

THE LINUX PROGRAMMING INTERFACE

A Linux and UNIX® System Programming Handbook

MICHAEL KERRISK



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

目录.....	3
前言.....	36
主题.....	36
目标读者.....	36
Linux 和 UNIX.....	37
使用和组织.....	37
例子程序.....	38
习题.....	39
标准和可移植性.....	39
Linux 内核和 C 库版本.....	40
其它语言使用编程接口.....	40
关于作者.....	40
致谢.....	40
许可.....	41
网站和例子程序源代码.....	41
反馈.....	41
第 1 章 历史和标准	42
1.1 UNIX 和 C 简史	42
1.2 Linux 简史	45
1.2.1 GNU 项目	46
1.2.2 Linux 内核	47
1.3 标准化	52
1.3.1 C 编程语言	52
1.3.2 第一个 POSIX 标准.....	53
1.3.3 X/Open 公司和开放组织	54
1.3.4 SUSv3 和 POSIX.1-2001	55

1.3.5 SUSv4 和 POSIX.1-2008	57
1.3.6 UNIX 标准时间线	58
1.3.7 实现标准	59
1.3.8 Linux、标准、和 Linux 标准基础	60
1.4 小结	61
第 2 章 基础概念	63
2.1 操作系统的核心：内核	63
2.2 Shell.....	66
2.3 用户和组	67
2.4 单一目录层次、目录、链接、和文件	68
2.5 文件 I/O 模型	72
2.6 程序	73
2.7 进程	73
2.8 内存映射	77
2.9 静态和共享库	78
2.10 进程间通信和同步	79
2.11 信号	80
2.12 线程	81
2.13 进程组和 shell 作业控制	81
2.14 会话、控制终端、和控制进程	82
2.15 伪终端	83
2.16 日期和时间	83
2.17 客户端-服务器体系架构	84
2.18 实时	85
2.19 /proc 文件系统.....	85
2.20 小结	86
第 3 章 系统编程概念	87
3.1 系统调用	87

3.2 库函数	90
3.3 标准 C 库; GNU C 库 (glibc)	91
3.4 系统调用和库函数的错误处理	92
3.5 本书示例程序的说明	95
3.5.1 命令行选项和参数	95
3.5.2 常用函数和头文件	95
3.6 可移植问题	106
3.6.1 特性测试宏	106
3.6.2 系统数据类型	109
3.6.3 各种可移植问题	112
3.7 小结	114
3.8 习题	115
第 4 章 文件 I/O: 通用 I/O 模型	116
4.1 概述	116
4.2 I/O 的通用性	119
4.3 打开文件: open().....	119
4.3.1 open()的 flags 参数	122
4.3.2 open()的错误	126
4.3.3 creat()系统调用	127
4.4 读取文件: read().....	128
4.5 写入文件: write().....	129
4.6 关闭文件: close().....	130
4.7 改变文件偏移: lseek().....	130
4.8 通用 I/O 模型之外的操作: ioctl().....	136
4.9 小结	137
4.10 习题	137
第 5 章 文件 I/O: 更多细节	138
5.1 原子性和竞争条件	138

5.2 文件控制操作: fcntl().....	142
5.3 打开文件状态标志	142
5.4 文件描述符和打开文件之间的关系	144
5.5 复制文件描述符	146
5.6 指定偏移位置的文件 I/O: pread()和 pwrite()	149
5.7 Scatter-Gather I/O: readv()和 writev()	150
5.8 截断文件: truncate()和 ftruncate().....	154
5.9 非阻塞 I/O	155
5.10 大文件 I/O	155
5.11 /dev/fd 目录	159
5.12 创建临时文件	160
5.13 小结	162
5.14 习题	163
第 6 章 进程	165
6.1 进程和程序	165
6.2 进程 ID 和父进程 ID.....	166
6.3 进程内存布局	167
6.4 虚拟内存管理	171
6.5 堆栈和堆栈帧	174
6.6 命令行参数 (argc, argv)	175
6.7 环境列表	178
6.8 执行非局部 goto: setjmp()和 longjmp().....	185
6.9 小结	192
6.10 习题	192
第 7 章 内存分配	194
7.1 在堆上分配内存	194
7.1.1 调整 Program Break: brk()和 sbrk()	194
7.1.2 在堆上分配内存: malloc()和 free()	195

7.1.3 实现 malloc()和 free()	200
7.1.4 在堆上分配内存的其它方法	204
7.2 在栈上分配内存: alloca().....	207
7.3 小结	208
7.4 习题	208
第 8 章 用户和组	209
8.1 密码文件: /etc/passwd	209
8.2 阴影密码文件: /etc/shadow	211
8.3 组文件: /etc/group	211
8.4 获取用户和组信息	213
8.5 密码加密和用户认证	219
8.6 小结	223
8.7 习题	223
第 9 章 进程凭证	224
9.1 实际用户 ID 和实际组 ID.....	224
9.2 有效用户 ID 和有效组 ID.....	224
9.3 设置用户 ID 和设置组 ID.....	225
9.4 保存的设置用户 ID 和保存的设置组 ID.....	227
9.5 文件系统用户 ID 和文件系统组 ID.....	228
9.6 附加组 ID	229
9.7 获取和修改进程凭证	229
9.7.1 获取和修改实际、有效、和保存的设置 ID	229
9.7.2 获取和修改文件系统 ID	235
9.7.3 获取和修改附加组 ID	236
9.7.4 修改进程凭证调用汇总	238
9.7.5 示例: 显示进程凭证	240
9.8 小结	242
9.9 习题	242

第 10 章 时间	244
10.1 日历时间	244
10.2 时间转换函数	245
10.2.1 <code>time_t</code> 转换为可打印格式	246
10.2.2 <code>time_t</code> 和分解时间互相转换	247
10.2.3 分解时间和可打印格式互相转换	249
10.3 时区	257
10.4 Locale	260
10.5 更新系统时钟	264
10.6 软时钟 (Jiffy)	266
10.7 进程时间	266
10.8 小结	271
10.9 习题	271
第 11 章 系统限制和选项	272
11.1 系统限制	273
11.2 运行时获取系统限制 (和选项)	277
11.3 运行时获取文件相关的限制 (和选项)	279
11.4 不确定限制	281
11.5 系统选项	282
11.6 小结	284
11.7 习题	285
第 12 章 系统和进程信息	286
12.1 /proc 文件系统	286
12.1.1 获取进程的信息: /proc/PID	287
12.1.2 /proc 下的系统信息	289
12.1.3 访问/proc 文件	291
12.2 系统标识: <code>uname()</code>	293
12.3 小结	295

12.4 习题	296
第 13 章 文件 I/O 缓冲	297
13.1 文件 I/O 的内核缓冲：缓冲区缓存	297
13.2 stdio 库的缓冲	301
13.3 控制文件 I/O 的内核缓冲	304
13.4 I/O 缓冲小结	309
13.5 向内核建议 I/O 模型	310
13.6 绕过缓冲区缓存：Direct I/O.....	311
13.7 混合库函数和系统调用的文件 I/O	314
13.8 小结	316
13.9 习题	316
第 14 章 文件系统	318
14.1 设备特殊文件（设备）	318
14.2 磁盘和分区	320
14.3 文件系统	321
14.4 i-node	324
14.5 虚拟文件系统（VFS）	326
14.6 日志文件系统	327
14.7 单目录层次结构和挂载点	329
14.8 挂载和卸载文件系统	331
14.8.1 挂载文件系统：mount()	332
14.8.2 卸载文件系统：umount()和 umount2().....	339
14.9 高级挂载特性	341
14.9.1 挂载文件系统至多个挂载点	341
14.9.2 堆叠多个挂载至相同挂载点	342
14.9.3 单次挂载的挂载标志选项	342
14.9.4 绑定挂载	343
14.9.5 递归绑定挂载	344

14.10 虚拟内存文件系统: tmpfs	346
14.11 获取文件系统信息: statvfs()	347
14.12 小结	349
14.13 习题	350
第 15 章 文件属性	351
15.1 获取文件信息: stat()	351
15.2 文件时间戳	358
15.2.1 修改文件时间戳: utime()和 utimes()	361
15.2.2 修改文件时间戳: utimensat()和 futimens()	363
15.3 文件所有权	365
15.3.1 新文件的所有权	365
15.3.2 改变文件所有权: chown(), fchown(), lchown()	366
15.4 文件权限	369
15.4.1 普通文件权限	370
15.4.2 目录权限	373
15.4.3 权限检查算法	373
15.4.4 检查文件可访问性: access()	375
15.4.5 设置用户 ID, 设置组 ID, sticky 位	377
15.4.6 进程文件模式创建掩码: umask()	378
15.4.7 修改文件权限: chmod()和 fchmod()	380
15.5 i-node 标志 (ext2 扩展文件属性)	382
15.6 小结	386
15.7 习题	387
第 16 章 扩展属性	389
16.1 概述	389
16.2 扩展属性实现细节	391
16.3 操作扩展属性的系统调用	392
16.4 小结	397

16.5 习题	397
第 17 章 访问控制列表	398
17.1 概述	398
17.2 ACL 权限检查算法	401
17.3 ACL 的长文本和短文本格式	402
17.4 ACL_MASK 条目和 ACL 组类别	403
17.5 getfacl 和 setfacl 命令	405
17.6 默认 ACL 和文件创建	407
17.7 ACL 实现限制	409
17.8 ACL API	409
17.9 小结	419
17.10 习题	420
第 18 章 目录和链接	421
18.1 目录和（硬）链接	421
18.2 符号（软）链接	424
18.3 创建和删除（硬）链接：link()和 unlink()	428
18.4 文件重命名：rename()	432
18.5 操作符号链接：symlink()和 readlink()	433
18.6 创建和删除目录：mkdir()和 rmdir()	434
18.7 删除文件或目录：remove()	436
18.8 读取目录：opendir()和 readdir()	436
18.9 遍历文件树：nftw()	443
18.10 进程的当前工作目录	449
18.11 目录文件描述符相关的操作	451
18.12 改变进程的根目录：chroot()	454
18.13 resolve 路径名：realpath()	456
18.14 解析路径名字符串：dirname()和 basename()	458
18.15 小结	460

18.16 习题	461
第 19 章 监控文件事件	463
19.1 概述	463
19.2 inotify API.....	464
19.3 inotify 事件	466
19.4 读取 inotify 事件	468
19.5 队列限制和/proc 文件.....	475
19.6 监控文件事件的旧机制: dnotify.....	476
19.7 小结	477
19.8 习题	477
第 20 章 信号: 基础概念	478
20.1 概念和概述	478
20.2 信号类型和默认动作	481
20.3 改变信号部署: signal().....	489
20.4 信号处理器介绍	491
20.5 发送信号: kill()	495
20.6 检查进程是否存在	497
20.7 发送信号的其它方法: raise()和 killpg()	499
20.8 显示信号描述信息	501
20.9 信号集	502
20.10 信号掩码 (阻塞信号递送)	505
20.11 未决信号	508
20.12 信号不会排队	508
20.13 改变信号部署: sigaction().....	513
20.14 等待信号: pause()	515
20.15 小结	516
20.16 习题	517
第 21 章 信号: 信号处理器	518

21.1 设计信号处理器	518
21.1.1 信号不会排队（回顾）	519
21.1.2 可重入和异步信号安全的函数	519
21.1.3 全局变量和 <code>sig_atomic_t</code> 数据类型	526
21.2 终止信号处理器的其它方法	527
21.2.1 从信号处理器中执行非局部跳转	527
21.2.2 异常地终止进程： <code>abort()</code>	532
21.3 在备用堆栈中处理信号： <code>sigaltstack()</code>	533
21.4 SA_SIGINFO 标志	537
21.5 系统调用的中断和重启	544
21.6 小结	548
21.7 习题	549
第 22 章 信号：高级特性	550
22.1 Core Dump 文件	550
22.2 递送、部署、和处理的特殊情况	553
22.3 可中断和不可中断的进程睡眠状态	554
22.4 硬件产生的信号	555
22.5 同步和异步信号生成	556
22.6 信号递送的时间和顺序	557
22.7 <code>signal()</code> 的实现和可移植性	559
22.8 实时信号	561
22.8.1 发送实时信号	564
22.8.2 处理实时信号	566
22.9 使用掩码来等待信号： <code>sigsuspend()</code>	570
22.10 同步等待信号	576
22.11 通过文件描述符获取信号	580
22.12 使用信号进行进程间通信	583
22.13 早期信号 API（System V 和 BSD）	584

22.14 小结	587
22.15 习题	589
第 23 章 定时器和睡眠	590
23.1 间隔定时器	590
23.2 定时器的调度和精确度	590
23.3 设置阻塞操作的超时	590
23.4 固定间隔暂停执行（睡眠）	590
23.4.1 低精度睡眠: sleep()	590
23.4.2 高精度睡眠: nanosleep()	590
23.5 POSIX 时钟	590
23.5.1 获取时钟的值: clock_gettime()	590
23.5.2 设置时钟的值: clock_settime()	590
23.5.3 获取特定进程或线程的时钟 ID	590
23.5.4 增强的高精度睡眠: clock_nanosleep()	590
23.6 POSIX 间隔定时器	590
23.6.1 创建定时器: timer_create()	590
23.6.2 装备和解除定时器: timer_settime()	590
23.6.3 获取定时器的当前值: timer_gettime()	591
23.6.4 删除定时器: timer_delete()	591
23.6.5 通过信号通知	591
23.6.6 定时器超时	591
23.6.7 通过线程通知	591
23.7 通过文件描述符通知的定时器: timerfd API	591
23.8 小结	591
23.9 习题	591
第 24 章 进程创建	592
24.1 fork(), exit(), wait(), execve() 概述	592
24.2 创建新进程: fork()	592

24.2.1 父子进程文件共享	592
24.2.2 fork()的内存语义.....	592
24.3 vfork()系统调用	592
24.4 fork()之后的竞争条件.....	592
24.5 通过信号同步来避免竞争条件	592
24.6 小结	592
24.7 习题	592
第 25 章 进程终止	593
25.1 终止进程: _exit()和 exit().....	593
25.2 进程终止的细节	593
25.3 Exit 处理器.....	593
25.4 fork()、stdio 缓冲区、_exit()之间的关联	593
25.5 小结	593
25.6 习题	593
第 26 章 监控子进程	594
26.1 等待子进程	594
26.1.1 wait()系统调用	594
26.1.2 waitpid()系统调用	594
26.1.3 等待的状态值	594
26.1.4 进程从信号处理器终止	594
26.1.5 waitid()系统调用	594
26.1.6 wait3()和 wait4()系统调用	594
26.2 孤儿进程（Orphan）和僵尸进程（Zombie）	594
26.3 SIGCHLD 信号	594
26.3.1 创建 SIGCHLD 信号处理器	594
26.3.2 子进程停止时递送 SIGCHLD	595
26.3.3 忽略死亡的子进程	595
26.4 小结	595

26.5 习题	595
第 27 章 程序执行	596
27.1 执行新程序: execve()	596
27.2 exec()库函数	596
27.2.1 PATH 环境变量	596
27.2.2 指定程序参数为列表	596
27.2.3 传递调用方环境给新程序	596
27.2.4 执行描述符引用的文件: fexecve()	596
27.3 解释器脚本	596
27.4 文件描述符和 exec()	596
27.5 信号和 exec()	596
27.6 执行 shell 命令: system()	596
27.7 实现 system()	596
27.8 小结	596
27.9 习题	596
第 28 章 进程创建和程序执行的更多细节	597
28.1 进程会计	597
28.2 clone()系统调用	597
28.2.1 clone()的 flags 参数	597
28.2.2 clone 子进程的 waitpid()扩展	597
28.3 进程创建的速度	597
28.4 exec()和 fork()对进程属性的影响	597
28.5 小结	597
28.6 习题	597
第 29 章 线程: 介绍	598
29.1 概述	598
29.2 pthread API 的背景细节	598
29.3 线程创建	598

29.4 线程终止	598
29.5 线程 ID	598
29.6 等待线程终止	598
29.7 分离线程	598
29.8 线程属性	598
29.9 线程 VS 进程	598
29.10 小结	598
29.11 习题	598
第 30 章 线程：同步	599
30.1 保护共享变量访问：Mutex	599
30.1.1 静态分配的 Mutex	599
30.1.2 Mutex 加锁和解锁	599
30.1.3 Mutex 的性能	599
30.1.4 Mutex 死锁	599
30.1.5 动态初始化 Mutex	599
30.1.6 Mutex 属性	599
30.1.7 Mutex 类型	599
30.2 状态变化通知：条件变量	599
30.2.1 静态分配的条件变量	599
30.2.2 通知和等待条件变量	599
30.2.3 测试条件变量的 Predicate	599
30.2.4 示例程序：等待任意线程终止	599
30.2.5 动态分配的条件变量	599
30.3 小结	600
30.4 习题	600
第 31 章 线程：线程安全和线程存储	601
31.1 线程安全（回顾可重入）	601
31.2 一次性初始化	601

31.3 线程特定数据	601
31.3.1 库函数中的线程特定数据	601
31.3.2 线程特定数据 API 概述	601
31.3.3 线程特定数据 API 细节	601
31.3.4 使用线程特定数据 API	601
31.3.5 线程特定数据的实现限制	601
31.4 线程本地存储	601
31.5 小结	601
31.6 习题	601
第 32 章 线程：线程取消	602
32.1 取消线程	602
32.2 取消状态和类型	602
32.3 取消点	602
32.4 测试线程取消	602
32.5 清理处理器	602
32.6 异步取消	602
32.7 小结	602
第 33 章 线程：更多细节	603
33.1 线程堆栈	603
33.2 线程和信号	603
33.2.1 UNIX 信号模型如何映射到线程	603
33.2.2 操作线程信号掩码	603
33.2.3 向线程发送信号	603
33.2.4 理智地处理异步信号	603
33.3 线程和进程控制	603
33.4 线程实现模型	603
33.5 POSIX 线程的 Linux 实现	603
33.5.1 LinuxThreads	603

33.5.2 NPTL	603
33.5.3 选择哪个线程实现？	603
33.6 pthread API 的高级特性	603
33.7 小结	603
33.8 习题	604
第 34 章 进程组、会话和作业控制	605
34.1 概述	605
34.2 进程组	605
34.3 会话	605
34.4 控制终端和控制进程	605
34.5 前台和后台进程组	605
34.6 SIGHUP 信号	605
34.6.1 shell 对 SIGHUP 的处理	605
34.6.2 SIGHUP 和控制进程的终止	605
34.7 作业控制	605
34.7.1 在 shell 中使用作业控制	605
34.7.2 实现作业控制	605
34.7.3 处理作业控制信号	605
34.7.4 孤儿进程组（回顾 SIGHUP）	605
34.8 小结	605
34.9 习题	606
第 35 章 进程优先级和调度	607
35.1 进程优先级（Nice 值）	607
35.2 实时进程调度概述	607
35.2.1 SCHED_RR 策略	607
35.2.2 SCHED_FIFO 策略	607
35.2.3 SCHED_BATCH 和 SCHED_IDLE 策略	607
35.3 实时进程调度 API	607

35.3.1 实时优先级范围	607
35.3.2 修改和获取策略和优先级	607
35.3.3 放弃 CPU	607
35.3.4 SCHED_RR 时间片	607
35.4 CPU 亲和力	607
35.5 小结	607
35.6 习题	607
第 36 章 进程资源	608
36.1 进程资源使用	608
36.2 进程资源限制	608
36.3 特定资源限制的细节	608
36.4 小结	608
36.5 习题	608
第 37 章 Daemon	609
37.1 概述	609
37.2 创建 Daemon	609
37.3 Daemon 编写指南	609
37.4 使用 SIGHUP 来重新初始化 Daemon	609
37.5 使用 syslog 记录日志和错误信息	609
37.5.1 概述	609
37.5.2 syslog API	609
37.5.3 /etc/syslog.conf 文件	609
37.6 小结	609
37.7 习题	609
第 38 章 编写安全的特权程序	610
38.1 是否需要设置用户 ID 和设置组 ID 的程序？	610
38.2 以最小权限执行操作	610
38.3 执行程序时要小心	610

38.4 避免暴露敏感信息	610
38.5 限制进程	610
38.6 小心信号和竞争条件	610
38.7 执行文件操作和文件 I/O 的陷阱	610
38.8 不要相信输入和环境	610
38.9 小心缓冲区溢出	610
38.10 小心拒绝服务攻击	610
38.11 检查返回状态和安全地失败	610
38.12 小结	610
38.13 习题	610
第 39 章 能力	611
39.1 能力的基本原理	611
39.2 Linux 能力	611
39.3 进程和文件能力	611
39.3.1 进程能力	611
39.3.2 文件能力	611
39.3.3 进程允许和有效能力集的作用	611
39.3.4 文件允许和有效能力集的作用	611
39.3.5 进程和文件可继承能力集的作用	611
39.3.6 shell 中查看和赋予文件能力	611
39.4 现代的能力实现	611
39.5 exec()时进程能力的转移	611
39.5.1 能力边界集	611
39.5.2 保留 root 语义	611
39.6 改变用户 ID 对进程能力的影响	611
39.7 编程改变进程能力	612
39.8 创建只有能力的环境	612
39.9 发现程序所需的能力	612

39.10 没有文件能力的老内核和系统	612
39.11 小结	612
39.12 习题	612
第 40 章 登录会计	613
40.1 utmp 和 wtmp 文件概述	613
40.2 utmpx API.....	613
40.3 utmpx 结构体	613
40.4 从 utmp 和 wtmp 文件中获取信息	613
40.5 获取登录名称: getlogin().....	613
40.6 为登录会话更新 utmp 和 wtmp 文件	613
40.7 lastlog 文件	613
40.8 小结	613
40.9 习题	613
第 41 章 共享库基础	614
41.1 对象库	614
41.2 静态库	614
41.3 共享库概述	614
41.4 创建和使用共享库 – A First Pass	614
41.4.1 创建共享库	614
41.4.2 位置无关的代码	614
41.4.3 使用共享库	614
41.4.4 共享库 Soname	614
41.5 使用共享库的有用工具	614
41.6 共享库版本和命名惯例	614
41.7 安装共享库	614
41.8 兼容 VS 不兼容库	614
41.9 升级共享库	614
41.10 在对象文件中指定库搜索目录	614

41.11 运行时查找共享库	615
41.12 运行时符号解析	615
41.13 使用静态库代替共享库	615
41.14 小结	615
41.15 习题	615
第 42 章 共享库高级特性	616
42.1 动态装载库	616
42.1.1 打开共享库: <code>dlopen()</code>	616
42.1.2 诊断错误: <code>dlerror()</code>	616
42.1.3 获取符号地址: <code>dlsym()</code>	616
42.1.4 关闭共享库: <code>dlclose()</code>	616
42.1.5 获取已装载符号的信息: <code>dladdr()</code>	616
42.1.6 在主程序中访问符号	616
42.2 控制符号可见性	616
42.3 链接器版本脚本	616
42.3.1 使用版本脚本控制符号可见性	616
42.3.2 符号版本	616
42.4 初始化和终止化函数	616
42.5 预装载共享库	616
42.6 监控动态链接器: <code>LD_DEBUG</code>	616
42.7 小结	617
42.8 习题	617
第 43 章 进程间通信概述	618
43.1 IPC 机制分类	618
43.2 通信机制	618
43.3 同步机制	618
43.4 IPC 机制对比	618
43.5 小结	618

43.6 习题	618
第 44 章 管道和 FIFO	619
44.1 概述	619
44.2 创建和使用管道	619
44.3 管道作为进程同步的方法	619
44.4 使用管道连接过滤器	619
44.5 使用管道与 shell 命令交互: <code>popen()</code>	619
44.6 管道和 <code>stdio</code> 缓冲	619
44.7 FIFO	619
44.8 使用 FIFO 的客户端-服务器应用	619
44.9 非阻塞 I/O	619
44.10 管道和 FIFO 的 <code>read()</code> 和 <code>write()</code> 语义	619
44.11 小结	619
44.12 习题	619
第 45 章 System V IPC 介绍	620
45.1 API 概述	620
45.2 IPC Key	620
45.3 相关的数据结构和对象权限	620
45.4 IPC 标识符和客户端-服务器应用	620
45.5 System V IPC 被调用时采用的算法	620
45.6 <code>ipcs</code> 和 <code>ipcrm</code> 命令	620
45.7 获取所有 IPC 对象列表	620
45.8 IPC 的限制	620
45.9 小结	620
45.10 习题	620
第 46 章 System V 消息队列	621
46.1 创建或打开消息队列	621
46.2 交换消息	621

46.2.1 发送消息	621
46.2.2 接收消息	621
46.3 消息队列控制操作	621
46.4 消息队列相关的数据结构	621
46.5 消息队列的限制	621
46.6 显示系统中所有消息队列	621
46.7 消息队列的客户端-服务器编程	621
46.8 使用消息队列的文件服务器应用	621
46.9 System V 消息队列的缺点	621
46.10 小结	621
46.11 习题	621
第 47 章 System V 信号量.....	622
47.1 概述	622
47.2 创建或打开信号量	622
47.3 信号量控制操作	622
47.4 信号量相关的数据结构	622
47.5 信号量初始化	622
47.6 信号量操作	622
47.7 处理多个阻塞的信号量操作	622
47.8 信号量撤消值	622
47.9 实现二进制信号量协议	622
47.10 信号量的限制	622
47.11 System V 信号量的缺点	622
47.12 小结	622
47.13 习题	622
第 48 章 System V 共享内存.....	623
48.1 概述	623
48.2 创建或打开共享内存段	623

48.3 使用共享内存	623
48.4 示例：通过共享内存传输数据	623
48.5 共享内存在虚拟内存中的位置	623
48.6 在共享内存中存储指针	623
48.7 共享内存控制操作	623
48.8 共享内存相关的数据结构	623
48.9 共享内存的限制	623
48.10 小结	623
48.11 习题	623
第 49 章 内存映射	624
49.1 概述	624
49.2 创建映射：mmap().....	624
49.3 解除映射区域：munmap()	624
49.4 文件映射	624
49.4.1 私有文件映射	624
49.4.2 共享文件映射	624
49.4.3 边界情况	624
49.4.4 内存保护和文件访问模式的相互作用	624
49.5 同步映射区域：msync().....	624
49.6 额外的 mmap()标志.....	624
49.7 匿名映射	624
49.8 重新映射区域：mremap()	624
49.9 MAP_NORESERVE 和交换空间过量使用	624
49.10 MAP_FIXED 标志	624
49.11 非线性映射：remap_file_pages()	625
49.12 小结	625
49.13 习题	625
第 50 章 虚拟内存操作	626

50.1 改变内存保护: mprotect()	626
50.2 内存锁: mlock()和 mlockall()	626
50.3 确定内存所在: mincore().....	626
50.4 建议未来的内存使用模式: madvise()	626
50.5 小结	626
50.6 习题	626
第 51 章 POSIX IPC 介绍	627
51.1 API 概述	627
51.2 比较 System V IPC 和 POSIX IPC.....	627
51.3 小结	627
第 52 章 POSIX 消息队列	628
52.1 概述	628
52.2 打开、关闭、删除消息队列	628
52.3 描述符和消息队列的关联	628
52.4 消息队列属性	628
52.5 交换消息	628
52.5.1 发送消息	628
52.5.2 接收消息	628
52.5.3 带超时的消息发送和接收	628
52.6 消息通知	628
52.6.1 通过信号接收通知	628
52.6.2 通过线程接收通知	628
52.7 Linux 特定的特性	628
52.8 消息队列的限制	628
52.9 比较 POSIX 和 System V 消息队列	628
52.10 小结	629
52.11 习题	629
第 53 章 POSIX 信号量	630

53.1 概述	630
53.2 命名信号量	630
53.2.1 打开命名信号量	630
53.2.2 关闭信号量	630
53.2.3 删除命名信号量	630
53.3 信号量操作	630
53.3.1 等待信号量	630
53.3.2 发布信号量	630
53.3.3 获取信号量的当前值	630
53.4 未命名信号量	630
53.4.1 初始化未命名信号量	630
53.4.2 销毁未命名信号量	630
53.5 与其它同步技术对比	630
53.6 信号量的限制	630
53.7 小结	631
53.8 习题	631
第 54 章 POSIX 共享内存	632
54.1 概述	632
54.2 创建共享内存对象	632
54.3 使用共享内存对象	632
54.4 删 除 共 享 内 存 对 象	632
54.5 各 种 共 享 内 存 API 的 对 比	632
54.6 小结	632
54.7 习题	632
第 55 章 文件锁	633
55.1 概述	633
55.2 flock()文件锁	633
55.2.1 锁继承和释放的语义	633

55.2.2 flock()的限制	633
55.3 fcntl()记录锁	633
55.3.1 死锁	633
55.3.2 示例：交互式的锁程序	633
55.3.3 示例：锁函数库	633
55.3.4 锁限制和性能	633
55.3.5 锁继承和释放的语义	633
55.3.6 锁饥饿和排队锁请求的优先级	633
55.4 强制锁	633
55.5 /proc/locks 文件	633
55.6 只运行程序的一个实例	633
55.7 老的锁技术	634
55.8 小结	634
55.9 习题	634
第 56 章 Sockets：介绍	635
56.1 概述	635
56.2 创建 Socket: socket()	635
56.3 绑定 Socket 到地址: bind()	635
56.4 通用 Socket 地址结构体: struct sockaddr	635
56.5 流 Socket	635
56.5.1 监听传入的连接: listen()	635
56.5.2 接受连接: accept()	635
56.5.3 连接到端 Socket: connect()	635
56.5.4 流 Socket I/O	635
56.5.5 终止连接: close()	635
56.6 数据报 Socket	636
56.6.1 交换数据报: recvfrom()和 sendto()	636
56.6.2 对数据报 Socket 使用 connect()	636

56.7 小结	636
第 57 章 Sockets: UNIX Domain	637
57.1 UNIX Domain Socket 地址: struct sockaddr_un	637
57.2 UNIX Domain 中的流 Socket	637
57.3 UNIX Domain 中的数据报 Socket	637
57.4 UNIX Domain Socket 权限	637
57.5 创建一对互连的 Socket: socketpair().....	637
57.6 Linux 抽象 Socket 命名空间	637
57.7 小结	637
57.8 习题	637
第 58 章 Sockets: TCP/IP 网络基础.....	638
58.1 因特网	638
58.2 网络协议和分层	638
58.3 数据链路层	638
58.4 网络层: IP	638
58.5 IP 地址	638
58.6 传输层	638
58.6.1 端口号	638
58.6.2 用户数据报协议 (UDP)	638
58.6.3 传输控制协议 (TCP)	638
58.7 Requests For Comments (RFC)	638
58.8 小结	638
第 59 章 Sockets: Internet Domain	639
59.1 Internet Domain Socket	639
59.2 网络字节序	639
59.3 数据表示	639
59.4 Internet Socket 地址	639
59.5 主机和服务转换函数概述	639

59.6 inet_pton()和 inet_ntop()函数	639
59.7 客户端-服务器例子（数据报 Socket）	639
59.8 域名系统（DNS）	639
59.9 /etc/services 文件	639
59.10 协议无关的主机和服务转换	639
59.10.1 getaddrinfo()函数	639
59.10.2 释放 addrinfo 列表: freeaddrinfo()	639
59.10.3 诊断错误: gai_strerror()	639
59.10.4 getnameinfo()函数	639
59.11 客户端-服务器例子（流 Socket）	640
59.12 一个 Internet Domain Socket 库	640
59.13 已废弃的主机和服务转换 API	640
59.13.1 inet_aton()和 inet_ntoa()函数	640
59.13.2 gethostbyname()和 gethostbyaddr()函数	640
59.13.3 getservbyname()和 getservbyport()函数	640
59.14 UNIX vs Internet Domain Socket	640
59.15 更多信息	640
59.16 小结	640
59.17 习题	640
第 60 章 Sockets: 服务器设计	641
60.1 Iterative 和并发服务器	641
60.2 Iterative UDP echo 服务器	641
60.3 并发 TCP echo 服务器	641
60.4 其它并发服务器设计	641
60.5 inetd (Internet Superserver) Daemon	641
60.6 小结	641
60.7 习题	641
第 61 章 Sockets: 高级主题	642

61.1 流 Socket 的部分读取和写入.....	642
61.2 shutdown()系统调用	642
61.3 Socket 特定的 I/O 系统调用: recv()和 send()	642
61.4 sendfile()系统调用	642
61.5 获取 Socket 地址.....	642
61.6 深入 TCP	642
61.6.1 TCP 段的格式	642
61.6.2 TCP 序号和确认	642
61.6.3 TCP 状态机和状态转换图	642
61.6.4 TCP 连接建立	642
61.6.5 TCP 连接终止	642
61.6.6 对 TCP Socket 调用 shutdown()	642
61.6.7 TIME_WAIT 状态.....	642
61.7 监控 Socket: netstat	642
61.8 使用 tcpdump 来监控 TCP 流量.....	643
61.9 Socket 选项	643
61.10 SO_REUSEADDR Socket 选项.....	643
61.11 多个 accept()的标志和选项继承	643
61.12 TCP vs UDP	643
61.13 高级特性	643
61.13.1 带外 (Out-of-Band) 数据	643
61.13.2 sendmsg()和 recvmsg()系统调用	643
61.13.3 传递文件描述符	643
61.13.4 获取发送方凭证	643
61.13.5 顺序包 Socket.....	643
61.13.6 SCTP 和 DCCP 传输层协议.....	643
61.14 小结	643
61.15 习题	643

第 62 章 终端	644
62.1 概述	644
62.2 获取和修改终端属性	644
62.3 stty 命令	644
62.4 终端特殊字符	644
62.5 终端标志	644
62.6 终端 I/O 模式	644
62.6.1 Canonical 模式	644
62.6.2 非 Canonical 模式	644
62.6.3 Cooked, Cbreak, Raw 模式	644
62.7 终端行速 (Bit Rate)	644
62.8 终端行控制	644
62.9 终端窗口大小	644
62.10 终端标识	644
62.11 小结	644
62.12 习题	645
第 63 章 其它 I/O 模型	646
63.1 概述	646
63.1.1 Level 触发和 Edge 触发通知	646
63.1.2 I/O 模型结合非阻塞 I/O	646
63.2 I/O 多路复用	646
63.2.1 select() 系统调用	646
63.2.2 poll() 系统调用	646
63.2.3 什么时候文件描述符准备好？	646
63.2.4 比较 select() 和 poll()	646
63.2.5 select() 和 poll() 的问题	646
63.3 信号驱动 I/O	646
63.3.1 什么时候信号通知“I/O 可用”？	646

63.3.2 改进信号驱动 I/O 的使用	646
63.4 epoll API	646
63.4.1 创建 epoll 实例: epoll_create().....	646
63.4.2 修改 epoll 兴趣列表: epoll_ctl()	647
63.4.3 等待事件: epoll_wait()	647
63.4.4 深入 epoll 语义	647
63.4.5 epoll 及 I/O 复用的性能对比	647
63.4.6 Edge 触发通知.....	647
63.5 等待信号和文件描述符	647
63.5.1 pselect()系统调用	647
63.5.2 Self-Pipe 技巧	647
63.6 小结	647
63.7 习题	647
第 64 章 伪终端	648
64.1 概述	648
64.2 UNIX 98 伪终端	648
64.2.1 打开未使用 Master: posix_openpt()	648
64.2.2 修改 Slave 所有权和权限: grantpt().....	648
64.2.3 解锁 Slave: unlockpt()	648
64.2.4 获取 Slave 的名字: ptsname()	648
64.3 打开 Master: ptyMasterOpen()	648
64.4 连接进程到伪终端: ptyFork().....	648
64.5 伪终端 I/O	648
64.6 实现 script(1)	648
64.7 终端属性和窗口大小	648
64.8 BSD 伪终端	648
64.9 小结	648
64.10 习题	648

附录 A: 跟踪系统调用	650
附录 B: 解析命令行参数	651
附录 C: 转换 NULL 指针	652
附录 D: 内核配置	653
附录 E: 更多信息来源.....	654
附录 F: 部分习题解答.....	655
参考书目.....	656
索引.....	657

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

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、作业控制（第 34 章）；进程优先级和调度（第 35 章）；进程资源（第 36 章）；daemon（第 37 章）；编写安全的特权程序（第 38 章）；能力（第 39 章）；登录会计（第 40 章）；共享库（第 41 章到第 42 章）。
6. 进程间通信（IPC）：IPC 简介（第 43 章）；管道和 FIFO（第 44 章）；System V IPC——消息队列、信号量、共享内存（第 45 章到第 48 章）；内存映射（第 49 章）；虚拟内存操作（第 50 章）；POSIX IPC——消息队列、信号量、共享内存（第 51 章到第 54 章）；文件锁（第 55 章）。
7. Socket 和网络编程：IPC 和 socket 网络编程（第 56 章到第 61 章）。
8. 高级 I/O 主题：终端（第 62 章）；可选 I/O 模型（第 63 章）；伪终端（第 64 章）。

例子程序

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

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

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

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

习题

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

标准和可移植性

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

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

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

Linux 内核和 C 库版本

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

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

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

其它语言使用编程接口

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

关于作者

（略）

致谢

（略）

许可

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

网站和例子程序源代码

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

反馈

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

Michael Timothy Kerrisk

Munich, Germany and Christchurch, New Zealand

August 2010

mtk@man7.org

第 1 章 历史和标准

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

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

1.1 UNIX 和 C 简史

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

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

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

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

UNIX 第一至第六版

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

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

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

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

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

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

BSD 和 System V 的诞生

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

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

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

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

1993 年的最终发布版：4.4BSD。

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

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

1.2 Linux 简史

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

1.2.1 GNU 项目

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

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

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

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

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

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

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

1.2.2 Linux 内核

Linus Torvalds 在 1991 年还是芬兰赫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，Torvalds 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\(128.214.6.100\)/pub/OS/Linux](http://nic.funet.fi(128.214.6.100)/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。

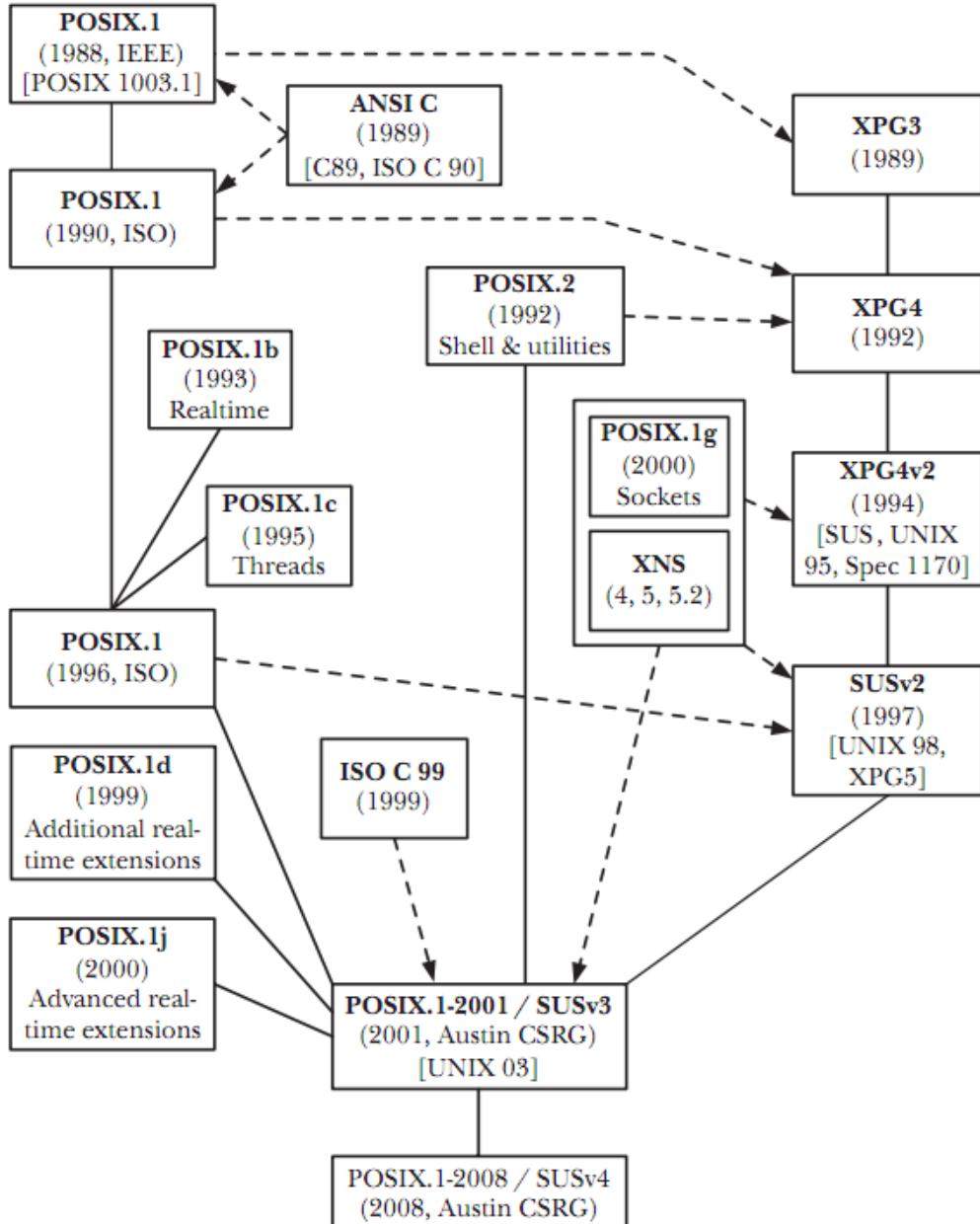


图 1-1：各种 UNIX 和 C 标准之间的关系

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。
- 狹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年来变得非常庞大，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、作业控制、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：

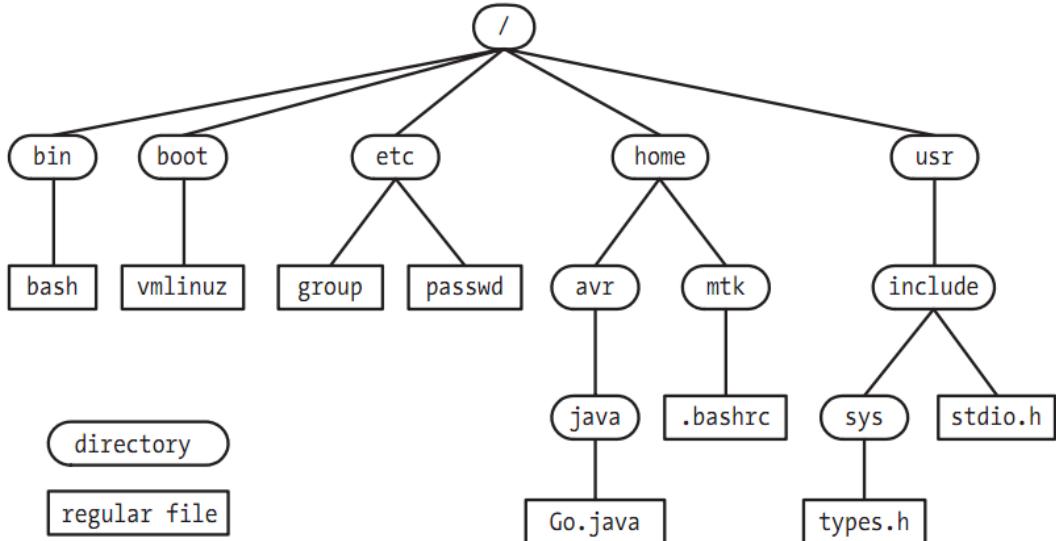


图 2-1: Linux 单一目录层次的子集

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

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 作业控制

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、创建进程间通信的管道等等(`syscalls(2)`手册页列出了 Linux 的所有系统调用)。

在描述系统调用工作的细节之前，我们先看一些基本要点：

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

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

1. 应用程序通过调用 C 库的包装函数发起系统调用。

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

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

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

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

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

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

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

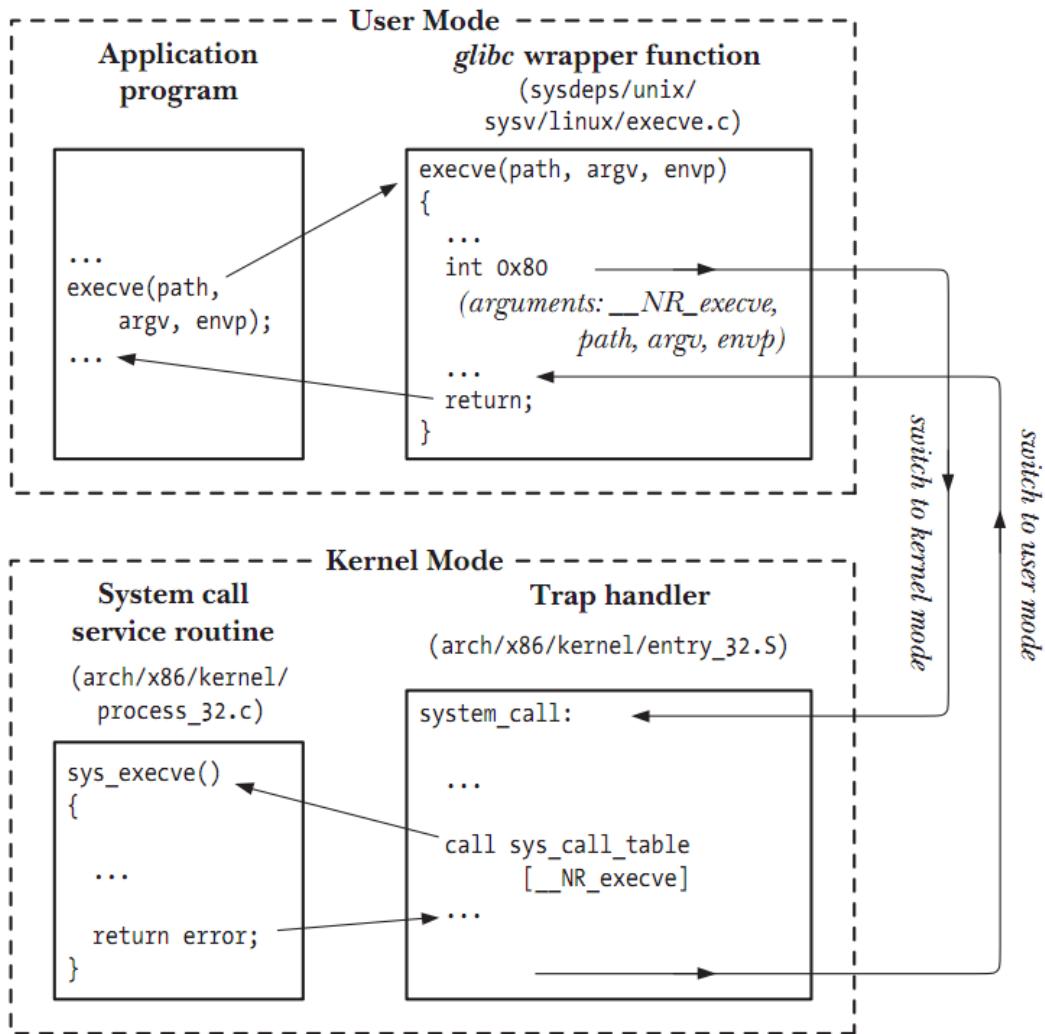


图 3-1：执行系统调用的步骤

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

3.2 库函数

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

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

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

3.3 标准 C 库；GNU C 库（glibc）

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

确定系统中 glibc 的版本

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

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

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

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

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

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

```
const char *gnu_get_libc_version(void);
```

返回一个 `null` 结尾的静态字符串，包含 GNU C 库版本号

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

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

3.4 系统调用和库函数的错误处理

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

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

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

处理系统调用错误

每个系统调用的手册页都注明了可能的返回值，并显示哪些返回值表示错误。

通常-1 表示错误，因此可以用以下代码检查系统调用的返回值：

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

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

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

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

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

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

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

```
#include <stdio.h>

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

处理系统调用失败的最简单方法如下：

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

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

```
#include <string.h>

char *strerror(int errnum);
```

返回对应于 `errnum` 的错误字符串

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

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

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

处理库函数错误

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

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

用的错误处理一样。

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

3.5 本书示例程序的说明

在这一节中，我们描述本书示例程序经常采用的惯例和特性。

3.5.1 命令行选项和参数

本书的许多示例程序依赖于命令行选项和参数，来确定自己的行为。

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

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

3.5.2 常用函数和头文件

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

常用头文件

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

算最小和最大值的宏。使用这个头文件，我们的示例程序会更加短小简洁。

清单 3-1：多数示例程序使用的头文件

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

错误诊断函数

为了简化示例程序的错误处理，我们使用错误诊断函数，如清单 3-2 所示：

清单 3-2：常用错误处理函数声明

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

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

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

```
#include "tlpi_hdr.h"

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

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

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

`err_exit()` 函数类似于 `errExit()`，但有以下两个区别：

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

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

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

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

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

我们可以使用下面代码来诊断 POSIX 线程函数：

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

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

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

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

```
#include "tlpi_hdr.h"

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

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

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

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

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

我们的错误诊断函数实现如清单 3-3 所示:

清单 3-3: 所有示例程序使用的错误处理函数

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

#include <stdarg.h>
#include "error_functions.h"
#include "tlpi_hdr.h"
#include "ename.c.inc"           /* Defines ename and MAX_ENAME */
#ifndef __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
    sprintf(errText, BUF_SIZE, ":");

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

	是数组类型	
<code>msglen_t</code>	无符号整数	System V 消息队列允许的字节数 (46.4 节)
<code>msgqnum_t</code>	无符号整数	System V 消息队列消息计数 (46.4 节)
<code>nfds_t</code>	无符号整数	<code>poll()</code> 的文件描述符数量 (63.2.2 节)
<code>nlink_t</code>	整数	某个文件的硬链接计数 (15.1 节)
<code>off_t</code>	带符号整数	文件偏移量或大小 (4.7 和 15.1 节)
<code>pid_t</code>	带符号整数	进程 ID, 进程组 ID, 会话 ID (6.2 、 34.2 、 34.3 节)
<code>ptrdiff_t</code>	带符号整数	两个指针的差
<code>rlim_t</code>	无符号整数	资源限制 (36.2 节)
<code>sa_family_t</code>	无符号整数	socket 地址家族 (56.4 节)
<code>shmatt_t</code>	无符号整数	System V 共享内存段的连接进程计数 (48.8 节)
<code>sig_atomic_t</code>	整数	可以被原子访问的数据类型 (21.1.3 节)
<code>siginfo_t</code>	结构体类型	信号来源相关的信息 (21.4 节)
<code>sigset_t</code>	整数或结构体类型	信号集 (20.9 节)
<code>size_t</code>	无符号整数	对象的字节大小
<code>socklen_t</code>	至少 32 位整数类型	socket 地址结构体的字节大小 (56.3 节)
<code>speed_t</code>	无符号整数	终端行速度 (62.7 节)
<code>ssize_t</code>	带符号整数	字节大小或指示错误
<code>stack_t</code>	结构体类型	可选信号堆栈的描述 (21.3 节)
<code>suseconds_t</code>	带符号整数, 允许范围 [-1, 1000000]	微秒的时间间隔 (10.1 节)
<code>tcflag_t</code>	无符号整数	终端模式标志位掩码 (62.2 节)
<code>time_t</code>	整数或浮点	日历时间, 从 Epoch 开始的秒数 (10.1 节)
<code>timer_t</code>	算术类型	POSIX.1b 时间函数定时器标识符 (23.6 节)
<code>uid_t</code>	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
<code>O_RDONLY</code>	打开文件只能读取
<code>O_WRONLY</code>	打开文件只能写入
<code>O_RDWR</code>	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?
<code>O_RDONLY</code>	打开为只能读取	v3
<code>O_WRONLY</code>	打开为只能写入	v3
<code>O_RDWR</code>	打开为同时读写	v3
<code>O_CLOEXEC</code>	设置“close-on-exec”标志（Linux 2.6.23 开始）	v4
<code>O_CREAT</code>	如果文件不存在则创建	v3
<code>O_DIRECT</code>	文件 I/O 绕过缓冲区缓存	
<code>O_DIRECTORY</code>	如果 <code>pathname</code> 不是目录则失败	v4
<code>O_EXCL</code>	和 <code>O_CREAT</code> 一起使用：创建唯一的文件	v3
<code>O_LARGEFILE</code>	在 32 位系统中打开大文件	
<code>O_NOATIME</code>	<code>read()</code> 时不更新文件的最后访问时间（Linux 2.6.8 开始）	
<code>O_NOCTTY</code>	不让 <code>pathname</code> 成为控制终端	v3
<code>O_NOFOLLOW</code>	不解引用符号链接	v4
<code>O_TRUNC</code>	截断现有文件为 0 长度	v3
<code>O_APPEND</code>	写入总是追加至文件末尾	v3
<code>O_ASYNC</code>	I/O 可操作时产生一个信号	
<code>O_DSYNC</code>	提供同步的 I/O 数据完整性（Linux 2.6.33 开始）	v3
<code>O_NONBLOCK</code>	非阻塞模式打开	v3
<code>O_SYNC</code>	使文件写入同步	v3

表 4-3 中的常量可以分为以下几组：

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

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

`flags` 常量的细节如下：

O_APPEND

写入总是追加到文件末尾。我们在 [5.1 节](#) 讨论这个标志的重要性。

O_ASYNC

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

O_CLOEXEC (Linux 2.6.23 开始)

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

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

O_CREAT

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

O_DIRECT

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

O_DIRECTORY

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

O_DSYNC (Linux 2.6.33 开始)

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

O_EXCL

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

从而解决了这个安全问题。

O_LARGEFILE

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

O_NOATIME (Linux 2.6.8 开始)

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

O_NOCTTY

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

O_NOFOLLOW

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

O_NONBLOCK

以非阻塞模式打开文件，请参考 [5.9 节](#)。

O_SYNC

以同步 I/O 模式打开文件。参考 [13.3 节](#) 内核 I/O 缓冲的相关讨论。

O_TRUNC

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

4.3.2 open() 的错误

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

EACCES

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

EISDIR

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

EMFILE

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

ENFILE

已经达到系统级的打开文件描述符数量限制。

ENOENT

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

EROFS

指定的文件在只读文件系统中，调用方却试图打开来写入。

ETXTBSY

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

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

4.3.3 `creat()` 系统调用

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

```
#include <fcntl.h>

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

返回文件描述符；出错返回-1

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

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

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

4.4 读取文件：read()

read()系统调用从描述符 fd 引用的文件中读取数据：

```
#include <unistd.h>

ssize_t read(int fd, void *buffer, size_t count);
```

返回读取的字节数，EOF 返回 0，错误时返回-1

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

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

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

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

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

使用 read()从终端获取一系列输入字符，我们可以使用如下代码：

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

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

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

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

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

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

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

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

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

4.5 写入文件: `write()`

`write()` 系统调用向打开文件写入数据。

```
#include <unistd.h>

ssize_t write(int fd, void *buffer, size_t count);
```

返回实际写入的字节数，出错时返回-1

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

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

4.6 关闭文件：close()

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

```
#include <unistd.h>

int close(int fd);
```

成功时返回 0；失败返回 -1

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

和其它系统调用一样，调用 close()也应该检查相关的错误，如下所示：

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

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

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

4.7 改变文件偏移：lseek()

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

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

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

`whence` 指定的位置。

```
#include <unistd.h>

off_t lseek(int fd, off_t offset, int whence);
```

成功时返回新的文件偏移，出错返回-1

`offset` 参数指定字节数（SUSv3 规定 `off_t` 数据类型为带符号整数类型）。

`whence` 参数指示 `offset` 的基点，可以取如下值之一：

`SEEK_SET`

文件偏移设为文件起始位置后的 `offset` 字节。

`SEEK_CUR`

文件偏移调整为相对当前文件偏移的 `offset` 字节。

`SEEK_END`

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

图 4-1 显示了 `whence` 参数是如何被解释的。

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

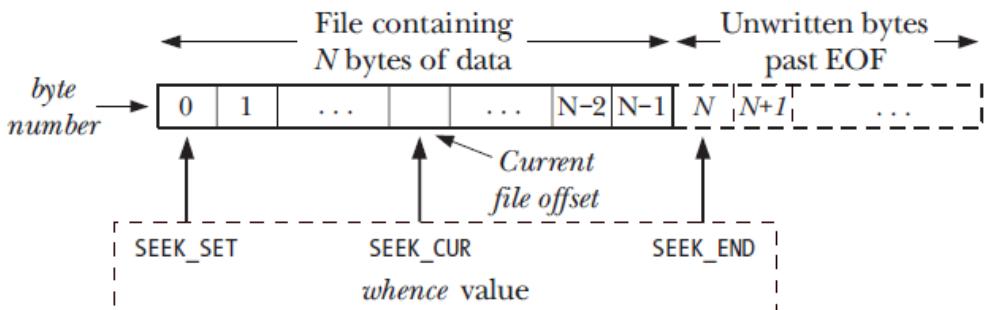


图 4-1：解释 `lseek()` 的 `whence` 参数

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

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

文件偏移：

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

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

下面是 lseek() 调用的一些例子，后面的注释描述了新的文件偏移：

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

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

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

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

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

文件空洞

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

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

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

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

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

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

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

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

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

示例程序

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

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

清单 4-3: *read()*, *write()*, *lseek()* 演示

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

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

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

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

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

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

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

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

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

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

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

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

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

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

default:
    cmdLineErr("Argument must start with [rRws]: %s\n", argv[ap]);
}
}

exit(EXIT_SUCCESS);
-----fileio/seek_io.c
```

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

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

4.8 通用 I/O 模型之外的操作：ioctl()

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

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

int ioctl(int fd, int request, ... /* argp */);
```

成功时返回 `request` 相关的值，出错返回-1

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

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

我们在后面章节会看到许多 `ioctl()` 的使用（比如 [15.5 节](#)）。

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

4.9 小结

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

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

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

`ioctl()` 系统调用提供标准文件 I/O 模型之外的设备和文件操作功能。

4.10 习题

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

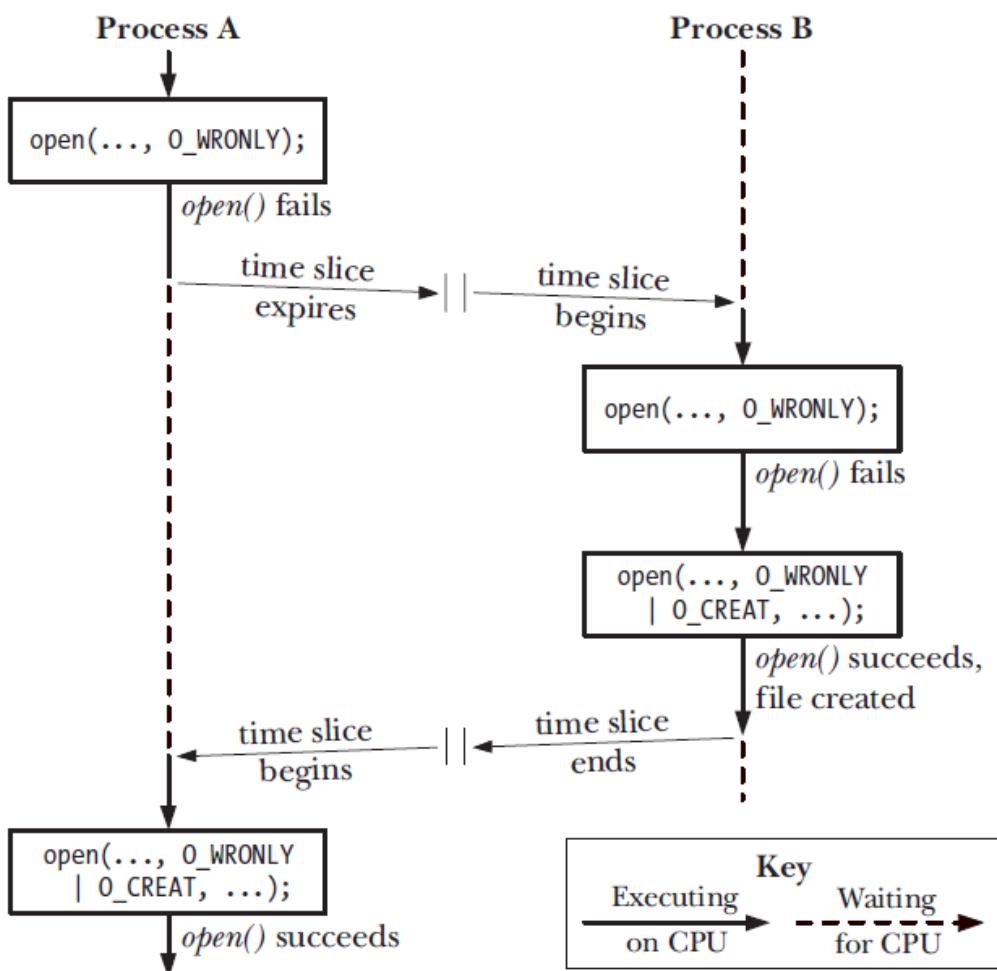


图 5-1：互斥创建文件失败

要演示上面代码存在的真正问题，我们可以替换清单 5-1 中已经注释掉的那一行 `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。

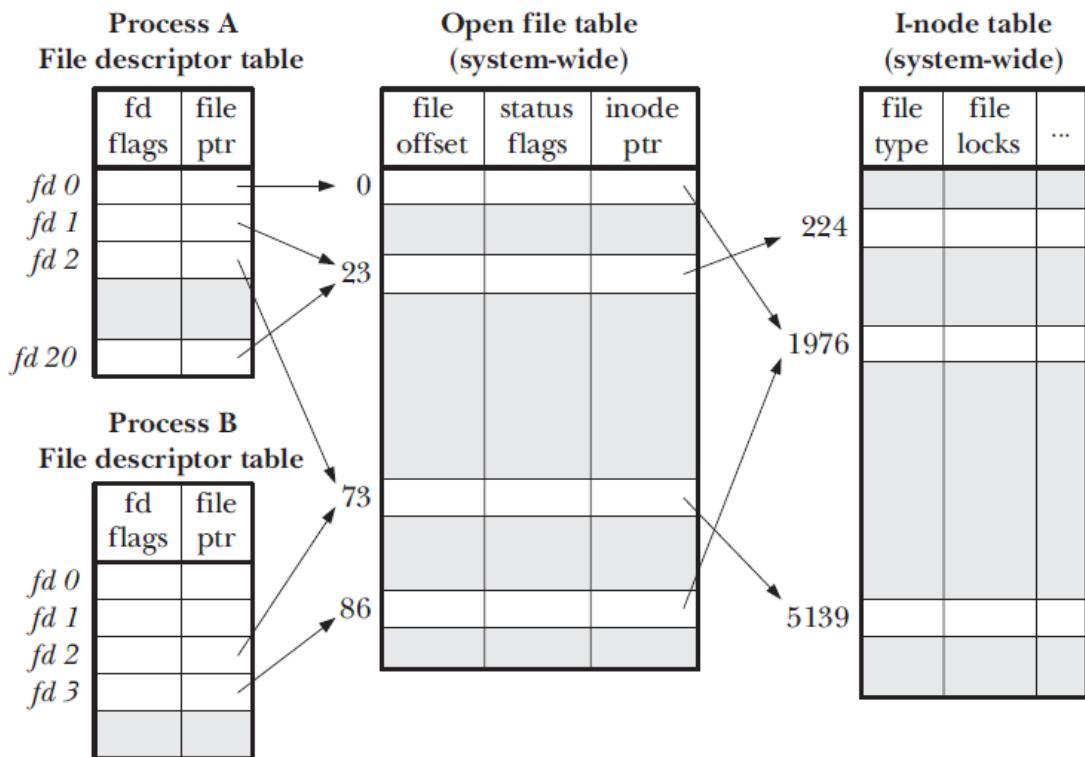


图 5-2: 文件描述符、打开文件、i-node 的关系

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()` 系统调用可以完成这个任务。

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

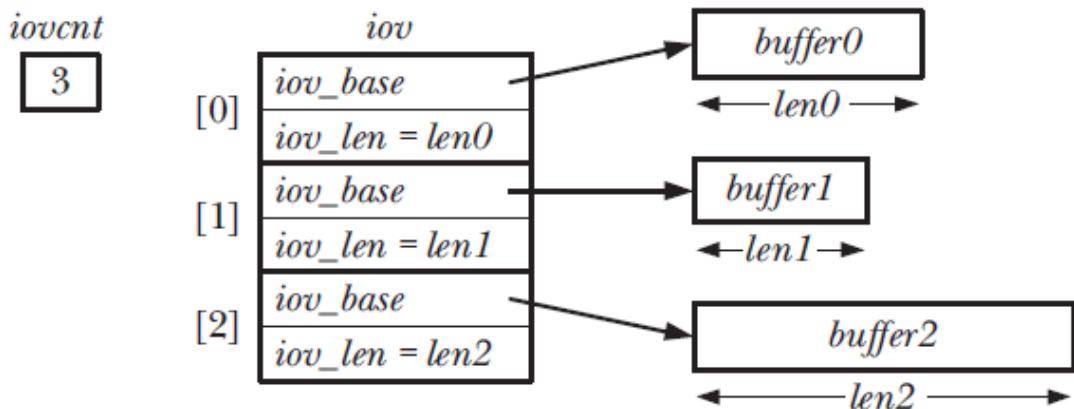


图 5-3: iovec 数组和缓冲区的关联系示例

Scatter 输入

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

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

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

清单 5-2 演示了 `readv()`的使用。

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

清单 5-2：使用 `readv()`执行 scatter 输入

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

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

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

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

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

    totRequired = 0;

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

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

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

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

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

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

```
(long) totRequired, (long) numRead);  
  
    exit(EXIT_SUCCESS);  
}  
-----fileio/t_readv.c
```

Gather 输出

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

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

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

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

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

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

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

指定偏移位置执行 scatter-gather I/O

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

但在现代 BSD 系统中也可用。

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

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

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

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

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

5.8 截断文件: `truncate()`和`ftruncate()`

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

```
#include <unistd.h>

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

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

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

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

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

5.9 非阻塞 I/O

打开文件时指定 `O_NONBLOCK` 标志有两个作用：

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

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

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

我们会在[第 44.9 节](#) 和第 63 章更详尽地讨论非阻塞 I/O。

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

5.10 大文件 I/O

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

文件大小限制为 $2^{31}-1$ 字节（也就是 2GB）。

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

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

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

我们可以按两种方式来编写支持 LFS 功能的应用：

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

过渡型 LFS API

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

要访问大文件，我们只需要使用这些函数的 64 位版本。例如，要打开一个

大文件，我们可以编写如下代码：

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

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

除了上面提到的这些函数，LFS API 还增加了一些新的数据类型，包括：

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

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

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

清单 5-3：访问大文件

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

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

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

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

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

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

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

exit(EXIT_SUCCESS);
}

-----fileio/large_file.c
```

FILE_OFFSET_BITS 宏

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

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

或者我们也可以在源代码包含任何头文件之前定义这个宏：

```
#define _FILE_OFFSET_BITS 64
```

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

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

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

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

传递 off_t 值给 printf()

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

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

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

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

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

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

5.11 /dev/fd 目录

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

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

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

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

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

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

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

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

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

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

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

5.12 创建临时文件

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

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

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

```
int mkstemp(char *template);
```

成功时返回文件描述符，出错返回-1

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

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

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

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

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

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

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

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

`tmpfile()` 函数创建一个唯一命名的临时文件，然后打开为读取和写入（这个文件以 `O_EXCL` 标志打开，确保其它进程不会创建相同名字的文件）。

```
#include <stdio.h>

FILE *tmpfile(void);
```

成功时返回文件指针，出错返回 `NULL`

成功时 `tmpfile()` 返回文件流，可以用于其它的 `stdio` 库函数。临时文件在关闭时将自动删除，`tmpfile()` 内部在打开文件后立即调用了 `unlink()`，来移除该文件。

5.13 小结

在本章的课程里，我们介绍了原子性的概念，以及其对某些系统调用正确操作至关重要的作用。特别是 `open()` 的 `O_EXCL` 标志允许调用方确保自己是文件的创建者，而 `open()` 的 `O_APPEND` 标志则确保多个进程向同一文件添加数据不会互相影响。

`fcntl()` 系统调用执行一系列文件控制操作，包括改变打开文件状态标志和复制文件描述符。复制文件描述符也可以通过 `dup()` 和 `dup2()` 来实现。

我们查看了文件描述符、打开文件、文件 i-node 之间的关联，并且强调这三个对象所包含的信息是不同的。复制文件描述符会引用同一个打开文件，因此共享打开文件状态标志和文件偏移。

我们描述了一些能够扩展 `read()` 和 `write()` 功能的系统调用。`pread()` 和 `pwrite()` 系统调用在指定文件位置执行 I/O 操作，并且不改变文件偏移。`readv()` 和 `writev()` 执行 scatter-gather I/O；`preadv()` 和 `pwritev()` 调用则组合了 scatter-gather I/O 和指定文件位置 I/O 的功能。

`truncate()` 和 `ftruncate()` 系统调用可以用来减少文件大小，并丢弃超出的字节；也可以用来增加文件大小，并以 0 字节或文件空洞进行填充。

我们简短地介绍了非阻塞 I/O 的概念，后面章节会更加详细地讨论。

LFS 规范定义了一组大文件 I/O 扩展，可允许 32 位系统执行超过 32 位系统限制的大文件 I/O 操作。

`/def/fd` 虚拟目录下的文件允许进程通过文件描述符数值来访问自己的打开文件，这在 shell 命令中特别有用。

`mkstemp()` 和 `tmpfile()` 函数允许应用创建临时文件。

5.14 习题

5-1. 修改清单 5-3 中的程序，使用标准文件 I/O 系统调用（`open()`和 `lseek()`）以及 `off_t` 数据类型。把 `_FILE_OFFSET_BITS` 宏设置为 64 然后编译程序，并测试该程序能够成功地创建大文件。

5-2. 编写一个程序，以 `O_APPEND` 标志打开一个现有文件，然后在每次写入数据之前 `seek` 到文件起始位置。写入文件中的数据会在哪里？为什么？

5-3. 这个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(olfd, F_GETFL)` 是否成功来实现), 如果 `olfd` 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、以及其他许多信息。

6.2 进程 ID 和父进程 ID

每个进程都有一个进程 ID (PID)，它是一个唯一标识系统中进程的正整数值。进程 ID 在一系列系统调用中使用和返回。例如 `kill()` 系统调用（[20.5 节](#)）允许调用方向指定 ID 的进程发送一个信号。如果我们需要针对特定进程创建唯一标识，进程 ID 也非常有用。一个常见的例子是使用进程 ID 作为进程唯一文件名。

`getpid()` 系统调用返回调用进程的进程 ID。

```
#include <unistd.h>

pid_t getpid(void);
```

总是成功返回调用进程 ID

SUSv3 规定 `pid_t` 数据类型为整数类型，用来存储进程 ID。

除了少数系统进程（如 `init` 的进程 ID 为 1），程序和进程 ID 之间没有必然联系（程序运行时的进程 ID 不会固定）。

Linux 内核限制进程 ID 小于等于 32767。当新进程创建时，会被顺序地赋予下一个可用的进程 ID。每当达到 32767 的限制，内核就重置进程 ID 计数器，因此进程 ID 又重新从最低整数值开始分配。

一旦达到 32767，进程 ID 计数器将重置为 300 而不是 1。这是因为许多低数值的进程 ID 已经被系统进程和 daemon 占用，搜索这个范围内未使用的进程 ID 纯粹是浪费时间。

在 Linux 2.4 及更早版本，进程 ID 的 32767 限制由内核常量 `PID_MAX` 定义。在 Linux 2.6 中又有了新的变化。进程 ID 默认的最大限制仍然是 32767，但这个限制现在可以通过 Linux 特定的 `/proc/sys/kernel/pid_max` 文件来修改。（这个文件的值是最

大进程 ID 的值加 1)。在 32 位平台中，这个文件的最大值是 32768；而在 64 位平台中，这个值最大可以调整为 2^{22} (大约 4 百万)，这几乎可以满足任意数量的进程要求。

每个进程都有一个父进程（也就是创建该进程的进程）。进程可以使用 `getppid()` 系统调用来获得父进程 ID。

```
#include <unistd.h>

pid_t getppid(void);
```

总是成功返回父进程 ID

实际上系统中所有进程的父进程属性组成了树形进程关系图。每个进程的父进程也有自己的父进程，并且依次递归最终回到进程 1 (`init`)，它是所有进程的祖先（可以使用 `pstree` 命令查看进程树）。

如果子进程由于父进程终止而变成“孤儿”进程，则子进程将自动由 `init` 进程收养，随后的 `getppid()` 调用将返回 1（参考 [26.2 节](#)）。

任何进程的父进程都可以通过 Linux 特定的 `/proc/PID/status` 文件中的 `Ppid` 域获得。

6.3 进程内存布局

每个进程分配的内存由许多部分组成，通常称为“段”。进程拥有如下段：

- `text` 段：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。

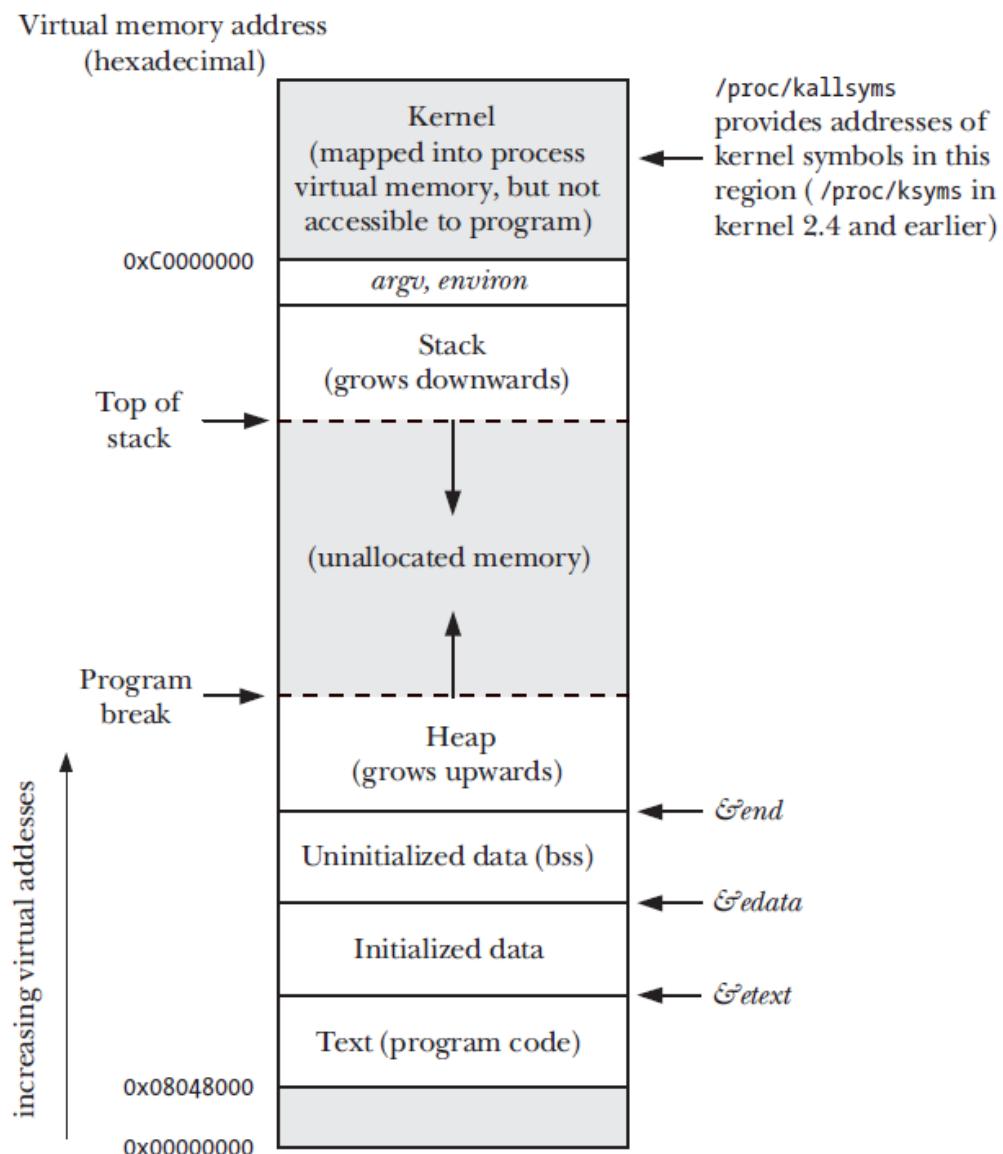


图 6-1: Linux/x86-32 典型的进程内存布局

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。

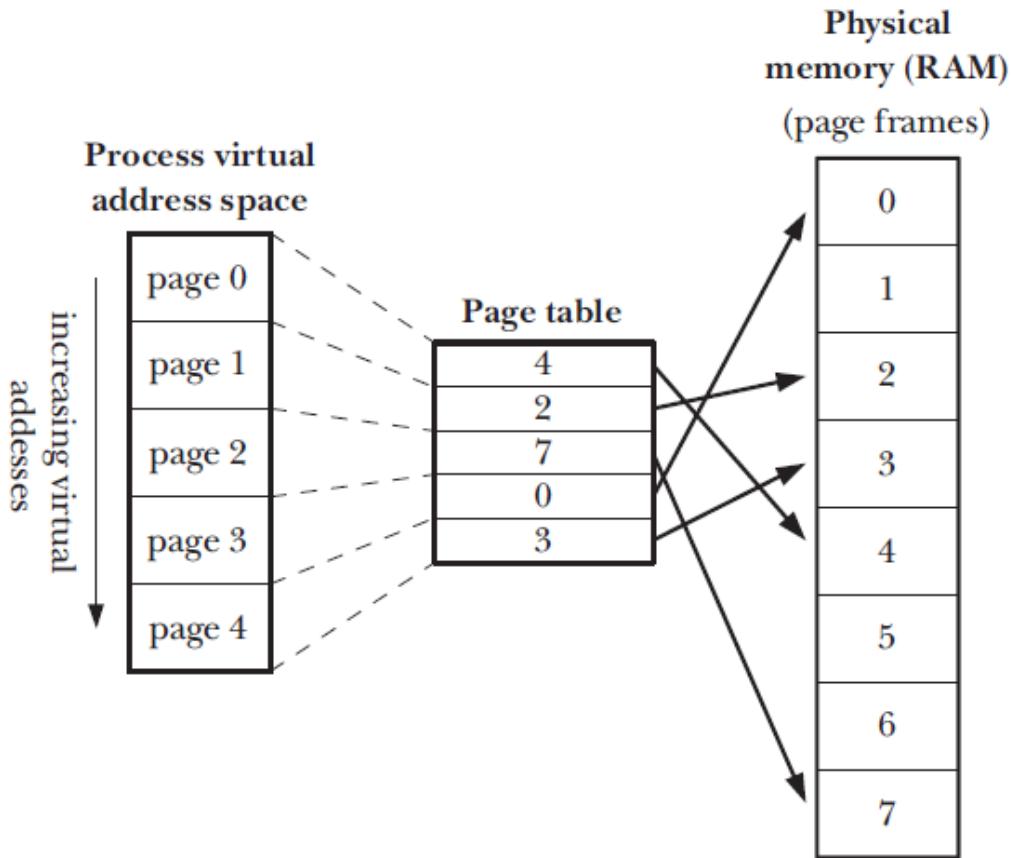


图 6-2：虚拟内存概观

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。内存共享有三种常见的场景：
 - 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。

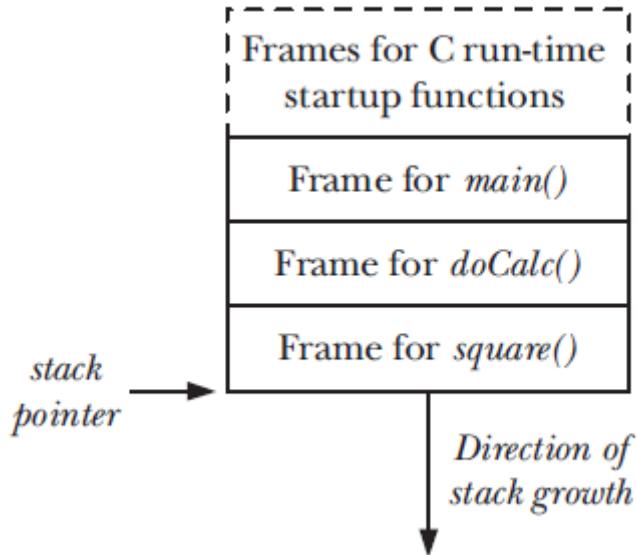


图 6-3: 进程堆栈示例

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`)。

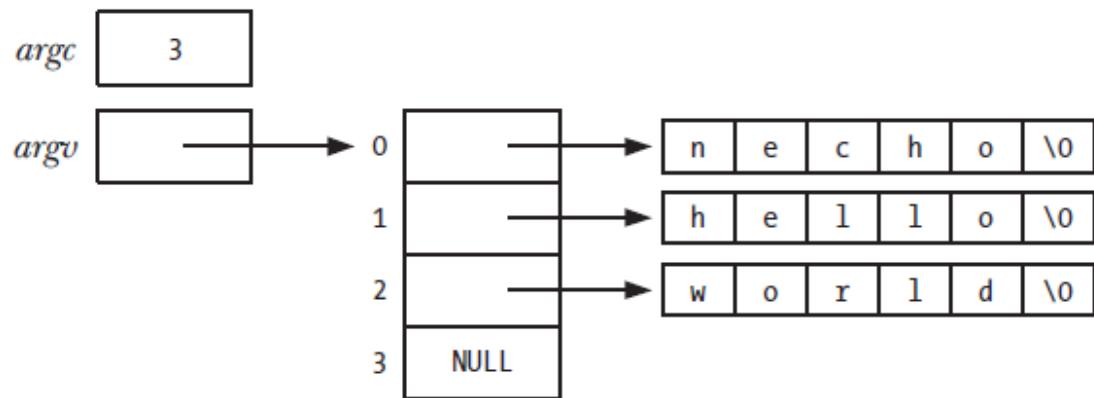


图 6-4: necho hello world 命令的 argc 和 argv 值

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。

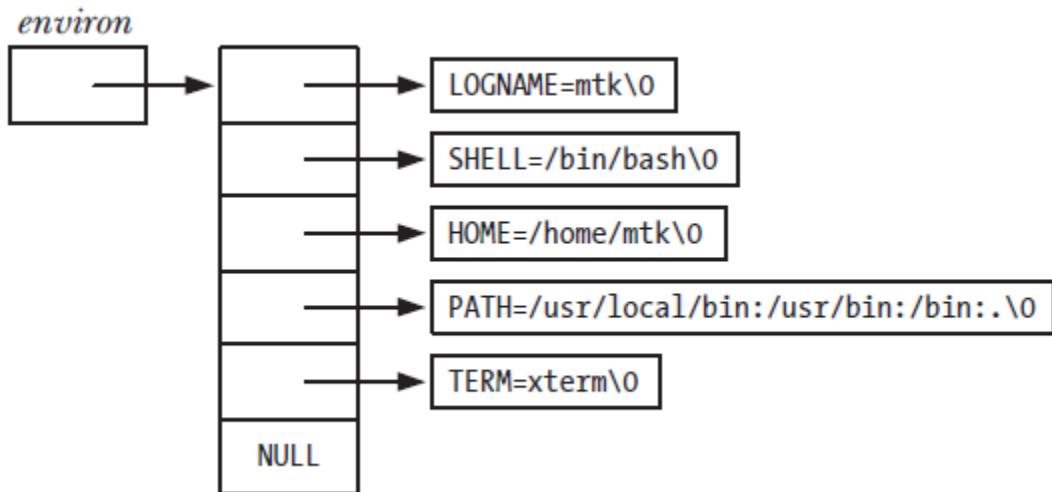


图 6-5：进程环境列表数据结构示例

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是，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。非局部语句表示 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
```

但是如果我们将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() 把剩余的内存在放到自由列表中。

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：

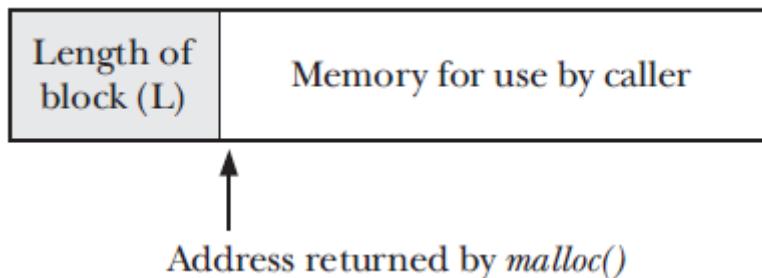


图 7-1：malloc() 返回的内存块

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：

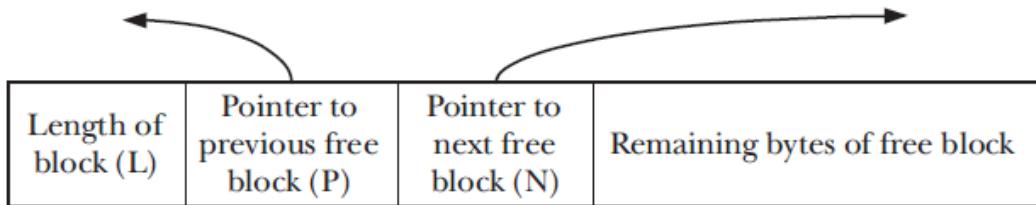


图 7-2：自由列表中的一个块

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：

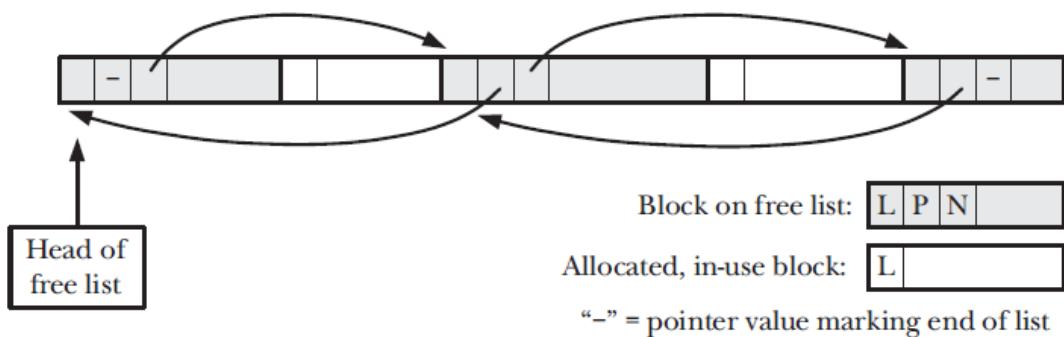


图 7-3：堆包含已分配块和自由列表

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 -fmccheck 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再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，这些内存再调用 `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），则密码域将不会被忽略。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。但是如果我们将使用了 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。

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

`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实际的 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。

2. 特权进程可以任意修改 ID。
3. 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:
 - a) 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)
 - b) 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 `setsuid()` 和 `setfsuid()` 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。

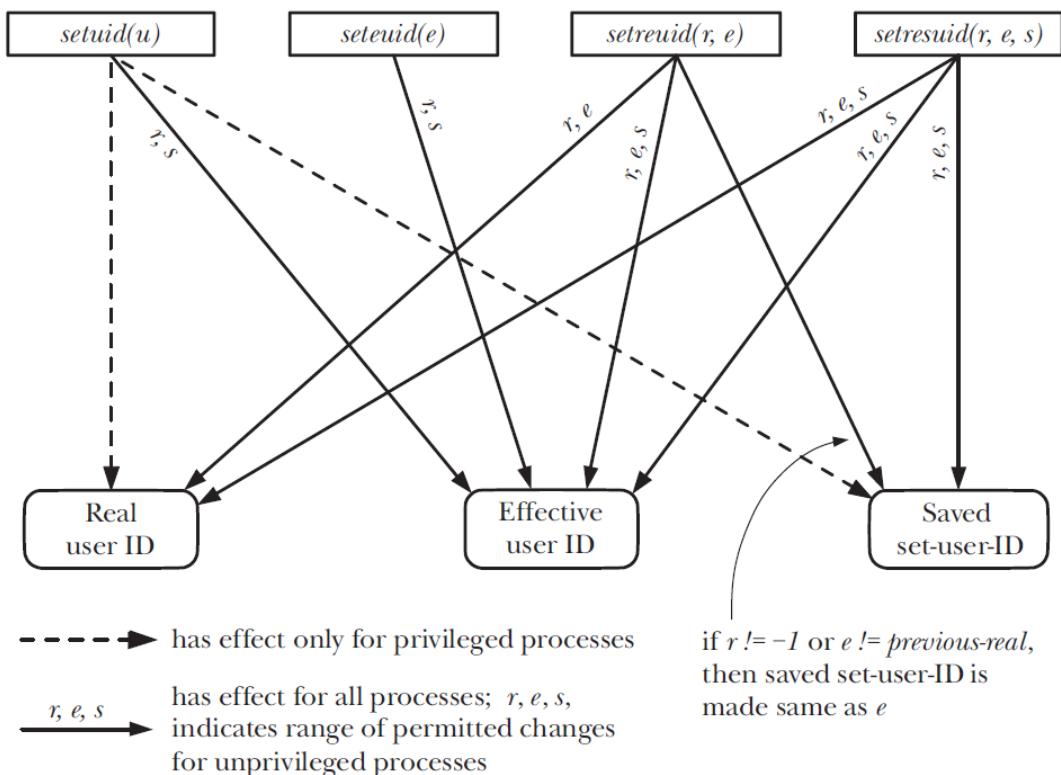


图 9-1：修改进程用户 ID 函数的作用

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)	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
setgid(g)			
seteuid(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
setegid(e)			

<code>setreuid(r, e)</code>	(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
<code>setresuid(r, e, s)</code> <code>setresgid(r, e, s)</code>	(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
<code>setsuid(u)</code> <code>setfsgid(u)</code>	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
<code>setgroups(n, l)</code>	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：

- a) 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）。
- b) 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)。

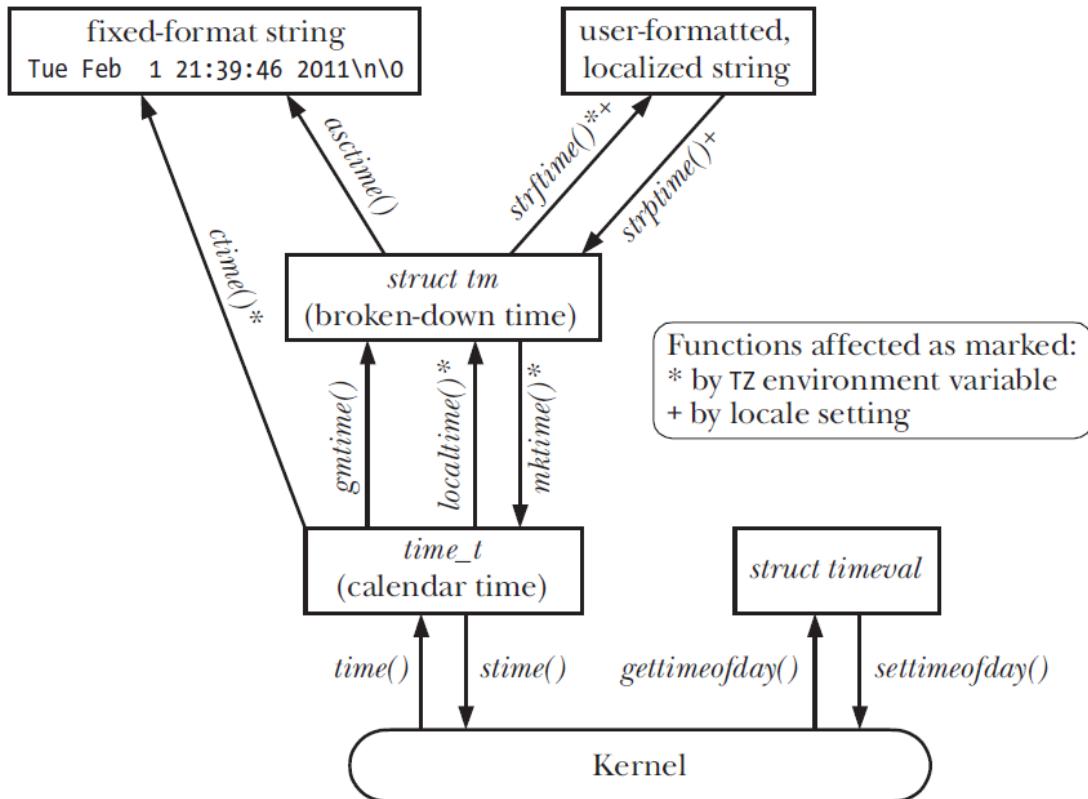


图 10-1: 获取和转换日历时间的函数

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), 用来计算闰秒, 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)。

`mktimes()` 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 mktimes(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
```

`mktimes()` 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, `mktimes()` 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, `mktimes()` 会调整该域的值, 适当地设置该域, 并调整其它域。`mktimes()` 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，当年的最后一周则是第 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

Specifier	Description	Example
%%	A % character	%
%a	Abbreviated weekday name	Tue
%A	Full weekday name	Tuesday
%b, %h	Abbreviated month name	Feb
%B	Full month name	February
%c	Date and time	Tue Feb 1 21:39:46 2011
%d	Day of month (2 digits, 01 to 31)	01
%D	American date (same as %m/%d/%y)	02/01/11
%e	Day of month (2 characters)	1
%F	ISO date (same as %Y-%m-%d)	2011-02-01
%H	Hour (24-hour clock, 2 digits)	21
%I	Hour (12-hour clock, 2 digits)	09
%j	Day of year (3 digits, 001 to 366)	032
%m	Decimal month (2 digits, 01 to 12)	02
%M	Minute (2 digits)	39
%p	AM/PM	PM
%P	am/pm (GNU extension)	pm
%R	24-hour time (same as %H:%M)	21:39
%S	Second (00 to 60)	46
%T	Time (same as %H:%M:%S)	21:39:46
%u	Weekday number (1 to 7, Monday = 1)	2
%U	Sunday week number (00 to 53)	05
%w	Weekday number (0 to 6, Sunday = 0)	2
%W	Monday week number (00 to 53)	05
%x	Date (localized)	02/01/11
%X	Time (localized)	21:39:46
%y	2-digit year	11
%Y	4-digit year	2011
%Z	Timezone name	CET

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：

```
$ ./strftime "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：

```
$ ./strftime "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 strftime.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/strftime.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
<code>LC_CTYPE</code>	这个文件包含字符分类（参考 <code>isalpha(3)</code> ）和大小写转换的规则
<code>LC_COLLATE</code>	这个文件包含字符集的校对规则
<code>LC_MONETARY</code>	这个文件包含货币数值（参考 <code>localeconv(3)</code> 和 <code><locale.h></code> ）的格式化规则
<code>LC_NUMERIC</code>	这个文件包含非货币数值（参考 <code>localeconv(3)</code> 和 <code><locale.h></code> ）的格式化规则
<code>LC_TIME</code>	这个文件包含日期和时间的格式化规则
<code>LC_MESSAGES</code>	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 II，排序紧随字母 I；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少量的值, 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 round robin 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间的时间，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/types.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、运行时可增减值。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下），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、作业控制、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 <code>sysconf()/pathconf()</code> 名	描述	标注
<code>_POSIX_ASYNCNROUNOUS_IO</code> (<code>_SC_ASYNCNROUNOUS_IO</code>)	异步 I/O	
<code>_POSIX_CHOWN_RESTRICTED</code> (<code>_PC_CHOWN_RESTRICTED</code>)	只有特权进程才能使用 <code>chown()</code> 和 <code>fchown()</code> 来改变文件的用户 ID 和组 ID 为任意值（ 15.3.2 节 ）	*
<code>_POSIX_JOB_CONTROL</code> (<code>_SC_JOB_CONTROL</code>)	作业控制（ 34.7 节 ）	+

<code>_POSIX_MESSAGE_PASSING (_SC_MESSAGE_PASSING)</code>	POSIX 消息队列 (52 章)	
<code>_POSIX_PRIORITY_SCHEDULING (_SC_PRIORITY_SCHEDULING)</code>	进程调度 (35.3 节)	
<code>_POSIX_REALTIME_SIGNALS (_SC_REALTIME_SIGNALS)</code>	实时信号扩展 (22.8 节)	
<code>_POSIX_SAVED_IDS (none)</code>	进程拥有设置用户 ID 和保存的设置组 ID (9.4 节)	+
<code>_POSIX_SEMAPHORES (_SC_SEMAPHORES)</code>	POSIX 信号量 (53 章)	
<code>_POSIX_SHARED_MEMORY_OBJECTS (_SC_SHARED_MEMORY_OBJECTS)</code>	POSIX 共享内存对象 (54 章)	
<code>_POSIX_THREADS (_SC_THREADS)</code>	POSIX 线程	
<code>_XOPEN_UNIX (_SC_XOPEN_UNIX)</code>	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, <code>getpid()</code>)
Pid: 1	实际上是线程 ID (<code>gettid()</code>)
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
FDSSize: 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: ffffffe5770d8fc	忽略的信号
SigCgt: 0000000280b2603	捕获的信号
CapInh: 0000000000000000	可中断 capabilities
CapPrm: 00000000ffffffffff	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。

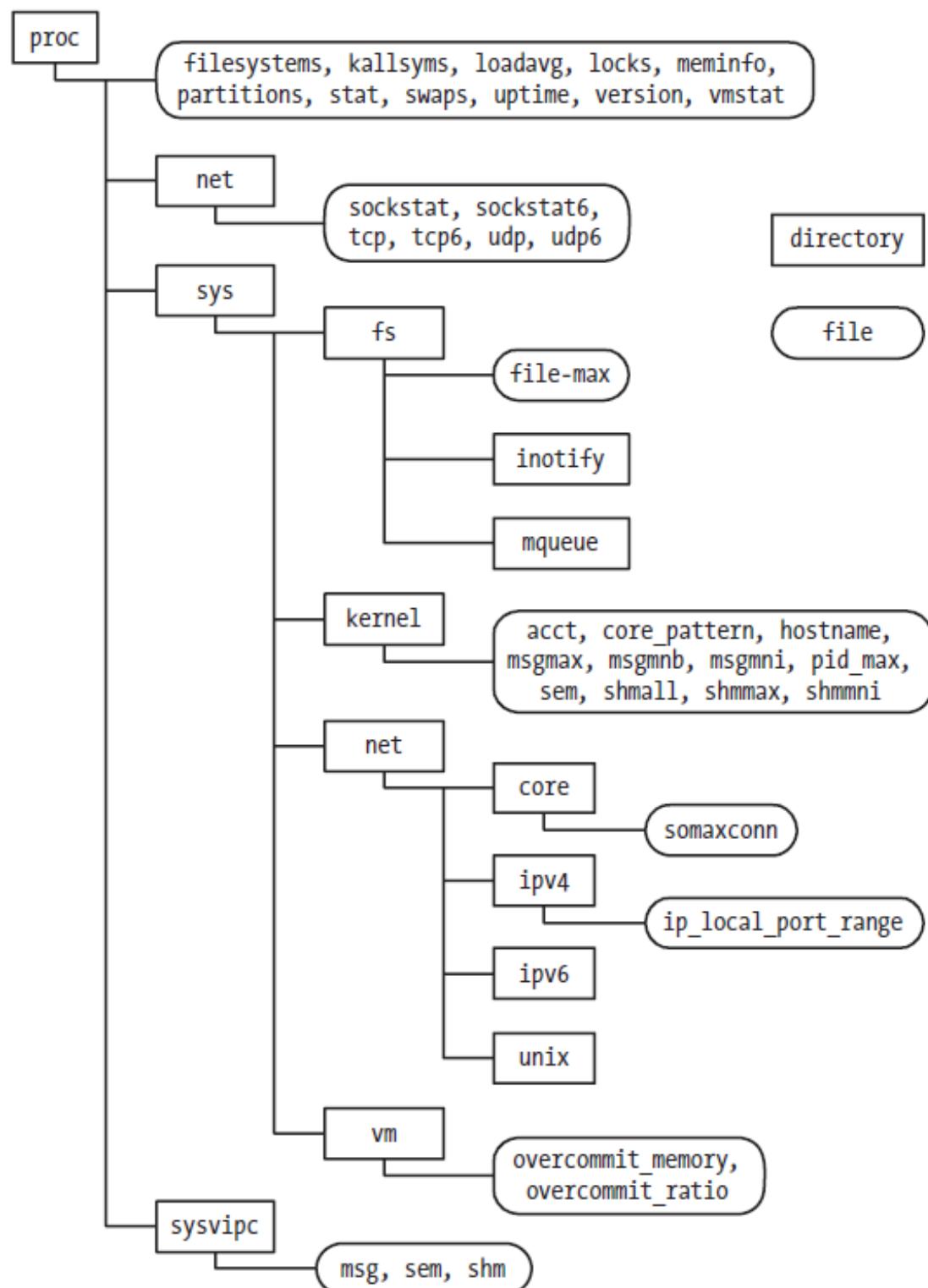


图 12-1: /proc 下的部分文件和子目录

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
    char domainname[_UTSNAME_LENGTH]; /* 下面 Linux 特定 */
    /* 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	107.43	107.32	8.20	99.12
2	54.16	53.89	4.13	49.76
4	31.72	30.96	2.30	28.66
8	15.59	14.34	1.08	13.26
16	7.50	7.14	0.51	6.63
32	3.76	3.68	0.26	3.41
64	2.19	2.04	0.13	1.91
128	2.16	1.59	0.11	1.48
256	2.06	1.75	0.10	1.65
512	2.06	1.03	0.05	0.98
1024	2.05	0.65	0.02	0.63
4096	2.05	0.38	0.01	0.38
16384	2.05	0.34	0.00	0.33
65536	2.06	0.32	0.00	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，已经是完全可以忽略的。

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调用，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

BUF_SIZE	Time (seconds)			
	Elapsed	Total CPU	User CPU	System CPU
1	72.13	72.11	5.00	67.11
2	36.19	36.17	2.47	33.70
4	20.01	19.99	1.26	18.73
8	9.35	9.32	0.62	8.70
16	4.70	4.68	0.31	4.37
32	2.39	2.39	0.16	2.23
64	1.24	1.24	0.07	1.16
128	0.67	0.67	0.04	0.63
256	0.38	0.38	0.02	0.36
512	0.24	0.24	0.01	0.23
1024	0.17	0.17	0.01	0.16
4096	0.11	0.11	0.00	0.11
16384	0.10	0.10	0.00	0.10
65536	0.09	0.09	0.00	0.09

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

BUF_SIZE	Time required (seconds)			
	Without <code>O_SYNC</code>		With <code>O_SYNC</code>	
	Elapsed	Total CPU	Elapsed	Total CPU
1	0.73	0.73	1030	98.8
16	0.05	0.05	65.0	0.40
256	0.02	0.02	4.07	0.03
4096	0.01	0.01	0.34	0.03

`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。

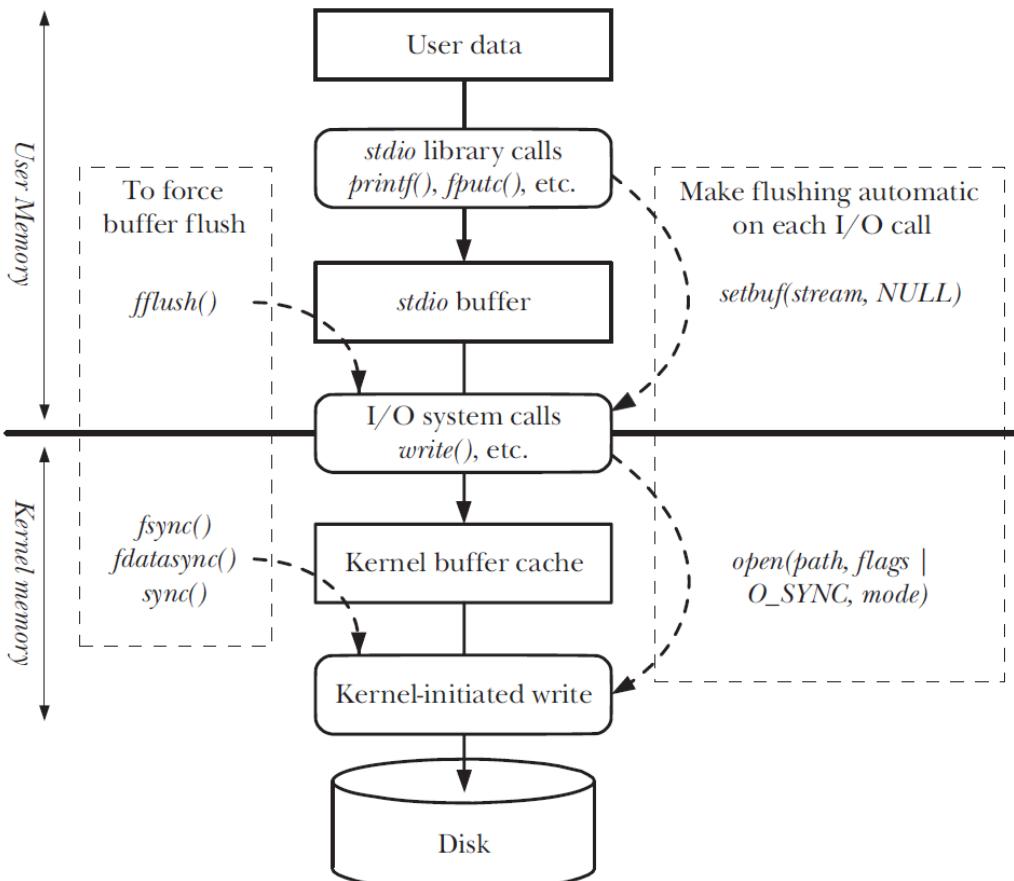


图 13-1: I/O 缓冲小结

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：

- a) 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。

- b) 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？
- c) 尝试对不同文件系统（如 `ext3, XFS, Btrfs, 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为交换分区。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。

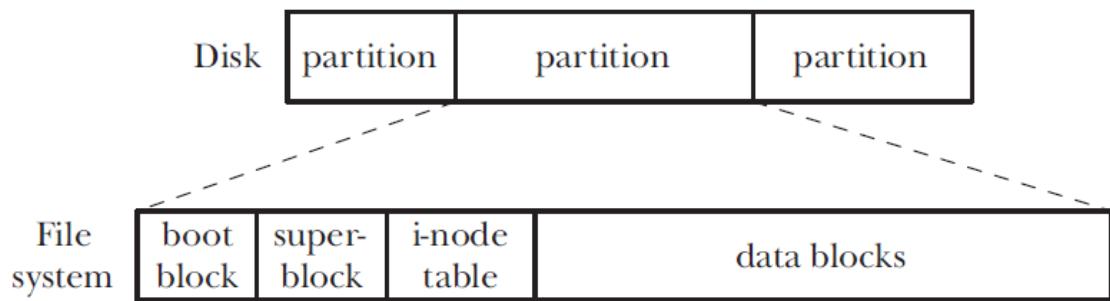


图 14-1：磁盘分区和文件系统的布局

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。

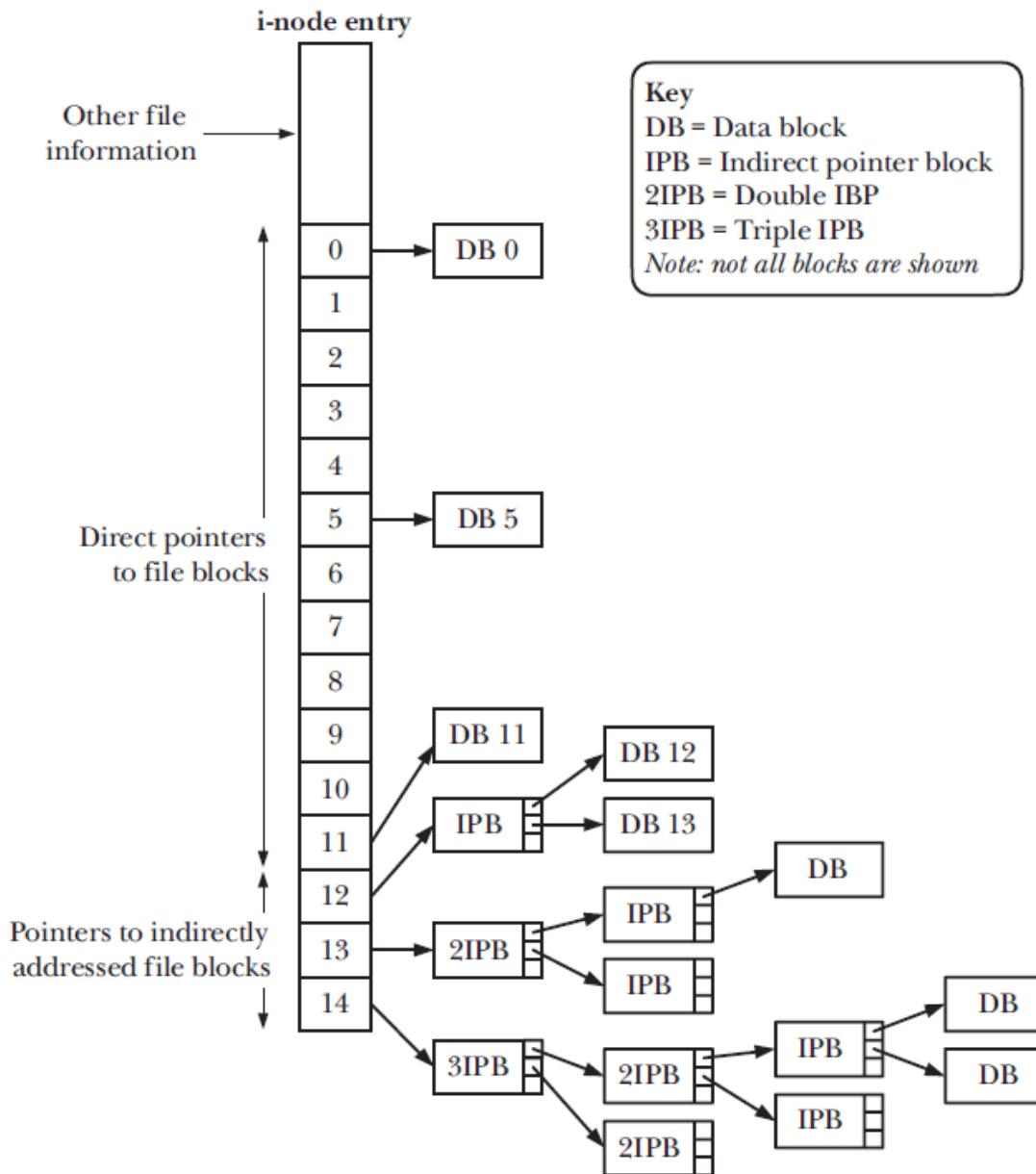


图 14-2: ext2 文件系统某个文件的块结构

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。

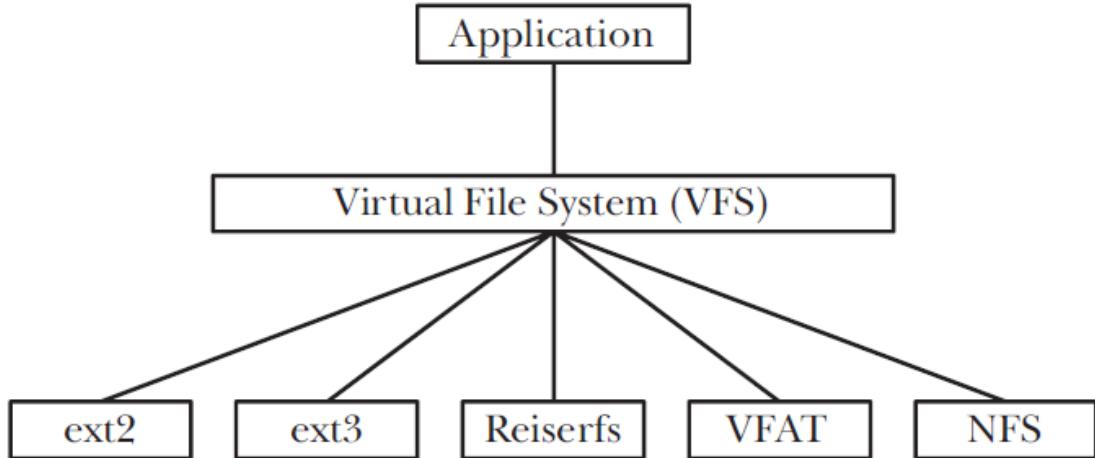


图 14-3：虚拟文件系统

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命名空间。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。

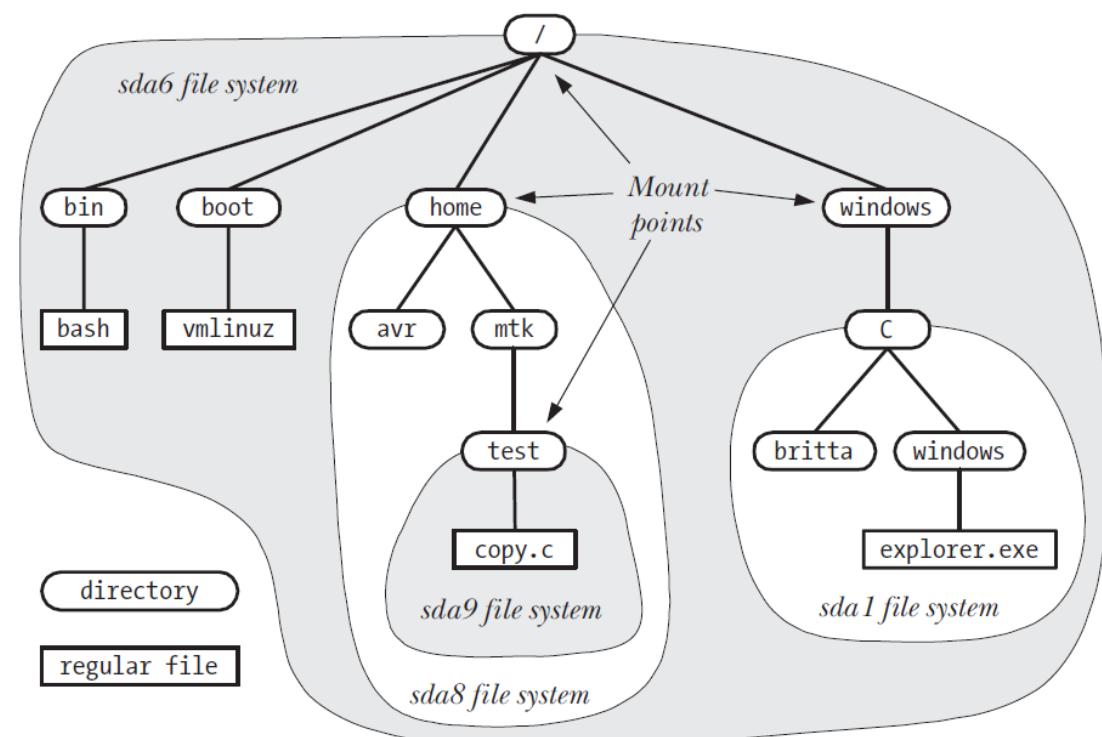


图 14-4：文件系统挂载点和目录层次结构示例

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“单个进程挂载命名空间”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命名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 <i>mountflags</i> 和 <i>data</i> 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，例如单个进程挂载命名空间（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命名空间。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  
Password:  
# mkdir /testfs  
# mkdir /demo  
# mount /dev/sda12 /testfs  
# mount /dev/sda12 /demo  
# mount | grep sda12  
/dev/sda12 on /testfs type ext3 (rw)  
/dev/sda12 on /demo type ext3 (rw)  
# touch /testfs/myfile  
# ls /demo  
myfile
```

需要特权来使用 `mount(8)`

创建两个目录用于挂载点

挂载文件系统到一个挂载点

挂载文件系统到第二个挂载点

验证操作

通过第一个挂载点修改文件

在第二个挂载点查看该文件

`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。下面 shell 会话所示：

\$ su	需要特权才能使用 <code>mount</code>
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  
Password:  
# pwd  
/testfs  
# mkdir d1  
需要特权来使用 mount  
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  
# mkdir src1  
# touch src1/aaa  
# mount --bind src1 top  
# mkdir top/sub  
# mkdir src2  
# touch src2/bbb  
# mount --bind src2 top/sub  
# find top  
top  
top/aaa  
top/sub  
top/sub/bbb
```

这是顶层挂载点
我们会把它挂载到 `top` 下
创建普通绑定挂载
创建目录，用于 `top` 下的子挂载
我们会把它挂载到 `top/sub` 下
创建普通绑定挂载
验证 `top` 挂载树下的内容
这是子挂载

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  
# find dir1  
dir1  
dir1/aaa  
dir1/sub
```

这里我们使用普通绑定挂载

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 *statvbuf);
int fstatvfs(int fd, struct statvfs *statvbuf);
```

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_bsiz` 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_bsiz` 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 `filesystems` 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 `Istat()` 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，`Istat()` 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。

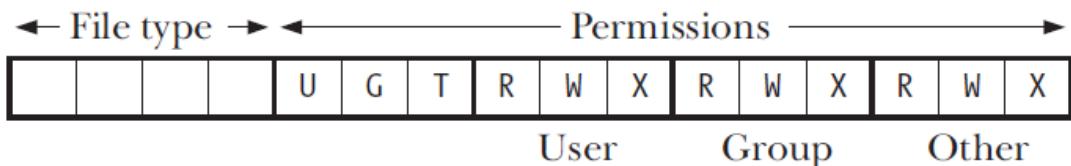


图 15-1: `st_mode` 位掩码布局

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 `Istat()` 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

Constant	Test macro	File type
<code>S_IFREG</code>	<code>S_ISREG()</code>	Regular file
<code>S_IFDIR</code>	<code>S_ISDIR()</code>	Directory
<code>S_IFCHR</code>	<code>S_ISCHR()</code>	Character device
<code>S_IFBLK</code>	<code>S_ISBLK()</code>	Block device
<code>S_IFIFO</code>	<code>S_ISFIFO()</code>	FIFO or pipe
<code>S_IFSOCK</code>	<code>S_ISSOCK()</code>	Socket
<code>S_IFLNK</code>	<code>S_ISLNK()</code>	Symbolic link

`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）；对于共享内存对象（第 54 章），这个域包含该对象的大小。

`st_blocks` 域指示已经为文件分配块的总数量，块单元为 512 字节大小。这个总数量包括了分配给指针块的空间（参考图 14-2，325 页）。选择以 512 字节为单位进行计量，主要是历史原因，512 是 UNIX 下已经实现的所有文件系统的最小块大小。多数现代文件系统使用更大的逻辑块大小。例如 ext2 逻辑块大小为 1024、2048、4096 字节时，`st_blocks` 的值总是文件实际块数量的 2、4、8 倍。

SUSv3 没有定义 `st_blocks` 计量的单位，并允许实现为 `st_blocks` 使用 512 字节之外的单位。多数 UNIX 实现确实使用 512 字节为单位，但 HP-UX 11 却使用文件系统特定的计量单位（例如某些情况下是 1024 字节）。

`st_blocks` 域记录了已经实际分配的磁盘块数量。如果文件包含空洞([4.7 节](#))，这个值就会小于文件大小 (`st_size`) 相应的块数量。（磁盘用量命令 `du -k file`，可以显示文件已经实际分配的空间，单位是千字节；这个值是使用 `st_blocks` 而不是 `st_size` 计算出来的）。

`st_blksize` 域是个令人误导的名字，它不是底层文件系统的块大小，而是在这个文件系统中执行 I/O 的最佳块大小（字节）。小于这个值的 I/O 操作效率将会大打折扣（参考 [13.1 节](#)）。`st_blksize` 的典型值是 4096。

文件时间戳

`st_atime`, `st_mtime`, `st_ctime` 域分别包含文件最后访问、文件最后修改、文件状态最后变化的时间。这三个域的类型为 `time_t`，是标准的 UNIX 时间格式（自 Epoch 起的秒数）。我们在 [15.2 节](#) 更详细地讨论这三个域。

示例程序

清单 15-1 的程序使用 `stat()` 获取命令行指定文件的信息。如果指定了“-l”命令行选项，则程序使用 `Istat()` 来获取符号链接本身的信息。程序打印返回的 `stat` 结构体的所有域。（我们把 `st_size` 和 `st_blocks` 域强制转换为 `long long`，参考 [5.10 节](#)）。程序中使用的 `filePermStr()` 函数定义在程序清单 15-4 中。

下面是使用该程序的示例：

```
$ echo 'All operating systems provide services for programs they run' > apue
$ chmod g+s apue                      打开设置组 ID 位；最后状态变化时间发生变化
```

```
$ cat apue          文件最后访问时间变化
All operating systems provide services for programs they run
$ ./t_stat apue
File type:           regular file
Device containing i-node:   major=3 minor=11
I-node number:        234363
Mode:                 102644 (rw-r--r--)
special bits set:    set-GID
Number of (hard) links: 1
Ownership:            UID=1000 GID=100
File size:            61 bytes
Optimal I/O block size: 4096 bytes
512B blocks allocated: 8
Last file access:     Mon Jun 8 09:40:07 2011
Last file modification: Mon Jun 8 09:39:25 2011
Last status change:   Mon Jun 8 09:39:51 2011
```

清单 15-1：获取和解析文件的 stat 信息

```
-----files/t_stat.c
#define _BSD_SOURCE /* Get major() and minor() from <sys/types.h> */
#include <sys/types.h>
#include <sys/stat.h>
#include <time.h>
#include "file_perms.h"
#include "tlpi_hdr.h"

static void
displayStatInfo(const struct stat *sb)
{
    printf("File type:                ");
    switch (sb->st_mode & S_IFMT) {
        case S_IFREG: printf("regular file\n"); break;
        case S_IFDIR: printf("directory\n"); break;
        case S_IFCHR: printf("character device\n"); break;
        case S_IFBLK: printf("block device\n"); break;
        case S_IFLNK: printf("symbolic (soft) link\n"); break;
        case S_IFIFO: printf("FIFO or pipe\n"); break;
        case S_IFSOCK: printf("socket\n"); break;
        default: printf("unknown file type?\n"); break;
    }
}
```

```
printf("Device containing i-node:    major=%ld minor=%ld\n",
       (long) major(sb->st_dev), (long) minor(sb->st_dev));

printf("I-node number:              %ld\n", (long) sb->st_ino);
printf("Mode:                      %lo (%s)\n",
       (unsigned long) sb->st_mode, filePermStr(sb->st_mode, 0));

if (sb->st_mode & (S_ISUID | S_ISGID | S_ISVTX))
    printf("  special bits set:    %s%s%s\n",
           (sb->st_mode & S_ISUID) ? "set-UID " : "",
           (sb->st_mode & S_ISGID) ? "set-GID " : "",
           (sb->st_mode & S_ISVTX) ? "sticky " : "");

printf("Number of (hard) links:    %ld\n", (long) sb->st_nlink);

printf("Ownership:                  UID=%ld GID=%ld\n",
       (long) sb->st_uid, (long) sb->st_gid);

if (S_ISCHR(sb->st_mode) || S_ISBLK(sb->st_mode))
    printf("Device number (st_rdev): major=%ld; minor=%ld\n",
           (long) major(sb->st_rdev), (long) minor(sb->st_rdev));

printf("File size:                  %lld bytes\n",
       (long long) sb->st_size);
printf("Optimal I/O block size:    %ld bytes\n",
       (long) sb->st_blksize);
printf("512B blocks allocated:    %lld\n",
       (long long) sb->st_blocks);

printf("Last file access:          %s", ctime(&sb->st_atime));
printf("Last file modification:    %s", ctime(&sb->st_mtime));
printf("Last status change:        %s", ctime(&sb->st_ctime));
}

int
main(int argc, char *argv[])
{
    struct stat sb;
    Boolean statLink; /* True if "-l" specified (i.e., use lstat) */
    int fname; /* Location of filename argument in argv[] */

    statLink = (argc > 1) && strcmp(argv[1], "-l") == 0;
```

```

/* Simple parsing for "-l" */

fname = statLink ? 2 : 1;

if (fname >= argc || (argc > 1 && strcmp(argv[1], "--help") == 0))
    usageErr("%s [-l] file\n"
             "          -l = use lstat() instead of stat()\n", argv[0]);

if (statLink) {
    if (lstat(argv[fname], &sb) == -1)
        errExit("lstat");
} else {
    if (stat(argv[fname], &sb) == -1)
        errExit("stat");
}

displayStatInfo(&sb);

exit(EXIT_SUCCESS);
}
-----files/t_stat.c

```

15.2 文件时间戳

stat 结构体的 `st_atime`, `st_mtime`, `st_ctime` 域包含文件的时间戳，这三个域分别记录了文件的最后访问时间、最后修改时间、最后状态变化时间（文件 i-node 信息最后一次修改的时间）。时间戳记录为自 Epoch (1970 年 1 月 1 日，参考 [10.1 节](#)) 起的秒数 (UNIX 标准时间)。

多数 Linux 和 UNIX 本土的文件系统支持所有这三个时间戳，但某些非 UNIX 文件系统却可能不支持。

表 15-2 汇总了本书讨论的各种系统调用和库函数对文件时间戳的改变(某些情况下是改变父目录的相应时间戳)。表的标题 `a`, `m`, `c` 分别对应于 `st_atime`, `st_mtime`, `st_ctime` 域。多数情况下，相应的时间戳会被系统调用设置为当前系统时间；但 `utime()` 和类似的调用 ([15.2.1](#) 和 [15.2.2](#) 节讨论) 可以显式地设置文件的最后访问时间和最后修改时间为任意值。

表 15-2：各种函数对文件时间戳的作用

函数	文件或目录			父目录			备注
	a	m	c	a	m	c	
chmod()			●				fchmod()也一样
chown()			●				lchown()和 fchown()也一样
exec()	●						
link()			●		●	●	影响第二个参数指定的父目录
mkdir()	●	●	●		●	●	
mkfifo()	●	●	●		●	●	
mknod()	●	●	●		●	●	
mmap()	●	●	●				只有更新 MAP_SHARED 映射时才会修改 st_mtime 和 st_ctime
msync()		●	●				只有修改文件时才会改变
open(), creat()	●	●	●		●	●	当创建新文件时
open(), creat()		●	●				当截断现有文件时
pipe()	●	●	●				
read()	●						readv(), pread(), preadv()也一样
readdir()	●						readdir()可能缓冲目录项；只有读取了目录才会更新时间戳
removexattr()			●				fremovexattr(), lremovexattr()也一样
rename()			●				影响两个父目录的时间戳；SUSv3 没有规定是否修改文

						件的 <code>st_ctime</code> , 但是请注意某些实现会修改
<code>rmdir()</code>				●	●	<code>remove(directory)</code> 也一样
<code>sendfile()</code>	●					修改输入文件的时间戳
<code>setxattr()</code>			●			<code>fsetxattr()</code> , <code>Isetxattr()</code> 也一样
<code>symlink()</code>	●	●	●		●	设置链接文件的时间戳（而不是目标文件）
<code>truncate()</code>		●	●			<code>ftruncate()</code> 也一样；只有文件大小改变时才会修改时间戳
<code>unlink()</code>			●		●	<code>remove(file)</code> 也一样；如果之前的链接数量大于 1, 则修改文件的 <code>st_ctime</code>
<code>utime()</code>	●	●	●			<code>utimes()</code> , <code>futimes()</code> , <code>futimens()</code> , <code>lutimes()</code> , <code>utimensat()</code> 也一样
<code>write()</code>		●	●			<code>writev()</code> , <code>pwrite()</code> , <code>pwritev()</code> 也一样

在 [14.8.1](#) 和 [15.5 节](#), 我们讨论了 `mount(2)` 的选项和单个文件标志, 可以阻止更新文件的最后访问时间。[4.3.1 节](#) 讨论的 `open()` 的 `O_NOATIME` 标志也可以实现类似的功能。在某些应用中, 这样做对于性能有很大帮助, 因为降低了访问文件时需要的磁盘操作数量。

尽管多数 UNIX 系统不记录文件的创建时间, 但在最新的 BSD 系统中, `stat` 结构体的 `st_birthtime` 域记录了文件创建时间。

纳秒时间戳

从内核 2.6 开始, Linux 的 `stat` 结构体支持三个纳秒精度的时间戳域。纳秒精度增强了某些程序 (如 `make`) 的准确性, 这些程序需要根据文件时间戳的相对顺序来做出决定。

SUSv3 没有为 `stat` 结构体规定纳秒时间戳，但 SUSv4 增加了这个规定。

不是所有文件系统都支持纳秒时间戳，目前 JFS, XFS, ext4, Btrfs 支持；但 ext2, ext3, Reiserfs 不支持。

在 glibc API 中（版本 2.3 起），时间戳域都被定义为 `timespec` 结构体（我们在本节后面讨论 `utimensat()` 时描述这个结构体）。`timespec` 以秒和纳秒来表示时间。我们需要定义适当的宏，才能使用传统的域名（`st_atime`, `st_mtime`, `st_ctime`）来访问这些结构体的秒数部分。如果要获取文件最后访问时间的纳秒部分，我们可以使用 `st_atim.tv_nsec`（注意是 `st_atim`，没有 `e`）。

15.2.1 修改文件时间戳：`utime()` 和 `utimes()`

文件 i-node 中的最后访问时间和最后修改时间戳，可以显式地使用 `utime()` 或相关的系统调用来修改。例如 `tar` 和 `unzip` 这样的程序在解压时使用这些系统调用来重置文件的时间戳。

```
#include <utime.h>

int utime(const char *pathname, const struct utimbuf *buf);
成功时返回 0; 出错时返回 -1
```

`pathname` 参数指定要修改时间戳的文件，如果 `pathname` 是一个符号链接，将首先进行解引用。`buf` 参数可以是 `NULL`，或者指向 `utimbuf` 结构体的指针：

```
struct utimbuf {
    time_t actime; /* 访问时间 */
    time_t modtime; /* 修改时间 */
};
```

这个结构体的域以 Epoch 起的秒数来计量（[10.1 节](#)）。

以下两种不同情况决定了 `utime()` 的工作方式：

- 如果 `buf` 指定为 `NULL`，则最后访问时间和最后修改时间都设置为当前时间。在这种情况下，要么进程的有效用户 ID 匹配文件的用户 ID（拥有者）；要么进程拥有文件的写入权限（逻辑上，进程拥有文件的写权限，就可

以采用其它具有修改文件时间戳副作用的系统调用); 要么进程必须拥有特权 (CAP_FOWNER 或 CAP_DAC_OVERRIDE)。(准确地说, 在 Linux 中, 应该是检查进程的文件系统用户 ID, 而不是有效用户 ID, 来匹配文件的用户 ID, 如 [9.5 节](#) 所述)。

- 如果 buf 指定为 utimbuf 结构体的指针, 则文件的最后访问时间和最后修改时间更新为这个结构体指定的相应值。在这种情况下, 进程的有效用户 ID 必须匹配文件的用户 ID (只拥有文件的写入权限是不够的); 或者调用进程拥有特权 (CAP_FOWNER)。

如果要修改其中一个文件时间戳, 我们首先使用 `stat()` 获取这两个时间, 使用其中之一来初始化 `utimbuf` 结构体, 然后设置另外一个域为我们想要的值。下面代码演示了这个方法, 修改文件的最后修改时间等于最后访问时间:

```
struct stat sb;
struct utimbuf utb;

if (stat(pathname, &sb) == -1)
    errExit("stat");
utb.actime = sb.st_atime; /* Leave access time unchanged */
utb.modtime = sb.st_atime;
if (utime(pathname, &utb) == -1)
    errExit("utime");
```

成功调用 `utime()` 总是设置文件的最后状态变化时间为当前时间。

Linux 还提供起源于 BSD 的 `utimes()` 系统调用, 功能与 `utime()` 类似。

<pre>#include <sys/time.h> int utimes(const char *pathname, const struct timeval tv[2]);</pre>	成功时返回 0; 出错时返回 -1
---	-------------------

`utime()` 和 `utimes()` 之间最大的区别在于 `utimes()` 允许时间值指定为微秒精度 (`timeval` 结构体在 [10.1 节](#) 讨论过)。这样就能够 (部分) 访问 Linux 2.6 提供的

纳秒精度文件时间戳。`tv[0]`指定新的文件最后访问时间，`tv[1]`指定新的文件最后修改时间。

本书源码包中的 `files/t_utimes.c` 提供了一个使用 `utimes()` 的例子。

`futimes()` 和 `lutimes()` 库函数的功能与 `utimes()` 类似。区别在于指定修改文件的参数。

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

int futimes(int fd, const struct timeval tv[2]);
int lutimes(const char *pathname, const struct timeval tv[2]);
```

成功时都返回 0；出错时都返回 -1

`futimes()` 使用打开文件描述符 `fd` 来指定要修改的文件。

`lutimes()` 使用路径名来指定要修改的文件，与 `utimes()` 的区别在于符号链接，`lutimes()` 不对符号链接执行解引用，而是直接修改符号链接本身的时间戳。

glibc 2.3 开始支持 `futimes()` 函数；glibc 2.6 开始支持 `lutimes()` 函数。

15.2.2 修改文件时间戳：`utimensat()` 和 `futimens()`

`utimensat()` 系统调用（内核 2.6.22 开始支持）和 `futimens()` 库函数（glibc 2.6 开始支持）提供了设置文件最后访问时间和最后修改时间的扩展功能。同时这两个接口还拥有以下优点：

- 我们可以设置纳秒精度的时间戳，相比 `utimes()` 的微秒精度更为精确。
- 可以独立地设置时间戳（一次设置一个）。正如前面所述，使用老的接口修改某个时间戳，我们必须首先调用 `stat()` 获得另一个时间戳的值，然后把这个值与我们想要设置的时间戳一起指定给老接口。（如果另一个进程在这两个步骤之间更新时间戳，可能导致竞争条件）。
- 我们可以独立地设置任一时间戳为当前值。使用老接口来修改某个时间戳为当前值，我们需要首先调用 `stat()` 来获得另一个时间戳的值，然后调用 `gettimeofday()` 来获得当前时间。

SUSv3 没有规定这两个接口，但 SUSv4 对这两个接口进行了规定。

`utimensat()` 系统调用更新 `pathname` 指定文件的时间戳，`times` 数组指定要修改的时间戳。

```
#define _XOPEN_SOURCE 700      /* Or define _POSIX_C_SOURCE >= 200809 */
#include <sys/stat.h>

int utimensat(int dirfd, const char *pathname,
              const struct timespec times[2], int flags);

```

成功时返回 0；出错时返回 -1

如果 `times` 指定为 `NULL`，则文件的两个时间戳都更新为当前时间。如果 `times` 不是 `NULL`，则最后访问时间戳更新为 `times[0]`，最后修改时间戳更新为 `times[1]`。`times` 数组的元素是一个结构体，定义如下：

```
struct timespec {
    time_t tv_sec;      /* 秒 ('time_t' 是整数类型) */
    long tv_nsec;        /* 纳秒 */
};
```

该结构体中的域指定从 Epoch（[10.1 节](#)）开始的秒数和纳秒数。

要设置其中一个时间戳为当前时间，我们可以为相应的 `tv_nsec` 域指定特殊值 `UTIME_NOW`；要保留其中一个时间戳不变，我们可以为相应的 `tv_nsec` 域指定特殊值 `UTIME OMIT`。上面两种情况下，`tv_sec` 域都被忽略。

`dirfd` 参数可以指定为 `AT_FDCWD`，这种情况下 `pathname` 参数按 `utimes()` 的方式来解释；或者指定为引用目录的文件描述符，这个用法我们在 [18.11 节](#) 描述。

`flags` 参数可以是 `0`；或者 `AT_SYMLINK_NOFOLLOW`，表示不对符号链接解引用（修改符号链接本身的时间戳）。与之相反，`utimes()` 总是对符号链接解引用。

以下代码片断设置最后访问时间为当前值，并保持最后修改时间不变：

```
struct timespec times[2];

times[0].tv_sec = 0;
times[0].tv_nsec = UTIME_NOW;
```

```
times[1].tv_sec = 0;
times[1].tv_nsec = UTIME_OMIT;
if (utimensat(AT_FDCWD, "myfile", times, 0) == -1)
    errExit("utimensat");
```

使用 `utimensat()` 和 `futimens()` 修改时间戳的权限规则和老的 API 类似，细节请参考 `utimensat(2)` 手册页。

`futimens()` 库函数更新打开文件描述符 `fd` 引用文件的时间戳。

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

int futimens(int fd, const struct timespec times[2]);
成功时返回 0；出错时返回 -1
```

`futimens()` 的 `times` 参数用法和 `utimensat()` 一样。

15.3 文件所有权

每个文件都有一个相关联的用户 ID (UID) 和组 ID (GID)。这两个 ID 确定了文件所属的用户和组。接下来我们讨论确定新文件所有权的规则，以及修改文件所有权的系统调用。

15.3.1 新文件的所有权

当一个新文件被创建时，它的用户 ID 是进程的有效用户 ID；新文件的组 ID 要么是进程的有效组 ID (System V 的默认行为)，要么是父目录的组 ID (BSD 的默认行为)。如果要创建一个项目目录，其中所有文件都属于某个特定的组，并且可以被该组的所有成员访问，采用 BSD 的行为就是非常有用的。新文件的组 ID 如何确定，依赖于各种因素，包括新文件创建的文件系统类型。下面我们先介绍 ext2 和其它一些文件系统采用的规则。

准确地说，在 Linux 中，本节前面所说的有效用户或组 ID 实际上应该是文件系统用户或文件系统组 ID ([9.5 节](#))。

挂载 ext2 文件系统时，mount 命令只能指定-o grpid（或同义的-o bsdgroups）和-o nogrpid（或同义的-o sysvgroups）选项之一。如果这两个选项均未指定，默认为-o nogrpid。如果指定了-o grpid，则新文件总是继承父目录的组 ID；如果指定了-o nogrpid，则新文件的组 ID 默认为进程的有效组 ID。但是如果启用了目录的设置组 ID 位（通过 chmod g+s），则文件的组 ID 仍将继承自父目录的组 ID。表 15-3 汇总了这些规则。

在 [18.6 节](#)，我们会看到当目录启用了设置组 ID 位时，则新创建的子目录也将自动启用设置组 ID 位。因此上面描述的目录的设置组 ID 对新文件所有权的影响对整个目录树都有效。

表 15-3：确定新创建文件所属组的规则

文件系统挂载选项	父目录的设置组 ID 位是否启用？	新文件所属的组
-o grpid, -o bsdgroups	(忽略)	父目录的组 ID
-o nogrpid, -o sysvgroups (默认)	否	进程的有效组 ID
	是	父目录的组 ID

在本书写作之时，只有 ext2, ext3, ext4、和 XFS（Linux 2.6.14 起）支持 grpid 和 nogrpid 挂载选项。其它文件系统都遵循 nogrpid 规则。

15.3.2 改变文件所有权：chown(), fchown(), lchown()

chown(), fchown(), lchown() 系统调用可以修改文件的拥有者（用户 ID）和组（组 ID）。

```
#include <unistd.h>

int chown(const char *pathname, uid_t owner, gid_t group);

#define _XOPEN_SOURCE 500          /* Or: #define _BSD_SOURCE */
#include <unistd.h>
```

```
int lchown(const char *pathname, uid_t owner, gid_t group);  
int fchown(int fd, uid_t owner, gid_t group);
```

成功时都返回 0; 出错时都返回 -1

这些系统调用之间的区别类似于 `stat()` 系统调用家族:

- `chown()` 修改 `pathname` 参数指定文件的所有权;
- `lchown()` 和 `chown()` 一样, 但是如果 `pathname` 是符号链接, 则修改链接文件本身的所有权, 而不是符号链接所引用的文件;
- `fchown()` 修改打开文件描述符 `fd` 引用文件的所有权。

`owner` 参数指定文件的新用户 ID, `group` 参数则指定文件的新组 ID。如果只需要修改其中一个 ID, 我们可以为另一个参数指定 -1, 表示相应的 ID 不变。

在 Linux 2.2 之前, `chown()` 不对符号链接解引用。Linux 2.2 改变了 `chown()` 的语义, 对符号链接进行解引用; 同时增加了新的 `lchown()` 系统调用, 提供老的 `chown()` 系统调用的行为。

只有特权 (`CAP_CHOWN`) 进程才能使用 `chown()` 修改文件的用户 ID。非特权进程只能使用 `chown()` 修改文件的组 ID 为自己所属的其它组, 而且进程必须拥有该文件 (进程的有效用户 ID 匹配文件的用户 ID)。特权进程可以修改文件的组 ID 为任意值。

如果文件的拥有者或组发生改变, 则设置用户 ID 和设置组 ID 权限位将同时被关闭。这是安全上的预防措施, 确保以下情况的发生: 普通用户启用某个可执行文件的设置用户 ID 或设置组 ID 位, 然后通过某种手段修改文件的所有权为特权用户 (或组), 从而在执行文件时获得特权身份。

SUSv3 没有规定超级用户修改可执行文件所有权时, 是否要关闭文件的设置用户 ID 和设置组 ID 位。Linux 2.0 在这种情况下会关闭这两个权限位, 而某些早期的 2.2 版本内核 (到 2.2.12 为止) 却不会关闭。之后的 2.2 内核又重新采用了 2.0 内核的行为, 超级用户修改文件所有权的行为和任何用户都一样, 而且随后的所有内核版本都采用这种行为。(但是如果我们以 root 身份使用 `chown` 命令来修改文件的所有权, 在调用 `chown()` 之后, `chown` 命令会使用 `chmod()` 系统调用来重新启用文件的设置用户 ID 和设置组 ID 位)。

在修改拥有者和组时，如果组执行权限已经关闭，或者修改目录的所有权，则设置组 ID 权限位将不会被关闭。在上面这两种情况里，设置组 ID 位的用途并不是创建设置组 ID 程序，因此不需要关闭设置组 ID 位。设置组 ID 位的这两种用法如下所述：

- 如果组执行权限位关闭，则设置组 ID 权限位用于启用强制文件锁（[55.4 节](#)讨论）。
- 对于目录来说，设置组 ID 位用于控制该目录下新创建文件的所有权（[15.3.1 节](#)）。

清单 15-2 演示了 chown() 的用法，该程序允许用户修改任意数量文件的拥有者和组，由命令行参数指定要修改的文件。（这个程序使用了第 216 页清单 8-1 中的 userIdFromName() 和 groupIdFromName() 函数，用于转换用户名和组名为相应的数值 ID）。

清单 15-2：修改文件的拥有者和组

```
-----files/t_chown.c
#include <pwd.h>
#include <grp.h>
#include "ugid_functions.h" /* Declarations of userIdFromName()
                           and groupIdFromName() */
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    uid_t uid;
    gid_t gid;
    int j;
    Boolean errFnd;

    if (argc < 3 || strcmp(argv[1], "--help") == 0)
        usageErr("%s owner group [file...]\\n"
                 " owner or group can be '-' , "
                 "meaning leave unchanged\\n", argv[0]);
```

```
if (strcmp(argv[1], "-") == 0) {      /* "-" ==> don't change owner */
    uid = -1;
} else {                                /* Turn user name into UID */
    uid = userIdFromName(argv[1]);
    if (uid == -1)
        fatal("No such user (%s)", argv[1]);
}

if (strcmp(argv[2], "-") == 0) {      /* "-" ==> don't change group */
    gid = -1;
} else {                                /* Turn group name into GID */
    gid = groupIdFromName(argv[2]);
    if (gid == -1)
        fatal("No group user (%s)", argv[1]);
}

/* Change ownership of all files named in remaining arguments */

errFnd = FALSE;
for (j = 3; j < argc; j++) {
    if (chown(argv[j], uid, gid) == -1) {
        errMsg("chown: %s", argv[j]);
        errFnd = TRUE;
    }
}

exit(errFnd ? EXIT_FAILURE : EXIT_SUCCESS);
}

-----files/t_chown.c
```

15.4 文件权限

在这一节，我们讨论应用于文件和目录的权限机制。尽管我们这里讨论的权限主要应用于普通文件和目录，但是这些规则可以应用到所有文件类型，包括设备、FIFO、UNIX 域 socket。此外，System V 和 POSIX 进程间通信对象（共享内存、信号量、消息队列）也有权限掩码，使用的规则也类似于这里讨论的文件权限。

15.4.1 普通文件权限

正如 [15.1 节](#)所述，`stat` 结构体的 `st_mode` 域的后 12 位定义了文件的权限。其中前 3 位比较特殊，分别是设置用户 ID、设置组 ID、sticky 位（在图 15-1 中分别标记为 U, G, T）。我们在 [15.4.5 节](#)再详细地讨论这三个位。掩码中剩下的 9 位定义了三种类型用户访问文件的权限。文件权限掩码把用户类型划分为三类：

- 拥有者（也称为用户）：授予文件拥有者的权限。
术语用户用于 `chmod` 等命令，这些命令使用缩写 `u` 来引用这个权限分类。
- 组：授予文件组成员用户的权限。
- 其它：授予所有其它人的权限。

每种用户分类又被分配了以下三种权限：

- 读：可以读取文件内容。
- 写：可以修改文件内容。
- 执行：可以执行文件（如果是一个程序或脚本）。要执行一个脚本文件（如 `bash` 脚本），必须同时拥有读和执行权限。

使用命令 `ls -l` 可以查看文件的权限和所有权，如下面例子所示：

```
$ ls -l myscript.sh
-rwxr-x---    1 mtk      users     1667 Jan 15 09:22 myscript.sh
```

在上面例子中，文件权限显示为 `rwxr-x---`（最前面的横线表示文件类型为普通文件）。我们需要把这 9 个字符分解为 3 组字符串来解释文件的权限，分别指示是否启用了读、写、执行权限。第 1 组显示了拥有者的权限，拥有读、写、执行权限；第 2 组显示了组权限，拥有读和执行权限，但不能写；最后第 3 组是其它人的权限，在上面例子中其它人没有启用任何权限。

头文件`<sys/stat.h>`中定义了一组常量，`stat` 结构体的 `st_mode` 域可以与之相与（`&`），从而检查某个特定的权限位是否设置。（包含`<fcntl.h>`也能获得这些常量的定义，`<fcntl.h>`还声明了 `open()` 系统调用的原型）。这些常量如表 15-4 所示：

表 15-4：文件权限位常量

常量	八进制值	权限位
S_ISUID	04000	设置用户 ID
S_ISGID	02000	设置组 ID
S_ISVTX	01000	Sticky
S_IRUSR	0400	用户可读
S_IWUSR	0200	用户可写
S_IXUSR	0100	用户可执行
S_IRGRP	040	组可读
S_IWGRP	020	组可写
S_IXGRP	010	组可执行
S_IROTH	04	其它可读
S_IWOTH	02	其它可写
S_IXOTH	01	其它可执行

除了表 15-4 所列的常量，每种分类的三个权限掩码合在一起还定义了另外三个常量，分别是拥有者：S_IRWXU(0700)、组：S_IRWXG(070)、其它：S_IRWXO(07)。

清单 15-3 的头文件声明了 `filePermStr()` 函数，给定一个文件权限掩码，该函数返回一个静态分配的字符串，按 `ls` 命令的风格描述了该权限掩码。

清单 15-3: `file_perms.c` 的头文件

```
-----files/file_perms.h
#ifndef FILE_PERMS_H
#define FILE_PERMS_H

#include <sys/types.h>

#define FP_SPECIAL 1           /* 在返回字符串中包括设置用户 ID、设置组 ID、
                                Sticky 位的信息 */
```

```

char *filePermStr(mode_t perm, int flags);

#endif
-----files/file_perms.h

```

如果 filePermStr() 的 flags 参数设置了 FP_SPECIAL 标志，则返回的字符串包含设置用户 ID、设置组 ID、Sticky 位的设置信息，同样也采用了 ls 命令的风格。

清单 15-4 显示了 filePermStr() 函数的实现。我们在清单 15-1 的程序中使用了这个函数。

清单 15-4：转换文件权限掩码为字符串

```

-----files/file_perms.c
#include <sys/stat.h>
#include <stdio.h>
#include "file_perms.h" /* 这个实现的接口 */

#define STR_SIZE sizeof("rwxrwxrwx")

char *           /* 为文件权限掩码返回 ls 风格的字符串 */
filePermStr(mode_t perm, int flags)
{
    static char str[STR_SIZE];

    snprintf(str, STR_SIZE, "%c%c%c%c%c%c%c%c",
        (perm & S_IRUSR) ? 'r' : '-',
        (perm & S_IWUSR) ? 'w' : '-',
        (perm & S_IXUSR) ?
            (((perm & S_ISUID) && (flags & FP_SPECIAL)) ? 's' : 'x') :
            (((perm & S_ISUID) && (flags & FP_SPECIAL)) ? 'S' : '-'),
        (perm & S_IRGRP) ? 'r' : '-',
        (perm & S_IWGRP) ? 'w' : '-',
        (perm & S_IXGRP) ?
            (((perm & S_ISGID) && (flags & FP_SPECIAL)) ? 's' : 'x') :
            (((perm & S_ISGID) && (flags & FP_SPECIAL)) ? 'S' : '-'),
        (perm & S_IROTH) ? 'r' : '-',
        (perm & S_IWOTH) ? 'w' : '-',
        (perm & S_IXOTH) ?
            (((perm & S_ISVTX) && (flags & FP_SPECIAL)) ? 't' : 'x') :
            (((perm & S_ISVTX) && (flags & FP_SPECIAL)) ? 'T' : '-'));

    return str;
}
-----files/file_perms.c

```

15.4.2 目录权限

目录的权限机制和文件一样，但是对三个权限位的解释是不同的：

- 读：可以列出（如 ls 命令）目录的内容（也就是文件名列表）。

要体验目录读权限位的验证操作，请注意某些 Linux 发行版的 ls 命令可能是包含了标志（如-F）的别名，需要访问目录中文件的 i-node 信息，而这个操作要求拥有目录的执行权限。为了确保我们使用原始的 ls 命令，可以在执行命令时指定完整的路径（/bin/ls）。

- 写：可以在目录中创建和删除文件。注意删除文件操作，并不需要拥有文件本身的所有权限。
- 执行：可以访问目录中的文件。目录的执行权限有时候也叫做 search 权限。

在访问一个文件时，需要拥有路径名中所有目录的执行权限。例如读取文件 /home/mtk/x，要求拥有 /, /home, /home/mtk 的执行权限（同时还需要文件 x 本身的读权限）。如果当前工作目录是 /home/mtk/sub1，我们访问相对路径 .. /sub2/x，则需要 /home/mtk 和 /home/mtk/sub2 的执行权限（但不需要 / 和 /home）。

目录的读权限只允许我们查看该目录下的文件名列表。我们必须拥有目录的执行权限才能访问目录下文件的内容和 i-node 信息。

反过来，如果我们拥有目录的执行权限，但是没有读权限，只要我们知道目录下某个文件的名字就可以访问它，但是我们不能列出目录的内容。这是一个简单而常用的技术，用来控制公共目录的内容访问。

要在目录中增加或删除文件，我们需要同时拥有目录的执行和写权限。

15.4.3 权限检查算法

当我们为系统调用指定路径来访问文件或目录时，内核都会检查文件的权限。如果路径名包含目录前缀，则除了检查文件本身的权限，内核还会检查每个目录的执行权限。权限检查使用进程的有效用户 ID、有效组 ID、和附加组 ID。（严格

地说，Linux 对文件权限的检查，使用的是文件系统用户和组 ID，而不是相应的有效 ID，[9.5 节](#)对此有详细描述)。

一旦使用 `open()` 打开了文件，随后的系统调用使用文件描述符，就不再需要权限检查（例如 `read()`, `write()`, `fstat()`, `fcntl()`, `mmap()`）。

内核检查权限时使用如下规则：

1. 如果进程拥有特权，允许所有访问。
2. 如果进程的有效用户 ID 等于文件的用户 ID（拥有者），则根据文件的拥有者权限来授予访问权限。例如文件的权限掩码开启了拥有者可读权限位，则允许读访问；否则不允许读访问。
3. 如果进程的有效组 ID 或者进程的任一附加组 ID，匹配文件的组 ID（组拥有者），则根据文件的组权限来授予访问权限。
4. 否则，根据文件的 other 权限来授予访问权限。

在内核代码中，只有进程经过上面其它检查，无法获得所需权限时，才会进行进程是否特权的检查。这样做是为了避免不必要的设置 ASU 进程会计标志，表示进程正在使用超级用户特权 ([28.1 节](#))。

内核按顺序检查拥有者、组、其它权限，一旦发现可用的规则就立即停止检查。这样可能导致非预期的结果：例如组权限超过拥有者权限，则拥有者实际上比文件组成员拥有的权限要少，下面例子说明了这种情况：

```
$ echo 'Hello world' > a.txt
$ ls -l a.txt
-rw-r--r-- 1 mtk users 12 Jun 18 12:26 a.txt
$ chmod u-rw a.txt                         去掉拥有者的读和写权限
$ ls -l a.txt
----r--r-- 1 mtk users 12 Jun 18 12:26 a.txt
$ cat a.txt
cat: a.txt: Permission denied            拥有者现在不能读取文件
$ su avr
Password:                                 换另一个用户
$ groups
users staff teach cs                     哪些用户在拥有者的组中
$ cat a.txt
Hello world                             因此可以读取文件
```

如果其它人的权限超过拥有者和组，同样也是上面的情况。

由于文件权限和所有权信息维护在文件的 i-node 中，引用相同 i-node 的所有文件名（链接）都将共享这些权限信息。

Linux 2.6 提供了访问控制列表（ACL），能够以每个用户和每个组的粒度来定义文件权限。如果一个文件拥有 ACL，则使用修改版的权限检查算法。我们在[第 17 章](#)讨论 ACL。

特权进程的权限检查

在上面我们说过，如果进程拥有特权，在检查权限时将授予所有访问权限。现在我们要为这句话增加一个附加条件。对于文件（不是目录）来说，只有文件的某个权限分类（拥有者、组、其它）拥有执行权限，Linux 才会授予特权进程对文件的执行权限。在某些 UNIX 实现中，即使文件完全没有执行权限，特权进程也能够执行它。当访问目录时，特权进程总是获得目录的执行（查找）权限。

我们可以使用 Linux 进程能力的两个术语：CAP_DAC_READ_SEARCH 和 CAP_DAC_OVERRIDE（[39.2 节](#)），来重新对特权进程进行描述。拥有 CAP_DAC_READ_SEARCH 能力的进程总是拥有任何文件的读权限，并且总是拥有目录的读和执行权限（可以访问目录中的文件和目录的文件列表）。拥有 CAP_DAC_OVERRIDE 能力的进程总是拥有任何文件的读写权限，并且只要文件某一种类别（拥有者、组、其它）的执行权限位开启，就能拥有该文件的执行权限；这样的进程还拥有任意目录的执行权限。

15.4.4 检查文件可访问性：access()

[15.4.3 节](#)已经描述过，有效用户和组 ID，加上附加组 ID，用于确定进程访问文件时拥有的权限。另外基于进程的实际用户和组 ID 来检查文件的可访问性也是可能的（如设置用户 ID 和设置组 ID 程序）。

access() 系统调用使用进程的实际用户 ID 和实际组 ID（以及附加组 ID），来检查 pathname 指定文件的可访问性。

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

```
int access(const char *pathname, int mode);
```

所有权限都允许时返回 0；否则返回 -1

如果 `pathname` 是一个符号链接，`access()`会对其解引用。

`mode` 参数是位掩码，由表 15-5 中的一个或多个常量“或”组成。如果 `mode` 指定的所有权限在 `pathname` 上都允许，则 `access()` 返回 0；否则只要有任何一个权限不可用，`access()` 返回 -1（或者出错时）。

表 15-5: `access()` 的 `mode` 常量

常量	描述
<code>F_OK</code>	文件是否存在？
<code>R_OK</code>	文件能否读取？
<code>W_OK</code>	文件能否写入？
<code>X_OK</code>	文件能否被执行？

调用 `access()` 和随后对文件的操作之间存在一定的时间间隙，这样就无法保证 `access()` 返回的信息在后面操作的时候（不管间隔有多短）仍然是正确的。这种情况在某些应用设计中可能会导致安全漏洞。

例如我们有一个 `set-user-ID-root` 的程序使用 `access()` 来检测程序的实际用户 ID 对某个文件是否可以访问，假设 `access()` 返回可以访问，于是应用对文件执行操作（如 `open()` 或 `exec()`）。

问题是如果 `access` 指定的路径名是一个符号链接，在第二步访问文件之前，恶意用户故意修改链接使其指向不同的文件，则 `set-user-ID-root` 程序可能最终会操作一个实际用户 ID 没有权限的文件。（这是检测时间和使用时间所导致竞争条件的一个例子，我们在 [38.6 节](#) 会详细讨论）。由于这个原因，推荐的最佳实践是完全避免使用 `access()`（如[Borisov, 2005]）。在上面的例子中，我们可以临时地修改设置用户 ID 进程的有效（或文件系统）用户 ID，尝试我们需要的操作（如 `open()` 或 `exec()`），然后通过检查返回值，来确定操作是否由于权限问题而失败。

GNU C 库提供一个类似的非标准函数：`euidaccess()`（或者同义的 `eaccess()`），使用进程的有效用户 ID 来检查对某个文件的可访问性。

15.4.5 设置用户 ID, 设置组 ID, sticky 位

除了拥有者、组、其它的 9 个权限位以外，文件权限掩码另外还包含 3 个位，分别是设置用户 ID (04000 位)、设置组 ID (02000 位)、Sticky 位 (01000 位)。我们在 [9.3 节](#) 已经讨论过使用设置用户 ID 和设置组 ID 权限位来创建特权程序。前面还描述过设置组 ID 位的另外两个用途：使用 `nogrpid` 选项挂载时控制目录下新创建文件的组所有权 ([15.3.1 节](#))；允许文件的强制锁 ([55.4 节](#))。在本节的剩余部分，我们只讨论 `sticky` 位的使用。

在老的 UNIX 实现中，`sticky` 位用来加速常用程序的运行。如果一个程序文件设置了 `sticky` 位，则第一次执行该程序时，会在交换区域保存一份程序文本 (program text) 的拷贝（这也是 `sticky` 名称的由来，表示它“粘”在交换区域），以此来加速随后执行程序的速度。现代 UNIX 实现拥有更加复杂和强大的内存管理系统，已经替代和废弃了 `sticky` 权限位的这种用法。

`sticky` 权限位的常量名可以在表 15-4 中找到：`S_ISVTX`，根据 `sticky` 位的另一个名字 `saved-text` 来命名。

在现代 UNIX 实现（包括 Linux）中，`sticky` 权限位起着另外一个完全不同的作用。对于目录，`sticky` 位的作用是“限制删除标志”。如果目录设置了这个位，则非特权进程必须拥有目录的写权限，并且是目录或者相应文件的拥有者，才可以 `unlink(unlink(), rmdir())` 和重命名(`rename()`)目录中的文件(拥有 `CAP_FOWNER` 能力的进程可以绕过后一个所有权检查)。这样就可以创建一个许多用户共享的目录，用户只能在目录中创建和删除自己的文件，但不能删除其它用户的文件。由于这个原因，`/tmp` 目录通常都设置了 `sticky` 权限位。

文件的 `sticky` 权限位可以通过 `chmod` 命令来设置 (`chmod +t file`)，或者使用 `chmod()` 系统调用。如果文件设置了 `sticky` 位，`ls -l` 会在文件的其它人执行权限位上显示出来，如果其它人的执行权限启用，则显示小写的字母 `t`，如果其它人的执行权限关闭，则显示大写的字母 `T`。

```
$ touch tfile
$ ls -l tfile
-rw-r--r--    1 mtk      users   0 Jun 23 14:44    tfile
```

```
$ chmod +t tfile
$ ls -l tfile
-rw-r--r-T    1 mtk      users  0 Jun 23 14:44    tfile
$ chmod o+x tfile
$ ls -l tfile
-rw-r--r-t    1 mtk      users  0 Jun 23 14:44    tfile
```

15.4.6 进程文件模式创建掩码: umask()

现在让我们来更加深入地讨论新创建文件或目录的权限。对于新文件，内核使用 `open()` 或 `creat()` 指定的 `mode` 参数来分配权限；对于目录，则根据 `mkdir()` 的 `mode` 参数来设置权限。但是这些设置可以通过文件模式创建掩码来修改，我们一般称之为 `umask`。`umask` 是一个进程属性，规定该进程创建新文件和目录时哪些权限位总是被关闭。

通常进程直接使用从父 `shell` 中继承而来的 `umask`，因此用户可以通过 `shell` 内建的 `umask` 命令来控制执行进程的 `umask`。`shell` 的 `umask` 命令修改所有 `shell` 进程的 `umask`。

多数 `shell` 的初始化文件都设置默认的 `umask` 为八进制值 `022`(`----``w-``w-`)，这个值规定组和其它人的写入权限总是被关闭。因此假设 `open()` 调用的 `mode` 参数是 `0666`（允许所有用户的读和写权限，典型值），则新文件的拥有者可以读取和写入，其他人则只有读权限（`ls -l` 显示为 `rw-r--r--`）。相应地，假设 `mkdir()` 的 `mode` 指定为 `0777`（允许所有用户的所有权限），则新目录的拥有者拥有所有权限，组和其它人则只有读和执行权限（`rwxr-xr-x`）。

`umask()` 系统调用修改进程的 `umask` 为 `mask` 指定的值。

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

mode_t umask(mode_t mask);
```

总是成功返回之前的进程 `umask`

`mask` 参数可以指定为八进制数值，或者“或”(|)一组常量指定，前面的表 15-4 列出了这些常量。

调用 `umask()` 总是成功，并返回之前的进程 `umask`。

清单 15-5 演示了 `umask()` 的使用，结合 `open()` 和 `mkdir()`。运行这个程序，可以得到如下输出：

```
$ ./t_umask
Requested file perms:    rw-rw----
```

这是我们请求的权限

```
Process umask:           ----wx-wx
```

这是我们拒绝的权限

```
Actual file perms:      rw-r----
```

因此这是最终的权限

```
Requested dir. perms:   rwxrwxrwx
```

```
Process umask:           ----wx-wx
```

```
Actual dir. perms:      rwxr--r--
```

在清单 15-5 中，我们使用 `mkdir()` 和 `rmdir()` 系统调用来创建和删除一个目录，使用 `unlink()` 系统调用来删除一个文件。我们在[第 18 章](#)讨论这些系统调用。

清单 15-5：使用 `umask()`

```
-----files/t_umask.c
#include <sys/stat.h>
#include <fcntl.h>
#include "file_perms.h"
#include "tlpi_hdr.h"

#define MYFILE "myfile"
#define MYDIR "mydir"
#define FILE_PERMS (S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP)
#define DIR_PERMS (S_IRWXU | S_IRWXG | S_IRWXO)
#define UMASK_SETTING (S_IWGRP | S_IXGRP | S_IWOTH | S_IXOTH)

int
main(int argc, char *argv[])
{
    int fd;
    struct stat sb;
    mode_t u;

    umask(UMASK_SETTING);

    fd = open(MYFILE, O_RDWR | O_CREAT | O_EXCL, FILE_PERMS);
    if (fd == -1)
        errExit("open-%s", MYFILE);
    if (mkdir(MYDIR, DIR_PERMS) == -1)
        errExit("mkdir-%s", MYDIR);
```

```

u = umask(0); /* Retrieves (and clears) umask value */

if (stat(MYFILE, &sb) == -1)
    errExit("stat-%s", MYFILE);
printf("Requested file perms: %s\n", filePermStr(FILE_PERMS, 0));
printf("Process umask: %s\n", filePermStr(u, 0));
printf("Actual file perms: %s\n\n", filePermStr(sb.st_mode, 0));

if (stat(MYDIR, &sb) == -1)
    errExit("stat-%s", MYDIR);
printf("Requested dir. perms: %s\n", filePermStr(DIR_PERMS, 0));
printf("Process umask: %s\n", filePermStr(u, 0));
printf("Actual dir. perms: %s\n", filePermStr(sb.st_mode, 0));

if (unlink(MYFILE) == -1)
    errMsg("unlink-%s", MYFILE);
if (rmdir(MYDIR) == -1)
    errMsg("rmdir-%s", MYDIR);

exit(EXIT_SUCCESS);
}
-----files/t_umask.c

```

15.4.7 修改文件权限: chmod() 和 fchmod()

chmod() 和 fchmod() 系统调用修改文件的权限。

```

#include <sys/stat.h>

int chmod(const char *pathname, mode_t mode);

#define _XOPEN_SOURCE 500 /* Or: #define _BSD_SOURCE */
#include <sys/stat.h>

int fchmod(int fd, mode_t mode);

```

成功时都返回 0; 出错时都返回 -1

chmod() 系统调用修改 pathname 指定文件的权限，如果 pathname 是一个符号链接，则先进行解引用，修改链接所引用文件的权限，而不是符号链接本身。（符号链接创建时，链接文件本身总是启用所有用户的读、写、执行权限，并且

这些权限不能被修改。在解引用符号链接时，将忽略这些权限)。

`fchmod()`系统调用修改打开文件描述符 `fd` 所引用文件的权限。

`mode` 参数指定文件的新权限，可以是数值（八进制），或者表 15-4 中常量“或”所得的掩码。要修改文件的权限，要么进程拥有特权（`CAP_FOWNER`），要么进程的有效用户 ID 匹配文件的拥有者（用户 ID）。(严格地说，在 Linux 中，非特权进程的文件系统用户 ID 必须匹配文件的用户 ID，参考 [9.5 节](#))。

以下代码设置所有用户对文件只拥有读权限：

```
if (chmod("myfile", S_IRUSR | S_IRGRP | S_IROTH) == -1)
    errExit("chmod");
/* Or equivalently: chmod("myfile", 0444); */
```

要修改文件权限的某几个位，我们首先使用 `stat()`获取文件的现有权限，按我们的要求修改其中的位，然后使用 `chmod()`来更新文件的权限：

```
struct stat sb;
mode_t mode;

if (stat("myfile", &sb) == -1)
    errExit("stat");
mode = (sb.st_mode | S_IWUSR) & ~S_IROTH;
/* owner-write on, other-read off, remaining bits unchanged */
if (chmod("myfile", mode) == -1)
    errExit("chmod");
```

上面代码的作用等同于以下 shell 命令：

```
$ chmod u+w,o-r myfile
```

在 [15.3.1 节](#)，我们提到过如果目录存在于以`-o bsdgroups` 选项挂载的 ext2 文件系统中；或者以`-o sysvgroups` 选项挂载，并且目录的设置组 ID 权限位打开，则目录中新创建的文件从父目录得到组所有权，而不是创建进程的有效组 ID。因此这个文件的组 ID 可能不匹配创建进程的任何组 ID。由于这个原因，当非特权进程（不拥有 `CAP_FSETID` 能力）调用 `chmod()`（或 `fchmod()`）修改文件权限，而文件的组 ID 不匹配调用进程的有效组 ID、任何附加组 ID 时，内核总是清除文件的设置组 ID 权限位。这是一个安全性的措施，防止用户创建自己不是成员的组

的设置组 ID 程序。下面 shell 命令显示系统阻止了这一未遂企图：

```
$ mount | grep test          Hmmm, /test is mounted with -o bsdgroups
/dev/sda9 on /test type ext3 (rw,bsdgroups)
$ ls -ld /test               Directory has GID root, writable by anyone
drwxrwxrwx 3 root root 4096 Jun 30 20:11 /test
$ id                         I'm an ordinary user, not part of root group
uid=1000(mtk) gid=100(users) groups=100(users),101(staff),104(teach)
$ cd /test
$ cp ~/myprog .              Copy some mischievous program here
$ ls -l myprog               Hey! It's in the root group!
-rwxr-xr-x 1 mtk root 19684 Jun 30 20:43 myprog
$ chmod g+s myprog          Can I make it set-group-ID to root?
$ ls -l myprog               Hmm, no...
-rwxr-xr-x 1 mtk root 19684 Jun 30 20:43 myprog
```

15.5 i-node 标志 (ext2 扩展文件属性)

某些 Linux 文件系统允许对文件和目录设置各种 i-node 标志。这个特性是非标准的 Linux 扩展。

现代 BSD 也提供类似的 i-node 标志特性，可以使用 chflags 命令和 chflags() 系统调用进行设置。

第一个支持 i-node 标志的 Linux 文件系统是 ext2，因此这些标志有时候也称为 ext2 扩展文件属性。随后，其它文件系统也增加了 i-node 标志的支持，包括 Btrfs, ext3, ext4, Reiserfs (Linux 2.4.19 起), XFS (Linux 2.4.25 和 2.6 起)，以及 JFS (Linux 2.6.17 起)。

不同文件系统对 i-node 标志的支持范围存在差异。要在 Reiserfs 文件系统中使用 i-node 标志，我们必须使用 mount -o attrs 选项来挂载文件系统。

在 shell 中，我们可以使用 chattr 和 lsattr 命令来设置和查看 i-node 标志，如下面例子所示：

```
$ lsattr myfile
----- myfile
$ chattr +ai myfile      打开 Append Only 和 Immutable 标志
$ lsattr myfile
---ia-- myfile
```

在程序中，可以使用 `ioctl()` 系统调用来获取和修改 i-node 标志，我们马上就会详细描述。

i-node 标志可以对普通文件和目录进行设置。多数 i-node 标志作用于普通文件，少数则也可用于目录。表 15-6 汇总了可用的 i-node 标志，显示了相应的标志名（`<linux/fs.h>` 中定义），用于程序中调用 `ioctl()`；表中还列出了 `chattr` 命令使用的选项字母。

在 Linux 2.6.19 之前，表 15-6 中显示的 `FS_*` 常量没有定义在 `<linux/fs.h>` 中。相反有一组文件系统特定的头文件，定义了文件系统特定的常量名，不过都是相同的值。因此 ext2 的 `EXT2_APPEND_FL` 定义在 `<linux/ext2_fs.h>` 中；Reiserfs 的 `REISERFS_APPEND_FL` 则定义在 `<linux/reiser_fs.h>` 中，但是值都是一样的，依次类推。由于各个头文件定义的相应常量都是相同的值，在不提供 `<linux/fs.h>` 的老系统中，我们可以包含文件系统特定的头文件，并使用文件系统特定的名字。

表 15-6: i-node 标志

常量	<code>chattr</code> 选项	用途
<code>FS_APPEND_FL</code>	<code>a</code>	Append only (要求特权)
<code>FS_COMPR_FL</code>	<code>c</code>	启用文件压缩 (未实现)
<code>FS_DIRESYNC_FL</code>	<code>D</code>	同步目录更新 (Linux 2.6 起)
<code>FS_IMMUTABLE_FL</code>	<code>i</code>	Immutable 不可变 (要求特权)
<code>FS_JOURNAL_DATA_FL</code>	<code>j</code>	启用数据日志 (要求特权)
<code>FS_NOATIME_FL</code>	<code>A</code>	不更新文件的最后访问时间
<code>FS_NODUMP_FL</code>	<code>d</code>	不 Dump
<code>FS_NOTAIL_FL</code>	<code>t</code>	不尾部填充
<code>FS_SECRM_FL</code>	<code>s</code>	安全删除 (未实现)
<code>FS_SYNC_FL</code>	<code>S</code>	同步文件 (和目录) 更新
<code>FS_TOPDIR_FL</code>	<code>T</code>	作为 Orlov 的顶层目录 (Linux 2.6 起)
<code>FS_UNRM_FL</code>	<code>u</code>	文件可以被撤消删除 (未实现)

这些 `FL_*` 标志的含义如下：

FS_APPEND_FL

如果指定了 `O_APPEND` 标志，该文件可以打开为只能写入，并且强制所有文件更新添加到文件的末尾。这个标志可以很好地用于日志文件。只有特权（`CAP_LINUX_IMMUTABLE`）进程才能设置这个标志。

FS_COMPR_FL

以压缩格式存储文件到磁盘。没有哪个主流的 Linux 本地文件系统在标准版中实现了这个特性（不过 `ext2` 和 `ext3` 有扩展包实现了这个特性）。虽然可以降低磁盘存储的消耗，但是 CPU 在压缩和解压时会引入额外的开销，而且压缩文件也意味着我们随机访问文件内容时（通过 `lseek()`），不再是一件简单的事情。文件压缩对于多数应用都是不需要的。

FS_DIRSYNC_FL (Linux 2.6 起)

使目录更新（如 `open(pathname, O_CREAT)`, `link()`, `unlink()`, `mkdir()`）同步。类似于 [13.3 节](#)讨论的同步文件更新机制。和同步文件更新一样，同步目录更新也会导致相关的性能影响。这个设置只能应用于目录。（[14.8.1 节](#)讨论的 `MS_DIRSYNC` 挂载标志也提供类似的功能，但只能应用于单次挂载）。

FS_IMMUTABLE_FL

使文件不可变。不能更新文件数据（`write()` 和 `truncate()`），同时也阻止元数据更新（`chmod()`, `chown()`, `unlink()`, `link()`, `rename()`, `rmdir()`, `utime()`, `setxattr()`, `removexattr()`）。只有特权进程（`CAP_LINUX_IMMUTABLE`）才能为文件设置这个标志。当这个标志被设置时，即使是特权进程也不能修改文件的内容和元数据。

FS_JOURNAL_DATA_FL

启用数据日志。这个标志只有 `ext3` 和 `ext4` 文件系统才支持。这两个文件系统提供三种级别的日志：`journal`, `ordered`, `writeback`。所有模式的日志都更新文件元数据，但是 `journal` 模式还额外增加了文件数据的更新。在使用 `ordered` 和 `writeback` 模式日志的文件系统中，特权进程（`CAP_SYS_RESOURCE`）可以使用这个标志，以单个文件的粒度启用数据更新日志。（`mount(8)` 手册页讨论了 `ordered` 和 `writeback` 这两种模式之间的区别）。

FS_NOATIME_FL

在文件被访问时，不更新文件最后访问时间。这消除了每次访问文件都需要更新文件 i-node 信息的开销，因此提高了 I/O 性能（参考 [14.8.1 节](#) 关于 MS_NOATIME 标志的讨论）。

FS_NODUMP_FL

备份模式下使用 `dump(8)` 命令时不包括这个文件。这个标志的作用依赖于 `dump(8)` 命令的 -h 选项，请参考 `dump(8)` 手册页。

FS_NOTAIL_FL

禁止尾部填充。这个标志只有 Reiserfs 文件系统支持。它禁止 Reiserfs 的尾部填充特性，这个特性填充小文件（以及大文件的最后碎片）和文件元数据到相同的磁盘块。尾部填充还可以使用 Reiserfs 文件系统的 `mount -notail` 挂载选项来禁止。

FS_SECRM_FL

安全地删除文件。这个未实现的特性原本的意图是，当删除文件时，使用安全删除技术，也就是首先进行覆盖，来阻止磁盘扫描程序读取或重建该文件。（真正安全的删除文件非常复杂：实际上需要多次写入磁性媒介，来安全地擦除之前记录的数据；参考[Gutmann, 1996]）。

FS_SYNC_FL

使文件更新同步。当应用于文件时，这个标志将导致文件写入同步（等同于这个文件的所有 `open` 都使用 `O_SYNC` 标志）。当应用于目录时，这个标志和上面讨论的同步目录更新标志的作用相同。

FS_TOPDIR_FL (Linux 2.6 起)

在 Orlov 块分配策略下，对该目录进行特殊对待。Orlov 策略是来自于 BSD 的灵感，用于修改 ext2 块分配策略，试图提高相关文件（如处于同一目录下的多个文件）的机率，并将这些文件存放在磁盘的相邻位置，这样就可以改进磁盘寻址时间。细节请参考 [Corbet, 2002] 和 [Kumar, et al. 2008]。

`FS_TOPDIR_FL` 标志只对 ext2 及后续文件系统有效，包括 ext3 和 ext4。

FS_UNRM_FL

如果文件被删除，允许恢复（反删除）该文件。这个特性并没有实现，因为我们完全可以在内核外面实现文件恢复机制。

通常，当 i-node 标志设置于目录时，会自动被该目录下新创建的文件和子目录继承。下面是一些例外的情况：

- FS_DIRSYNC_FL (chattr +D) 标志，只能应用于目录，而且只被目录下新创建的子目录继承。
- FS_IMMUTABLE_FL (chattr +i) 标志应用于目录时，目录下新创建的文件和子目录不会继承该标志，因为这个标志阻止往目录中增加新的项目。

在程序中，我们可以使用 ioctl() 的 FS_IOC_GETFLAGS 和 FS_IOC_SETFLAGS 操作来获取和修改 i-node 标志。（这些常量定义在<linux/fs.h>中）。下面代码显示了如何对打开文件描述符 fd 引用的文件启用 FS_NOATIME_FL 标志：

```
int attr;

if (ioctl(fd, FS_IOC_GETFLAGS, &attr) == -1) /* 获取当前标志 */
    errExit("ioctl");
attr |= FS_NOATIME_FL;
if (ioctl(fd, FS_IOC_SETFLAGS, &attr) == -1) /* 更新标志 */
    errExit("ioctl");
```

要修改一个文件的 i-node 标志，要么进程的有效用户 ID 必须匹配文件的用户 ID（拥有者）；要么进程必须拥有特权 (CAP_FOWNER)。（精确地说，在 Linux 中，非特权进程的文件系统用户 ID，而不是有效用户 ID，必须匹配文件的用户 ID，[9.5 节](#)已经对此详细描述）。

15.6 小结

stat() 系统调用可以获取文件的信息（元数据），多数信息从文件的 i-node 中得到。包括：文件所有权、文件权限、文件时间戳。

程序可以使用 `utime()`, `utimes()` 和许多类似的接口, 来更新文件的最后访问时间和最后修改时间。

每个文件都有一个相关联的用户 ID (拥有者) 和组 ID, 以及一组权限位。文件权限按用户类型分为三类: 拥有者 (用户)、组、其它。每一类用户都授予了三种权限: 读、写、执行。目录同样也是采用这种权限分配方式, 但是权限位的含义稍有不同。`chown()` 和 `chmod()` 系统调用修改文件的所有权和权限。`umask()` 系统调用设置权限掩码, 在调用进程创建文件时总是关闭某些权限位。

文件和目录还有另外三个额外的权限位。设置用户 ID、设置组 ID 权限位可以应用于程序文件, 执行程序可以获得不同的有效用户或组标识的特权 (程序文件的拥有者)。对于目录来说, 如果文件系统以 `nogrpid(sysvggroups)` 选项挂载, 设置组 ID 权限位可以用来控制目录中新创建的文件从进程的有效组 ID 或父目录的组 ID 继承组所有权。**Sticky** 权限位应用于目录时, 作用是“限制删除标志”。

i-node 标志控制文件和目录的各种行为。尽管最早由 `ext2` 定义, 这些标志现在已经支持许多其它的文件系统。

15.7 习题

15-1. 前面 [15.4 节](#) 讨论过各种文件系统操作要求的权限。使用 shell 命令或者编写程序来验证或回答以下问题:

- a) 移除文件拥有者的所有权限, 将拒绝文件拥有者访问该文件, 但是组和其它人却可以访问。
- b) 目录拥有读权限, 但是没有执行权限, 我们可以列出目录下的文件名列表, 但是文件本身不能被访问, 无论怎样设置文件的权限。
- c) 要创建新文件、打开文件读取、打开文件写入、删除文件, 分别需要父目录和文件本身的哪些权限? 重命名 (`rename`) 一个文件, 分别需要源和目标目录的什么权限? 如果 `rename` 操作的目标文件已经存在, 我们需要那个文件的什么权限? 设置目录的 **Sticky** 权限位 (`chmod +t`), 是怎样影响 `rename` 和删除操作的呢?

15-2. `stat()` 系统调用会修改文件的三个时间戳吗？如果不会，请解释为什么。

15-3. 在运行 Linux 2.6 的系统中，修改清单 15-1 (`t_stat.c`) 程序，使它以纳秒精度显示文件的时间戳。

15-4. `access()` 系统调用使用进程的实际用户和组 ID 来检查权限。编写一个相应的函数，根据进程的有效用户和组 ID 来执行权限检查。

15-5. 如 [15.4.6 节](#) 所述，`umask()` 总是设置进程的 `umask`，并且同时返回老 `umask`。我们怎样能获得进程的当前 `umask`，而又不修改它呢？

15-6. `chmod a+rX file` 命令启用所有三种类型用户的读权限。如果 `file` 是一个目录，或者 `file` 任一类型用户的执行权限已启用，则启用所有类型用户的执行权限，如下面例子所示：

```
$ ls -ld dir file prog
dr----- 2 mtk users 48      May 4 12:28 dir
-r----- 1 mtk users 19794 May 4 12:22 file
-r-x----- 1 mtk users 19336 May 4 12:21 prog
$ chmod a+rX dir file prog
$ ls -ld dir file prog
dr-xr-xr-x 2 mtk users 48      May 4 12:28 dir
-r--r--r-- 1 mtk users 19794 May 4 12:22 file
-r-xr-xr-x 1 mtk users 19336 May 4 12:21 prog
```

编写一个程序，使用 `stat()` 和 `chmod()` 来执行 `chmod a+rX` 等价的操作。

15-7. 编写一个简单版本的 `chattr(1)` 命令，能够修改文件的 i-node 标志。参考 `chattr(1)` 手册页获得 `chattr` 命令行接口的详细信息。（你不需要实现 `-R`, `-V`, `-v` 选项）。

第 16 章 扩展属性

本章讨论扩展属性 (EA)，允许任意元数据以 name-value 成对的形式，关联到文件的 i-node。EA 由 Linux 2.6 版本引入。

16.1 概述

EA 主要用于实现访问控制列表（[第 17 章](#)）和文件能力（[第 39 章](#)）。但是 EA 的设计非常通用，也可以直接用于其它用途。例如可以使用 EA 来记录文件版本号、MIME 类型信息、文件字符集、以及图形化图标（一个指针）。

SUSv3 没有规定 EA。但是有一些其它 UNIX 实现也提供类似的特性，例如现代 BSD（参考 `extattr(2)`）和 Solaris 9 及之后版本（参考 `fsattr(5)`）。

EA 要求底层文件系统提供支持。目前 Btrfs、ext2、ext3、ext4、JFS、Reiserfs、XFS 都提供对 EA 的支持。

文件系统支持 EA 特性是可选的，并且可以通过内核配置选项的文件系统菜单进行控制。Linux 2.6.7 开始在 Reiserfs 上支持 EA。

EA 命名空间

EA 的名字格式为 `namespace.name`。其中 `namespace` 用于区分 EA 为独立的功能分类。`name` 则在给定的 `namespace` 内部唯一地标识一个 EA。

`namesapce` 支持四个值：`user`, `trusted`, `system`, `security`。这四种类型 EA 的使用规则如下：

- `User` EA 可以被非特权进程操作，并服从于文件权限检查机制：要获取一个 `user` EA 的值要求拥有文件的读权限；修改一个 `user` EA 的值要求拥有文件的写权限。（缺少权限将导致 `EACCES` 错误）。要在 `ext2`, `ext3`, `ext4`, `Reiserfs` 文件系统中关联一个文件的 `user` EA，底层文件系统必须以 `user_xattr` 选项挂载：

```
$ mount -o user_xattr device directory
```

- `Trusted` EA 和 `user` EA 一样也可以被用户进程操作。二者的区别在于进程必须拥有特权（`CAP_SYS_ADMIN`）才能操作 `Trusted` EA。

- System EA 由内核使用，关联系统对象到文件。当前唯一支持的对象类型是访问控制列表（[第 17 章](#)）。
- Security EA 用于操作系统安全模块存储文件安全标签，并且关联能力到可执行文件（[39.3.2 节](#)）。Security EA 最初设计用于支持“增强安全的 Linux”（SELinux，<http://www.nsa.gov/research/selinux/>）。

在相同命名空间或不同命名空间下，一个 i-node 都可能拥有多个关联的 EA。不同命名空间下的 EA 名是相互独立的。在 user 和 trusted 命名空间下，EA 名可以是任意字符串。在 system 命名空间中，则只允许内核明确允许的名字（例如那些用于访问控制列表的名字）。

JFS 支持另一个命名空间：os2，但是其它文件系统没有实现。os2 命名空间用于支持遗留的 OS/2 文件系统 EA。进程创建 os2 EA 并不需要特权。

shell 中创建和查看 EA

在 shell 中，我们可以使用 setfattr(1) 和 getfattr(1) 命令来设置和查看某个文件的 EA：

```
$ touch tfile
$ setfattr -n user.x -v "The past is not dead." tfile
$ setfattr -n user.y -v "In fact, it's not even past." tfile
$ getfattr -n user.x tfile          获取单个 EA 的值
# file: tfile                    getfattr 获取的信息
user.x="The past is not dead."   getfattr 命令在每个文件属性之后
                                  打印一个空行

$ getfattr -d tfile              打印所有 user EA 的值
# file: tfile
user.x="The past is not dead."
user.y="In fact, it's not even past."


$ setfattr -n user.x tfile      修改 EA 的值为空字符串
$ getfattr -d tfile
# file: tfile
user.x
user.y="In fact, it's not even past."
```

```
$ setfattr -x user.y tfile      删除一个 EA  
$ getfattr -d tfile  
# file: tfile  
user.x
```

上面 shell 会话显示的一个要点是 EA 的值可以是空字符串，这和 EA 未定义是完全不一样的。（在上面 shell 会话结束时，user.x 的值是空字符串，而 user.y 则未定义）。

默认 getfattr 只列出 user EA 的值。-m 选项可以用来指定一个正则表达式，选择我们想要显示的 EA 名：

```
$ getfattr -m 'pattern' file
```

pattern 的默认值是 '^user\.'，我们可以使用如下命令来列出某个文件的所有 EA：

```
$ getfattr -m - file
```

16.2 扩展属性实现细节

在这一节，我们扩充前面小节的概述，描述 EA 实现的一些具体细节。

user 扩展属性的限制

我们只能对文件和目录设置 user EA。其它文件类型由于以下原因被排除在外：

- 对于符号链接，所有用户的所有权限都已经启用，而且这些权限不能修改（符号链接的权限在 Linux 上没有任何意义，[18.2 节](#)有详细描述）。这意味着权限机制不能阻止任何用户放置 EA 到符号链接上。因此解决办法就是禁止所有用户在符号链接上创建 user EA。
- 对于设备文件、socket、FIFO，权限控制用户对设备的访问，授予用户对底层对象执行 I/O 的权限。操作这些权限来控制用户创建 user EA 会与这个目标产生冲突。

此外，如果目录的 Sticky 位（[15.4.5 节](#)）已经设置，而非特权进程不是该目

录的拥有者，则不能放置 user EA 到该目录上。这样阻止任意用户添加 EA 到类似于/tmp 这样的目录，这些目录是公共可写的（因此也允许任意用户操作目录的 EA），设置 Sticky 位就可以阻止用户删除其它用户的文件。

实现限制

Linux VFS 对所有文件系统的 EA 强加了以下限制：

- EA 名的长度限制为 255 字符。
- EA 值限制为 64KB。

此外，某些文件系统还强加了更多限制，包括 EA 大小、和一个文件能够关联的 EA 数量：

- 在 ext2, ext3, ext4 中，一个文件的所有 EA 的名字和值占用的总字节数，限制为单个逻辑磁盘块的大小 ([14.3 节](#))：1024、2048、4096 字节。
- 在 JFS 中，一个文件的所有 EA 的名字和值占用的总字节数上限为 128KB。

16.3 操作扩展属性的系统调用

在这一节，我们讨论更新、获取、删除 EA 的系统调用。

创建和修改 EA

setxattr(), lsetxattr(), fsetxattr() 系统调用可以设置一个文件 EA 的值。

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

int setxattr(const char *pathname, const char *name, const void *value,
             size_t size, int flags);
int lsetxattr(const char *pathname, const char *name, const void *value,
              size_t size, int flags);
int fsetxattr(int fd, const char *name, const void *value,
              size_t size, int flags);
```

成功时都返回 0；出错时都返回 -1

这三个系统调用的区别类似于 `stat()`, `lstat()`, `fstat()` ([15.1 节](#)):

- `setxattr()`通过 `pathname` 标识一个文件，如果是符号链接，则解引用；
- `lsetxattr()`通过 `pathname` 标识一个文件，但是不对符号链接解引用；
- `fsetxattr()`通过打开文件描述符 `fd` 来标识一个文件。

同样的区别也适用于本节随后讨论的其它系统调用组。

`name` 参数是一个 `null` 终止的字符串，定义了 EA 的名字。`value` 参数是一个缓冲区指针，定义了 EA 的新值。`size` 参数则指定了这个缓冲区的大小。

默认情况下，如果 `name` 指定的 EA 不存在，这些系统调用会创建一个新的 EA；如果指定的 EA 已存在，就替换它的值。`flags` 参数提供对这个行为更精确的控制。它可以指定为 0 使用默认行为，或者使用以下常量：

XATTR_CREATE

如果 `name` 指定的 EA 已经存在，则失败 (EEXIST)。

XATTR_REPLACE

如果 `name` 指定的 EA 不存在，则失败 (ENODATA)。

下面是使用 `setxattr()` 创建一个 user EA 的示例代码：

```
char *value;
value = "The past is not dead.";
if (setxattr(pathname, "user.x", value, strlen(value), 0) == -1)
    errExit("setxattr");
```

获取一个 EA 的值

`getxattr()`, `lgetxattr()`, `fgetxattr()` 系统调用获取一个 EA 的值。

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

ssize_t getxattr(const char *pathname, const char *name, void *value,
                 size_t size);
ssize_t lgetxattr(const char *pathname, const char *name, void *value,
                 size_t size);
```

```
ssize_t fgetxattr(int fd, const char *name, void *value,  
                  size_t size);
```

成功时都返回 EA 值的大小（非负）；出错时都返回 -1

`name` 参数是一个 null 终止的字符串，标识要获取其值的 EA。EA 值返回在 `value` 指向的缓冲区。这个缓冲区必须由调用方分配，而且长度需在 `size` 中指定。成功时，这些系统调用返回复制到 `value` 中的字节数。

如果文件没有名为 `name` 的属性，这些系统调用以错误 ENODATA 失败。如果 `size` 太小，这些系统调用以错误 ERANGE 失败。

`size` 也可以指定为 0，在这种情况下 `value` 将被忽略，但系统调用仍会返回 EA 值的大小。这就提供了一种机制，让我们确定随后实际获取 EA 值时，`value` 缓冲区的精确大小。但是请注意，我们仍然不能保证返回的大小在随后获取 EA 值时是足够大的，因为其它进程可能在同一时间给 EA 赋予更大的值，甚至完全删除这个 EA。

删除一个 EA

`removexattr()`, `lremovexattr()`, `fremovexattr()` 系统调用从文件中删除一个 EA。

```
#include <sys/xattr.h>  
  
int removexattr(const char *pathname, const char *name);  
int lremovexattr(const char *pathname, const char *name);  
int fremovexattr(int fd, const char *name);
```

成功时都返回 0；出错时都返回 -1

`name` 是 null 终止的字符串，标识了要删除的 EA。删除不存在的 EA 将以错误 ENODATA 失败。

获取一个文件关联的所有 EA 名

`listxattr()`, `llistxattr()`, `flistxattr()` 系统调用返回一个列表，包含文件相关联的所有 EA 名。

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

ssize_t listxattr(const char *pathname, char *list, size_t size);
ssize_t llistxattr(const char *pathname, char *list, size_t size);
ssize_t flistxattr(int fd, char *list, size_t size);

成功时都返回复制到 list 中的字节数；出错时都返回-1
```

EA 名列表返回为一系列 `null` 终止的字符串，存储在 `list` 指向的缓冲区中。缓冲区的大小需用 `size` 指定。成功时，这些系统调用返回复制到 `list` 中的字节数。

和 `getxattr()`一样，`size` 指定为 0 也是可以的，这种情况下忽略 `list`，但系统调用仍然会返回存储 EA 名列表所需的缓冲区大小，随后的调用可以使用这个大小来分配缓冲区（假设 EA 列表没有变化）。

要获取一个文件关联的 EA 名列表，我们只需要能访问文件即可（也就是拥有 `pathname` 中所有目录的执行权限）。不要求拥有文件本身的所有权限。

由于安全的原因，`list` 中返回的 EA 名列表可能会排除那些调用进程无权访问的属性。例如非特权进程调用 `listxattr()` 时，多数文件系统会忽略 `trusted` 属性。不过注意前面说的是“可能”，意味着文件系统并不一定会这样实现。因此我们获得 EA 名列表后，随后调用 `getxattr()` 获取 EA 的值时，可能会由于进程没有权限而失败。（类似的失败，也可能在另一个进程删除某个属性时发生）。

示例程序

清单 16-1 中的程序获取和显示文件的所有 EA 名和值，文件由命令行指定。对于每个文件，程序使用 `listxattr()` 获取所有的 EA 名，然后循环调用 `getxattr()`，来获取相应的 EA 值。默认情况下，属性值显示为纯文本。如果指定了 `-x` 选项，则属性值将显示为十六进制字符串。下面 shell 会话日志演示了这个程序的使用：

```
$ setfattr -n user.x -v "The past is not dead." tfile
$ setfattr -n user.y -v "In fact, it's not even past." tfile
$ ./xattr_view tfile
tfile:
    name=user.x; value=The past is not dead.
    name=user.y; value=In fact, it's not even past.
```

清单 16-1：显示文件扩展属性

```
-----xattr/xattr_view.c
#include <sys/xattr.h>
#include "tlpi_hdr.h"

#define XATTR_SIZE 10000

static void
usageError(char *progName)
{
    fprintf(stderr, "Usage: %s [-x] file...\n", progName);
    exit(EXIT_FAILURE);
}

int
main(int argc, char *argv[])
{
    char list[XATTR_SIZE], value[XATTR_SIZE];
    ssize_t listLen, valueLen;
    int ns, j, k, opt;
    Boolean hexDisplay;

    hexDisplay = 0;
    while ((opt = getopt(argc, argv, "x")) != -1) {
        switch (opt) {
        case 'x': hexDisplay = 1; break;
        case '?': usageError(argv[0]);
        }
    }

    if (optind >= argc + 2)
        usageError(argv[0]);

    for (j = optind; j < argc; j++) {
        listLen = listxattr(argv[j], list, XATTR_SIZE);
        if (listLen == -1)
            errExit("listxattr");

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

        /* Loop through all EA names, displaying name + value */
    }
}
```

```
for (ns = 0; ns < listLen; ns += strlen(&list[ns]) + 1) {
    printf(" name=%s; ", &list[ns]);

    valueLen = getxattr(argv[j], &list[ns], value, XATTR_SIZE);
    if (valueLen == -1) {
        printf("couldn't get value");
    } else if (!hexDisplay) {
        printf("value=%.*s", (int) valueLen, value);
    } else {
        printf("value=");
        for (k = 0; k < valueLen; k++)
            printf("%02x ", (unsigned int) value[k]);
    }

    printf("\n");
}

printf("\n");
}

exit(EXIT_SUCCESS);
}
-----xattr/xattr_view.c
```

16.4 小结

从 2.6 版本开始，Linux 支持扩展属性，允许任意元数据关联到一个文件，元数据格式为 name-value。

16.5 习题

- 16-1. 编写一个程序，可以创建和修改一个文件的 user EA（简单版本的 setfattr 命令）。文件名和 EA 名、EA 值通过程序的命令行参数指定。

第 17 章 访问控制列表

[15.4 节](#)讨论了传统的 UNIX（和 Linux）文件权限机制。对于许多应用来说，这个机制已经足够。但是某些应用可能需要更加精确的权限控制，授予特定用户和组特定的权限。为了满足这个需求，许多 UNIX 系统对传统的 UNIX 文件权限模型实现了一个扩展，称为访问控制列表（ACL）。ACL 允许文件权限指定为单个用户和单个组，而且用户和组的数量无限。Linux 从内核 2.6 开始提供 ACL。

文件系统支持 ACL 是可选的，而且由内核配置选项的文件系统菜单控制。Reiserfs 从内核 2.6.7 开始才支持 ACL。

要在 ext2, ext3, ext4, Reiserfs 文件系统中创建 ACL，文件系统必须以 mount -o acl 选项挂载。

ACL 并未被 UNIX 系统正式标准化。POSIX.1e 和 POSIX.2c 标准草案试图对 ACL 进行标准化，目标分别是为 ACL 规定应用编程接口（API）和 shell 命令接口（以及其他特性，如能力）。不过最终这个标准化尝试失败了，而这些标准草案也被撤销。无论如何，许多 UNIX 实现（包括 Linux）的 ACL 实现都基于这些标准草案（通常是最后一个版本，Draft 17）。但是由于各个 ACL 实现之间存在许多差异（部分原因是标准草案的不完整性），编写可移植的 ACL 程序也存在一定的困难。

本章描述 ACL 及其使用，同时还讨论了一些操作和获取 ACL 的库函数。我们不会详细地讨论所有这些函数，因为数量实在是太多了（细节请参考手册页）。

17.1 概述

ACL 由一系列的 ACL 条目组成，每个条目定义了单个用户或组的文件权限（参考图 17-1）。

Tag type	Tag qualifier	Permissions
ACL_USER_OBJ	-	rwx
ACL_USER	1007	r--
ACL_USER	1010	rwx
ACL_GROUP_OBJ	-	rwx
ACL_GROUP	102	r--
ACL_GROUP	103	-w-
ACL_GROUP	109	--x
ACL_MASK	-	rwx
ACL_OTHER	-	r--

图 17-1：一个访问控制列表

ACL 条目

每个 ACL 条目包含以下组成部分：

- tag type (标签类型)，表示该条目应用于某个用户、某个组、或者其它用户类型；
- 可选的 tag qualifier (标签限定)，标识某个特定的用户或组（如用户 ID 或组 ID）；
- permission set (权限集)，指定授予该条目的权限（读、写、执行）。

标签类型可以是以下值之一：

ACL_USER_OBJ

这个条目规定授予文件拥有者的权限。每个 ACL 有且只有一个 ACL_USER_OBJ 条目。这个条目对应于传统的文件拥有者 (user) 权限。

ACL_USER

这个条目规定 tag qualifier 标识的用户的权限。每个 ACL 可以包含零个或多个 ACL_USER 条目，但是一个特定的用户最多只能定义一个 ACL_USER 条目。

ACL_GROUP_OBJ

这个条目规定了授予文件组的权限。每个 ACL 有且只有一个 ACL_GROUP_OBJ

条目。这个条目对应于传统的文件组 (group) 权限，除非 ACL 还包含一个 `ACL_MASK` 条目。

`ACL_GROUP`

这个条目规定了 `tag qualifier` 所标识的组的权限。每个 ACL 可以包含零个或多个 `ACL_GROUP` 条目，但是某个特定组最多只能定义一个 `ACL_GROUP` 条目。

`ACL_MASK`

这个条目规定了 `ACL_USER`、`ACL_GROUP_OBJ`、`ACL_GROUP` 能够获得的最大权限。每个 ACL 最多只能定义一个 `ACL_MASK` 条目。如果 ACL 包含 `ACL_USER` 或 `ACL_GROUP` 条目，则 `ACL_MASK` 条目强制必须定义。我们马上就会详细地讨论这个标签类型。

`ACL_OTHER`

这个条目规定了不匹配已定义 ACL 条目的其它所有用户的权限。每个 ACL 有且只有一个 `ACL_OTHER` 条目。这个条目对应于传统的 `other` 文件权限。

只有 `ACL_USER` 和 `ACL_GROUP` 条目才能使用 `tag qualifier` (标签限定)，后者指定了某个特定的用户 ID 或组 ID。

最小和扩展 ACL

最小 ACL 在语义上等价于传统的文件权限集，它包含且只包含三个条目：`ACL_USER_OBJ`、`ACL_GROUP_OBJ`、`ACL_OTHER`。扩展 ACL 则是额外包含 `ACL_USER`、`ACL_GROUP`、`ACL_MASK` 条目的 ACL。

区分最小 ACL 和扩展 ACL 是有原因的：首先后者对传统权限模型提供了语义扩展；其次则与 Linux 的 ACL 实现有关，ACL 实现为 system 扩展属性 ([第 16 章](#))。用于维护文件 ACL 的扩展属性命名为 `system.posix_acl_access`。这个扩展属性只在文件拥有扩展 ACL 时才需要。最小 ACL 包含的权限信息可以直接存储在传统的文件权限位中（实际上也是这么做的）。

17.2 ACL 权限检查算法

文件拥有 ACL 时，在特定情况下（和传统文件权限模型一样，[15.4.3 节](#)），也需要对文件执行权限检查。权限检查按以下顺序进行，除非匹配了某个条件：

1. 如果进程拥有特权，允许所有访问权限。这句话有一个例外，类似于[15.4.3 节](#)讨论的传统权限模型。当执行一个文件时，只有至少一个 ACL 条目启用了执行权限，特权进程才能获得文件的执行权限。
2. 如果进程的有效用户 ID 匹配文件的拥有者（用户 ID），则进程获得 ACL_USER_OBJ 条目规定的权限。（严格地说，在 Linux 中，实际检查的是进程的文件系统 ID，而不是进程的有效 ID，如[9.5 节](#)所述）。
3. 如果进程的有效用户 ID 匹配某个 ACL_USER 条目的 tag qualifier，则进程获得该条目规定的权限，但是必须首先和 ACL_MASK 条目的值进行掩码运算（“与”）。
4. 如果进程的某个组 ID（有效组 ID 或任意附加组 ID）匹配文件组（对于 ACL_GROUP_OBJ 条目），或者匹配任意 ACL_GROUP 条目的 tag qualifier，则按以下规则继续检查，直到匹配某条规则：
 - a) 如果进程的某个组 ID 匹配文件的组，则按 ACL_GROUP_OBJ 条目授予请求的权限，然后这个条目确定进程对文件的访问权限。授予的访问权限需与 ACL_MASK 条目的值（如果有的话）进行掩码。
 - b) 如果进程的某个组 ID 匹配文件某个 ACL_GROUP 条目的 tag qualifier，则按该条目授予请求的权限，然后这个条目决定最终进程获得的访问权限。授予的权限必须和 ACL_MASK 条目的值进行掩码。
 - c) 否则，拒绝访问。
5. 否则，进程获得 ACL_OTHER 条目规定的权限。

我们可以使用一些例子来阐明组 ID 相关的规则。假设我们有一个文件，组 ID 为 100，而且文件由图 17-1 中的 ACL 所保护。如果进程的组 ID 为 100，调用 `access(file, R_OK)`，则该调用会成功返回 0（我们在[15.4.4 节](#)讨论了 `access()`）。反

过来，虽然 `ACL_GROUP_OBJ` 条目授予了所有权限，调用 `access(file, R_OK | W_OK | X_OK)` 却会失败返回-1，`errno` 设置为 `EACCES`。因为 `ACL_GROUP_OBJ` 权限需要与 `ACL_MASK` 条目进行掩码，而 `ACL_MASK` 拒绝了执行权限。

使用图 17-1 再举一个例子，假设我们有一个进程，组 ID 为 102，同时附加组 ID 还包含组 ID 103。这个进程调用 `access(file, R_OK)` 和 `access(file, W_OK)` 都将成功，因为分别匹配了组 ID 为 102 和 103 的 `ACL_GROUP` 条目；反过来，`access(file, R_OK | W_OK)` 则会失败，因为没有哪个 `ACL_GROUP` 条目同时授予了读和写权限。

17.3 ACL 的长文本和短文本格式

使用 `setfacl` 和 `getfacl` 命令（马上讨论）操作 ACL，或使用某个 ACL 库函数时，我们需要指定 ACL 条目的文本表示法。ACL 允许两种文本表示方法：

- 长文本格式 ACL，每一行包含一个 ACL，并且可能包含注释（从#字符开始直到行尾）。`getfacl` 命令以长文本格式显示 ACL。`setfacl -M acl_file` 选项从文件中取得 ACL 描述，也要求该描述是长文本格式。
- 短文本格式 ACL，包含一组 ACL 条目，以逗号（","）分隔。

在这两种格式中，每个 ACL 条目都包含三个组成部分，并以冒号分隔：

`tag-type:[tag-qualifier]: permissions`

`tag-type` 是表 17-1 第一列显示的某个值。`tag-type` 后面可以跟一个可选的 `tag-qualifier`，用于标识某个用户或组，名字或者数值 ID 都可以。`tag-qualifier` 只能用在 `ACL_USER` 和 `ACL_GROUP` 条目中。

下面全部是短文本格式的 ACL，对应于传统的权限掩码 0650：

```
u::rw-,g::r-x,o::---
u::rw,g::rx,o::-
user::rw,group::rx,other::-
```

下面短文本格式的 ACL 则包含两个用户、一个组、以及一个掩码条目：

```
u::rw,u:paulh:rw,u:annabel:rw,g::r,g:teach:rw,m::rwx,o::-
```

表 17-1: ACL 条目文本格式的解释

Tag 的文本格式	是否需要 Tag qualifier?	相应的 tag 类型	条目类型
u, user	否	ACL_USER_OBJ	文件拥有者（用户）
u, user	是	ACL_USER	指定用户
g, group	否	ACL_GROUP_OBJ	文件组
g, group	是	ACL_GROUP	指定组
m, mask	否	ACL_MASK	组类别的掩码
o, other	否	ACL_OTHER	其它用户

17.4 ACL_MASK 条目和 ACL 组类别

如果 ACL 包含 ACL_USER 或 ACL_GROUP 条目，则必须包含一个 ACL_MASK 条目。如果 ACL 不包含任何 ACL_USER 和 ACL_GROUP 条目，则 ACL_MASK 条目是可选的。

ACL_MASK 条目的作用是控制 ACL group class (组类别) 授予的权限上限。组类别是 ACL 中所有 ACL_USER, ACL_GROUP, ACL_GROUP_OBJ 条目的集合。

ACL_MASK 为那些不知道 ACL 为何物的应用，提供一致的行为。我们举个例子说明为什么需要掩码条目，假设某个文件的 ACL 包含以下条目：

```
user::rwx          # ACL_USER_OBJ
user:paulh:r-x    # ACL_USER
group::r-x        # ACL_GROUP_OBJ
group:teach:--x   # ACL_GROUP
other::--x        # ACL_OTHER
```

现在假设某个程序对这个文件执行下面 chmod() 调用：

```
chmod(pathname, 0700); /* 设置权限为 rwx----- */
```

对于那些不知道 ACL 的应用，这意味着“拒绝文件拥有者之外的任何人访问”。在使用 ACL 的时候，我们需要保持这句话仍然正确。如果没有 ACL_MASK 条目，

我们也可以按许多方式来实现这个行为，但是每种方式都存在一定的问题：

- 简单地修改 `ACL_GROUP_OBJ` 和 `ACL_USER_OBJ` 条目为 “---”，但这样是不够的，因为用户 `paulh` 和组 `teach` 仍然拥有文件的部分权限。
- 另外一种方式是把新的组和其它权限设置应用于所有 `ACL_USER`, `ACL_GROUP`, `ACL_GROUP_OBJ`, `ACL_OTHER` 条目上（禁止所有权限）：

```
user::rwx      # ACL_USER_OBJ
user:paulh:--- # ACL_USER
group::---     # ACL_GROUP_OBJ
group:teach:--- # ACL_GROUP
other::---     # ACL_OTHER
```

这个方法的问题在于 `ACL-unaware` 应用会无意地摧毁 `ACL-aware` 应用建立的所有文件权限语义。例如下面调用就没有办法还原 `ACL_USER` 和 `ACL_GROUP` 条目到原来的状态：

```
chmod(pathname, 751);
```

- 为了避免这些问题，我们可能会考虑使用 `ACL_GROUP_OBJ` 条目来限制所有的 `ACL_USER` 和 `ACL_GROUP` 条目。但是这就意味着 `ACL_GROUP_OBJ` 权限总是要设置为所有 `ACL_USER` 和 `ACL_GROUP` 条目权限的总和。而 `ACL_GROUP_OBJ` 条目本身是用于确定文件组权限，这样就产生了冲突。

`ACL_MASK` 条目就是设计来解决这些问题的。它提供一个机制，允许实现传统的 `chmod()` 操作，而又不毁坏 `ACL` 应用建立的文件权限语义。当 `ACL` 拥有 `ACL_MASK` 条目时：

- 通过 `chmod()` 修改传统组权限时，直接修改 `ACL_MASK` 条目（而不是 `ACL_GROUP_OBJ` 条目）；
- 调用 `stat()` 时，在 `st_mode` 域的组权限中返回 `ACL_MASK` 权限（而不是 `ACL_GROUP_OBJ` 权限）。

`ACL_MASK` 提供了一个方法，在面临 `ACL-unaware` 应用时能够保留 `ACL` 信息，但是反过来就无法保证了。`ACL` 的存在会覆盖文件组权限的传统操作。例如我们

对某个文件设置了下面 ACL:

```
user::rw-,group::---,mask::---,other::r--
```

如果我们对文件执行 chmod g+rw 命令，则 ACL 会变成:

```
user::rw-,group::---,mask::rw-,other::r--
```

在这种情况下，组仍然不能访问文件。一种变通的办法是修改 ACL 组条目，授予所有权限。这样，组就能总是获得 ACL_MASK 条目授予的权限。

17.5 getfacl 和 setfacl 命令

在 shell 中，我们可以使用 getfacl 命令来查看文件的 ACL。

```
$ umask 022          设置 shell umask 为已知状态
$ touch tfile        创建一个新文件
$ getfacl tfile
# file: tfile
# owner: mtk
# group: users
user::rwgroup:::
r--
other::r--
```

从 getfacl 命令的输出中，我们可以看到新文件以最小 ACL 创建。当以文本格式显示这个 ACL 时，getfacl 在 ACL 条目前显示了三行信息，分别是文件名和文件所有权。我们可以指定--omit-header 选项来屏蔽这三行输出。

接下来，我们使用传统的 chmod 命令来修改文件权限，并查看 ACL 如何改变：

```
$ chmod u=rwx,g=rx,o=x tfile
$ getfacl --omit-header tfile
user::rwx
group::r-x
other::--x
```

setfacl 命令修改某个文件的 ACL。这里我们使用 setfacl -m 命令来增加一个 ACL_USER 和一个 ACL_GROUP 条目：

```
$ setfacl -m u:paulh:rx,g:teach:x tfile
$ getfacl --omit-header tfile
```

```

user::rwx
user:paulh:r-x          ACL_USER entry
group::r-x
group:teach:--x          ACL_GROUP entry
mask::r-x                ACL_MASK entry
other::--x

```

`setfacl -m` 选项修改现有 ACL 条目，如果指定的 tag type 和 qualifier 不存在，则创建新的 ACL 条目。我们可以额外地使用-R 选项来递归地应用指定 ACL 到目录树下的所有文件。

从 `getfacl` 命令的输出中，我们可以看到 `setfacl` 命令自动为 ACL 创建了 `ACL_MASK` 条目。

增加的 `ACL_USER` 和 `ACL_GROUP` 条目将这个 ACL 转变成了扩展 ACL，`ls -l` 在传统的文件权限掩码后面附加了一个加号(+):

```
$ ls -l tfile
-rwxr-x--x+ 1 mtk users 0 Dec 3 15:42 tfile
```

下面我们使用 `setfacl` 禁用 `ACL_MASK` 条目执行之外的所有权限，然后使用 `getfacl` 再次查看 ACL:

```

$ setfacl -m m:::x tfile
$ getfacl --omit-header tfile
user::rwx
user:paulh:r-x          #effective:--x
group::r-x               #effective:--x
group:teach:--x
mask::--x
other::--x

```

`getfacl` 在用户 `paulh` 和文件组 (`group::`) 条目后面打印了#effective: 注释，告诉我们在与 `ACL_MASK` 进行掩码 (与) 之后，最终授予的权限会少于这些条目指定的权限。

然后我们再次使用 `ls -l` 来查看传统的文件权限位。可以看到显示的组类别权限位反映出了 `ACL_MASK` (--x)，而不是 `ACL_GROUP` 条目 (r-x):

```
$ ls -l tfile
-rwx--x--x+ 1 mtk  users      0 Dec 3 15:42 tfile
```

`setfacl -x` 命令可以从 ACL 中删除条目。下面我们删除用户 `paulh` 和组 `teach` 的 ACL 条目（删除条目时不需要指定权限）：

```
$ setfacl -x u:paulh,g:teach tfile
$ getfacl --omit-header tfile
user::rwx
group::r-x
mask::r-x
other::---
```

注意在上面的操作过程中，`setfacl` 自动调整掩码条目为所有组类别条目的并集（这里只有一个组条目 `ACL_GROUP_OBJ`）。如果我们想阻止这个自动调整，必须为 `setfacl` 命令指定 `-n` 选项。

最后，`setfacl -b` 选项可以从 ACL 中删除所有扩展条目，仅保留最小 ACL 条目（`user, group, other`）。

17.6 默认 ACL 和文件创建

到目前为止，我们已经讨论过了 `access` ACL，正如名字所暗示的，`access` ACL 用来确定进程访问文件的权限。除了 `access` ACL，我们对目录还可以设置第二种类型的 ACL： `default` ACL。

`default` ACL 对于确定目录访问权限没有任何关系，而是用来确定目录中创建的文件或子目录的 ACL 和权限。（`default` ACL 存储为扩展属性 `system.posix_acl_default`）。

要查看和设置目录的默认 ACL，我们使用 `getfacl` 和 `setfacl` 命令的 `-d` 选项：

```
$ mkdir sub
$ setfacl -d -m u::rwx,u:paulh:rwx,g::rwx,g:teach:rwx,o::- sub
$ getfacl -d --omit-header sub
user::rwx
user:paulh:rwx
group::r-x
group:teach:rwx
mask::rwx          setfacl 自动生成 ACL_MASK 条目
other::---
```

我们可以使用 `setfacl -k` 选项来删除一个目录的默认 ACL。

如果一个目录拥有默认 ACL，则：

- 目录中新创建的子目录继承默认 ACL，换句话说，默认 ACL 会自动传播到所有新创建的子目录树。

- 目录中新创建的文件或子目录会将默认 ACL 继承为自己的 access ACL。

创建文件或子目录的系统调用（`open()`, `mkdir()`等）的 `mode` 参数指定的传统权限位，会与 ACL 条目中相应的传统权限位相与。“相应的 ACL 条目”指的是：

- `ACL_USER_OBJ`;
- `ACL_MASK`, 或者 `ACL_MASK` 不存在，则是 `ACL_GROUP_OBJ`;
- `ACL_OTHER`。

当一个目录拥有默认 ACL，在确定新创建文件的 access ACL 条目时，完全不会使用进程 `umask` ([15.4.6 节](#))。

我们用一个例子来说明新文件如何继承父目录的默认 ACL 为自己的 access ACL，假设我们使用下面 `open()` 调用在目录中创建一个新文件：

```
open("sub/tfile", O_RDWR | O_CREAT,
      S_IRWXU | S_IXGRP | S_IXOTH); /* rwx--x--x */
```

新文件会拥有以下 access ACL:

```
$ getfacl --omit-header sub/tfile
user::rwx
user:paulh:r-x          #effective:--x
group::r-x               #effective:--x
group:teach:rwx          #effective:--x
mask::---x
other::---
```

如果一个目录没有默认 ACL，则：

- 目录中新创建的子目录也没有默认 ACL。
- 新文件或子目录的权限遵循传统的规则 ([15.4.6 节](#))：`open()`、`mkdir()` 调用的 `mode` 参数的文件权限与进程 `umask` 相与，结果就是文件或目录的权限。这个结果也就是新文件的最小 ACL。

17.7 ACL 实现限制

各种文件系统的 ACL 实现强加了一定的限制：

- 在 ext2, ext3, ext4 文件系统中，一个文件的所有扩展属性(name 和 value)必须包含在单个逻辑磁盘块中 ([16.2 节](#))，一个文件的 ACL 总数量因此也受到了限制。每个 ACL 条目要求 8 字节，所以一个文件的最大 ACL 条目数量要稍微少于块大小的八分之一（稍微是因为扩展属性名字需要一定的额外开销）。4096 字节块大小允许大约 500 个 ACL 条目。（另外 2.6.11 之前的内核还限制 ext2 和 ext3 的 ACL 最多 32 个条目）。
- 在 XFS 中，ACL 限制为 25 个条目。
- 在 Reiserfs 和 JFS 中，ACL 可以包含 8191 个条目。主要是因为 VFS 对扩展属性强加了 64KB 的大小限制 ([16.2 节](#))。

在本书写作的时候，Btrfs 限制 ACL 数量大约为 500 条目。不过由于 Btrfs 仍然处于快速开发过程中，这个限制可能会改变。

虽然上面某些文件系统允许创建大量的 ACL 条目，我们应该尽量避免这样做，理由如下：

- 维护大量的 ACL 会造成复杂和可能错误的系统管理工作；
- 扫描 ACL 以匹配某个条目（或者匹配 group ID 检查）所需的时间，随着 ACL 条目的增加也会线性增长。

通常，我们可以通过在系统组文件 ([8.3 节](#)) 中定义适当的组，并在 ACL 中使用这些组，来保持 ACL 条目数量为合理的数值。

17.8 ACL API

POSIX.1e 草案标准定义了一组非常多的函数和数据结构，来操作 ACL。由于数量众多，我们不会详细地讨论所有函数。本书只提供这些函数的使用简介，并以一个示例程序结束。

程序需要包含`<sys/acl.h>`来使用 ACL API, 如果程序还使用了 Linux 对 POSIX.1e 草案标准的扩展, 可能还需要包含`<acl/libacl.h>`。(Linux 扩展可以参考 `acl(5)` 手册页)。程序必须以`-lacl` 选项进行编译, 才能链接到 `libacl` 库。

前面已经说过, Linux 使用扩展属性来实现 ACL, ACL API 则实现为一组库函数, 操作用户空间的数据结构, 必要时, 调用 `getxattr()` 和 `setxattr()` 来获取和修改磁盘中 `system` 扩展属性, 这些属性则保存了 ACL 描述信息。因此程序也可以直接使用 `getxattr()` 和 `setxattr()` 来操作 ACL (非常不推荐)。

简介

`acl(5)` 手册页列出了 ACL API 的所有函数。第一眼看上去, 会感觉函数和数据结构实在太多, 甚至有些混乱。图 17-2 描绘了所有函数和数据结构, 相互之间的关联, 以及这些函数的使用。

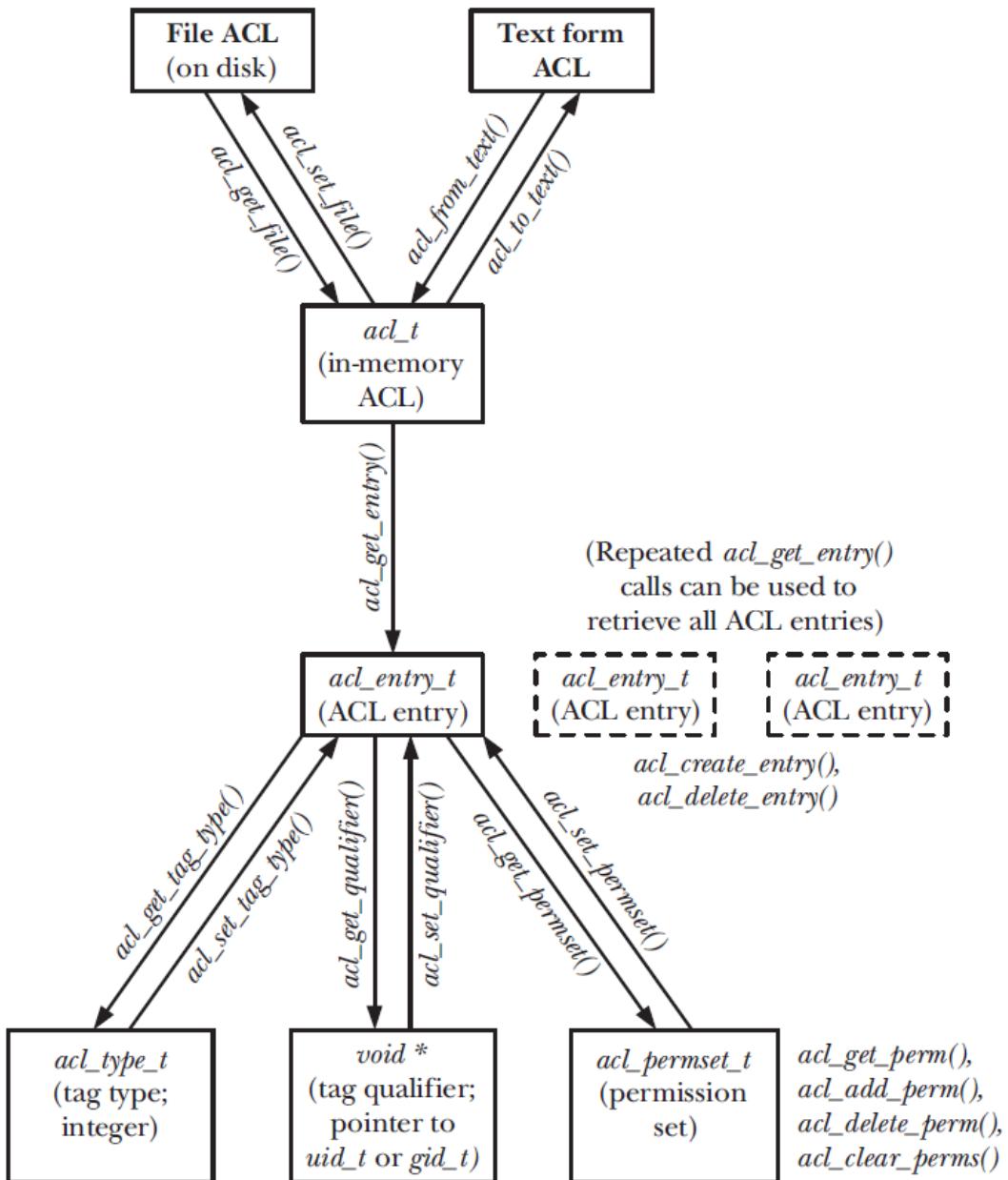


图 17-2: ACL 库函数和数据结构之间的关联

从图 17-2 中，我们可以看到 ACL API 把 ACL 看成分层的对象：

- 一个 ACL 包含一个或多个 ACL 条目
- 每个 ACL 条目包含一个 tag type、一个可选的 tag qualifier、和一个权限集。

现在我们简要地查看各种 ACL 函数，多数情况下，我们不会详细地讨论每个函数的返回错误。返回整数（状态）的函数通常返回 0 表示成功，-1 表示错误；

返回 handle（指针）的函数则返回 NULL 表示错误。我们可以使用通常的 errno 来诊断具体的错误。

handle 是一个抽象术语，很多技术使用 handle 来引用对象或数据结构。handle 的具体表示是 API 实现私有的，可能是指针、数组索引、hash 键等。

获得一个文件的 ACL 到内存

acl_get_file() 函数获取 pathname 参数引用文件的 ACL 拷贝。

```
acl_t acl;
acl = acl_get_file(pathname, type);
```

如果 type 的值为 ACL_TYPE_ACCESS，函数获取文件的 access ACL；如果 type 值为 ACL_TYPE_DEFAULT，则函数获取文件的 default ACL。acl_get_file() 函数返回一个 handle（类型为 acl_t），用于后面的其它 ACL 函数。

从内存中 ACL 获得 entry

acl_get_entry() 函数也返回一个 handle（类型为 acl_entry_t），表示 ACL 的一个 entry（项），其中 acl 参数表示内存中的 ACL。返回的 entry handle 存储在 entry 参数指定的位置。

```
acl_entry_t entry;
status = acl_get_entry(acl, entry_id, &entry);
```

entry_id 参数确定返回哪个 entry 的 handle。如果 entry_id 参数指定为 ACL_FIRST_ENTRY，则返回 ACL 的第一个 entry；如果 entry_id 参数指定为 ACL_NEXT_ENTRY，则返回最后一次获取 entry 的下一个 entry。第一次使用 ACL_FIRST_ENTRY 调用 acl_get_entry()，然后使用 ACL_NEXT_ENTRY 循环调用 acl_get_entry()，就可以遍历整个 ACL 的所有 entry。

acl_get_entry() 函数成功获取 ACL entry 时返回 1；如果没有更多的 entry，或者发生错误，则返回 -1。

获取和修改一个 ACL entry 的属性

acl_get_tag_type() 和 acl_set_tag_type() 函数分别获取和修改 entry 参数引用的

ACL entry 的 tag type。

```
acl_tag_t tag_type;
status = acl_get_tag_type(entry, &tag_type);
status = acl_set_tag_type(entry, tag_type);
```

tag_type 参数的类型为 acl_type_t (整数类型), 值可以是 ACL_USER_OBJ、
ACL_USER、ACL_GROUP_OBJ、ACL_GROUP、ACL_OTHER、ACL_MASK 之一。

acl_get_qualifier()和 acl_set_qualifier()函数分别获取和修改 ACL entry 的 tag
qualifier。下面是一个例子, 这里我们假设通过检查 tag type, 已经确定这是一个
ACL_USER entry:

```
uid_t *qualp; /* Pointer to UID */
qualp = acl_get_qualifier(entry);
status = acl_set_qualifier(entry, qualp);
```

只有 ACL entry 的 tag type 是 ACL_USER 或 ACL_GROUP 时, tag qualifier 才是
合法的。对于 ACL_USER, qualp 是指向用户 ID 的指针(uid_t *); 对于 ACL_GROUP,
qualp 是指向组 ID 的指针 (gid_t *)。

acl_get_permset()和 acl_set_permset()函数获取和设置 ACL entry 的权限集。

```
acl_permset_t permset;
status = acl_get_permset(entry, &permset);
status = acl_set_permset(entry, permset);
```

acl_permset_t 数据类型是一个 handle, 引用 ACL entry 的权限集。

下面代码可用来操作权限集的内容:

```
int is_set;
is_set = acl_get_perm(permset, perm);
status = acl_add_perm(permset, perm);
status = acl_delete_perm(permset, perm);
status = acl_clear_perms(permset);
```

在上面的每次调用中, `perm` 指定为 `ACL_READ`、`ACL_WRITE`、或 `ACL_EXECUTE`, 其含义不言自明。函数的具体使用方法如下:

- 如果 `perm` 参数指定的权限在 `permset` 权限集中已经启用, `acl_get_perm()` 函数返回 1 (true), 否则返回 0。`acl_get_perm()` 是 Linux 对 POSIX.1e 草案标准的扩展函数。
- `acl_add_perm()` 函数添加 `perm` 指定的权限到 `permset` 权限集中。
- `acl_delete_perm()` 函数从 `permset` 权限集中移除 `perm` 指定的权限。(注意移除权限集中不存在的权限并不算是错误)。
- `acl_clear_perms()` 函数从 `permset` 权限集中移除所有权限。

创建和删除 ACL entry

`acl_create_entry()` 函数在现有 ACL 中创建一个新的 entry。函数的结果是一个引用该新 entry 的 handle, 并通过第二个参数返回。

```
acl_entry_t entry;
status = acl_create_entry(&acl, &entry);
```

然后就可以使用前面描述的函数装填这个新 entry。

`acl_delete_entry()` 函数从 ACL 中删除一个 entry。

```
status = acl_delete_entry(acl, entry);
```

更新文件的 ACL

`acl_set_file()` 函数的作用和 `acl_get_file()` 函数相反, 它使用 `acl` 参数引用的内存中的 ACL 来更新磁盘里的 ACL 内容。

```
int status;
status = acl_set_file(pathname, type, acl);
```

`type` 参数可以取值 `ACL_TYPE_ACCESS`, 表示更新 access ACL; 或者取值 `ACL_TYPE_DEFAULT`, 表示更新一个目录的 default ACL。

in-memory 和 text 格式的 ACL 转换

`acl_from_text()`函数将 text 格式的 ACL 字符串转换为内存中的 ACL 格式，长短格式的 text ACL 都可以。`acl_from_text()`函数返回一个 handle，用于后面函数操作该 ACL。

```
acl = acl_from_text(acl_string);
```

`acl_to_text()`函数则执行反向转换，返回一个长文本格式的 ACL 字符串。

```
char *str;
ssize_t len;
str = acl_to_text(acl, &len);
```

如果 `len` 参数不是 `NULL`，则用来存储字符串的长度。

ACL API 的其它函数

下面描述其它几个常用的 ACL 函数，这些函数没有显示在图 17-2 中。

`acl_calc_mask()`函数计算和设置内存中 `ACL_MASK` 项的权限，通常我们在创建或修改一个 ACL 时会使用这个函数。`ACL_MASK` 权限计算为所有 `ACL_USER`、`ACL_GROUP`、`ACL_GROUP_OBJ` 项权限的联合。这个函数在 `ACL_MASK` 项不存在时会自动创建，这是一个非常有用特性。意味着如果我们添加 `ACL_USER` 和 `ACL_GROUP` 项到一个最小 ACL 时，可以使用这个函数来确保创建 `ACL_MASK` 项。

如果 `acl` 有效，`acl_valid(acl)`函数返回 0，否则返回-1。一个 ACL 在以下情况全部满足时才是有效的：

- `ACL_USER_OBJ`、`ACL_GROUP_OBJ`、`ACL_OTHER` 项都有且只有一个；
- 如果存在 `ACL_USER` 或 `ACL_GROUP` 项，则必须存在 `ACL_MASK`；
- 最多只有一个 `ACL_MASK` 项；
- 每个 `ACL_USER` 项都拥有唯一的用户 ID；
- 每个 `ACL_GROUP` 项都拥有唯一的组 ID；

`acl_check()`和 `acl_error()`函数（后者是 Linux 扩展）类似于 `acl_valid()`，可移植性稍差，但对于错误的 ACL 提供更加精确的错误描述。细节请参考手册页。

`acl_delete_def_file(pathname)` 函数删除 `pathname` 目录的默认 ACL。

`acl_init(count)` 函数创建一个新的、空白 ACL 结构体，包含 `count` 个初始 ACL 项的空间。（`count` 参数是对系统的提示，仅表示期望的用量，而不是硬性限制）。函数返回一个新的 ACL handle。

`acl_dup(acl)` 函数复制一个 `acl`，并返回复制 ACL 的 handle。

`acl_free(handle)` 函数释放其它 ACL 函数分配的内存。例如我们在调用 `acl_from_text()`, `acl_to_text()`, `acl_get_file()`, `acl_init()`, `acl_dup()` 之后，必须使用 `acl_free()` 来释放它们分配的内存。

示例程序

清单 17-1 演示了一些 ACL 库函数的使用。这个程序获取并显示一个文件的 ACL (`getfacl` 命令的功能子集)。如果指定了-d 命令行选项，程序显示默认 ACL (目录)，而不是 access ACL。

下面是这个程序的使用示例：

```
$ touch tfile
$ setfacl -m 'u:annie:r,u:paulh:rw,g:teach:r' tfile
$ ./acl_view tfile
user_obj          rw-
user      annie    r--
user      paulh    rw-
group_obj         r--
group     teach    r--
mask                rw-
other                r--
```

本书的源码包还包含一个 `acl/acl_update.c` 程序，用于更新一个文件的 ACL (`setfacl` 命令的功能子集)。

清单 17-1：显示一个文件的 *access* 或 *default ACL*

```
-----acl/acl_view.c
#include <acl/libacl.h>
#include <sys/acl.h>
#include "ugid_functions.h"
#include "tlpi_hdr.h"
```

```
static void
usageError(char *progName)
{
    fprintf(stderr, "Usage: %s [-d] filename\n", progName);
    exit(EXIT_FAILURE);
}

int
main(int argc, char *argv[])
{
    acl_t acl;
    acl_type_t type;
    acl_entry_t entry;
    acl_tag_t tag;
    uid_t *uidp;
    gid_t *gidp;
    acl_permset_t permset;
    char *name;
    int entryId, permVal, opt;

    type = ACL_TYPE_ACCESS;
    while ((opt = getopt(argc, argv, "d")) != -1) {
        switch (opt) {
        case 'd': type = ACL_TYPE_DEFAULT; break;
        case '?': usageError(argv[0]);
        }
    }

    if (optind + 1 != argc)
        usageError(argv[0]);

    acl = acl_get_file(argv[optind], type);
    if (acl == NULL)
        errExit("acl_get_file");

    /* Walk through each entry in this ACL */

    for (entryId = ACL_FIRST_ENTRY; ; entryId = ACL_NEXT_ENTRY) {
        if (acl_get_entry(acl, entryId, &entry) != 1)
            break;           /* Exit on error or no more entries */
    }
}
```

```
/* Retrieve and display tag type */
if (acl_get_tag_type(entry, &tag) == -1)
    errExit("acl_get_tag_type");

printf("%-12s",   (tag == ACL_USER_OBJ) ? "user_obj" :
       (tag == ACL_USER) ? "user" :
       (tag == ACL_GROUP_OBJ) ? "group_obj" :
       (tag == ACL_GROUP) ? "group" :
       (tag == ACL_MASK) ? "mask" :
       (tag == ACL_OTHER) ? "other" : "????");

/* Retrieve and display optional tag qualifier */
if (tag == ACL_USER) {
    uidp = acl_get_qualifier(entry);
    if (uidp == NULL)
        errExit("acl_get_qualifier");

    name = groupNameFromId(*uidp);
    if (name == NULL)
        printf("%-8d ", *uidp);
    else
        printf("%-8s ", name);

    if (acl_free(uidp) == -1)
        errExit("acl_free");
} else if (tag == ACL_GROUP) {
    gidp = acl_get_qualifier(entry);
    if (gidp == NULL)
        errExit("acl_get_qualifier");

    name = groupNameFromId(*gidp);
    if (name == NULL)
        printf("%-8d ", *gidp);
    else
        printf("%-8s ", name);

    if (acl_free(gidp) == -1)
        errExit("acl_free");
} else {
    printf(" ");
}
```

```
/* Retrieve and display permissions */
if (acl_get_permset(entry, &permset) == -1)
    errExit("acl_get_permset");

permVal = acl_get_perm(permset, ACL_READ);
if (permVal == -1)
    errExit("acl_get_perm - ACL_READ");

printf("%c", (permVal == 1) ? 'r' : '-');
permVal = acl_get_perm(permset, ACL_WRITE);
if (permVal == -1)
    errExit("acl_get_perm - ACL_WRITE");
printf("%c", (permVal == 1) ? 'w' : '-');
permVal = acl_get_perm(permset, ACL_EXECUTE);
if (permVal == -1)
    errExit("acl_get_perm - ACL_EXECUTE");
printf("%c", (permVal == 1) ? 'x' : '-');

printf("\n");
}

if (acl_free(acl) == -1)
    errExit("acl_free");

exit(EXIT_SUCCESS);
}
-----acl/acl_view.c
```

17.9 小结

从内核 2.6 版本开始，Linux 支持 ACL。ACL 扩展了传统的 UNIX 文件权限模型，允许文件权限以每个用户和每个组的粒度来进行控制。

更多信息

POSIX.1e 和 POSIX.2c 标准草案的最终版本（Draft 17）可以在 <http://wt.tuxomania.net/publications posix.1e/> 找到。

acl(5) 手册页提供了 ACL 综述，各种 ACL 库函数在 Linux 中的实现，以及各自的可移植性。

Linux 对 ACL 的实现细节和扩展属性，可以在[Grünbacher, 2003]中找到。Andreas Grünbacher 还维护了一个网站：<http://acl.bestbits.at/>，包含许多 ACL 相关的信息。

17.10 习题

- 17-1. 编写一个程序，显示 ACL 项中特定用户或组的权限。程序有两个命令行参数，第一个参数是字母 u 或 g，表示第二个参数标识用户还是组（清单 8-1 的程序(第 216 页)，可以允许第二个参数指定为数值，或者名字)。如果特定用户或组相应的 ACL 项属于组类别，则程序还应该显示 ACL mask 项修改后的 ACL 权限。

第 18 章 目录和链接

在这一章，我们讨论目录与链接，并以此来结束我们对文件相关主题的讨论。首先我们简述目录与链接的实现，讨论创建和删除目录与链接的系统调用。然后学习扫描和遍历目录树内容的库函数。

每个进程都拥有两个目录相关的属性：`root` 目录确定绝对路径解析的入口点；当前工作目录则确定相对路径解析的入口点。我们会查看进程修改这两个属性的系统调用。

本章末尾我们讨论路径解析的相关库函数。

18.1 目录和（硬）链接

目录和普通文件在文件系统中使用类似的存储方式，但以下两点使目录区别于普通文件：

- 目录的 i-node 项 ([14.4 节](#)) 被标记为不同的文件类型
- 目录是拥有特殊组成结构的文件。本质上是一个表，包含一组文件名和 i-node 数值

在多数 Linux 本地文件系统中，文件名最多可以长达 255 个字符。图 18-1 阐明了目录和 i-node 之间的关联，该图以 `/etc/passwd` 文件为例，显示了文件系统 i-node 表和相关目录文件的部分内容。

虽然进程可以打开一个目录，但不能使用 `read()` 来读取目录的内容。要获取一个目录的内容，进程必须使用本章后面讨论的系统调用和库函数。（在某些 UNIX 实现中，可以对目录执行 `read()` 操作，但这是不可移植的）。进程也不能直接使用 `write()` 来修改一个目录的内容；只能间接地使用系统调用（也就是请求内核）来修改目录的内容，如 `open()`（创建一个新文件）、`link()`、`mkdir()`、`symlink()`、`unlink()`、`rmdir()`。（本章马上就会讨论所有这些系统调用，除了 `open()` 已经在 [4.3 节](#) 讨论过了）。

i-node 表从 1 开始编号，而不是 0，因为一个目录项 i-node 域为 0 表示这个项未被使用。i-node 1 用来记录文件系统中的坏块。文件系统的根目录（`/`）总是存储在 i-node 项 2 中（如图 18-1 所示），因此内核知道从哪里开始解析路径名。

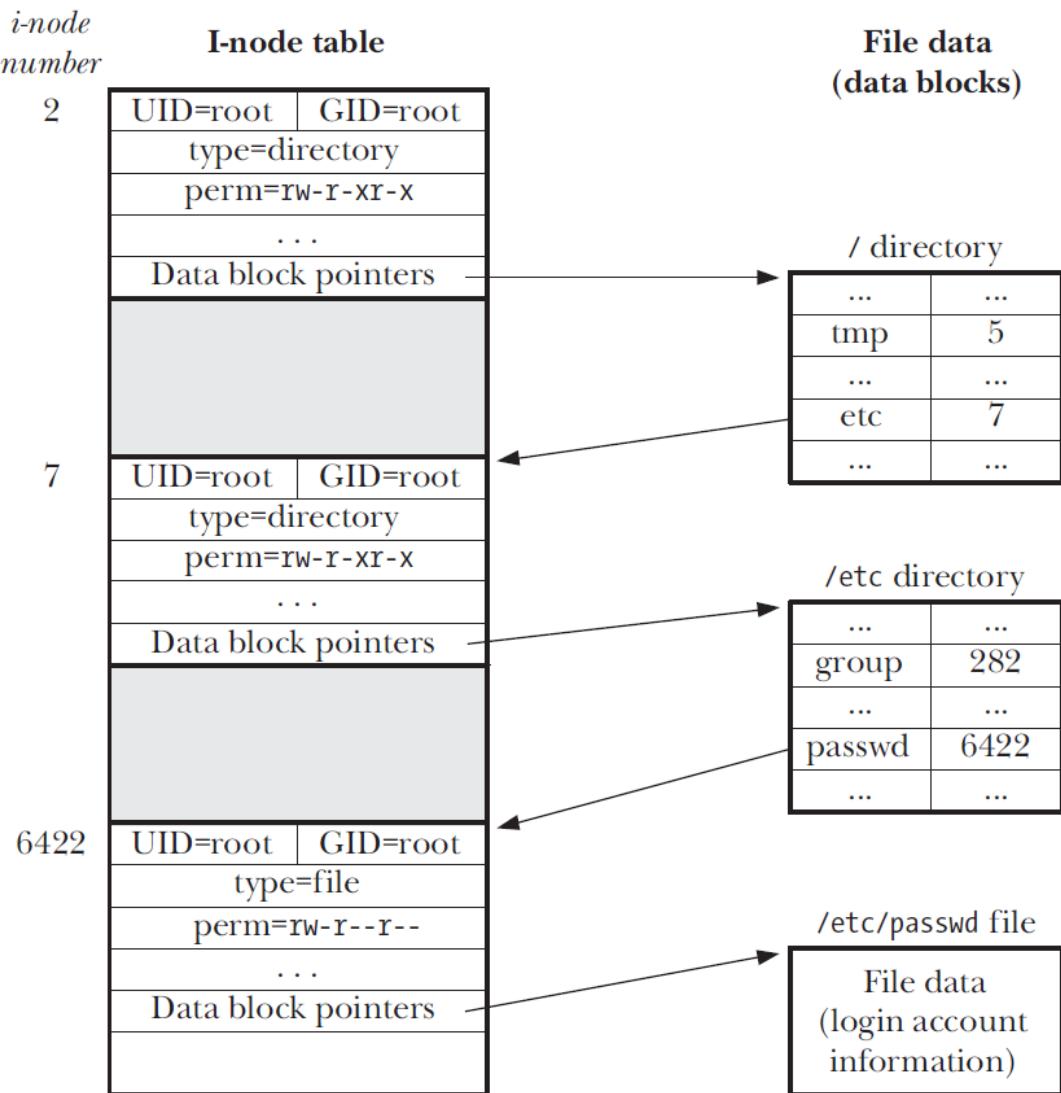


图 18-1: /etc/passwd 文件的 i-node 和目录结构之间的关联

如果我们回顾一下文件 i-node 中存储的信息列表 ([14.4 节](#))，可以看到 i-node 并没有包含文件名；里面只有目录列表的映射，由此来定义一个文件的文件名。这样设计有一个好处：我们可以创建多个名字（在相同或不同的目录中），每个名字都引用相同的 i-node。这些名字就是我们所知的链接（link），有时候也叫做硬链接，以区别于我们马上要讨论的符号链接。

所有 Linux 和 UNIX 本地文件系统都支持硬链接。但是许多非 UNIX 文件系统（如 Microsoft 的 VFAT）却不支持。（Microsoft 的 NTFS 文件系统是支持硬链接的）。

在 shell 中，我们可以使用 `ln` 命令对已经存在的文件创建新的硬链接，如下面 shell 会话日志所示：

```
$ echo -n 'It is good to collect things,' > abc
$ ls -li abc
122232 -rw-r--r-- 1 mtk      users      29 Jun 15 17:07 abc
$ ln abc xyz
$ echo ' but it is better to go on walks.' >> xyz
$ cat abc
It is good to collect things, but it is better to go on walks.
$ ls -li abc xyz
122232 -rw-r--r-- 2 mtk      users      63 Jun 15 17:07 abc
122232 -rw-r--r-- 2 mtk      users      63 Jun 15 17:07 xyz
```

从 `cat` 命令的输出, 以及 `ls -li` 命令显示的 i-node 号(第一列)我们可以看出:
`abc` 和 `xyz` 引用相同的 i-node 项, 因此引用相同文件。`ls -li` 显示的第三个域是当前 i-node 的链接数目。执行 `ln abc xyz` 命令之后, `abc` 文件 i-node 的链接数增加为 2, 因为现在已经有两个名字引用同一个文件了。(文件 `xyz` 的链接数目是一样的, 因为都引用同一个 i-node)。

如果某个文件名被删除, 另一个名字以及文件本身会继续存在:

```
$ rm abc
$ ls -li xyz
122232 -rw-r--r-- 1 mtk      users      63 Jun 15 17:07 xyz
```

只有 i-node 的链接数目降为 0 时, 也就是文件的所有名字都被删除, 文件的 i-node 项和数据块才会被实际地删除(释放)。总结起来说: `rm` 命令从目录列表中删除一个文件名, 相应地 i-node 链接数目减 1, 如果链接数目降为 0, 才会释放 i-node 和数据块。

一个文件的所有名字(链接)都是等价的, 没有哪个名字拥有优先权。正如上面例子所示, 当文件最初的名字被删除时, 物理文件仍然存在, 此时可以使用另一个名字进行访问。

网上论坛里经常提的一个问题是“我怎样才能找出程序中文件描述符 X 相关联的文件名?”。简单地说我们无法做到, 至少不能可移植或明确地实现。因为一个文件描述符引用一个 i-node, 而多个文件名可能同时引用该 i-node(甚至也可能没有任何文件名引用该 i-node, 如 [18.3 节](#)所述)。

在 Linux 中，我们可以使用 `readdir()` ([18.8 节](#)) 来扫描 Linux 特定的 /proc/PID/fd 目录的内容，该目录包含进程已经打开的所有文件描述符的符号链接，从而获得进程当前已经打开的所有文件。`lsof(1)` 和 `fuser(1)` 工具也可以实现类似的目的，这两个工具已经迁移到许多 UNIX 系统。

硬链接有两个局限，而使用符号链接可以避免这两个限制：

- 由于目录项（硬链接）使用 i-node 编号来引用文件，而 i-node 号只在单个文件系统中唯一，因此硬链接必须与引用的文件处于相同的文件系统。
- 硬链接不能引用一个目录。这样可以防止创建环形链接，后者会使许多系统程序产生混乱。

早期 UNIX 实现允许超级用户创建目录的硬链接，这是必要的，因为这些 UNIX 实现没有提供 `mkdir()` 系统调用。在这些系统中，需要使用 `mknod()` 来创建目录，然后再创建“.” 和“..”两个链接 ([Vahalia, 1996])。尽管这个特性已经不再需要，某些现代 UNIX 实现仍然保留了它，以便向后兼容。

使用 `bind` 挂载 ([14.9.4 节](#)) 可以实现类似的目录硬链接。

18.2 符号（软）链接

符号链接有时候也称为软链接，是一种特殊的文件类型，符号链接文件的数据是另一个文件的名字。图 18-2 说明了两个硬链接 /home/erena/this 和 /home/allyn/that 引用相同文件，以及一个符号链接 /home/kiran/other 引用 /home/erena/this。

在 shell 中，使用 `ln -s` 命令来创建符号链接。`ls -F` 命令会在符号链接的末尾显示一个@字符。

一个符号链接引用的路径名可能是绝对路径，也可能是相对路径。相对路径符号链接会相对于链接本身的位置进行解析。

符号链接与硬链接并不相同，最重要的区别是符号链接不包含所引用文件的链接数目（因此图 18-2 中 i-node 61 的链接数目是 2 而不是 3）。因此如果符号链接引用的文件名被删除，符号链接本身会继续存在，当然此时符号链接无法进行解引用（跟随）。我们称这样的符号链接为 `dangling link`。甚至文件名不存在时，也可以创建符号链接。

符号链接由 4.2BSD 引入，尽管 POSIX.1-1990 标准没有包含符号链接，随后却被整合到了 SUSv1 标准，因此 SUSv3 也是包含符号链接的。

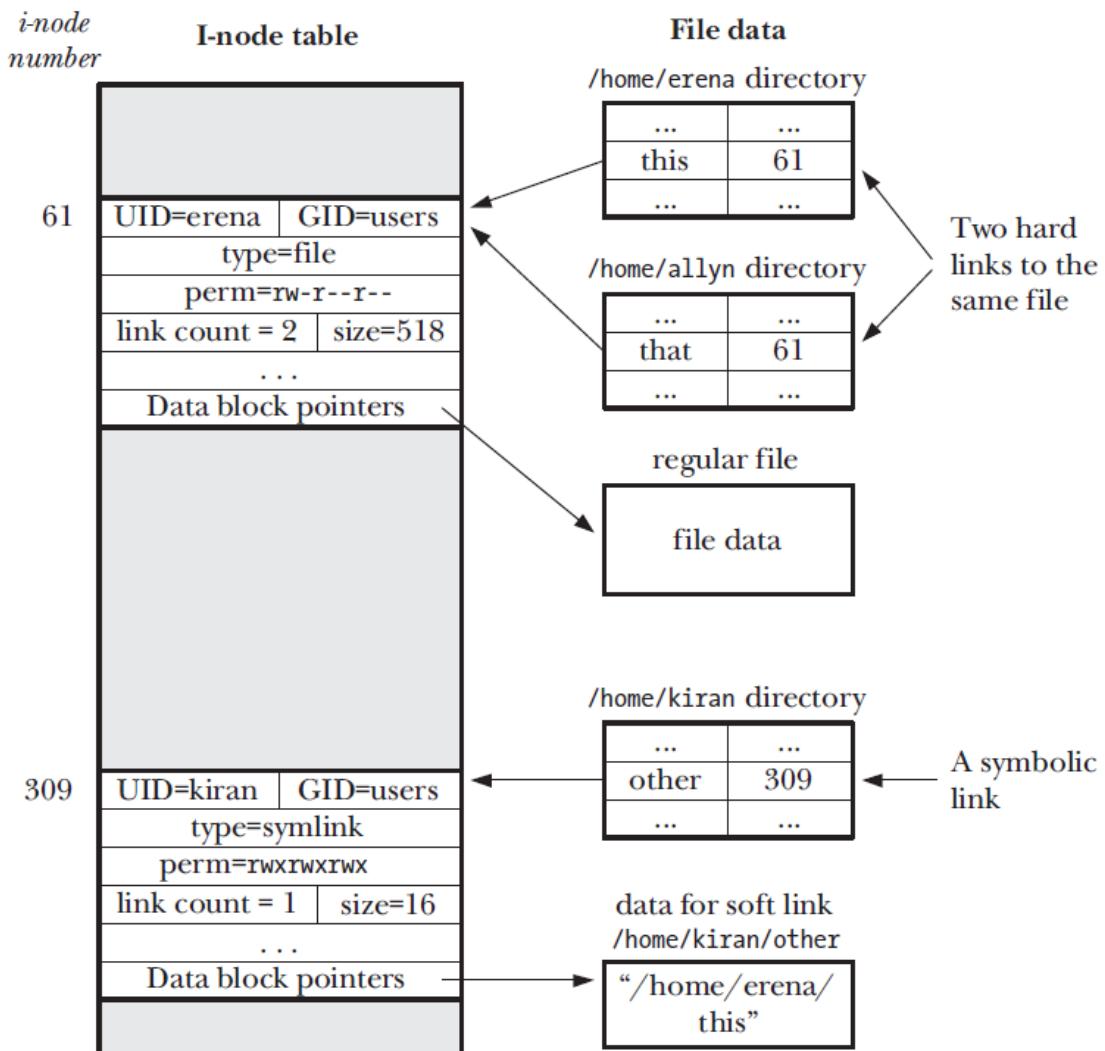


图 18-2：硬链接和符号链接的描述

由于符号链接引用文件名而不是 *i-node* 号，因此可以用来链接不同文件系统的文件。另外符号链接也没有硬链接的另一个局限：我们可以创建目录的符号链接。从 `find` 和 `tar` 等工具可以看出硬链接和符号链接之间的区别，两者默认都不会解引用符号链接，也不能避免符号链接环形引用所造成的陷阱。

我们创建符号链接的符号链接（例如 `a` 是 `b` 的符号链接，而 `b` 又是 `c` 的符号链接）。调用文件系统相关的各种系统调用时，如果指定符号链接，内核会解引用这一系列的符号链接，并到达最终的文件。

SUSv3 要求允许至少 `_POSIX_SYMLOOP_MAX` 解引用每个符号链接的路径名组成，标准为 `_POSIX_SYMLOOP_MAX` 指定的值为 8。不过在内核 2.6.18 之前，Linux 在解引用符号链接（chain）时，限制为 5。从内核 2.6.18 开始，Linux 实现了 SUSv3 规定的最小 8 个解引用。另外 Linux 还限制完整路径名最多 40 个解引用。这些限制用来阻止非常长的符号链接链，以及符号链接回环，从而避免符号链接 resolve 相关内核代码的堆栈溢出。

除了图 18-2 和上面正文所描述，某些 UNIX 文件系统还会执行一个优化。当符号链接内容的字符串总长度足够小，可以存储在 i-node 数据指针的位置时，链接字符串就会存储在该数据指针处。这样可以节省分配一个磁盘块，同时能加速对符号链接信息的访问，因为可以直接在文件 i-node 中一起获取。例如，ext2, ext3, ext4 都采用了这个技术，将短符号链接字符串直接存储在 60 字节的数据块指针处。在实践中，这是一个非常高效的优化。在作者的系统中，经检查 20700 个符号链接，有 97% 都小于 60 个字节。

系统调用对符号链接的解释

多数系统调用会解引用（跟随）符号链接，因此会对链接引用的文件进行操作。某些系统调用则不会解引用符号链接，直接对链接文件本身进行操作。本书在讲解每个系统调用时，都描述了其对符号链接的行为。同时表 18-1 还汇总了系统调用对符号链接的行为。

少数情况下，系统调用需要对符号链接引用文件和链接本身都执行类似的功能，这时候通常会提供一组系统调用：其中一个会解引用链接，另一个则直接对链接进行操作，后者通常会以字母“l”作为前缀，例如 `stat()` 和 `Istat()`。

有一点是通用的：路径名目录部分（最后一个斜杠之前的所有组成）中的符号链接总是会被解引用。因此在路径 /somedir/somesubdir/file 中，`somedir` 和 `somesubdir` 如果是符号链接，总是会解引用；而 `file` 则根据不同的系统调用，可能会进行解引用（也可能不会）。

在 [18.11 节](#)，我们讨论一组 Linux 2.6.16 新增的系统调用，扩展了表 18-1 中某些接口的功能。对于这里边的某些系统调用，符号链接的行为可以由系统调用的 `flags` 参数控制。

符号链接的文件权限和所有权

多数操作会忽略符号链接的所有权和权限（符号链接创建时总是启用了所有权限）。实际上检查的是链接所引用文件的所有权和权限，从而确定操作是否允许。只有在启用了 sticky 权限位 ([15.4.5 节](#)) 的目录中，删除或重命名目录下的符号链接本身时，才会检查符号链接的所有权。

表 18-1：各种函数对符号链接的解释

函数	跟随链接？	备注
access()	●	
acct()	●	
bind()	●	UNIX domain sockets 拥有路径名
chdir()	●	
chmod()	●	
chown()	●	
chroot()	●	
creat()	●	
exec()	●	
getxattr()	●	
lchown()		
lgetxattr()		
link()		参考 18.3 节
listxattr()	●	
llistxattr()		
lremovexattr()		
lsetxattr()		
lstat()		

<code>lutimes()</code>		
<code>open()</code>	●	除非指定了 <code>O_NOFOLLOW</code> 或者 <code>O_EXEC O_CREAT</code> 标志
<code>opendir()</code>	●	
<code>pathconf()</code>	●	
<code>pivot_root()</code>	●	
<code>quotactl()</code>	●	
<code>readlink()</code>		
<code>removexattr()</code>	●	
<code>rename()</code>		所有链接参数都不会跟随
<code>rmdir()</code>		如果参数是符号链接，以 <code>ENOTDIR</code> 失败
<code>setxattr()</code>	●	
<code>stat()</code>	●	
<code>statfs(), statvfs()</code>	●	
<code>swapon(), swapoff()</code>	●	
<code>truncate()</code>	●	
<code>unlink()</code>		
<code>uselib()</code>	●	
<code>utime(), utimes()</code>	●	

18.3 创建和删除（硬）链接：link() 和 unlink()

link() 和 unlink() 系统调用创建和删除一个硬链接

```
#include <unistd.h>

int link(const char *oldpath, const char *newpath);
```

成功时返回 0；出错时返回 -1

`oldpath` 指定一个现有文件的路径, `link()` 系统调用会使用 `newpath` 指定的路径创建一个新链接。如果 `newpath` 已经存在, 则不会覆盖, 返回 `EEXIST` 错误。

在 Linux 中, `link()` 系统调用不会解引用符号链接。如果 `oldpath` 是一个符号链接, 则 `newpath` 将创建为该符号链接文件的硬链接 (换句话说, `newpath` 本身也是一个符号链接, 和 `oldpath` 都引用同一个文件)。这个行为没有依从 SUSv3, 标准规定所有执行路径 `Resolve` 的函数都应该解引用符号链接, 除非标准特别指出不解引用 (但 `link()` 并没有特别指明这一点)。多数其它 UNIX 系统都采用了 SUSv3 规定的行为。不过 Solaris 例外, 它提供的默认行为和 Linux 是一致的, 但如果指定了适当的编译器选项, Solaris 可以提供 SUSv3 依从的行为。不同系统对 `link()` 实现之间的不一致, 结果就是要求可移植应用避免为 `link()` 的 `oldpath` 参数指定一个符号链接。

SUSv4 承认现有实现对 `link()` 的不一致, 并规定 `link()` 是否解引用符号链接是实现定义的。SUSv4 同时还增加了 `linkat()` 规范, 执行 `link()` 相同的任务, 但拥有一个 `flags` 参数, 可以用来控制是否解引用符号链接。更多细节请参考 [18.11 节](#)。

```
#include <unistd.h>

int unlink(const char *pathname);
```

成功时返回 0; 出错时返回 -1

`unlink()` 系统调用删除一个链接 (删除一个文件名), 并且如果这是文件的最后一个链接, 同时还删除文件本身。如果 `pathname` 指定的链接不存在, `unlink()` 以 `ENOENT` 错误失败。

我们不能使用 `unlink()` 来删除一个目录, 需要使用 `rmdir()` 或 `remove()`, [18.6 节](#) 会进行讨论。

SUSv3 规定如果 `pathname` 指定为目录, `unlink()` 应该以错误 `EPERM` 失败。但是在 Linux 中, `unlink()` 在这种情况下会以 `EISDIR` 错误失败。(LSB 显式地允许这个 SUSv3 背离行为)。可移植应用在检查这种情况时, 需要准备好处理这两个错误值。

`unlink()` 系统调用不会解引用符号链接。如果 `pathname` 指定为符号链接, 会删除符号链接本身, 而不是其引用的文件名。

已打开文件只有所有文件描述符关闭时才会删除

除了维护每个 i-node 的链接计数，内核还为文件计算了打开文件描述符数量（参考图 5-2）。一个文件的最后一个链接被移除时，只要有任何进程保持了该文件的打开文件描述符，文件就不会实际地删除，直到所有描述符都关闭。这是一个非常有用的特性，因为它允许我们 `unlink` 一个文件时，不需要担心是否有其它进程打开了该文件。（但是我们不能重新附加一个名字到一个链接计数为 0 的打开文件）。此外我们还可以使用这个特性实现一些小技巧，如创建和打开一个临时文件，立即 `unlink` 它，然后在我们的程序中继续使用这个打开文件，因为文件只有在我们关闭文件描述符时才会被销毁——要么显式关闭，要么程序退出时隐式关闭。（这也是我们在 [5.12 节](#) 讨论的 `tmpfile()` 函数采用的方法）。

清单 18-1 中的程序显示了即使文件的最后一个链接被移除，也只有所有打开文件描述符关闭时，文件才会被删除。

清单 18-1：使用 `unlink()` 移除一个链接

```
-----dirs_links/t_unlink.c
-----
#include <sys/stat.h>
#include <fcntl.h>
#include "tlpi_hdr.h"

#define CMD_SIZE 200
#define BUF_SIZE 1024

int
main(int argc, char *argv[])
{
    int fd, j, numBlocks;
    char shellCmd[CMD_SIZE];           /* Command to be passed to system() */
    char buf[BUF_SIZE];                /* Random bytes to write to file */

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

    numBlocks = (argc > 2) ? getInt(argv[2], GN_GT_0, "num-1kB-blocks")
                           : 100000;
```

```

fd = open(argv[1], O_WRONLY | O_CREAT | O_EXCL, S_IRUSR | S_IWUSR);
if (fd == -1)
    errExit("open");

if (unlink(argv[1]) == -1)          /* Remove filename */
    errExit("unlink");

for (j = 0; j < numBlocks; j++) /* Write lots of junk to file */
    if (write(fd, buf, BUF_SIZE) != BUF_SIZE)
        fatal("partial/failed write");

snprintf(shellCmd, CMD_SIZE, "df -k `dirname %s`", argv[1]);
system(shellCmd);                  /* View space used in file system */

if (close(fd) == -1)              /* File is now destroyed */
    errExit("close");
printf("***** Closed file descriptor\n");

system(shellCmd);                /* Review space used in file system */
exit(EXIT_SUCCESS);
}
-----dirs_links/t_unlink.c

```

上面程序接受两个命令行参数，第一个参数指定程序创建的文件名，程序打开这个文件，然后立即 `unlink` 这个文件名。尽管文件名消失了，文件本身仍然存在。程序然后写入随机数据到这个文件。程序的第二个可选命令行参数指定这些数据块的数量。然后程序使用 `df` 命令来显示文件系统使用的空间数量。接下来程序关闭这个文件描述符，此时文件将被删除，并再次使用 `df` 命令来显示文件使用的磁盘空间已经被释放。下面 shell 会话演示了清单 18-1 程序的使用：

```

$ ./t_unlink /tmp/tfile 1000000
Filesystem      1K-blocks  Used      Available  Use%  Mounted on
/dev/sda10      5245020   3204044   2040976   62%   /
*****
Closed file descriptor
Filesystem      1K-blocks  Used      Available  Use%  Mounted on
/dev/sda10      5245020   2201128   3043892   42%   /

```

在清单 18-1 中，我们使用 `system()` 函数来执行一个 shell 命令。我们会在 [27.6 节](#) 详细讨论 `system()`。

18.4 文件重命名: `rename()`

`rename()`系统调用可以重命名一个文件，也可以移动文件到同一个文件系统的另一个目录。

```
#include <stdio.h>

int rename(const char *oldpath, const char *newpath);
```

成功时返回 0；出错时返回 -1

`oldpath` 参数是一个现有的路径名，将会被重命名到 `newpath` 指定的路径名。

`rename()`系统调用只是操作目录项（entry），它不会移动文件数据。重命名一个文件不会影响文件的其它硬链接，也不会影响任何保持了该文件打开描述符的进程，因为这些描述符引用打开文件描述符，后者（在 `open()`调用之后）与文件名并没有任何关联。

使用 `rename()`时可以应用以下规则：

- 如果 `newpath` 已经存在，会被覆盖
- 如果 `newpath` 和 `oldpath` 引用相同文件，则不会产生任何修改（调用会成功返回）。这是非常违反直觉的，从上一条规则来看，我们一般会期望如果两个文件名 `x` 和 `y` 存在，则调用 `rename("x", "y")`会移除名字 `x`。但是当 `x` 和 `y` 是同一个文件的链接时，是不会移除 `x` 的。

这条规则背后的原理，起源于最初的 BSD 实现，可能是为了简化内核必须执行的检查，为了确保 `rename("x", "x")`、`rename("x", "./x")`、`rename("x", "somedir/..x")` 等调用不会移除文件。

- `rename()`系统调用对两个参数都不会解引用符号链接。如果 `oldpath` 是一个符号链接，则符号链接会被重命名。如果 `newpath` 是一个符号链接，也当作普通路径名处理（已存在的 `newpath` 路径名会被覆盖）。
- 如果 `oldpath` 引用一个文件而不是目录，则 `newpath` 不能指定为一个目录（错误为 `EISDIR`）。要重命名一个文件到某个目录下（如移动文件到另一个目录），`newpath` 必须包含新的文件名。下面调用在移动文件到新目录的同时还进行了重命名：

```
rename("sub1/x", "sub2/y");
```

- `oldpath` 指定为目录名，允许重命名这个目录。在这种情况下，`newpath` 必须不存在，或者必须是一个空目录的名字。如果 `newpath` 是一个已经存在的文件，或者是已经存在的非空目录，则分别返回 `ENOTDIR` 和 `ENOTEMPTY` 错误。
- 如果 `oldpath` 是一个目录，则 `newpath` 不能包含一个和 `oldpath` 相同的目录前缀，否则返回 `EINVAL` 错误。例如，我们不能重命名`/home/mtk` 为 `/home/mtk/bin`。
- `oldpath` 和 `newpath` 引用的文件必须在相同文件系统中。因为一个目录实际上是硬链接的列表，引用相同文件系统中的 i-node，所以这点是必须的。前面已经讲过，`rename()` 只是操作目录列表的内容。试图重命名一个文件到另一个文件系统，会以 `EXDEV` 错误失败。(要达到期望的结果，我们必须从一个文件系统中，复制文件的内容到另一个文件系统，然后删除旧文件。`mv` 命令在这种情况下也正是这样做的)。

18.5 操作符号链接：`symlink()` 和 `readlink()`

现在我们来讨论创建符号链接，以及查看符号链接内容的系统调用。

`symlink()` 系统调用为 `filepath` 指定的路径名创建一个新的符号链接 `linkpath`。
(要删除一个符号链接，我们使用 `unlink()`)。

```
#include <unistd.h>

int symlink(const char *filepath, const char *linkpath);
```

成功时返回 0；出错时返回 -1

如果 `linkpath` 指定的路径名已经存在，则上面调用将失败（`errno` 设置为 `EEXIST`）。`filepath` 指定的路径名可以是绝对或相对路径。

执行 `symlink` 调用时，`filepath` 指定的文件或目录并不需要一定存在。即使创建符号链接时 `filepath` 指定的源文件或目录存在，我们也没有办法阻止文件稍后被删除。当符号链接指向的文件或目录被删除时，`linkpath` 将成为摇摆链接（`dangling link`），此时对符号链接进行解引用将导致错误（通常是 `ENOENT`）。

如果我们 `open()` 时指定一个符号链接，则会打开链接引用的文件。但是有时候我们只是想要获取链接本身的内容，也就是 `pathname` 所指向的链接文件。`readlink()` 系统调用可以完成这个工作，它会将符号链接字符串存储在 `buffer` 所指向的数组。

```
#include <unistd.h>

ssize_t readlink(const char *pathname, char *buffer, size_t bufsiz);
成功时返回 buffer 中存储的字节数；出错时返回 -1
```

`bufsiz` 参数是一个整数，告诉 `readlink()` 系统调用 `buffer` 数组的长度。

如果调用成功，`readlink()` 返回实际存储在 `buffer` 中的字节数。如果链接的长度超出 `bufsiz`，则会将字符串截断再存储在 `buffer` 中，此时 `readlink()` 返回 `bufsiz`。

`readlink()` 不会存储 null 终止字符到 `buffer` 中，因此我们不能区分 `readlink()` 是否返回已经截断的字符串。如果你想要检查是否截断，可以重新分配一个更大的 `buffer`，并再次调用 `readlink()`；或者你也可以使用 `PATH_MAX` 常量（参考 [11.1 节](#)），这个常量定义了路径名的最长长度。

我们在清单 18-4 中演示了 `readlink()` 的使用。

SUSv3 定义了一个新的限制：`SYMLINK_MAX`，用来指示符号链接能够存储的最大字节数，标准要求这个限制至少支持 255 字节。在本书写作的时候，Linux 并没有定义这个限制。因此我们建议使用 `PATH_MAX`，因为这个常量至少与 `SYMLINK_MAX` 一样大（可能更大，能够满足要求）。

在 SUSv2 中，`readlink()` 的返回类型指定为 `int`，而且很多系统（包括 Linux 中老版本的 glibc）仍然遵循这个规定。SUSv3 将返回类型修改为 `ssize_t`。

18.6 创建和删除目录：`mkdir()` 和 `rmdir()`

`mkdir()` 系统调用创建一个新目录。

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

int mkdir(const char *pathname, mode_t mode);
成功时返回 0；出错时返回 -1
```

`pathname` 参数指定新目录的路径，这个路径可以是绝对或相对的。如果

`pathname` 指定的文件已经存在，则 `mkdir()` 将返回 `EEXIST` 错误。

新目录的所有权则根据 [15.3.1 节](#) 中描述的规则进行设置。

`mode` 参数指定新目录的权限（我们已经在 [15.3.1](#)、[15.3.2](#)、[15.4.5](#) 节讨论过目录的权限位）。`mode` 参数可以使用表 15-4 ([371 页](#)) 中的常量进行“或”来指定；但是和 `open()` 一样，我们也可以使用八进制数值来指定。`mode` 参数会与进程 `umask` 相“与”([15.4.6 节](#))。此外，设置用户 ID 位 (`S_ISUID`) 总是被关闭，因为对于目录来说，设置用户 ID 位是没有意义的。

如果 `mode` 中设置了 Sticky 位 (`S_ISVTX`)，则新目录将会启用 Sticky 位。

`mode` 中的设置组 ID 位 (`S_ISGID`) 设置将会被忽略，但是如果父目录的设置组 ID 位已经启用，则新创建的目录也会启用设置组 ID 位。在 [15.3.1 节](#)，我们讨论过目录的设置组 ID 权限位如果启用，则该目录下新创建的文件会从父目录的组 ID 继承自己的组 ID，而不是通过进程的有效组 ID。`mkdir()` 系统调用会按这种方式来传播设置组 ID 权限位，因此所有子目录都共享相同的行为。

[SUSv3](#) 明确规定 `mkdir()` 对设置用户 ID、设置组 ID、Sticky 位的处理是依赖于实现的。在某些 UNIX 实现中，新目录的这三个位总是被关闭。

新创建的目录包含两个项：.（点）是目录本身的链接；..（点点）是父目录的链接。

[SUSv3](#) 并不要求目录包含 . 和 .. 项。标准只要求系统能够正确地解释路径中的 . 和 ..。可移植应用不应该依赖于目录中一定存在这两个项。

`mkdir()` 系统调用只会创建 `pathname` 最后部分指定的目录。换句话说，调用 `mkdir("aaa/bbb/ccc", mode)` 时，只有当目录 aaa 和目录 aaa/bbb 都已经存在时，`mkdir` 调用才会成功。（相应的 `mkdir(1)` 命令的默认行为也是如此，但 `mkdir(1)` 命令提供 `-p` 选项来创建所有不存在的中间目录）。

GNU C 库提供了 `mkdtemp(template)` 函数，是 `mkstemp()` 临时文件函数的目录版。`mkdtemp()` 创建一个唯一名字的目录，只有拥有者具有读、写、执行权限，其它用户没有任何权限。和 `mkstemp()` 函数不一样，`mkdtemp()` 函数并不返回一个文件描述符，而是返回一个字符串指针，包含实际创建的目录名。[SUSv3](#) 没有规定 `mkdtemp()` 函数，而且所有 UNIX 实现都不可用。[SUSv4](#) 最终还是规定了这个函数。

`rmdir()`系统调用删除 `pathname` 指定的目录，`pathname` 可以是绝对或相对路径。

```
#include <unistd.h>

int rmdir(const char *pathname);
```

成功时返回 0；出错时返回 -1

只有目录为空时，`rmdir()`才会成功。如果 `pathname` 的最后部分是一个符号链接，`rmdir()`不会进行解引用，此时返回 `ENOTDIR` 错误。

18.7 删除文件或目录：remove()

`remove()`库函数删除一个文件或一个空目录。

```
#include <stdio.h>

int remove(const char *pathname);
```

成功时返回 0；出错时返回 -1

如果 `pathname` 是一个文件，则 `remove()`会调用 `unlink()`；如果 `pathname` 是一个目录，则 `remove()`会调用 `rmdir()`。

和 `unlink()`与 `rmdir()`一样，`remove()`也不会解引用符号链接。如果 `pathname` 是一个符号链接，`remove()`会删除链接本身，而不是引用的文件或目录。

如果我们想要删除一个文件，然后再创建一个相同名字的文件，则使用 `remove()`会更加简单。因为如果使用 `unlink()`或 `rmdir()`，需要首先检查路径是一个文件还是一个目录。

`remove()`函数是为标准 C 库而创建的，标准 C 库需要在 UNIX 和非 UNIX 系统中实现。但多数非 UNIX 系统并不支持硬链接，因此使用 `unlink()`来删除一个文件是没有任何意义的。

18.8 读取目录：opendir() 和 readdir()

本节讨论的库函数可以用来打开一个目录，并一个一个地获取目录中包含文件的名字。

读取目录的库函数处于 `getdents()` 系统调用 (SUSv3 并没有规定 `getdents`) 的层次之上，但这些库函数提供的接口更容易使用。Linux 还提供了一个 `readdir(2)` 系统调用 (和这里讨论的 `readdir(3)` 库函数不一样)，`readdir(2)` 系统调用执行类似的工作，但是已经被 `getdents()` 系统调用所废弃 (`obsolete`)。

`opendir()` 函数打开一个目录，并返回一个 `handle` 用在后续函数中引用这个目录。

```
#include <dirent.h>

DIR *opendir(const char *dirpath);
```

返回目录的 `stream handle`；出错时返回 `NULL`

`opendir()` 函数打开 `dirpath` 指定的目录，并返回一个指向 `DIR` 结构体的指针。`DIR` 结构体被称为目录流 (`directory stream`)，后面的函数需要我们传递这个 `handle`。当 `opendir()` 返回时，目录流定位在目录列表的第一个项。

`fdopendir()` 函数类似于 `opendir()`，但是使用文件描述符 `fd` 来指定要打开的目录流。

```
#include <dirent.h>

DIR *fdopendir(int fd);
```

返回目录的 `stream handle`；出错时返回 `NULL`

`fdopendir()` 函数可以让应用避免 [18.11 节](#) 讨论的竞争条件。

成功调用 `fdopendir()` 之后，文件描述符将处于系统控制之下，此时程序只能使用本节后面讨论的函数来操作它，不能使用任何其它方式来直接操作 `fd` 文件描述符。

SUSv4 规定了 `fdopendir()` 函数 (但 SUSv3 没有规定)。

`readdir()` 函数依次从目录流中读取后续的项。

```
#include <dirent.h>

struct dirent *readdir(DIR *dirp);
```

成功时返回一个静态分配的结构体指针，指向下一个目录项 出错或者到达目录尾时返回 NULL

每次调用 `readdir()` 会从 `dirp` 目录流中读取下一个目录项，并返回一个静态分配的结构体指针，类型为 `dirent`，包含以下信息：

```
struct dirent {
    ino_t d_ino;      /* 文件 i-node 号 */
    char d_name[];    /* 文件名, null 结尾 */
};
```

每次调用 `readdir()` 都会覆盖这个结构体。

上面的定义忽略了 Linux 系统中 `dirent` 结构体的许多非标准域，因为使用这些非标准的域将使应用不可移植。这些非标准域中最有意思的是 `d_type`，BSD 系列里也定义了这个域，但其它 UNIX 实现则没有提供。`d_type` 域指示 `d_name` 域引用文件的类型，例如 `DT_REG`（普通文件）、`DT_DIR`（目录）、`DT_LNK`（符号链接）、`DT_FIFO`（FIFO）。这些常量类似于第 354 页表 15-1 中列出的宏。使用 `d_type` 域中存储的信息，可以避免调用 `Istat()` 来查看文件的类型。不过注意在本书写作的时候，`d_type` 只被 Btrfs、ext2、ext3、ext4 完全支持。

要获取 `d_name` 域指向文件的更多信息，可以组合 `dirpath` 参数和 `d_name` 来构造文件路径（中间加一个斜杠 “/”），然后再调用 `stat()` 获取文件信息。

`readdir()` 返回的文件名没有经过排序，而是以文件在目录中出现的顺序来读取（依赖于文件系统添加文件到目录的顺序，以及文件系统如何重新填充已经删除的文件项）。`ls -f` 命令会按 `readdir()` 读取文件的未排序顺序来列出所有文件。

我们可以使用 `scandir(3)` 函数来获取已经排序的文件列表，由程序员定义排序规则；更多信息请参考手册页。尽管 SUSv3 没有规定 `scandir()`，多数 UNIX 系统都提供了 `scandir()` 函数。

到达目录末尾和出错时，`readdir()` 都返回 `NULL`，出错时 `readdir()` 会设置 `errno` 来指示错误。要区分这两种情况，我们可以使用下面代码：

```
errno = 0;
direntp = readdir(dirp);
if (direntp == NULL) {
    if (errno != 0) {
        /* Handle error */
    } else {
        /* We reached end-of-directory */
    }
}
```

当程序正在使用 `readdir()` 扫描一个目录时，如果目录内容发生变化，程序将可能无法查看到这些变化。SUSv3 明确规定：从上一次调用 `opendir()` 或 `rewinddir()` 开始到 `readdir()` 扫描目录，期间目录中添加或删除的文件是否能够被 `readdir()` 返回，**是未定义的**。`opendir()` 或 `rewinddir()` 之前添加或删除的文件名，则确保能够被 `readdir()` 返回。

`rewinddir()` 函数将目录流重置到开始位置，下一次调用 `readdir()` 将从目录的第一个文件重新开始。

```
#include <dirent.h>

void rewinddir(DIR *dirp);
```

`closedir()` 函数关闭 `dirp` 指定的目录流，并释放目录流使用的资源。

```
#include <dirent.h>

int closedir(DIR *dirp);
```

成功时返回 0；出错时返回 -1

此外还有两个 SUSv3 规定的函数：`telldir()` 和 `seekdir()`，允许随机地访问目录流。更多信息请参考这两个函数的手册页。

目录流和文件描述符

目录流有一个关联的文件描述符，`dirfd()` 函数可以返回 `dirp` 指定的目录流所关联的文件描述符。

```
#include <dirent.h>

int dirfd(DIR *dirp);
```

成功时返回文件描述符；出错时返回 -1

例如我们可以使用 `dirfd()` 获取目录流的文件描述符，并传递给 `fchdir()`（[参考 18.10 节](#)）来改变进程的当前工作目录。或者我们也可以传递文件描述符给 [18.11 节](#) 描述的几个函数的 `dirfd` 参数。

`dirfd()` 函数也可以在 BSD 系统中使用，但其它 UNIX 系统则可能没有提供该函

数。SUSv3 没有规定 `dirfd()`，但 SUSv4 明确规定了 `dirfd()`。

现在有必要提一点，`opendir()`自动为目录流所关联的文件描述符设置了 `close-on-exec` 标志 (`FD_CLOEXEC`)。这样就确保执行 `exec()` 时，能够自动地关闭该文件描述符 (SUSv3 强制要求这个行为)。我们在 [27.4 节](#) 详细讨论 `close-on-exec` 标志。

示例程序

清单 18-2 使用 `opendir()`、`readdir()`、`closedir()`，来列出命令行参数指定目录的内容（如果没有提供命令行参数，则列出当前工作目录的内容）。下面是这个程序的使用示例：

```
$ mkdir sub          创建一个测试目录
$ touch sub/a sub/b    在测试目录中创建几个文件
$ ./list_files sub      列出目录的内容
sub/a
sub/b
```

清单 18-2：扫描一个目录

```
-----dirs_links/list_files.c
#include <dirent.h>
#include "tlpi_hdr.h"

static void           /* 列出目录'dirPath'的所有文件 */
listFiles(const char *dirpath)
{
    DIR *dirp;
    struct dirent *dp;
    Boolean isCurrent;      /* 如果'dirpath'是"."时为 True */

    isCurrent = strcmp(dirpath, ".") == 0;

    dirp = opendir(dirpath);
    if (dirp == NULL) {
        errMsg("opendir failed on '%s'", dirpath);
        return;
    }
```

```
/* For each entry in this directory, print directory + filename */

for (;;) {
    errno = 0;           /* 为了区分错误和目录末尾 */
    dp = readdir(dirp);
    if (dp == NULL)
        break;

    if (strcmp(dp->d_name, ".") == 0 || strcmp(dp->d_name, "..") == 0)
        continue;          /* 跳过 . 和 .. */

    if (!isCurrent)
        printf("%s/", dirpath);

    printf("%s\n", dp->d_name);
}

if (errno != 0)
    errExit("readdir");

if (closedir(dirp) == -1)
    errMsg("closedir");
}

int
main(int argc, char *argv[])
{
    if (argc > 1 && strcmp(argv[1], "--help") == 0)
        usageErr("%s [dir...]\\n", argv[0]);

    if (argc == 1)           /* 没有参数，使用当前目录 */
        listFiles(".");
    else
        for (argv++; *argv; argv++)
            listFiles(*argv);

    exit(EXIT_SUCCESS);
}
-----dirs_links/list_files.c
```

readdir_r()函数

`readdir_r()`函数是 `readdir()`的一个变种，关键的区别在于 `readdir_r()`是可重入的，而 `readdir()`则不可重入。因为 `readdir_r()`通过调用方分配的 `entry` 参数来返回文件项，而 `readdir()`则是通过静态分配结构体的指针来返回文件项。我们在 [21.1.2 节](#) 和 [31.1 节](#) 讨论可重入（`reentrancy`）。

```
#include <dirent.h>

int readdir_r(DIR *dirp, struct dirent *entry, struct dirent **result);
成功时返回 0; 出错时返回一个正数错误数值
```

`dirp` 是之前使用 `opendir()` 打开的目录流，`readdir_r()`会把下一个目录项的信息存储在 `entry` 参数引用的 `dirent` 结构体中。此外 `result` 还存储了指向这个结构体的指针，然后 `readdir_r()` 返回 0。出错时 `readdir_r()` 不会返回 -1，而是返回某个正数的 `errno` 错误值。

在 Linux 中，`dirent` 结构体的 `d_name` 域是 256 字节的数组，能够保存最大可能长度的文件名。尽管某些其它 UNIX 实现也定义 `d_name` 为 256，SUSv3 标准并未对 `d_name` 的大小进行规定，而某些 UNIX 实现则定义这个域为 1 字节数组，要求调用程序分配正确大小的结构体。当我们负责分配这个结构体时，应该为 `d_name` 域分配 `NAME_MAX` 再加一个额外的字节（`null` 终止字节）。可移植应用应该总是按如下方式分配 `dirent` 结构体：

```
struct dirent *entryp;
size_t len;

len = offsetof(struct dirent, d_name) + NAME_MAX + 1;
entryp = malloc(len);
if (entryp == NULL)
    errExit("malloc");
```

`offsetof()` 宏定义在 `<stddef.h>` 头文件中，使用它可以避免 `dirent` 结构体 `d_name` 域之前的部分在不同系统中存在的差异。注意 `d_name` 域总是 `dirent` 结构体的最后一个域。

`offsetof()`宏需要两个参数：一个结构体类型，和结构体中某个域的名字，`offsetof()`返回这个域在结构体中的偏移位置。使用这个宏是非常有必要的，因为不同编译器可能会在结构体中插入填充字节来满足对齐的要求，这种情况下一个域的偏移可能会比它之前所有域大小的总和还要大一些。

18.9 遍历文件树：`nftw()`

`nftw()`函数允许程序递归地遍历一个完整的目录子树，并对子树的每个文件执行某些操作（也就是调用某些程序员定义的函数）。

`nftw()`函数是对老的 `ftw()` 函数的一个增强版本，两者完成类似的任务。新应用应该使用 `nftw()`（也就是 `new ftw`），因为 `nftw()` 提供更多功能，并且可以按可预测的方式处理符号链接（SUSv3 允许 `ftw()` 跟随或不跟随符号链接）。SUSv3 规定了 `nftw()` 和 `ftw()`，但 SUSv4 已经标注 `ftw()` 为废弃函数（obsolete）。

GNU C 库还提供了起源于 BSD 的 `fts` API (`fts_open()`, `fts_read()`, `fts_children()`, `fts_set()`, `fts_close()`)。这些函数允许应用完成类似于 `ftw()` 和 `nftw()` 的操作，为应用遍历目录树提供了强大得多的灵活性。但是这些 API 未被标准化，除了 BSD 系列之外，只有少数 UNIX 系统提供了实现，因此这里我们不对 `fts` API 进行讨论。

`nftw()` 函数遍历 `dirpath` 指定的目录树，并对目录树中的每一个文件调用程序员定义的 `func` 函数。

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

int nftw(const char *dirpath,
         int (*func) (const char * pathname, const struct stat *statbuf,
                     int typeflag, struct FTW *ftwbuf),
         int nopenfd, int flags);

```

成功遍历整棵树之后返回 0
出错时返回 -1
或者调用 `func` 时第一次返回的非 0 值

默认情况下，`nftw()` 执行未排序，前序遍历指定的目录树。`nftw()` 会首先处理每个目录，然后再处理目录下的文件和子目录。

当 `nftw()` 正在遍历目录树时，函数对目录树的每个级别最多只会打开一个文件描述符。`nopenfd` 参数规定 `nftw()` 可以使用的文件描述符最大数量。如果目录树的深度超出了这个最大值，`nftw()` 会执行一些记录工作，然后关闭和重新打开

描述符，以避免同时保持打开的文件描述符数量超出 `nopenfd` 的限制（后果就是执行效率变慢）。`nopenfd` 参数的出现主要是由于老的 UNIX 实现，每个进程打开的文件描述符数量被系统限制为 20。现代 UNIX 系统允许一个进程打开非常多的文件描述符，因此我们可以为 `nopenfd` 指定一个慷慨的数值（如 10 或更多）。

`nftw()` 函数的 `flags` 参数是以下常量“或”组合而成，可以修改函数的操作：

FTW_CHDIR

在处理每个目录的内容之前，先执行一次 `chdir()` 进入该目录。当 `func` 函数需要在 `pathname` 文件所在的目录中执行某些操作时，这个常量非常有用。

FTW_DEPTH

对目录树执行后序遍历，这意味着 `nftw()` 会先对目录中的所有文件和子目录调用 `func`，然后再对目录本身执行 `func`。（这个标志的名字有些误导，`nftw()` 总是执行深度优先，而不是广度优先遍历。这个标志的作用只是将默认的前序遍历改为后序遍历）。

FTW_MOUNT

不要跨越到另一个文件系统，因此如果某个子目录是一个挂载点，就不对它进行遍历。

FTW_PHYS

默认情况下 `nftw()` 会解引用符号链接。这个标志告诉 `nftw()` 不要解引用符号链接，相反该符号链接会传递给 `func`，并且 `typeflag` 的值将设为 `FTW_SL`。下面会详细讨论。

对于每个文件，`nftw()` 会传递四个参数给 `func` 函数。第一个参数是 `pathname`，也就是文件的路径名。如果 `dirpath` 指定为绝对路径，则这个路径名也是绝对的；如果 `dirpath` 指定为相对路径，则这个路径就是相对于调用进程当前工作目录的相对路径。第二个参数是 `statbuf`，这是一个指向 `stat` 结构体（[第 15.1 节](#)）的指针，包含当前文件的相关信息。第三个参数是 `typeflag`，提供了文件的更多信息，它的值可能是以下之一：

FTW_D

这是一个目录。

FTW_DNR

这是一个目录，但无法读取（因此 `nftw()` 不能遍历它的子目录）。

FTW_DP

`nftw()` 正在对一个目录执行后序遍历（`FTW_DEPTH`），而当前目录的文件和子目录已经被处理过了。

FTW_F

这可能是一个任何类型的文件，但不是目录也不是符号链接。

FTW_NS

对当前文件调用 `stat()` 失败，可能是由于权限问题导致。`statbuf` 中的值是未定义的。

FTW_SL

这是一个符号链接，只有以 `FTW_PHYS` 标志调用 `nftw()` 时才会返回这个值。

FTW_SLN

这是一个摇摆符号链接，只有调用 `nftw()` 时没有指定 `FTW_PHYS` 标志时，才会返回这个值。

第四个参数是 `ftwbuf`，这是一个指向下面结构体的指针：

```
struct FTW {  
    int base;      /* Offset to basename part of pathname */  
    int level;     /* Depth of file within tree traversal */  
};
```

`base` 域是传递给 `func` 函数的文件路径中“文件名部分”的偏移量（也就是最后一个“/”之后的部分）。`level` 域是当前项相对于遍历起点的深度（起点的 `level` 值为 0）。

每次 `func()` 函数被调用时，必须返回一个整数值，而且 `nftw()` 会检查这个返回值。`func()` 返回 0 告诉 `nftw()` 继续文件树遍历，如果所有 `func()` 都返回 0，则 `nftw()` 也返回 0；`func()` 返回非 0 值则告诉 `nftw()` 立即中止文件树遍历，在这种情况下

nftw()会返回相同的非 0 值给调用方。

由于 nftw()使用动态分配的数据结构，程序只能通过 func()返回非 0 值来中止 nftw()函数的遍历操作。使用 longjmp()（[第 6.8 节](#)）从 func()函数中退出可能会导致非预料的结果，而且程序肯定会内存泄漏。

示例程序

清单 18-3 演示了 nftw()函数的使用

清单 18-3：使用 nftw() 来遍历一个目录树

```
-----dirs_links/nftw_dir_tree.c
#define _XOPEN_SOURCE 600      /* Get nftw() and S_IFSOCK declarations */
#include <ftw.h>
#include "tlpi_hdr.h"
static void
usageError(const char *progName, const char *msg)
{
    if (msg != NULL)
        fprintf(stderr, "%s\n", msg);
    fprintf(stderr, "Usage: %s [-d] [-m] [-p] [directory-path]\n",
            progName);
    fprintf(stderr, "\t-d Use FTW_DEPTH flag\n");
    fprintf(stderr, "\t-m Use FTW_MOUNT flag\n");
    fprintf(stderr, "\t-p Use FTW_PHYS flag\n");
    exit(EXIT_FAILURE);
}

static int      /* Function called by nftw() */
dirTree(const char *pathname, const struct stat *sbuf, int type,
        struct FTW *ftwb)
{
    switch (sbuf->st_mode & S_IFMT) { /* Print file type */
    case S_IFREG: printf("-"); break;
    case S_IFDIR: printf("d"); break;
    case S_IFCHR: printf("c"); break;
    case S_IFBLK: printf("b"); break;
    case S_IFLNK: printf("l"); break;
    case S_IFIFO: printf("p"); break;
    case S_IFSOCK: printf("s"); break;
    default: printf(?"); break; /* Should never happen (on Linux) */
    }
}
```

```
}

printf(" %s ",
       (type == FTW_D) ? "D" : (type == FTW_DNR) ? "DNR" :
       (type == FTW_DP) ? "DP" : (type == FTW_F) ? "F" :
       (type == FTW_SL) ? "SL" : (type == FTW_SLN) ? "SLN" :
       (type == FTW_NS) ? "NS" : " ");
if (type != FTW_NS)
    printf("%7ld ", (long) sbuf->st_ino);
else
    printf(" ");

printf(" %*s", 4 * ftwb->level, ""); /* Indent suitably */
printf("%s\n", &pathname[ftwb->base]); /* Print basename */
return 0; /* Tell nftw() to continue */
}

int
main(int argc, char *argv[])
{
    int flags, opt;
    flags = 0;
    while ((opt = getopt(argc, argv, "dmp")) != -1) {
        switch (opt) {
        case 'd': flags |= FTW_DEPTH; break;
        case 'm': flags |= FTW_MOUNT; break;
        case 'p': flags |= FTW_PHYS; break;
        default: usageError(argv[0], NULL);
        }
    }

    if (argc > optind + 1)
        usageError(argv[0], NULL);

    if (nftw((argc > optind) ? argv[optind] : ".", dirTree, 10, flags)
        == -1) {
        perror("nftw");
        exit(EXIT_FAILURE);
    }
    exit(EXIT_SUCCESS);
}
-----dirs_links/nftw_dir_tree.c
```

上面程序会缩排地打印目录树的层次结构，每个文件一行输出，同时还打印了文件类型和 i-node 数值。我们可以使用命令行参数来指定调用 `nftw()` 时使用的 `flags` 参数。下面 shell 会话显示了该程序的运行示例。首先我们创建一个空目录，然后在里面创建各种类型的文件：

```
$ mkdir dir
$ touch dir/a dir/b          创建一些普通文件
$ ln -s a dir/sl             以及一个符号链接
$ ln -s x dir/dsl            以及一个摇摆符号链接
$ mkdir dir/sub               以及一个子目录
$ touch dir/sub/x             并且在里面创建一个文件
$ mkdir dir/sub2              然后创建另一个目录
$ chmod 0 dir/sub2            并使其不可读
```

然后我们使用上面程序调用 `nftw()`，`flags` 参数设为 0：

```
$ ./nftw_dir_tree dir
d D      2327983   dir
- F      2327984   a
- F      2327985   b
- F      2327984   sl      符号链接 sl 成功找到了文件 a
l SLN   2327987   ds1
d D      2327988   sub
- F      2327989   x
d DNR   2327994   sub2
```

在上面输出中，我们可以看到符号 `sl` 成功找到了文件。

接下来我们设置 `nftw()` 函数的 `flags` 参数包含 `FTW_PHYS` 和 `FTW_DEPTH`，并再次运行该命令：

```
$ ./nftw_dir_tree -p -d dir
- F      2327984   a
- F      2327985   b
l SL    2327986   sl      符号链接本身
l SL    2327987   ds1
- F      2327989   x
d DP    2327988   sub
d DNR   2327994   sub2
d DP    2327983   dir
```

我们可以看到符号链接 `sl` 并没有解决到真正的文件。

nftw()的 FTW_ACTIONRETVAL 标志

从 glibc 2.3.3 开始, nftw()函数允许对 flags 参数指定一个额外的非标准标志: FTW_ACTIONRETVAL, 可以改变 nftw()解释 func()函数返回值的方式。当指定了 FTW_ACTIONRETVAL 标志时, func()需要返回以下值:

FTW_CONTINUE

继续处理当前目录下的其它项, 等同于传统的 func()返回 0;

FTW_SKIP_SIBLINGS

不再处理当前目录下的任何其它项, 返回父目录继续处理;

FTW_SKIP_SUBTREE

如果 pathname 是一个目录 (也就是 typeflag 为 FTW_D), 则不调用 func()处理这个目录下的项。继续处理同级的下一个目录。

FTW_STOP

不再处理目录树下的任何项, 等同于传统的 func()返回非 0 值。nftw()会返回 FTW_STOP 值给调用方。

我们必须定义 `_GNU_SOURCE` 特性测试宏, 才能从`<ftw.h>`头文件中获得 FTW_ACTIONRETVAL 的定义。

18.10 进程的当前工作目录

进程的当前工作目录定义了进程解释所有相对路径的起点。新进程会继承父进程的当前工作目录。

获取当前工作目录

进程可以使用 `getcwd()` 来获取自己的当前工作目录。

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

```
char *getcwd(char *cwdbuf, size_t size);
```

成功时返回 `cwdbuf`; 出错时返回 `NULL`

`getcwd()` 函数会把进程的当前工作目录的绝对路径存放在 `cwdbuf` 指向的缓冲区中，最后以 `null` 字符结束。调用方必须为 `cwdbuf` 缓冲区分配至少 `size` 个字节（通常我们使用 `PATH_MAX` 常量来定义 `cwdbuf` 缓冲区）。

成功时 `getcwd()` 返回指向 `cwdbuf` 的指针，如果当前工作目录的长度超过了 `size` 字节，则 `getcwd()` 返回 `NULL`，同时设置 `errno` 为 `ERANGE`。

在 Linux/x86-32 中，`getcwd()` 缓冲区的长度最大为 4096 (`PATH_MAX`) 字节。如果当前工作目录 (`cwdbuf` 和 `size`) 超过了这个限制，则会截断路径名，从字符串的开头移除一些目录前缀（返回的字符串仍然是 `null` 终止的）。换句话说，如果当前工作目录的绝对路径的长度超过了这个限制，则使用 `getcwd()` 是不可靠的。

实际上，Linux 的 `getcwd()` 系统调用会为返回的路径名，在内部分配一个虚拟内存页。在 x86-32 体系架构中，页大小为 4096 字节，但在拥有更大内存页的体系架构中（如 Alpha 的页大小为 8192 字节），`getcwd()` 可以返回更大的路径名。

如果 `cwdbuf` 参数指定为 `NULL`，而且 `size` 指定为 0，则 glibc 对 `getcwd()` 系统调用的封装函数会在内部分配一个足够大的缓冲区，并返回这个指针作为 `getcwd()` 函数的返回值。为了避免内存泄漏，调用方必须在稍后使用 `free()` 释放这个缓冲区。可移植程序应该避免依赖于 glibc 的这个特性。多数其它实现只对 SUSv3 规范提供一个简单的扩展：如果 `cwdbuf` 为 `NULL`，则 `getcwd()` 会分配 `size` 字节的缓冲区，并使用这个缓冲区返回结果给调用方。glibc 的 `getcwd()` 实现也提供了这个特性。

GNU C 库还提供了另外两个获取当前工作目录的函数：其中之一是起源于 BSD 的 `getwd(path)` 函数，但由于它没有提供指定返回路径最大长度的方法，因此可能导致缓冲区溢出 (buffer overruns) 而易受攻击；另一个 `get_current_dir_name()` 函数返回当前工作目录名的字符串，这个函数使用非常简单，但是却不可移植。从安全和可移植的角度来看，`getcwd()` 函数要优于这两个函数（使用 `getcwd()` 时也尽量不要使用 GNU 对该函数的扩展）。

只要拥有适当的权限（基本上只要我们拥有进程，或者拥有 `CAP_SYS_PTRACE` 能力），我们就能通过读取 (`readlink()`) Linux 特定的 `/proc/PID/cwd` 符号链接，来确定任意进程的当前工作目录。

改变当前工作目录

`chdir()`系统调用可以改变调用进程的当前工作目录，`pathname` 指定新的当前工作目录的相对或绝对路径，如果 `pathname` 是符号链接，则会先进行解引用。

```
#include <unistd.h>

int chdir(const char *pathname);
```

成功时返回 0；出错时返回 -1

`fchdir()`系统调用的功能和 `chdir()`完全一样，只不过参数使用文件描述符而不是路径名。

```
#define _XOPEN_SOURCE 500      /* Or: #define _BSD_SOURCE */
#include <unistd.h>

int fchdir(int fd);
```

成功时返回 0；出错时返回 -1

我们可以使用 `fchdir()`改变进程的当前工作目录到另一个位置，随后再返回到原始的位置，如下面代码所示：

```
int fd;

fd = open(".", O_RDONLY);      /* 记住我们的位置 */
chdir(somepath);              /* 进到另外一个位置 */
fchdir(fd);                   /* 返回到原先的位置 */
close(fd);
```

使用 `chdir()`的版本如下：

```
char buf[PATH_MAX];

getcwd(buf, PATH_MAX);        /* 记住我们的位置 */
chdir(somepath);              /* 进到另外一个位置 */
chdir(buf);                  /* 返回到原先的位置 */
```

18.11 目录文件描述符相关的操作

Linux 内核 2.6.16 开始提供一系列新的系统调用，他们执行的任务类似于传统的系统调用，但新系统调用提供了额外的功能，对于某些应用来说非常有用。

这些系统调用汇总在表 18-2，我们之所以在本章讨论这些系统调用，是因为它们对进程的当前工作目录的传统语义提供了变化。

表 18-2：使用目录文件描述符来解析相对路径的系统调用

新接口	传统调用	备注
<code>faccessat()</code>	<code>access()</code>	支持 <code>AT_EACCESS</code> 和 <code>AT_SYMLINK_NOFOLLOW</code> 标志
<code>fchmodat()</code>	<code>chmod()</code>	
<code>fchownat()</code>	<code>chown()</code>	支持 <code>AT_SYMLINK_NOFOLLOW</code> 标志
<code>fstatat()</code>	<code>stat()</code>	支持 <code>AT_SYMLINK_NOFOLLOW</code> 标志
<code>linkat()</code>	<code>link()</code>	支持 <code>AT_SYMLINK_NOFOLLOW</code> 标志 (Linux 2.6.18 开始)
<code>mkdirat()</code>	<code>mkdir()</code>	
<code>mkfifoat()</code>	<code>mkfifo()</code>	<code>mknodat()</code> 之上的库函数
<code>mknodat()</code>	<code>mknod()</code>	
<code>openat()</code>	<code>open()</code>	
<code>readlinkat()</code>	<code>readlink()</code>	
<code>renameat()</code>	<code>rename()</code>	
<code>symlinkat()</code>	<code>symlink()</code>	
<code>unlinkat()</code>	<code>unlink()</code>	支持 <code>AT_REMOVEDIR</code> 标志
<code>utimensat()</code>	<code>utimes()</code>	支持 <code>AT_SYMLINK_NOFOLLOW</code> 标志

我们使用 `openat()` 来解释这些系统调用的使用方法：

```
#define _XOPEN_SOURCE 700 /* Or define _POSIX_C_SOURCE >= 200809 */
#include <fcntl.h>

int openat(int dirfd, const char *pathname,
           int flags, ... /* mode_t mode */);

```

成功时返回文件描述符；出错时返回 -1

`openat()`系统调用类似于传统的 `open()`系统调用，但增加了一个 `dirfd` 参数，使用方法如下：

- 如果 `pathname` 指定为相对路径，则相对于 `dirfd` 引用的目录来解释路径，而不是根据进程的当前工作目录来解释相对路径。
- 如果 `pathname` 指定为相对路径，并且 `dirfd` 指定为特殊值 `AT_FDCWD`，则 `pathname` 仍然根据进程的当前工作目录来解释路径（和 `open(2)` 的行为完全一致）。
- 如果 `pathname` 指定为绝对路径，则忽略 `dirfd` 参数。

`openat()` 的 `flags` 参数的作用和 `open()` 相同。但是表 18-2 所列的某些系统调用也支持一个 `flags` 参数，相应的传统系统调用却没有相应的 `flags` 参数，这时 `flags` 参数用于修改该系统调用的语义和行为。我们最常使用的一个标志是 `AT_SYMLINK_NOFOLLOW`，规定如果 `pathname` 是一个符号链接，则系统调用直接操作链接文件本身，而不是链接所引用的实际文件。`(linkat())` 系统调用还提供一个 `AT_SYMLINK_FOLLOW` 标志，执行相反的动作，它会修改 `linkat()` 的默认行为，如果 `oldpath` 是符号链接，则先进行解引用）。其它标志的细节信息，请参考相应手册页。

Linux 支持表 18-2 中的系统调用，主要有两个原因（这里我们仍然以 `openat()` 为例）：

- 使用 `openat()` 系统调用允许应用避免 `open()` 可能出现的竞争条件，当使用 `open()` 打开不是当前工作目录位置的文件时就可能会出现竞争条件，因为调用 `open()` 时 `pathname` 路径中的某个目录可能会同时发生变化。通过先打开一个目标目录的文件描述符并传递给 `openat()`，就可以避免这种竞争条件。
- 在第 29 章，我们会看到当前工作目录是一个进程属性，被进程的所有线程共享。对于某些应用来说，不同线程拥有不同的“虚拟”工作目录可能会非常有用。应用可以维护一组目录的文件描述符，并使用 `openat()` 来模拟多个线程工作目录。

SUSv3 没有对这些系统调用进行标准化，但 SUSv4 已经包含了这些系统调用。要在应用中暴露这些系统调用的声明，必须定义 `_XOPEN_SOURCE` 特性测试宏的值大于或等于 700；或者也可以定义 `_POSIX_C_SOURCE` 宏的值大于或等于 200809。（在 glibc 2.10 之前，还必须定义 `_ATFILE_SOURCE` 宏，才能暴露这些系统调用的声明）。

Solaris 9 及之后的版本支持表 18-2 中系统调用的一部分，但语义稍有不同。

18.12 改变进程的根目录：chroot()

每个进程都有一个根目录（root directory），这是所有绝对路径（以 / 开始的路径）解释的起点。默认情况下，进程的根目录就是文件系统的根目录。（新进程会继承父进程的根目录）。某些情况下，进程可能会需要修改自己的根目录，特权进程（`CAP_SYS_CHROOT`）可以使用 `chroot()` 系统调用来实现。

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

int chroot(const char *pathname);
```

成功时返回 0；失败时返回 -1

`chroot()` 系统调用修改进程的根目录为 `pathname` 指定的路径（如果 `pathname` 是符号链接，会先进行解引用）。之后所有绝对路径都会相对这个路径进行解释。有时候我们会把修改根目录称为设置根监狱（chroot jail），因为此时程序只能访问文件系统的特定区域。

SUSv2 规定了 `chroot()` 并标记为遗留接口（LEGACY），但 SUSv3 移除了这个标记。无论如何，`chroot()` 在多数 UNIX 实现中都可用。

`chroot` 命令使用了 `chroot()` 系统调用，允许我们在 `chroot jail` 中执行 shell 命令。任何进程的根目录都可以通过读取（`readlink()`）Linux 特定的 `/proc/PID/root` 符号链接来得到。

`chroot()` 的典型例子是 `ftp` 程序，为了安全的考量，当用户匿名登录 FTP 时，`ftp` 程序会调用 `chroot()`，设置新进程的根目录为匿名登录的目录。在调用 `chroot()`

之后，匿名用户就限制在文件系统的某个特定子树，阻止他们访问整个文件系统。（注意根目录的父目录是自己，也就是说 `../` 指向的是 `/`，所以在根目录 `/` 中使用 `cd ..` 会让用户仍然停留在相同的目录。`chroot` 依赖于这一点才能限制用户始终受限于根目录范围之内）。

某些 UNIX 实现（但不包括 Linux）允许多个硬链接指向同一个目录，这样在子目录中就可以创建一个硬链接指向父目录（或者祖父目录）。在这样的系统中，这样的硬链接可以破坏 `chroot jail`；符号链接就不存在这样的问题，因为符号链接始终相对于进程的根目录来解释，它们不能到达 `chroot jail` 的外部。

通常很多程序都不能在 `chroot jail` 中执行，因为多数程序都动态链接到共享库。因此要么我们只能执行静态链接的程序；要么复制包含这些共享库的许多系统目录到 `chroot jail` 中（包括 `/lib`, `/usr/lib` 等），这种情况下，[14.9.4 节](#)介绍的绑定挂载特性就会非常有用。

注意 `chroot()` 系统调用并不是完全安全的监狱机制，例如特权程序完全可以再次使用 `chroot()` 来打破安全监狱的限制。特权程序（`CAP_MKNOD`）可以使用 `mknod()` 来创建一个内存设备文件（类似于 `/dev/mem`），允许访问 RAM 的内容，然后程序基本上就可以做任何事情。通常来说，在 `chroot jail` 文件系统中，不应该包含 `set-user-ID-root` 的程序。

即使是非特权程序，我们也必须非常小心，避免出现以下情况，导致 `chroot jail` 出现越狱：

- 调用 `chroot()` 不会改变进程的当前工作目录。因此 `chroot()` 和 `chdir()` 基本上必须同时使用（如调用 `chroot()` 之后紧接着再调用 `chdir("/")`）。如果你不这样做，进程就可以使用相对路径来访问 `chroot jail` 之外的文件和目录。（某些 BSD 变种会自动阻止这种情况的发生，如果当前工作目录处于根目录之外，`chroot()` 会修改进程的当前工作目录为新的根目录。）
- 如果进程已经打开了一个 `chroot jail` 之外的文件描述符，结合 `fchdir()` 再加上 `chroot()` 就可以完成越狱，如下面代码所示：

```
int fd;
fd = open("/", O_RDONLY);
chroot("/home/mtk");      /* 进入监狱 */
fchdir(fd);
```

```
chroot(".");
/* 越狱成功 */
```

要阻止这种可能性，我们必须关闭所有 chroot jail 之外的已打开文件描述符。(某些 UNIX 系统提供了 fchroot()系统调用，同样也可以达到上面代码的越狱目的)。

- 就算杜绝了上面几个问题，仍然不能确保阻止非特权程序越狱。对于那些我们无法直接控制的程序，它们可以使用 UNIX 域 socket 从另一个进程获得文件描述符，如果这个文件描述符处于 chroot jail 之外，进程仍然可以使用 fchdir()来设置当前工作目录，从而使用符号链接来完成越狱并访问任意文件。(我们会在 61.13.3 节简短地介绍如何使用 socket 在两个进程间传递文件描述符)。

某些 BSD 变种提供一个 jail()系统调用，实现了 chroot jail 的功能，同时还解决了上面提到的这些问题。使用 jail()创建的安全监狱对于特权进程也仍然是安全的。

18.13 resolve 路径名： realpath()

realpath()库函数解引用 pathname 路径中的所有符号链接，并 resolve 所有/. 和/..的引用，从而生成并返回相应的绝对路径 (null 终止的字符串)。

```
#include <stdlib.h>

char *realpath(const char *pathname, char *resolved_path);
成功时返回已解决的路径名；出错时返回 NULL
```

结果字符串存放在 resolved_path 缓冲区中，该缓冲区的大小至少应该等于 PATH_MAX。成功时 realpath()还会返回这个已解决的字符串。

glibc 版本的 realpath()允许调用者指定 resolved_path 为 NULL，此时 realpath()会分配一个最多 PATH_MAX 字节的缓冲区，并返回缓冲区指针作为函数结果。(调用者必须在使用完之后调用 free()释放这个缓冲区)。SUSv3 没有规定这个扩展行为，但 SUSv4 对此做出了明确规定。

清单 18-4 的程序使用 readlink()和 realpath()来读取一个符号链接的内容，并解决到该链接的绝对路径。下面是这个程序的使用示例：

```
$ pwd          我们在哪里?  
/home/mtk  
$ touch x      创建一个文件  
$ ln -s x y    添加一个符号链接指向它  
$ ./view_symlink y  
readlink: y --> x  
realpath: y --> /home/mtk/x
```

清单 18-4：读取并解决一个符号链接

```
-----dirs_links/view_symlink.c  
#include <sys/stat.h>  
#include <limits.h>      /* For definition of PATH_MAX */  
#include "tlpi_hdr.h"  
  
#define BUF_SIZE PATH_MAX  
  
int  
main(int argc, char *argv[])  
{  
    struct stat statbuf;  
    char buf[BUF_SIZE];  
    ssize_t numBytes;  
  
    if (argc != 2 || strcmp(argv[1], "--help") == 0)  
        usageErr("%s pathname\n", argv[0]);  
  
    if (lstat(argv[1], &statbuf) == -1)  
        errExit("lstat");  
  
    if (!S_ISLNK(statbuf.st_mode))  
        fatal("%s is not a symbolic link", argv[1]);  
  
    numBytes = readlink(argv[1], buf, BUF_SIZE - 1);  
    if (numBytes == -1)  
        errExit("readlink");  
    buf[numBytes] = '\0'; /* Add terminating null byte */  
    printf("readlink: %s --> %s\n", argv[1], buf);  
  
    if (realpath(argv[1], buf) == NULL)  
        errExit("realpath");  
    printf("realpath: %s --> %s\n", argv[1], buf);
```

```
    exit(EXIT_SUCCESS);
}
-----dirs_links/view_symlink.c
```

18.14 解析路径名字符串： dirname() 和 basename()

dirname() 和 basename() 函数分解路径名字符串为目录和文件名两个部分。
(这些函数执行的任务类似于 dirname(1) 和 basename(1) 命令)。

```
#include <libgen.h>
```

```
char *dirname(char *pathname);
char *basename(char *pathname);
```

两个函数都返回 `null` 终止的字符串 (可能是静态分配的)

例如对于路径名 /home/britta/prog.c， dirname() 函数返回 /home/britta； basename() 则返回 prog.c。我们把 dirname() 返回的字符串、斜杠、 basename() 返回的字符串连接起来就能得到完整的路径名。

使用 dirname() 和 basename() 的时候请注意以下关键点：

- 路径名末尾的斜杠会被忽略
- 如果 pathname 不包含任何斜杠，则 dirname() 返回字符串 "."； basename() 返回 pathname。
- 如果 pathname 只包含一个斜杠，则 dirname() 和 basename() 都返回字符串 "/"。按我们上面说的连接规则来重新组合路径名，会得到字符串 "///"。这也是一个合法的路径名。由于多个连续的斜杠等同于单个斜杠，路径名 "///" 实际上就是路径名 "/"。
- 如果 pathname 是 NULL 指针或空字符串，则 dirname() 和 basename() 都返回字符串 "."。(手工连接两个函数的返回值会得到 "./"，等同于当前目录 ".")。

表 18-3 举例说明了 dirname() 和 basename() 在不同输入时返回的字符串。

表 18-3: *dirname()* 和 *basename()* 返回字符串示例

路径名字符串	dirname()	basename()
/	/	/
/usr/bin/zip	/usr/bin	zip
/etc/passwd///	/etc	passwd
/etc///passwd	/etc	passwd
etc/passwd	etc	passwd
passwd	.	passwd
passwd/	.	passwd
..	.	..
NULL	.	.

清单 18-5: 使用 *dirname()* 和 *basename()*

```
-----dirs_links/t_dirbasename.c
#include <libgen.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    char *t1, *t2;
    int j;

    for (j = 1; j < argc; j++) {
        t1 = strdup(argv[j]);
        if (t1 == NULL)
            errExit("strdup");
        t2 = strdup(argv[j]);
        if (t2 == NULL)
            errExit("strdup");

        printf("%s ==> %s + %s\n", argv[j], dirname(t1), basename(t2));

        free(t1);
        free(t2);
    }
}
```

```
}

    exit(EXIT_SUCCESS);
}

-----dirs_links/t_dirbasename.c
```

`dirname()` 和 `basename()` 都可能会修改 `pathname` 指向的字符串。因此如果我们希望保留路径名字符串，就必须传递路径名的拷贝给 `dirname()` 和 `basename()` 函数，如上面清单 18-5 的代码所示。这个程序使用 `strdup()`（内部会调用 `malloc()`）来复制字符串，然后再传递给 `dirname()` 和 `basename()` 函数，上面代码最后使用 `free()` 来释放 `strdup()` 复制出来的字符串。

最后，注意 `dirname()` 和 `basename()` 有可能会返回静态分配的字符串，重复调用 `dirname()` 或 `basename()` 可能会修改之前相同函数返回的字符串。

18.15 小结

`i-node` 不包含文件名信息，系统通过目录中的项（`entry`）来赋予文件名，目录项列出了文件名和相应的 `i-node` 数值。这样的目录项称为（硬）链接。文件可以有多个硬链接，所有链接都共享相同的状态。我们可以使用 `link()` 和 `unlink()` 来创建和删除链接。使用 `rename()` 系统调用可以重命名文件。

使用 `symlink()` 可以创建符号链接（软链接）。符号链接在某些方面类似于硬链接，但符号链接区别于硬链接的地方在于：符号链接可以跨越文件系统边界，也可以引用到目录。符号链接本质上只是一个包含另一个文件名的文件；这个名字可以通过 `readlink()` 来获取。符号链接不会包含在目标文件 `i-node` 的链接计数中，如果目标文件被删除，则符号链接就是摇摆的（`dangling`）。某些系统调用会自动解引用符号链接；另外一些系统调用则不会进行解引用。有时候这两种版本的系统调用都存在，其中一个解引用，另一个则不会解引用。例如 `stat()` 和 `Istat()`。

使用 `mkdir()` 和 `rmdir()` 可以创建和删除目录。使用 `opendir()`、`readdir()` 以及相关的函数，我们可以遍历一个目录。`nftw()` 函数则允许程序遍历整个目录树，对目录树中的每个文件，都会调用我们指定的处理函数。

`remove()` 函数可以删除一个文件（链接）或者一个空目录。

每个进程都有一个根目录，决定了解释绝对路径的起点；还有一个当前工作目录，决定了解释相对路径的起点。`chroot()`和 `chdir()`系统调用可以修改进程的根目录和当前工作目录。`getcwd()`函数则返回进程的当前工作目录。

Linux 提供一组新的系统调用（如 `openat()`），行为类似于相应的传统系统调用（如 `open()`），但是这些函数可以提供一个目录文件描述符，用来指定相对路径解释的起点（而不是使用进程的当前工作目录来解释相对路径）。使用这些新的系统调用可以避免传统系统调用存在的某些竞争条件，也可以实现单个线程的虚拟工作目录。

`realpath()`函数可以解决到实际的绝对路径：解引用所有符号链接，并解决所有“.”和“..”到相应的目录。`dirname()`和 `basename()`函数可以分离路径名为目录和文件名两个部分。

18.16 习题

18-1. 在 [4.3.2 节](#)，我们提到过如果文件正在被执行，就没有办法再打开这个文件（`open()`返回-1，`errno` 设置为 `ETXTBSY`）。但是在 shell 中，我们可以执行如下操作：

```
$ cc -o longrunner longrunner.c
$ ./longrunner &                                保持在后台运行
$ vi longrunner.c                               对源代码做一些修改
$ cc -o longrunner longrunner.c
```

最后一条命令会覆盖正在被执行的文件，为什么可以这样做呢？（小小的提示，每次编译后使用 `ls -li` 查看可执行文件的 i-node 数值）。

18-2. 为什么下面代码调用 `chmod()`会失败呢？

```
mkdir("test", S_IRUSR | S_IWUSR | S_IXUSR);
chdir("test");
fd = open("myfile", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
symlink("myfile", "../mylink");
chmod("../mylink", S_IRUSR);
```

18-3. 实现 `realpath()`。

- 18-4. 修改清单 18-2 (`list_files.c`) 的程序，使用 `readdir_r()` 而不是 `readdir()`。
- 18-5. 编写一个函数实现 `getcwd()` 的功能。给你一个有用的提示，使用 `opendir()` 和 `readdir()` 来遍历父目录 `(..)` 的每个项，找出其中 i-node 和设备编号相同的那个目录项(也就是 `stat()` 或 `lstat()` 返回的 `st_ino` 和 `st_dev` 两个域)，它就是当前工作目录。这时候我们就已经得到了当前工作目录的文件名。然后我们再调用 `chdir("..")`，并再次执行上述遍历和扫描来得到父目录的文件名；重复上述步骤遍历整个目录树就可以得到整个路径的所有目录名，当父目录和当前工作目录相同时我们就可以终止遍历（记得 `../` 等同于 `/`）。把所有得到的目录名组合起来就是我们需要的结果。你的函数无论成功或失败，都必须保证不会改变进程原来的当前工作目录（`open()` 加上 `fchdir()` 可以很好地完成这个任务）。
- 18-6. 修改清单 18-3 的程序 (`nftw_dir_tree.c`) 使用 `FTW_DEPTH` 标志。注意修改后程序遍历目录树的顺序有何差别。
- 18-7. 编写一个程序，使用 `nftw()` 来遍历一个目录树，打印出各种文件类型的计数和百分比（普通文件、目录、符号链接、等等）。
- 18-8. 实现 `nftw()` 函数。（你会使用到 `opendir()`, `readdir()`, `closedir()`, `stat()` 等系统调用）。
- 18-9. 在 [18.10 节](#)，我们使用了两种技术 (`fchdir()` 和 `chdir()`) 在修改当前工作目录之后，再次返回到之前的当前工作目录。假设我们需要反复执行这个操作，你觉得哪个方法更高效？为什么？请编写一个程序来验证你的答案。

第 19 章 监控文件事件

某些应用需要监控文件或目录，确定这些被监控的对象是否发生特定的事件。例如图形化的文件管理器，在显示某个目录下的文件内容时，还需要能够确定目录中是否有文件被添加和删除；另外 daemon 也可能需要监控自己的配置文件，从而在文件修改的时候重新装载配置信息。

从内核 2.6.13 开始，Linux 开始提供 inotify 机制，允许应用监控文件事件，本章会详细讨论 inotify 的使用方法。

inotify 机制替换了老式的 dnotify 机制，后者提供的功能只是 inotify 的子集。本章末尾我们会简短地介绍 dnotify，并且说明为什么 inotify 更好。

inotify 和 dnotify 机制都是 Linux 特定的。（有些 UNIX 系统也提供类似的机制，例如 BSD 提供了 kqueue API）。

有一些库提供了类似的功能，但是 API 比 inotify 和 dnotify 更加抽象和可移植。对于某些应用来，可以优先考虑使用这些上层库。这些库内部使用了 inotify 或 dnotify（如果可用的话）。两个比较有名的库分别是：FAM（File Alteration Monitor，<http://oss.sgi.com/projects/fam/>）；Gamin（<http://www.gnome.org/~veillard/gamin/>）

19.1 概述

使用 inotify API 的关键步骤如下：

1. 应用使用 `inotify_init()` 来创建一个 inotify 实例。系统调用会返回一个文件描述符，我们在随后的操作需要使用这个描述符。
2. 应用使用 `inotify_add_watch()` 来添加文件项到监控列表，通知内核自己感兴趣的文件及事件。每个监控项都由路径名和相关联的位掩码组成。掩码指定了需要对该路径监控的具体事件。`inotify_add_watch()` 返回一个监控描述符，用于后面的操作。`inotify_add_watch()` 系统调用也可以执行相反的删除操作，将之前添加的监控项从 inotify 实例中删除。
3. 要获得事件通知，应用需要对 inotify 文件描述符执行 `read()` 操作。成功时 `read()` 返回一个或多个 `inotify_event` 结构体，每个结构体包含了一个文件的一个事件的详细信息。

4. 当应用完成监控时，需要关闭 `inotify` 文件描述符。这样做会自动删除该 `inotify` 实例的所有监控项。

`inotify` 机制可以用来监控文件和目录。当监控一个目录时，应用会得到目录本身、以及目录下所有文件的事件通知。

`inotify` 监控机制不是递归的，如果应用希望监控整个目录树的事件，必须对目录树中的每个目录都调用 `inotify_add_watch()`。

`inotify` 文件描述符可以使用 `select()`、`poll()`、`epoll` 来进行监控，从 Linux 2.6.25 开始，还可以使用信号驱动 I/O。如果有事件可读，这些接口就会报告 `inotify` 文件描述符可读。这些接口的更多信息请参考第 63 章

`inotify` 机制是 Linux 内核的一个可选组件，你可以通过内核的 `CONFIG_INOTIFY` 和 `CONFIG_INOTIFY_USER` 选项来进行配置。

19.2 `inotify` API

`inotify_init()` 系统调用创建一个新的 `inotify` 实例。

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

int inotify_init(void);
```

成功时返回文件描述符；出错时返回 -1

`inotify_init()` 成功时返回一个文件描述符，在随后的操作中使用这个句柄来访问 `inotify` 实例。

从内核 2.6.27 开始，Linux 支持一个新的非标准系统调用：`inotify_init1()`。这个系统调用的功能与 `inotify_init()` 类似，但提供了一个额外的 `flags` 参数，可以用来修改系统调用的行为。`flags` 支持两个标志，`IN_CLOEXEC` 标志通知内核为新的文件描述符启用 `close-on-exec` 标志 (`FD_CLOEXEC`)。第 4.3.1 节 我们介绍了 `open()` 的 `O_CLOEXEC` 标志，两者的原理是一样的。另外一个标志是 `IN_NONBLOCK` 标志，通知内核为底层文件描述符启用 `O_NONBLOCK` 标志，以后的读取操作都将是非阻塞的。这样无需额外调用 `fcntl()` 就可以达到同样的结果。

`inotify_add_watch()` 系统调用添加或删除一个监控项，`fd` 参数是引用 `inotify` 实例的文件描述符（参见图 19-1）。

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

int inotify_add_watch(int fd, const char *pathname, uint32_t mask);
成功时返回监控描述符；出错时返回-1
```

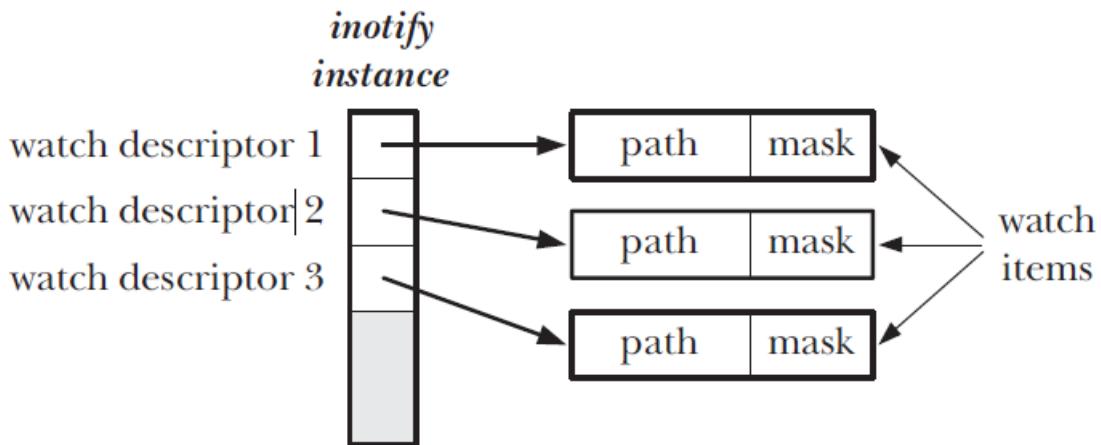


图 19-1: *inotify* 实例和相关的内核数据

`pathname` 参数指定要创建或修改监控项对应的文件，调用方必须对该文件具有读取权限（文件权限检查只会执行一次，也就是调用 `inotify_add_watch()` 期间，只要监控项一直存在，调用方就会持续地收到文件通知，即使文件权限已经改变，调用方不再拥有文件的读取权限）。

`mask` 参数是位掩码，指定要监控的事件。我们马上就会详细讨论监控事件对应的每一个位掩码。

如果 `pathname` 之前没有添加到 `fd` 描述符的监控列表，则 `inotify_add_watch` 会创建一个新的监控项，将其添加到监控列表中，并返回这个新创建的非负的监控描述符，后续操作需要使用到这个监控描述符。在单个 `inotify` 实例内部，所有监控描述符都是唯一的。

如果 `pathname` 之前已经添加到 `fd` 描述符的监控列表，则 `inotify_add_watch` 会修改 `pathname` 相应的现有监控项的掩码，并返回该监控项的监控描述符。（这里返回的监控描述符和最开始调用 `inotify_add_watch` 添加监控项时返回的描述符是一样的）。我们下一节讨论 `IN_MASK_ADD` 标志时会详细讲解如何修改掩码。

`inotify_rm_watch()` 系统调用从 `fd` 指定的 `inotify` 实例中删除 `wd` 指定的监控项。

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

int inotify_rm_watch(int fd, uint32_t wd);
```

成功时返回 0；出错时返回 -1

`wd` 参数是一个监控描述符，之前调用 `inotify_add_watch()` 时得到。`(uint32_t)` 数据类型是无符号型 32 位整数)。

删除监控项会导致相应的监控描述符触发一个 `IN_IGNORED` 事件，我们马上就会详细讨论。

19.3 inotify 事件

当我们使用 `inotify_add_watch()` 创建或修改监控项时，位掩码参数指定了该路径要监控的各种事件。掩码中可以指定的事件位如表 19-1 中“输入事件”一列所示：

表 19-1：`inotify` 事件

位掩码	输入事件	输出事件	描述
<code>IN_ACCESS</code>	●	●	文件被访问 (<code>read()</code>)
<code>IN_ATTRIB</code>	●	●	文件元数据发生变化
<code>IN_CLOSE_WRITE</code>	●	●	为写入而打开的文件被关闭
<code>IN_CLOSE_NOWRITE</code>	●	●	为只读而打开的文件被关闭
<code>IN_CREATE</code>	●	●	监控目录下创建了文件/目录
<code>IN_DELETE</code>	●	●	监控目录下删除了文件/目录
<code>IN_DELETE_SELF</code>	●	●	监控文件/目录本身被删除
<code>IN MODIFY</code>	●	●	文件被修改
<code>IN_MOVE_SELF</code>	●	●	监控文件/目录本身被移动

IN_MOVED_FROM	●	●	文件移出被监控目录
IN_MOVED_TO	●	●	文件被移入监控目录
IN_OPEN	●	●	文件被打开
IN_ALL_EVENTS	●		上面所有事件
IN_MOVE	●		IN_MOVED_FROM IN_MOVED_TO
IN_CLOSE	●		IN_CLOSE_WRITE IN_CLOSE_NOWRITE
IN_DONT_FOLLOW	●		不要解引用符号链接 (Linux 2.6.15 起)
IN_MASK_ADD	●		添加事件到 pathname 当前的监控掩码
IN_ONESHOT	●		只监控 pathname 一次事件
IN_ONLYDIR	●		如果 pathname 不是目录则失败 (Linux 2.6.15)
IN_IGNORED		●	应用或内核已经移除该监控
IN_ISDIR		●	name 返回的文件名是一个目录
IN_Q_OVERFLOW		●	事件队列已经溢出
IN_UNMOUNT		●	文件系统已经被卸载

表 19-1 的多数掩码的含义从名字都可以清晰看出。下面则更进一步地阐明一些细节：

- IN_ATTRIB 事件触发于文件元数据变化，例如权限、所有权、链接计数、扩展属性、用户 ID、组 ID 等发生变化。
- IN_DELETE_SELF 事件触发于被监控的对象（文件或目录）本身被删除。
IN_DELETE 事件则表示监控对象是目录，当目录中某个文件被删除时就会触发 IN_DELETE 事件。
- IN_MOVE_SELF 事件触发于被监控对象本身被重命名（移动）。而 IN_MOVED_FROM 和 IN_MOVED_TO 则表示被监控对象是一个目录，前者表示该目录下有文件移出，后者表示该目录下有文件移入。

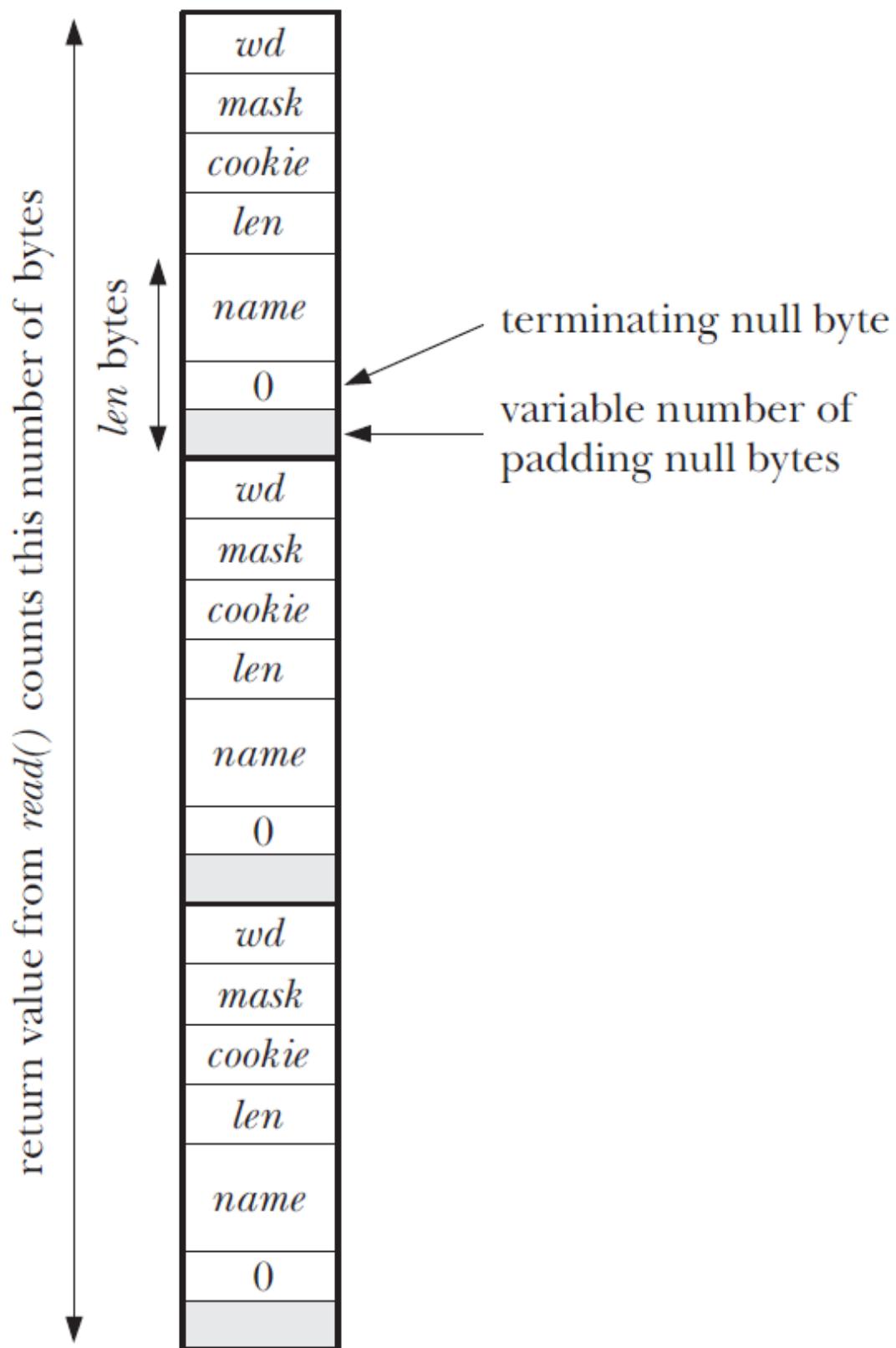
- `IN_DONT_FOLLOW`、`IN_MASK_ADD`、`IN_ONESHOT`、`IN_ONLYDIR` 四个掩码不用来指定要监控的事件，而是用来控制 `inotify_add_watch()` 的相关操作。
- `IN_DONT_FOLLOW` 指定 `pathname` 如果是符号链接，不要对其进行解引用。这允许应用监控符号链接文件，而不是链接引用的文件。
- 如果我们对已经被监控的 `pathname` 再次调用 `inotify_add_watch()`，并且指定相同的 `inotify` 文件描述符，则默认情况下 `inotify_add_watch()` 会使用新的掩码替换之前的掩码。我们可以指定 `IN_MASK_ADD` 标志，可以将新掩码与原来的掩码“或”起来，添加到一起而不是替换。
- `IN_ONESHOT` 允许应用只监控 `pathname` 的一次事件。在事件到达之后，监控项会自动从监控列表中删除。
- `IN_ONLYDIR` 允许应用只在 `pathname` 是目录时才对其进行监控。如果 `pathname` 不是目录，则 `inotify_add_watch()` 会以 `ENOTDIR` 错误失败。如果我们想先确定某个对象是一个目录，然后才对其进行监控，则使用这个标志可以避免竞争条件。

19.4 读取 inotify 事件

在监控列表中注册好项目之后，应用可以通过 `read()` 读取 `inotify` 文件描述符，从而确定相关的文件事件。如果暂时还没有事件发生，`read()` 会阻塞直到某个事件发生（除非对文件描述符设置了 `O_NONBLOCK` 状态标志，这时如果没有事件发生，`read()` 会立即失败，并返回 `EAGAIN` 错误）。

当事件发生后，每次 `read()` 调用将返回一个缓冲区（图 19-2），包含一个或多个以下类型的结构体：

```
struct inotify_event {  
    int wd;           /* 事件发生的监控描述符 */  
    uint32_t mask;    /* 所发生事件的位掩码 */  
    uint32_t cookie;  /* 相关事件的 Cookie (rename()) */  
    uint32_t len;     /* name 域的大小 */  
    char name[];     /* null 终止的可选文件名 */  
};
```

图 19-2：包含三个 *inotify_event* 结构体的缓冲区

`wd` 域告诉我们事件发生的监控描述符，它的值也就是之前某次调用 `inotify_add_watch()` 返回的描述符。当应用通过同一个 `inotify` 文件描述符监控多个文件或目录时，`wd` 域就非常有用。应用根据 `wd` 的值可以确定发生事件的特定文件或目录（应用必须自己维护监控描述符映射到路径的数据结构）。

`mask` 域返回描述事件的位掩码。表 19-1 的输出事件列包含了所有可能出现的掩码，请注意以下特定位掩码的描述：

- `IN_IGNORED` 事件发生在删除监控时。两种情况下可能会发生：应用使用 `inotify_rm_watch()` 调用显式地删除一个监控；或者内核隐式地删除监控，因为监控对象已经被删除或文件系统已经被卸载。当使用 `IN_ONESHOT` 来建立监控时，不会发生 `IN_IGNORED` 事件。因为 `IN_ONESHOT` 在发出第一个事件之后就会自动删除该监控。
- 如果事件的目标是一个目录，则除了其它相关的事件掩码，还一定会包含 `IN_ISDIR` 位掩码。
- `IN_UNMOUNT` 事件通知应用包含监控对象的文件系统已经被卸载。在这个事件之后，还会触发一个 `IN_IGNORED` 事件。
- 我们在[第 19.5 节](#)讨论 `IN_Q_OVERFLOW`，在那里我们会讨论 `inotify` 事件的队列限制。

`cookie` 域用来绑定相关的事件到一起，目前这个域仅用于文件被重命名。当文件被重命名时，会对文件原来的目录触发一个 `IN_MOVED_FROM` 事件，然后对文件的新目录触发一个 `IN_MOVED_TO` 事件（如果文件只是改变名字，并不改变目录，则上面两个事件都发生在同一个目录上）。这两个事件的 `cookie` 域将是相同的唯一值，应用可以通过这个值来关联这两个事件。

当监控对象是目录，当目录下的某个文件触发事件时，`name` 域返回 `null` 终止的字符串，用于标识该文件。如果监控对象本身触发了事件，则 `name` 域未使用，同时 `len` 域的值为 0。

`len` 域用来指示 `name` 域中分配的字节数。这个域是必要的，因为当 `read()` 读取到多个 `inotify_event` 结构体时，前一个 `name` 域存储的字符串到下一个

`inotify_event` 结构体之间可能会存在额外的填充字节（参考图 19-2）。所以单个 `inotify` 事件的大小实际上是 `sizeof(struct inotify_event) + len`。

如果传递给 `read()` 的缓冲区太小，无法存储下一个 `inotify_event` 结构体，则 `read()` 会以 `EINVAL` 错误失败（在内核 2.6.21 之前，`read()` 在这种情况下会返回 0，现在改成返回 `EINVAL` 错误，能够更好地通知应用发生了程序错误）。应用检测到 `EINVAL` 错误时，可以提供更大的缓冲区，并再次调用 `read()`。不过我们完全可以避免出现这个问题，只要确保提供给 `read()` 的缓冲区足够大，至少能够保存一个事件：大小至少为 `(sizeof(struct inotify_event) + NAME_MAX + 1)` 字节，其中 `NAME_MAX` 是文件名的最大长度，最后再加上一个 `null` 终止字节。

使用更大的缓冲区允许应用一次调用 `read()` 能够接收多个事件。调用 `read()` 读取 `inotify` 文件描述符返回的事件数量，由当前可用的事件数，以及缓冲区能够存储的事件数共同决定。

调用 `ioctl(fd, FIONREAD, &numbytes)`，可以返回文件描述符 `fd` 引用的 `inotify` 实例当前可以读取的字节数。

从 `inotify` 文件描述符读取出来的事件是有序的队列，例如当文件被重命名时，`IN_MOVED_FROM` 事件总是在 `IN_MOVED_TO` 事件之前。

当内核添加新的事件到事件队列末尾时，如果新事件与队列末尾的事件拥有相同的 `wd`, `mask`, `cookie`, `name` 值，内核就会将这两个事件合并。这样应用就不会接收到重复的相同事件，并且丢弃多余的事件也能减少事件队列占用的（内核）内存。但是这也意味着我们不能使用 `inotify` 来可靠地监控一个周期性事件发生的具体次数。

示例程序

尽管前面描述了相当多细节，但是 `inotify` API 其实使用非常简单。清单 19-1 演示了 `inotify` 的用法：

清单 19-1：使用 inotify API

```
-----inotify/demo_inotify.c
#include <sys/inotify.h>
#include <limits.h>
#include "tlpi_hdr.h"

static void /* 打印 inotify_event 结构体的信息 */
displayInotifyEvent(struct inotify_event *i)
{
    printf(" wd =%2d; ", i->wd);
    if (i->cookie > 0)
        printf("cookie =%4d; ", i->cookie);

    printf("mask = ");
    if (i->mask & IN_ACCESS) printf("IN_ACCESS ");
    if (i->mask & IN_ATTRIB) printf("IN_ATTRIB ");
    if (i->mask & IN_CLOSE_NOWRITE) printf("IN_CLOSE_NOWRITE ");
    if (i->mask & IN_CLOSE_WRITE) printf("IN_CLOSE_WRITE ");
    if (i->mask & IN_CREATE) printf("IN_CREATE ");
    if (i->mask & IN_DELETE) printf("IN_DELETE ");
    if (i->mask & IN_DELETE_SELF) printf("IN_DELETE_SELF ");
    if (i->mask & IN_IGNORED) printf("IN_IGNORED ");
    if (i->mask & IN_ISDIR) printf("IN_ISDIR ");
    if (i->mask & IN MODIFY) printf("IN MODIFY ");
    if (i->mask & IN_MOVE_SELF) printf("IN_MOVE_SELF ");
    if (i->mask & IN_MOVED_FROM) printf("IN_MOVED_FROM ");
    if (i->mask & IN_MOVED_TO) printf("IN_MOVED_TO ");
    if (i->mask & IN_OPEN) printf("IN_OPEN ");
    if (i->mask & IN_Q_OVERFLOW) printf("IN_Q_OVERFLOW ");
    if (i->mask & IN_UNMOUNT) printf("IN_UNMOUNT ");
    printf("\n");

    if (i->len > 0)
        printf(" name = %s\n", i->name);
}

#define BUF_LEN (10 * (sizeof(struct inotify_event) + NAME_MAX + 1))

int
main(int argc, char *argv[])
{
    int inotifyFd, wd, j;
```

```
char buf[BUF_LEN];
ssize_t numRead;
char *p;
struct inotify_event *event;

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

① inotifyFd = inotify_init();           /* 创建 inotify 实例 */
if (inotifyFd == -1)
    errExit("inotify_init");

for (j = 1; j < argc; j++) {
②     wd = inotify_add_watch(inotifyFd, argv[j], IN_ALL_EVENTS);
    if (wd == -1)
        errExit("inotify_add_watch");

    printf("Watching %s using wd %d\n", argv[j], wd);
}

for (;;) {                                /* 一直读取事件 */
③     numRead = read(inotifyFd, buf, BUF_LEN);
    if (numRead == 0)
        fatal("read() from inotify fd returned 0!");

    if (numRead == -1)
        errExit("read");

    printf("Read %ld bytes from inotify fd\n", (long) numRead);

/* 处理 read() 返回缓冲区中的所有事件 */
for (p = buf; p < buf + numRead; ) {
    event = (struct inotify_event *) p;
    displayInotifyEvent(event);

    p += sizeof(struct inotify_event) + event->len;
}
}

exit(EXIT_SUCCESS);
}

-----inotify/demo_inotify.c
```

清单 19-1 的程序执行了以下步骤：

- 使用 `inotify_init()` 创建一个 `inotify` 文件描述符①
- 使用 `inotify_add_watch()` 为命令行参数指定的每个文件添加一个监控项②。每个监控项都监控所有可能的事件。
- 执行无限循环：
 - 从 `inotify` 文件描述符读取事件缓冲区③
 - 调用 `displayInotifyEvent()` 函数来显示缓冲区中每个 `inotify_event` 结构体的内容④

下面 shell 会话演示了清单 19-1 程序的使用。首先我们在后台运行该程序，并监控两个目录：

```
$ ./demo_inotify dir1 dir2 &
[1] 5386
Watching dir1 using wd 1
Watching dir2 using wd 2
```

然后我们执行一些命令，在这两个目录中生成一些事件。首先使用 `cat(1)` 创建一个文件：

```
$ cat > dir1/aaa
Read 64 bytes from inotify fd
    wd = 1; mask = IN_CREATE
        name = aaa
    wd = 1; mask = IN_OPEN
        name = aaa
```

上面的输出是由后台程序打印的，显示了 `read()` 获得的缓冲区中的两个事件的信息。我们为文件输入一些内容，最后添加一个 end-of-file 字符：

```
Hello world
Read 32 bytes from inotify fd
    wd = 1; mask = IN_MODIFY
        name = aaa
Type Control-D
Read 32 bytes from inotify fd
    wd = 1; mask = IN_CLOSE_WRITE
        name = aaa
```

然后我们重命名文件到另一个监控目录，这会产生两个事件，一个产生于移出文件的目录（监控描述符 1），另一个产生于移入文件的目录（监控描述符 2）：

```
$ mv dir1/aaa dir2/bbb
Read 64 bytes from inotify fd
    wd = 1; cookie = 548; mask = IN_MOVED_FROM
        name = aaa
    wd = 2; cookie = 548; mask = IN_MOVED_TO
        name = bbb
```

这两个事件的 cookie 值相同，因此应用可以链接这两个事件。

当我们在监控目录下创建子目录时，事件的 mask 会包含 IN_ISDIR 位，表示事件是由目录触发的：

```
$ mkdir dir2/ddd
Read 32 bytes from inotify fd
    wd = 1; mask = IN_CREATE IN_ISDIR
        name = ddd
```

现在我们需要重申：inotify 监控不是递归的。如果应用希望监控新创建的子目录，就需要再次发起一次 `inotify_add_watch()` 调用，并指定子目录的路径。

最后，我们删除其中一个监控目录：

```
$ rmdir dir1
Read 32 bytes from inotify fd
    wd = 1; mask = IN_DELETE_SELF
    wd = 1; mask = IN_IGNORED
```

最后一个 IN_IGNORED 事件，用来通知应用内核已经从监控列表中移除这个监控项。

19.5 队列限制和/proc 文件

inotify 事件队列需要使用内核内存。因此内核对 inotify 机制设置了一些限制。超级用户可以通过`/proc/sys/fs/inotify` 目录下的三个文件，来配置这些限制：

max_queued_events

调用 `inotify_init` 系统调用时，这个值可以设置新创建的 `inotify` 实例的事件队列的最大事件数量限制。如果达到了这个限制，就会触发 `IN_Q_OVERFLOW` 事件，并丢弃后面其余的事件。溢出事件的 `wd` 域的值总是-1。

max_user_instances

每个实际用户 ID 能够创建的 `inotify` 实例的上限。

max_user_watches

每个实际用户 ID 能够创建的监控项的上限。

这三个限制的典型值分别为 16384、128、8192。

19.6 监控文件事件的旧机制：dnotify

Linux 还提供另一个监控文件事件的机制，被称为 `dnotify`，从内核 2.4 开始可用，但目前已经被废弃，并由 `inotify` 取代。`dnotify` 机制相比 `inotify` 机制存在许多局限：

- `dnotify` 机制通过向应用发送信号来进行事件通知。使用信号作为通知机制会导致应用设计更加复杂（[第 22.12 节](#)）。而且在库（library）里面使用 `dnotify` 也会更加困难，因为调用程序可能会改变通知信号的部署。而 `inotify` 机制则完全不使用信号。
- `dnotify` 的监控单元是目录，该目录下任何文件的操作都会向应用触发事件。相反 `inotify` 可以用来监控目录，也可以监控单个文件。
- 要监控一个目录，`dnotify` 要求应用为该目录打开一个文件描述符。使用目录文件描述符会导致两个问题，首先由于占用了该目录，文件系统将无法被卸载；其次由于每个目录都需要一个文件描述符，应用可能会消耗大量的文件描述符。`inotify` 不使用文件描述符，因此避免了这些问题。
- `dnotify` 提供的文件事件的信息不如 `inotify` 精确。当监控目录下的文件发生变化时，`dnotify` 会告诉我们发生了一个事件，但不会告诉我们哪个文件发生了变化。应用必须缓存目录的所有内容，才能确定具体是哪个文

件发生了变化。此外，`inotify` 提供的事件类型比 `dnotify` 也更加详细。

- 在某些情况下，`dnotify` 无法对文件事件提供可靠的通知。

`dnotify` 的更多信息，可以在 `fcntl(2)` 手册页下的 `F_NOTIFY` 操作描述中找到，内核源文件 `Documentation/dnotify.txt` 也有详细描述。

19.7 小结

Linux 特定的 `inotify` 机制允许应用获得文件事件的通知，包括文件打开、关闭、创建、删除、修改、重命名等等。`inotify` 可以监控单个或一组文件和目录。目前 `inotify` 机制已经取代了老的 `dnotify` 机制。

19.8 习题

- 19-1. 编写一个程序，监控命令行参数指定的目录，记录该目录下所有文件创建、删除、和重命名。程序需要监控指定目录下所有子目录的事件。要获取这些子目录的事件，你需要使用 `nftw()` ([第 18.9 节](#))。当新目录添加到指定目录树，或者目录树下有目录被删除时，你还需要相应地更新监控子目录集。

第 20 章 信号：基础概念

本章和接下来的两章讨论信号，尽管信号的基本概念非常简单，我们的讨论却非常长，因为需要覆盖的细节非常多。

本章讲解以下主题：

- 不同类型的信号及其用途；
- 内核可能向进程发送信号的各种场景，以及进程向其它进程发送信号的系统调用；
- 进程默认如何响应一个信号，以及进程如何改变信号的处理方式：使用信号处理器，当信号到来时自动调用程序员自定义的函数；
- 进程如何使用掩码来阻塞信号，以及未决信号的相关概念；
- 进程如何暂停执行并等待信号到达。

20.1 概念和概述

每个信号都是向进程通知有一个事件已经发生，信号有时候也被称为软中断，类似于硬件中断，可以中断一个程序的正常执行流程。多数情况下，我们不可能预测信号什么时候会到来。

进程可以（如果拥有适当的权限）向其它进程发送信号。按这种用法，信号可以作为一种同步机制，甚至作为一种原始的进程间通信（IPC）机制。当然进程也可以给自己发送信号。不过向进程发送信号最常见的来源是内核，内核在很多情况下会向进程发送各种信号，包括：

- 发生硬件异常，意味着硬件检测到一个错误条件（fault condition）并通知给内核，而内核则向相关的进程发送相应的信号。硬件异常的例子包括执行了非法的机器语言指令、除以 0、引用不可访问的内存等。
- 用户按下某个终端特殊字符而产生信号，包括中断字符（通常是 Ctrl-C）、暂停字符（通常是 Ctrl-Z）。
- 发生了软件事件。例如文件描述符的输入可用、终端窗口大小调整、定时器超时、进程 CPU 时间超过限制、子进程终止等。

每种信号都定义为唯一的（小）整数，从 1 开始顺序往下排。这些整数定义在<signal.h>头文件中，符号名的格式为 SIGxxxx。由于各个平台实现为每种信号定义的实际数值不尽相同，应用程序总是应该使用这些符号名，而不是直接使用数值。例如当用户按下中断字符时，会向进程发送 SIGINT（信号 2）。

信号可以分为两种大的类型，第一种是传统或标准信号，内核用来通知进程各种事件。在 Linux 中，标准信号的编号从 1 到 31。这一章我们讨论的就是标准信号；另一种是实时信号，[第 22.8 节](#)讨论了实时信号与标准信号的区别。

信号是由某种事件产生的，一旦生成了信号，随后就会被递送给一个进程，进程则采用某种动作来响应该信号。信号生成和信号递送之间的这段时间，该信号就称为未决（pending）。

通常一个未决信号会被尽可能快地递送给一个进程。如果进程正在运行（如进程向自己发送信号），就会立即递送；如果进程处于等待执行状态，就会在下一次被调度执行时递送信号。但是有时候我们需要确保一段代码不会被信号递送所中断，这时候可以把某些信号添加到进程的信号掩码，阻塞这个信号集中的信号递送。如果此时产生了这种信号，就会一直保持未决状态，直到信号被解锁（从信号掩码中移除）。许多系统调用可以允许进程向信号掩码中添加和删除信号。

信号递送到进程时，根据不同的信号类型，进程可能会采取以下默认动作：

- 忽略信号。内核直接丢弃该信号，对进程没有任何影响（进程完全不知道发生了这件事）。
- 进程被终止（killed）。有时候也称为进程异常终止，这与进程使用 `exit()` 来正常终止是相反的。
- 生成 core dump 文件，然后进程被终止。core dump 是包含进程虚拟内存镜像的文件，可以装载到调试器中，从而检查进程终止时的各种状态。
- 进程被停止，也就是进程被停止执行。
- 进程从之前的停止状态恢复执行。

除了接受特定信号的默认动作，程序也可以改变信号被送达时的动作。也就是设置信号部署（disposition）。程序可以为信号设置以下某个部署：

- 采用信号的默认动作。如果之前修改过信号的部署，可以使用这个方法来撤销之前的修改。
- 忽略信号。对于很多默认动作为终止进程的信号来说，忽略信号是很有用的。
- 执行信号处理器。

信号处理器是一个函数，由程序员编写，用于响应信号递送并执行适当的任务。例如 `shell` 为 `SIGINT`（中断字符 `Ctrl-C` 触发）信号设置了一个处理器，用于停止当前正在执行的操作并返回到主输入循环（`main input loop`），这样用户就可以重新回到 `shell` 提示符下。告诉内核信号到达时需要执行某个处理器函数的过程，通常被称为安装或建立一个信号处理器。信号到达时调用信号处理器就被称为信号被处理，或者信号被捕获。

注意我们不能设置信号部署为终止，也不能设置为 `core dump`（除非信号的默认动作就是这样）。如果要达到类似的效果，我们需要安装一个处理器，并在处理器中调用 `exit()` 或 `abort()`。`abort()` 函数（[第 21.2.2 节](#)）会触发一个 `SIGABRT` 信号，并导致进程 `core dump` 并终止。

Linux 特定的 `/proc/PID/status` 文件包含各种位掩码域，可以用来确定进程对信号的处理方式。位掩码域显示为十六进制数值，最低位代表信号 1，下一位则代表信号 2，依次类推。这些域分别是 `SigPnd`（线程级未决信号）、`ShdPnd`（进程级未决信号，Linux 2.6 开始）、`SigBlk`（阻塞的信号）、`SigIgn`（忽略的信号）、`SigCgt`（捕获的信号）。（`SigPnd` 和 `ShdPnd` 域的区别，我们会在[第 33.2 节](#)讨论多线程处理信号时详细描述）。我们也可以在 `ps(1)` 命令的各种选项中找到相关的信息。

很早期的 UNIX 系统就已经出现信号的概念，随后又经历了许多重大的变化。在早期系统中，信号在某些情况下可能会丢失（无法递送至目标进程）。此外，尽管提供了执行特定代码时阻塞信号递送的机制，但在某些情况下，阻塞机制并不可靠。这些问题在 4.2BSD 中得到了解决，从而提供了所谓的可靠信号机制。（此外，BSD 还创新地使用额外的信号来支持 shell 作业控制，我们会在[第 34.7 节](#)讨论这个主题）。

System V 系统也实现了可靠信号机制，但却采用了与 BSD 不兼容的模型。直

到 POSIX.1-1990 标准的出现才解决这些兼容性问题，该标准大体上基于 BSD 模型来对信号进行了规范。

我们在[第 22.7 节](#)讨论可靠和不可靠信号的细节，并且在[第 22.13 节](#)简要地描述老的 BSD 和 System V 的信号 API。

20.2 信号类型和默认动作

前面我们提到 Linux 标准信号的编号从 1 到 31，但是 Linux 的 `signal(7)` 手册页列出了超过 31 个信号名。超出的部分有些只是某些信号的同义名称，为了与其它 UNIX 系统实现源代码兼容性而定义；其它一些则只是定义而并未使用。下面详细地说明各种信号：

SIGABRT

当进程调用 `abort()` 函数（[第 21.2.2 节](#)）时会收到这个信号。默认情况下这个信号会终止进程并产生 `core dump` 文件。这也是 `abort()` 调用原本的用途：产生一个用于调试的 `core dump` 文件。

SIGALRM

调用 `alarm()` 或 `setitimer()` 设置实时定时器之后，在定时器超期时内核就会生成这个信号。实时定时器根据墙上时钟时间（wall clock time，也就是人类概念里的经过时间）来计算定时器。更多细节请参考[第 23.1 节](#)。

SIGBUS

这个信号（总线错误）用来表示几种内存访问错误。一个典型的例子是使用 `mmap()` 创建内存映射文件，如果我们试图访问底层内存映射文件末尾之后的地址，就会产生这个错误，[第 49.4.3 节](#) 对此有详细描述。

SIGCHLD

当进程的某个子进程终止时（不管是调用 `exit()` 自然退出，还是被信号杀死），内核会向父进程发送这个信号。当子进程被信号停止（Stop）、或被信号继续执行（Resume）时，内核也可能会向父进程发送这个信号。我们在[第 26.3 节](#) 详细讨论 `SIGCHLD`。

SIGCLD

等同于 SIGCHLD

SIGCONT

当发送这个信号给一个停止的进程时，会导致该进程恢复执行（稍后被重新调度执行）。当未被停止的进程接收到这个信号时，默认会忽略这个信号。进程可以捕获这个信号，这样可以在恢复运行时执行某些特定的动作。[第 22.2 节](#) 和 [第 34.7 节](#) 会更加详细地讨论这个信号。

SIGEMT

在一般的 UNIX 系统中，这个信号用来表示实现相关的硬件错误。在 Linux 中，这个信号仅用在 Sun SPARC 实现。后缀 EMT 起源于仿真器陷阱(emulator trap)，是 Digital PDP-11 上的汇编助记符(assembler mnemonic)。

SIGFPE

某些类型的算术错误会生成这个信号，例如除以 0。后缀 FPE 是浮点异常(floating-point exception)的缩写，但是这个信号也可能被整数算术错误触发。信号的具体触发条件依赖于硬件体系架构和 CPU 控制寄存器的设置。例如在 x86-32，整数除以 0 总是会触发一个 SIGFPE 信号，但浮点数除以 0 的处理则依赖于是否启用 FE_DIVBYZERO 异常。如果这个异常启用了(使用 feenableexcept())，则浮点数除以 0 会生成 SIGFPE；否则就会为浮点操作数运算出 IEEE 标准结果(代表无穷大的浮点表示)。更多细节请参考 [fenv\(3\)](#) 手册页和<fenv.h>头文件。

SIGHUP

当发生终端断开(挂断)时，会向终端的控制进程发送这个信号。我们在[34.6 节](#) 讨论控制进程的概念，和各种发送 SIGHUP 信号的场景。SIGHUP 的第二种用法与 daemon 相关(如 init, httpd, inetd)，许多 daemon 在接收到 SIGHUP 信号后，会重新初始化自己并重新读取配置文件。系统管理员可以手工向 daemon 发送 SIGHUP 信号(使用显式的 kill 命令或者执行类似的程序或脚本)，从而触发 daemon 执行上述动作。

SIGILL

如果进程试图执行非法（格式不正确）的机器语言指令，会收到这个信号。

SIGINFO

在 Linux 中，这个信号等同于 SIGPWR。在 BSD 系统中，按下 Ctrl-T 会触发 SIGINFO 信号，用来获取前台进程组的状态信息。

SIGINT

当用户输入终端中断字符（通常是 Ctrl-C），终端驱动器会发送这个信号给前台进程组。这个信号的默认动作是终止进程。

SIGIO

使用 fcntl() 系统调用，可以在特定类型的打开文件描述符（如终端和 Socket）发生 I/O 事件（如输入变为可用）时，触发这个信号。[第 63.3 节](#) 更加详细地讨论了这个特性。

SIGIOT

在 Linux 中，这个信号等同于 SIGABRT。在某些其它 UNIX 系统中，这个信号表示实现相关（implementation-defined）的硬件故障。

SIGKILL

这是一个必杀信号，不能被阻塞、忽略、也不能被信号处理器捕获，因此总是会终止一个进程。

SIGLOST

这个信号名存在于 Linux 中，但未被使用。在某些其它 UNIX 系统中，如果 NFS 服务器崩溃后恢复，NFS 客户端重新获得锁失败，就会发送这个信号给本地持有锁的 NFS 客户端进程。（这个特性在 NFS 规范中并没有标准化）。

SIGPIPE

当进程试图写入数据到管道、FIFO、Socket，但却没有相应的读取进程，会触发这个信号。这个信号通常是由于读取进程关闭了 IPC 通道的文件描述符而产生。更多细节请参考[第 44.2 节](#)。

SIGPOLL

这个信号起源于 System V，在 Linux 中等同于 SIGIO。

SIGPROF

调用 `setitimer()` 设置分析定时器后（[第 23.1 节](#)），当定时器超期时，内核会生成这个信号。分析定时器是一种按进程使用的 CPU 时间计时的定时器。和虚拟定时器（参考下面的 `SIGVTALRM`）不同的是，分析定时器会同时计算用户模式和内核模式的 CPU 时间。

SIGPWR

这是一个电源故障信号。在那些拥有不间断电源（UPS）的系统中，我们可以设置一个 `daemon` 进程在电源故障事件发生后监控后备电池等级。如果电池电量快要耗尽（长时间停电后），监控进程会发送 `SIGPWR` 给 `init` 进程，`init` 进程接收到这个信号后，会将这个请求按快速而有序的方式进行处理，从而关闭系统。

SIGQUIT

当用户在键盘上按下退出字符（通常是 `Ctrl-\`），会把这个信号发送给前台进程组。默认这个信号会终止一个进程，并产生一个 `core dump` 文件用于调试。那些处于无限循环或失去响应的程序，按这种方式使用 `SIGQUIT` 是非常有用的。输入 `Ctrl-\`，然后在 `gdb` 调试器中装载 `core dump` 文件，并使用 `backtrace` 命令来获得堆栈跟踪记录，我们可以找出哪段代码正在被执行（[Matloff, 2008] 描述了 `gdb` 的使用）。

SIGSEGV

这个信号非常流行，几乎每个菜鸟程序员都会莫名其妙地遇到它。当程序引用非法内存时会产生这个信号。非法内存引用可能是引用的页不存在（该页处于堆和栈之间某个未映射的区域）、进程试图更新只读内存位置的数据（程序 `text` 段或标记为只读的映射内存）、运行于用户模式下的进程试图访问内核内存（[第 2.1 节](#)）。在 C 语言中最常见的原因是，解引用一个坏地址的指针（未初始化指针），或者函数调用中传递非法参数，就会出现这个信号。这个信号的名字起源于段违规（segmentation violation）。

SIGSTKFLT

`signal(7)` 文档中描述为“协处理器堆栈错误”，Linux 定义了它，但未使用。

SIGSTOP

这是一个确保停止的信号，不能被阻塞、忽略，也不能被信号处理器捕获。因此总是会停止一个进程。

SIGSYS

如果进程发起“坏”的系统调用，就会产生这个信号。意味着进程执行的指令解释为一个系统调用陷阱，但相关联的系统调用号非法（参考[第 3.1 节](#)）。

SIGTERM

这是用来终止一个进程的标准信号，也是 `kill` 和 `killall` 命令默认发送的信号。用户有时候会使用 `kill -KILL` 或 `kill -9` 来显式地发送 `SIGKILL` 信号给一个进程，但通常这样做是错误的。设计良好的应用会为 `SIGTERM` 设置一个信号处理器，用来清理临时文件并释放其它资源，从而优雅地退出应用。通过 `SIGKILL` 信号杀死进程绕过了 `SIGTERM` 信号处理器。因此我们应该首先尝试使用 `SIGTERM` 来终止一个进程，如果进程不响应 `SIGTERM` 信号并拒绝终止，我们再使用杀手锏 `SIGKILL` 来强制杀死进程。

SIGTRAP

这个信号用来实现调试器断点和系统调用跟踪，也就是 `strace(1)`（[附录 A](#)）所执行的任务。更多信息请参考 `ptrace(2)` 手册页。

SIGTSTP

这是作业控制停止信号，当用户在键盘上输入暂停字符（通常是 `Ctrl-Z`）时，用于停止前台进程组。[第 34 章](#) 详细地讨论了进程组（作业）和作业控制，以及程序何时及如何处理这个信号。`SIGTSTP` 的信号名起源于“终端停止”（terminal stop）。

SIGTTIN

在作业控制 `shell` 中，当后台进程组试图从终端执行 `read()` 操作时，终端驱动会发送这个信号给后台进程组。这个信号默认会停止一个进程。

SIGTTOU

这个信号的作用和 `SIGTTIN` 类似，但是作用于后台作业的终端输出。当运行于作业控制 `shell` 时，如果终端的 `TOSTOP`（终端输出停止）选项已经启用（通

过命令 `stty tostop`），终端驱动会在后台进程组试图 `write()` 写入终端时向其发送 `SIGTTOU` 信号（参考[第 34.7.1 节](#)）。这个信号默认会停止一个进程。

SIGUNUSED

如信号名所示，这个信号未被使用。在 Linux 2.4 及之后版本，这个信号在很多体系架构中等同于 `SIGSYS`。换句话说，这个信号数值在这些体系架构中不再被使用，信号名保留只是为了保持向后兼容性。

SIGURG

这个信号发送给进程，用于表示 `socket` 上出现了 `out-of-band`（也称为 `urgent`）数据（[第 61.13.1 节](#)）。

SIGUSR1

这个信号和 `SIGUSR2` 用于程序员自定义的作用，内核不会产生这两个信号。进程可以使用这两个信号来通知其它进程某个事件已经发生，或者用于进程同步。在早期 UNIX 系统中，这是应用可以自由使用的唯二信号。（实际上，进程可以向另一个进程发送任何信号，但是存在潜在的混淆，因为内核也可能向进程产生相同的信号）。现代 UNIX 系统提供一组丰富的实时信号，可供程序员自定义使用（[第 22.8 节](#)）。

SIGUSR2

参考 `SIGUSR1` 的描述。

SIGVTALRM

调用 `setitimer()` 设置虚拟定时器后（[第 23.1 节](#)），当定时器超期时，内核会向进程发送这个信号。虚拟定时器只计算进程在用户模式下的 CPU 时间。

SIGWINCH

在窗口环境中，当终端窗口大小改变时（用户手工调整大小，或程序通过调用 `ioctl()` 调整大小，[第 62.9 节](#) 讨论），这个信号会发送给前台进程组。通过安装信号处理器，如 `vi` 和 `less` 这样的程序可以在窗口大小调整之后，重新绘制自己的输出。

SIGXCPU

当进程超出 CPU 时间资源限制（`RLIMIT_CPU`，[第 36.3 节](#)），会收到这个信号。

SIGXFSZ

如果进程试图（使用 `write()` 或 `truncate()`）增加文件的大小，但却超出了进程的文件大小资源限制（`RLIMIT_FSIZE`, [第 36.3 节](#)），会发送这个信号给该进程。

表 20-1 汇总了 Linux 各种信号的相关信息，注意以下关键信息：

- 信号数值列显示了各种硬件体系架构下信号相关联的数值。除非我们明确标注，信号在所有体系架构下的编号都是一样的。信号数值在不同体系架构的差异标注在小括号中，并且会标明 Sun SPARC 和 SPARC64(S)、HP/Compaq/Digital Alpha(A)、MIPS(M)、HP PA-RISC(P)体系架构。在这个列，`undef` 表示符号在该体系架构下未定义。
- 在 SUSv3 列中显示该信号是否由 SUSv3 标准化。
- 默认（Default）列显示该信号的默认动作：`term` 表示信号会终止进程；`core` 表示进程会产生一个 `core dump` 文件并终止；`ignore` 表示进程会忽略信号；`stop` 表示信号会停止进程；`cont` 则表示信号会继续一个之前停止的进程。

之前列出的某些信号没有显示在表 20-1 中：SIGCLD（等同于 SIGCHLD）、SIGINFO（未使用）、SIGIOT（等同于 SIGABRT）、SIGLOST（未使用）、SIGUNUSED（许多体系架构下等同于 SIGSYS）

表 20-1: Linux 信号

名称	信号数值	描述	SUSv3	默认
SIGABRT	6	中止进程	●	core
SIGALRM	14	实时定时器超时	●	term
SIGBUS	7 (SAMP=10)	内存访问错误	●	core
SIGCHLD	17 (SA=20, MP=18)	子进程终止或停止	●	ignore
SIGCONT	18 (SA=19, M=25, P=26)	停止进程继续	●	cont
SIGEMT	undef (SAMP=7)	硬件故障		term
SIGFPE	8	算术异常	●	core

SIGHUP	1	挂断	●	term
SIGILL	4	非法指令	●	core
SIGINT	2	终端中断	●	term
SIGIO / SIGPOLL	29 (SA=23, MP=22)	I/O 可用	●	term
SIGKILL	9	必杀	●	term
SIGPIPE	13	管道断开	●	term
SIGPROF	27 (M=29, P=21)	分析定时器超时	●	term
SIGPWR	30 (SA=29, MP=19)	电源快要失败		term
SIGQUIT	3	终端退出	●	core
SIGSEGV	11	非法内存引用	●	core
SIGSTKFLT	16 (SAM=undef, P=36)	协处理器堆栈错误		term
SIGSTOP	19 (SA=17, M=23, P=24)	必须停止	●	stop
SIGSYS	31 (SAMP=12)	非法系统调用	●	core
SIGTERM	15	终止进程	●	term
SIGTRAP	5	跟踪/断点陷阱	●	core
SIGTSTP	20 (SA=18, M=24, P=25)	终端停止	●	stop
SIGTTIN	21 (M=26, P=27)	后台进程组读取终端	●	stop
SIGTTOU	22 (M=27, P=28)	后台进程组写入终端	●	stop
SIGURG	23 (SA=16, M=21, P=29)	socket 的 urgent 数据	●	ignore
SIGUSR1	10 (SA=30, MP=16)	用户自定义信号 1	●	term
SIGUSR2	12 (SA=31, MP=17)	用户自定义信号 2	●	term
SIGVTALRM	26 (M=28, P=20)	虚拟定时器超时	●	term
SIGWINCH	28 (M=20, P=23)	终端窗口大小改变		ignore
SIGXCPU	24 (M=30, P=33)	超出 CPU 时间限制	●	core
SIGXFSZ	25 (M=31, P=34)	超出文件大小限制	●	core

注意表 20-1 中某些信号默认行为的以下备注信息：

- 在 Linux 2.2 中，信号 SIGXCPU, SIGXFSZ, SIGSYS 和 SIGBUS 的默认动作是终止进程而不产生 core dump。从内核 2.4 开始，Linux 遵循 SUSv3 的要求，这些信号会产生 core dump 并终止。在其它一些 UNIX 系统中，信号 SIGXCPU 和 SIGXFSZ 按 Linux 2.2 的方式处理。
- 信号 SIGPWR 在其它 UNIX 系统中通常会被忽略，或者根本就不存在。
- 信号 SIGIO 在某些 UNIX 系统（特别是基于 BSD 的 UNIX）默认会被忽略。
- 尽管没有任何标准规定了 SIGEMT，多数 UNIX 系统都有这个信号。但是这个信号在其它 UNIX 系统中通常会生成 core dump 并终止进程。
- 在 SUSv1 标准中，信号 SIGURG 的默认动作被规定为终止进程，这也是某些老的 UNIX 系统的默认动作。SUSv2 采用了当前规定（忽略）。

20.3 改变信号部署：signal()

UNIX 系统提供两个方法来改变信号部署：signal()和 sigaction()。我们在这一节讨论 signal()系统调用，这是设置信号部署的原始 API，提供的接口比 sigaction()更加简单。反过来说，sigaction()能够提供 signal()没有的功能。此外，不同的 UNIX 系统提供的 signal()在行为上有一些差异（[第 22.7 节](#)），这也意味着可移植程序不应该使用 signal()来创建信号处理器。由于 signal()存在这些可移植性问题，sigaction()是建立信号处理器的首选 API。在[第 20.13 节](#)介绍了 sigaction()的用法之后，我们在所有的示例程序中都只使用 sigaction()来建立信号处理器。

尽管 Linux 在手册页的第 2 节对 signal()提供了文档，但实际上在 Linux 中，signal()是 glibc 中实现的库函数，基于 sigaction()系统调用而实现。

```
#include <signal.h>

void ( *signal(int sig, void (*handler)(int)) ) (int);
                                              成功时返回之前的信号部署；出错时返回 SIG_ERR
```

signal()的函数原型需要解释一下，第一个参数 sig 标识我们需要设置部署的信号；第二个参数 handler 是信号处理器函数的地址，也就是信号被递送时需要

调用的函数。这个函数返回类型为 `void`，并且只接受一个整数参数。因此信号处理器的通用格式如下：

```
void handler(int sig)
{
    /* Code for the handler */
}
```

我们在[第 20.4 节](#)讨论信号处理器函数的 `sig` 参数的作用。

`signal()`的返回值是该信号之前的部署，和 `handler` 参数一样，这也是一个指向返回 `void`，只有一个整型参数的函数指针。换句话说，我们可以按下面这样编写代码，为信号设置一个部署之后，稍后可以重置信号部署为之前的部署：

```
void (*oldHandler)(int);

oldHandler = signal(SIGINT, newHandler);
if (oldHandler == SIG_ERR)
    errExit("signal");

/* Do something else here. During this time, if SIGINT is
   delivered, newHandler will be used to handle the signal. */

if (signal(SIGINT, oldHandler) == SIG_ERR)
    errExit("signal");
```

在不改变信号部署的情况下，我们无法使用 `signal()` 来获取信号的当前部署。要实现这个目标，我们必须使用 `sigaction()`。

我们可以对信号处理器函数使用下面类型定义，从而使 `signal()` 的原型更加易于理解：

```
typedef void (*sighandler_t)(int);
```

然后我们就可以按下面这样重新声明 `signal()`：

```
sighandler_t signal(int sig, sighandler_t handler);
```

如果定义了`_GNU_SOURCE` 特性测试宏，`glibc` 会在头文件`<signal.h>` 中暴露非标准的 `sighandler_t` 类型。

除了为 `signal()` 的 `handler` 参数指定一个函数指针，我们也可以使用下面两个值之一：

`SIG_DFL`

重置信号部署为默认动作（表 20-1）。调用 `signal()` 改变信号部署之后，可以使用这个值再次调用 `signal()` 来撤销该部署。

`SIG_IGN`

忽略信号。如果为该进程产生了信号，内核会默默地丢弃信号。而进程根本就不会知道有信号发生。

成功调用 `signal()` 会返回信号之前的部署，可能是之前安装的信号处理器函数地址，也可能是常量 `SIG_DFL` 或 `SIG_IGN`。出错时 `signal()` 返回 `SIG_ERR`。

20.4 信号处理器介绍

信号处理器（也称为信号捕捉器）是一个函数，当特定信号递送到进程时会被调用。我们在本节讨论信号处理器的基础概念，然后在[第 21 章](#)详细讨论信号处理器的细节。

调用信号处理器可能会在任何时候中断主程序流程，内核会代表进程直接调用信号处理器，当处理器返回时，再从之前中断的位置继续程序的执行。执行顺序如图 20-1 所示：

尽管信号处理器理论上可以执行任何操作，但是通常需要设计得尽可能简单，我们在[第 21.1 节](#)会详细论述这一点。

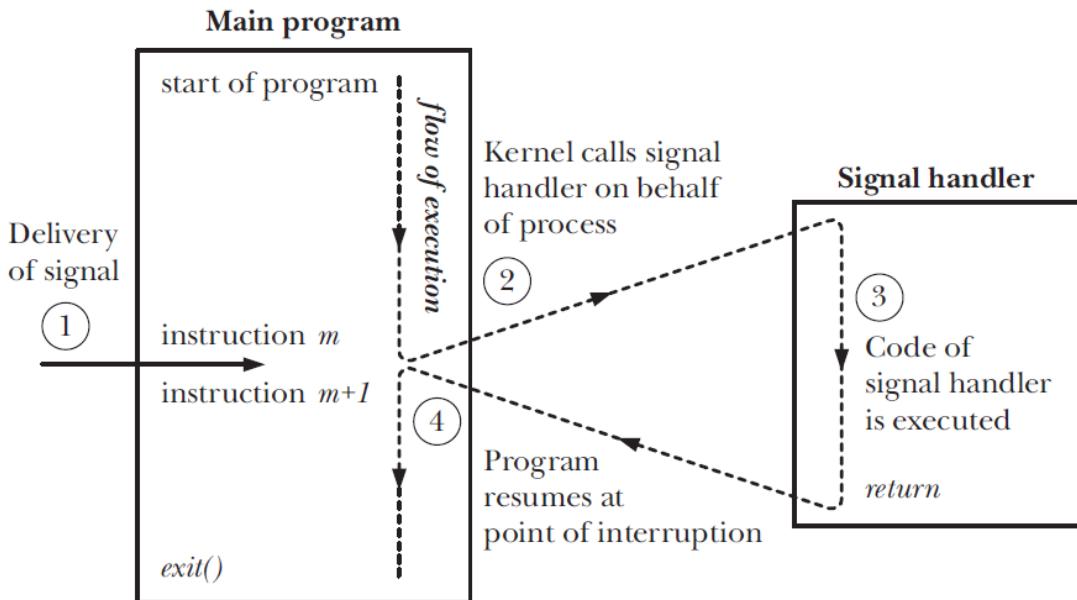


图 20-1: 信号递送和处理器执行

清单 20-1: 为 *SIGINT* 安装一个处理器

```
-----signals/ouch.c
#include <signal.h>
#include "tlpi_hdr.h"

static void
sigHandler(int sig)
{
    printf("Ouch!\n"); /* 不安全! (参考第 21.1.2 节) */
}

int
main(int argc, char *argv[])
{
    int j;

    if (signal(SIGINT, sigHandler) == SIG_ERR)
        errExit("signal");

    for (j = 0; ; j++) {
        printf("%d\n", j);
        sleep(3); /* 缓慢地循环... */
    }
}
-----signals/ouch.c
```

清单 20-1 的示例程序为信号 SIGINT（按下终端中断字符时，通常是 Ctrl-C，终端驱动器会产生这个信号）安装了一个信号处理器。该信号处理器只是简单地打印一条消息然后就返回。

主程序持续地循环，在每次迭代中，程序会增加计数器并打印出来，然后程序睡眠几秒钟（这里我们使用了 `sleep()` 函数，可以使调用方暂停执行指定的秒数，我们在[第 23.4.1 节](#)讨论 `sleep()` 函数）。

运行清单 20-1 的程序，可以得到如下输出：

```
$ ./ouch
0                         主程序循环，显示连续的整数
Type Control-C
Ouch!                     信号处理器执行然后返回
1                         将控制返回给主程序
2
3
Type Control-C again
Ouch!
4
Type Control-\ (the terminal quit character)
Quit (core dumped)
```

当内核调用一个信号处理器时，会向处理器传递导致其被调用的信号数值（也就是清单 20-1 程序中信号处理器的 `sig` 参数）。如果一个信号处理器只捕获一种类型的信号，那么 `sig` 参数就几乎是没用的。但是我们可以同时使用相同信号处理器来捕获不同类型的信号，这时候就需要使用 `sig` 参数来确定是哪个信号导致处理器被调用。

清单 20-2 演示了这种用法，程序使用同一个处理器来捕获 SIGINT 和 SIGQUIT 两个信号（当我们按下终端退出字符，通常是 Ctrl-\，终端驱动会产生 SIGQUIT 信号）。信号处理器通过检查 `sig` 参数来区分这两个信号，并采取不同的动作。在 `main()` 函数中，我们使用 `pause()` 函数（[第 20.14 节](#) 描述）来阻塞进程，直到捕获到某个信号。

下面 shell 会话日志演示了该程序的使用：

```
$ ./intquit
Type Control-C
```

```
Caught SIGINT (1)
Type Control-C again
Caught SIGINT (2)
and again
Caught SIGINT (3)
Type Control-\
Caught SIGQUIT - that's all folks!
```

在清单 20-1 和清单 20-2 中，我们在信号处理器中使用 `printf()` 来打印消息。

根据我们在[第 21.1.2 节](#)中讨论的理由，真实世界的应用通常不应该在信号处理器中调用 `stdio` 库函数。但是在我们的许多示例程序中，还是在信号处理器中调用了 `printf()`，只是用在处理器被调用时简单地打印一些提示信息。

清单 20-2：为两个不同的信号建立相同的处理器

```
-----signals/intquit.c
#include <signal.h>
#include "tlpi_hdr.h"

static void
sigHandler(int sig)
{
    static int count = 0;

    /* 不安全：这个处理器使用了非异步信号安全的函数
       printf(), exit(); 参考第 21.1.2 节 */

    if (sig == SIGINT) {
        count++;
        printf("Caught SIGINT (%d)\n", count);
        return;          /* 从中断的位置继续执行 */
    }

    /* 肯定是 SIGQUIT - 打印一条消息并终止进程 */

    printf("Caught SIGQUIT - that's all folks!\n");

    exit(EXIT_SUCCESS);
}
```

```

int
main(int argc, char *argv[])
{
    /* Establish same handler for SIGINT and SIGQUIT */

    if (signal(SIGINT, sigHandler) == SIG_ERR)
        errExit("signal");
    if (signal(SIGQUIT, sigHandler) == SIG_ERR)
        errExit("signal");

    for (;;)           /* 无限循环，等待信号 */
        pause();        /* 阻塞直到捕获到信号 */
}

-----signals/intquit.c

```

20.5 发送信号：kill()

进程可以使用 `kill()` 系统调用向另一个进程发送信号，`kill()` 系统调用的功能类似于 `kill` 命令（选择术语 `kill`，是因为早期 UNIX 系统多数信号的默认动作都是终止进程）。

```
#include <signal.h>
```

```
int kill(pid_t pid, int sig);
```

成功时返回 0；出错时返回 -1

`pid` 参数标识一个或多个发送信号 `sig` 的目标进程，`pid` 参数有以下四种解释：

- 如果 `pid` 大于 0，信号会发送给进程 ID 为 `pid` 的进程
- 如果 `pid` 等于 0，信号会发送给调用进程所在进程组的所有进程，包括调用进程本身（SUSv3 规定信号应该发送给相同进程组的所有进程，不包括“未指定的系统进程”，同时这条规则还应用于下面的两种情况）。
- 如果 `pid` 小于 -1，信号会发送给进程组 ID 等于 `pid` 绝对值的进程组中的所有进程。发送信号给特定进程组的所有进程对于 `shell` 作业控制（[第 34.7 节](#)）有特殊用途。
- 如果 `pid` 等于 -1，信号会发送给调用进程拥有发送信号权限的所有进程，除了 `init`（进程 ID 为 1）和调用进程本身。如果特权进程发起这样的调

用，则系统中的所有进程都会得到该信号（当然 `init` 和进程本身除外）。

很明显，按这种方式发送的信号就被称为广播信号。（SUSv3 没有要求调用进程不能接收这种信号，但 Linux 遵循 BSD 的语义，将调用进程排除在外）。

如果没有进程匹配 `pid`，则 `kill()` 会失败并设置 `errno` 为 `ESRCH`(`No such process`)。

进程需要适当的权限，才能向另一个进程发送信号。权限规则如下：

- 特权进程 (`CAP_KILL`) 可以向任何进程发送信号。
- `init` 进程 (`ID` 为 1) 是一个特例，它以 `root` 用户和组运行。`init` 进程安装了一些信号的处理器，其它进程只能向 `init` 发送这些已经安装了处理器的信号。这样可以防止系统管理员意外地杀掉 `init` 进程，因为 `init` 进程是系统正常运行的基础。
- 对于非特权进程，只有发送进程的实际用户 `ID` 或有效用户 `ID`，等于接收进程的实际用户 `ID` 或已保存设置用户 `ID`，才能向其发送信号，如图 20-2 所示。这个规则允许用户向自己启动的设置用户 `ID` 程序发送信号，无论目标进程的有效用户 `ID` 当前如何设置。检查权限时不包括目标进程的有效用户 `ID`，主要是互补目的：如果用户 2 以用户 1 的身份运行一个设置用户 `ID` 的程序，则用户 1 将不能向用户 2 的这个进程发送信号，而用户 2 则可以向该进程发送信号。（SUSv3 强制要求图 20-2 中的规则，但 Linux 2.0 之前的版本遵循的规则稍有不同，请参考 `kill(2)` 手册页）。
- `SIGCONT` 信号会被特殊处理，非特权进程可以向相同会话中的任何其它进程发送 `SIGCONT` 信号，无需检查用户 `ID`。这个规则允许作业控制 `shell` 重新开始已经停止的作业（进程组），即使作业中的进程已经改变了自己的用户 `ID`（例如作业进程拥有特权，使用[第 9.7 节](#)讨论的系统调用来改变进程的凭证）。

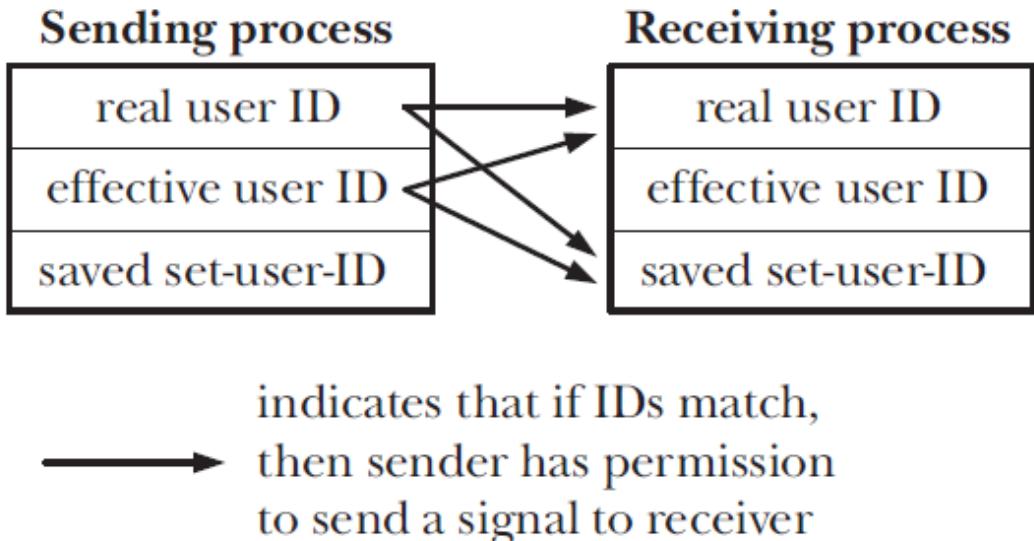


图 20-2：非特权进程发送信号需要的权限

如果一个进程不具备向指定 `pid` 发送信号的权限，则 `kill()` 会失败并设置 `errno` 为 `EPERM`。当 `pid` 指定为一组进程时（`pid` 为负），只要有一个进程能够得到信号，`kill()` 就会成功。

我们在清单 20-3 中演示 `kill()` 的使用。

20.6 检查进程是否存在

`kill()` 系统调用还有另一个用途，如果 `sig` 参数指定为 0（也就是所谓的 `null` 信号），则 `kill()` 不会执行信号发送操作，相反 `kill()` 只是执行错误检查，以确定指定的进程是否能够被发送信号。反过来说，意味着我们可以使用 `null` 信号来测试指定进程 ID 的进程是否存在。如果发送 `null` 信号时 `kill()` 以错误 `ESRCH` 失败，则表示进程不存在。如果以 `EPERM` 错误失败（表示进程存在，但我们没有权限向其发送信号），或者 `kill()` 返回成功（表示我们拥有向进程发送信号的权限），这两种情况都表示指定的进程肯定存在。

检测特定进程 ID 的进程存在，并不能确保该特定进程仍然在运行。因为内核随着进程诞生和死亡会循环使用进程 ID，相同的进程 ID 随着时间的过去，可能会引用到另一个进程。此外，特定进程 ID 虽然存在，但可能是一个僵尸（Zombie）

进程（进程已经死亡，但父进程尚未执行 `wait()` 操作来获取子进程的终止状态，请参考[第 26.2 节](#)）。

除了 `kill()`，我们还可以使用其它一些技术来检查特定进程是否正在运行，包括如下几种：

- `wait()` 系列系统调用：[第 26 章](#) 讨论了这些系统调用，只有检测的目标进程是调用进程的子进程时，才能使用这个技术。
- 信号量（semaphore）和互斥文件锁：如果进程一直持有信号量或者文件锁，那么如果我们能够获得这个信号量或文件锁，就表明目标进程已经终止。我们在[第 47 章](#)和[第 53 章](#)讨论信号量，在[第 55 章](#)讨论文件锁。
- IPC 通道如管道和 FIFO：我们可以设置目标进程，在进程存活期间，一直持有一个打开为写入的文件描述符引用该 IPC 通道。同时检测进程持有一个打开为读取的文件描述符也引用该 IPC 通道，当通道的写入端关闭时（如果读取到 `end-of-file`），则表示目标进程已经终止。检测进程可以从文件描述符一直读取，也可以使用[第 63 章](#)介绍的技术来监控文件描述符。
- `/proc/PID` 接口：如果进程 ID 为 12345 的进程存在，则目录`/proc/12345`也会存在，我们可以使用 `stat()` 来进行检查。

上面所有技术，除了最后一个，都不会受到进程 ID 循环使用的影响。

清单 20-3 演示了 `kill()` 的使用。这个程序接受两个命令行参数，分别是信号数值和进程 ID，使用 `kill()` 向指定进程发送指定的信号。如果命令行参数指定信号 0（null 信号），则程序会报告目标进程是否存在。

清单 20-3：使用 `kill()` 系统调用

```
-----signals/t_kill.c
#include <signal.h>
#include "tlpi_hdr.h"
```

```
int
main(int argc, char *argv[])
{
    int s, sig;

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

    sig = getInt(argv[2], 0, "sig-num");

    s = kill(getLong(argv[1], 0, "pid"), sig);

    if (sig != 0) {
        if (s == -1)
            errExit("kill");
    } else { /* Null 信号：检查进程是否存在 */
        if (s == 0)
            printf("Process exists and we can send it a signal\n");
        else {
            if (errno == EPERM)
                printf("Process exists, but we don't have "
                       "permission to send it a signal\n");
            else if (errno == ESRCH)
                printf("Process does not exist\n");
            else
                errExit("kill");
        }
    }

    exit(EXIT_SUCCESS);
}
-----signals/t_kill.c
```

20.7 发送信号的其它方法：raise() 和 killpg()

有时候进程需要向自己发送信号（我们在[第 34.7.3 节](#)会看到这样的例子），`raise()`函数可以执行这个操作。

```
#include <signal.h>

int raise(int sig);
```

成功时返回 0；出错时返回非 0

在单线程程序中，调用 `raise()` 等同于如下 `kill()` 调用：

```
kill(getpid(), sig);
```

在支持线程的系统中，`raise(sig)` 被实现为：

```
pthread_kill(pthread_self(), sig)
```

我们在[第 33.2.3 节](#)讨论 `pthread_kill()` 函数，现在只需要知道这个实现只会发送信号给调用 `raise()` 的特定线程。相反调用 `kill(getpid(), sig)` 则会发送信号给调用进程，而信号可能会被递送给进程中任何线程。

`raise()` 函数源于 C89 标准，C 标准并没有涵盖操作系统细节，如进程 ID。但 `raise()` 函数并不需要引用进程 ID，所以能够在 C 标准中进行规范。

当进程使用 `raise()`（或 `kill()`）向自己发送信号时，信号会被立即递送（在 `raise()` 函数返回之前）。

注意 `raise()` 出错时返回非 0 值（不一定是 -1）。`raise()` 唯一可能的错误是 `EINVAL`，表示 `sig` 非法。因此当我们使用 `SIGXXXX` 常量时，就不再需要检查 `raise()` 函数的返回值了。

`killpg()` 函数发送一个信号给一个进程组的所有进程。

```
#include <signal.h>

int killpg(pid_t pgrp, int sig);
```

成功时返回 0；出错时返回 -1

调用 `killpg()` 等同于如下 `kill()` 调用：

```
kill(-pgrp, sig);
```

如果 `pgrp` 指定为 0，则信号会发送给相同进程组的所有进程。SUSv3 没有规定这一点，但多数 UNIX 系统（包括 Linux）都是这样处理的。

20.8 显示信号描述信息

每个信号都有一个相关联的可打印描述信息，这些信息定义在数组 `sys_siglist` 中。例如，我们可以引用 `sys_siglist[SIGPIPE]` 来获得信号 `SIGPIPE`（管道断开）的描述信息。但是除了直接使用 `sys_siglist` 数组，更好的方法是使用 `strsignal()` 函数。

```
#define _BSD_SOURCE
#include <signal.h>

extern const char *const sys_siglist[];

#define _GNU_SOURCE
#include <string.h>

char *strsignal(int sig);
```

返回信号描述字符串的指针

`strsignal()` 函数会对 `sig` 参数进行边界检查，然后返回一个可打印信号描述字符串的指针，如果信号数值非法，就返回一个错误字符串的指针。（某些 UNIX 系统在 `sig` 非法时 `strsignal()` 会返回 `NULL`）。

除了边界检查，`strsignal()` 比直接使用 `sys_siglist` 数组还有另外一个优点。`strsignal()` 函数是区域敏感的（locale-sensitive，[第 10.4 节](#)），因此信号描述会显示为本地语言。

清单 20-4 显示了 `strsignal()` 函数的使用。

`psignal()` 函数显示（在标准错误）`msg` 字符串，然后是一个冒号，接下来是 `sig` 相应的信号描述信息。和 `strsignal()` 一样，`psignal()` 也是区域敏感的。

```
#include <signal.h>

void psignal(int sig, const char *msg);
```

虽然 SUSv3 没有规定 `psignal()`、`strsignal()`、和 `sys_siglist`，但在许多 UNIX 系统上都可用。（SUSv4 增加了 `psignal()` 和 `strsignal()` 的规定）。

20.9 信号集

许多信号相关的系统调用需要一组不同信号的表示方法。例如 `sigaction()` 和 `sigprocmask()` 允许程序指定进程能够阻塞的一组信号，而 `sigpending()` 则返回进程当前未决的一组信号（我们马上就会讨论这些系统调用）。

多个信号使用信号集数据结构来描述，由系统数据类型 `sigset_t` 来表示。SUSv3 规定了一组函数来操作信号集，我们现在就来讨论这些函数。

在 Linux 和多数 UNIX 系统中，`sigset_t` 数据类型是位掩码。不过 SUSv3 没有强制要求这样做，SUSv3 只要求 `sigset_t` 类型可以直接赋值。实际上信号集可以方便地实现为许多不同类型的数据类型，但由于 SUSv3 的要求，要么实现为标量类型（如整数），要么实现为 C 语言结构体（如结构体包含一个整数数组）。

`sigemptyset()` 函数初始化并清空一个信号集。`sigfillset()` 函数则初始化信号集并使其包含所有信号（包括所有实时信号）。

```
#include <signal.h>

int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
```

成功时都返回 0；出错时都返回 -1

必须使用 `sigemptyset()` 或 `sigfillset()` 之一来初始化一个信号集。因为 C 语言不会自动初始化自动变量，而且静态变量初始化为 0 也不能移植地表示一个空信号集，因为信号集有可能实现为结构体而不是位掩码。（同样的理由，使用 `memset(3)` 对一个信号集置 0 来清空信号集的内容，也是不正确的）。

初始化信号集之后，可以使用 `sigaddset()` 和 `sigdelset()` 来添加和删除单个信号。

```
#include <signal.h>

int sigaddset(sigset_t *set, int sig);
int sigdelset(sigset_t *set, int sig);
```

成功时都返回 0；出错时都返回 -1

这两个函数的 `sig` 参数分别表示要添加和删除的信号数值。

`sigismember()`函数用来检测某个信号是否属于指定的信号集。

```
#include <signal.h>

int sigismember(const sigset_t *set, int sig);
如果 sig 是信号集 set 的成员则返回 1; 否则返回 0
```

如果 `sig` 是信号集 `set` 的成员, `sigismember()`函数返回 1 (`true`); 否则返回 0 (`false`)。

GNU C 库实现了三个非标准的函数, 提供了上述标准信号集函数的补充功能。

```
#define __GNU_SOURCE
#include <signal.h>

int sigandset(sigset_t *dest, sigset_t *left, sigset_t *right);
int sigorset(sigset_t *dest, sigset_t *left, sigset_t *right);
成功时都返回 0; 出错时都返回 -1

int sigisemptyset(const sigset_t *set);
如果 set 为空则返回 1; 否则返回 0
```

这三个函数的功能如下:

- `sigandset()`把 `left` 和 `right` 两个信号集的交集放到 `dest` 信号集中
- `sigorset()`把 `left` 和 `right` 两个信号集的并集放到 `dest` 信号集中
- `sigisemptyset()`检查信号集是否为空

示例程序

使用本节讨论的函数, 我们编写了清单 20-4 的几个函数, 后续章节将会使用到这些函数。第一个 `printSigset()`可以显示指定信号集中的所有信号成员, 这个函数使用了 `NSIG` 常量, 定义在`<signal.h>`头文件中, 它的值等于最大的信号数值加一。我们使用 `NSIG` 作为循环的上限, 依次测试所有信号是否是指定信号集的成员。

尽管 SUSv3 没有规定 NSIG，但多数 UNIX 系统都定义了它。不过可能需要使用实现相关的编译器选项来暴露它。例如在 Linux 中，我们必须定义以下某个特性测试宏：`_BSD_SOURCE`、`_SVID_SOURCE`、`_GNU_SOURCE`

`printSigMask()`和 `printPendingSigs()` 函数使用 `printSigset()` 来分别显示进程信号掩码和进程当前未决信号集。同时 `printSigMask()` 和 `printPendingSigs()` 函数分别使用了 `sigprocmask()` 和 `sigpending()` 系统调用来实现自己的功能。我们在[第 20.10 节](#)和[第 20.11 节](#)分别讨论 `sigprocmask()` 和 `sigpending()` 系统调用。

清单 20-4：显示信号集的函数

```
-----signals/signal_functions.c
#define _GNU_SOURCE
#include <string.h>
#include <signal.h>
#include "signal_functions.h"      /* Declares functions defined here */
#include "tlpi_hdr.h"

/* 注意：下面的所有函数都使用了非异步信号安全（第 21.1.2 节）的 fprintf()，因此
这些函数同样也不是异步信号安全的（不要在信号处理器函数使用它们） */

void          /* 打印信号集中的信号列表 */
printSigset(FILE *of, const char *prefix, const sigset_t *sigset)
{
    int sig, cnt;

    cnt = 0;
    for (sig = 1; sig < NSIG; sig++) {
        if (sigismember(sigset, sig)) {
            cnt++;
            fprintf(of, "%s%d (%s)\n", prefix, sig, strsignal(sig));
        }
    }

    if (cnt == 0)
        fprintf(of, "%s<empty signal set>\n", prefix);
}

int          /* 打印当前进程阻塞信号的掩码 */
printSigMask(FILE *of, const char *msg)
```

```
{  
    sigset_t currMask;  
  
    if (msg != NULL)  
        fprintf(of, "%s", msg);  
  
    if (sigprocmask(SIG_BLOCK, NULL, &currMask) == -1)  
        return -1;  
  
    printSigset(of, "\t\t", &currMask);  
  
    return 0;  
}  
  
int /* 打印当前进程的未决信号集 */  
printPendingSigs(FILE *of, const char *msg)  
{  
    sigset_t pendingSigs;  
  
    if (msg != NULL)  
        fprintf(of, "%s", msg);  
  
    if (sigpending(&pendingSigs) == -1)  
        return -1;  
  
    printSigset(of, "\t\t", &pendingSigs);  
  
    return 0;  
}  
-----signals/signal_functions.c
```

20.10 信号掩码（阻塞信号递送）

内核为每个进程维护了一个信号掩码——进程当前阻塞递送的一组信号。如果一个阻塞信号要发送给该进程，会延迟信号递送直到进程从信号掩码中移除该信号。（在[第 33.2.1 节](#)，我们会看到信号掩码实际上是每个线程的属性，多线程进程的每个线程都可以独立地检查和修改自己的信号集，使用的函数是 `pthread_sigmask()`）。

信号可以按以下方式添加到信号掩码中：

- 当信号处理器被调用时，导致处理器被调用的信号可以被自动添加到信号掩码。使用 `sigaction()` 来建立信号处理器时，我们可以通过参数来控制是否要将信号添加到信号掩码。
- 当使用 `sigaction()` 建立信号处理器时，还可以指定额外的信号集，当信号处理器被调用时，也一并添加到信号掩码并阻塞。
- 可以使用 `sigprocmask()` 系统调用显式地添加信号到信号掩码，也可以从信号掩码中移除信号。

我们将在[第 20.13 节](#)讨论 `sigaction()` 时讲解前面两种情况，现在先介绍 `sigprocmask()` 的使用。

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
成功时返回 0; 出错时返回 -1
```

我们可以使用 `sigprocmask()` 来改变进程的信号掩码、获得当前掩码，或者同时执行这两个操作。`how` 参数确定了 `sigprocmask()` 如何对信号掩码进行修改：

SIG_BLOCK

信号集 `set` 中指定的信号将添加到进程的信号掩码。换句话说，新的信号掩码将设置为当前掩码与 `set` 信号集的并集。

SIG_UNBLOCK

信号集 `set` 中指定的信号将从进程的信号掩码中移除。如果要解除阻塞的信号并不存在于当前的信号掩码中，`sigprocmask()` 并不会返回错误。

SIG_SETMASK

信号集 `set` 将直接赋值给进程的信号掩码。

三种情况下，如果 `oldset` 参数不是 `NULL`，它指向的 `sigset_t` 缓冲区都会用来返回之前的信号掩码。

如果我们想获取但不改变进程的信号掩码，可以指定 `set` 参数为 `NULL`，这种情况下 `how` 参数将被忽略。

要临时阻止一个信号的递送，我们可以使用清单 20-5 中的一系列调用，首先阻塞信号，然后通过重置信号掩码为之前的状态，来解除信号的阻塞。

清单 20-5：临时阻塞一个信号的递送

```
-----  
sigset_t blockSet, prevMask;  
  
/* Initialize a signal set to contain SIGINT */  
  
sigemptyset(&blockSet);  
sigaddset(&blockSet, SIGINT);  
  
/* Block SIGINT, save previous signal mask */  
  
if (sigprocmask(SIG_BLOCK, &blockSet, &prevMask) == -1)  
    errExit("sigprocmask1");  
  
/* ... Code that should not be interrupted by SIGINT ... */  
  
/* Restore previous signal mask, unblocking SIGINT */  
  
if (sigprocmask(SIG_SETMASK, &prevMask, NULL) == -1)  
    errExit("sigprocmask2");  
-----
```

SUSv3 规定如果任意未决信号由 `sigprocmask()` 解除阻塞，则在 `sigprocmask()` 调用返回之前至少会有一个这样的未决信号递送给进程。换句话说，如果我们解除阻塞一个未决信号，它会被立即递送给进程。

试图阻塞 `SIGKILL` 和 `SIGSTOP` 信号会被默默地忽略，如果我们试图阻塞这两个信号，`sigprocmask()` 既不会响应请求，也不会报告出错。这意味着我们可以使用下面代码来阻塞所有信号，除了 `SIGKILL` 和 `SIGSTOP`：

```
sigfillset(&blockSet);  
if (sigprocmask(SIG_BLOCK, &blockSet, NULL) == -1)  
    errExit("sigprocmask");
```

20.11 未决信号

如果一个进程接收到当前正被阻塞的信号，这个信号会被添加到进程的未决信号集。稍后信号被解除阻塞时，才会被递送至进程。我们可以使用 `sigpending()` 来确定进程哪些信号是未决的。

```
#include <signal.h>
```

```
int sigpending(sigset_t *set);
```

成功时返回 0；出错时返回 -1

`sigpending()` 系统调用返回调用进程当前的未决信号集，并将结果保存在 `set` 指向的 `sigset_t` 结构体。我们可以使用[第 20.9 节](#)讨论的 `sigismember()` 函数来检查该信号集。

如果我们修改了未决信号的部署，当信号稍后被解除阻塞时，会根据新的信号部署来调用信号处理器。虽然不是很常用，但这个技术也有一个应用的场景，我们可以把未决信号的部署设置为 `SIG_IGN` 或 `SIG_DFL`（如果信号的默认动作就是忽略的话），从而阻止未决信号的递送。这样设置的结果就是，信号会从进程的未决信号集中移除，因此不会被递送。

20.12 信号不会排队

未决信号集只是一个掩码，表示是否有信号已经发生，但不能表示信号发生了多少次。换句话说，如果相同的信号在阻塞期间发生了多次，会在未决信号集中记录下来，但最后只会递送一次。（标准信号和实时信号的区别之一就是实时信号会进行排队，[第 22.8 节](#)将会讨论）。

清单 20-6 和清单 20-7 中的两个程序，可以用来观察信号不会排队这一事实。

清单 20-6 的程序最多可接受四个命令行参数，如下所示：

```
$ ./sig_sender PID num-sigs sig-num [sig-num-2]
```

第一个参数是程序要发送信号的进程 ID，第二个参数指定向目标进程发送信号的次数。第三个参数指定发送信号的信号数值。如果提供了第四个参数（可选），

则程序在最后会发送一次该信号。在下面的 shell 示例会话中，我们使用了第四个参数来发送 SIGINT 信号给目标进程。这个参数的作用很快你就会明白。

清单 20-6：发送多个信号

```
-----signals/sig_sender.c
#include <signal.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int numSigs, sig, j;
    pid_t pid;

    if (argc < 4 || strcmp(argv[1], "--help") == 0)
        usageErr("%s pid num-sigs sig-num [sig-num-2]\n", argv[0]);

    pid = getLong(argv[1], 0, "PID");
    numSigs = getInt(argv[2], GN_GT_0, "num-sigs");
    sig = getInt(argv[3], 0, "sig-num");

    /* Send signals to receiver */

    printf("%s: sending signal %d to process %ld %d times\n",
           argv[0], sig, (long) pid, numSigs);

    for (j = 0; j < numSigs; j++)
        if (kill(pid, sig) == -1)
            errExit("kill");

    /* If a fourth command-line argument was specified, send that signal */

    if (argc > 4)
        if (kill(pid, getInt(argv[4], 0, "sig-num-2")) == -1)
            errExit("kill");

    printf("%s: exiting\n", argv[0]);
    exit(EXIT_SUCCESS);
}
-----signals/sig_sender.c
```

清单 20-7 中的程序用来捕获和报告清单 20-6 程序发送的信号。这个程序执行以下步骤：

- 程序设置一个信号处理器捕获所有信号②。(捕获 SIGKILL 和 SIGSTOP 是不可能的，我们忽略为这些信号建立处理器时产生的错误)。对于多数信号类型来说，处理器①只是使用数组对其进行简单的计数。如果接收到 SIGINT 信号，处理器会设置一个标志 (gotSigint) 从而导致程序退出主循环 (下面描述的 while 循环)。(我们会在[第 21.1.3 节](#)解释 gotSigint 变量的 volatile 修饰符和 sig_atomic_t 数据类型)
- 如果指定了命令行参数，程序会根据这个参数阻塞所有信号指定的秒数，然后在解除信号阻塞之前，打印出未决信号集③。这样在程序执行下面步骤之前，我们可以向它发送各种信号。
- 程序执行 while 循环，持续地消耗 CPU 时间直到 gotSigint 标志被设置④。
([第 20.14 节](#)和[第 22.9 节](#)讨论了 pause()和 sigsuspend()的使用，可以更高效地等待信号到达)。
- 退出 while 循环之后，程序会打印所有接收到的信号计数⑤。

我们首先使用这两个程序来说明阻塞信号只会被递送一次，无论信号实际产生了多少次。通过指定接收信号方睡眠时间，在睡眠时间完成之前发送所有信号来验证这一点。

```
$ ./sig_receiver 15 &                                接收方阻塞信号 15 秒
[1] 5368
./sig_receiver: PID is 5368
./sig_receiver: sleeping for 15 seconds

$ ./sig_sender 5368 1000000 10 2      发送 SIGUSR1 信号，最后 SIGINT
./sig_sender: sending signal 10 to process 5368 1000000 times
./sig_sender: exiting
./sig_receiver: pending signals are:
                2 (Interrupt)
                10 (User defined signal 1)
./sig_receiver: signal 10 caught 1 time
[1]+ Done ./sig_receiver 15
```

发送程序的命令行参数指定了 SIGUSR1 和 SIGINT 信号，在 Linux/x86 中信号数值分别为 10 和 2。

从上面的输出我们可以看到，即使发送了一百万个信号，最后只有一个递送到了接收进程。

就算进程不阻塞信号，接收到的信号也可能会少于发送的信号次数。当信号发送的速度非常快，在接收进程被内核重新调度下一次执行之前，如果有多个信号到达，就会出现这种情况。因为多个相同的信号在进程的未决信号集中只会记录一次。如果我们不带命令行参数执行清单 20-7 的程序（不阻塞信号），那么可以看到如下输出：

```
$ ./sig_receiver &
[1] 5393
./sig_receiver: PID is 5393
$ ./sig_sender 5393 1000000 10 2
./sig_sender: sending signal 10 to process 5393 1000000 times
./sig_sender: exiting
./sig_receiver: signal 10 caught 52 times
[1]+ Done ./sig_receiver
```

发送了一百万个信号，接收进程仅仅捕获到 52 个（捕获信号的数量在不同的内核调度算法下会有不同）。原因在于每次发送进程调度执行时，它会发送许多信号给接收进程。但是当接收进程被调度运行时，只会有一个信号被标记为未决并进行递送。

清单 20-7：捕获信号并计数

```
-----signals/sig_receiver.c
#define _GNU_SOURCE
#include <signal.h>
#include "signal_functions.h" /* Declaration of printSigset() */
#include "tlpi_hdr.h"

static int sigCnt[NSIG]; /* Counts deliveries of each signal */
static volatile sig_atomic_t gotSigint = 0;
/* Set nonzero if SIGINT is delivered */
```

```
static void
handler(int sig)
{①
    if (sig == SIGINT)
        gotSigint = 1;
    else
        sigCnt[sig]++;
}

int
main(int argc, char *argv[])
{
    int n, numSecs;
    sigset_t pendingMask, blockingMask, emptyMask;

    printf("%s: PID is %ld\n", argv[0], (long) getpid());

    ② for (n = 1; n < NSIG; n++)      /* Same handler for all signals */
        (void) signal(n, handler);  /* Ignore errors */

    /* If a sleep time was specified, temporarily block all signals,
       sleep (while another process sends us signals), and then
       display the mask of pending signals and unblock all signals */

    ③ if (argc > 1) {
        numSecs = getInt(argv[1], GN_GT_0, NULL);

        sigfillset(&blockingMask);
        if (sigprocmask(SIG_SETMASK, &blockingMask, NULL) == -1)
            errExit("sigprocmask");

        printf("%s: sleeping for %d seconds\n", argv[0], numSecs);
        sleep(numSecs);

        if (sigpending(&pendingMask) == -1)
            errExit("sigpending");

        printf("%s: pending signals are: \n", argv[0]);
        printSigset(stdout, "\t\t", &pendingMask);

        sigemptyset(&emptyMask); /* Unblock all signals */
        if (sigprocmask(SIG_SETMASK, &emptyMask, NULL) == -1)
            errExit("sigprocmask");
    }
}
```

```

        errExit("sigprocmask");
    }

④ while (!gotSigint)           /* Loop until SIGINT caught */
    continue;

⑤ for (n = 1; n < NSIG; n++) /* Display number of signals received */
    if (sigCnt[n] != 0)
        printf("%s: signal %d caught %d time%s\n", argv[0], n,
               sigCnt[n], (sigCnt[n] == 1) ? "" : "s");

    exit(EXIT_SUCCESS);
}
-----signals/sig_receiver.c

```

20.13 改变信号部署：sigaction()

sigaction() 系统调用是 signal() 的高级版本，用来设置信号的部署。尽管 sigaction() 的使用比 signal() 稍微复杂，但却提供了更加丰富的功能和灵活性。特别是 sigaction() 允许我们在不改变信号部署的情况下获得信号部署，并且可以设置各种属性，来精确地控制信号处理器被调用时如何处理信号。此外，我们在[第 22.7 节](#)还将阐述，sigaction() 在建立信号处理器时比 signal() 更加可移植。

```
#include <signal.h>

int sigaction(int sig, const struct sigaction *act, struct sigaction
             *oldact);

```

成功时返回 0；出错时返回 -1

sig 参数标识我们要获取或修改部署的信号，这个参数可以是任意信号，除了 SIGKILL 和 SIGSTOP。

act 参数是指定信号新部署的结构体指针。如果我们只想获得信号现有的部署，则可以指定 act 为 NULL。**oldact** 参数也是指向信号部署结构体的指针，用来返回信号之前的部署。如果我们不需要获得信号之前的部署，可以指定 oldact 为 NULL。act 和 oldact 指向的结构体类型定义如下：

```

struct sigaction {
    void (*sa_handler)(int);      /* 信号处理器地址 */
    sigset_t sa_mask;            /* 处理器调用时阻塞的信号 */
    int sa_flags;                /* 控制处理器调用的标志 */
    void (*sa_restorer)(void);   /* 应用程序不能使用 */
};

```

`sigaction` 结构体实际上比这里显示的更加复杂，我们在[第 21.4 节](#)讨论更多细节。

`sa_handler` 域对应于 `signal()` 中的 `handler` 参数，指定了信号处理器的地址，或者是常量 `SIG_IGN` 或 `SIG_DFL`。`sa_mask` 和 `sa_flags` 域我们马上会讨论，只有当 `sa_handler` 域是一个信号处理器时（不是 `SIG_IGN` 或 `SIG_DFL` 两个常量时）才会使用它们。剩下的 `sa_restorer` 域不是给应用程序使用的（`SUSv3` 没有对其进行规定）。

`sa_restorer` 域只在内部使用，用来确保信号处理器完成时，能够发起一次特殊用途的 `sigreturn()` 系统调用，`sigreturn()` 会还原进程的执行上下文，这样进程才能够从信号处理器中断的位置继续进程的执行。这种用法可以在 `glibc` 源文件 `sysdeps/unix/sysv/linux/i386/sigaction.c` 中找到示例。

`sa_mask` 域定义了一组调用信号处理器时需要阻塞的信号。当信号处理器被调用时，在执行信号处理器之前，`sa_mask` 域中的任何不在当前进程信号掩码中的信号，都会被自动添加到进程的信号掩码。这些信号会保留在进程的信号掩码中，直到信号处理器返回，此时这些信号又会自动从进程信号掩码中移除。`sa_mask` 域允许我们指定一组信号，使其不能中断当前信号处理器的执行。此外，导致信号处理器被调用的信号本身，会自动添加到进程的信号掩码。这意味着信号处理器不会被第二个相同的信号递归地中断。由于阻塞信号不会排队，如果在执行信号处理器时，这些阻塞信号重复产生，稍后只会被递送一次。

`sa_flags` 域是一个位掩码，指定各种控制信号如何处理的选项。下面这些位可以“或”(|)起来使用：

SA_NOCLDSTOP

如果 `sig` 是 `SIGCHLD`，当子进程由于接收到信号而停止或继续执行时，不产生这个信号，参考[第 26.3.2 节](#)。

SA_NOCLDWAIT (Linux 2.6 起)

如果 `sig` 是 `SIGCHLD`, 当子进程终止时不转变为 `Zombie` 进程。更多信息请参考[第 26.3.3 节](#)。

SA_NODEFER

当信号被捕获时，在信号处理器执行时不要自动将其添加到进程的信号掩码。名字 `SA_NOMASK` 历史上等同于 `SA_NODEFER`, 但由于 SUSv3 标准规定了后者，所以我们应该优先使用 `SA_NODEFER`。

SA_ONSTACK

使用 `sigaltstack()` 安装的备用堆栈来调用信号处理器，参考[请 21.3 节](#)。

SA_RESETHAND

当信号被捕获时，在调用信号处理器之前重置信号的部署为默认（也就是 `SIG_DFL`）。（默认情况下，安装的信号处理器会一直保留，直到再次调用 `sigaction()` 建立新的信号部署）。名字 `SA_ONESHOT` 也是历史上等同于 `SA_RESETHAND`, 但后者由 SUSv3 标准化，因此优先使用。

SA_RESTART

自动重新开始被信号处理器中断的系统调用，参考[第 21.5 节](#)。

SA_SIGINFO

使用信号相关的额外参数（更多信息）来调用信号处理器，我们会在[第 21.4 节](#)讨论这个标志。

上面所有选项都由 SUSv3 标准规定。

清单 21-1 提供了 `sigaction()` 的使用示例。

20.14 等待信号：`pause()`

调用 `pause()` 可以暂停进程的执行，直到某个信号处理器中断 `pause()` 调用（或者某个未处理的信号终止进程）。

```
#include <unistd.h>

int pause(void);
```

总是返回-1 并设置 errno 为 EINTR

当信号处理器处理一个信号时，`pause()`会被中断，并总是返回-1，设置 `errno` 为 `EINTR`（我们会在[第 21.5 节](#)更加详细地讨论 `EINTR`）。

清单 20-2 演示了 `pause()` 的使用。

在[第 22.9 节](#)、[第 22.10 节](#)、[第 22.11 节](#)，我们会介绍程序暂停执行并等待信号的其它方法。

20.15 小结

信号是通知发生了某种事件的机制，内核和进程都可能会向进程发送各种信号，进程也可以向自己发送信号。系统定义了一组标准信号类型，每种信号都拥有唯一的数值和用途。

典型的信号递送是异步的，意味着进程可能会在任何时候被信号中断。在某些情况下（如硬件产生的信号），信号也可能被同步递送，意味着信号递送是可预测的，并且可以在程序执行期间特定时间点重新产生。

信号的默认动作可能是忽略、终止进程（生成或不生成 `core dump` 文件）、停止进程执行、继续执行已停止的进程。不同信号类型的默认动作是不一样的。此外程序可以使用 `signal()` 或 `sigaction()` 显式地忽略一个信号，或者建立程序员自定义的信号处理器。由于可移植性等原因，我们最好使用 `sigaction()` 来建立信号处理器。

进程（拥有适当的权限）可以使用 `kill()` 向其它进程发送信号。发送 `null` 信号（0）可以检测特定进程 ID 是否正被系统使用。

每个进程都拥有一个信号掩码，也就是当前阻塞递送的信号集。我们可以使用 `sigprocmask()` 向信号掩码中添加或删除信号。

如果一个信号在被阻塞时接收到，会保持未决直到该信号被解除阻塞。标准信号不会被排队，也就是说一个信号只会被标记为未决一次，稍后也只会被递送

一次。进程可以使用 `sigpending()` 系统调用来获取当前未决的信号集。

`sigaction()` 系统调用在设置信号部署时，能够提供比 `signal()` 更多的控制选项和灵活性。首先我们可以指定一组额外的信号，在信号处理器被调用时进行阻塞。此外还可以使用许多标志来控制信号处理器被调用时的各种动作。例如有些标志可以选择使用老式的不可靠信号语义（不阻塞导致信号处理器调用的当前信号，以及调用信号处理器之前重置信号的部署为默认动作）。

使用 `pause()` 可以暂停进程的执行，直到某个信号到达。

更多信息

[Bovet & Cesati, 2005] 和 [Maxwell, 1999] 提供了 Linux 信号实现的背景。[Goodheart & Cox, 1994] 详细描述了 System V 发布版 4 的信号实现。GNU C 库手册包含丰富的信号描述信息 (<http://www.gnu.org/>)。

20.16 习题

- 20-1. 如第 20.3 节所述，`sigaction()` 在建立信号处理器时比 `signal()` 更加可移植。
使用 `sigaction()` 替换 `signal()`，重新实现清单 20-7 的程序 (`sig_receiver.c`)。
- 20-2. 编写一个程序，验证当未决信号的部署改变为 `SIG_IGN` 时，程序将不会捕获到该信号。
- 20-3. 编写一个程序，验证 `sigaction()` 建立信号处理器时，`SA_RESETHAND` 和 `SA_NODEFER` 标志的作用。
- 20-4. 使用 `sigaction()` 实现 `siginterrupt()`。

第 21 章 信号：信号处理器

本章继续上一章对信号的讨论，集中讲解信号处理器，并扩充了[第 20.4 节](#)关于信号处理器的讨论。主题如下：

- 怎样设计一个信号处理器，涉及可重入和异步信号安全函数的讨论；
- 信号处理器除了正常返回，还有哪些其它方式可以退出，特别是非局部跳转技术的应用；
- 在备用堆栈中处理信号；
- 使用 `sigaction()` 的 `SA_SIGINFO` 标志，允许一个信号处理器获得信号的额外信息；
- 信号处理器怎样中断一个阻塞的系统调用，以及怎样重新开始系统调用。

21.1 设计信号处理器

通常我们应该编写尽可能简单的信号处理器。很重要的一个原因是这样可以降低出现竞争条件的危险。下面是两种常见的信号处理器设计：

- 信号处理器设置一个全局标志并退出。主程序定期检查这个标志，如果标志已经设置，则采取适当的动作。（如果主程序需要监控一个或多个文件描述符来查看 I/O 是否可用，从而不能执行全局标志的定期检查，则信号处理器也可以向一个专用管道写入一个字节，该管道的读取端也同时纳入主程序监控的文件描述符列表，我们在[第 63.5.2 节](#)会使用这个技术编写一个示例程序）。
- 信号处理器执行某些清理工作，然后要么终止进程，要么使用非局部跳转（第 21.2.1 节）来展开堆栈，并返回到预先确定的主程序位置。

接下来我们会详细讨论这些理念，以及设计信号处理器的其它重要概念。

21.1.1 信号不会排队（回顾）

在[第 20.10 节](#)，我们提到在执行一个信号的处理器时，会阻塞该信号的再次递送（除非我们为 `sigaction()` 指定 `SA_NODEFER` 标志）。如果在执行信号处理器时信号再次产生，会将信号标记为未决，稍后处理器返回才会再次递送信号。我们还提到信号不会排队，如果信号处理器执行时产生了多个相同的信号，则只会标记并递送一次。

信号在这种情况下会“消失”，对我们设计信号处理器有潜在的影响。首先，我们不能准确地计算信号产生的次数；此外信号对应的事件还可能有多种，信号处理器需要能够处理这种情况，我们在[第 26.3.1 节](#)会看到 `SIGCHLD` 信号的这种例子及使用。

21.1.2 可重入和异步信号安全的函数

不是所有系统调用和库函数都能够在信号处理器中安全地调用。要理解这一点我们需要解释两个概念：可重入函数和异步信号安全函数。

可重入和不可重入函数

要解释可重入函数，我们首先需要区分单线程和多线程程序。传统的 UNIX 程序只有单个线程：`CPU` 从单个逻辑执行流程中处理指令。在多线程程序中，相同进程会有多个独立的并发逻辑执行流程。

在[第 29 章](#)，我们会学习如何显式地创建包含多个执行线程的程序。但是多线程执行同样也会影响到信号处理器。因为一个信号处理器可能会在任意时间任意位置异步地中断程序的执行，主程序和信号处理器在效果上形成两个独立（虽然不是并发）的执行线程。

如果一个函数可以被相同进程的多个线程同时安全地执行，就称为可重入函数。在这里的上下文中，“安全”意味着函数能够得到预期的结果，不会受到其它线程执行的影响。

SUSv3 定义可重入函数为“函数的效果，在被两个或多个线程同时调用时，虽然实际的执行是交错的，但能够确保和各个线程按任意顺序独立运行函数一样”。

如果一个函数更新全局或静态数据结构，就是不可重入的。（函数必须只使用局部变量才能确保是可重入的）。如果函数的两个调用（如两个线程执行）同时试图更新相同的全局变量或数据结构，则更新操作就会互相影响并产生错误的结果。举例来说，假设一个线程更新一个链表数据结构，在添加一个新的节点时，另一个线程也更新同一个链表。由于添加一个节点到链表要求更新多个指针，如果另一个线程中断了这些步骤，而且也更新了相同指针，就会导致混乱。

实际上这种可能性在标准 C 库中比比皆是。例如我们在[第 7.1.3 节](#)讨论的 `malloc()` 和 `free()`，它们在堆中维护了一个自由内存块的链表。如果主程序中的 `malloc()` 调用被信号处理器中断，而信号处理器也调用了 `malloc()`，则这个链表就可能会被损坏。正因为如此，`malloc()` 系列函数，以及使用了 `malloc()` 的所有函数，都是不可重入的。

还有一些库函数因为返回了静态分配的内存，因此也是不可重入的。例如 `crypt()`, `getpwnam()`, `gethostname()`, `getservbyname()` 等等。如果一个信号处理器也使用了这些函数，就会覆盖主程序中之前调用相同函数的信息（反之亦然）。

如果函数使用静态数据结构作为内部记录，也可能是不可重入的。最明显的例子是 `stdio` 库的函数（`printf()`, `scanf()` 等等），它们会为缓冲 I/O 更新内部数据结构。因此在信号处理器中使用 `printf()` 时，如果信号处理器在主程序执行 `printf()` 时中断了主程序，我们有时候会看到奇怪的输出（甚至是程序崩溃或数据损坏）。

即使我们完全不使用不可重入函数，可重入的问题仍然存在。如果一个信号处理器更新了程序员自定义的全局数据结构，而主程序也更新相同的全局数据结构，则我们就说信号处理器相对主程序是不可重入的。

如果一个函数不可重入，则它的手册页通常会显式或隐式地标注这一点。请特别关注使用或返回信息是静态分配变量的那些函数。

示例程序

清单 21-1 演示了 `crypt()` 函数（[第 8.5 节](#)）的不可重入。程序接受两个命令行参数字符串，程序主要执行以下步骤：

1. 调用 `crypt()` 加密第一个参数字符串，并使用 `strdup()` 复制该字符串到另一个缓冲区中。
2. 为 `SIGINT` 信号（通常输入 `Ctrl-C`）建立一个处理器，信号处理器调用 `crypt()` 加密第二个命令行参数字符串。
3. 进入无限 `for` 循环，使用 `crypt()` 不停地加密第一个命令行参数字符串，并检查返回字符串是否等于第 1 步保存的字符串。

如果没有信号产生，第 3 步的两个字符串永远是匹配的。但是如果 `SIGINT` 信号到达，假如主程序的 `for` 循环正在执行 `crypt()` 调用，但尚未进行两个字符串比较，此时信号处理器将主程序中断。你可以看到主程序会报告两个字符串不匹配。我们运行该程序，可以看到如下输出：

```
$ ./non_reentrant abc def
Repeatedly type Control-C to generate SIGINT
Mismatch on call 109871 (mismatch=1 handled=1)
Mismatch on call 128061 (mismatch=2 handled=2)
Many lines of output removed
Mismatch on call 727935 (mismatch=149 handled=156)
Mismatch on call 729547 (mismatch=150 handled=157)
Type Control-\ to generate SIGQUIT
Quit (core dumped)
```

通过比较 `mismatch` 和 `handled` 两个值，我们发现当信号处理器被调用时，在主程序调用 `crypt()` 和字符串比较之间，多数情况下都会覆盖静态分配的缓冲区。

清单 21-1：在 `main()` 和信号处理器中同时调用不可重入函数

```
-----signals/nonreentrant.c
#define _XOPEN_SOURCE 600
#include <unistd.h>
#include <signal.h>
```

```
#include <string.h>
#include "tlpi_hdr.h"

static char *str2;                  /* Set from argv[2] */
static int handled = 0;             /* Counts number of calls to handler */

static void
handler(int sig)
{
    crypt(str2, "xx");
    handled++;
}

int
main(int argc, char *argv[])
{
    char *cr1;
    int callNum, mismatch;
    struct sigaction sa;

    if (argc != 3)
        usageErr("%s str1 str2\n", argv[0]);

    str2 = argv[2];                  /* Make argv[2] available to handler */
    cr1 = strdup(crypt(argv[1], "xx")); /* Copy statically allocated string
                                         to another buffer */

    if (cr1 == NULL)
        errExit("strdup");

    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sa.sa_handler = handler;
    if (sigaction(SIGINT, &sa, NULL) == -1)
        errExit("sigaction");

    /* Repeatedly call crypt() using argv[1]. If interrupted by a
       signal handler, then the static storage returned by crypt()
       will be overwritten by the results of encrypting argv[2], and
       strcmp() will detect a mismatch with the value in 'cr1'. */

    for (callNum = 1, mismatch = 0; ; callNum++) {
```

```

    if (strcmp(crypt(argv[1], "xx"), cr1) != 0) {
        mismatch++;
        printf("Mismatch on call %d (mismatch=%d handled=%d)\n",
               callNum, mismatch, handled);
    }
}
-----signals/nonreentrant.c

```

标准异步信号安全函数

异步信号安全的函数是指函数确保能够被信号处理器安全地调用。异步信号安全的函数要么是可重入的，要么就不会被信号处理器中断。

表 21-1 列出了各个标准要求实现异步信号安全的函数，后面没有标注 v2 和 v3 的函数由 POSIX.1-1990 规定为异步信号安全；SUSv2 添加的异步信号安全函数标注为 v2；SUSv3 添加的异步信号安全函数则标注为 v3；某些 UNIX 系统可能还会有其它一些异步信号安全的函数，但所有遵循标准的 UNIX 系统都必须确保表 21-1 所列函数是异步信号安全的（如果系统提供了这些函数的话，如 Linux 就并未提供表中的所有函数）。

SUSv4 则对表 21-1 做出了如下修改：

- 删除了下面函数：fpathconf()、pathconf()、和 sysconf()；
- 添加了下面函数：execl()，execv()，faccessat()，fchmodat()，fchownat()，fexecve()，fstatat()，futimens()，linkat()，mkdirat()，mkfifoat()，mknod()，mknodat()，openat()，readlinkat()，renameat()，symlinkat()，unlinkat()，utimensat()，utimes()。

表 21-1：POSIX.1-1990、SUSv2、SUSv3 要求的异步信号安全函数

_Exit() (v3)	getpid()	sigaddset()
_exit()	getppid()	sigdelset()
abort() (v3)	getsockname() (v3)	sigemptyset()
accept() (v3)	getsockopt() (v3)	sigfillset()
access()	getuid()	sigismember()
aio_error() (v2)	kill()	signal() (v2)
aio_return() (v2)	link()	sigpause() (v2)

aio_suspend() (v2)	listen() (v3)	sigpending()
alarm()	lseek()	sigprocmask()
bind() (v3)	lstat() (v3)	sigqueue() (v2)
cfgetispeed()	mkdir()	sigset() (v2)
cfgetospeed()	mkfifo()	sigsuspend()
cfsetispeed()	open()	sleep()
cfsetospeed()	pathconf()	socket() (v3)
chdir()	pause()	socketmark() (v3)
chmod()	pipe()	socketpair() (v3)
chown()	poll() (v3)	stat()
clock_gettime() (v2)	posix_trace_event()	symlink() (v3)
close()	(v3)	sysconf()
connect() (v3)	pselect() (v3)	tcdrain()
creat()	raise() (v2)	tcflow()
dup()	read()	tcflush()
dup2()	readlink() (v3)	tcgetattr()
execle()	recv() (v3)	tcgetpgrp()
execve()	recvfrom() (v3)	tcsendbreak()
fchmod() (v3)	recvmsg() (v3)	tcsetattr()
fchown() (v3)	rename()	tcsetpgrp()
fcntl()	rmdir()	time()
fdasyncc() (v2)	select() (v3)	timer_getoverrun() (v2)
fork()	sem_post() (v2)	timer_gettime() (v2)
fpathconf() (v2)	send() (v3)	timer_settime() (v2)
fstat()	sendmsg() (v3)	times()
fsync() (v2)	sendto() (v3)	umask()
ftruncate() (v3)	setgid()	uname()
getegid()	setpgid()	unlink()
geteuid()	setsid()	utime()
getgid()	setsockopt() (v3)	wait()
getgroups()	setuid()	waitpid()
getpeername() (v3)	shutdown() (v3)	write()
getpgrp()	sigaction()	

SUSv3 指出所有未列在表 21-1 的函数都认为是对信号不安全的，但也同时指出只有信号处理器中断一个不安全函数的执行，并且处理器本身也调用该不安全函数，这些函数才是不安全的。换句话说，编写信号处理器时我们有两个选择：

- 确保信号处理器本身是可重入的，只调用异步信号安全的函数；
- 在主程序调用不安全函数时，或者主程序使用信号处理器也会更新的全

局数据结构时，阻塞信号的递送。

第二种方法的问题是，复杂的程序很难确保信号处理器不会中断主程序调用某个不安全的函数。因此通常我们会强制要求信号处理器只调用异步信号安全的函数。

如果我们设置同一个信号处理器函数来处理多个不同的信号，或者使用 `sigaction()` 的 `SA_NODEFER` 标志，则信号处理器可以还会中断自己。因此，如果信号处理器更新全局或静态变量，则即使主程序不使用这些全局或静态变量，信号处理器也是不可重入的。

在信号处理器中使用 `errno`

由于我们可能会更新 `errno`，信号处理器使用表 21-1 中的函数也可能导致信号处理器不可重入，因为可能会覆盖主程序调用函数时设置的 `errno` 值。解决的办法是在进入信号处理器时保存 `errno` 的值，然后使用表 21-1 中的异步信号安全函数，最后在退出信号处理器之前再还原之前保存的 `errno` 值。如下面示例代码所示：

```
void
handler(int sig)
{
    int savedErrno;
    savedErrno = errno;

    /* Now we can execute a function that might modify errno */

    errno = savedErrno;
}
```

本书示例程序使用不安全函数

尽管 `printf()` 不是异步信号安全的函数，但我们在本书许多程序的信号处理器中都使用了它。这样做是因为 `printf()` 提供了简便且精练的方式来演示信号处理器被调用的情况，以及显示信号处理器中相关变量的内容。类似地原因，我们偶尔也会在信号处理器中使用其它一些不安全函数，例如其它 `stdio` 函数和 `strsignal()`。

真实世界的应用程序应该避免在信号处理器中调用非异步信号安全的函数。为了清晰地表现这一点，本书示例程序的信号处理器如果使用了不安全函数，都会使用注释标明此用法是不安全的：

```
printf("Some message\n"); /* UNSAFE */
```

21.1.3 全局变量和 `sig_atomic_t` 数据类型

虽然存在可重入的问题，但主程序和信号处理器之间共享全局变量还是非常有用的。只要主程序意识到信号处理器可能会在任何时候修改全局变量的值，并且采取相应的措施进行处理，就可以安全地使用全局变量。例如一个常用的设计是信号处理器只负责设置全局标志，而主程序则定期地检查这个标志，当标志被设置（也就是信号被递送）时采取适当地动作响应（并清除该标志）。信号处理器按这种方式访问全局变量时，我们应该把全局变量声明为 `volatile`([第 6.8 节](#))，防止编译器执行优化而导致变量被存储在寄存器中。

读取和写入全局变量可能需要多个机器语言指令，而信号处理器可能会在这些指令序列中间中断主程序（我们称这些变量的访问是非原子的）。由于这个原因，C 语言标准和 SUSv3 规定了 `sig_atomic_t` 整数类型，可以确保原子地读取和写入。因此主程序和信号处理器共享的全局变量都应该声明为如下形式：

```
volatile sig_atomic_t flag;
```

我们在清单 22-5 的程序中演示了 `sig_atomic_t` 数据类型的使用。

注意 C 语言的递增（`++`）和递减（`--`）操作并不能得到 `sig_atomic_t` 提供的原子保护。在某些硬件体系架构中，这些操作并不是原子的（参考[第 30.1 节](#)）。唯一能够确保安全的是在信号处理器中设置一个 `sig_atomic_t` 变量，然后在主程序中访问它（或者反过来使用）。

C99 和 SUSv3 规定系统应该在`<stdint.h>`中定义两个常量：`SIG_ATOMIC_MIN` 和 `SIG_ATOMIC_MAX`，定义了 `sig_atomic_t` 整数类型能够赋值的范围。标准要求如果 `sig_atomic_t` 是带符号类型，这个范围至少达到`-127 ~ 127`；如果 `sig_atomic_t` 是无符号类型，这个范围至少达到`0 ~ 255`。在 Linux 中，这两个常量分别等于带符号 32 位整数的负数和正数最大限制值。

21.2 终止信号处理器的其它方法

目前为止我们编写的所有信号处理器，都是通过 `return` 返回控制给主程序。但是有时候仅仅从信号处理器中返回是不够的，或者直接返回不是非常有用（我们在[第 22.4 节](#)讨论硬件生成的信号时，会看到这样的例子）。

终止信号处理器还有其它一些方式：

- 使用 `_exit()` 来终止进程，在此之前信号处理器可以执行一些清理工作。注意我们不能使用 `exit()` 来终止信号处理器，因为 `exit()` 不是表 21-1 中列出的异步信号安全函数。`exit()` 会在调用 `_exit()` 之前冲洗（flush）`stdio` 缓冲区，因此是不安全的，我们在[第 25.1 节](#)有相关描述。
- 使用 `kill()` 或 `raise()` 发送一个信号来杀死进程（那些默认动作是终止进程的信号都可以）。
- 在信号处理器中执行一个非局部跳转
- 使用 `abort()` 函数来终止进程并生成 `core dump` 文件。

接下来我们详细讨论最后两个技术的使用。

21.2.1 从信号处理器中执行非局部跳转

[第 6.8 节](#) 讨论了使用 `setjmp()` 和 `longjmp()` 执行非局部跳转，从一个函数跳转到某个调用函数。我们可以在信号处理器中使用这个技术。使用这个方法我们可以在硬件异常（如内存访问错误）导致信号递送之后进行恢复，也允许我们捕获一个信号并返回到程序的特定位置。例如，在接收到 `SIGINT` 信号后（通常是输入 `Ctrl-C`），`shell` 会执行一个非局部跳转，返回控制给 `shell` 的主输入循环（因此用户可以继续输入命令）。

但是使用 `longjmp()` 函数退出信号处理器也有一个问题，我们前面提到过，当进入信号处理器时，内核自动添加调用信号以及 `sa_mask` 域指定的所有信号，到进程的信号掩码，当信号处理器正常退出时从进程信号掩码中移除这些信号。

如果我们使用 `longjmp()` 退出信号处理器，会发生什么呢？答案依赖于特定的

UNIX 系统家族。在 System V 中，`longjmp()`不会还原信号掩码，因此离开信号处理器时不会解除信号阻塞。Linux 遵循了 System V 的行为（通常这不是我们想要的行为，因为这样会使导致信号处理器被调用的信号一直阻塞）。在基于 BSD 的 UNIX 系统中，`setjmp()`会保存 `env` 参数中的信号掩码，然后 `longjmp()`会还原已保存的信号掩码。（基于 BSD 的 UNIX 系统还提供了另外两个函数：`_setjmp()`和 `_longjmp()`，采用了 System V 的语义）。换句话说，我们不能可移植地使用 `longjmp()` 来退出一个信号处理器。

如果我们编译程序时定义了 `_BSD_SOURCE` 特性测试宏，则（glibc）的 `setjmp()` 会遵循 BSD 的语义。

由于上述两个主流 UNIX 变种的这个差异，POSIX.1-1990 并没有规定 `setjmp()` 和 `longjmp()` 如何处理信号掩码。相反，标准定义了一对新的函数：`sigsetjmp()` 和 `siglongjmp()`，可以在执行非局部跳转时显式地控制信号掩码。

```
#include <setjmp.h>

int sigsetjmp(sigjmp_buf env, int savesigs);
    初始调用时返回 0; 通过 siglongjmp() 返回时返回非 0
void siglongjmp(sigjmp_buf env, int val);
```

`sigsetjmp()` 和 `siglongjmp()` 函数的操作与 `setjmp()` 和 `longjmp()` 类似。唯一的区别在于 `env` 参数的类型（`sigjmp_buf` 而不是 `jmp_buf`），以及 `sigsetjmp()` 的额外参数 `savesigs`。如果 `savesigs` 非 0，`sigsetjmp()` 调用会保存进程的当前信号掩码到 `env` 参数，并在 `siglongjmp()` 调用时使用相同的 `env` 参数进行还原。如果 `savesigs` 为 0，则不保存和还原进程的信号掩码。

`longjmp()` 和 `siglongjmp()` 都不是表 21-1 中的异步信号安全函数，因为在执行非局部跳转之后，调用任何非异步信号安全的函数，都面临在信号处理器中调用该函数一样的风险。此外如果主程序正在更新一个数据结构，而被信号处理器中断，然后信号处理器又通过非局部跳转退出，则对数据结构不完全的更新将导致其处于不一致的状态。要避免这种情况，我们可以在更新敏感数据时，使用 `sigprocmask()` 临时阻塞信号。

示例程序

清单 21-2 演示了两种非局部跳转对信号掩码处理的区别。这个程序为 SIGINT 信号建立了一个信号处理器，程序允许通过 `setjmp() + longjmp()` 或 `sigsetjmp() + siglongjmp()` 来退出信号处理器，具体使用哪个由编译时是否定义 `USE_SIGSETJMP` 宏来决定。程序会在进入信号处理器，和非局部跳转到主程序时，分别显示进程的当前信号掩码。

当我们使用 `longjmp()` 来退出信号处理器时，下面是程序运行的会话日志：

```
$ make -s sigmask_longjmp      默认编译使用 setjmp()
$ ./sigmask_longjmp
Signal mask at startup:
<empty signal set>
Calling setjmp()
按下 Ctrl-C 来产生 SIGINT
Received signal 2 (Interrupt), signal mask is:
    2 (Interrupt)
After jump from handler, signal mask is:
    2 (Interrupt)
(此时再按 Ctrl-C 已经没有作用，因为 SIGINT 已经被阻塞)
按下 Ctrl-\ 来杀死程序
Quit
```

从程序输出我们可以看到，`longjmp()` 从信号处理器跳转之后，信号掩码保持最初进入信号处理器时的设置。

在上面 shell 会话中，我们使用了本书源码包中提供的 `makefile` 来编译程序。`-s` 选项告诉 `make` 不要显示正在执行的命令。我们使用这个选项避免 shell 会话日志过于凌乱。（[Mecklenburg, 2005] 提供了 `GNU make` 工具的详细描述）。

当我们编译相同程序，让其使用 `siglongjmp()` 来退出信号处理器，我们可以看到如下输出：

```
$ make -s sigmask_siglongjmp    使用 cc -DUSE_SIGSETJMP 编译
$ ./sigmask_siglongjmp x
Signal mask at startup:
<empty signal set>
Calling sigsetjmp()
按下 Ctrl-C
Received signal 2 (Interrupt), signal mask is:
    2 (Interrupt)
```

```
After jump from handler, signal mask is:  
<empty signal set>
```

此时 SIGINT 没有被阻塞，因为 siglongjmp()已经还原了进程的信号掩码。下面我们可以再次按下 Ctrl-C，再次触发信号处理器被调用：

```
按下 Ctrl-C  
Received signal 2 (Interrupt), signal mask is:  
    2 (Interrupt)  
After jump from handler, signal mask is:  
<empty signal set>  
按下 Ctrl-\来杀死程序  
Quit
```

从上面输出我们可以看到，siglongjmp()还原进程信号掩码为 sigsetjmp()调用时的状态（空信号集）。

清单 21-2 还演示了信号处理器执行非局部跳转的另一个有用的技术。由于信号可能会在任何时候产生，有可能会在 sigsetjmp()（或 setjmp()）设置跳转目标之前就发生信号。为了防止出现这种可能性（会导致信号处理器使用未初始化的 env 缓冲区执行非局部跳转），我们采用了一个守卫变量 canJump，以显示 env 缓冲区是否已经初始化完成。如果 canJump 为 false，信号处理器直接返回而不执行非局部跳转。另一种方式是重新调整程序代码，把调用 sigsetjmp()（或 setjmp()）的代码放在建立信号处理器之前。不过在复杂的程序中，我们可能很难确保这两个步骤谁先谁后，而使用守卫变量则更加简单。

注意编写清单 21-2 的程序时，使用#ifndef 是遵循标准的情况下最简单的实现方式。特别注意我们不能替换#ifndef 为以下运行时检查：

```
if (useSiglongjmp)  
    s = sigsetjmp(senv, 1);  
else  
    s = setjmp(env);  
if (s == 0)  
    ...
```

这样做是不可能的，因为 SUSv3 不允许 setjmp()和 sigsetjmp()用在赋值语句中（参考[第 6.8 节](#)）。

清单 21-2：信号处理器执行非局部跳转

```
-----signals/sigmask_longjmp.c
#define _GNU_SOURCE /* Get strsignal() declaration from <string.h> */
#include <string.h>
#include <setjmp.h>
#include <signal.h>
#include "signal_functions.h" /* Declaration of printSigMask() */
#include "tlpi_hdr.h"

static volatile sig_atomic_t canJump = 0;
                           /* Set to 1 once "env" buffer has been
                                initialized by [sig]setjmp() */

#ifndef USE_SIGSETJMP
static sigjmp_buf senv;
#else
static jmp_buf env;
#endif

static void
handler(int sig)
{
    /* UNSAFE: This handler uses non-async-signal-safe functions
       (printf(), strsignal(), printSigMask(); see Section 21.1.2) */

    printf("Received signal %d (%s), signal mask is:\n", sig,
           strsignal(sig));
    printSigMask(stdout, NULL);

    if (!canJump) {
        printf("'env' buffer not yet set, doing a simple return\n");
        return;
    }

#ifndef USE_SIGSETJMP
    siglongjmp(senv, 1);
#else
    longjmp(env, 1);
#endif
}
```

```
int
main(int argc, char *argv[])
{
    struct sigaction sa;

    printSigMask(stdout, "Signal mask at startup:\n");

    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sa.sa_handler = handler;
    if (sigaction(SIGINT, &sa, NULL) == -1)
        errExit("sigaction");

#ifndef USE_SIGSETJMP
    printf("Calling sigsetjmp()\n");
    if (sigsetjmp(senv, 1) == 0)
#else
    printf("Calling setjmp()\n");
    if (setjmp(env) == 0)
#endif
        canJump = 1; /* Executed after [sig]setjmp() */

    else /* Executed after [sig]longjmp() */
        printSigMask(stdout, "After jump from handler, signal mask is:\n ");

    for (;;) /* Wait for signals until killed */
        pause();
}

-----signals/sigmask_longjmp.c
```

21.2.2 异常地终止进程：abort()

abort()函数终止调用进程，并生成一个 core dump 文件。

```
#include <stdlib.h>

void abort(void);
```

abort()函数通过发起 SIGABRT 信号来终止调用进程。SIGABRT 信号的默认动作是产生 core dump 文件并终止进程。core dump 文件可以用在调试器中，检查

程序在调用 `abort()` 时的各种状态。

SUSv3 要求 `abort()` 废除 `SIGABRT` 信号的阻塞或忽略效果。此外 SUSv3 还规定 `abort()` 必须终止进程，即使进程使用信号处理器捕获信号并且信号处理器不执行返回操作。后面一句话需要我们花点时间来解释。[第 21.2 节](#) 讨论的信号处理器终止方法中，和 `abort()` 相关的是非局部跳转退出信号处理器。如果信号处理器确实执行了非局部跳转，则 `abort()` 将是无效的；否则 `abort()` 总是会终止进程。在多数 UNIX 系统中，`abort()` 终止进程有如下保证：如果发送一次 `SIGABRT` 信号还没有终止掉进程（如信号处理器捕获了信号并返回，然后继续从 `abort()` 调用的位置继续执行），`abort()` 会重置 `SIGABRT` 信号的部署为 `SIG_DFL`，并再次发送 `SIGABRT` 信号，这样就可以确保终止进程。

如果 `abort()` 成功地终止了进程，它还会冲洗并关闭 `stdio` 流。

清单 3-3 提供的错误处理函数演示了 `abort()` 的使用方法。

21.3 在备用堆栈中处理信号：`sigaltstack()`

通常当一个信号处理器被调用时，内核会在进程堆栈中为其创建一个栈帧。但是如果进程试图扩展堆栈并超过了堆栈的最大限制，就会出现问题。例如堆栈增长得太大，已经遇到了映射内存区域（[第 48.5 节](#)）或向上增长堆，或者堆栈达到了 `RLIMIT_STACK` 资源限制（[第 36.3 节](#)）。

当一个进程试图增大堆栈却超过了最大可能的大小时，内核会为进程生成一个 `SIGSEGV` 信号。但是由于堆栈空间已经耗尽，内核不能为 `SIGSEGV` 信号处理器创建栈帧。因此信号处理器不会被调用到，进程也将终止（`SIGSEGV` 信号的默认动作就是终止进程）。

如果我们想确保 `SIGSEGV` 信号在这种情况下也能被处理，可以执行如下操作：

1. 分配一块内存，称为备用信号堆栈，用作信号处理器的栈帧。
2. 使用 `sigaltstack()` 系统调用通知内核，告知这个备用信号堆栈的存在。
3. 当建立信号处理器时，指定 `SA_ONSTACK` 标志，告诉内核该信号处理器的栈帧将在备用堆栈中创建。

`sigaltstack()` 系统调用创建一个备用信号堆栈，并返回之前已经建立的备用信号堆栈的相关信息。

```
#include <signal.h>

int sigaltstack(const stack_t *sigstack, stack_t *old_sigstack);
```

成功时返回 0；出错时返回 -1

`sigstack` 参数指向一个结构体，指定了新的备用信号堆栈的位置和属性。`old_sigstack` 参数也指向这样一个结构体，用来返回之前建立的备用信号堆栈的信息（如果有的话）。这两个参数都可以指定为 `NULL`，例如指定 `sigstack` 为 `NULL`，则我们可以在不改变现有备用信号堆栈的情况下，获得它的相关信息；指定 `old_sigstack` 为 `NULL`，则表示无需返回之前建立的备用信号堆栈的信息。`stack_t` 结构体定义如下：

```
typedef struct {
    void *ss_sp;          /* 备用堆栈的起始地址 */
    int ss_flags;         /* 标志：SS_ONSTACK, SS_DISABLE */
    size_t ss_size;        /* 备用堆栈的大小 */
} stack_t;
```

`ss_sp` 和 `ss_size` 域指定备用信号堆栈的位置和大小，当内核使用备用信号堆栈时，内核会根据硬件体系架构，为 `ss_sp` 地址自动选择合适的内存对齐边界。

通常备用信号堆栈是在堆上静态或动态分配的内存区域。`SUSv3` 规定常量 `SIGSTKSZ` 作为备用信号堆栈的典型大小，`MINSIGSTKSZ` 常量则是调用一个信号处理器需要的最小大小。在 `Linux/x86-32` 中，这两个常量分别为 8192 和 2048。

内核不会调整备用信号堆栈的大小。如果堆栈溢出了我们为其分配的大小，会导致不可预料的混乱（如覆盖堆栈之外的变量）。通常这并不存在问题，因为我们通常只是使用备用信号堆栈来处理标准信号溢出的特殊情况，备用信号堆栈上只会分配一到两个栈帧。`SIGSEGV` 信号处理器要么执行某些清理工作并终止进程，要么使用非局部跳转展开（`unwind`）标准堆栈。

`ss_flags` 域的两个值如下：

SS_ONSTACK

如果获取当前已经建立的备用信号堆栈（`old_sigstack`）时设置了这个标志，表示进程当前正在该备用信号堆栈中执行。进程正在备用信号堆栈中执行时试图创建新的备用信号堆栈，`sigaltstack()`会以 `EPERM` 错误失败。

SS_DISABLE

如果在 `old_sigstack` 中返回，这个标志表示当前并没有建立备用信号堆栈。在 `sigstack` 中指定这个标志，会禁止当前已经建立的备用信号堆栈。

清单 21-3 演示了建立和使用备用信号堆栈。在建立备用信号堆栈，并在该堆栈上建立 `SIGSEGV` 信号处理器之后，程序调用函数进入无限递归，从而引起堆栈溢出并产生 `SIGSEGV` 信号。当我们运行这个程序时，可以看到如下输出：

```
$ ulimit -s unlimited
$ ./t_sigaltstack
Top of standard stack is near 0xbffff6b8
Alternate stack is at 0x804a948-0x804cff
Call 1 - top of stack near 0xbff0b3ac
Call 2 - top of stack near 0xbfe1714c
Many intervening lines of output removed
Call 2144 - top of stack near 0x4034120c
Call 2145 - top of stack near 0x4024cfac
Caught signal 11 (Segmentation fault)
Top of handler stack near 0x804c860
```

在这个 shell 会话日志中，`ulimit` 命令用来移除 shell 可能已经设置的所有 `RLIMIT_STACK` 资源限制。我们在[第 36.3 节](#)讨论这个资源限制。

清单 21-3：使用 `sigaltstack()`

```
-----signals/t_sigaltstack.c
#define _GNU_SOURCE /* Get strsignal() declaration from <string.h> */
#include <string.h>
#include <signal.h>
#include "tlpi_hdr.h"

static void
sigsegvHandler(int sig)
```

```
{  
    int x;  
  
    /* UNSAFE: This handler uses non-async-signal-safe functions  
     * (printf(), strsignal(), fflush(); see Section 21.1.2) */  
  
    printf("Caught signal %d (%s)\n", sig, strsignal(sig));  
    printf("Top of handler stack near %10p\n", (void *) &x);  
    fflush(NULL);  
  
    _exit(EXIT_FAILURE);      /* Can't return after SIGSEGV */  
}  
  
static void          /* A recursive function that overflows the stack */  
overflowStack(int callNum)  
{  
    char a[100000];           /* Make this stack frame large */  
  
    printf("Call %4d - top of stack near %10p\n", callNum, &a[0]);  
    overflowStack(callNum+1);  
}  
  
int  
main(int argc, char *argv[])  
{  
    stack_t sigstack;  
    struct sigaction sa;  
    int j;  
  
    printf("Top of standard stack is near %10p\n", (void *) &j);  
  
    /* Allocate alternate stack and inform kernel of its existence */  
  
    sigstack.ss_sp = malloc(SIGSTKSZ);  
    if (sigstack.ss_sp == NULL)  
        errExit("malloc");  
  
    sigstack.ss_size = SIGSTKSZ;  
    sigstack.ss_flags = 0;  
    if (sigaltstack(&sigstack, NULL) == -1)  
        errExit("sigaltstack");  
    printf("Alternate stack is at    %10p-%p\n",
```

```

    sigstack.ss_sp, (char *) sbrk(0) - 1);

    sa.sa_handler = sigsegvHandler; /* Establish handler for SIGSEGV */
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_ONSTACK;      /* Handler uses alternate stack */
    if (sigaction(SIGSEGV, &sa, NULL) == -1)
        errExit("sigaction");

    overflowStack(1);
}

-----signals/t_sigaltstack.c

```

21.4 SA_SIGINFO 标志

使用 `sigaction()` 建立信号处理器时设置 `SA_SIGINFO` 标志，允许信号处理器获得关于被递送信号的额外信息。要获得这些额外信息，我们必须按如下原型声明信号处理器：

```
void handler(int sig, siginfo_t *siginfo, void *ucontext);
```

第一个 `sig` 参数是信号数值，和标准信号处理器的 `sig` 参数一样。第二个 `siginfo` 参数是一个结构体，用于提供信号的额外信息，下面马上会讨论这个结构体。最后一个 `ucontext` 参数也在下面讨论。

由于上面信号处理器与标准信号处理器函数的原型并不相同，而 C 语言类型规则不允许我们使用 `sigaction` 结构体的 `sa_handler` 域来指定两种处理器。因此我们必须使用另一个 `sa_sigaction` 域。换句话说，`sigaction` 结构体的定义实际上比我们在[第 20.13 节](#)所示要更加复杂，完整的 `sigaction` 结构体定义如下：

```

struct sigaction {
    union {
        void (*sa_handler)(int);
        void (*sa_sigaction)(int, siginfo_t *, void *);
    } __sigaction_handler;
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void);
};

```

```

/* 以下定义使得 union 域看上去和简单的域一样 */

#define sa_handler __sigaction_handler.sa_handler
#define sa_sigaction __sigaction_handler.sa_sigaction

```

`sigaction` 结构体使用 `union` 来组合 `sa_sigaction` 和 `sa_handler` 两个域（多数其它 UNIX 系统也使用了类似的 `union`）。由于每次调用 `sigaction()` 只需要使用其中的一个域，因此这里可以使用 `union`。（但是如果我们天真地独立设置 `sa_handler` 和 `sa_sigaction` 两个域，例如重用一个 `sigaction` 结构体在多个 `sigaction()` 调用，来建立不同信号的处理器，就会出现奇怪的 bug）。

下面是使用 `SA_SIGINFO` 来建立信号处理器的示例：

```

struct sigaction act;

sigemptyset(&act.sa_mask);
act.sa_sigaction = handler;
act.sa_flags = SA_SIGINFO;

if (sigaction(SIGINT, &act, NULL) == -1)
    errExit("sigaction");

```

完整的 `SA_SIGINFO` 标志示例程序，请看清单 22-3 和清单 23-5。

`siginfo_t` 结构体

`siginfo_t` 结构体是使用 `SA_SIGINFO` 标志建立信号处理器时的第二个参数，定义如下：

```

typedef struct {
    int si_signo;                      /* 信号数值 */
    int si_code;                       /* 信号代码 */
    int si_trapno;                     /* 硬件生成信号的陷阱数值
                                         (多数体系架构未使用) */
    union sigval si_value;             /* sigqueue() 的相关数据 */
    pid_t si_pid;                      /* 发送进程的 ID */
    uid_t si_uid;                      /* 发送进程的实际用户 ID */
    int si_errno;                      /* 错误数值 (通常未使用) */
    void *si_addr;                     /* 产生信号的地址 (仅硬件生成的信号) */
}

```

```

int si_overrun;           /* 超限次数
                           (Linux 2.6, POSIX 定时器) */
int si_timerid;          /* (内核内部) 定时器 ID
                           (Linux 2.6, POSIX 定时器) */
long si_band;             /* Band event (SIGPOLL/SIGIO) */
int si_fd;                /* 文件描述符 (SIGPOLL/SIGIO) */
int si_status;            /* 退出状态或信号 (SIGCHLD) */
clock_t si_utime;         /* 用户 CPU 时间 (SIGCHLD) */
clock_t si_stime;          /* 系统 CPU 时间 (SIGCHLD) */
} siginfo_t;

```

`_POSIX_C_SOURCE` 特性测试宏的值必须定义大于或等同 199309，才能使 `siginfo_t` 结构体的定义暴露在`<signal.h>`中。

在 Linux 和多数 UNIX 系统中，`siginfo_t` 中的许多域都被组合在 `union` 中，因为每个信号只需要使用其中的部分域（细节请参考`<bits/siginfo.h>`）

当进入一个信号处理器时，`siginfo_t` 结构体的域会被如下设置：

si_signo

这个域会为所有信号设置，包含导致信号处理器被调用的信号数值，等同于信号处理器的 `sig` 参数。

si_code

这个域会为所有信号设置，包含关于信号起源更多信息的一个代码，如表 21-1 所示。

si_value

这个域包含通过 `sigqueue()`发送的信号的伴随数据，我们会在[第 22.8.1 节](#)讨论 `sigqueue()`。

si_pid

对于通过 `kill()`或 `sigqueue()`发送的信号，这个域设置为发送进程的 ID。

si_uid

对于通过 `kill()`或 `sigqueue()`发送的信号，这个域设置为发送进程的实际用户 ID。系统提供发送进程的实际用户 ID，是因为实际用户 ID 比有效用户 ID 的信息更加准确。考虑第 20.5 节描述的发送信号的权限规则：如果有效用户 ID

授予发送进程发送信号的权限，则用户 ID 必须为 0（特权进程），或者等同于接收进程的实际用户 ID 或已保存的设置用户 ID。在这种情况下，接收信号进程如果能够知道发送进程的实际用户 ID，就更加有用（实际用户 ID 可能不等于有效用户 ID，如设置用户 ID 程序）。

si_errno

如果这个域设置为非 0 值，则包含一个错误数值（类似于 `errno`），表示信号的起因。这个域在 Linux 中通常未被使用。

si_addr

这个域只会被硬件生成的信号设置：`SIGBUS`, `SIGSEGV`, `SIGILL` 和 `SIGFPE`。对于 `SIGBUS` 和 `SIGSEGV` 信号，这个域包含导致非法内存引用的地址；对于 `SIGILL` 和 `SIGFPE` 信号，这个域包含导致信号产生的程序指令地址。

下面两个域是 Linux 的非标准扩展，仅被 POSIX 定时器（[第 23.6 节](#)）超期产生的信号设置：

si_timerid

这个域包含内核内部标识该定时器的 ID。

si_overrun

这个域设置为该定时器的超限次数。

接下来的两个域仅被 `SIGIO` 信号设置（[第 63.3 节](#)）：

si_band

这个域包含 I/O 事件相关联的“band event”值。（在 glibc 版本 2.3.2 之前，`si_band` 的类型为 `int`）。

si_fd

这个域包含 I/O 事件相关联的文件描述符。`SUSv3` 没有规定这个域，但许多 UNIX 实现都可用。

以下域仅由 SIGCHLD 信号设置（[第 26.3 节](#)）：

si_status

这个域要么包含子进程的退出状态（如果 `si_code` 是 `CLD_EXITED`），要么包含发送给子进程的信号数值（也就是终止或停止子进程的信号数值，如[第 26.1.3 节](#)所述）。

si_utime

这个域包含子进程使用的用户 CPU 时间。在内核 2.6.27 之前，用户 CPU 时间都是由系统时钟滴答来测量（除以 `sysconf(_SC_CLK_TCK)`）。在内核 2.6.27 之前的 2.6 系列内核中，一个 bug 意味着这个域的时间被报告为按（用户可配置的）jiffies 来测量（参考[第 10.6 节](#)）。SUSv3 没有规定这个域，但许多 UNIX 实现都可用。

si_stime

这个域包含子进程使用的系统 CPU 时间。参考关于 `si_utime` 域的相关讨论。SUSv3 没有规定这个域，但许多 UNIX 实现都可用。

`si_code` 域提供信号起源的更多信息，它的值如表 21-2 所列。表中第二列的值并不是所有 UNIX 系统和硬件体系架构（特别是四个硬件生成的信号：`SIGBUS`, `SIGSEGV`, `SIGILL`, 和 `SIGFPE`）都存在。不过 SUSv3 和 Linux 定义了大部分的常量。注意以下关于表 21-2 的关键点：

- `SI_KERNEL` 和 `SI_SIGIO` 是 Linux 特定的，SUSv3 没有规定，其它 UNIX 系统也没有定义。
- `SI_SIGIO` 只在 Linux 2.2 中使用，从 Linux 内核 2.4 开始，使用表中的 `POLL_*` 常量代替。

SUSv4 规定了 `psiginfo()` 函数，作用类似于 `psignal()`（[第 20.8 节](#)）。`psiginfo()` 函数有两个参数：指向 `siginfo_t` 的指针，和一个字符串。它指向标准错误的消息字符串，接着是 `siginfo_t` 结构体描述的信号信息。`psiginfo()` 函数由 glibc 2.10 开始提供。glibc 的实现会打印信号描述，信号起源（`si_code` 域），某些信号还会打印 `siginfo_t` 结构体的其它域。`psiginfo()` 是 SUSv4 定义的新函数，并不是所有 UNIX 系统都可用。

表 21-2: *siginfo_t* 结构体 *si_code* 域返回的值

信号	<i>si_code</i> 值	信号起源
任何信号	SI_ASYNCIO SI_KERNEL SI_MESGQ SI_QUEUE SI_SIGIO SI_TIMER SI_TKILL SI_USER	异步 I/O (AIO) 操作完成 由内核发送 (如终端驱动) POSIX 消息队列消息到达 (Linux 2.6.6 起) sigqueue() 用户进程实时信号 SIGIO 信号 (仅 Linux 2.2) POSIX (实时) 定时器超时 用户进程 <code>tkill()</code> 或 <code>tkill()</code> (Linux 2.4.19 起) 用户进程调用 <code>kill()</code> 或 <code>raise()</code>
SIGBUS	BUS_ADRALN BUS_ADRERR BUS_MCEERR_AO BUS_MCEERR_AR BUS_OBJERR	地址对齐非法 不存在的物理地址 硬件内存错误; 可选动作 (Linux 2.6.32 起) 硬件内存错误; 要求动作 (Linux 2.6.32 起) 对象特定的硬件错误
SIGCHLD	CLD_CONTINUED CLD_DUMPED CLD_EXITED CLD_KILLED CLD_STOPPED CLD_TRAPPED	子进程由 <code>SIGCONT</code> 继续 (Linux 2.6.9 起) 子进程异常终止, 有 core dump 子进程退出 子进程正常终止, 无 core dump 子进程停止 停止跟踪子进程
SIGFPE	FPE_FLTDIV FPE_FLTINV FPE_FLTOVF FPE_FLTRES FPE_FLTUND FPE_INTDIV FPE_INTOVF	浮点除以 0 非法浮点操作 浮点溢出 浮点不精确结果 浮点下溢 整数除以 0 整数溢出

	FPE_SUB	下标超出范围
SIGILL	ILL_BADSTK	内部堆栈错误
	ILL_COPROC	协处理器错误
	ILL_ILLADR	非法寻址模式
	ILL_ILOPC	非法操作码
	ILL_ILLOPN	非法操作数
	ILL_ILLTRP	非法陷阱
	ILL_PRVOPC	特权操作码
	ILL_PRVREG	特权寄存器
SIGPOLL/ SIGIO	POLL_ERR	I/O 错误
	POLL_HUP	设备断开连接
	POLL_IN	输入数据可用
	POLL_MSG	输入消息可用
	POLL_OUT	输出缓冲区可用
	POLL_PRI	高优先级输入可用
SIGSEGV	SEGV_ACCERR	映射对象权限非法
	SEGV_MAPERR	地址没有映射到对象
SIGTRAP	TRAP_BRANCH	进程分支陷阱
	TRAP_BRKPT	进程断点
	TRAP_HWBKPT	硬件断点/观察点
	TRAP_TRACE	进程跟踪陷阱

ucontext 参数

传递给使用 `SA_SIGINFO` 标志建立的信号处理器的最后一个参数是 `ucontext`，这是一个指向 `ucontext_t` 结构体的指针（定义在`<ucontext.h>`）。（SUSv3 规定这个参数为 `void` 指针，因为标准没有规定这个参数的细节）。这个结构体提供被称为“用户上下文”的信息，描述了进程调用信号处理器之前的状态，包括之前的进

程掩码和已保存的寄存器值（如程序计数器和堆栈指针）。这些信息很少在信号处理器中使用到，所以我们这里略过细节。

`ucontext_t` 结构体的另一个用法是 `getcontext()`、`makecontext()`、`setcontext()`、`swapcontext()` 函数，分别允许进程获取、创建、修改、交换执行上下文。（这些操作有一些类似 `setjmp()` 和 `longjmp()`，但更加通用）。这些函数可以用来实现 `coroutine`（协程），也就是进程的执行线程在两个（或多个）函数之间交替进行。SUSv3 规定了这些函数，但已经标记为废弃。SUSv4 移除了这些函数，并建议应用使用 POSIX 线程代替。`glibc` 手册提供了这些函数的更多信息。

21.5 系统调用的中断和重启

考虑以下场景：

1. 我们为信号建立一个处理器
2. 我们发起一个阻塞的系统调用，例如 `read()` 一个终端设备，会阻塞直到输入可用
3. 当系统调用被阻塞期间，信号被递送并调用信号处理器

信号处理器返回之后会发生什么呢？默认情况下，系统调用会以错误 `EINTR`（`Interrupted function`）失败。这可以是一个有用的特性，在[第 23.3 节](#)，我们会看到怎样使用定时器（会导致 `SIGALRM` 信号递送）来设置阻塞系统调用（如 `read()`）的超时。

但是通常我们更愿意继续执行被中断的系统调用，我们可以使用如下代码，在系统调用被信号处理器中断之后，手工重新启动一个系统调用：

```
while ((cnt = read(fd, buf, BUF_SIZE)) == -1 && errno == EINTR)
    continue;           /* Do nothing loop body */

if (cnt == -1)          /* read() failed with other than EINTR */
    errExit("read");
```

如果我们需要频繁地使用上述代码，可以定义一个宏简化代码，如下：

```
#define NO_EINTR(stmt) while ((stmt) == -1 && errno == EINTR);
```

使用这个宏，我们可以重写上面的 `read()` 代码示例：

```
NO_EINTR(cnt = read(fd, buf, BUF_SIZE));
if (cnt == -1)          /* read() failed with other than EINTR */
    errExit("read");
```

GNU C 库提供了一个（非标准的）宏（在`<unistd.h>`中），作用和我们的 `NO_EINTR()` 相同。这个宏名叫 `TEMP_FAILURE_RETRY()`，必须定义 `_GNU_SOURCE` 特性测试宏才能使用这个宏。

即使我们采用 `NO_EINTR()` 这样的宏，直接让信号处理器中断系统调用也非常不方便，因为我们必须为每个阻塞型系统调用增加代码（如果我们希望重新启动所有这些系统调用的话）。相反我们可以在 `sigaction()` 建立信号处理器时指定 `SA_RESTART` 标志，这样内核就会代替进程自动重新启动被中断的系统调用。这也意味着我们不需要处理这些系统调用的 `EINTR` 错误。

`SA_RESTART` 标志是单个信号的设置，换句话说，我们可以允许某些信号的处理器中断系统调用，其它信号的处理器则自动重新启动系统调用。

SA_RESTART 有效的系统调用（和库函数）

不幸的是，并不是所有阻塞型系统调用都能够指定 `SA_RESTART` 标志来自动重启。部分原因是历史性的：

- 4.2BSD 引入了重启系统调用，并涵盖了 `wait()` 和 `waitpid()` 的中断调用，以及以下 I/O 系统调用：`read()`, `readv()`, `write()`, `writev()`, 阻塞 `ioctl()` 操作。只有 I/O 系统调用操作“慢”设备时，才是可中断的，才能自动被 `SA_RESTART` 标志重新启动。慢设备包括终端、管道、FIFO、Socket。在这些文件类型中，许多 I/O 操作可能会阻塞。反过来，磁盘文件并不属于慢设备，因为磁盘 I/O 操作通常可以通过缓冲区缓存立即完成。如果要求执行磁盘 I/O 操作，内核会让进程睡眠直到 I/O 完成。
- 许多其它阻塞型系统调用起源于 System V，后者最初并不提供系统调用重启特性。

在 Linux 中，如果建立信号处理器时设置了 `SA_RESTART` 标志，以下阻塞型系统调用（和库函数）被信号处理器中断之后，会自动重启：

- 用来等待子进程的系统调用（[第 26.1 节](#)）：`wait()`, `waitpid()`, `wait3()`, `wait4()` 和 `waitid()`。
- 应用于慢设备的 I/O 系统调用：`read()`, `readv()`, `write()`, `writev()`, `ioctl()`。如果信号递送时数据已经传输了一部分，输入和输出系统调用会被中断，但会返回一个成功状态：表示已经成功传输的字节数。
- `open()` 系统调用阻塞时（如打开 FIFO，[第 44.7 节](#) 讨论）。
- 各种 `socket` 相关的系统调用：`accept()`, `accept4()`, `connect()`, `send()`, `sendmsg()`, `sendto()`, `recv()`, `recvfrom()`, `recvmsg()`。（在 Linux 中，如果使用 `setsockopt()` 设置了超时，这些系统调用不会自动重启，细节请参考 `signal(7)` 手册页）。
- 用于 POSIX 消息队列的系统调用：`mq_receive()`, `mq_timedreceive()`, `mq_send()`, `mq_timedsend()`。
- 用于放置文件锁的系统调用和库函数：`flock()`, `fcntl()`, `lockf()`。
- Linux 特定的 `futex()` 系统调用的 `FUTEX_WAIT` 操作。
- 用来递减 POSIX 信号量的 `sem_wait()` 和 `sem_timedwait()` 函数（在某些 UNIX 实现中，`sem_wait()` 在指定了 `SA_RESTART` 标志时也会重启）。
- 用于同步 POSIX 线程的函数：`pthread_mutex_lock()`, `pthread_mutex_trylock()`, `pthread_mutex_timedlock()`, `pthread_cond_wait()`, `pthread_cond_timedwait()`。

在内核 2.6.22 之前，无论是否设置 `SA_RESTART` 标志，`futex()`, `sem_wait()`，和 `sem_timedwait()` 总是会以错误 `EINTR` 失败。

以下阻塞型系统调用（和基于系统调用的库函数）永远不会自动重启（即使设置了 `SA_RESTART` 标志）：

- `poll()`, `ppoll()`, `select()`, 和 `pselect()` 等 I/O 多路复用调用。（SUSv3 明确规定

`select()` 和 `pselect()` 被信号处理器中断的行为未定义，无论是否设置 `SA_RESTART` 标志)。

- Linux 特定的 `epoll_wait()` 和 `epoll_pwait()` 系统调用。
- Linux 特定的 `io_getevents()` 系统调用。
- System V 消息队列和信号量相关的阻塞型系统调用：`semop()`, `semtimedop()`, `msgrcv()`, 和 `msgsnd()`。(尽管 System V 最初没有提供自动重启系统调用机制，但在某些 UNIX 实现中，如果指定了 `SA_RESTART` 标志，这些系统调用也会自动重启)。
- `read()` 读取 `inotify` 文件描述符。
- 用于暂停程序执行指定时间的系统调用和库函数：`sleep()`, `nanosleep()`, 和 `clock_nanosleep()`。
- 用于等待特定信号递送的系统调用：`pause()`, `sigsuspend()`, `sigtimedwait()`, 和 `sigwaitinfo()`。

为某个信号修改 `SA_RESTART` 标志

`siginterrupt()` 函数修改一个信号的 `SA_RESTART` 设置。

```
#include <signal.h>
```

```
int siginterrupt(int sig, int flag);
```

成功时返回 0；出错时返回 -1

如果 `flag` 为 `true` (1)，则该信号的处理器会中断阻塞型系统调用。如果 `flag` 为 `false` (0)，则阻塞型系统调用会在被信号处理器中断后自动重启。

`siginterrupt()` 函数内部使用 `sigaction()` 来获得信号当前的部署，修改返回 `oldact` 结构体中的 `SA_RESTART` 标志，然后再次调用 `sigaction()` 来更新信号的部署。

SUSv4 标记 `siginterrupt()` 已废弃，推荐使用 `sigaction()` 来替代。

未处理停止信号对某些 Linux 系统调用会生成 EINTR

在 Linux 中，某些阻塞型系统调用在没有信号处理器时也可能会返回 `EINTR`。例如系统调用阻塞期间，进程被信号停止 (`SIGSTOP`, `SIGTSTP`, `SIGTTIN`, `SIGTTOU`)，随后进程又被 `SIGCONT` 信号继续执行。

以下系统调用和函数就拥有这个行为：`epoll_pwait()`, `epoll_wait()`, `read()` 读取 `inotify` 文件描述符, `semop()`, `semtimedop()`, `sigtimedwait()`, `sigwaitinfo()`。

内核 2.6.24 之前，`poll()` 也有上述行为，以及 `sem_wait()`, `sem_timedwait()`, `futex()`(`FUTEX_WAIT`)；`msgrcv()` 在内核 2.6.22 之前，`msgsnd()` 在内核 2.6.9 之前，`nanosleep()` 在内核 2.4 之前，都拥有该行为。

在 Linux 2.4 之前，`sleep()` 也会按这种方式中断，但不会返回错误，而是返回剩余的未睡眠秒数。

这个行为的结果表示，即使我们的程序没有安装停止信号的处理器，也可能会被信号停止和继续，此时我们可能需要编写代码来重新启动这些系统调用。

21.6 小结

在这一章，我们讨论了一系列影响信号处理器设计和操作的因素。

由于信号不会排队，信号处理器必须处理以下情况：特定类型的多个事件发生，但最终只递送了一个信号。可重入影响了我们更新全局变量的方式，并限制了我们能够在信号处理器中安全使用的函数。

除了正常返回，信号处理器还可以按许多其它方式终止，包括调用 `_exit()`、通过发送信号(`kill()`, `raise()`, `abort()`)来终止进程、执行非局部跳转。使用 `sigsetjmp()` 和 `siglongjmp()`，程序在执行非局部跳转时，可以显式地控制进程的信号掩码。

我们可以使用 `sigaltstack()` 为进程定义一个备用信号堆栈。这块区域的内存替代标准进程堆栈，用于调用信号处理器使用。备用信号堆栈在标准堆栈耗尽时（此时内核会向进程发送 `SIGSEGV` 信号）处理信号非常有用。

`sigaction()` 的 `SA_SIGINFO` 标志允许建立信号处理器，可以获得信号的额外信息。这些信息通过 `siginfo_t` 结构体来提供，并作为参数传递给信号处理器。

当一个信号处理器中断阻塞型系统调用时，系统调用会以错误 `EINTR` 失败。

我们可以利用这个行为，为阻塞型系统调用设置定时器。如果有需要，我们可以手动重启被中断的系统调用。或者使用 `sigaction()` 的 `SA_RESTART` 标志来建立信号处理器，这样系统调用被中断后会自动重启。

更多信息

参考[第 20.15 节](#)列出的资源。

21.7 习题

- 21-1. 实现 `abort()`。

第 22 章 信号：高级特性

本章结束从[第 20 章](#)开始的关于信号的讨论，主要关注一些更加高级的主题，包括以下部分：

- core dump 文件；
- 信号递送、部署、处理的特殊情况；
- 信号的同步和异步生成；
- 信号递送的时间和顺序；
- 信号处理器中断系统调用，以及如何自动重启被中断的系统调用；
- 实时信号；
- 使用 `sigsuspend()` 来设置进程信号掩码并等待信号到达；
- 使用 `sigwaitinfo()`（以及 `sigtimedwait()`）来同步等待信号到达；
- 使用 `signalfd()` 通过文件描述符接收信号；
- 老式的 BSD 和 System V 信号 API。

22.1 Core Dump 文件

很多信号会导致进程创建一个 core dump 文件并终止（表 20-1）。core dump 文件包含进程终止时的内存镜像。（术语 `core` 起源于老的内存技术）。这个内存镜像可以装载到调试器中，从而检查信号到达时，进程的代码和数据的状态。

输入退出字符（通常是 `Ctrl-\`）是导致程序生成 core dump 的一种方式，退出字符会产生 `SIGQUIT` 信号：

```
$ ulimit -c unlimited          Explained in main text
$ sleep 30
Type Control-\
Quit (core dumped)
$ ls -l core                  Shows core dump file for sleep(1)
-rw----- 1 mtk users 57344 Nov 30 13:39 core
```

在这个例子中，shell 打印了“Quit (core dumped)”消息，表示 shell 检测到子进程（运行 sleep 的进程）被 SIGQUIT 杀死并生成了一个 core dump 文件。

core dump 文件创建在进程的当前工作目录，名字为 core。这是 core dump 文件默认的位置和名字；很快我们会讨论怎样改变这些默认选项。

许多系统提供工具（如 FreeBSD 和 Solaris 的 gcore）来获取正在运行进程的 core dump。Linux 也提供类似的功能，可以使用 gdb 附加到一个正在运行的进程，然后使用 gcore 命令来完成。

core dump 文件不会生成的情况

以下情况不会生成 core dump 文件：

- 进程没有权限来写入 core dump 文件。这可能是由于进程缺乏创建 core dump 文件所在目录的权限；或者相同名字的文件已经存在，而该文件无法写入或者不是一个普通文件（比如是一个目录或符号链接）。
- 相同名字的普通文件已经存在并且可以写入，但是有超过一个硬链接指向该文件。
- core dump 文件创建所在的目录并不存在。
- 进程资源限制中 core dump 文件的大小设置为 0。这个 RLIMIT_CORE 限制会在[第 36.3 节](#)更加详细地讨论。在上面示例，我们使用了 ulimit 命令（C shell 中使用 limit）来确保不对 core 文件进行限制。
- 进程资源限制 RLIMIT_FSIZE 的值设置为 0，这个 RLIMIT_FSIZE 可以限制进程能够生成的文件大小。我们会在[第 36.3 节](#)讨论这个限制。
- 进程执行的二进制可执行文件没有启用读权限。这可以阻止用户使用 core dump 来获取程序代码的拷贝。
- 当前工作目录所在的文件系统以只读方式挂载、空间已满、或 i-node 耗尽。或者用户已经达到文件系统的限额。
- 设置用户 ID(设置组 ID)程序不是由文件拥有者执行，不会生成 core dump 文件。这可以阻止恶意用户复制安全程序的镜像并检查它的敏感信息，如密码等。

使用 Linux 特定的 `prctl()` 系统调用的 `PR_SET_DUMPABLE` 操作，我们可以设置一个进程是否可以 `dump` 的标志。这样当非文件拥有者运行一个设置用户 ID（设置组 ID）程序时，也能够生成 `core dump` 文件。Linux 2.4 起支持 `PR_SET_DUMPABLE` 操作。更多细节请参考 `prctl(2)` 手册页。此外，从内核 2.6.13 开始，`/proc/sys/fs/suid_dumpable` 文件提供系统级控制，可以控制设置用户 ID 和设置组 ID 进程是否产生 `core dump` 文件。更多细节请参考 `proc(5)` 手册页。

从内核 2.6.23 开始，Linux 特定的 `/proc/PID/coredump_filter` 文件可以为单个进程设置过滤器，从而决定哪些类型的内存映射会被写入 `core dump` 文件。（我们在[第 49 章](#)讨论内存映射）。这个文件的值是四个位的掩码，分别对应于四种类型的内存映射：私有匿名映射、私有文件映射、共享匿名映射、共享文件映射。该文件的默认值提供了 Linux 的传统行为：只有私有匿名和共享匿名映射会被 `dump`。更多细节请参考 `core(5)` 手册页。

命名 `core dump` 文件：`/proc/sys/kernel/core_pattern`

从 Linux 2.6 开始，Linux 特定的 `/proc/sys/kernel/core_pattern` 文件中的格式字符串，控制了系统生成 `core dump` 文件的命名规则。默认 `core_pattern` 文件包含字符串“`core`”。特权用户可以按照表 22-1 显示的格式说明符，来修改这个格式字符串。这些格式说明符会被表中右侧的值替换，从而形成最终的 `core dump` 文件名。此外，格式字符串还可以包含斜杠 (/)，换句话说，我们不仅可以控制 `core dump` 的文件名，还可以指定其所在的目录（绝对或相对都可以）。替换了所有格式说明符之后，最终的路径名字符串会被截断为最多 128 个字符（Linux 2.6.19 之前最多 64 字符）。

从内核 2.6.19 开始，Linux 的 `core_pattern` 文件还支持一种额外的语法。如果该文件包含的字符串以管道符号 (|) 起始，则剩余的字符会解释为一个程序（可选的程序命令行参数也可以包含表 22-1 中的格式说明符），进程生成 `core` 时会调用这个程序，并且将 `core dump` 作为该程序的标准输入，而不会写入到文件。更多细节请参考 `core(5)` 手册页。

有些 UNIX 系统也提供类似于 `core_pattern` 的机制。例如 BSD 变种会把程序名附加到 `core dump` 文件名，最终的格式为 `core.progname`。Solaris 则提供一个工具 (`coreadm`)，允许用户选择 `core dump` 的文件名和目录。

表 22-1：/proc/sys/kernel/core_pattern 的格式说明符

说明符	替换为
%c	Core 文件大小软资源限制（字节，Linux 2.6.24 起）
%e	可执行文件名（不带路径前缀）
%g	被 dump 进程的实际组 ID
%h	主机系统名
%p	被 dump 进程的进程 ID
%s	终止进程的信号数值
%t	dump 时间，Epoch 起的秒数
%u	被 dump 进程的实际用户 ID
%%	单个%字符

22.2 递送、部署、和处理的特殊情况

对于某些特定信号，其递送、部署、处理会应用特殊的规则，如本节所述。

SIGKILL 和 SIGSTOP

我们不能改变这两个信号的默认动作，SIGKILL 总是会终止一个进程；而 SIGSTOP 则总是停止一个进程。使用 `signal()` 和 `sigaction()` 来修改这两个信号的部署都会返回错误。这两个信号也不能被阻塞，这是一个明确的设计选择。不允许修改这两个信号的默认动作，意味着总是可以使用它们来杀死或停止一个进程。

SIGCONT 和停止信号

前面已经讲过，SIGCONT 信号用来继续一个之前停止的进程（停止信号 SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU）。由于这些信号拥有独特作用，某些情况下内核对它们的处理与其它信号会有不同。

如果一个进程当前被停止，则 SIGCONT 信号到达总是会导致进程继续执行，

即使进程当前阻塞或忽略 **SIGCONT** 信号。这个特性是必要的，否则我们将无法继续这些停止的进程。（如果停止进程阻塞 **SIGCONT** 信号，并为 **SIGCONT** 建立了一个信号处理器，当进程继续执行之后，信号处理器只有在稍后解除阻塞 **SIGCONT** 之后才会被调用）。

如果其它信号发送给一个已经停止的进程，信号不会被递送给进程，直到进程被 **SIGCONT** 信号继续执行。唯一的例外是 **SIGKILL**，该信号总是会杀死一个进程，即使进程当前已经停止。

无论何时 **SIGCONT** 被递送至一个进程，该进程的任何未决停止信号都会被丢弃（进程永远不会看到这些信号）。相反如果任何停止信号递送至进程，则任何未决 **SIGCONT** 信号也会被自动丢弃。采取这些步骤，是为了防止 **SIGCONT** 信号的动作被之前发送的停止信号撤销；或者停止信号的动作被之前发送的 **SIGCONT** 信号撤销。

不要改变被忽略的终端产生信号的部署

如果程序被执行时，发现一个终端产生的信号的部署被设置为 **SIG_IGN**（忽略），则通常程序不应该试图修改该信号的部署。系统并没有强制要求遵守这个规则，而是编写应用时应该遵循的一个惯例。我们在[第 34.7.3 节](#)解释这样做的原因。这条规则相关的信号包括 **SIGHUP**, **SIGINT**, **SIGQUIT**, **SIGTTIN**, **SIGTTOU**, **SIGTSTP**。

22.3 可中断和不可中断的进程睡眠状态

之交我们提到 **SIGKILL** 和 **SIGSTOP** 总是立即杀死或停止进程，但现在我们需要对此增加一个附加条件。在许多时候，内核可能会把进程设置为睡眠，而且这里存在两种不同的睡眠状态：

- **TASK_INTERRUPTIBLE**: 进程等待某些事件。例如等待终端输入、等待数据写入当前空管道、等待 System V 信号量增加。进程可能会在这个状态下等待任意长的时间。如果此时有信号产生，会中断进程的等待操作，信号递送唤醒进程。使用 **ps(1)** 命令时，**TASK_INTERRUPTIBLE** 状态的进程会在 **STAT**（进程状态）域中显示字母 **S**。

- **TASK_UNINTERRUPTIBLE**: 进程正在等待某种特殊类型的事件，例如磁盘 I/O 完成。如果进程在此状态下，有信号产生，则信号不会递送，直到进程脱离 **TASK_UNINTERRUPTIBLE** 状态。使用 `ps(1)` 命令时，这种状态下的进程会在 `STAT` 域中标记为字母 `D`。

由于一个进程通常只会在 **TASK_UNINTERRUPTIBLE** 状态中保持非常短的时间，因此只有进程离开这个状态才会递送信号这一事实，其实基本上可以忽略。但是在少数场景中，进程可能会挂起在 **TASK_UNINTERRUPTIBLE** 状态，例如硬件故障、NFS 问题、内核 bug 等。在这种情况下，`SIGKILL` 也不能终止挂起的进程。如果底层问题不能解决的话，我们就必须重启系统才能消除该进程。

TASK_INTERRUPTIBLE 和 **TASK_UNINTERRUPTIBLE** 状态在多数 UNIX 系统中都存在。从内核 2.6.25 开始，Linux 增加了第三个状态来解决我们刚刚讨论的挂起进程的问题：

- **TASK_KILLABLE**: 这个状态类似于 **TASK_UNINTERRUPTIBLE**，但在接收到 `fatal` 信号（也就是能杀死进程的信号）时会唤醒进程。通过修改内核相关部分的代码来使用这个状态，原先许多场景下需要强制重启系统来终止挂起进程的情况得以避免。相反可以向进程发送 `fatal` 信号来终止它。第一个使用 **TASK_KILLABLE** 状态的内核代码模块是 NFS。

22.4 硬件产生的信号

`SIGBUS`, `SIGFPE`, `SIGILL`, `SIGSEGV` 信号会由于硬件异常而产生，当然也可以使用 `kill()` 来发送这些信号，不过非常少见。当硬件出现异常时，SUSv3 规定如果进程从信号处理器中返回，或者进程忽略或阻塞这些信号，则进程的行为是未定义的。原因如下：

- **从信号处理器返回**: 假设由于机器语言指令产生了上述某个信号，并导致信号处理器被调用。从信号处理器中正常返回时，程序会试图在中断位置继续执行。但这个位置正是之前产生该信号的那条指令，因此会再次产生该信号。结果通常就是程序进入无限循环，反复地调用信号处理

器。

- **忽略信号：**忽略一个硬件产生的信号几乎没有丝毫意义，因为程序并不知道该如何继续执行（如算术异常之后）。当上述某个信号由于硬件异常而产生之后，Linux 会强制信号递送，即使程序已经请求忽略该信号。
- **阻塞信号：**和上一种情况一样，阻塞硬件产生的信号也没有意义，因为程序也不知道该如何继续执行。在 Linux 2.4 及之前，内核简单地忽略对硬件产生信号的阻塞；信号无论如何都会递送给进程，然后要么终止进程，要么被信号处理器捕获。从 Linux 2.6 开始，如果硬件产生的信号被阻塞，则进程总是会立即被信号杀死，即使进程已经为该信号安装了处理器。（Linux 2.6 对硬件产生信号这样处理的原因，是因为 Linux 2.4 的行为隐藏了 bug，会导致多线程程序出现死锁）。

本书源码包中的 `signals/demo_SIGFPE.c` 程序，可以用来演示忽略和阻塞 SIGFPE 信号、以及捕获 SIGFPE 信号之后从信号处理器中正常返回，会出现的结果。

处理硬件产生信号正确的方式，要么是接受信号的默认动作（进程终止），要么编写不会正常返回的信号处理器。信号处理器可以通过调用 `_exit()` 来终止进程，也可以调用 `siglongjmp()`（[第 21.2.1 节](#)）来确保程序的执行流程转到某个确定的位置，而不是之前导致硬件异常的指令位置。

22.5 同步和异步信号生成

我们已经知道进程通常不能预测何时会收到信号，现在我们需要通过区分同步和异步信号产生，来重新阐述这一点。

到目前为止我们考虑的都是隐式的异步信号产生，这样的信号由其它进程或内核生成，基于某个独立于接收进程执行流程的事件发生。（例如用户按下中断字符或子进程终止）。对于异步生成的信号来说，进程确实无法预测信号生成的准确时间。

但是在某些情况下，信号是由正在执行的进程自己生成的。我们已经提到两个这样的例子：

- [第 22.4 节](#) 讨论的硬件产生的信号 (SIGBUS, SIGFPE, SIGILL, SIGSEGV, SIGEMT)，这些信号由于进程执行了特定的机器语言指令导致硬件异常而产生。
- 进程可以使用 `raise()`, `kill()`, `killpg()` 向自己发送信号。

在这些情况下，信号生成是同步的（信号立即递送，除非信号被阻塞，参考[第 22.4 节](#)关于阻塞硬件产生信号的处理）。换句话说，之前关于信号递送是不可预测的描述，在这里并不适用。对于同步生成的信号，递送是可预测且可重复的。

注意同步是描述信号怎样产生的一个属性，而不是描述信号本身。所有信号都可以同步生成（进程使用 `kill()` 向自己发送信号），也可以异步生成（如其它进程使用 `kill()` 向某个进程发送信号）。

22.6 信号递送的时间和顺序

本节的第一个主题，讨论未决信号何时被递送。然后我们考虑多个阻塞的未决信号同时解锁时，会发生什么。

信号何时递送？

如[第 22.5 节](#)所述，同步生成的信号会被立即递送。例如硬件异常会立即触发信号；而进程使用 `raise()` 向自己发送信号时，信号会在 `raise()` 调用返回之前被递送。

对于异步生成的信号，未决信号在产生和实际递送之间，可能会有（很短的）延迟时间，即使我们并没有阻塞该信号。原因在于内核只有在下一次从内核模式切换到用户模式并执行该进程时，才会递送未决信号。在实践中，这意味着信号可能会在以下几个时间点被递送：

- 进程在之前时间片用完之后被重新调度（下一个时间片开始时）；
- 某个系统调用完成的时候（信号递送可能会导致一个阻塞系统调用提前完成）。

多个解除阻塞的信号的递送顺序

如果进程使用 `sigprocmask()` 解除阻塞多个未决信号，则所有这些信号会被立即递送给进程。

在当前的实现中，Linux 内核会按升序来递送信号。例如，未决信号 `SIGINT`（信号 2）和 `SIGQUIT`（信号 3）同时被解除阻塞，则 `SIGINT` 信号会先于 `SIGQUIT` 信号被递送，无论这两个信号产生的顺序先后。

但是我们不能依赖于（标准）信号的递送顺序，因为 SUSv3 规定多个信号的递送顺序是实现自定义的。（这个描述只适用于标准信号，我们在[第 22.8 节](#)将会看到，标准明确规定了实时信号的递送顺序）。

当多个解除阻塞的信号等待递送时，如果在执行信号处理器时发生了内核模式与用户模式之间的切换，则原来信号处理器的执行会被第二个信号处理器的调用中断（依次类推），如图 22-1 所示。

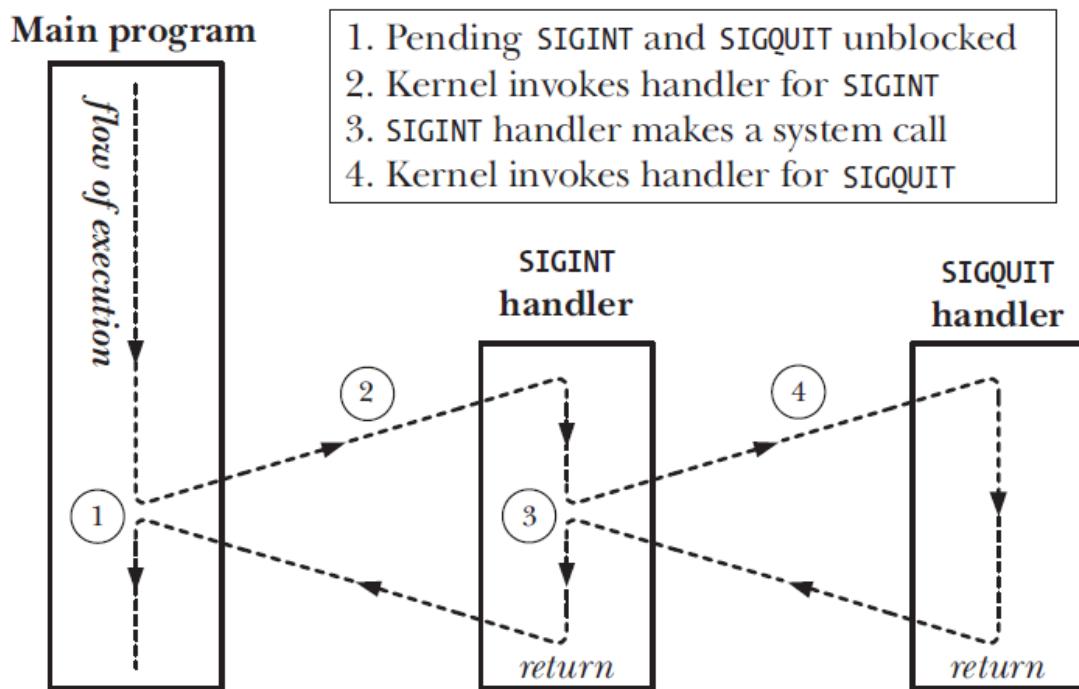


图 22-1：多个解除阻塞的信号递送

22.7 signal() 的实现和可移植性

在这一节，我们讨论如何使用 `sigaction()` 来实现 `signal()`。这种实现方式非常直接，但需要考虑不同 UNIX 系统的历史和差异，`signal()` 拥有许多不同的语义。特别需要注意的一点是，早期信号实现是不可靠的，意味着：

- 进入信号处理器时，信号的部署会重置为默认动作（这对应于[第 20.13 节](#)讨论的 `SA_RESETHAND` 标志）。为了在随后相同信号递送时能够再次调用到信号处理器，程序员需要在信号处理器中显式地使用 `signal()` 再次建立信号处理器。这里的问题在于进入信号处理器和重新建立处理器之间存在一个很小的时间窗口，在此期间如果相同信号第二次到达，可能会按信号的默认部署来进行处理。
- 执行信号处理器时，信号的再次递送不会被阻塞（这对应于[第 20.13 节](#)讨论的 `SA_NODEFER` 标志）。这意味着信号处理器被执行时，如果信号再次被递送，则信号处理器可能会被递归地调用。如果信号源源不断地到达，大量的递归调用信号处理器可能会导致堆栈溢出。

除了不可靠之外，早期 UNIX 实现还不提供系统调用的自动重启（也就是[第 21.5 节](#)讨论的 `SA_RESTART` 标志的行为）。

4.2BSD 提供的可靠信号实现改正了这些限制，其它一些 UNIX 系统也如法炮制。但是老式的语义在今天 System V 的 `signal()` 中仍然存在，而且当前的 SUSv3 和 C99 标准都故意规定 `signal()` 的这些方面未定义。

把上面所有信息综合起来，我们在清单 22-1 中实现了 `signal()`。默认这个实现会提供现代的信号语义；如果以 `-DOLD_SIGNAL` 进行编译，则它提供早期的不可靠信号语义，并且不启用系统调用的自动重启。

清单 22-1： signal() 的一种实现

```
-----signals/signal.c
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t
signal(int sig, sighandler_t handler)
{
    struct sigaction newDisp, prevDisp;

    newDisp.sa_handler = handler;
    sigemptyset(&newDisp.sa_mask);
#ifndef OLD_SIGNAL
    newDisp.sa_flags = SA_RESETHAND | SA_NODEFER;
#else
    newDisp.sa_flags = SA_RESTART;
#endif

    if (sigaction(sig, &newDisp, &prevDisp) == -1)
        return SIG_ERR;
    else
        return prevDisp.sa_handler;
}
-----signals/signal.c
```

一些 glibc 细节

glibc 对 signal() 库函数的实现随着时间而有一定变化。在较新版本的 glibc 中（glibc 2 及之后），默认提供的是现代信号语义。在老版本的库中，则提供了早期不可靠信号语义（兼容 System V）。

Linux 内核实现 signal() 为一个系统调用，提供老式、不可靠的语义。但是 glibc 绕过了这个系统调用，以库函数的形式提供 signal()，底层则调用了 sigaction()。

如果你想在现代版本的 glibc 中获得不可靠的信号语义，可以显式地替换 signal() 函数为非标准的 sysv_signal() 函数。

```
#define _GNU_SOURCE  
#include <signal.h>  
  
void (*sysv_signal(int sig, void (*handler)(int)))(int);  
成功时返回之前的信号部署；出错时返回 SIG_ERR
```

sysv_signal()函数的参数和 signal()完全一样。

如果编译程序时没有定义_BSD_SOURCE 特性测试宏，glibc 会隐式地重定义所有 signal()为 sysv_signal()调用，意味着 signal()拥有不可靠信号语义。默认情况下_BSD_SOURCE 特性测试宏已经定义，但是如果显式定义了其它特性测试宏(_SVID_SOURCE 或_XOPEN_SOURCE)，则会禁用_BSD_SOURCE 特性测试宏。

sigaction()是建立信号处理器的首选 API

由于上面讨论的 System V 和 BSD（以及老式和现代的 glibc 版本）存在的可移植性问题，我们最好总是使用 sigaction()来建立信号处理器。本书剩余部分遵循了这个建议（另一个选择是编写我们自己的 signal()，类似于清单 22-1，指定我们需要的标志，并在应用中采用相应的信号语义）。但是注意使用 signal()来设置信号部署为 SIG_IGN 或 SIG_DFL 是可移植的，相比 sigaction()也更加简单，因此我们通常会在这种情况下使用 signal()。

22.8 实时信号

POSIX.1b 定义了实时信号，来弥补标准信号的许多限制。实时信号相比标准信号有以下优点：

- 实时信号提供了更多可供应用自定义使用的信号，标准信号则只提供两个应用可自由使用的信号：SIGUSR1 和 SIGUSR2。
- 实时信号会排对。如果相同的实时信号被多次发送给某个进程，信号会被递送多次。与之相比，如果进程的某个信号仍然是未决（pending）状态，我们再多次发送同一个信号给进程，则信号只会被递送一次。

- 在发送实时信号时，可以为信号指定相应的数据（一个整数或指针）。接收进程的信号处理器可以取得这个数据。
- 不同的实时信号被递送的顺序是确定的。如果多个不同的实时信号都是未决状态，则最小编号的信号会被首先递送。换句话说，信号拥有优先级，数值小的信号拥有更高优先级。当多个相同类型的信号进行排队时，会按发送信号的顺序依次递送。

SUSv3 要求至少实现 `_POSIX_RTSIG_MAX`（定义的值是 8）个不同的实时信号。

Linux 内核定义了 32 个不同的实时信号，编号从 32 到 63。`<signal.h>` 头文件定义了 `RTSIG_MAX` 常量来标明可用的实时信号数量，以及常量 `SIGRTMIN` 和 `SIGRTMAX` 来标明实时信号最小和最大的编号。

在采用 `LinuxThreads` 线程模型的系统中，`SIGRTMIN` 被定义为 35 而不是 32，因为 `LinuxThreads` 内部使用了 32-34 这三个编号的实时信号。在采用 `NPTL` 线程模型的系统中，`SIGRTMIN` 被定义为 34，因为 `NPTL` 内部使用了 32、33 两个实时信号。

标准信号使用不同的常量来标识每个单独的信号，但是实时信号却不是这样的。应用不能用硬编码的整数来使用实时信号，因为实时信号的编号范围在不同的 UNIX 系统是不一样的。因此需要通过 `SIGRTMIN` 加上一个偏移量来引用实时信号，例如(`SIGRTMIN + 1`)引用的就是第二个实时信号。

注意 SUSv3 没有要求 `SIGRTMAX` 和 `SIGRTMIN` 必须是简单的整数类型，它们可能会被定义为函数（Linux 就是这样做的）。这意味着我们不能编写下面这样的预处理代码：

```
#if SIGRTMIN+100 > SIGRTMAX /* WRONG! */
#error "Not enough realtime signals"
#endif
```

我们必须在运行时才能执行这个检查操作。

实时信号排队的数量限制

实时信号排队（以及伴随信号的数据）要求内核为每个进程维护一个排队信号列表的数据结构。这些数据结构会消耗内核内存，因此内核会对实时信号排队

的数量进行限制。

SUSv3 允许系统设置每个进程所有类型的实时信号排队数量的上限，但要求这个限制值至少为 `_POSIX_SIGQUEUE_MAX`（定义为 32）。系统可以定义常量 `SIGQUEUE_MAX` 来指示实时信号排队的最大数量，也可以通过下面调用来返回这个信息：

```
lim = sysconf(_SC_SIGQUEUE_MAX);
```

在 Linux 中，这个调用会返回 -1。原因是 Linux 采用了不同的模型来限制某个进程实时信号排队的数量。Linux 2.6.7（包含）之前的版本，内核强制使用了一个系统级的限制值，作为所有进程所有实时信号排队的总数量限制。我们可以通过 Linux 特定的 `/proc/sys/kernel/rtsig-max` 文件来查看和修改（需要权限）这个限制值。默认的限制值是 1024，当前排队的实时信号数量则可以在 Linux 特定的 `/proc/sys/kernel/rtsig-nr` 文件中查看到。

从 Linux 2.6.8 开始，改变了实时信号排队数量限制的模型，前面提到的 `/proc` 接口被移除。新的模型使用了 `RLIMIT_SIGPENDING` 软资源限制，它定义了某个特定的实际用户 ID 拥有的所有进程的实时信号排队数量限制。我们会在[第 36.3 节](#)进一步地讨论这个限制值。

使用实时信号

要在两个进程之间发送和接收实时信号，SUSv3 要求以下两点：

- 发送进程使用 `sigqueue()` 系统调用来发送信号和数据。

实时信号也可以使用 `kill()`, `killpg()`, `raise()` 来发送，但是 SUSv3 让系统来决定是否需要对这样的实时信号进行排队。在 Linux 中，这些接口都会对实时信号进行排队；但是在许多其它 UNIX 系统中，这些接口不会对实时信号排队。

- 接收进程使用 `sigaction()` 并指定 `SA_SIGINFO` 标志来建立信号处理器。这样信号处理器才会带有额外参数，其中一个参数包含了实时信号指定的数据。

在 Linux 中，即使接收进程在建立信号处理器时不指定 `SA_SIGINFO` 标志，实时信

号也会进行排队，当然这种情况下是无法得到实时信号传递的数据。SUSv3 不要求实现在未指定 SA_SIGINFO 标志时也必须对实时信号排队，因此我们不能依赖于 Linux 的这个特性。

22.8.1 发送实时信号

`sigqueue()`系统调用向 pid 指定的进程发送 sig 指定的实时信号。

```
#define _POSIX_C_SOURCE 199309
#include <signal.h>

int sigqueue(pid_t pid, int sig, const union sigval value);
```

成功时返回 0；出错时返回-1

使用 `sigqueue()`发送信号和 `kill()`需要相同的权限（参考[第 20.5 节](#)），也可以像 `kill()`那样发送 null 信号（信号 0）。（和 `kill()`不一样的是，我们不能为 `sigqueue()` 的 pid 指定负数值，也就是说不能向整个进程组发送信号）。

清单 22-2：使用 `sigqueue()` 来发送实时信号

```
----- signals/t_sigqueue.c
#define _POSIX_C_SOURCE 199309
#include <signal.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int sig, numSigs, j, sigData;
    union sigval sv;

    if (argc < 4 || strcmp(argv[1], "--help") == 0)
        usageErr("%s pid sig-num data [num-sigs]\n", argv[0]);

    /* Display our PID and UID, so that they can be compared with the
       corresponding fields of the siginfo_t argument supplied to the
       handler in the receiving process */

    printf("%s: PID is %ld, UID is %ld\n",
           argv[0],
           (long) getpid(), (long) getuid());
```

```

    sig = getInt(argv[2], 0, "sig-num");
    sigData = getInt(argv[3], GN_ANY_BASE, "data");
    numSigs = (argc > 4) ? getInt(argv[4], GN_GT_0, "num-sigs") : 1;

    for (j = 0; j < numSigs; j++) {
        sv.sival_int = sigData + j;
        if (sigqueue(getLong(argv[1], 0, "pid"), sig, sv) == -1)
            errExit("sigqueue %d", j);
    }

    exit(EXIT_SUCCESS);
}
----- signals/t_sigqueue.c

```

`value` 参数指定信号的数据，格式如下：

```

union sigval {
    int sival_int; /* 信号整数数据 */
    void *sival_ptr; /* 信号指针数据 */
};

```

对这个参数的解释是由应用自定义的，应用需要自己决定使用联合体的 `sival_int` 或 `sival_ptr` 域。不过 `sival_ptr` 域对 `sigqueue` 来说其实是很少用的，因为某个进程的指针数据对于另一个进程来说意义不大。但是对于其它使用这个联合体的函数来说，`sival_ptr` 域就是非常有用的。例如[第 23.6 节](#)讨论的 POSIX 定时器，以及[第 52.6 节](#)讨论的 POSIX 消息队列通知。

很多 UNIX 系统（包括 Linux）定义了一个 `sigval_t` 数据类型，作为 `union sigval` 的代名词。不过 SUSv3 并没有规定 `sigval_t` 类型，在某些 UNIX 中并不可用，因此可移植应用应该避免使用 `sigval_t`。

调用 `sigqueue()` 可能会由于信号排队数量超限而失败，在这种情况下，`errno` 会被设置为 `EAGAIN`，表示我们需要稍后再次发送这个信号（等当前排队信号被递送之后）。

清单 22-2 提供了使用 `sigqueue()` 的示例，这个程序至多接受四个参数，前三个是强制要求的：信号编号、目标进程 ID、信号数据（整数值）。如果需要多次发送这个信号，可以使用第四个参数来指定信号的发送数量；在这种情况下，每次发送信号时传递的数据会加 1。我们会在[第 22.8.2 节](#)演示这个程序的使用。

22.8.2 处理实时信号

我们可以像标准信号那样处理实时信号，使用普通的信号处理器（单个参数）；或者也可以指定 `SA_SIGINFO` 标志（[第 21.4 节](#)），此时需要使用三个参数的信号处理器来处理实时信号。下面是使用 `SA_SIGINFO` 标志的示例：

```
struct sigaction act;

sigemptyset(&act.sa_mask);
act.sa_sigaction = handler;
act.sa_flags = SA_RESTART | SA_SIGINFO;

if (sigaction(SIGRTMIN + 5, &act, NULL) == -1)
    errExit("sigaction");
```

当我们使用 `SA_SIGINFO` 标志时，信号处理器的第二个参数是 `siginfo_t` 结构体，包含了实时信号的额外信息。我们在[第 21.4 节](#)已经详细讨论了这个结构体。对于实时信号来说，`siginfo_t` 结构体的下列域会被设置：

- `si_signo` 域的值与信号处理器的第一个参数相同；
- `si_code` 域指示信号的来源，包含表 21-2 中的某个值。如果使用 `sigqueue()` 来发送实时信号，这个域的值总是 `SI_QUEUE`；
- `si_value` 域包含实时信号的数据，也就是进程使用 `sigqueue()` 发送信号时指定的 `sigval` 联合体。前面已经提到过，这个数据由应用解释。（如果使用 `kill()` 来发送信号，这个域的值是非法的。）
- `si_pid` 和 `si_uid` 域分别表示发送信号进程的进程 ID 和实际用户 ID。

清单 22-3 提供了处理实时信号的示例程序。这个程序捕获信号并显示传递给信号处理器的 `siginfo_t` 结构体的各种信息。这个程序可以有两个可选的整数命令行参数，如果指定了第一个参数，主程序会阻塞所有信号，然后睡眠这个参数指定的秒数，在这段时间内，发送给这个进程的信号会排队，因此我们可以查看信号解除阻塞时发生的事情；第二个参数指定信号处理器在返回之前需要睡眠的秒数，指定一个非 0 值（默认值是 1）可以让程序慢下来，我们可以更容易地看到

多个信号被处理时会发生什么。

我们可以配合清单 22-3 和清单 22-2 的程序使用，来分析实时信号的行为，如以下 shell 会话日志所示：

```
$ ./catch_rtsigs 60 &
[1] 12842
$ ./catch_rtsigs: PID is 12842 Shell prompt mixed with program output
./catch_rtsigs: signals blocked - sleeping 60 seconds
Press Enter to see next shell prompt
$ ./t_sigqueue 12842 54 100 3 Send signal three times
./t_sigqueue: PID is 12843, UID is 1000
$ ./t_sigqueue 12842 43 200
./t_sigqueue: PID is 12844, UID is 1000
$ ./t_sigqueue 12842 40 300
./t_sigqueue: PID is 12845, UID is 1000
```

catch_rtsigs 程序结束睡眠，它会显示信号处理器捕获各种信号的消息。（catch_rtsigs 程序在后台运行，所以它的输出消息会在下一行）。我们首先可以发现实时信号是最小编号首先递送，然后 `siginfo_t` 结构体中包含了发送信号进程的进程 ID 和用户 ID。

```
$ ./catch_rtsigs: sleep complete
caught signal 40
    si_signo=40, si_code=-1 (SI_QUEUE), si_value=300
    si_pid=12845, si_uid=1000
caught signal 43
    si_signo=43, si_code=-1 (SI_QUEUE), si_value=200
    si_pid=12844, si_uid=1000
```

接下来的输出由三个相同的实时信号产生。我们可以查看 `si_value` 域，从而能够确定信号是按发送顺序被递送的：

```
caught signal 54
    si_signo=54, si_code=-1 (SI_QUEUE), si_value=100
    si_pid=12843, si_uid=1000
caught signal 54
    si_signo=54, si_code=-1 (SI_QUEUE), si_value=101
    si_pid=12843, si_uid=1000
caught signal 54
    si_signo=54, si_code=-1 (SI_QUEUE), si_value=102
    si_pid=12843, si_uid=1000
```

接下来我们使用 shell 的 kill 命令向 catch_rtsigs 程序发送信号。和前面一样，我们看到 siginfo_t 结构体中包含了发送进程的进程 ID 和用户 ID，但在这种情况下，si_code 的值是 SI_USER：

```
Press Enter to see next shell prompt
$ echo $$ Display PID of shell
12780
$ kill -40 12842 Uses kill(2) to send a signal
$ caught signal 40
    si_signo=40, si_code=0 (SI_USER), si_value=0
    si_pid=12780, si_uid=1000 PID is that of the shell
Press Enter to see next shell prompt
$ kill 12842 Kill catch_rtsigs by sending SIGTERM
Caught 6 signals
按下回车来查看后台任务终止的通知
[1]+ Done ./catch_rtsigs 60
```

清单 22-3：处理实时信号

```
----- signals/catch_rtsigs.c
#define _GNU_SOURCE
#include <string.h>
#include <signal.h>
#include "tlpi_hdr.h"

static volatile int handlerSleepTime;
static volatile int sigCnt = 0; /* Number of signals received */
static volatile int allDone = 0;

static void /* Handler for signals established using SA_SIGINFO */
siginfoHandler(int sig, siginfo_t *si, void *ucontext)
{
    /* UNSAFE: This handler uses non-async-signal-safe functions
       (printf()); see Section 21.1.2) */

    /* SIGINT or SIGTERM can be used to terminate program */

    if (sig == SIGINT || sig == SIGTERM) {
        allDone = 1;
        return;
    }
}
```

```
sigCnt++;

printf("caught signal %d\n", sig);

printf(" si_signo=%d, si_code=%d (%s), ", si->si_signo, si->si_code,
       (si->si_code == SI_USER) ? "SI_USER" :
       (si->si_code == SI_QUEUE) ? "SI_QUEUE" : "other");
printf("si_value=%d\n", si->si_value.sival_int);
printf(" si_pid=%ld, si_uid=%ld\n", (long) si->si_pid,
       (long) si->si_uid);

sleep(handlerSleepTime);
}

int
main(int argc, char *argv[])
{
    struct sigaction sa;
    int sig;
    sigset_t prevMask, blockMask;

    if (argc > 1 && strcmp(argv[1], "--help") == 0)
        usageErr("%s [block-time [handler-sleep-time]]\n", argv[0]);

    printf("%s: PID is %ld\n", argv[0], (long) getpid());

    handlerSleepTime = (argc > 2) ?
        getInt(argv[2], GN_NONNEG, "handler-sleep-time") : 1;

    /* Establish handler for most signals. During execution of the handler,
       mask all other signals to prevent handlers recursively interrupting
       each other (which would make the output hard to read). */

    sa.sa_sigaction = siginfoHandler;
    sa.sa_flags = SA_SIGINFO;
    sigfillset(&sa.sa_mask);

    for (sig = 1; sig < NSIG; sig++)
        if (sig != SIGTSTP && sig != SIGQUIT)
            sigaction(sig, &sa, NULL);

    /* Optionally block signals and sleep, allowing signals to be
```

```

    sent to us before they are unblocked and handled */

if (argc > 1) {
    sigfillset(&blockMask);
    sigdelset(&blockMask, SIGINT);
    sigdelset(&blockMask, SIGTERM);

    if (sigprocmask(SIG_SETMASK, &blockMask, &prevMask) == -1)
        errExit("sigprocmask");

    printf("%s: signals blocked - sleeping %s seconds\n",
           argv[0], argv[1]);
    sleep(getInt(argv[1], GN_GT_0, "block-time"));
    printf("%s: sleep complete\n", argv[0]);

    if (sigprocmask(SIG_SETMASK, &prevMask, NULL) == -1)
        errExit("sigprocmask");
}

while (!allDone) /* Wait for incoming signals */
    pause();
}
----- signals/catch_rtsigs.c

```

22.9 使用掩码来等待信号：sigsuspend()

在解释 `sigsuspend()` 的作用之前，我们首先讨论需要使用它的一个场景。我们在进行信号编程时，有时候会遇到下面的情况：

1. 我们临时阻塞了一个信号，这样该信号的处理器就不会中断某些关键临界代码区的执行。
2. 我们解除信号的阻塞，然后程序挂起直到信号被递送。

要做到这些，我们可能会编写如下清单 22-4 的代码。

清单 22-4：解除阻塞并等待信号的错误做法

```
sigset_t prevMask, intMask;
```

```
struct sigaction sa;

sigemptyset(&intMask);
sigaddset(&intMask, SIGINT);

sigemptyset(&sa.sa_mask);
sa.sa_flags = 0;
sa.sa_handler = handler;

if (sigaction(SIGINT, &sa, NULL) == -1)
    errExit("sigaction");

/* Block SIGINT prior to executing critical section. (At this
   point we assume that SIGINT is not already blocked.) */

if (sigprocmask(SIG_BLOCK, &intMask, &prevMask) == -1)
    errExit("sigprocmask - SIG_BLOCK");

/* Critical section: do some work here that must not be
   interrupted by the SIGINT handler */

/* End of critical section - restore old mask to unblock SIGINT */

if (sigprocmask(SIG_SETMASK, &prevMask, NULL) == -1)
    errExit("sigprocmask - SIG_SETMASK");

/* BUG: what if SIGINT arrives now... */

pause(); /* Wait for SIGINT */
-----
```

上面代码有一个问题，假设 SIGINT 信号在执行第二个 `sigprocmask()` 之后，但是执行 `pause()` 之前被递送。（信号可能在执行临界区代码的任意时间已经产生，一旦解除阻塞就会立即递送）。此时递送 SIGINT 信号会导致信号处理器被调用，在信号处理器返回之后，主程序会继续执行。然后 `pause()` 调用会阻塞直到第二个 SIGINT 信号被递送。这样这段代码的目的就没有达到，我们的本意是解除 SIGINT 阻塞并等待第一个信号的到来。

虽然 SIGINT 信号在临界区开始（第一个 `sigprocmask()` 调用）和 `pause()` 调用

之间产生的概率很低，这无论如何也是上面代码中的一个 bug。依赖于时间的 bug 是竞争条件（[第 5.1 节](#)）的一种例子。通常竞争条件发生在两个进程或两个线程共享相同的资源时。不过这里的竞争条件，发生在主程序和信号处理器之间。

为了避免这个问题，我们需要一种解除信号阻塞和挂起进程的自动化机制。这正是 `sigsuspend()` 系统调用的作用。

```
#include <signal.h>

int sigsuspend(const sigset_t *mask);
```

(通常) 返回-1 并设置 errno 为 EINTR

`sigsuspend()` 系统调用使用 `mask` 指定的信号集替换进程的信号掩码，然后挂起进程执行，直到捕获到信号和信号处理器返回。一旦信号处理器返回，`sigsuspend()` 就会还原进程信号掩码为调用之前的值。

调用 `sigsuspend()` 等同于自动执行以下操作：

```
sigprocmask(SIG_SETMASK, &mask, &prevMask); /* Assign new mask */
pause();
sigprocmask(SIG_SETMASK, &prevMask, NULL); /* Restore old mask */
```

虽然还原老的信号掩码（上面代码的最后一个步骤）看上去不是很方便，但如果我们要反复地等待信号，那么这个步骤是避免竞争条件的关键之处。在这种情况下，信号必须保持阻塞，除非正在执行 `sigsuspend()` 调用。如果我们稍后需要解除信号阻塞，我们可以调用一次 `sigprocmask()`。

当 `sigsuspend()` 被信号递送中断时，它会返回-1，并设置 `errno` 为 `EINTR`。如果掩码没有指向一个合法的地址，`sigsuspend()` 会以错误 `EFAULT` 失败。

示例程序

清单 22-5 显示了 `sigsuspend()` 的使用，这个程序执行以下步骤：

- 使用 `printSigMask()` 函数（清单 20-4）打印进程信号掩码的初始值；
- 阻塞 `SIGINT` 和 `SIGQUIT`，并保存原始的进程信号掩码；
- 为 `SIGINT` 和 `SIGQUIT` 建立相同的信号处理器，这个处理器打印一条消息，如果递送的信号是 `SIGQUIT`，会设置全局变量 `gotSigquit`；

- 循环直到 `gotSigquit` 被设置，每次循环又执行以下步骤：
 - 使用 `printSigMask()` 函数打印当前的信号掩码
 - 执行 CPU 繁忙的循环几秒钟，模拟代码临界区域
 - 使用 `printPendingSigs()` 函数（清单 20-4）打印未决信号的掩码
 - 使用 `sigsuspend()` 来解除 `SIGINT` 和 `SIGQUIT` 阻塞，并等待信号到来（如果没有信号未决的话）
- 使用 `sigprocmask()` 来还原进程信号掩码为原来的状态值，然后我们使用 `printSigMask()` 来打印信号掩码。

清单 22-5：使用 `sigsuspend()`

```
----- signals/t_sigsuspend.c
#define _GNU_SOURCE /* Get strsignal() declaration from <string.h> */
#include <string.h>
#include <signal.h>
#include <time.h>
#include "signal_functions.h"      /* Declarations of printSigMask()
                                         and printPendingSigs() */
#include "tlpi_hdr.h"

static volatile sig_atomic_t gotSigquit = 0;

static void
handler(int sig)
{
    printf("Caught signal %d (%s)\n", sig, strsignal(sig));
                                /* UNSAFE (see Section 21.1.2) */
    if (sig == SIGQUIT)
        gotSigquit = 1;
}

int
main(int argc, char *argv[])
{
    int loopNum;
    time_t startTime;
    sigset_t origMask, blockMask;
    struct sigaction sa;
```

```
printSigMask(stdout, "Initial signal mask is:\n");

sigemptyset(&blockMask);
sigaddset(&blockMask, SIGINT);
sigaddset(&blockMask, SIGQUIT);
if (sigprocmask(SIG_BLOCK, &blockMask, &origMask) == -1)
    errExit("sigprocmask - SIG_BLOCK");

sigemptyset(&sa.sa_mask);
sa.sa_flags = 0;
sa.sa_handler = handler;

if (sigaction(SIGINT, &sa, NULL) == -1)
    errExit("sigaction");
if (sigaction(SIGQUIT, &sa, NULL) == -1)
    errExit("sigaction");

for (loopNum = 1; !gotSigquit; loopNum++) {
    printf("==== LOOP %d\n", loopNum);

    /* Simulate a critical section by delaying a few seconds */

    printSigMask(stdout,
                 "Starting critical section, signal mask is:\n");

    for (startTime = time(NULL); time(NULL) < startTime + 4; )
        continue; /* Run for a few seconds elapsed time */

    printPendingSigs(stdout,
                     "Before sigsuspend() - pending signals:\n");

    if (sigsuspend(&origMask) == -1 && errno != EINTR)
        errExit("sigsuspend");
}

if (sigprocmask(SIG_SETMASK, &origMask, NULL) == -1)
    errExit("sigprocmask - SIG_SETMASK");

printSigMask(stdout, "==== Exited loop\nRestored signal mask to:\n");

/* Do other processing... */
```

```
    exit(EXIT_SUCCESS);
}

----- signals/t_sigsuspend.c
```

下面 shell 会话日志显示了这个程序运行的情况：

```
$ ./t_sigsuspend
Initial signal mask is:
<empty signal set>
==== LOOP 1
Starting critical section, signal mask is:
    2 (Interrupt)
    3 (Quit)
Type Control-C; SIGINT is generated,
but remains pending because it is blocked
Before sigsuspend() - pending signals:
    2 (Interrupt)
Caught signal 2 (Interrupt) sigsuspend() is called, signals are unblocked
```

输出的最后一行在程序调用 `sigsuspend()` 时出现，会解除 `SIGINT` 的阻塞，在这个时间点，信号处理器已经被调用并打印了这行输出。

主程序继续循环：

```
==== LOOP 2
Starting critical section, signal mask is:
    2 (Interrupt)
    3 (Quit)
Type Control-\ to generate SIGQUIT
Before sigsuspend() - pending signals:
    3 (Quit)
Caught signal 3 (Quit) sigsuspend() is called, signals are unblocked
==== Exited loop Signal handler set gotSigquit
Restored signal mask to:
<empty signal set>
```

这一次我们输入 `Control-\`，会导致信号处理器设置 `gotSigquit` 标志，从而让主程序终止循环。

22.10 同步等待信号

在[第 22.9 节](#)，我们解释了使用信号处理器加上 `sigsuspend()` 来挂起进程执行，直到信号被递送给该进程。但是这样做需要编写一个信号处理器，还需要处理信号异步递送的复杂性，因此这个做法对某些应用来说是比较烦琐的。另一个方案是我们可以使用 `sigwaitinfo()` 系统调用来同步接受一个信号。

```
#define _POSIX_C_SOURCE 199309
#include <signal.h>

int sigwaitinfo(const sigset_t *set, siginfo_t *info);
```

成功时返回递送信号的编号；出错时返回-1

`sigwaitinfo()` 系统调用挂起进程执行，直到参数 `set` 指定的信号集中的信号被递送。如果在调用 `sigwaitinfo` 时，`set` 信号集中的某个信号已经是未决状态（pending），这个调用会立即返回。被递送的信号会从进程的未决信号列表中移除，然后 `sigwaitinfo()` 返回该信号的编号。如果 `info` 参数不是 `NULL`，调用返回之后它就会指向一个 `siginfo_t` 结构体，包含的内容和信号处理器的 `siginfo_t` 参数完全一样（[第 21.4 节](#)）。

使用 `sigwaitinfo()` 接收信号时，信号的递送顺序和排队规则和采用信号处理器来捕获是一样的。也就是说，标准信号不会排队；而实时信号会排队并按最小编号先递送。

除了不需要编写额外的信号处理器，使用 `sigwaitinfo()` 来等待信号还会比信号处理器+`sigsuspend()` 更加快一些（参考练习 22-3）。

通常 `sigwaitinfo()` 只用来阻塞我们感兴趣要等待的信号集。（就算信号已经被阻塞，我们也可以使用 `sigwaitinfo()` 来获取该未决信号）。但是如果信号在我们调用 `sigwaitinfo()` 之前到来，或者在两个 `sigwaitinfo()` 调用之间到来，那么信号将会按当前的信号部署进行处理。

清单 22-6 的代码演示了 `sigwaitinfo()` 的使用，这个程序首先阻塞所有信号，然后延迟命令行参数（可选）指定的秒数。这样就可以在程序调用 `sigwaitinfo()` 之前向它发送信号。然后程序会循环地调用 `sigwaitinfo()` 来接收信号，直到 `SIGINT`

或 SIGTERM 信号到来才结束。

下面 shell 会话日志演示了这个程序的使用，我们在后台运行这个程序，并且指定程序在调用 `sigwaitinfo()` 之前延迟 60 秒，然后我们向它发送两个信号：

```
$ ./t_sigwaitinfo 60 &
./t_sigwaitinfo: PID is 3837
./t_sigwaitinfo: signals blocked
./t_sigwaitinfo: about to delay 60 seconds
[1] 3837
$ ./t_sigqueue 3837 43 100 Send signal 43
./t_sigqueue: PID is 3839, UID is 1000
$ ./t_sigqueue 3837 42 200 Send signal 42
./t_sigqueue: PID is 3840, UID is 1000
```

程序完成睡眠后，会循环地调用 `sigwaitinfo()` 来接收已排队的信号。（由于程序在后台运行，所以它的输出会与下一行的 shell 提示符混合在一起）。我们可以看到信号处理器捕获实时信号时，编号小的信号会先被递送，并且信号处理器的 `siginfo_t` 结构体参数可以获得发送进程的进程 ID 和实际用户 ID：

```
$ ./t_sigwaitinfo: finished delay
got signal: 42
    si_signo=42, si_code=-1 (SI_QUEUE), si_value=200
    si_pid=3840, si_uid=1000
got signal: 43
    si_signo=43, si_code=-1 (SI_QUEUE), si_value=100
    si_pid=3839, si_uid=1000
```

然后我们使用 shell 的 `kill` 命令来向该程序发送信号，这一次我们可以看到 `si_code` 域被设置为 `SI_USER` 而不是 `SI_QUEUE`：

```
Press Enter to see next shell prompt
$ echo $$ Display PID of shell
3744
$ kill -USR1 3837 Shell sends SIGUSR1 using kill()
$ got signal: 10 Delivery of SIGUSR1
    si_signo=10, si_code=0 (SI_USER), si_value=100
    si_pid=3744, si_uid=1000 3744 is PID of shell
Press Enter to see next shell prompt
$ kill %1 Terminate program with SIGTERM
$
Press Enter to see notification of background job termination
```

```
[1]+ Done ./t_sigwaitinfo 60
```

在接收 SIGUSR1 信号的输出中，我们可以看到 si_value 域的值是 100，这其实是我们使用 sigqueue()发送上一个信号时对这个域赋的值。前面我们已经提到过，只有使用 sigqueue()来发送信号，si_value 域才会包含有效的值。

清单 22-6：使用 sigwaitinfo()来同步等待信号

```
----- signals/t_sigwaitinfo.c
#define _GNU_SOURCE
#include <string.h>
#include <signal.h>
#include <time.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int sig;
    siginfo_t si;
    sigset(SIG_BLOCK, &allSigs);

    if (argc > 1 && strcmp(argv[1], "--help") == 0)
        usageErr("%s [delay-secs]\n", argv[0]);

    printf("%s: PID is %ld\n", argv[0], (long) getpid());

    /* Block all signals (except SIGKILL and SIGSTOP) */

    sigfillset(&allSigs);
    if (sigprocmask(SIG_SETMASK, &allSigs, NULL) == -1)
        errExit("sigprocmask");
    printf("%s: signals blocked\n", argv[0]);

    if (argc > 1) { /* Delay so that signals can be sent to us */
        printf("%s: about to delay %s seconds\n", argv[0], argv[1]);
        sleep(getInt(argv[1], GN_GT_0, "delay-secs"));
        printf("%s: finished delay\n", argv[0]);
    }

    for (;;) { /* Fetch signals until SIGINT (^C) or SIGTERM */
```

```

    sig = sigwaitinfo(&allSigs, &si);
    if (sig == -1)
        errExit("sigwaitinfo");

    if (sig == SIGINT || sig == SIGTERM)
        exit(EXIT_SUCCESS);

    printf("got signal: %d (%s)\n", sig, strsignal(sig));
    printf(" si_signo=%d, si_code=%d (%s), si_value=%d\n",
           si.si_signo, si.si_code,
           (si.si_code == SI_USER) ? "SI_USER" :
           (si.si_code == SI_QUEUE) ? "SI_QUEUE" : "other",
           si.si_value.sival_int);
    printf(" si_pid=%ld, si_uid=%ld\n",
           (long) si.si_pid, (long) si.si_uid);
}
}

----- signals/t_sigwaitinfo.c

```

下面介绍的 `sigtimedwait()` 系统调用是 `sigwaitinfo()` 的一个变种，唯一的区别在于 `sigtimedwait()` 允许我们指定一个等待的时间。

```

#define _POSIX_C_SOURCE 199309
#include <signal.h>

int sigtimedwait(const sigset_t *set, siginfo_t *info,
                  const struct timespec *timeout);

```

成功时返回递送信号的编号；出错或超时返回-1(EAGAIN)

`timeout` 参数指定 `sigtimedwait()` 系统调用最大的等待时间，它是一个指向下面类型的结构体指针：

```

struct timespec {
    time_t tv_sec;      /* 秒数, time_t 是整数类型 */
    long tv_nsec;       /* 纳秒 */
};

```

通过结构体的这两个域我们可以指定 `sigtimedwait()` 最多等待的秒数和纳秒。如果两个域都指定为 0，则 `sigtimedwait()` 会立即超时——也就是立即检查指定信号集是否有未决信号。如果超时却没有信号被递送，会设置 `errno` 为 `EAGAIN`。

如果 `timeout` 参数指定为 `NULL`，则 `sigtimedwait()` 等同于 `sigwaitinfo()`。`SUSv3` 没有规定 `timeout` 为 `NULL` 的行为，少数 UNIX 实现可能会立即超时并检查信号。

22.11 通过文件描述符获取信号

从内核 2.6.22 开始，Linux 提供了（非标准的）`signalfd()` 系统调用，用来创建一个特殊的文件描述符，从而读取信号的信息。`signalfd()` 可以作为 `sigwaitinfo()` 同步接收信号的可选机制。

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

```
int signalfd(int fd, const sigset_t *mask, int flags);
```

成功时返回文件描述符；出错时返回 -1

`mask` 参数是一个信号集，指定我们想要通过 `signalfd` 机制来读取的信号。和 `sigwaitinfo()` 一样，我们应该使用 `sigprocmask()` 来阻塞所有信号，这样信号就不会被默认的信号部署捕获并处理掉，然后我们才能读取到该信号的数据。

如果 `fd` 指定为 -1，`signalfd()` 会创建一个新的文件描述符；否则 `signalfd()` 会直接修改 `fd` 参数引用的描述符，注意 `fd` 必须是之前某次调用 `signalfd()` 返回的值。

在最早的实现中，`flags` 参数保留为未来版本使用，必须指定为 0。不过从 Linux 2.6.27 开始，支持以下两个标志：

SFD_CLOEXEC

为新建的文件描述符设置 `close-on-exec` 标志 (`FD_CLOEXEC`)，用法和 `open()` 的 `O_CLOEXEC` 标志是一样的，[参考 4.3.1 节](#)。

SFD_NONBLOCK

为底层打开的文件描述符设置 `O_NONBLOCK` 标志，这样读取描述符时将不会阻塞。设置这个标志就可以不再需要额外地调用 `fcntl()` 来达到同样的效果。

创建了文件描述符之后，我们可以使用 `read()` 来读取信号。提供给 `read()` 的缓冲区的大小必须至少能够容纳一个 `signalfd_siginfo` 结构体，该结构体类型定义在头文件 `<sys/signalfd.h>` 中：

```

struct signalfd_siginfo {
    uint32_t ssi_signo; /* Signal number */
    int32_t ssi_errno; /* Error number (generally unused) */
    int32_t ssi_code; /* Signal code */
    uint32_t ssi_pid; /* Process ID of sending process */
    uint32_t ssi_uid; /* Real user ID of sender */
    int32_t ssi_fd; /* File descriptor (SIGPOLL/SIGIO) */
    uint32_t ssi_tid; /* Kernel timer ID (POSIX timers) */
    uint32_t ssi_band; /* Band event (SIGPOLL/SIGIO) */
    uint32_t ssi_tid; /* (Kernel-internal)timer ID(POSIX timers) */
    uint32_t ssi_overrun; /* Overrun count (POSIX timers) */
    uint32_t ssi_trapno; /* Trap number */
    int32_t ssi_status; /* Exit status or signal (SIGCHLD) */
    int32_t ssi_int; /* Integer sent by sigqueue() */
    uint64_t ssi_ptr; /* Pointer sent by sigqueue() */
    uint64_t ssi_utime; /* User CPU time (SIGCHLD) */
    uint64_t ssi_stime; /* System CPU time (SIGCHLD) */
    uint64_t ssi_addr; /* Address that generated signal
                        (hardware-generated signals only) */
};

signalfd_siginfo 结构体和 siginfo_t 结构体（第 21.4 节）返回相同的信息。
每次调用 read() 会返回尽可能多的未决信号的信息，交装填到提供给 read()
的缓冲区中。如果 read() 时没有未决信号，read() 会阻塞直到信号到来。我们
也可以使用 fcntl() F_SETFL 操作来设置文件描述符的 O_NONBLOCK 标志，这样读取
就是非阻塞的，如果没有未决信号，read() 会以 EAGAIN 错误失败。
当信号从文件描述符中读取出来之后，信号对于该进程来说，就已经被消耗，
并且不再是未决状态。


清单 22-7： 使用 signalfd() 来读取信号



```
----- signals/signalfd_sigval.c
#include <sys/signalfd.h>
#include <signal.h>
#include "tlpi_hdr.h"

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


```

```
sigset_t mask;
int sfd, j;
struct signalfd_siginfo fdsi;
ssize_t s;

if (argc < 2 || strcmp(argv[1], "--help") == 0)
    usageErr("%s sig-num...\n", argv[0]);

printf("%s: PID = %ld\n", argv[0], (long) getpid());

sigemptyset(&mask);
for (j = 1; j < argc; j++)
    sigaddset(&mask, atoi(argv[j]));

if (sigprocmask(SIG_BLOCK, &mask, NULL) == -1)
    errExit("sigprocmask");

sfd = signalfd(-1, &mask, 0);
if (sfd == -1)
    errExit("signalfd");

for (;;) {
    s = read(sfd, &fdsi, sizeof(struct signalfd_siginfo));
    if (s != sizeof(struct signalfd_siginfo))
        errExit("read");

    printf("%s: got signal %d", argv[0], fdsi.ssi_signo);
    if (fdsi.ssi_code == SI_QUEUE) {
        printf("; ssi_pid = %d; ", fdsi.ssi_pid);
        printf("ssi_int = %d", fdsi.ssi_int);
    }
    printf("\n");
}

----- signals/signalfd_sigval.c
```

signalfd 文件描述符可以和其它描述符一起被 select(), poll(), epoll() 机制监控（[第 63 章](#)）。这个特性提供了[第 63.5.2 节](#)讨论的 self-pipe 技巧的另一种实现机制。如果信号变为未决状态，这些机制就会指示文件描述符可以读取。

当我们不再需要 signalfd 文件描述符时，我们应该关闭它，从而释放相关的

内核资源。

上面程序演示了 `signalfd()` 的用法，这个程序根据命令行参数创建信号数值的掩码，阻塞这些信号，然后创建一个 `signalfd` 文件描述符来读取这些信号。接下来程序循环读取文件描述符，并显示 `signalfd_siginfo` 结构体返回的一些信息。在下面的 shell 会话日志中，我们在后台运行这个程序，并使用清单 22-2 的程序 (`t_sigqueue.c`) 来向它发送实时信号：

```
$ ./signalfd_sigval 44 &
./signalfd_sigval: PID = 6267
[1] 6267
$ ./t_sigqueue 6267 44 123 Send signal 44 with data 123 to PID 6267
./t_sigqueue: PID is 6269, UID is 1000
./signalfd_sigval: got signal 44; ssi_pid=6269; ssi_int=123
$ kill %1 Kill program running in background
```

22.12 使用信号进行进程间通信

从某个角度来说，我们可以认为信号也是进程间通信 (IPC) 的一种机制。但是信号作为 IPC 机制会有一些受到限制的地方。首先信号和我们后面将讨论的其它 IPC 机制相比，编程更加烦琐和困难，理由如下：

- 信号的异步行为意味着我们将面临各种问题，包括可重入 (reentrancy)、竞争条件、在信号处理器中正确地处理全局变量。（如果我们使用 `sigwaitinfo()` 或 `signalfd()` 来同步等待信号，这些问题基本上不会出现）。
- 标准信号不会排队，即使是实时信号也有排队的最大数量限制。因此为了避免丢失信息，接收信号的进程必须有一个方法，来通知发送进程自己已经准备好接收另一个信号。实现这个机制最明显的方法就是接收进程向发送进程发送信号。

另外一个问题时信号只能携带少量的信息：信号数值、实时信号还能带上一个整数或指针。低带宽使得信号相比管道等 IPC 机制要慢许多。

总而言之，信号很少会被用于 IPC。

22.13 早期信号 API (System V 和 BSD)

我们对于信号的讨论仅仅关注于 POSIX 信号 API。现在我们简短地介绍历史上 System V 和 BSD 提供的信号 API。尽管新的应用都应该使用 POSIX API，我们从其它 UNIX 系统移植老的应用时，仍然可能会遇到这些过时的信号 API。由于 Linux（以及许多其它 UNIX 系统）提供了 System V 和 BSD 兼容的 API，我们迁移程序时其实只需要在 Linux 上重新编译这些程序就好了。

System V 信号 API

前面已经提到过，System V 信号 API 的一个重要区别是，我们使用 `signal()` 来建立处理器时，会得到老旧的不可靠的信号语义。这意味着信号不会被添加到进程的信号掩码，信号处理器被调用时会重置信号为默认部署，并且系统调用被信号中断后不会自动重启。

下面我们简短地讨论 System V 的信号 API，手册页提供了详细的文档。SUSv3 规定了所有这些函数，但标注应优先使用现代的 POSIX API。SUSv4 已经标记这些函数废弃。

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

void (*sigset(int sig, void (*handler)(int)))(int);
成功时：返回 sig 之前的部署，sig 之前被阻塞的话则返回 SIG_HOLD
出错时返回 -1
```

要使用可靠信号语义来建立信号处理器，System V 提供了 `sigset()` 系统调用（函数原型类似于 `signal()`）。和 `signal()` 一样，信号处理器参数可以指定为 `SIG_IGN`, `SIG_DFL` 或信号处理器的地址。此外还可以指定为 `SIG_HOLD`，来添加信号到进程的信号掩码，同时保持信号的部署不变。

如果处理器没有指定为 `SIG_HOLD`, `sig` 信号将会从进程信号掩码中移除（也就是如果信号之前被阻塞，将会解除阻塞）。

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

int sighold(int sig);
int sigrelse(int sig);
int sigignore(int sig);                                成功时总是返回 0; 出错时返回 -1

int sigpause(int sig);                               总是返回 -1 并设置 errno 为 EINTR
```

`sighold()` 函数添加一个信号到进程的信号掩码。`sigrelse()` 函数从进程信号掩码中移除一个信号。`sigignore()` 函数设置一个信号的部署为忽略。`sigpause()` 函数类似于 `sigsuspend()`，但是在挂起进程等待信号到来之前，只从进程信号掩码中移除一个信号。

BSD 信号 API

POSIX 信号 API 大量地吸收了 4.2BSD API，所以 BSD 函数基本上类似于 POSIX 中的相应 API。

和前面介绍 System V 信号 API 一样，我们列出 BSD 信号 API 的相关函数原型，然后简短地解释每个函数。同样，手册页提供了详细的信息。

```
#define _BSD_SOURCE
#include <signal.h>

int sigvec(int sig, struct sigvec *vec, struct sigvec *ovec);      成功时返回 0; 出错时返回 -1
```

`sigvec()` 函数类似于 `sigaction()`。`vec` 和 `ovec` 参数是指向下面结构体的指针：

```
struct sigvec {
    void (*sv_handler)();
    int sv_mask;
    int sv_flags;
};
```

`sigvec` 结构体的域基本上等同于 `sigaction` 中结构体相应的域。第一个显著的区别是 `sv_mask` 域是一个整数而不是 `sigset_t` (对应于 `sa_mask`)，这意味着在 32 位体系架构中，最多只能有 31 个不同的信号。另一个区别是 `sv_flags` 域的 `SV_INTERRUPT` 标志 (对应于 `sa_flags`)。由于系统调用在 4.2BSD 中是默认自动重启的，这个标志用来指定某些很慢的系统调用应该被信号处理器中断。(POSIX API 在使用 `sigaction()` 建立信号处理器时，我们必须显式地指定 `SA_RESTART`，才能启用系统调用重启)。

```
#define _BSD_SOURCE
#include <signal.h>

int sigblock(int mask);
int sigsetmask(int mask);

int sigpause(int sigmask);

int sigmask(sig);
```

两个函数都返回之前的信号掩码

总是返回 -1 并设置 `errno` 为 `EINTR`

设置掩码的 `sig` 位并返回

`sigblock()` 函数添加一组信号到进程的信号掩码，类似于 `sigprocmask()` 的 `SIG_BLOCK` 操作。`sigsetmask()` 调用直接设置信号掩码的值，类似于 `sigprocmask()` 的 `SIG_SETMASK` 操作。

`sigpause()` 函数类似于 `sigsuspend()`。注意这个函数在 System V 和 BSD API 中被定义为不同的原型，GNU C 库默认提供 System V 的版本，除非我们在编译程序时指定 `_BSD_SOURCE` 特性测试宏。

`sigmask()` 宏启用 32 位掩码中对应于信号编号的位掩码，位掩码可以“或”起来形成一组信号，如下所示：

```
sigblock(sigmask(SIGINT) | sigmask(SIGQUIT));
```

22.14 小结

某些信号会导致进程创建 core dump 文件并终止，core dump 文件包含的信息可以用调试器来诊断进程终止时的各种状态。默认情况下 core dump 文件命名为 core，但是 Linux 提供了 /proc/sys/kernel/core_pattern 文件来控制 core dump 文件的命名规则。

信号可以按异步或同步方式产生，内核或其它进程发送的信号是异步产生的。接收进程无法预测何时异步信号会被递送（异步信号通常会在下一次接收进程从内核模式切换到用户模式时被递送）。同步信号一般是进程自己执行代码来直接产生信号，例如执行指令引起硬件异常或调用 raise()。同步信号的递送是可以精确预测的，它们会立即发生。

实时信号是 POSIX 对原有信号模型的扩展，与标准信号的区别在于排队、拥有规定的递送顺序、并且能够携带信号数据。实时信号设计用于应用程序自定义的用途。实时信号使用 sigqueue() 系统调用来发送，信号处理器会得到一个额外的参数（siginfo_t 结构体），可以获得信号携带的数据，以及发送进程的进程 ID 和实际用户 ID。

sigsuspend() 系统调用允许程序自动修改进程的信号掩码，并挂起进程直到信号到来，sigsuspend() 的原子性是非常重要的，可以避免解除信号阻塞、和挂起进程等待信号之间的竞争条件。

我们可以使用 sigwaitinfo() 和 sigtimedwait() 来同步等待信号。这样我们就不需要设计和编写信号处理器，如果我们只是想要等待信号的递送，可能并不需要信号处理器。

类似于 sigwaitinfo()，sigtimedwait()，Linux 特定的 signalfd() 系统调用可以用来同步等待信号。这个接口的独特之处在于可以通过文件描述符来读取信号。文件描述符还可以通过 select()，poll()，epoll() 来监控。

尽管信号可以看成是一种 IPC 机制，但是信号的许多限制导致信号基本上不能用于 IPC：异步特性、信号不会排队、低带宽等等。信号更常见的用途是进程间同步，以及事件通知、作业控制、定时器超时等。

尽管信号的基本概念是比较直接的，我们却花了三章对其进行讨论，因为信号包含了太多细节。信号在许多系统调用 API 中都扮演了重要的角色，我们在后面许多章节还会再次碰到它们。此外，许多信号相关的函数还与线程有关（如 `pthread_kill()`, `pthread_sigmask()` 等等），我们会在[第 33.2 节](#)详细讨论它们。

更多信息

参考[第 20.15 节](#)提到的资源列表

22.15 习题

- 22-1. [第 22.2 节](#) 提到一个停止的进程如果为 SIGCONT 信号建立了处理器，并且阻塞了该信号，稍后在接收到 SIGCONT 信号时会继续执行，然后信号处理器只有当 SIGCONT 信号被解除阻塞时才会被调用。编写一个程序来验证这个情况，记住一个进程可以通过终端挂起字符（通常是 Control-Z）来停止，我们可以使用命令行 kill -CONT 向进程发送 SIGCONT 信号（或者使用 shell 的 fg 命令）。
- 22-2. 如果进程同时有一个实时信号和一个标准信号未决，SUSv3 没有规定哪个信号先被递送。编写一个程序来显示 Linux 在这种情况下是如何处理和递送信号的。（让程序为所有信号设置信号处理器，阻塞信号一段时间，这样我们就能向进程发送各种信号，然后解除所有信号阻塞）。
- 22-3. [第 22.10 节](#) 说明了 `sigwaitinfo()` 接收信号比使用信号处理器+`sigsuspend()` 更快一些。本书源代码中提供的 `signals/sig_speed_sigsuspend.c` 程序使用了 `sigsuspend()` 在父进程和子进程之间发送和接收信号。计算这个程序在两个进程之间交换一百万个信号需要的时间（程序的命令行参数可以指定信号交换的次数）。修改这个程序使用 `sigwaitinfo()`，然后重新测定期。这两个程序花费的时间差异有多少？
- 22-4. 使用 POSIX 信号 API，实现 System V 函数 `sigset()`, `sighold()`, `sigrelse()`, `sigignore()`, `sigpause()`。

第 23 章 定时器和睡眠

23.1 间隔定时器

23.2 定时器的调度和精确度

23.3 设置阻塞操作的超时

23.4 固定间隔暂停执行（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 定时器超时

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 进程终止

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()` 之间的关联

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 等待的状态值

26.1.4 进程从信号处理器终止

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 信号处理器

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（回顾可重入）

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：线程取消

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、会话和作业控制

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 作业控制

34.7.1 在 shell 中使用作业控制

34.7.2 实现作业控制

34.7.3 处理作业控制信号

34.7.4 孤儿进程组（回顾 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() 时进程能力的转移

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 创建只有能力的环境

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 使用静态库代替共享库

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

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 消息队列的客户端-服务器编程

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 特定的特性

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 发布信号量

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 监听传入的连接: 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 Iterative 和并发服务器

60.2 Iterative 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 连接建立

61.6.5 TCP 连接终止

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 其它 I/O 模型

63.1 概述

63.1.1 Level 触发和 Edge 触发通知

63.1.2 I/O 模型结合非阻塞 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 什么时候信号通知“I/O 可用”？

63.3.2 改进信号驱动 I/O 的使用

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 所有权和权限: 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