核向磁盘写入缓冲区数据（flush）。（因此我们说系统调用并没有与磁盘操作同步）。如果在这期间，另一个进程试图读取文件的这些字节，内核会自动从buffer cache中提供数据，而不是从文件中（文件中的内容实际上已经过期）。

对于输入也是类似，内核从磁盘中读取数据，并存储在内核缓冲区中。调用read()会从这个缓冲区中获取数据，直到缓冲区数据用完，这时候内核就会读取文件的下一段内容到buffer cache。（这是简化的描述，对于顺序文件访问，内核通常会执行预读，在读取进程需要之前，就会把文件的下一个块也读取到buffer cache中，我们会在13.5节再次讨论预读）。

这个设计的主要目标是实现快速的read()和write()，因为它们不需要等待（相对缓慢）的磁盘操作。同时这个设计还很高效，因为它减少了内核需要执行的磁盘数据传输的次数。

Linux内核没有对缓冲区缓存的大小上限强加固定的限制。内核会按需要尽可能地分配缓冲区缓存页，仅受限于可用物理内存数量和其它用途的物理内存使用需求（例如存放运行进程所需的文本和数据页）。如果可用内存不足，内核会冲洗某些修改的缓冲区缓存页到磁盘中，来释放并重用这些页。

严格地说，从内核2.4开始，Linux就不再维护独立的buffer cache。相反文件I/O缓冲区被含在页缓存中，还包含内存映射文件等页。无论如何，在本书的讨论中，我们使用术语buffer cache，因为历史上在UNIX实现中常用。

**缓冲区大小对I/O系统调用性能的影响**

不管我们执行1000次单字节写入，还是一次写入1000字节，内核都执行相同数量的磁盘访问。但是后者是最好的选择，因为它只需要一个系统调用，而前者则需要1000次。尽管系统调用比磁盘操作要快很多，但不管怎么样还是需要一定的时间，因为内核要执行陷阱、检查系统调用参数的合法性、在用户空间和内核空间之间传输数据（更多细节请参考3.1节）。

使用不同的BUF\_SIZE值来运行清单4-1中的程序，我们可以看到不同的缓冲区大小对文件I/O性能的影响（BUF\_SIZE常量指定每个read()和write()调用要传输的字节数）。表13-1显示了在Linux ext2文件系统中，使用不同的BUF\_SIZE值，这个程序复制1亿字节文件所需的时间。注意以下补充信息：

* “过去的时间”和“总CPU时间”两列的意思很明显。用户CPU和系统CPU列则分解了总CPU时间，分别列出了在用户模式下执行代码、和执行内核代码所花费的时间。
* 表中显示的测试使用了2.6.30 vanilla内核，文件系统为ext2，块大小为4096字节。

vanilla内核表示未打补丁的主线内核。这样的内核不同于多数发行版提供的内核，后者通常包含各种bug修复或额外特性的补丁。

* 表中的每一行显示了20次运行相同缓冲区大小的平均值。在这些测试中，以及本章后面的其它测试，在每次执行程序时都会卸载然后挂载文件系统，确保文件系统的缓冲区缓存被清空，不影响测试结果。我们使用shell的time命令来记时。

*表13-1：复制1亿字节文件所需的时间*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **BUF\_SIZE** | **时间（秒）** | | | |
| **过去的时间** | **总CPU** | **用户CPU** | **系统CPU** |
| 1  2  4  8  16  32  64  128  256  512  1024 | 107.43  54.16  31.72  15.59  7.50  3.76  2.19  2.16  2.06  2.06  2.05 | 107.32  53.89  30.96  14.34  7.14  3.68  2.04  1.59  1.75  1.03  0.65 | 8.20  4.13  2.30  1.08  0.51  0.26  0.13  0.11  0.10  0.05  0.02 | 99.12  49.76  28.66  13.26  6.63  3.41  1.91  1.48  1.65  0.98  0.63 |
| 4096  16384  65536 | 2.05  2.05  2.06 | 0.38  0.34  0.32 | 0.01  0.00  0.00 | 0.38  0.33  0.32 |

由于传输的数据总量是相同的，磁盘操作的数量也相同，表13-1说明的是read()和write()调用的开销。缓冲区大小为1时，需要调用1亿次read()和write()。缓冲区大小为4096时，大概需要调用每个系统调用24000次，此时已经接近于最佳性能。缓冲区大小过了这个点之后，性能不再有重大提升，因为此时read()和write()系统调用的开销，相比在用户空间和内核空间之间传输数据以及执行实际的磁盘I/O，已经是可以忽略了。

表13-1的最后一行可以粗略地估计用户空间和内核空间之间传输数据，以及文件I/O所需的时间。因为最后一行的系统调用数量已经非常小，它们花费的时间几乎可以忽略。因此我们可以认为系统CPU时间就是用户空间和内核空间传输数据所需的时间，而过去的时间则可以估算出数据从磁盘读取和写入磁盘所需的时间（我们马上就会看到，这个时间主要是磁盘读取消耗的）。

总之如果我们要读写文件的大量数据，可以使用更大的块来缓冲数据，这样可以执行更少的系统，从而大大地提高I/O性能。

表13-1的数据测量了一系列因素：执行read()和write()系统调用的时间、在用户和内核空间传输数据的时间、在内核缓冲区和磁盘之间传输数据的时间。让我们进一步考虑最后一个因素，很明显从输入文件传输数据到缓冲区缓存是不可避免的。但是我们可以看到write()在数据从用户空间传输到内核缓冲区缓存之后立即就返回。因为测试系统的RAM大小（4GB）远远超过文件的大小（100MB），我们可以认为在程序完成时，输出文件实际上还没有写入到磁盘。因此我们又做了另一个实验，我们使用不同的write()缓冲区大小，向文件中写入任意数量的数据，结果如表13-2所示。

表13-2的数据仍然是内核2.6.30上获取的，文件系统为ext2，块大小为4096，表中的每一行都是20次运行的平均值。我们在这里没有列出测试程序（filebuff/write\_bytes.c），不过在本书的源代码包中可以找到它。

*表13-2：向文件写入1亿字节所需的时间*

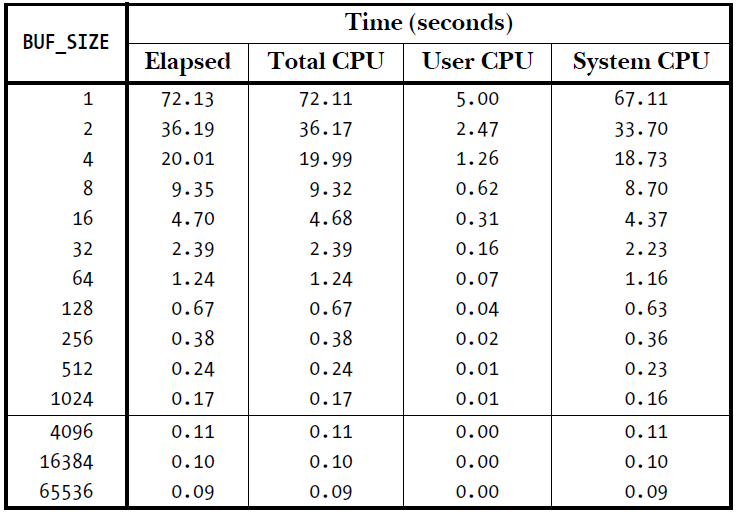


表13-2显示了不同的write()缓冲区大小，使用write()系统调用，以及从用户空间向内核缓冲区缓存传输数据所需的时间。对于更大的缓冲区大小，我们可以看到结果与表13-1中的数据变化非常大。例如缓冲区为65536，表13-1中过去的时间为2.06秒，而表13-2则为0.09秒。这是因为在后面一种情况下，并没有执行实际的磁盘I/O。换句话说，对于表13-1中的大缓冲区，主要的时间是用在读取磁盘数据上。

我们在13.3节将会看到，当我们强制输出操作阻塞，直到数据被传输至磁盘。write()调用的时间将剧烈增加。

最后值得注意的是，表13-2（以及表13-3）显示的信息只表现了文件系统瓶颈的一种形式。此外不同文件系统的结果可能会有变化。文件系统可以通过许多其它因素来测量，例如多用户高负载性能、文件创建和删除速度、在大目录中查找文件的时间、存储小文件所需的空间、在系统崩溃时维护文件完整性等等。当I/O性能或其它文件系统操作至关重要时，通常目标平台的应用瓶颈无可替代。

## 13.2 stdio库的缓冲

C库I/O函数（如fprintf(), fscanf(), fgets(), fputs(), fputc(), fgetc()）在操作磁盘文件时，会缓冲数据到大的块中来减少系统调用次数。因此使用stdio库，可以解放我们无需处理write()输出和read()输入的数据缓冲。

**设置stdio流的缓冲模式**

setvbuf()函数可以控制stdio库采用的缓冲模式。

|  |
| --- |
| #include <stdio.h>  int setvbuf(FILE \*stream, char \*buf, int mode, size\_t size);  成功时返回0；出错时返回非0 |

stream参数指定要修改缓冲的文件流。在流打开之后，setvbuf()必须在任何其它stdio函数之前使用。成功调用setvbuf()之后，会影响指定流所有接下来的stdio操作的行为。

不要把stdio库的streams流和System V的STREAMS机制混淆。Linux主干内核没有实现System V STREAMS机制。

buf和size参数指定stream要使用的缓冲区。这两个参数可以按以下两种方式指定：

* 如果buf非NULL，则它指向的size字节内存块将作为stream流的缓冲区。此后buf指向的缓冲区将被stdio库使用，因此这个内存块要么静态分配，要么在堆上动态分配（使用malloc()或类似函数）。千万不要使用栈上分配的局部变量，因为函数返回栈帧回收时将导致严重的错误。
* 如果buf为NULL，则stdio库自动为stream流分配一个缓冲区（除非我们选择不缓冲I/O，下面讨论）。SUSv3允许但不要求实现必须使用参数size来分配缓冲区。在glibc实现中，这种情况下忽略size参数。

mode参数指定缓冲类型，取以下值之一：

**\_IONBF**

不缓冲I/O。每个stdio库调用都会立即产生write()或read()系统调用。buf和size参数被忽略，可以分别指定为NULL和0。这是stderr的默认缓冲模式，因此错误输出总是确保能够立即显示。

**\_IOLBF**

采用行缓冲I/O。这是终端设备流使用的默认缓冲模式。对于输出流，数据将一直被缓冲，直到遇到换行字符（除非缓冲区满）；对于输入流，每次读取一行数据。

**\_IOFBF**

采用完全缓冲I/O。数据将以缓冲区大小的单元来读取（通过read()）或者写入（通过write()）。这是磁盘文件流使用的默认缓冲模式。

下面代码演示了setvbuf()的使用：

#define BUF\_SIZE 1024

static char buf[BUF\_SIZE];

if (setvbuf(stdout, buf, \_IOFBF, BUF\_SIZE) != 0)

errExit("setvbuf");

注意setvbuf()出错时返回非0值（不一定是-1）。

setbuf()函数基于setvbuf()实现，执行类似的任务。

|  |
| --- |
| #include <stdio.h>  void setbuf(FILE \*stream, char \*buf); |

除了setbuf没有返回值之外，调用setbuf(fp, buf)等价于：

setvbuf(fp, buf, (buf != NULL) ? \_IOFBF: \_IONBF, BUFSIZ);

buf参数要么指定为NULL（无缓冲），要么指向调用方分配的BUFSIZ字节的缓冲区。（BUFSIZ定义在<stdio.h>中，在glibc实现中，这个常量的值是8192，这是一个非常典型的值）。

setbuffer()函数类似于setbuf()，但是允许调用方指定buf的大小。

|  |
| --- |
| #define \_BSD\_SOURCE  #include <stdio.h>  void setbuffer(FILE \*stream, char \*buf, size\_t size); |

调用setbuffer(fp, buf, size)等价于下面调用：

setvbuf(fp, buf, (buf != NULL) ? \_IOFBF : \_IONBF, size);

SUSv3没有规定setbuffer()函数，但是在多数UNIX实现中都可用。

**冲洗（Flushing）stdio缓冲区**

无论采用哪种缓冲模式，在任何时候我们都可以使用fflush()库函数，强制冲洗stdio输出流的数据写出（通过write()冲洗到内核缓冲区）。fflush()函数冲洗stream指定流的输出缓冲区。

|  |
| --- |
| #include <stdio.h>  int fflush(FILE \*stream);  成功时返回0；出错时返回EOF |

如果stream为NULL，fflush()冲洗所有stdio缓冲区。

fflush()函数也可以应用于输入流，将导致任何已经缓冲的输入被丢弃（缓冲区会被程序下次从流中读取的数据重新填满）。

stdio缓冲区在相应的流关闭时会被自动冲洗。

在许多C库的实现中（包括glibc），如果stdin和stdout引用终端，则在每次从stdin中读取输入时，都会自动执行隐式的fflush(stdout)。作用是把所有已经写到stdout，但不包含终止换行字符（例如printf("Date: ")）的提示信息冲洗出去。但是SUSv3和C99并没有规定这个行为，也不是所有C库都实现了这一行为。可移植程序应该显式地调用fflush(stdout)来确保提示信息及时地显示。

C99对同时打开为输入和输出的流提出了两个要求。首先，输出操作之后不能立即执行输入操作，必须先调用fflush()或某个文件定位函数（fseek(), fsetpos(), 或rewind()）；第二，输入操作之后不能立即执行输出操作，必须先调用某个文件定位函数，除非输入操作已经遇到文件尾。

## 13.3 控制文件I/O的内核缓冲

我们可以强制冲洗输出文件的内核缓冲区。有时候某些应用（如数据库日志进程）必须确保输出确实写入到磁盘（或者至少已经到磁盘的硬件缓存中），这时候就需要控制内核缓冲。

在我们讨论控制内核缓冲的系统调用之前，我们需要首先考虑SUSv3的几个相关定义。

**同步I/O数据完整性和同步I/O文件完整性**

SUSv3定义了术语“同步I/O完成”，表示“I/O操作已经成功传输（到磁盘），或者诊断为不成功”。

SUSv3定义了两种不同类型的同步I/O完成。两种类型的区别主要与文件的元数据（“数据的数据”）相关，内核存储了文件数据和文件元数据。我们在14.4节讨论文件i-node时详细描述文件元数据，现在我们只需要知道文件元数据包含文件的各种信息，如文件拥有者、文件拥有者所属的组、文件权限、文件大小、硬链接数量、文件最后访问时间戳、最后修改时间戳、元数据最后修改时间戳、以及文件数据块指针等。

SUSv3定义的第一种同步I/O完成是“同步I/O数据完整性完成”。主要是确保文件数据更新时传输了足够的信息，稍后可以重新获取这些数据。

* 对于读操作，这意味着请求的文件数据已经（从磁盘）传输给进程。如果有任何影响请求数据的未决写入操作，将在读取之前首先把这些数据传输至磁盘。
* 对于写操作，这意味着写入操作指定的数据已经被传输至磁盘，而且重新获取这些数据所需的所有文件元数据也已经传输完毕。这里需要注意的重点是，要重新获取文件数据，并非所有修改的文件元数据属性都必须传输。必须传输的文件元数据的典型例子是文件大小（如果写入操作扩展了文件）。反过来，在接下来重新获取文件数据时，文件修改时间戳就不一定非得传输至磁盘。

SUSv3定义的另一种同步I/O完成是“同步I/O文件完整性完成”，这是同步I/O数据完整性完成的超集。区别在于这种类型的I/O完成在更新文件时，要求所有已更新的文件元数据被传输至磁盘，即使这些元数据对于随后重新读取文件数据无关紧要。

**控制文件I/O内核缓冲的系统调用**

fsync()系统调用把文件描述符fd相关的所有缓冲数据和元数据冲洗到磁盘。调用fsync()强制文件达到同步I/O文件完整性完成状态。

|  |
| --- |
| #include <unistd.h>  int fsync(int fd);  成功时返回0；出错时返回-1 |

fsync()调用在所有数据传输到磁盘设备（至少设备缓存）之后，才会返回。

fdatasync()系统调用类似于fsync()，但是只强制文件达到同步I/O数据完整性完成状态。

|  |
| --- |
| #include <unistd.h>  int fdatasync(int fd);  成功时返回0；出错时返回-1 |

fsync()需要两次磁盘操作，使用fdatasync()可能减少一次磁盘操作。例如文件数据已经修改，但是文件大小没有变化，则调用fdatasync()只要求更新文件数据。（我们在上面说过，同步I/O数据完成不要求文件修改时间戳等元数据传输至磁盘）。相反，调用fsync()则要求元数据也被传输至磁盘。

对于性能至上，或者不需要精确维护某些元数据（如时间戳）的应用来说，按上面的方式减少磁盘I/O操作的数量是很有用的。对于更新许多文件的应用来说，这样做可以获得很大的性能提升，因为文件数据和元数据通常存储在磁盘的不同区域，同时更新二者要求执行向前或向后的磁盘定位操作。

在Linux 2.2和更早版本，fdatasync()调用fsync()来实现，因此没有带来任何性能收获。

从内核2.6.17开始，Linux提供非标准的sync\_file\_range()系统调用，在冲洗文件数据时，能够比fdatasync()更精确地控制。调用方可以指定冲洗的文件范围，并指定标志，控制系统调用是否阻塞在磁盘写入上。更多细节请参考sync\_file\_range手册页。

sync()系统调用冲洗所有包含更新文件信息（数据块、指针块、元数据等等）的缓冲区到磁盘中。

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

在Linux实现中，sync()在所有数据传输到磁盘设备（或者至少设备缓存）之后，才会返回。但是SUSv3允许sync()简单地调度I/O传输，并在I/O操作完成之前返回。

Linux有一个持续运行的内核线程，如果修改过的内核缓冲区在30秒内没有显式同步，内核线程就会冲洗这些缓冲区到磁盘中。这样做是为了确保缓冲区与相应的磁盘文件不会长时间不同步（否则系统崩溃等情况发生时将容易丢失数据）。在Linux 2.6中这个任务由pdflush内核线程执行。（在Linux 2.4中由kupdated内核线程执行）。

/proc/sys/vm/dirty\_expire\_centisecs文件指定了脏（dirty）缓冲区在被pdflush冲洗之前必须达到的时间（百分之一秒计量）。相同目录下的其它文件则控制pdflush操作的其它属性。

**使所有写入同步：O\_SYNC**

调用open()时指定O\_SYNC标志，将使随后的所有输出同步：

fd = open(pathname, O\_WRONLY | O\_SYNC);

在这个open()调用之后，向文件的每个write()都会自动冲洗文件数据和元数据到磁盘（也就是写入操作执行同步I/O文件完整性完成）。

老的BSD系统使用O\_FSYNC标志来提供O\_SYNC的功能。在glibc中，O\_FSYNC与O\_SYNC同义。

**O\_SYNC的性能影响**

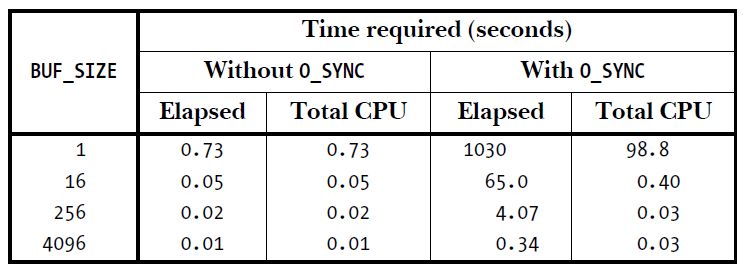
使用O\_SYNC标志（或者频繁地调用fsync, fdatasync(), sync()）会严重地影响性能。表13-3显示了写入1百万字节到新创建的文件中（ext2文件系统），在指定的缓冲区大小情况下，使用和不使用O\_SYNC标志时所需的时间。环境为无补丁的2.6.30内核、ext2文件系统、4906字节的块大小，使用filebuff/write\_bytes.c程序进行测试。表中每一行显示了在指定缓冲区大小下的20次测试平均值。

从表中可以看出，O\_SYNC急剧地增加了“过去的时间”，在1字节缓冲区的情况下，增加了1000多倍。同时请注意使用O\_SYNC写入时，过去的时间和总CPU时间之间巨大的偏差，这是因为缓冲区向磁盘传输时，程序一直被阻塞。

表13-3的结果还没有显示另一个O\_SYNC影响性能的因素。现代磁盘驱动器都拥有非常大的内部缓存，默认情况下，O\_SYNC只是导致数据被传输至该缓存。如果我们禁用磁盘缓存（使用命令hdparm -W0），则O\_SYNC的性能影响将更强烈。在1字节缓冲的情况下，过去的时间从1030秒提升到大约16000秒；在4096字节缓冲的情况下，过去的时间会从0.34秒提升到4秒。

总之，如果我们需要强制冲洗内核缓冲区，就应该考虑是否需要重新设计应用，使用大的write()缓冲区，或者审慎少量地调用fsync()或fdatasync()，而不是在打开文件时使用O\_SYNC标志。

*表13-3：O\_SYNC标志对写入1百万字节速度的影响*



**O\_DSYNC和O\_RSYNC标志**

SUSv3规定了另外两个打开文件状态标志，与同步I/O相关：O\_DSYNC和O\_RSYNC。

其中O\_DSYNC标志表示写入操作按同步I/O数据完整性完成的要求执行（fdatasync()）。这与O\_SYNC形成对比，后者写入操作按同步I/O文件完整性完成的要求执行（fsync()）。

O\_RSYNC标志需结合O\_SYNC或O\_DSYNC使用，作用是把后两个标志的作为扩展到读操作。打开文件时同时指定O\_RSYNC和O\_DSYNC，表示所有的读取操作也要满足同步I/O数据完整性要求（在执行读取操作之前，所有未决的写入操作都需要以O\_DSYNC完成）。打开文件时同时指定O\_RSYNC和O\_SYNC，则意味着所有读取操作都必须满足同步I/O文件完整性要求（在执行读取操作之前，所有的未决写入操作都必须以O\_SYNC完成）。

在内核2.6.33之前，Linux没有实现O\_DSYNC和O\_RSYNC标志，glibc头文件把这些常量的值定义为O\_SYNC。（实际上O\_RSYNC这样定义并不正确，因为O\_SYNC不能对读取操作提供任何作用）。

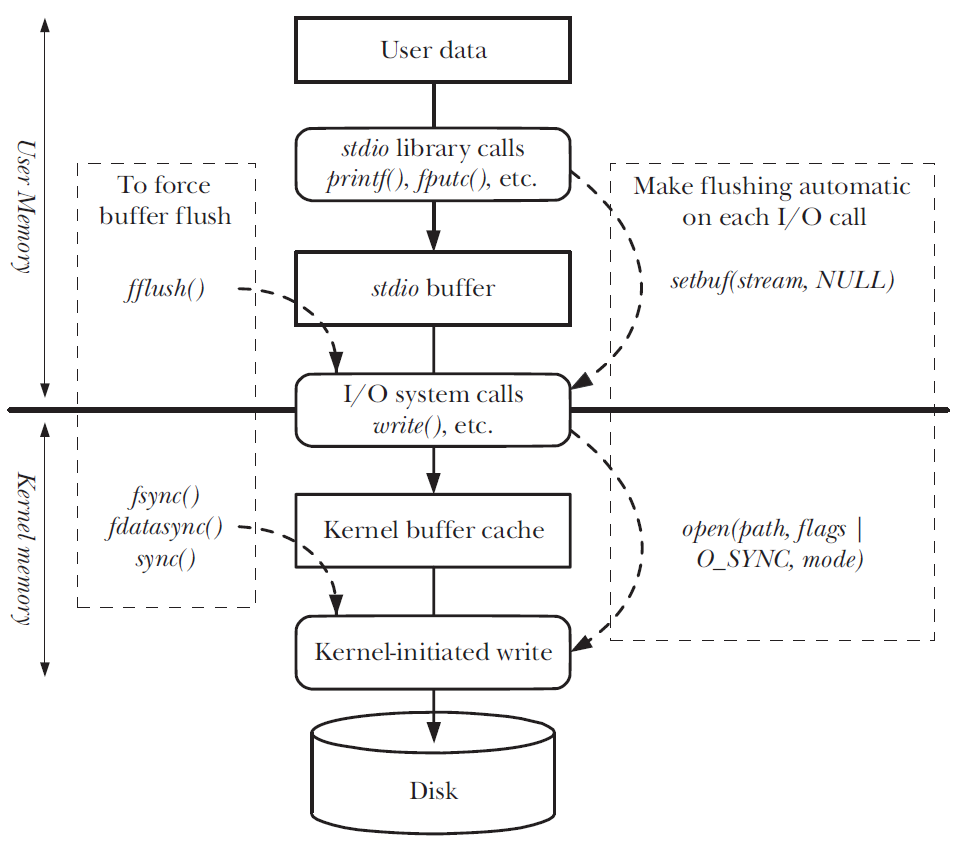
从内核2.6.33开始，Linux实现了O\_DSYNC。对于O\_RSYNC的实现，也很可能会在未来的内核发布中增加。

在内核2.6.33之前，Linux并没有完全实现O\_SYNC的语义。相反，O\_SYNC被实现为O\_DSYNC。为了维护针对旧内核构建的应用，保持应用行为一致；对于那些链接GNU C库旧版本的应用，系统将继续为O\_SYNC提供O\_DSYNC语义，即使是在新的Linux 2.6.33或之后内核中。

## 13.4 I/O缓冲小结

图13-1综合了stdio库和内核（对于输出文件）采用的缓冲，以及控制缓冲类型的机制。图中间从上到下，我们可以看到stdio库函数把用户数据传输到stdio缓冲区，这些都在用户内存空间中维护。当这个缓冲区满时，stdio库调用write()系统调用，把数据传输给内核缓冲区缓存（在内核内存中维护）。最后内核发起磁盘操作，将数据传输至磁盘。

图13-1的左侧显示了任意时候对缓冲区进行显式强制冲洗的调用。右侧则显示了用于自动冲洗的调用，通过禁止stdio缓冲、或使用同步文件输出系统调用，这样每个write()都立即冲洗到磁盘。



*图13-1：I/O缓冲小结*

## 13.5 向内核建议I/O模型

posix\_fadvise()系统调用允许进程向内核建议，告诉内核自己访问文件数据将要采用的I/O模型。

|  |
| --- |
| #define \_XOPEN\_SOURCE 600  #include <fcntl.h>  int posix\_fadvise(int fd, off\_t offset, off\_t len, int advice);  成功时返回0；出错时返回正的错误值 |

内核可以（但不是一定要）使用posix\_fadvise()提供的信息来优化自己的缓冲区缓存，从而为进程以及整个系统提高I/O性能。调用posix\_fadvise()对程序的语义没有任何影响。

fd参数是进程通知内核自己要访问文件的描述符，offset和len参数表示I/O模型建议要执行的文件区域：offset指定区域起始位置，len指定区域的字节大小，len为0表示从offset一直到文件末尾。（在内核2.6.6之前，len为0被解释为0字节）。

advice参数指定进程访问文件所希望的I/O模型，取值如下：

POSIX\_FADV\_NORMAL

进程对文件访问模型没有特别的建议，这也是不建议时的默认行为。在Linux中，这个操作会设置文件预计窗口为默认大小（128KB）。

POSIX\_FADV\_SEQUENTIAL

进程希望从低偏移位置向高偏移位置顺序地读取数据。在Linux中，这个操作会设置文件预计窗口为默认大小的两倍。

POSIX\_FADV\_RANDOM

进程希望以随机顺序访问文件数据。在Linux中，这个操作会禁止文件预读。

POSIX\_FADV\_WILLNEED

进程在短期内将会访问指定的文件区域。内核执行预读，把offset和len指定范围的数据预填到缓冲区缓存中。随后对文件调用read()就不会阻塞在磁盘I/O上，只需从缓冲区缓存中获取数据即可。内核不保证预读的数据在缓冲区缓存中保留的时间。如果其它进程或内核活动需要大量内存，则这些缓存页面最终可能被重用。换句话说，如果内存压力大，则我们应该确保posix\_fadvise()和随后read()调用之间的时间越短越好。（Linux特定的readahead()系统调用提供等同于POSIX\_FADV\_WILLNEED操作的功能）。

POSIX\_FADV\_DONTNEED

进程在短期内不希望访问文件的指定区域，向内核建议可以释放相应的缓存页面（如果有的话）。在Linux中，这个操作按两步执行：首先，如果底层设备当前没有阻塞在一系列排队的写入操作，内核会冲洗指定区域所有修改的页面；第二，内核尝试释放该区域占用的所有缓存页面。对于区域中已经修改的页面，只有第一步已经将页面写入底层设备中，第二步才会成功执行（也就是设备的写入队列已经不再阻塞）。由于应用无法控制设备的阻塞，确保缓存页面能够被释放的一种方法是在POSIX\_FADV\_DONTNEED操作之前，先使用sync()或fdatasync()系统调用手工冲洗缓冲区。

POSIX\_FADV\_NOREUSE

进程只希望访问指定文件区域的数据一次，不会再次访问。这提示内核在访问完成之后就可以释放这些页面。在Linux中，这个操作当前没有作用。

posix\_fadvise()是SUSv3新规定的接口，不是所有的UNIX实现都支持。Linux从内核2.6开始提供posix\_fadvise()。

## 13.6 绕过缓冲区缓存：Direct I/O

从内核2.4开始，Linux允许应用在执行磁盘I/O时绕过缓冲区缓存，直接从用户空间向文件或磁盘设备传输数据。我们把这个技术称为Direct I/O或原始I/O。

这里讨论的细节是Linux特定的，SUSv3并没有标准化。无论如何，多数UNIX实现都提供某种形式的直接I/O访问设备和文件。

Direct I/O有时候被错误地理解为能够获取更快的I/O性能。但是对于多数应用，使用direct I/O会强烈地降低性能。这是因为内核采用了一系列优化措施，通过缓冲区缓存、执行顺序预读、按磁盘块簇来执行I/O、允许进程访问相同文件来共享缓冲区缓存，能够大大地提高I/O性能。所有这些优化在使用direct I/O时都失去了。Direct I/O主要用于有特殊I/O需求的应用。例如数据库系统通常执行自己的缓存和I/O优化，不需要内核消耗CPU时间和内存去执行同样的任务。

我们可以对单个文件，或者对块设备（如磁盘）执行direct I/O。在使用open()打开文件或设备时指定O\_DIRECT标志。

内核2.4.10开始O\_DIRECT标志有效。不是所有的Linux文件系统和内核版本都支持这个标志。多数本地文件系统都支持O\_DIRECT，但是多数非UNIX文件系统（如VFAT）却不支持。我们可能需要测试文件系统是否支持（如果文件系统不支持O\_DIRECT，则open()会以EINVAL错误失败），或者阅读内核源代码来检查是否支持该标志。

如果某个进程打开文件时指定了O\_DIRECT，并且另一个进程以正常模式打开相同文件（因此使用了缓冲区缓存），则通过Direct I/O读取或写入数据与缓冲区缓存之间将不再一致。我们应该避免这种行为。

raw(8)手册页描述了直接访问磁盘设备更老的技术（现在已废弃）。

**Direct I/O的对齐限制**

由于Direct I/O（磁盘设备和文件）直接访问磁盘，我们必须注意执行Direct I/O时的一些限制：

* 传输的数据缓冲区必须对齐于块大小整数倍的内存边界；
* 数据开始传输的文件或设备偏移位置必须对齐于块大小的整数倍；
* 传输数据的长度必须是块大小的整数倍。

上述任何一个限制没有遵守都将导致EINVAL错误。在上面列表中，块大小是设备的物理块大小（通常是512字节）。

当执行Direct I/O时，Linux 2.4比Linux 2.6的限制更多：对齐、长度、偏移都必须是底层文件系统的逻辑块大小的整数倍。（典型的文件系统逻辑块大小是1024、2048、4096字节）。

**示例程序**

清单13-1提供了使用O\_DIRECT打开文件读取的简单例子。这个程序最多接受四个命令行参数，按顺序分别是：要读取的文件、从文件读取的字节数、程序从文件读取之前要seek的偏移、以及传递给read()的数据缓冲区对齐。最后两个参数是可选的，默认值分别为0和4096字节。下面是我们运行这个程序时的示例输出：

$ ./direct\_read /test/x 512 从偏移0读取512字节

Read 512 bytes 成功

$ ./direct\_read /test/x 256

ERROR [EINVAL Invalid argument] read 长度不是512的倍数

$ ./direct\_read /test/x 512 1

ERROR [EINVAL Invalid argument] read 偏移不是512的倍数

$ ./direct\_read /test/x 4096 8192 512

Read 4096 bytes 成功

$ ./direct\_read /test/x 4096 512 256

ERROR [EINVAL Invalid argument] read 对齐不是512的倍数

清单13-1的程序使用了memalign()函数来分配第一个参数整数倍的内存块。我们在7.1.4节讨论了memalign()函数。

*清单13-1：使用O\_DIRECT绕过缓冲区缓存*

-------------------------------------------------filebuff/direct\_read.c

#define \_GNU\_SOURCE /\* Obtain O\_DIRECT definition from <fcntl.h> \*/

#include <fcntl.h>

#include <malloc.h>

#include "tlpi\_hdr.h"

int

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

{

int fd;

ssize\_t numRead;

size\_t length, alignment;

off\_t offset;

void \*buf;

if (argc < 3 || strcmp(argv[1], "--help") == 0)

usageErr("%s file length [offset [alignment]]\n", argv[0]);

length = getLong(argv[2], GN\_ANY\_BASE, "length");

offset = (argc > 3) ? getLong(argv[3], GN\_ANY\_BASE, "offset") : 0;

alignment = (argc > 4) ? getLong(argv[4], GN\_ANY\_BASE, "alignment") : 4096;

fd = open(argv[1], O\_RDONLY | O\_DIRECT);

if (fd == -1)

errExit("open");

/\* memalign() allocates a block of memory aligned on an address that

is a multiple of its first argument. The following expression

ensures that 'buf' is aligned on a non-power-of-two multiple of

'alignment'. We do this to ensure that if, for example, we ask

for a 256-byte aligned buffer, then we don't accidentally get

a buffer that is also aligned on a 512-byte boundary.

The '(char \*)' cast is needed to allow pointer arithmetic (which

is not possible on the 'void \*' returned by memalign()). \*/

buf = (char \*) memalign(alignment \* 2, length + alignment) + alignment;

if (buf == NULL)

errExit("memalign");

if (lseek(fd, offset, SEEK\_SET) == -1)

errExit("lseek");

numRead = read(fd, buf, length);

if (numRead == -1)

errExit("read");

printf("Read %ld bytes\n", (long) numRead);

exit(EXIT\_SUCCESS);

}

-------------------------------------------------filebuff/direct\_read.c

## 13.7 混合库函数和系统调用的文件I/O

对同一个文件同时使用系统调用和标准C库函数来执行I/O，也是可能的。fileno()和fdopen()函数可以帮我们完成这个任务。

|  |
| --- |
| #include <stdio.h>  int fileno(FILE \*stream);  成功时返回文件描述符；出错时返回-1  FILE \*fdopen(int fd, const char \*mode);  成功时返回（新的）文件指针；出错时返回NULL |

给定一个流，fileno()返回相应的文件描述符（也就是stdio库为这个流打开的描述符）。然后这个文件描述符可以按正常的方式使用I/O系统调用，如read(), write(), dup(), fcntl()。

fdopen()函数是fileno()的相反。给定一个文件描述符，fdopen()创建一个相应的流，使用该描述符来执行I/O。mode参数和fopen()的参数相同；例如r是读取、w是写入、a是附加。如果这个参数与相应的文件描述符fd的访问模式不兼容，则fdopen()将失败。

fdopen()函数对于那些不指向普通文件的描述符特别有用。我们在后面章节将会看到，创建socket和管道的系统调用总是返回文件描述符。要对这些文件类型使用stdio库，我们就必须使用fdopen()来创建相应的文件流。

当结合使用stdio库函数和I/O系统调用对磁盘文件执行I/O时，我们必须时刻注意缓冲相关的问题。I/O系统调用直接传输数据到内核缓冲区缓存，而stdio库则等到用户空间缓冲区满时才调用write()传输缓冲区到内核缓冲区缓存。考虑以下写入到标准输出的代码：

printf("To man the world is twofold, ");

write(STDOUT\_FILENO, "in accordance with his twofold attitude.\n", 41);

在普通情况下，printf()输出会出现在write()输出之后，因此这段代码将得到以下输出：

in accordance with his twofold attitude.

To man the world is twofold,

当混合I/O系统调用和stdio库函数时，可能需要明智地使用fflush()来避免这个问题。我们还可以使用setvbuf()或setbuf()来禁止缓冲区，但这样做可能会影响应用的I/O性能，因为每个输出操作都会导致执行write()系统调用。

SUSv3对应用混合使用I/O系统调用和stdio库函数，提出了更多的要求。更多信息请参考General Information in the System Interfaces (XSH)卷的Interaction of File Descriptors and Standard I/O Streams节。

## 13.8 小结

内核执行了输入和输出数据的缓冲，同时stdio库也执行了数据缓冲。某些情况下，我们可能希望阻止缓冲，但我们需要理解这样做对应用可能会有性能影响。许多系统调用和库函数可以用来控制内核和stdio缓冲，并执行一次性缓冲冲洗。

进程可以使用posix\_fadvise()向内核建议自己将要对特定文件特定区域访问数据的I/O模型。内核可能会使用这个信息来优化缓冲区缓存，从而提高I/O性能。

Linux特定的open()打开标志O\_DIRECT，允许特殊应用绕过缓冲区缓存。

fileno()和fdopen()函数协助我们完成混合系统调用和标准C库函数，对相同文件执行I/O的任务。给定一个流，fileno()返回相应的文件描述符；fdopen()执行相反的操作，使用给定的文件描述符创建一个新的流。

**更多信息**

[Bach, 1986]讨论了System V的缓冲区缓存实现以及优势。[Goodheart & Cox, 1994]和[Vahalia, 1996]也讨论了System V缓冲区缓存的原理和实现。更多Linux的相关信息可以在[Bovet & Cesati, 2005]和[Love, 2010]中找到。

## 13.9 习题

13-1. 使用shell内建的time命令，在你的系统中测量清单4-1（copy.c）程序的时间：

1. 使用不同的文件和缓冲区大小进行实验。你可以在编译该程序时使用-DBUF\_SIZE=nbytes选项来设置缓冲区大小。
2. 修改open()系统调用包含O\_SYNC标志。这样做对不同缓冲区大小的速度有多大区别？
3. 尝试对不同文件系统（如ext3, XFS, Btrfd, JFS）执行时间测试。结果是类似的吗？当缓冲区由小变大时，变化趋势是一样的吗？

13-2. 使用不同的缓冲区大小和文件系统，测量filebuff/write\_bytes.c程序（本书源码包中提供）。

13-3. 下面语句的作用是什么？

fflush(fp);

fsync(fileno(fp));

13-4. 标准输出被重定向到终端或磁盘文件时，为什么下面代码的输出会不一样？请解释。

printf("If I had more time, \n");

write(STDOUT\_FILENO,

"I would have written you a shorter letter.\n", 43);

13-5. 命令tail [-n num] file可以打印file文件的最后num行（默认是10行）。使用I/O系统调用（lseek(), read(), write()等等）实现这个命令。注意本章讨论的缓冲区事项，以保证高效地实现其功能。

# 第14章 文件系统

第4、5、13章讨论了文件I/O，特别关注于普通（磁盘）文件。在本章和接下来的几章，我们更加深入地探讨一系列文件相关的主题：

* 本章讨论文件系统；
* 第15章描述文件相关联的各种属性，包括时间戳、拥有者、和权限；
* 第16和17章讨论Linux 2.6的两个新特性：扩展属性和访问控制列表（ACL）。扩展属性是向文件关联任意元数据的方法。ACL则是传统的UNIX文件权限模型的扩展。
* 第18章讨论目录和链接

本章主要关注文件系统，文件系统用于组织文件和目录集。我们解释一系列文件系统的概念，偶尔使用传统的Linux ext2文件系统作为特定的例子。我们还简短地说明了Linux上的某些日志文件系统。

本章的末尾，我们讨论了挂载和卸载文件系统的系统调用，以及用来获取已挂载文件系统信息的库函数。

## 14.1 设备特殊文件（设备）

本章频繁地提及磁盘设备，因此我们首先概述设备文件的概念。

设备特殊文件对应于系统中的一个设备。在内核中，每种类型的设备都有相应的设备驱动，后者为设备处理所有I/O请求。设备驱动是内核代码单元，实现了一组操作，通常对应于相关联硬件的输入和输出动作。设备驱动提供的API是固定的，包含的操作也对应于系统调用open(), close(), read(), write(), mmap(), 和ioctl()。所有设备驱动都提供统一的接口，隐藏了各种设备操作的不同性，因此实现了通用I/O（4.2节）。

某些设备是真实的，例如鼠标、磁盘、磁带等驱动器。其它一些设备则是虚拟的，意味着没有对应的硬件设备，相反内核提供了一层抽象（通过设备驱动），所提供的API与真实设备相同。

设备可以划分为以下两种类型：

* 字符设备，按一个字符一个字符的方式处理数据。终端和键盘是典型的字符设备。
* 块设备，按块来处理数据。块大小依赖于具体的设备类型，但通常是512字节的倍数。磁盘和磁带驱动器是典型的块设备。

设备文件也出现于文件系统中，就像其它文件一样，通常在/dev目录下。超级用户可以通过mknod命令来创建设备文件，特权程序（CAP\_MKNOD）也可以使用mknod()系统调用来执行相同的任务。

我们不详细描述mknod()（make file-system i-node）系统调用，因为它的使用方法非常直接，而且这个调用在今天的唯一作用是创建设备文件，这不是常见应用的需求。我们还可以使用mknod()来创建FIFO（44.7节），但是我们优先使用mkfifo()函数来完成这个工作。历史上，某些UNIX实现还可以使用mknod()来创建目录，但是这个用法目前已经被mkdir()系统调用取代。无论如何，某些UNIX实现为了向后兼容性，仍然保留了mknod()创建目录的能力（但Linux不是这样）。更多细节请参考mknod(2)手册页。

在早期版本的Linux中，/dev包含系统中所有可能设备的入口，即使该设备实际上并没有连接到系统中。这意味着/dev可能包含数千个未使用的项，降低了需要扫描/dev目录的程序的速度，并且我们无法通过扫描该目录来发现系统中已经实际连接的设备。在Linux 2.6，这些问题都通过udev程序得到了解决。udev程序依赖于sysfs文件系统，后者通过挂载到/sys下的伪文件系统，暴露了设备和其它内核对象的信息到用户空间。

**设备ID**

每个设备文件都有一个主ID数值和副ID数值。主ID标识设备的通用类型，内核用来为该类型设备查找适当的驱动。副ID则唯一地标识了该类型的某个特定设备。设备文件的主和副ID可以通过ls -l命令来显示。

设备的主副ID记录在设备文件的i-node中（我们在14.4节描述i-node）。每种设备驱动都注册自己关联到特定的主设备ID，这个关联提供了设备特殊文件和设备驱动之间的连接。设备文件的名字对于内核查找设备驱动是不相关的。

在Linux 2.4和更早版本中，由于主副ID都是8位，系统的设备总数受到了限制。加上主设备ID是固定的，而且需要集中分配（由Linux名称和号码分配管理局执行，<http://www.lanana.org/>），更是加剧了这个限制。Linux 2.6通过使用更多位保存主副设备ID（分别为12和20位），减轻了这个限制。

## 14.2 磁盘和分区

普通文件和目录一般存放于硬盘设备中（文件和目录也可以存在于其它设备中，如CD-ROM、闪存卡、和虚拟磁盘，但为了更好地讨论，我们主要集中于硬盘设备）。在接下来的几节，我们讨论磁盘如何组织和划分为分区。

**磁盘驱动器**

硬盘驱动器是一种机械设备，由一个或多个盘片组成，盘片以高速旋转（每分钟数千转）。磁盘表面使用磁记录的信息，通过径向移动读/写头跨越磁盘来获取和修改。物理上讲，磁盘表面的信息位于一组同心圆上，称为轨道。轨道本身又被划分为一组扇区，每个扇区包含一系列物理块。物理块的大小通常是512字节（或者某个倍数），代表了驱动器可以读取和写入的最小信息单元。

尽管现代磁盘很快，向磁盘读取和写入信息仍然需要花费大量时间。磁头必须首先移动到适当的轨道（寻道时间），然后驱动器必须等待适当的扇区旋转到磁头下方（转动延迟），最后还需要把所有的块数据传输过去（传输时间）。这类操作所需的总时间通常是毫秒级。相比起来，现代CPU能够在这段时间内执行百万条指令。

**磁盘分区**

每个磁盘都划分为一个或多个（不能重叠）分区。内核对每个分区都当作一个独立的设备，分别存放在/dev目录下。

系统管理员可以使用fdisk命令确定磁盘分区的数量、类型、和大小。fdisk -l列出磁盘的所有分区。Linux特定的/proc/partitions文件则列出了系统中每个磁盘分区的主副设备号、大小、和名字。

磁盘分区可以保存任意类型的信息，但通常包含以下几种：

* 文件系统保存普通文件和目录，14.3节讨论；
* 数据分区，作为原始模式设备访问，13.6节已经讨论过了（某些数据库管理系统使用这个技术）；
* 交换分区，内核用来做内存管理。

交换分区使用mkswap(8)命令来创建。特权进程（CAP\_SYS\_ADMIN）可以使用swapon()系统调用来通知内核，把某个磁盘分区用作交换分区。swapoff()系统调用执行相反的操作，告诉内核停止使用某个磁盘分区作用交换分区。SUSv3标准没有规定这些系统调用，但许多UNIX实现都提供。更多信息请参考swapon(2), swapon(8)手册页。

Linux特定的/proc/swaps文件可以用来显示当前系统启用的交换分区的信息。这些信息包括每个交换分区的大小，以及交换分区当前使用的数量。

## 14.3 文件系统

文件系统是组织化的普通文件和目录集。我们可以使用mkfs命令来创建文件系统。

Linux的优势之一就是支持许多类型的文件系统，包括以下：

* 传统的ext2文件系统；
* 各种本土的UNIX文件系统，例如Minix、System V、和BSD文件系统；
* Microsoft FAT、FAT32、和NTFS文件系统；
* ISO 9660 CD-ROM文件系统；
* Apple Macintosh HFS；
* 各种网络文件系统，包括广泛使用的Sun公司NFS（更多关于Linux对NFS实现的信息，可以在<http://nfs.sourceforge.net/>上找到）、IBM和Microsoft的SMB、Novell的NCP、以及卡耐基梅隆大学开发的Coda文件系统；
* 各种日志文件系统，包括ext3、ext4、Reiserfs、JFS、XFS、和Btrfs。

内核支持的文件系统类型可以通过Linux特定的/proc/filesystems文件获取。

Linux 2.6.14增加了用户空间文件系统（FUSE）机制。可以向内核添加钩子，完全在用户空间程序中实现一个文件系统，而不需要为内核打补丁或重新编译内核。更多信息请参考<http://fuse.sourceforge.net/>。

**ext2文件系统**

ext2（Second Extended File System）是很多年来Linux使用最广泛的文件系统，它是最初的Linux文件系统ext的升级版。近年来由于各种日志文件系统的兴起，ext2的使用逐渐衰落。虽然我们要讨论的是通用文件系统的概念，但以一个特定的文件系统实现作为例子来讲解，也是非常有用的。因此我们使用ext2作为本章随后主要的例子。

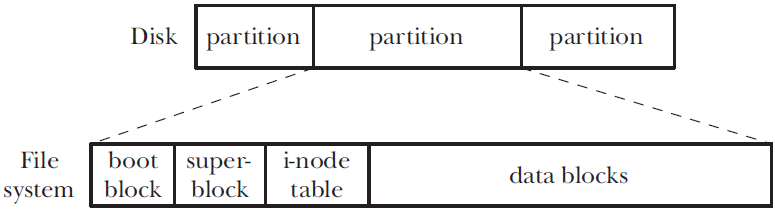
ext2文件系统由Remy Card编写，ext2的代码量非常小（大约5000行C代码），但却成为了其它几个文件系统实现的基本模型。ext2文件系统的主页是<http://e2fsprogs.sourceforge.net/ext2.html>。这个网站包含了一份ext2实现非常好的概览文件。David Rusling编写的The Linux Kernel一书也描述了ext2，该书可以在<http://www.tldp.org/>上找到。

**文件系统结构**

文件系统分配空间的基本单元是逻辑块，后者通常由多个连续的磁盘设备物理块组成。例如ext2的逻辑块大小为1024、2048、或4096字节（使用mkfs命令创建文件系统时，命令参数可以指定逻辑块的大小）。

特权程序（CAP\_SYS\_RAWIO）可以使用FIBMAP ioctl()操作来确定文件特定逻辑块对应的物理位置。该调用的第三个参数是输入输出型整数。在调用之前，需要设置这个参数为某个逻辑块数值（逻辑块从0起始）；在调用完成之后，这个参数会被设置为逻辑块所对应的物理块起始位置值。

图14-1显示了磁盘分区和文件系统之间的关系，并且显示了（通用）文件系统的一部分。



*图14-1：磁盘分区和文件系统的布局*

文件系统包含以下部分：

* 引导块：总是文件系统的第一个块。文件系统本身并不使用引导块，引导块包含的信息用于引导操作系统。尽管操作系统只需要一个引导块，所有文件系统都有一个引导块（多数引导块没有被使用）。
* 超级块：也是单一的块，位置紧随引导块，包含该文件系统相关的参数信息，包括：
  + i-node表的大小；
  + 文件系统逻辑块的大小；
  + 文件系统逻辑块的数量（文件系统的大小）

相同物理设备上的不同文件系统可以拥有不同的类型和大小，文件系统的参数设置也可以不同（例如块大小）。这也是为什么要划分磁盘为多个分区的原因之一。

* i-node表：文件系统中的每个文件或目录在i-node表中都有唯一的项。这项记录存储了文件的各种信息，下一节会更加详细地讨论i-node。i-node表有时候也称为i-list。
* 数据块：文件系统的大部分空间用作数据块，组成了该文件系统的所有文件和目录。

对于ext2文件系统来说，情况会比上面的描述更加复杂一些。在第一个引导块之后，文件系统被分解为一组相同大小的“块组”。每个块组都包含自己的超级块、参数信息、i-node表、以及数据块。通过存储每个文件的所有块到同一个块组，ext2文件系统希望顺序访问文件时降低seek时间。更多信息，请参考Linux源代码文件Documentation/filesystems/ext2.txt，e2fsprogs包中的dumpe2fs程序的源代码，以及[Bovet & Cesati, 2005]。

## 14.4 i-node

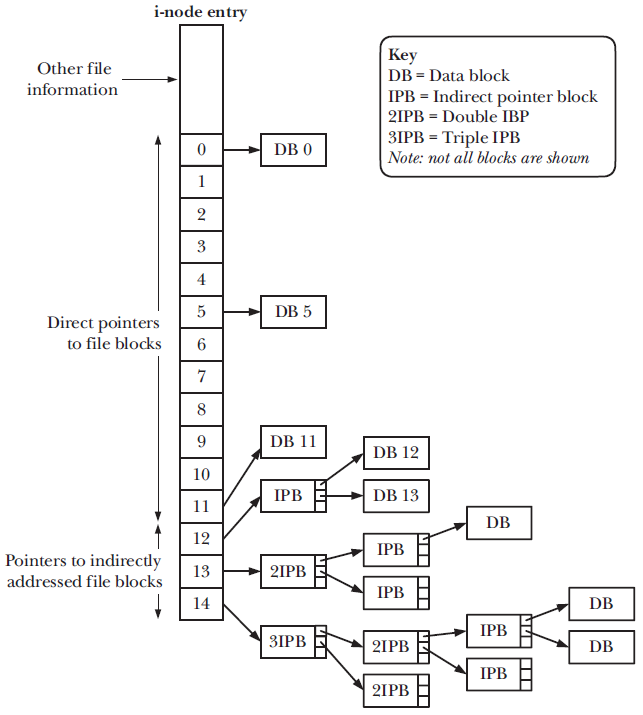
文件系统的i-node表为每个文件分配了一个i-node（index node）。i-node通过其在i-node表中的顺序位置（数值）来标识。文件的i-node号（或者简称为i-number）也就是ls -li命令显示的第一个域。i-node包含的信息如下：

* 文件类型（如普通文件、目录、符号链接、字符设备）
* 文件的拥有者（也称用户ID或UID）
* 文件所属的组（也称组ID或GID）
* 三类用户的访问权限：拥有者（用户）、组、其它（所有其它人）。15.4节提供了更多细节。
* 三个时间戳：文件最后访问时间（ls -lu）、文件最后修改时间（ls -l默认显示的时间）、文件状态最后变化时间（i-node信息最后一次变化，ls -lc显示）。值得注意的是，和其它UNIX实现一样，多数Linux文件系统并不记录文件创建时间。
* 文件的硬链接数量
* 文件的大小（字节）
* 为该文件分配的实际块数量，按每块512字节计算。这个数值和文件的字节大小并没有直接的对应关系，因为文件可能包含空洞（4.7节），从而相比文件的字节大小，实际需要分配的块可能更少。
* 指向文件数据块的指针

**ext2的i-node和数据块指针**

和多数UNIX文件系统一样，ext2文件系统不会连续地存储文件的数据块，甚至顺序也可能是乱的（尽管ext2确实会尽可能邻近地存储数据块）。为了分配文件的数据块，内核维护了i-node中的一组指针。ext2文件系统的i-node指针如图14-2所示。

文件系统不连续地存储文件数据块，能够更高效地使用空间。特别是降低了磁盘自由空间产生碎片的可能性——无数不连续的小块自由空间带来的空间浪费，这些块都太小而无法使用（如果要求连续存储文件块的话）。反过来，我们也可以说更高效地使用磁盘自由空间，代价则是把文件分割填满磁盘的各个自由空间。



*图14-2：ext2文件系统某个文件的块结构*

在ext2文件系统中，每个i-node包含15个指针。前面12个指针（图14-2中数值为0至11）指向文件在文件系统中前12个块的位置。第13个指针是一个“指向指针块的指针”，里面包含文件的第13块和随后的数据块的位置。指针块中的指针数量依赖于文件系统的块大小。每个指针需要4字节，因此每个指针块中的指针数量范围是256（1024字节块大小）至1024（4096字节块大小）。这样就能够允许非常大的文件，对于更大的文件，第14个指针（图中标为13）是双重间接指针——它指向一个指针块，后者的每个指针仍然指向指针块，然后该指针块的指针才指向文件的数据块。如果遇到超大文件，还有更深一层的间接指针：i-node的最后一个指针是三重间接指针。

这个表面看似复杂的系统，主要是设计用来满足各种需求。首先，它允许i-node结构的大小固定，同时又允许文件拥有任意大小。此外，它还允许文件系统不连续地存储文件的数据块，同时又允许通过lseek()随机地访问文件数据块；内核只需要计算出跟随的指针即可。最后由于多数文件系统大多数都是小文件，这个方法还能允许通过i-node的直接指针快速地访问文件数据块。

作为一个例子，作者测量了一个包含大约150000个文件的系统，超过30%的文件都小于1000字节，80%的文件小于10000字节。假设块大小为1024字节，上面所有文件都可以通过i-node的12个直接指针访问，这些指针所能访问的块的总大小是12288字节。使用4096字节块大小，更能进一步地提升至49152字节（该系统95%的文件都在这个限制之内）。

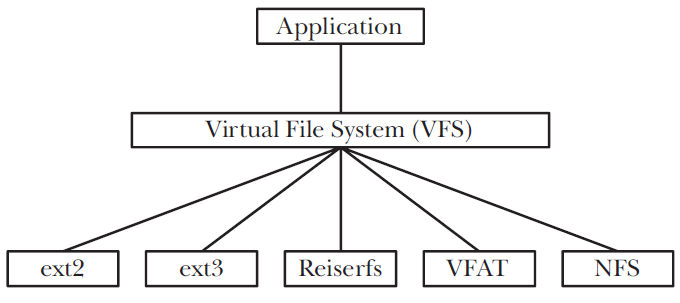
这个设计同时还允许巨大的文件大小，以4096字节块为例，文件大小的理论上限稍大于1024\*1024\*1024\*4096，大约是4TB（4096GB）。（我们说稍大于是因为直接、间接、双重间接指针还指向了数据块。这些数据块相对于三重指针所能引用的数据块来说，是无足轻重的）。

这个设计还带来另一个好处就是文件能够拥有空洞，如4.7节所讨论的那样。系统不用为空洞的null字节分配数据块，只需标记i-node中的适当指针为0，在间接指针块中标识出并不引用实际磁盘块即可。

## 14.5 虚拟文件系统（VFS）

Linux中的每个文件系统的实现细节都有区别，这些区别包括很多方面。例如文件块以什么方式分配、目录以什么方式组织等等。如果每个程序都需要理解每个文件系统的特定细节，则编写工作于不同文件系统的程序几乎是不可能的。虚拟文件系统（VFS，有时候也称为virtual file switch）是内核为解决这个问题而开发的特性，通过为文件系统操作（图14-3）提供一个抽象层，隐藏了不同文件系统的区别。VFS背后的概念是非常直接的：

* VFS定义了一组文件系统操作的通用接口。所有需要操作文件的程序使用这个通用接口来指定所需的操作。
* 每个文件系统提供自己的VFS接口实现。



*图14-3：虚拟文件系统*

按照这个设计，程序只需理解VFS接口，可以忽略各个文件系统实现的细节。

VFS接口包含对应于所有通用的文件系统和目录的系统调用操作，例如open(), read(), write(), lseek(), close(), truncate(), stat(), mount(), umount(), mmap(), mkdir(), link(), unlink(), symlink(), rename()。

VFS抽象层紧密地建模于传统的UNIX文件系统模型之上。当然，某些文件系统（特别是非UNIX文件系统）不支持所有的VFS操作（例如Microsoft的VFAT不支持符号链接的概念）。在这种情况下，底层文件系统传递一个错误代码到VFS层，指示不支持该操作，然后VFS再把这个错误代码传回给应用。

## 14.6 日志文件系统

ext2文件系统是传统UNIX文件系统很好的例子，也面临这类文件系统的一个经典局限：在系统崩溃后，必须在系统重启时执行文件系统一致性检查（fsck），以确保文件系统的完整性。这是必要的，因为在系统崩溃时，文件更新可能只完成了一部分，文件系统元数据（目录项、i-node信息、文件数据块指针）可能处于不一致状态，如果不进行修复，可能导致文件系统进一步被破坏。文件系统持续性检查确保文件系统元数据的一致性。当可能时会执行修复；否则丢弃无法修复的信息（可能包括文件数据）。

问题是一致性检查要求检查整个文件系统。在小型文件系统中，这可能需要几秒到几分钟之间；对于大型文件系统，这可能需要几个小时才能完成，这对于那些必须保持高可用性的系统（如网络服务器）是一个非常严重的问题。

日志文件系统消除了系统崩溃后冗长的文件系统一致性检查的要求。日志文件系统在实际修改元数据之前，先记录所有元数据更新信息到特殊的磁盘日志文件中。相关联的元数据更新被记录到一个组里（事务）。在事务正在处理的过程中，如果系统崩溃，这些日志可以用来快速地重做未完成的更新，并把文件系统恢复到一致的状态。（借用数据库的说法，我们可以说日志文件系统确保文件元数据事务总是作为完整的单元提交）。即使是超大型日志文件系统，在系统崩溃后，通常也可以在几秒内恢复可用，使其对于高可用性要求的系统具有非常大的吸引力。

记录日志最大的缺点在于增加了文件更新的时间，当然通过良好的设计，我们可以使这个开销非常低。

某些日志文件系统只确保文件元数据的一致性。因为它们并不记录文件数据的日志，数据仍然可能在系统崩溃中丢失。ext3, ext4, Reiserfs文件系统提供记录数据更新日志的选项，但是由于增加了工作量，可能导致更低的文件I/O性能。

Linux下可用的日志文件系统包括以下：

* Reiserfs是第一个整合到内核（2.4.1）中的日志文件系统。Reiserfs提供tail packing（或tail merging）特性：小文件（以及大文件的最后碎片）和文件元数据被塞进同一个磁盘块。由于许多系统拥有（以及某些应用创建）大量的小文件，这样做可以节省相当数量的磁盘空间。
* ext3文件系统向ext2增加了日志记录，该项目以最小化影响ext2的方式实现了这一目标。从ext2迁移到ext3非常简单（无需备份和还原），同时还可以反向迁移。ext3文件系统被整合到内核2.4.15。
* JFS由IBM开发，被整合到内核2.4.20。
* XFS（<http://oss.sgi.com/projects/xfs/>）最早由Silicon Graphics（SGI）公司在1990年代前期为自己的Irix而开发，Irix是SGI私有的UNIX实现。在2001年，XFS被移植到Linux，并且成为开源软件项目。XFS被整合到内核2.4.24。

在配置内核时，可以使用内核选项的File Systems菜单对内核支持的各种文件系统进行配置。

在本书写作的过程中，两个其它的日志文件系统正在开发中，它们还提供许多其它高级特性：

* ext4文件系统（<http://ext4.wiki.kernel.org/>）是ext3的下一代。部分实现最早被整合到内核2.6.19，许多特性随后被增加到后续内核版本中。ext4计划（或已经实现）的特性包括：extents（预留连续存储块）以及其它分配特性（主要目的在于降低文件碎片）、在线文件系统碎片整理、更快的文件系统检查、以及支持纳秒级时间戳。
* Btrfs（B-tree FS，通常发音为"butter FS"：<http://btrfs.wiki.kernel.org/>）是一个重新设计的新文件系统，目标是提供一系列现代特性，包括extents、可写的快照（提供等同于元数据和数据日志的功能）、数据和元数据checksum、在线文件系统检查、在线文件系统碎片整理、空间高效地打包小文件、空间高效地目录索引。Btrfs被整合到内核2.6.29。

## 14.7 单目录层次结构和挂载点

在Linux和其它UNIX系统中，所有文件系统的所有文件都存在于单一的目录树中。树根就是root目录"/"。其它文件系统被挂载到root目录之下，在整体目录层次结构中显示为子树。超级用户可以使用以下命令挂载一个文件系统：

$ mount device directory

这个命令把名为device的文件系统附加到directory指定的目录层次中，也就是文件系统的挂载点。我们可以修改文件系统挂载的位置，首先使用umount命令卸载文件系统，然后再次挂载到不同挂载点。

在Linux 2.4.19及之后版本，事情变得更加复杂。内核现在支持每个进程的挂载名字空间。这意味着每个进程可以拥有自己的文件系统挂载点，因此不同进程可能看到不同的单一目录层次结构。我们在28.2.1节讨论CLONE\_NEWNS标志时会更详细地解释这一点。

要列出当前已挂载的文件系统，我们可以使用mount命令不带任何参数，以下是示例输出（输出被有意删减）：

$ mount

/dev/sda6 on / type ext4 (rw)

proc on /proc type proc (rw)

sysfs on /sys type sysfs (rw)

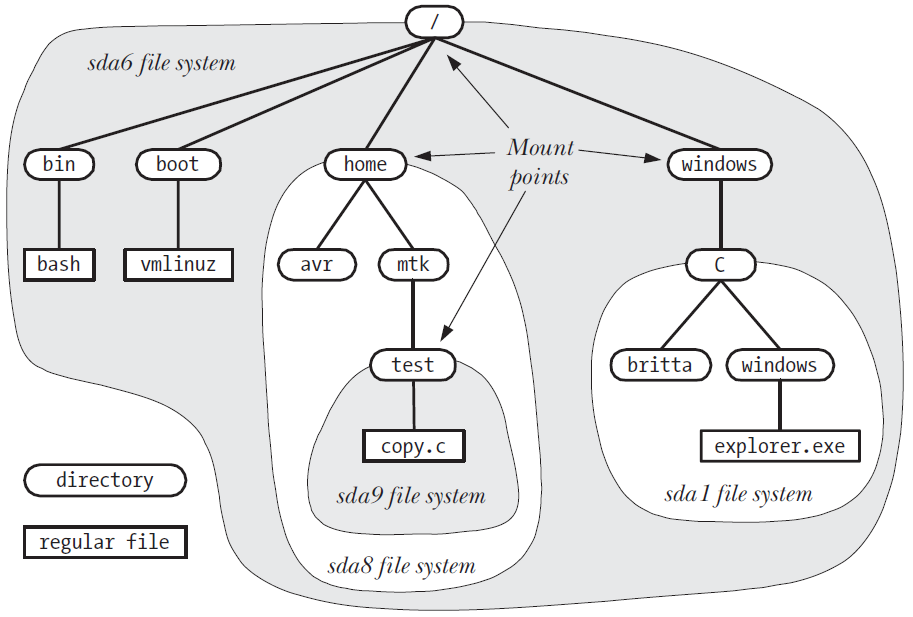
devpts on /dev/pts type devpts (rw,mode=0620,gid=5)

/dev/sda8 on /home type ext3 (rw,acl,user\_xattr)

/dev/sda1 on /windows/C type vfat (rw,noexec,nosuid,nodev)

/dev/sda9 on /home/mtk/test type reiserfs (rw)

图14-4显示了运行上面命令的系统的部分目录和文件结构。这个图显示了挂载点如何映射到目录层次结构。



*图14-4：文件系统挂载点和目录层次结构示例*

## 14.8 挂载和卸载文件系统

mount()和umount()系统调用允许特权进程（CAP\_SYS\_ADMIN）挂载和卸载文件系统。多数UNIX实现都提供这两个系统调用，但是SUSv3并没有对其标准化，而且它们的操作在不同UNIX实现和文件系统中存在差别。

在讨论这些系统调用之前，先来看三个非常有用的文件，包含了已经挂载和可以挂载的文件系统的各种信息：

* /proc/mounts：Linux特定的该虚拟文件中可以读取到当前已经挂载的文件系统列表。/proc/mounts是内核数据结构的接口，因此它总是包含已挂载文件系统的准确信息。

在增加了前面提到的“单个进程挂载名字空间”特性的情况下，每个进程现在都有一个/proc/PID/mounts文件，列出了组成进程挂载名字空间下的所有挂载点，此时/proc/mounts只是/proc/self/mounts文件的符号链接。

* /etc/mtab：mount(8)和umount(8)命令自动维护/etc/mtab文件，该文件包含类似于/proc/mounts的信息，但稍微详细一点。特别是/etc/mtab包含了mount(8)命令指定的文件系统特定选项，这在/proc/mounts文件中是没有的。但是由于mount()和umount()系统调用并不更新/etc/mtab文件，如果应用挂载或卸载文件系统却没有更新/etc/mtab，则该文件包含的信息可能不准确。
* /etc/fstab：系统管理员手工维护该文件，包含系统中所有可用文件系统的描述信息，mount(8), umount(8), fsck(8)命令需要使用这个文件。

/proc/mounts, /etc/mtab, /etc/fstab文件都采用相同的格式，fstab(5)手册页对此有详细描述。下面是/proc/mounts文件的一行示例：

/dev/sda9 /boot ext3 rw 0 0

上面一行包括6个域：

1. 已挂载设备的名字；
2. 设备的挂载点；
3. 文件系统类型；
4. 挂载标志。在上面示例中，rw表示该文件系统挂载为读写；
5. 这个数值用来控制dump(8)执行文件系统备份时的操作。这个域和下面这个都只用在/etc/fstab文件中；对于/proc/mounts和/etc/mtab，这两个域总是0；
6. 这个数值用来控制fsck(8)在系统启动时检查文件系统的顺序。

getfsent(3)和getmntent(3)手册页详细说明了从这些文件中读取记录的函数。

### 14.8.1 挂载文件系统：mount()

mount()系统调用挂载source指定设备包含的文件系统到target指定的目录下（挂载点）。

|  |
| --- |
| #include <sys/mount.h>  int mount(const char \*source, const char \*target, const char \*fstype,  unsigned long mountflags, const void \*data);  成功时返回0；出错时返回-1 |

mount()的前两个参数命名为source和target，是因为mount()除了挂载磁盘文件系统到某个目录下，还能执行其它任务。

fstype参数是一个字符串，标识设备包含文件系统的类型，例如ext4或btrfs。

mountflags参数是由零个或多个标志“或”起来的位掩码，表14-1列出了所有标志，下面会详细讨论。

mount()最后一个参数data是指向缓冲区的指针，其中的信息如何解释依赖于具体的文件系统。对于多数文件系统类型，这个参数是逗号分隔的选项设置字符串。这些选项的完整列表可以在mount(8)手册页中找到（或者文件系统相关的文档）。

*表14-1：mount()的mountflags值*

|  |  |
| --- | --- |
| **标志** | **作用** |
| MS\_BIND | 创建绑定挂载（Linux 2.4起） |
| MS\_DIRSYNC | 使目录同步更新（Linux 2.6起） |
| MS\_MANDLOCK | 允许文件强制锁 |
| MS\_MOVE | 自动移动挂载点到新位置 |
| MS\_NOATIME | 不更新文件的最后访问时间 |
| MS\_NODEV | 不允许访问设备 |
| MS\_NODIRATIME | 不更新目录的最后访问时间 |
| MS\_NOEXEC | 不允许程序被执行 |
| MS\_NOSUID | 禁止设置用户ID和设置组ID程序 |
| MS\_RDONLY | 只读挂载；不能创建和修改文件 |
| MS\_REC | 递归挂载（Linux 2.4.11起） |
| MS\_RELATIME | 只有最后访问时间早于最后修改时间或最后状态变化时间，才更新最后访问时间（Linux 2.6.20起） |
| MS\_REMOUNT | 以新的mountflags和data重新挂载 |
| MS\_STRICTATIME | 总是更新最后访问时间（Linux 2.6.30起） |
| MS\_SYNCHRONOUS | 使所有文件和目录同步更新 |

mountflags参数是上面标志的位掩码，用来修改mount()执行的操作。mountflags可以指定为以下零个或多个标志：

**MS\_BIND（Linux 2.4起）**

创建绑定挂载。我们在14.9.4节讨论这个特性。如果指定了这个标志，则fstype, mountflags, data参数将被忽略。

**MS\_DIRSYNC（Linux 2.6起）**

使目录更新同步。类似于open()的O\_SYNC标志的作用（13.3节），但只作用于目录更新。下面描述的MS\_SYNCHRONOUS标志则提供MS\_DIRSYNC功能的超集，确保文件和目录更新都同步执行。MS\_DIRSYNC标志允许应用确保目录更新（如open(papthname, O\_CREAT), rename(), link(), unlink(), symlink(), mkdir()）同步执行，而无需同步更新所有文件。FS\_DIRSYNC\_FL标志（15.5节）的作用类似于MS\_DIRSYNC，区别是FS\_DIRSYNC\_FL可以应用于单个目录。此外在Linux中，对引用目录的文件描述符调用fsync()提供单个目录同步更新的语义（SUSv3并没有规定这个Linux特定的fsync()行为）。

**MS\_MANDLOCK**

允许该文件系统下文件的强制记录锁。我们在第55章讨论记录锁。

**MS\_MOVE**

原子地移动source指定的现有挂载点到target指定的新位置。这个标志对应于mount(8)命令的--move选项。该标志的操作等同于卸载子树然后重新挂载到另外的位置，唯一的区别在于并不存在子树被卸载的时间段。source参数必须是之前mount()调用成功时指定的target字符串。当指定了这个标志时，fstype, mountflags, data参数将被忽略。

**MS\_NOATIME**

不更新该文件系统下的文件最后访问时间。这个标志的目的和下面马上要讨论的MS\_NODIRATIME标志一样，是为了消除每次访问文件时更新文件i-node所要求的额外磁盘访问操作。对于某些应用来说，维护文件的这个时间戳是无关紧要的，因此避免更新可以大大地提高性能。MS\_NOATIME标志的作用类似于FS\_NOATIME\_FL标志（15.5节），区别在于后者可以应用于单个文件。Linux还通过open()的O\_NOATIME标志提供类似的功能，可以对打开的单个文件选择这个行为（4.3.1节）。

**MS\_NODEV**

不允许访问该文件系统下的块和字符设备。这是一个安全特性，目的是防止用户执行越权操作。例如插入一个包含特殊设备文件的可移除磁盘，从而获得任意访问系统的权限。

**MS\_NODIRATIME**

不更新该文件系统下目录的最后访问时间（这个标志提供MS\_NOATIME功能的子集，后者阻止更新文件系统下所有类型文件的最后访问时间）。

**MS\_NOEXEC**

不允许执行该文件系统下的程序（或脚本）。对于包含非Linux可执行文件的文件系统，这个标志非常有用。

**MS\_NOSUID**

禁止该文件系统下的设置用户ID和设置组ID程序。这是一个安全特性，防止用户从可移除设备中运行设置用户ID和设置组ID的程序。

**MS\_RDONLY**

只读挂载文件系统，因此不能创建新文件，也不能修改现有文件。

**MS\_REC（Linux 2.4.11起）**

这个标志与其它标志（如MS\_BIND）结合使用，递归地应用挂载动作至子树下的所有挂载。

**MS\_RELATIME（Linux 2.6.20起）**

只有文件的最后访问时间小于或等于最后修改（或最后状态变化）时间戳时，才更新文件的最后访问时间戳。这个标志也拥有一定的性能优势，同时又能知道文件最后一次修改之后有没有被读取过。从Linux内核2.6.30开始，MS\_RELATIME标志提供的行为成为默认项（除非指定MS\_NOATIME标志），并且必须明确使用MS\_STRICTATIME标志才能得到经典的行为。此外，从Linux 2.6.30开始，如果最后访问时间戳已经过去24小时以上，即使当前值大于最后修改和最后状态变化时间，也会更新文件的最后访问时间戳。（对于某些监控目录查看文件是否被访问的系统程序非常有用）。

**MS\_REMOUNT**

修改某个已经挂载文件系统的mountflags和data参数（如只读文件系统变得可写）。当使用这个标志时，source和target参数必须与原来的mount()调用相同，而且fstype参数将被忽略。这个标志避免了卸载并重新挂载磁盘，因为有时候不可能执行这样的操作，例如任意进程打开了文件，或者当前工作目录处于当前文件系统（root文件系统永远无法卸载），我们就不能卸载该文件系统。另一个需要使用MS\_REMOUNT标志的例子是tmpfs（基于内存）文件系统（14.10节），在不丢失其内容的情况下根本无法卸载。注意不是所有的mountflags都能够被修改，细节请参考mount(2)手册页。

**MS\_STRICTATIME（Linux 2.6.30起）**

访问该文件系统下的文件时，总是更新文件的最后访问时间戳。这是Linux 2.6.30以前的默认行为。如果指定了MS\_STRICTATIME标志，则即使同时指定MS\_NOATIME和MS\_RELATIME标志，后面两个也将被忽略。

**MS\_SYNCHRONOUS**

该文件系统下的所有文件和目录同步更新。（对于文件来说，相当于文件总是以O\_SYNC标志打开）。

从内核2.6.15开始，Linux提供另外四个新的mount标志，以支持共享子树的概念。四个新标志是MS\_PRIVATE, MS\_SHARED, MS\_SLAVE, MS\_UNBINDABLE。（这些标志可以结合MS\_REC一起使用，将各自的作用应用到挂载子树的所有子挂载上）。共享子树设计用于某些高级文件系统特性，例如单个进程挂载名字空间（参考28.2.1节关于CLONE\_NEWNS的描述）、用户空间文件系统（FUSE）机制。共享子树机制允许文件系统以可控的方式挂载到多个挂载名字空间。共享子树的更多细节可在内核源代码文件Documentation/filesystems/sharedsubtree.txt和[Viro & Pai, 2006]中找到。

**示例程序**

清单14-1的程序为mount()系统调用提供了命令行接口，实际上就是mount命令的一个粗糙实现。下面的shell会话日志演示了该程序的使用方法。我们首先创建一个目录用做挂载点，然后挂载文件系统：

$ su 需要特权来挂载文件系统

Password:

# mkdir /testfs

# ./t\_mount -t ext2 -o bsdgroups /dev/sda12 /testfs

# cat /proc/mounts | grep sda12 验证设置

/dev/sda12 /testfs ext3 rw 0 0 不显示bsdgroups

# grep sda12 /etc/mtab

我们发现上面的grep命令没有产生输出，因为我们的程序不更新/etc/mtab文件。下面重新挂载文件系统为只读：

# ./t\_mount -f Rr /dev/sda12 /testfs

# cat /proc/mounts | grep sda12 验证已变化

/dev/sda12 /testfs ext3 ro 0 0

从/proc/mounts中显示的一行中的ro字符串表示这是只读挂载。

最后我们移动挂载点到新的位置，新位置在同一个目录层次结构下：

# mkdir /demo

# ./t\_mount -f m /testfs /demo

# cat /proc/mounts | grep sda12 验证已变化

/dev/sda12 /demo ext3 ro 0

清单14-1：使用mount()

------------------------------------------------------filesys/t\_mount.c

#include <sys/mount.h>

#include "tlpi\_hdr.h"

static void

usageError(const char \*progName, const char \*msg)

{

if (msg != NULL)

fprintf(stderr, "%s", msg);

fprintf(stderr, "Usage: %s [options] source target\n\n", progName);

fprintf(stderr, "Available options:\n");

#define fpe(str) fprintf(stderr, " " str) /\* Shorter! \*/

fpe("-t fstype [e.g., 'ext2' or 'reiserfs']\n");

fpe("-o data [file system-dependent options,\n");

fpe(" e.g., 'bsdgroups' for ext2]\n");

fpe("-f mountflags can include any of:\n");

#define fpe2(str) fprintf(stderr, " " str)

fpe2("b - MS\_BIND create a bind mount\n");

fpe2("d - MS\_DIRSYNC synchronous directory updates\n");

fpe2("l - MS\_MANDLOCK permit mandatory locking\n");

fpe2("m - MS\_MOVE atomically move subtree\n");

fpe2("A - MS\_NOATIME don't update atime (last access time)\n");

fpe2("V - MS\_NODEV don't permit device access\n");

fpe2("D - MS\_NODIRATIME don't update atime on directories\n");

fpe2("E - MS\_NOEXEC don't allow executables\n");

fpe2("S - MS\_NOSUID disable set-user/group-ID programs\n");

fpe2("r - MS\_RDONLY read-only mount\n");

fpe2("c - MS\_REC recursive mount\n");

fpe2("R - MS\_REMOUNT remount\n");

fpe2("s - MS\_SYNCHRONOUS make writes synchronous\n");

exit(EXIT\_FAILURE);

}

int

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

{

unsigned long flags;

char \*data, \*fstype;

int j, opt;

flags = 0;

data = NULL;

fstype = NULL;

while ((opt = getopt(argc, argv, "o:t:f:")) != -1) {

switch (opt) {

case 'o':

data = optarg;

break;

case 't':

fstype = optarg;

break;

case 'f':

for (j = 0; j < strlen(optarg); j++) {

switch (optarg[j]) {

case 'b': flags |= MS\_BIND; break;

case 'd': flags |= MS\_DIRSYNC; break;

case 'l': flags |= MS\_MANDLOCK; break;

case 'm': flags |= MS\_MOVE; break;

case 'A': flags |= MS\_NOATIME; break;

case 'V': flags |= MS\_NODEV; break;

case 'D': flags |= MS\_NODIRATIME; break;

case 'E': flags |= MS\_NOEXEC; break;

case 'S': flags |= MS\_NOSUID; break;

case 'r': flags |= MS\_RDONLY; break;

case 'c': flags |= MS\_REC; break;

case 'R': flags |= MS\_REMOUNT; break;

case 's': flags |= MS\_SYNCHRONOUS; break;

default: usageError(argv[0], NULL);

}

}

break;

default:

usageError(argv[0], NULL);

}

}

if (argc != optind + 2)

usageError(argv[0], "Wrong number of arguments\n");

if (mount(argv[optind], argv[optind + 1], fstype, flags, data) == -1)

errExit("mount");

exit(EXIT\_SUCCESS);

}

------------------------------------------------------filesys/t\_mount.c

### 14.8.2 卸载文件系统：umount()和umount2()

umount()系统调用卸载一个已经挂载的文件系统。

|  |
| --- |
| #include <sys/mount.h>  int umount(const char \*target);  成功时返回0；出错时返回-1 |

target参数指定要卸载的文件系统挂载点。

在Linux 2.2和更早版本，文件系统可以按两种方式标识：挂载点、或包含该文件系统的设备名。从内核2.4开始，Linux不再允许后一种标识方法，因为现在单个文件系统可以被挂载到多个位置，因此这样为target指定文件系统存在歧义。我们会在14.9.1节更加详细地解释这一点。

我们无法卸载正在使用（被占用）的文件系统：文件系统有打开的文件、进程的当前工作目录在文件系统中。对正被占用的文件系统调用umount()将得到EBUSY错误。

umount2()系统调用是umount()的扩展版本。它允许通过flags参数更加精细地控制卸载操作。

|  |
| --- |
| #include <sys/mount.h>  int umount2(const char \*target, int flags);  成功时返回0；出错时返回-1 |

flags位掩码参数由以下0个或多个标志“或”组合而成：

**MNT\_DETACH（Linux 2.4.11起）**

执行延迟（lazy）卸载。对挂载点打上标记，新进程不能再访问该文件系统，但正在使用文件系统的进程可以继续访问。当所有进程不再使用挂载时，文件系统将被实际地卸载掉。

**MNT\_EXPIRE（Linux 2.6.8起）**

标记挂载点为“过期”。如果最早的umount2()调用指定了这个标志，而且挂载点不busy，则该umount2()调用以EAGAIN错误失败，挂载点被标记为“过期”。（如果挂载点busy，则调用以EBUSY错误失败，并且不会标记挂载点为过期）。只要随后没有进程使用该挂载，挂载点将一直保持过期。再次指定MNT\_EXPIRE标志调用umount2()会卸载这个过期的挂载点。这个机制可以用来卸载那些一段时间内没有被使用过的文件系统。这个标志不能与MNT\_DETACH或MNT\_FORCE同时使用。

**MNT\_FORCE**

即使设备busy，仍然强制卸载（仅限NFS挂载）。采用这个标志可能导致数据丢失。

**UMOUNT\_NOFOLLOW（Linux 2.6.34起）**

如果target是符号链接，不对其进行解引用。这个标志设计用于某些set-user-ID-root的程序，允许非特权用户执行卸载；这个标志主要是为了避免target是符号链接，指向不同的位置，从而可能导致的安全问题。

## 14.9 高级挂载特性

现在我们来看一些挂载文件系统时可以采用的更加高级的特性。我们使用mount(8)命令来演示这些高级特性的使用。当然，使用mount()系统调用的程序也能完成同样的操作。

### 14.9.1 挂载文件系统至多个挂载点

在内核2.4以前的版本，一个文件系统只能挂载到单一的挂载点。但是从内核2.4开始，一个文件系统可以被挂载到多个位置。由于每个挂载点都展现了相同的子树，任何一个挂载点的修改都可以在其它挂载点中看到，如下面shell会话所示：

$ su 需要特权来使用mount(8)

Password:

# mkdir /testfs 创建两个目录用于挂载点

# mkdir /demo

# mount /dev/sda12 /testfs 挂载文件系统到一个挂载点

# mount /dev/sda12 /demo 挂载文件系统到第二个挂载点

# mount | grep sda12 验证操作

/dev/sda12 on /testfs type ext3 (rw)

/dev/sda12 on /demo type ext3 (rw)

# touch /testfs/myfile 通过第一个挂载点修改文件

# ls /demo 在第二个挂载点查看该文件

lost+found myfile

ls命令的输出显示第一个挂载点（/testfs）下的修改，在第二个挂载点（/demo）下也是可见的。

我们在14.9.4节举了一个例子，说明了为什么挂载一个文件系统到多个挂载点是非常有用的。

正是由于Linux 2.4及之后的版本，设备可以挂载到多个挂载点，umount()系统调用不能使用设备名作为自己的参数。

### 14.9.2 堆叠多个挂载至相同挂载点

在内核2.4版本之前，一个挂载点只能使用一次。从内核2.4开始，Linux允许多个挂载堆叠到同一个挂载点上。每个新的挂载都会隐藏该挂载点之前可见的目录子树。当堆栈顶上的挂载被卸载时，之前隐藏的挂载又再次变得可见。如下面shell会话所示：

$ su 需要特权才能使用mount

Password:

# mount /dev/sda12 /testfs 在/testfs下创建第一个挂载

# touch /testfs/myfile 在子树中创建一个文件

# mount /dev/sda13 /testfs 堆叠第二个挂载到/testfs

# mount | grep testfs 验证操作

/dev/sda12 on /testfs type ext3 (rw)

/dev/sda13 on /testfs type reiserfs (rw)

# touch /testfs/newfile 在子树中创建一个文件

# ls /testfs 查看子树中的文件

newfile

# umount /testfs 从堆栈中弹出一个挂载

# mount | grep testfs

/dev/sda12 on /testfs type ext3 (rw) 现在/testfs上只有一个挂载

# ls /testfs 上一个挂载现在变得可见

lost+found myfile

堆叠挂载的一个用途是把新挂载堆叠到现有busy状态的挂载点。进程如果在老的挂载点打开了文件描述符、chroot()-jailed、或者当前工作目录处于挂载点下，可以继续操作这个挂载点；但是新进程访问这个挂载点则使用新的挂载。结合使用MNT\_DETACH卸载，可以实现多用户系统模式下的文件系统顺利迁移和切换。我们在14.10节讨论tmpfs文件系统时，还会看到另一个堆叠挂载非常有用的例子。

### 14.9.3 单次挂载的挂载标志选项

在内核版本2.4之前，文件系统和挂载点之间有一对一的对应关系。由于Linux 2.4及以后版本这个条件不再成立，14.8.1节讨论的某些mountflags值现在可以应用于单次挂载。这些标志包含MS\_NOATIME（Linux 2.6.16起）、MS\_NODEV、MS\_NODIRATIME（Linux 2.6.16起）、MS\_NOEXEC、MS\_NOSUID、MS\_RDONLY（Linux 2.6.26起）、和MS\_RELATIME。下面shell会话演示了MS\_NOEXEC标志应用于相同文件系统的不同挂载上的作用：

$ su

Password:

# mount /dev/sda12 /testfs

# mount -o noexec /dev/sda12 /demo

# cat /proc/mounts | grep sda12

/dev/sda12 /testfs ext3 rw 0 0

/dev/sda12 /demo ext3 rw,noexec 0 0

# cp /bin/echo /testfs

# /testfs/echo "Art is something which is well done"

Art is something which is well done

# /demo/echo "Art is something which is well done"

bash: /demo/echo: Permission denied

### 14.9.4 绑定挂载

从内核2.4开始，Linux允许创建绑定挂载。绑定挂载（使用mount()的MS\_BIND标志创建）允许一个目录或文件被挂载到文件系统层次结构下的其它位置。这样两个位置可以同时看到该目录或文件。绑定挂载类似于硬链接，但有两个区别：

* 绑定挂载可以跨文件系统挂载点（甚至是chroot jail）。
* 可以对目录执行绑定挂载。

我们可以在shell中使用mount(8)命令的--bind选项来创建绑定挂载，如下面例子所示。

在第一个例子中，我们绑定一个目录到另外的位置，并演示在目录下创建文件在另外的位置也是可见的：

$ su 需要特权来使用mount

Password:

# pwd

/testfs

# mkdir d1 创建目录，用于绑定到另外的位置

# touch d1/x 在目录下创建文件

# mkdir d2 创建d1要绑定的挂载点

# mount --bind d1 d2 建立绑定挂载：d1通过d2可见

# ls d2 验证通过d2可看到d1的内容

x

# touch d2/y 在d2目录下创建第二个文件

# ls d1 验证通过d1可看到d2的这个修改

x y

在第二个例子中，我们绑定一个文件到另一个位置，并演示通过其中之一修改文件，另一个挂载也是可见的：

# cat > f1 创建文件，用于绑定到另一位置

Chance is always powerful. Let your hook be always cast.

Type Control-D

# touch f2 这是新的挂载点

# mount --bind f1 f2 绑定f1为f2

# mount | egrep '(d1|f1)' 查看挂载点

/testfs/d1 on /testfs/d2 type none (rw,bind)

/testfs/f1 on /testfs/f2 type none (rw,bind)

# cat >> f2 修改f2

In the pool where you least expect it, will be a fish.

# cat f1 通过原始的f1文件可看到这个修改

Chance is always powerful. Let your hook be always cast.

In the pool where you least expect it, will be a fish.

# rm f2 不能删除，因为这是一个挂载点

rm: cannot unlink `f2': Device or resource busy

# umount f2 卸载

# rm f2 现在我们可以删除f2

我们需要使用绑定挂载的一个例子是创建chroot jail（18.12节）。我们不需要在jail中复制各种标准目录（如/lib），只需在jail中简单地为这些目录创建绑定挂载（最好是只读挂载）。

### 14.9.5 递归绑定挂载

默认情况下使用MS\_BIND为目录创建一个绑定挂载，则只有该目录会被挂载到新位置；如果源目录下有子挂载，则并不会复制这些子挂载到target目标挂载下。Linux 2.4.11增加了MS\_REC标志，可以与MS\_BIND结合使用，导致子挂载也被复制到目标挂载。这就称为“递归绑定挂载”。mount(8)命令提供--rbind选项在shell中实现这个功能，如以下shell会话所示。

首先我们创建一个目录树（src1）挂载到top下。同时top/sub下还包含一个子挂载（src2）：

$ su

Password:

# mkdir top 这是顶层挂载点

# mkdir src1 我们会把它挂载到top下

# touch src1/aaa

# mount --bind src1 top 创建普通绑定挂载

# mkdir top/sub 创建目录，用于top下的子挂载

# mkdir src2 我们会把它挂载到top/sub下

# touch src2/bbb

# mount --bind src2 top/sub 创建普通绑定挂载

# find top 验证top挂载树下的内容

top

top/aaa

top/sub 这是子挂载

top/sub/bbb

现在我们使用top作为源，创建另一个绑定挂载（dir1）。由于这个新挂载不是递归的，top下的子挂载不会被复制。

# mkdir dir1

# mount --bind top dir1 这里我们使用普通绑定挂载

# find dir1

dir1

dir1/aaa

dir1/sub

输出中没有/dir1/sub/bbb，表示子挂载top/sub并没有被复制。

现在我们使用top作为源创建一个递归绑定挂载（dir2）：

# mkdir dir2

# mount --rbind top dir2

# find dir2

dir2

dir2/aaa

dir2/sub

dir2/sub/bbb

输出中的dir2/sub/bbb显示子挂载top/sub已经被复制过来了。

## 14.10 虚拟内存文件系统：tmpfs

本章目前为止我们讨论的所有文件系统都存在于磁盘中，但是Linux还支持存在于内存中的“虚拟文件系统”的概念。对于应用来说，虚拟内存文件系统和其它任何文件系统完全一样，可以对文件和目录执行相同的操作（open(), read(), write(), link(), mkdir()等等）。不过一个重要的区别是：文件操作要快许多，因为不涉及磁盘访问。

Linux上已经开发了各种基于内存的文件系统，其中最复杂的当属tmpfs文件系统，最早出现于Linux 2.4。tmpfs文件系统区别于其它基于内存的文件系统，主要因为它是虚拟内存文件系统。意味着tmpfs不仅仅使用RAM，当RAM耗尽时，还会使用交换空间。（尽管这里讨论的tmpfs文件系统特定于Linux，多数UNIX实现都提供类似基于内存的文件系统）。

tmpfs文件系统是可选的Linux内核组件，通过CONFIG\_TMPFS选项进行配置。

要创建一个tmpfs文件系统，我们可以使用下面格式的命令：

# mount -t tmpfs source target

source可以是任意名字，唯一的作用是出现在/proc/mounts文件中，并被mount和df命令显示。target则是该文件系统的挂载点。注意我们不需要使用mkfs来提前创建文件系统，因为在mount()系统调用中，内核会自动地构建一个文件系统。

下面是使用tmpfs的一个例子，我们采用堆叠挂载（这样就不需要关心/tmp是否已经被使用）并创建一个tmpfs文件系统挂载到/tmp，如下所示：

# mount -t tmpfs newtmp /tmp

# cat /proc/mounts | grep tmp

newtmp /tmp tmpfs rw 0 0

有时候我们使用上面的命令（或者/etc/fstab中等价的记录）来提高特定应用的性能，这些应用大量地使用/tmp目录创建并使用临时文件（如编译器）。

默认情况下，tmpfs文件系统被允许增大到RAM大小的一半。我们可以使用mount的size=nbytes选项为文件系统指定一个不同的上限，在创建文件系统或重新挂载时都可以使用这个选项。（这个选项只是设置上限，tmpfs文件系统会根据实际的文件存储需要，消耗相应的内存和交换空间）。

如果我们卸载tmpfs文件系统，或者系统崩溃，则文件系统中的所有数据都将丢失。正因如此，这个文件系统才命名为tmpfs。

除了用户应用会使用到tmpfs文件系统，它还有两个特殊的用途：

* 内核内部会挂载一个不可见的tmpfs文件系统，用于实现System V共享内存（第48章）和共享匿名内存映射（第49章）。
* 挂载在/dev/shm下的tmpfs文件系统，glibc使用它实现POSIX共享内存和POSIX信号量。

## 14.11 获取文件系统信息：statvfs()

statvfs()和fstatvfs()库函数获取已挂载文件系统的相关信息。

|  |
| --- |
| #include <sys/statvfs.h>  int statvfs(const char \*pathname, struct statvfs \*statvfsbuf);  int fstatvfs(int fd, struct statvfs \*statvfsbuf);  成功时都返回0；出错时都返回-1 |

这两个函数的唯一区别是如何标识文件系统。对于statvfs()，我们使用pathname来指定文件系统中的任意文件名；对于fstatvfs()，我们使用fd来指定文件系统中的任意打开文件描述符。两个函数都返回一个statvfs结构体指针，包含文件系统的相关信息。结构体定义如下：

struct statvfs {

unsigned long f\_bsize; /\* 文件系统块大小(字节) \*/

unsigned long f\_frsize; /\* 基本的文件系统块大小(字节) \*/

fsblkcnt\_t f\_blocks; /\* 文件系统块的总数量(按f\_frsize) \*/

fsblkcnt\_t f\_bfree; /\* 可用块的总数量 \*/

fsblkcnt\_t f\_bavail; /\* 非特权进程可用块的总数量 \*/

fsfilcnt\_t f\_files; /\* i-node总数量 \*/

fsfilcnt\_t f\_ffree; /\* 可用的i-node总数量 \*/

fsfilcnt\_t f\_favail; /\* 非特权进程可用的i-node总数量 \*/

unsigned long f\_fsid; /\* 文件系统ID \*/

unsigned long f\_flag; /\* 挂载标志 \*/

unsigned long f\_namemax; /\* 文件系统允许的文件名最大长度 \*/

};

上面的注释已经清晰地说明了statvfs结构体大多数域的作用。我们特别标注以下信息：

* SUSv3规定fsblkcnt\_t和fsfilcnt\_t数据类型为整型。
* 对于多数Linux文件系统，f\_bsize和f\_frsize的值是相同的。但是某些文件系统支持“块片段”的概念，为文件末尾的数据分配空间时，如果不需要分配完整的块，就会分配一个更小的存储单元。这样就避免了分配完整块时的空间浪费。在这种文件系统中，f\_frsize是片段的大小，f\_bsize是整个块的大小。（UNIX文件系统的“片段”概念最早出现于1980年代早期的4.2BSD Fast文件系统）。
* 许多UNIX和Linux本土的文件系统支持为超级用户保留一部分文件系统块的概念，因此当文件系统满了之后，超级用户还可以继续登录系统，并且执行某些工作来解决这个问题。如果文件系统有保留的块，则f\_bfree和f\_bavail域的差就是文件系统保留的块数量。
* f\_flag域是用于挂载文件系统时的标志位掩码，包含的信息类似于mount()的mountflags参数。但是这个域使用的位常量名以ST\_开头，而mountflags常量以MS\_开头。SUSv3只规定了ST\_RDONLY和ST\_NOSUID常量，但glibc实现支持对应于mount()的mountflags参数的完整MS\_\*常量。
* f\_fsid域用于某些UNIX实现，返回文件系统的唯一标识。例如基于文件系统所在设备的标识生成。对于多数Linux文件系统，这个域为0。

SUSv3同时规定了statvfs()和fstatvfs()。在Linux（以及许多其它UNIX实现）中，这两个函数基于非常类似的statfs()和fstatfs()系统调用实现。（某些UNIX实现只提供statfs()系统调用，不提供statvfs()）。系统调用和库函数理论上的区别如下（除了某些域名字不同之外）：

* statvfs()和fstatvfs()函数返回f\_flag域，包含文件系统挂载标志的信息。（glibc通过扫描/proc/mounts或/etc/mtab文件来获取这个信息）。
* statfs()和fstatfs()系统调用返回f\_type域，表示文件系统的类型。（例如ext2文件系统返回0xef53）。

本书源代码包中的filesys子目录包含两个文件：t\_statvfs.c和t\_statfs.c，演示了statvfs()和statfs()的使用。

## 14.12 小结

系统的设备由/dev目录下的文件项来表示。每个设备都有相应的设备驱动，实现了一组标准的操作，包括对应于open(), read(), write(), close()等系统调用的操作。设备可以是真实的，表示连接了相应的硬件设备；也可以是虚拟的，表示没有连接实际的硬件设备，但内核提供相应的设备驱动，实现了与真实设备相同的API。

硬盘被划分为一个或多个分区，每个分区可以包含一个文件系统。文件系统是组织化的普通文件和目录集。Linux实现了大量的文件系统，包括传统的ext2文件系统。ext2文件系统概念上类似于早期的UNIX文件系统，由一个引导块、一个超级块、i-node表、和包含文件数据块的数据区域组成。每个文件在文件系统的i-node表中都有一条记录，包含了文件的许多信息，如类型、大小、链接数、拥有者、权限、时间戳、文件数据块指针等。

Linux提供许多日志文件系统，包括Reiserfs、ext3、ext4、XFS、JFS、Btrfs。日志文件系统在执行实际的文件更新操作之前，先记录元数据更新（某些文件系统还包括数据更新）到一个日志文件。这意味着当系统崩溃时，日志文件可以用来快速还原文件系统到一致的状态。日志文件系统的关键优点是避免了系统崩溃后，传统UNIX文件系统冗长的一致性检查。

Linux系统的所有文件系统都挂载到单一的目录树，目录/是根（root）。文件系统被挂载的目录树位置称为挂载点。

特权进程可以使用mount()和umount()系统调用来挂载和卸载文件系统。已挂载文件系统的信息可以通过statvfs()函数来获取。

**更多信息**

设备和设备驱动的详细信息，请参考[Bovet & Cesati, 2005]和[Corbet et al., 2005]（特别是后者）。内核源文件Documentation/devices.txt中也可以找到关于设备的一些有用信息。

有好几本书提供了关于文件系统的更多信息。[Tanenbaum, 2007]介绍了通用的文件系统结构和实现。[Bach, 1986]介绍了UNIX文件系统的实现，主要面向System V系统。[Vahalia, 1996]和[Goodheart & Cox, 1994]同样描述了System V 文件系统的实现。[Love, 2010]和[Bovet & Cesati, 2005]描述了Linux VFS的实现。

内核源代码目录Documentation/filesystems下可以找到各个文件系统的文档。Linux的多数文件系统实现都可以在网上找到相关资料。

## 14.13 习题

14-1. 编写一个程序，测量从一个目录中创建然后删除大量1字节文件所需的时间。程序创建的文件命名格式为xNNNNNN，其中NNNNNN替换为随机的六位数值。文件应该按名字生成的随机顺序创建，然后按数值递增的顺序进行删除（创建和删除的顺序必须不同）。文件数量（NF）和创建文件所在的目录由命令行参数指定。测量不同文件系统（ext2, ext3, XFS等）下不同的NF值（1000到20000）所需的时间。单个文件系统随着NF增长，你能看出时间变化的趋势吗？不同文件系统之间比较的情况如何？如果文件按递增顺序创建（x000001, x000002, x000003）然后按相同顺序删除，结果有变化吗？如果有变化，你认为可能是哪些原因？同样，不同文件系统下的测试结果有变化吗？

# 第15章 文件属性

本章研究文件的各种属性（文件元数据）。首先描述stat()系统调用，可返回包含许多文件属性的结构体，包括文件时间戳、文件拥有者、文件权限等。然后我们继续讨论许多更新文件属性的系统调用。（第17章继续文件权限的讨论，详细描述了访问控制列表）。本章末尾讨论了i-node标志（也称为ext2扩展文件属性），可用于控制内核处理文件的方式。

## 15.1 获取文件信息：stat()

stat(), lstat(), fstat()系统调用获取文件的信息，其中多数信息都是从文件i-node中得到。

|  |
| --- |
| #include <sys/stat.h>  int stat(const char \*pathname, struct stat \*statbuf);  int lstat(const char \*pathname, struct stat \*statbuf);  int fstat(int fd, struct stat \*statbuf);  成功时都返回0；出错时都返回-1 |

这三个系统调用的唯一区别是如何指定文件：

* stat()返回路径名指定的文件信息；
* lstat()类似stat()，但如果路径指定的文件是符号链接，则返回符号链接文件本身的信息，而不是符号链接所引用文件的信息；
* fstat()返回打开文件描述符所引用文件的信息。

stat()和lstat()系统调用不要求拥有文件的权限，但是要求pathname指定路径中所有父目录的执行（查询）权限。对于fstat()系统调用来说，如果提供了合法的文件描述符，fstat()总是成功。

所有三个系统调用都返回一个stat结构体，存储在statbuf指向的缓冲区中。stat结构体的定义如下：

struct stat {

dev\_t st\_dev; /\* 文件所在设备的ID \*/

ino\_t st\_ino; /\* 文件的i-node号 \*/

mode\_t st\_mode; /\* 文件类型和权限 \*/

nlink\_t st\_nlink; /\* 文件的（硬）链接数量 \*/

uid\_t st\_uid; /\* 文件拥有者的用户ID \*/

gid\_t st\_gid; /\* 文件拥有者的组ID \*/

dev\_t st\_rdev; /\* 设备特殊文件的ID \*/

off\_t st\_size; /\* 文件总大小(字节) \*/

blksize\_t st\_blksize; /\* 执行I/O的最佳块大小(字节) \*/

blkcnt\_t st\_blocks; /\* 已分配块(512B)的数量 \*/

time\_t st\_atime; /\* 文件的最后访问时间 \*/

time\_t st\_mtime; /\* 文件的最后修改时间 \*/

time\_t st\_ctime; /\* 文件的最后状态变化时间 \*/

};

stat结构体中各个域的数据类型全部由SUSv3标准规定。关于这些类型的更多信息请参考3.6.2节。

根据SUSv3的规定，当lstat()应用于符号链接时，只需要在st\_size域、st\_mode域的文件类型部分返回合法的信息即可。stat的其它域（如时间域）都不一定需要包含合法信息。这个规定允许系统不维护符号链接的这些域，某些系统出于效率的原因可能会选择这样实现。特别是早期UNIX标准允许符号链接实现为i-node或目录中的一项。如果采用了后一种实现，系统就无法提供stat要求的所有域。（在所有主流的当代UNIX系统中，符号链接都被实现为i-node）。在Linux中，lstat()应用于符号链接时将返回stat所有域的信息。

在接下来的几页，我们详细地讨论stat结构体的一些域，最后再以一个显示完整stat结构体信息的示例程序结束这部分的讨论。

**设备ID和i-node号**

st\_dev域标识了文件所在的设备。st\_ino域则包含文件的i-node号。st\_dev和st\_ino的组合可以在所有文件系统中唯一地标识一个文件。dev\_t类型记录了设备的主和副ID（14.1节）。

如果当前文件是一个设备，则st\_rdev域包含了该设备的主和副ID。

dev\_t的主和副ID可以使用两个宏来提取：major()和minor()。这两个宏声明的头文件在不同UNIX实现中可能不同。在Linux中，如果定义了\_BSD\_SOURCE宏，<sys/types.h>头文件将暴露major()和minor()的声明。

major()和minor()返回的整数类型在不同UNIX实现中也不同。为了可移植，我们需要把返回值强制转换为long，然后再打印输出（3.6.2节）。

**文件拥有者**

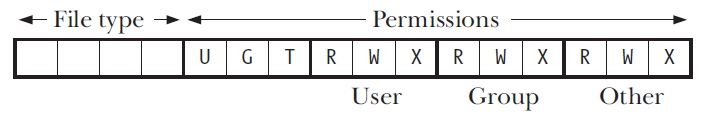
st\_uid和st\_gid域分别标识文件拥有者的用户ID和组ID。

**链接数**

st\_nlink域是文件的（硬）链接数量。我们在第18章详细讨论文件链接。

**文件类型和权限**

st\_mode域是位掩码，有两个作用：标识文件类型和文件权限。这个域的位布局如图15-1所示。



*图15-1：st\_mode位掩码布局*

把st\_mode域与常量S\_IFMT相与（&）可以提取出文件类型。（在Linux中，st\_mode域使用4位来存储文件类型。但是SUSv3没有规定实现如何表示文件类型，因此不同实现的细节可能不同）。与操作的结果值可以和一系列常量进行比较，从而确定文件类型，如下：

if ((statbuf.st\_mode & S\_IFMT) == S\_IFREG)

printf("regular file\n");

检查文件类型是常用的操作，因此系统提供了标准宏来简化上面代码：

if (S\_ISREG(statbuf.st\_mode))

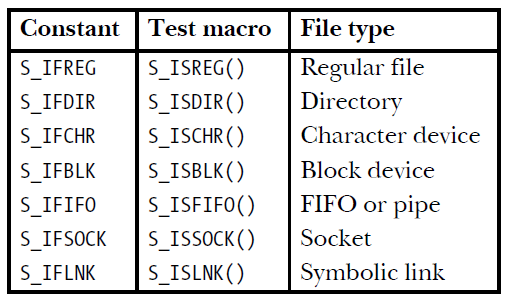
printf("regular file\n");

表15-1显示了文件类型检测宏的完整列表（定义在<sys/stat.h>中）。SUSv3规定了表15-1中的所有文件类型宏，而且在Linux中全部都可用。某些UNIX实现定义了额外的文件类型宏（如S\_IFDOOR是Solaris中的door文件）。只有调用lstat()时才有可能返回S\_IFLNK，因为调用stat()总是follow符号链接。

最初的POSIX.1标准并没有规定表15-1中第一列的常量，不过多数UNIX实现都提供这些常量。SUSv3标准则要求定义这些常量。

要从<sys/stat.h>中暴露S\_IFSOCK和S\_ISSOCK的定义，我们必须定义\_BSD\_SOURCE特性测试宏，或者定义\_XOPEN\_SOURCE的值大于或等于500。（这个规则在不同的glibc版本中稍有不同：有时候\_XOPEN\_SOURCE必须定义为大于或等于600）。

*表15-1：从stat结构体的st\_mode域中检测文件类型的宏*



st\_mode域的后面12位定义了文件的权限。我们在15.4节讨论文件权限位，现在我们只需要知道权限位的低9位指定了文件的权限，分别对应于拥有者、组、其它人的读、写、执行权限。

**文件大小、已分配块、最佳I/O块大小**

对于普通文件，st\_size域是文件的总大小（字节）；对于符号链接，这个域的值是链接所引用文件的路径名长度（字节）。

## 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 文件拥有者

### 15.3.1 新文件的拥有者

### 15.3.2 改变文件拥有者：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

## 24.1 fork(), exit(), wait(), execve()概述

## 24.2 创建新进程：fork()

### 24.2.1 父子进程文件共享

### 24.2.2 fork()的内存语义

## 24.3 vfork()系统调用

## 24.4 fork()之后的竞争条件

## 24.5 通过信号同步来避免竞争条件

## 24.6 小结

## 24.7 习题

# 第25章 进程结束

## 25.1 终止进程：\_exit()和exit()

## 25.2 进程终止的细节

## 25.3 Exit处理器

## 25.4 fork()、stdio缓冲区、\_exit()之间的交互

## 25.5 小结

## 25.6 习题

# 第26章 监控子进程

## 26.1 等待子进程

### 26.1.1 wait()系统调用

### 26.1.2 waitpid()系统调用

### 26.1.3 等待状态值

### 26.1.4 进程从信号处理器中终止

### 26.1.5 waitid()系统调用

### 26.1.6 wait3()和wait4()系统调用

## 26.2 孤儿进程（Orphan）和僵尸进程（Zombie）

## 26.3 SIGCHLD信号

### 26.3.1 创建SIGCHLD处理器

### 26.3.2 子进程停止时递送SIGCHLD

### 26.3.3 忽略死亡的子进程

## 26.4 小结

## 26.5 习题

# 第27章 程序执行

## 27.1 执行新程序：execve()

## 27.2 exec()库函数

### 27.2.1 PATH环境变量

### 27.2.2 指定程序参数为列表

### 27.2.3 传递调用方环境给新程序

### 27.2.4 执行描述符引用的文件：fexecve()

## 27.3 解释器脚本

## 27.4 文件描述符和exec()

## 27.5 信号和exec()

## 27.6 执行shell命令：system()

## 27.7 实现system()

## 27.8小结

## 27.9 习题

# 第28章 进程创建和程序执行的更多细节

## 28.1 进程会计

## 28.2 clone()系统调用

### 28.2.1 clone()的flags参数

### 28.2.2 clone子进程的waitpid()扩展

## 28.3 进程创建的速度

## 28.4 exec()和fork()对进程属性的影响

## 28.5 小结

## 28.6 习题

# 第29章 线程：介绍

## 29.1 概述

## 29.2 pthread API的背景细节

## 29.3 线程创建

## 29.4 线程终止

## 29.5 线程ID

## 29.6 等待线程终止

## 29.7 分离线程

## 29.8 线程属性

## 29.9 线程VS进程

## 29.10 小结

## 29.11 习题

# 第30章 线程：同步

## 30.1 保护共享变量访问：Mutex

### 30.1.1 静态分配的Mutex

### 30.1.2 Mutex加锁和解锁

### 30.1.3 Mutex的性能

### 30.1.4 Mutex死锁

### 30.1.5 动态初始化Mutex

### 30.1.6 Mutex属性

### 30.1.7 Mutex类型

## 30.2状态变化通知：条件变量

### 30.2.1 静态分配的条件变量

### 30.2.2 通知和等待条件变量

### 30.2.3 测试条件变量的Predicate

### 30.2.4 示例程序：等待任意线程终止

### 30.2.5 动态分配的条件变量

## 30.3 小结

## 30.4 习题

# 第31章 线程：线程安全和线程存储

## 31.1 线程安全（再论可重入）

## 31.2 一次性初始化

## 31.3 线程特定数据

### 31.3.1 库函数中的线程特定数据

### 31.3.2 线程特定数据API概述

### 31.3.3 线程特定数据API细节

### 31.3.4 使用线程特定数据API

### 31.3.5 线程特定数据的实现限制

## 31.4 线程本地存储

## 31.5 小结

## 31.6 习题

# 第32章 线程：取消线程

## 32.1 取消线程

## 32.2 取消状态和类型

## 32.3 取消点

## 32.4 测试线程取消

## 32.5 清理处理器

## 32.6 异步取消

## 32.7 小结

# 第33章 线程：更多细节

## 33.1 线程堆栈

## 33.2 线程和信号

### 33.2.1 UNIX信号模型如何映射到线程

### 33.2.2 操作线程信号掩码

### 33.2.3 向线程发送信号

### 33.2.4 理智地处理异步信号

## 33.3 线程和进程控制

## 33.4 线程实现模型

## 33.5 POSIX线程的Linux实现

### 33.5.1 LinuxThreads

### 33.5.2 NPTL

### 33.5.3 选择哪个线程实现？

## 33.6 pthread API的高级特性

## 33.7 小结

## 33.8 习题

# 第34章 进程组、会话和任务控制

## 34.1 概述

## 34.2 进程组

## 34.3 会话

## 34.4 控制终端和控制进程

## 34.5 前台和后台进程组

## 34.6 SIGHUP信号

### 34.6.1 shell对SIGHUP的处理

### 34.6.2 SIGHUP和控制进程的终止

## 34.7 任务控制

### 34.7.1 在shell中使用任务控制

### 34.7.2 实现任务控制

### 34.7.3 处理任务控制信号

### 34.7.4 孤儿进程组（再论SIGHUP）

## 34.8 小结

## 34.9 习题

# 第35章 进程优先级和调度

## 35.1 进程优先级（Nice值）

## 35.2 实时进程调度概述

### 35.2.1 SCHED\_RR策略

### 35.2.2 SCHED\_FIFO策略

### 35.2.3 SCHED\_BATCH和SCHED\_IDLE策略

## 35.3 实时进程调度API

### 35.3.1 实时优先级范围

### 35.3.2 修改和获取策略和优先级

### 35.3.3 放弃 CPU

### 35.3.4 SCHED\_RR时间片

## 35.4 CPU亲和力

## 35.5 小结

## 35.6 习题

# 第36章 进程资源

## 36.1 进程资源使用

## 36.2 进程资源限制

## 36.3 特定资源限制的细节

## 36.4 小结

## 36.5 习题

# 第37章 Daemon

## 37.1 概述

## 37.2 创建Daemon

## 37.3 Daemon编写指南

## 37.4 使用SIGHUP来重新初始化Daemon

## 37.5 使用syslog记录日志和错误信息

### 37.5.1 概述

### 37.5.2 syslog API

### 37.5.3 /etc/syslog.conf文件

## 37.6 小结

## 37.7 习题

# 第38章 编写安全的特权程序

## 38.1 是否需要设置用户ID和设置组ID的程序？

## 38.2 以最小权限执行操作

## 38.3 执行程序时要小心

## 38.4 避免暴露敏感信息

## 38.5 限制进程

## 38.6 小心信号和竞争条件

## 38.7 执行文件操作和文件I/O的陷阱

## 38.8 不要相信输入和环境

## 38.9 小心缓冲区溢出

## 38.10 小心拒绝服务攻击

## 38.11 检查返回状态和安全地失败

## 38.12 小结

## 38.13 习题

# 第39章 能力

## 39.1 能力的基本原理

## 39.2 Linux能力

## 39.3 进程和文件能力

### 39.3.1 进程能力

### 39.3.2 文件能力

### 39.3.3 进程允许和有效能力集的作用

### 39.3.4 文件允许和有效能力集的作用

### 39.3.5 进程和文件可继承能力集的作用

### 39.3.6 shell中查看和赋予文件能力

## 39.4 现代的能力实现

## 39.5 exec()时进程能力的转化

### 39.5.1 能力边界集

### 39.5.2 保留root语义

## 39.6 改变用户ID对进程能力的影响

## 39.7 编程改变进程能力

## 39.8 创建能力唯一环境

## 39.9 发现程序所需的能力

## 39.10 没有文件能力的老内核和系统

## 39.11 小结

## 39.12 习题

# 第40章 登录会计

## 40.1 utmp和wtmp文件概述

## 40.2 utmpx API

## 40.3 utmpx结构体

## 40.4 从utmp和wtmp文件中获取信息

## 40.5 获取登录名称：getlogin()

## 40.6 为登录会话更新utmp和wtmp文件

## 40.7 lastlog文件

## 40.8 小结

## 40.9 习题

# 第41章 共享库基础

## 41.1 对象库

## 41.2 静态库

## 41.3 共享库概述

## 41.4 创建和使用共享库 – A First Pass

### 41.4.1 创建共享库

### 41.4.2 位置无关的代码

### 41.4.3 使用共享库

### 41.4.4 共享库Soname

## 41.5 使用共享库的有用工具

## 41.6 共享库版本和命名惯例

## 41.7 安装共享库

## 41.8 兼容VS不兼容库

## 41.9 升级共享库

## 41.10 在对象文件中指定库搜索目录

## 41.11 运行时查找共享库

## 41.12 运行时符号解析

## 41.13 使用静态库而不是共享库

## 41.14 小结

## 41.15 习题

# 第42章 共享库高级特性

## 42.1 动态装载库

### 42.1.1 打开共享库：dlopen()

### 42.1.2 诊断错误：dlerror()

### 42.1.3 获取符号地址：dlsym()

### 42.1.4 关闭共享库：dlclose()

### 42.1.5 获取已装载符号的信息：dladdr()

### 42.1.6 在主程序中访问符号

## 42.2 控制符号可见性

## 42.3 链接器版本脚本

### 42.3.1 使用版本脚本控制符号可见性

### 42.3.2 符号版本

## 42.4 初始化和终止化函数

## 42.5 预装载共享库

## 42.6 监控动态链接器：LD\_DEBUG

## 42.7 小结

## 42.8 习题

# 第43章 进程间通信概述

## 43.1 IPC机制分类

## 43.2 通信机制

## 43.3 同步机制

## 43.4 IPC机制对比

## 43.5 小结

## 43.6 习题

# 第44章 管道和FIFO

## 44.1 概述

## 44.2 创建和使用管道

## 44.3 管道作为进程同步的方法

## 44.4 使用管道连接过滤器

## 44.5 使用管道与shell命令交互：popen()

## 44.6 管道和stdio缓冲区

## 44.7 FIFO

## 44.8 使用FIFO的客户端-服务器应用

## 44.9 非阻塞I/O

## 44.10 管道和FIFO的read()和write()语义

## 44.11 小结

## 44.12 习题

# 第45章 System V IPC介绍

## 45.1 API概述

## 45.2 IPC Key

## 45.3 相关的数据结构和对象权限

## 45.4 IPC标识符和客户端-服务器应用

## 45.5 System V IPC被调用时采用的算法

## 45.6 ipcs和ipcrm命令

## 45.7 获取所有IPC对象列表

## 45.8 IPC的限制

## 45.9 小结

## 45.10 习题

# 第46章 System V消息队列

## 46.1 创建或打开消息队列

## 46.2 交换消息

### 46.2.1 发送消息

### 46.2.2 接收消息

## 46.3 消息队列控制操作

## 46.4 消息队列相关的数据结构

## 46.5 消息队列的限制

## 46.6 显示系统中所有消息队列

## 46.7 客户端-服务器消息队列编程

## 46.8 使用消息队列的文件服务器应用

## 46.9 System V消息队列的缺点

## 46.10 小结

## 46.11 习题

# 第47章 System V信号量

## 47.1 概述

## 47.2 创建或打开信号量

## 47.3 信号量控制操作

## 47.4 信号量相关的数据结构

## 47.5 信号量初始化

## 47.6 信号量操作

## 47.7 处理多个阻塞的信号量操作

## 47.8 信号量撤消值

## 47.9 实现二进制信号量协议

## 47.10 信号量的限制

## 47.11 System V信号量的缺点

## 47.12 小结

## 47.13 习题

# 第48章 System V共享内存

## 48.1 概述

## 48.2 创建或打开共享内存段

## 48.3 使用共享内存

## 48.4 示例：通过共享内存传输数据

## 48.5 共享内存在虚拟内存中的位置

## 48.6 在共享内存中存储指针

## 48.7 共享内存控制操作

## 48.8 共享内存相关的数据结构

## 48.9 共享内存的限制

## 48.10 小结

## 48.11 习题

# 第49章 内存映射

## 49.1 概述

## 49.2 创建映射：mmap()

## 49.3 解除映射区域：munmap()

## 49.4 文件映射

### 49.4.1 私有文件映射

### 49.4.2 共享文件映射

### 49.4.3 边界情况

### 49.4.4 内存保护和文件访问模式的相互作用

## 49.5 同步映射区域：msync()

## 49.6 额外的mmap()标志

## 49.7 匿名映射

## 49.8 重新映射区域：mremap()

## 49.9 MAP\_NORESERVE和交换空间过量使用

## 49.10 MAP\_FIXED标志

## 49.11 非线性映射：remap\_file\_pages()

## 49.12 小结

## 49.13 习题

# 第50章 虚拟内存操作

## 50.1 改变内存保护：mprotect()

## 50.2 内存锁：mlock()和mlockall()

## 50.3 确定内存所在：mincore()

## 50.4 建议未来的内存使用模式：madvise()

## 50.5 小结

## 50.6 习题

# 第51章 POSIX IPC介绍

## 51.1 API概述

## 51.2 比较System V IPC和POSIX IPC

## 51.3 小结

# 第52章 POSIX消息队列

## 52.1 概述

## 52.2 打开、关闭、删除消息队列

## 52.3 描述符和消息队列的关联

## 52.4 消息队列属性

## 52.5 交换消息

### 52.5.1 发送消息

### 52.5.2 接收消息

### 52.5.3 带超时的消息发送和接收

## 52.6 消息通知

### 52.6.1 通过信号接收通知

### 52.6.2 通过线程接收通知

## 52.7 Linux特定特性

## 52.8 消息队列的限制

## 52.9 比较POSIX和System V消息队列

## 52.10 小结

## 52.11 习题

# 第53章 POSIX信号量

## 53.1 概述

## 53.2 命名信号量

### 53.2.1 打开命名信号量

### 53.2.2 关闭信号量

### 53.2.3 删除命名信号量

## 53.3 信号量操作

### 53.3.1 等待信号量

### 53.3.2 公告信号量

### 53.3.3 获取信号量的当前值

## 53.4 未命名信号量

### 53.4.1 初始化未命名信号量

### 53.4.2 销毁未命名信号量

## 53.5 与其它同步技术对比

## 53.6 信号量的限制

## 53.7 小结

## 53.8 习题

# 第54章 POSIX共享内存

## 54.1 概述

## 54.2 创建共享内存对象

## 54.3 使用共享内存对象

## 54.4 删除共享内存对象

## 54.5 各种共享内存API的对比

## 54.6 小结

## 54.7 习题

# 第55章 文件锁

## 55.1 概述

## 55.2 flock()文件锁

### 55.2.1 锁继承和释放的语义

### 55.2.2 flock()的限制

## 55.3 fcntl()记录锁

### 55.3.1 死锁

### 55.3.2 示例：交互式的锁程序

### 55.3.3 示例：锁函数库

### 55.3.4 锁限制和性能

### 55.3.5 锁继承和释放的语义

### 55.3.6 锁饥饿和排队锁请求的优先级

## 55.4 强制锁

## 55.5 /proc/locks文件

## 55.6 只运行程序的一个实例

## 55.7 老的锁技术

## 55.8 小结

## 55.9 习题

# 第56章 Sockets：介绍

## 56.1 概述

## 56.2 创建Socket：socket()

## 56.3 绑定Socket到地址：bind()

## 56.4 通用Socket地址结构体：struct sockaddr

## 56.5 流Socket

### 56.5.1 监听进来的连接：listen()

### 56.5.2 接受连接：accept()

### 56.5.3 连接到端Socket：connect()

### 56.5.4 流Socket I/O

### 56.5.5 终止连接：close()

## 56.6 数据报Socket

### 56.6.1 交换数据报：recvfrom()和sendto()

### 56.6.2 对数据报Socket使用connect()

## 56.7 小结

# 第57章 Sockets：UNIX Domain

## 57.1 UNIX Domain Socket地址：struct sockaddr\_un

## 57.2 UNIX Domain中的流Socket

## 57.3 UNIX Domain中的数据报Socket

## 57.4 UNIX Domain Socket权限

## 57.5 创建一对互连的Socket：socketpair()

## 57.6 Linux抽象Socket命名空间

## 57.7 小结

## 57.8 习题

# 第58章 Sockets：TCP/IP网络基础

## 58.1 因特网

## 58.2 网络协议和分层

## 58.3 数据链路层

## 58.4 网络层：IP

## 58.5 IP地址

## 58.6 传输层

### 58.6.1 端口号

### 58.6.2 用户数据报协议（UDP）

### 58.6.3 传输控制协议（TCP）

## 58.7 Requests For Comments（RFC）

## 58.8 小结

# 第59章 Sockets：Internet Domain

## 59.1 Internet Domain Socket

## 59.2 网络字节序

## 59.3 数据表示

## 59.4 Internet Socket地址

## 59.5 主机和服务转换函数概述

## 59.6 inet\_pton()和inet\_ntop()函数

## 59.7 客户端-服务器例子（数据报Socket）

## 59.8 域名系统（DNS）

## 59.9 /etc/services文件

## 59.10 协议无关的主机和服务转换

### 59.10.1 getaddrinfo()函数

### 59.10.2 释放addrinfo列表：freeaddrinfo()

### 59.10.3 诊断错误：gai\_strerror()

### 59.10.4 getnameinfo()函数

## 59.11 客户端-服务器例子（流Socket）

## 59.12 一个Internet Domain Socket库

## 59.13 已废弃的主机和服务转换API

### 59.13.1 inet\_aton()和inet\_ntoa()函数

### 59.13.2 gethostbyname()和gethostbyaddr()函数

### 59.13.3 getservbyname()和getservbyport()函数

## 59.14 UNIX vs Internet Domain Socket

## 59.15 更多信息

## 59.16 小结

## 59.17 习题

# 第60章 Sockets：服务器设计

## 60.1 迭代和并发服务器

## 60.2 迭代UDP echo服务器

## 60.3 并发TCP echo服务器

## 60.4 其它并发服务器设计

## 60.5 inetd（Internet Superserver）Daemon

## 60.6 小结

## 60.7 习题

# 第61章 Sockets：高级主题

## 61.1 流Socket的部分读取和写入

## 61.2 shutdown()系统调用

## 61.3 Socket特定的I/O系统调用：recv()和send()

## 61.4 sendfile()系统调用

## 61.5 获取Socket地址

## 61.6 深入TCP

### 61.6.1 TCP段的格式

### 61.6.2 TCP序号和确认

### 61.6.3 TCP状态机和状态转换图

### 61.6.4 TCP建立连接

### 61.6.5 TCP终止连接

### 61.6.6 对TCP Socket调用shutdown()

### 61.6.7 TIME\_WAIT状态

## 61.7 监控Socket：netstat

## 61.8 使用tcpdump来监控TCP流量

## 61.9 Socket选项

## 61.10 SO\_REUSEADDR Socket选项

## 61.11 多个accept()的标志和选项继承

## 61.12 TCP vs UDP

## 61.13 高级特性

### 61.13.1 带外（Out-of-Band）数据

### 61.13.2 sendmsg()和recvmsg()系统调用

### 61.13.3 传递文件描述符

### 61.13.4 获取发送方凭证

### 61.13.5 顺序包Socket

### 61.13.6 SCTP和DCCP传输层协议

## 61.14 小结

## 61.15 习题

# 第62章 终端

## 62.1 概述

## 62.2 获取和修改终端属性

## 62.3 stty命令

## 62.4 终端特殊字符

## 62.5 终端标志

## 62.6 终端I/O模式

### 62.6.1 Canonical模式

### 62.6.2 非Canonical模式

### 62.6.3 Cooked, Cbreak, Raw模式

## 62.7 终端行速（Bit Rate）

## 62.8 终端行控制

## 62.9 终端窗口大小

## 62.10 终端标识

## 62.11 小结

## 62.12 习题

# 第63章 可选I/O模型

## 63.1 概述

### 63.1.1 Level触发和Edge触发通知

### 63.1.2 通过可选I/O模型使用非阻塞I/O

## 63.2 I/O多路复用

### 63.2.1 select()系统调用

### 63.2.2 poll()系统调用

### 63.2.3 什么时候文件描述符准备好？

### 63.2.4 比较select()和poll()

### 63.2.5 select()和poll()的问题

## 63.3 信号驱动I/O

### 63.3.1 什么时候通知“I/O可用”？

### 63.3.2 信号驱动I/O使用精要

## 63.4 epoll API

### 63.4.1 创建epoll实例：epoll\_create()

### 63.4.2 修改epoll兴趣列表：epoll\_ctl()

### 63.4.3 等待事件：epoll\_wait()

### 63.4.4 深入epoll语义

### 63.4.5 epoll及I/O复用的性能对比

### 63.4.6 Edge触发通知

## 63.5 等待信号和文件描述符

### 63.5.1 pselect()系统调用

### 63.5.2 Self-Pipe技巧

## 63.6 小结

## 63.7 习题

# 第64章 伪终端

## 64.1 概述

## 64.2 UNIX 98 伪终端

### 64.2.1 打开未使用Master：posix\_openpt()

### 64.2.2 修改Slave拥有者和权限：grantpt()

### 64.2.3 解锁Slave：unlockpt()

### 64.2.4 获取Slave的名字：ptsname()

## 64.3 打开Master：ptyMasterOpen()

## 64.4 连接进程到伪终端：ptyFork()

## 64.5 伪终端I/O

## 64.6 实现script(1)

## 64.7 终端属性和窗口大小

## 64.8 BSD伪终端

## 64.9 小结

## 64.10 习题

# 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